



# 妙趣横生 的算法

## (C语言实现)

杨 峰 编著

5.5小时教学视频、86个趣味算法题、61个算法面试题，一学就会  
帮您开阔眼界，培养编程兴趣，提高编程能力，增强求职竞争力



清华大学出版社

# 妙趣横生的算法 (C语言实现)

## 本书精华内容

- 数据结构基础 (7个实例)
- 常用的查找与排序方法 (7个实例)
- 常用的算法思想 (10个实例)
- 编程基本功 (20个实例)
- 数学趣题 (39个实例)
- 数据结构问题 (14个实例)
- 数值计算问题 (7个实例)
- 综合问题 (6个实例)
- 常见的算法设计题 (31个实例)
- 常见的数据结构题 (30个实例)

## 本书教学视频内容

- 数据结构专题 (36分钟)
- 查找和排序算法专题 (75分钟)
- 常用算法专题 (63分钟)
- 重点编程实例解析专题 (89分钟)
- 重点面试题解析专题 (65分钟)

## 本书读者对象

- 对算法有兴趣的入门人员
- 有C语言基础需要提高的人员
- 准备参加C语言面试的人员
- 信息学竞赛的参赛者
- 大学生程序设计竞赛的参赛者
- 需要提高编程水平的人员
- 算法爱好者与研究者
- 一些数学问题的研究者

## 特别提示

本书配套多媒体教学视频和涉及的实例代码收录于本书配书光盘中。另外,本书适合作为相关学校的教材使用。为了方便老师授课,本书专门配备了相应的教学PPT。需要的老师请发电子邮件至bookservice2008@163.com索取,索取时请提供授课人的身份信息。

ISBN 978-7-302-21601-8



9 787302 216018 >

定价: 49.00元 (附视频光盘)





# 妙趣横生 的算法

## (C语言实现)

杨 峰 编著

清华大学出版社  
北 京

新  
学  
社  
PDG

## 内 容 简 介

本书理论与实践相结合,旨在帮助读者理解算法,并提高 C 语言编程能力,培养读者的编程兴趣,并巩固已有的 C 语言知识。全书分为 2 个部分共 10 章,内容涵盖了编程必备的基础知识(如数据结构、常用算法等),编程实例介绍,常见算法和数据结构面试题等。本书最大的特色在于实例丰富,题材新颖有趣,实用性强,理论寓于实践之中。通过本书的学习,可以使读者开阔眼界,提高编程的兴趣,提高读者的编程能力和应试能力。

本书附带 1 张光盘,内容为本书源代码和作者为本书录制的 5.5 小时多媒体教学视频。

本书可作为算法入门人员的教程,也可以作为学习过 C 语言程序设计的人士继续深造的理想读物,也可作为具有一定经验的程序设计人员巩固和提高编程水平,查阅相关算法实现和数据结构知识的参考资料,同时也为那些准备参与算法和数据结构相关的面试的读者提供一些有益的帮助。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

## 图书在版编目(CIP)数据

妙趣横生的算法(C语言实现)/杨峰编著. —北京:清华大学出版社,2010.4  
ISBN 978-7-302-21601-8

I. ①妙… II. ①杨… III. ①C 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字(2009)第 228162 号

责任编辑:夏兆彦

责任校对:徐俊伟

责任印制:何 芊

出版发行:清华大学出版社

地 址:北京清华大学学研大厦 A 座

<http://www.tup.com.cn>

邮 编:100084

社 总 机:010-62770175

邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印 刷 者:北京密云胶印厂

装 订 者:北京市密云县京文制本装订厂

经 销:全国新华书店

开 本:185×260 印 张:24.5 字 数:606 千字

(附光盘 1 张)

版 次:2010 年 4 月第 1 版

印 次:2010 年 4 月第 1 次印刷

印 数:1~5000

定 价:49.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请与清华大学出版社出版部联系调换。联系电话:(010)62770177 转 3103 产品编号:035513-01

# 前 言

程序 = 数据结构 + 算法

——著名的计算机科学家沃斯 (Niklaus Wirth)

自从著名的计算机科学家沃斯将程序设计形象地用上面的公式表示出来后, 这条“黄金定律”便成为了人们学习程序设计, 进行程序开发的准则。要想成为一名真正专业的程序设计人员, 基本的数据结构基础和常用的算法知识是必须掌握的。脱离了这两点, 编写出来的程序一定不是健壮的好程序。

然而单纯地掌握了一些数据结构基础和常用的算法知识也是远远不够的。空洞地掌握所谓的数据结构和算法等理论知识只是纸上谈兵, 这些知识必须要依托于一门程序设计语言才具有真正的生命力, 才能够转化为真实的程序代码, 才能真正地解决实际问题。

本书就是将数据结构基础和常用的算法知识与目前广泛应用、最具群众基础的 C 语言相结合而产生的。本书的写作思想是理论与实践相结合, 以实践为核心, 以实例为主要内容。

首先, 本书总结归纳了数据结构基础、常用的排序查找算法和经典的算法思想, 提纲挈领地阐述了核心的理论知识。这样可以使没有系统学习过或者不熟悉数据结构和算法等知识的读者对这部分知识有一个基本的了解, 并掌握基本的数据结构知识和常用而经典的算法思想, 以便更加深入地学习本书的其他内容。

其次, 本书列举了大量的编程实例, 这些题目都按照知识体系进行了内容上的划分。本书列举的这些编程实例都是一些比较灵活有趣的题目, 有些题目渗透了巧妙的算法思想, 有些题目则必须借助特殊的数据结构才能更加容易解答。通过这些题目的训练, 可以使读者开阔眼界, 启迪思维, 提高编程的兴趣。最重要的是能够提高读者算法设计的本领, 提高读者灵活应用各种数据结构的本领, 提高读者编写程序解决实际问题能力。

## 本书有何特点

### 1. 结构清晰, 知识全面

本书分为两部分。第 1 部分是基础知识介绍, 主要介绍数据结构的基础知识和一些常用的算法思想。这部分内容为核心的理论知识, 可以帮助读者学习和回顾数据结构和算法的知识, 使读者在理论水平上有所提高, 从而能够更加顺利地深入学习后续内容。第 2 部分主要是编程实例的介绍, 通过一些非常有趣的编程实例使读者开阔眼界, 发散思维, 提高算法设计本领, 提高灵活应用各种数据结构的本领, 提高读者编写程序解决实际问题能力。

## 2. 实例丰富，讲解到位

本书的写作思想就是以实践为核心，以编程实例为主要内容。因此本书中包含了大量的编程实例，并都附有详细的分析和解答。作者认为讲解到位是本书与同类书籍相比的一大特点。本书尽量做到深入浅出，多用简单的语句配以图示来讲解比较复杂的问题。而且尽量做到讲解透彻明白，不敷衍读者。

## 3. 题材新型，趣味性强

兴趣是最好的老师。本书在编写过程中始终贯穿这一思想。因此本书中的题目设置尽量做到既有练习意义，又富有趣味性。特别是在本书的第2部分中，列举了大量的兼顾难度和趣味性的经典题目，例如魔幻方阵、汉诺塔、魔王语言翻译、约瑟夫环、马踏棋盘、巧算24、八皇后问题等。这样使读者对所谓的难题也不再那么畏惧，而是更加愿意面对它。

## 4. 重点突出，实用性强

本书的写作意图是通过讲解大量生动有趣的实例，培养读者的编程能力、算法设计思想和对数据结构的灵活运用。归根到底就是通过程序设计解决实际问题的能力。因此本书中的所有题目都不只是给出答案而已，而是从算法思想的层面来剖析。涉及到复杂数据结构的内容，还通过图示的方法形象地加以说明。特别值得一提的是，本书的最后一章为算法设计与数据结构面试题精粹，这部分内容从实战和应试的角度出发，旨在巩固读者的知识水平和提高读者的应试能力，同时使得本书更具实用价值。

## 5. 配多媒体音视频讲解

为了方便读者理解本书中的一些重点内容，作者专门为本书录制了5.5小时媒体教学视频，收录于本书配书光盘中。另外，本书涉及的源代码也收录于本书配书光盘中。

# 本书内容及知识体系

全书包含两个部分，共10章。第1部分为基础篇；第2部分为编程实例解析。

第1部分包括第1~3章，主要介绍一些必备的基础知识，包括数据结构基础知识、常用的查找和排序算法、常用的算法思想。

第2部分包括第4~10章，主要介绍一些基于C语言的编程实例。包括编程的基本功、数学趣题、数据结构题、数值计算题、综合题、算法设计与数据结构面试题等。这些题目内涵丰富，兼顾趣味性，从不同侧面体现出对数据结构知识和算法设计思想的灵活运用，相信会对读者有一定帮助。

## 本书读者对象

本书适合对算法设计有兴趣的入门人员阅读，也可作为学习过C语言程序设计的人士继续深造的理想读物。另外，本书可以作为具有一定经验的程序设计人员巩固和提高编程水平，查阅相关算法实现和数据结构知识的参考资料，同时也为那些准备参加C语言面试

的读者提供一些帮助。

## 本书作者及编委会成员

本书主要由杨峰编写。其他参与编写和资料整理的人员有陈世琼、陈欣、陈智敏、董加强、范礼、郭秋滢、郝红英、蒋春蕾、黎华、刘建准、刘霄、刘亚军、刘仲义、柳刚、罗永峰、马奎林、马味、欧阳昉、蒲军、齐凤莲、王海涛、魏来科、伍生全、谢平、徐学英、杨艳、余月、岳富军、张健和张娜。在此一并表示感谢。

本书编委会成员有欧振旭、陈杰、陈冠军、项宇峰、张帆、陈刚、程彩红、毛红娟、聂庆亮、王志娟、武文娟、颜盟盟、姚志娟、尹继平、张昆、张薛。

编 者

新 知 识  
PDG

# 目 录

## 第 1 部分 基础篇

第 1 章 数据结构基础	2
1.1 什么是数据结构	2
1.2 顺序表	2
1.2.1 顺序表的定义	3
1.2.2 向顺序表中插入元素	4
1.2.3 从顺序表中删除元素	5
1.2.4 实例与分析	7
1.3 链表	10
1.3.1 创建一个链表	11
1.3.2 向链表中插入结点	12
1.3.3 从链表中删除结点	13
1.3.4 销毁一个链表	15
1.3.5 实例与分析	15
1.4 栈	17
1.4.1 栈的定义	18
1.4.2 创建一个栈	19
1.4.3 入栈操作	19
1.4.4 出栈操作	20
1.4.5 栈的其他操作	21
1.4.6 实例与分析	22
1.5 队列	24
1.5.1 队列的定义	24
1.5.2 创建一个队列	25
1.5.3 入队列操作	26
1.5.4 出队列操作	27
1.5.5 销毁一个队列	28
1.5.6 循环队列的概念	28
1.5.7 循环队列的实现	29
1.5.8 实例与分析	31

1.6	树结构 .....	32
1.6.1	树的概念 .....	33
1.6.2	树结构的计算机存储形式 .....	33
1.6.3	二叉树的定义 .....	34
1.6.4	二叉树的遍历 .....	35
1.6.5	创建二叉树 .....	36
1.6.6	实例与分析 .....	37
1.7	图结构 .....	38
1.7.1	图的概念 .....	39
1.7.2	图的存储形式 .....	39
1.7.3	邻接表的定义 .....	41
1.7.4	图的创建 .....	41
1.7.5	图的遍历 (1) —— 深度优先搜索 .....	43
1.7.6	图的遍历 (2) —— 广度优先搜索 .....	45
1.7.7	实例与分析 .....	47
第 2 章	常用的查找与排序方法 .....	51
2.1	顺序查找 .....	51
2.2	折半查找 .....	54
2.3	排序的概述 .....	57
2.4	直接插入排序 .....	58
2.5	选择排序 .....	60
2.6	冒泡排序 .....	63
2.7	希尔排序 .....	65
2.8	快速排序 .....	68
第 3 章	常用的算法思想 .....	72
3.1	什么是算法 .....	72
3.2	算法的分类表示及测评 .....	73
3.2.1	算法的分类 .....	73
3.2.2	算法的表示 .....	73
3.2.3	算法性能的测评 .....	75
3.3	穷举法思想 .....	75
3.3.1	基本概念 .....	75
3.3.2	寻找给定区间的素数 .....	76
3.3.3	TOM 的借书方案 .....	77
3.4	递归与分治思想 .....	78
3.4.1	基本概念 .....	78
3.4.2	计算整数的划分数 .....	79
3.4.3	递归的折半查找算法 .....	82
3.5	贪心算法思想 .....	84

3.5.1	基本概念 .....	84
3.5.2	最优装船问题 .....	85
3.6	回溯法 .....	87
3.6.1	基本概念 .....	88
3.6.2	四皇后问题求解 .....	89
3.7	数值概率算法 .....	93
3.7.1	基本概念 .....	93
3.7.2	计算定积分 .....	93

## 第 2 部分 编程实例解析

第 4 章	编程基本功 .....	96
4.1	字符类型统计器 .....	96
4.2	计算字符的 ASCII 码 .....	97
4.3	嵌套 if-else 语句的妙用 .....	98
4.4	基于 switch 语句的译码器 .....	100
4.5	判断闰年 .....	101
4.6	指针变量作参数 .....	102
4.7	矩阵的转置运算 .....	103
4.8	矩阵的乘法运算 .....	105
4.9	巧用位运算 .....	107
4.10	文件的读写 .....	108
4.11	计算文件的大小 .....	109
4.12	记录程序的运行时间 .....	110
4.13	十进制/二进制转化器 .....	111
4.14	打印特殊图案 .....	113
4.15	打印杨辉三角 .....	115
4.16	复杂级数的前 $n$ 项和 .....	117
4.17	寻找矩阵中的“鞍点” .....	118
4.18	$n$ 阶勒让德多项式求解 .....	120
4.19	递归反向输出字符串 .....	121
4.20	一年中的第几天 .....	123
第 5 章	数学趣题 (一) .....	125
5.1	舍罕王的失算 .....	125
5.2	求两个数的最大公约数和最小公倍数 .....	126
5.3	歌德巴赫猜想的近似证明 .....	127
5.4	三色球问题 .....	130
5.5	百钱买百鸡问题 .....	132



5.6	判断回文数字 .....	133
5.7	填数字游戏求解 .....	135
5.8	新郎和新娘 .....	137
5.9	爱因斯坦的阶梯问题 .....	139
5.10	寻找水仙花数 .....	141
5.11	猴子吃桃问题 .....	142
5.12	兔子产仔问题 .....	143
5.13	分解质因数 .....	144
5.14	常胜将军 .....	146
5.15	求 $\pi$ 的近似值 .....	148
5.16	魔幻方阵 .....	151
5.17	移数字游戏 .....	154
5.18	数字的全排列 .....	156
5.19	完全数 .....	158
5.20	亲密数 .....	159
5.21	数字翻译器 .....	162
5.22	递归实现数制转换 .....	164
5.23	谁在说谎 .....	167
第 6 章	数学趣题 (二) .....	169
6.1	连续整数固定和问题 .....	169
6.2	表示成两个数的平方和 .....	171
6.3	具有特殊性质的数 .....	173
6.4	验证角谷猜想 .....	174
6.5	验证四方定理 .....	176
6.6	递归法寻找最小值 .....	179
6.7	寻找同构数 .....	181
6.8	验证尼科彻斯定理 .....	183
6.9	三重回文数字 .....	185
6.10	马克思手稿中的数学题 .....	187
6.11	渔夫捕鱼问题 .....	188
6.12	寻找假币 .....	189
6.13	计算组合数 .....	193
6.14	递归法求幂 .....	194
6.15	汉诺 Hanoi 塔 .....	196
6.16	选美比赛 .....	198
第 7 章	数据结构趣题 .....	203
7.1	顺序表的就地逆置 .....	203
7.2	动态数列排序 .....	205
7.3	在原表空间进行链表的归并 .....	208

7.4	约瑟夫环 .....	213
7.5	二进制/八进制转换器 .....	217
7.6	回文字符串的判定 .....	222
7.7	括号匹配 .....	226
7.8	魔王语言翻译 .....	229
7.9	动态双向链表的应用 .....	234
7.10	判断完全二叉树 .....	239
7.11	动画模拟创建二叉树 .....	244
7.12	打印符号三角形 .....	247
7.13	递归函数的非递归求解 .....	251
7.14	任意长度整数加法 .....	254
<b>第 8 章</b>	<b>数值计算问题 .....</b>	<b>262</b>
8.1	递推化梯形法求解定积分 .....	262
8.2	求解低阶定积分 .....	265
8.3	迭代法开平方运算 .....	268
8.4	牛顿法解方程 .....	271
8.5	欧拉方法求解微分方程 .....	273
8.6	改进的欧拉方法求解微分方程 .....	275
8.7	雅可比迭代公式求解线性方程组 .....	278
<b>第 9 章</b>	<b>综合题 .....</b>	<b>282</b>
9.1	破碎的砝码 .....	282
9.2	计算 24 的问题 .....	285
9.3	马踏棋盘 .....	291
9.4	0-1 背包问题 .....	296
9.5	八皇后问题求解 .....	302
9.6	简易文件加密/解密系统 .....	306
<b>第 10 章</b>	<b>算法设计与数据结构面试题精粹 .....</b>	<b>315</b>
10.1	常见的算法设计题 .....	315
10.2	常见的数据结构题 .....	354



# 第 1 部分 基础篇

- » 第 1 章 数据结构基础
- » 第 2 章 常用的查找与排序方法
- » 第 3 章 常用的算法思想



# 第1章 数据结构基础

要想成为一名真正的程序员，数据结构是必备的基础知识。只有学过数据结构，才能真正有效规范地组织程序中的数据。而在实际编程中，有些问题必须通过特定的数据结构才能更方便地解决。因此数据结构是每一个搞计算机的人都应当必须掌握的知识。

要想全面而系统地学习数据结构的知识，这里的介绍显然是不充分的，建议寻找专门介绍数据结构的书籍学习。如果你只想掌握一般层次的知识，或是已经学过数据结构，只是为了深入学习本书后续的内容而进行回顾和复习，那么本章的介绍是足够的。

## 1.1 什么是数据结构

数据结构就是指计算机内部数据的组织形式和存储方法。数组就是一种简单而典型的线性数据结构类型。本章中将更加具体地介绍一些常用的数据结构，主要包括：线性结构、树、图。

线性结构是最常用，也是最简单的一种数据结构。所谓线性结构，就是指由  $n$  个数据元素构成的有限序列。直观地讲，它就是一张有限的一维数表。数组就是一种最为简单的线性结构表示。更加具体地说，线性结构主要包括：顺序表、链表、栈、队列等基本形式。其中顺序表和链表是从存储形式上区分的，而栈和队列是从逻辑功能上区分的。也就是说，顺序表和链表是线性数据结构的基础，队列和栈是基于顺序表和链表的，它们由顺序表或链表构成。

有时仅仅用线性结构存储管理数据是难以胜任的，一些数据之间存在着“一对多”的关系，这就构成了所谓的树形结构，简称树结构。例如人工智能领域常用的“博弈树”，数据挖掘和商业智能中使用的“决策树”，多媒体技术中的“哈夫曼树”等都是应用树结构存储管理数据的实例。

还有一种常用的数据结构叫做图状结构，简称图结构。图结构中数据元素之间存在着“多对多”的关系，因此图结构与树结构相比，其线性结构要复杂得多。在处理一些复杂的问题中，图结构往往能派上用场。例如：人工智能领域中的神经网络系统、贝叶斯网络等都是应用图结构存储管理数据的实例。

## 1.2 顺序表

在计算机内部存储一张线性表（线性结构的数表），最为方便简单的就是用一组连续地址的内存单元来存储整张线性表。这种存储结构称为顺序存储结构，这种存储结构下的

线性表就叫做顺序表。如图 1-1 所示，就是顺序表的示意。

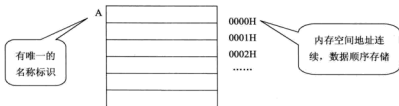


图 1-1 顺序表的示意

从上图中可以看出，一张顺序表包括以下特征。

- 有一个唯一的表名来标识该顺序表。
- 内存单元连续存储，也就是说，一张顺序表要占据一块连续的内存空间。
- 数据顺序存放，元素之间有先后关系。

因为数组满足上述特征，所以一个数组本身就是一张顺序表。

下面介绍如何定义顺序表以及对顺序表的几种操作。

### 1.2.1 顺序表的定义

定义一张顺序表也就是在内存中开辟一段连续的存储空间，并给它一个名字进行标识。只有定义了一个顺序表，才能利用该顺序表存放数据元素，也才能对该顺序表进行各种操作。

有两种定义顺序表的方法：一是静态地定义一张顺序表；二是动态地生成一张顺序表。静态地定义一张顺序表的方法与定义一个数组的方法类似。可以描述如下：

```
#define MaxSize 100
ElemType Sqlist[MaxSize];
int len;
```

为了更完整、更精确地描述一张顺序表，可以把顺序表定义成上面的代码形式。其中 `ElemType` 是指定的顺序表的类型，这里只是一个抽象的描述，要根据实际情况决定 `ElemType` 的内容。`ElemType` 可以是 `int`、`char` 等基本类型，也可以是其他构造类型（结构体类型等）。`len` 为顺序表的长度，定义这个变量目的是更加方便对顺序表进行操作。

这里要注意，上述这种定义方法，包括今后的其他数据类型的定义，都只是一种更完整、更精确的描述。读者不应刻板地机械模仿。例如这样定义：

```
int a[1000];
```

是定义一个数组，但同时也是静态定义一个顺序表。不一定非要预定义 `MaxSize` 和定义变量 `len`。因此读者学习时应灵活掌握。

动态地生成一张顺序表的方法可描述如下：

```
#define MaxSize 100
typedef struct{
    ElemType *elem;
    int length;
```

```

int listsize;
} SqList;
void initSqList(SqList *L) { /*初始化一个顺序表*/
    L->elem=(int *)malloc(MaxSize*sizeof(ElemType));
    if(!L->elem) exit(0);
    L->length=0;
    L->listsize= MaxSize;
}

```

动态生成一张顺序表可分为以下两步。

(1) 定义一个类型 SqList, 它是一个结构体, 其成员包括: 指向顺序表的首地址, 顺序表中表的长度(表中元素个数), 顺序表的存储空间容量(占据内存大小, 以 sizeof(ElemType) 为单位, 由 MaxSize 规定)。

(2) 通过调用一个函数 initSqList 实现动态生成一张顺序表。函数 initSqList 的参数是一个 SqList 类型的指针变量, 因此可以在函数 initSqList 中直接对顺序表进行操作。

函数 initSqList 的作用是动态初始化一个顺序表, 其操作步骤如下。

(1) 调用了 C 标准库函数 malloc 动态地分配一段长度为 MaxSize\*sizeof(ElemType) 的内存空间, 并将该段空间的首地址赋值给变量 L 的 elem 成员 L->elem。也就是说, L->elem 指向顺序表的首单元。

(2) 将 L->length 置为 0, 表明此时刚刚生成一张空的顺序表, 表内尚无内容, 将 L->listsize 置为 MaxSize, 表明该顺序表占据内存空间大小为 MaxSize (以 sizeof(ElemType) 为单位)。

这样 SqList 类型的变量 L 就代表了一张顺序表。其中, L->elem 指向该顺序表的表头地址, L->length 为该顺序表的长度, L->listsize 为该顺序表的容量。通过变量 L 可以灵活地操纵该顺序表, 因此 L 就代表了一张顺序表。

至此我们知道了如何静态地定义一张顺序表以及如何动态生成一张顺序表。

## 1.2.2 向顺序表中插入元素

下面介绍如何在长度为  $n$  的顺序表中的第  $i$  个位置插入新元素 item。

所谓在长度为  $n$  的顺序表中的第  $i$  个位置插入新元素, 是指在顺序表第  $i-1$  个数据元素和第  $i$  个数据元素之间插入一个新元素 item。例如顺序表为:

$$A(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$$

在第  $i$  个位置插入新元素 item 后, 该顺序表变为:

$$A(a_1, a_2, \dots, a_{i-1}, \text{item}, a_i, \dots, a_n)$$

而此时顺序表 A 的长度由  $n$  变为  $n+1$ 。

下面给出向静态顺序表中第  $i$  个位置插入元素 item 和向动态生成的顺序表中第  $i$  个位置插入元素 item 的代码描述。

按照 1.2.1 节中介绍的方法创建一个静态的顺序表 SqList[MaxSize], 那么向该静态顺序表中第  $i$  个位置插入元素 item 的代码描述如下:

```

void InsertElem(ElemType SqList[], int &n, int i, ElemType item) {
    /*向顺序表 SqList 中第 i 个位置插入元素 item, 该顺序表原长度为 n*/
    int t;
}

```

```

if (n==MaxSize||i<1||i>n+1)
    exit(0);          /*非法插入*/
for (t=n-1;t>=i-1;t--)
    Sqlist[t+1]=Sqlist[t];    /*将 i-1 以后的元素顺序后移一个元素的位置*/
Sqlist[i-1]=item;          /*在第 i 个位置上插入元素 item*/
n=n+1;                    /*表长加 1*/
}

```

函数 InsertElem()的作用是在顺序表 Sqlist 中第  $i$  个位置上插入元素 item，并将顺序表长度加 1。其实现过程如下。

(1) 判断插入元素的位置是否合法。一个长度为  $n$  的顺序表的可能插入元素的位置是  $1 \sim n+1$ ，因此如果  $i < 1$  或者  $i > n+1$ ，或者表已满、 $n == \text{MaxSize}$ （因为表的内存大小固定不变）的插入都是非法的。

(2) 将顺序表的  $i-1$  以后的元素顺序后移一个元素的位置，即：将顺序表从第  $i$  个元素到第  $n$  个元素顺序后移一个元素的位置。

(3) 在表的第  $i$  个位置（下标为  $i-1$ ）上插入元素 item，并将表长加 1。

按照 1.2.1 节中介绍的方法创建一个动态的顺序表 L，那么向该动态生成的顺序表中第  $i$  个位置插入元素 item 的代码描述如下：

```

void InsertElem(Sqlist *L, int i, ElemType item) {
    /*向顺序表 L 中第 i 个位置上插入元素 item*/
    ElemType *base, *insertPtr, *p;
    if (i<1||i>L->length+1) exit(0); /*非法插入*/
    if (L->length>=L->listsize)
    {
        base=(ElemType*)realloc(L->elem, (L->listsize+10)*sizeof(ElemType));
        /*重新追加空间*/
        L->elem=base;          /*更新内存基地址*/
        L->listsize=L->listsize+100; /*存储空间增大 100 单元*/
    }
    insertPtr=&(L->elem[i-1]); /*insertPtr 为插入位置*/
    for (p=&(L->elem[L->length-1]); p>= insertPtr; p--)
        *(p+1)=*p;          /*将 i-1 以后的元素顺序后移一个元素的位置*/
    *insertPtr=item;          /*在第 i 个位置上插入元素 item */
    L->length++;              /*表长加 1*/
}

```

上述算法与“向静态顺序表中第  $i$  个位置插入元素 item”的算法类似，都是利用将  $i-1$  以后的元素顺序后移一个元素的位置，然后再向第  $i$  个位置上插入元素 item 的方法实现。不同之处在于向静态顺序表插入元素时，由于表的内存大小固定不变，所以只能在 MaxSize 规定的范围之内顺序插入元素。而向动态生成的顺序表中第  $i$  个位置插入元素 item 时，由于顺序表是建立在动态存储区的，因此可以随时扩充。故当向表尾插入元素时，如果顺序表已满 ( $L \rightarrow \text{len} = L \rightarrow \text{listsize}$ )，可以追加一段内存空间，再并入原顺序表中。这就是动态顺序表的优势所在。

### 1.2.3 从顺序表中删除元素

下面介绍如何删除长度为  $n$  的顺序表中的第  $i$  个位置的元素。

所谓删除长度为  $n$  的顺序表中的第  $i$  个位置的元素,就是指将顺序表第  $i$  个位置上的元素去掉。例如顺序表为:

$$A(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$$

删除第  $i$  个位置的元素后,该顺序表变为:

$$A(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$$

而此时顺序表  $A$  的长度由  $n$  变为  $n-1$ 。

下面给出从静态顺序表中删除第  $i$  个位置元素和从动态生成的顺序表中删除第  $i$  个位置元素的代码描述。

按照 1.2.1 节中介绍的方法创建一个静态的顺序表 `Sqlist[MaxSize]`,那么从该静态顺序表中删除第  $i$  个位置元素的代码可描述如下:

```
void DelElem(ElemType Sqlist[],int &n,int i){
    int j;
    if(i<1||i>n)
        exit(0)                /*非法删除*/
    for(j=i;j<n;j++)
        Sqlist[j-1]=Sqlist[j];    /*将第 i 位置以后的元素依次前移*/
    n--;                          /*表长减 1*/
}
```

函数 `DelElem()` 的作用是从顺序表 `Sqlist` 中删除第  $i$  个位置的元素,并将表的长度值减 1。其实现过程如下。

(1) 判断要删除的元素是否合法。对于一个长度为  $n$  的顺序表,删除元素的合法位置是  $1 \sim n$ ,因此如果  $i < 1$  或者  $i > n$  都是不合法的。

(2) 将顺序表的第  $i$  位置以后的元素依次前移,这样会将第  $i$  个元素覆盖,也就起到删除第  $i$  个位置元素的作用。

(3) 最后将表长减 1。

如果按照 1.2.1 节中介绍的方法创建一个动态的顺序表  $L$ ,那么从该动态生成的顺序表中删除第  $i$  个位置元素的代码描述如下。

```
void DelElem(Sqlist *L,int i) {
    /*从顺序表 L 中删除第 i 个元素*/
    ElemType *delItem, *q;
    if(i<1||i>L->len) exit(0);    /*非法删除*/
    delItem=&(L->elem[i-1]);      /*delItem 指向第 i 个元素*/
    q=L->elem+L->length-1;        /*q 指向表尾*/
    for(++delItem; delItem<=q;++ delItem)*(delItem-1)=*delItem;
    /*将第 i 位置以后的元素依次前移*/
    L->length--;                  /*表长减 1*/
}
```

从动态生成的顺序表中删除第  $i$  个位置元素的方法与从静态顺序表中删除第  $i$  个位置元素的方法本质上是一样的,都是将第  $i$  个位置以后的元素依次前移,从而覆盖掉第  $i$  个元素。

以上介绍了顺序表的定义方法、顺序表的元素插入、删除等操作。顺序表是最简单的一种线性存储结构,它的优点是:构造简单,操作方便,且通过顺序表的首地址(或数组名)可直接对表进行随机存取,从而存取速度快,系统开销小。同时它也存在缺点:有



可能浪费存储空间,在插入或删除一个元素时,需要对插入或删除位置后面的所有元素逐个进行移动,从而导致操作效率较低。因此,顺序表数据结构适用于表的长度不经常发生变化的场合,例如批处理操作。

下面通过程序实例来理解顺序表的应用。

## 1.2.4 实例与分析

前面介绍了静态顺序表和动态顺序表的定义、创建、插入元素、删除元素等方法。下面通过具体的实例巩固学到的知识。

**【实例 1-1】** 创建一个静态的顺序表存放整数,大小为 10,完成以下的操作。

(1) 输入 6 个整数,打印出顺序表中的内容,并显示表中剩余的空间个数。

(2) 在顺序表中的第 3 个位置插入元素 0,打印出顺序表中的内容,并显示表中剩余的空间个数。

(3) 再试图插入表中第 11 个位置整数 0,程序提示超出范围。

(4) 删除表中第 6 个元素,打印出顺序表中的内容,并显示表中剩余的空间个数。

下面给出本题的程序清单。

程序清单 1-1

```

/*----- 1-1.c -----*/
#include "stdio.h"
#define MaxSize 10
/*静态顺序表的各种操作*/

/** 向顺序表中插入元素 */
/** 参数 Sqlist:表首地址 */
/** 参数*len:表的长度 */
/** 参数 i:插入元素的位置 */
/** 参数 x:待插入的元素值 */
void insertElem(int Sqlist[],int *len,int i,int x)
{
    int t;
    if(*len==MaxSize || i<1 || i>*len+1)
    {
        printf("This insert is illegal\n");
        return;
    }
    for(t=*len-1;t>=i-1;t--)
        Sqlist[t+1]=Sqlist[t];
    Sqlist[i-1]=x;
    *len=*len+1;
}

/** 向顺序表中删除元素 */
/** 参数 Sqlist:表首地址 */
/** 参数*len:表的长度 */
/** 参数 i:插入元素的位置 */
void DelElem(int Sqlist[],int *len,int i)
{
    int j;

```

```

    if(i<1 || i>len)
    {
        printf("This insert is illegal");
        return;
    }
    /*非法插入*/
    for(j=i;j<=len-1;j++)
        Sqlist[j-1]=Sqlist[j];
    /*将第 i 个元素之后的元素前移*/
    *len=*len-1;
    /*表长减 1*/
}

/**测试函数*/
main()
{
    /*按照题目要求进行测试*/
    int Sqlist[MaxSize];
    /*定义一个静态顺序表*/
    int len;
    int i;
    for(i=0;i<6;i++)
        scanf("%d",&Sqlist[i]);
    /*从键盘输入 6 个整数*/
    len=6;
    for(i=0;i<len;i++)
        printf("%d ",Sqlist[i]);
    /*输出顺序表中的 6 个整数*/
    printf("\nThe spare length is %d\n",MaxSize - len);
    /*显示表中的剩余空间*/
    insertElem(Sqlist,&len,3,0);
    /*在表中第 3 位置插入整数 0*/
    for(i=0;i<len;i++)
        printf("%d ",Sqlist[i]);
    /*输出顺序表中的所有元素*/
    printf("\nThe spare length is %d\n",MaxSize - len);
    /*显示表中的剩余空间*/
    insertElem(Sqlist,&len,11,0);
    /*在表中第 11 位置插入整数 0*/
    DelElem(Sqlist,&len,6);
    /*删除顺序表中的第 6 个元素*/
    for(i=0;i<len;i++)
        printf("%d ",Sqlist[i]);
    /*输出顺序表中的所有元素*/
    printf("\nThe spare length is %d\n",MaxSize - len);
    /*显示表中的剩余空间*/
}

```

在程序清单 1-1 中，定义了静态顺序表的插入元素操作 `insertElem()` 和删除元素操作 `DelElem()`。在前面已经讲过，`insertElem()` 的作用是将元素插入到指定的顺序表中第  $i$  个位置，同时顺序表的长度标记 `len` 加 1。`DelElem()` 的作用是将顺序表中的第  $i$  个位置的元素删除，同时顺序表的长度标记 `len` 减 1。

主函数作为测试函数，实现了题目的要求。通过主函数的测试可以得出结论：本程序中的 `insertElem()` 和 `DelElem()` 函数可以正确地实现静态顺序表插入元素和删除元素的功能。


本程序的运行结果如图 1-2 所示。

```

1 2 3 4 5 6
1 2 3 4 5 6
The spare length is 4
1 2 0 3 4 5 6
The spare length is 3
This insert is illegal
1 2 0 3 4 6
The spare length is 4

```

图 1-2 例 1-1 的运行结果

 注意: insertElem()和 DelElem()中所定义的第  $i$  个元素是指位于顺序表中的第  $i-1$  个元素, 因为顺序表的第一元素下标为 0, 因此第  $i$  个元素的下标为  $i-1$ 。

【实例 1-2】编写一个程序, 动态地创建一个顺序表。要求: 顺序表初始长度为 10, 向顺序表中输入 15 个整数, 并打印出来; 再删除顺序表中的第 5 个元素, 打印出删除后的结果。下面给出本题的程序清单。

程序清单 1-2

```

/----- 1-2.c -----*/
#include "stdio.h"
#include "conio.h"
#define MaxSize 10
typedef int ElemType; /*将 int 定义为 ElemType*/

typedef struct{
    int *elem;
    int length;
    int listsize;
} SqList;

/* 初始化一个顺序表 */
/* 参数 L: SqList 类型的指针 */
void InitSqList(SqList *L){
    L->elem=(int *)malloc(MaxSize*sizeof(ElemType));
    if(!L->elem) exit(0);
    L->length=0;
    L->listsize= MaxSize;
}

/* 向顺序表中插入元素 */
/* 参数 L: SqList 类型的指针 */
/* 参数 i: 插入元素的位置 */
/* 参数 item: 插入的元素 */
void InsertElem(SqList *L,int i,ElemType item){
    /*向顺序表 L 中第 i 个位置上插入元素 item*/
    ElemType *base,* insertPtr,*p;
    if(i<1||i>L->length+1) exit(0);
    if(L->length>=L->listsize)
    {
        base=(ElemType*)realloc(L->elem,(L->listsize+10)*sizeof
            (ElemType));
        L->elem=base;
        L->listsize=L->listsize+100;
    }
    insertPtr=&(L->elem[i-1]);
    for(p=&(L->elem[L->length-1]);p>= insertPtr;p--)
        *(p+1)=*p;
    * insertPtr=item;
    L->length++;
}

/* 从顺序表中删除元素 */
/* 参数 L: SqList 类型的指针 */
/* 参数 i: 删除元素的位置 */
void DelElem(SqList *L,int i){
    /*从顺序表 L 中删除第 i 个元素*/

```

```

ElemType *delItem, *q;
if(i<1||i>L->length) exit(0);
delItem=&(L->elem[i-1]);
q=L->elem+L->length-1;
for(++delItem;delItem<=q;++ delItem)*( delItem-1)=* delItem;
L->length--;
}

/** 测试函数 */
main()
{
    SqList l;
    int i;
    initSqList (&l);                /*初始化一个顺序表*/
    for(i=0;i<15;i++)
        InsertElem(&l,i+1,i+1);    /*向顺序表中插入1...15*/
    printf("\nThe content of the list is\n");
    for(i=0;i<l.length;i++)
        printf("%d ",l.elem[i]);    /*打印出顺序表中的内容*/
    DelElem(&l,5);                  /*删除第5个元素,即5*/
    printf("\nDelete the fifth element\n");
    for(i=0;i<l.length;i++)        /*打印出删除后的结果*/
        printf("%d ",l.elem[i]);
    getch();
}

```

本程序实现了一个动态顺序表,并按照题目的要求对该顺序表实施各种操作,具体操作如下。

- (1) 程序首先用函数 `initSqList()` 动态地创建了一个顺序表,起初始化长度为 `MaxSize 10`。
- (2) 然后通过函数 `InsertElem()` 动态地向顺序表中插入数据。这里通过一个循环,每次向顺序表的第  $i+1$  的位置插入整数  $i+1$ 。然后打印出该顺序表中的值。在插入顺序表的过程中,由于顺序表初始化长度为 10,而要插入 15 个元素,因此要调用 `realloc()` 函数为顺序表重新分配内存空间。
- (3) 再应用 `DelElem()` 函数删除表中第 5 个元素,也就是 5。
- (4) 最后打印出删除元素后该顺序表中的值。

本程序的运行结果如图 1-3 所示。

```

The content of the list is
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Delete the fifth element
1 2 3 4 6 7 8 9 10 11 12 13 14 15

```

图 1-3 例 1-2 的运行结果

## 1.3 链 表

与顺序表相同,链表也是一种线性表,它的数据的逻辑组织形式是一维的。而与顺序表不同的是,链表的物理存储结构是用一组地址任意的存储单元存储数据的。也就是说,

它不像顺序表那样占据一段连续的内存空间，而是将存储单元分散在内存的任意地址上。在链表结构中，存储的每个数据元素记录都存放到链表的一个结点（node）中，而每个结点之间由指针将其连接在一起，这样就形成了一条如同“链”的结构。

在 C 程序中，链表的每个结点可以是一个结构体类型元素，当然也可以是其他的构造类型元素。在链表的每个结点中，都必须有一个专门用来存放指针（地址）的域，用这个指针域来存放后继结点的地址，这样就达到了连接后继结点的目的。一条链表通常有一个“表头”，它是一个指针变量，用来存放第一个结点地址。此外，一条链表的最后一个结点的指针域要置空（NULL），因为它没有后继结点。链表的结构如图 1-4 所示。



图 1-4 链表结构示意图

从图 1-4 中可以看出链表存在以下特征。

- 每一个结点包括两部分：数据域和指针域。其中数据域用来存放数据元素本身的信息，指针域用来存放后继结点的地址。
- 链表逻辑上连续，物理上并不一定连续存储结点。
- 只要获得链表的头结点，就可以通过指针遍历整条链表。

一个链表结点可用 C 语言描述如下：

```
typedef struct node{
    ElemType data;      /*数据域*/
    struct node *next   /*指针域*/
}LNode,*LinkList;
```

这里采用自定义类型的方式将结构 struct node 定义为 LNode 类型。也就是说，该链表每个结点的类型为 LNode。另外，\*LinkList 是指向 LNode 类型数据的指针类型定义。也就是说，在定义一个指向 LNode 类型数据的指针类型变量时，语句

```
LNode *L;
```

和

```
LinkList L;
```

是等价的。

下面介绍如何建立一个链表以及如何操作链表。

### 1.3.1 创建一个链表

建立一个链表的过程可通过下面这段代码来描述。

```
LinkList GreatLinkList(int n){
    /*建立一个长度为 n 的链表*/
    LinkList p,r,list=NULL;
    ElemType e;
    int i;
```

```

for(i=1;i<=n;i++){
    Get(e);
    p=(LinkedList)malloc(sizeof(LNode));
    p->data=e;
    p->next=NULL;
    if(!list)
        list=p;
    else
        r->next=p;
    r=p;
}
return list;
}

```

上面这段代码描述了一个建立一条长度为  $n$  的链表的全过程，共分为以下几个步骤。

(1) 用 `malloc` 函数在内存的动态存储区中开辟一块大小为 `sizeof(LNode)` 的空间，并将其地址赋值给 `LinkedList` 类型变量 `p`（前面讲过，`LinkedList` 为指向 `LNode` 变量的类型，`LNode` 为前面定义的链表结点类型）。然后将数据 `e` 存入该结点的数据域 `data`，指针域存放 `NULL`。其中数据 `e` 由函数 `Get()` 获得。

(2) 如果指针变量 `list` 为空，说明本次生成的结点为第一个结点，所以将 `p` 赋值给 `list`，`list` 是 `LinkedList` 类型变量，只用来指向第一个链表结点，因此它是该链表的头指针，最后要返回。

(3) 如果指针变量 `list` 不为空，则说明本次生成的结点不是第一个结点，因此将 `p` 赋值给 `r->next`。这里 `r` 是一个 `LinkedList` 类型变量，它永远指向原先链表的最后一个结点，也就是要插入结点的前一个结点。

(4) 再将 `p` 赋值给 `r`，目的是使 `r` 再次指向最后的结点，以便生成链表的下一个结点，即保证 `r` 永远指向原先链表的最后一个结点。

(5) 最后将生成的链表的头指针 `list` 返回主调函数，通过 `list` 就可以访问到该链表的每一个结点，并对该链表进行操作。

### 1.3.2 向链表中插入结点

下面介绍如何在指针 `q` 指向的结点后面插入结点。该过程的步骤如下：

(1) 先创建一个新结点，并用指针 `p` 指向该结点。

(2) 将 `q` 指向的结点的 `next` 域的值（即 `q` 的后继结点的指针）赋值给 `p` 指向结点的 `next` 域。

(3) 将 `p` 的值赋值给 `q` 的 `next` 域。

通过以上 3 步就可以实现在链表中由指针 `q` 指向的结点后面插入 `p` 所指向的结点。可以通过图 1-5 形象地展示出这一过程。

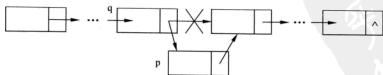


图 1-5 向链表插入结点过程

下面给出代码描述:

```
void insertList(LinkList *list, LinkList q, ElemType e) {
    /*向链表中由指针 q 指出的结点后面插入结点, 结点数据为 e*/
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));      /*生成一个新结点, 由 p 指向它*/
    p->data=e;                               /*向该结点的数据域赋值 e*/
    if(!*list){
        *list=p;
        p->next=NULL;
    }                                         /*当链表为空时*/
    else{
        p->next=q->next;
        /*将 q 指向的结点的 next 域的值赋值给 p 指向结点的 next 域*/
        q->next=p;
        /*将 p 的值赋值给 q 的 next 域*/
    }
}
```

上面的这段代码描述了如何在指针 q 指向的结点后面插入结点的过程。其过程包括以下几步。

(1) 首先生成一个新的结点, 大小为 `sizeof(LNode)`, 用 `LinkList` 类型的变量 `p` 指向该结点。将该结点的数据域赋值为 `e`。

(2) 接下来判断链表是否为空。如果链表为空, 则将 `p` 赋值给 `list`, `p` 的 `next` 域的值置为空。否则, 将 `q` 指向的结点的 `next` 域的值赋值给 `p` 指向结点的 `next` 域, 这样 `p` 指向的结点就与 `q` 指向结点的下一个结点连接到了一起。

(3) 然后再将 `p` 的值赋值给 `q` 所指结点的 `next` 域, 这样就将 `p` 指向的结点插入到了指针 `q` 指向结点的后面。

其实通过上面这段算法描述可以看出, 应用这个算法同样可以创建一个链表。这是因为当最开始时链表为空, 即 `list=NULL`, 该算法可自动为链表创建一个结点。在下面的创建其他结点的过程中, 只要始终将指针 `q` 指向链表的最后一个结点, 就可以创建出一个链表。

**注意:** 函数 `insertList()` 的参数中有一个 `LinkList *list`, 它是指向 `LinkList` 类型的指针变量, 相当于指向 `LNode` 类型的指针的指针。这是因为在函数中要对 `list`, 也就是表头指针进行修改, 而调用该函数时, 实参是 `&list`, 而不是 `list`。因此必须采取指针参数传递的办法, 否则无法在被调函数中修改主函数中定义的变量的内容。以下的代码也有类似的情况。

### 1.3.3 从链表中删除结点

下面介绍如何从非空链表中删除 `q` 所指的结点。在讨论这个问题时, 必须考虑以下 3 种情形。

- ☐ `q` 所指向的是链表的第一个结点。
- ☐ `q` 所指向的结点的前驱结点的指针已知。
- ☐ `q` 所指向的结点的前驱结点的指针未知。

下面给出不同情形的解决方法。

当  $q$  所指向的是链表的第一个结点时，只需将  $q$  所指结点的指针域  $next$  的值赋值给头指针  $list$ ，让  $list$  指向第二个结点，再释放掉  $q$  所指结点即可。该过程如图 1-6 所示。

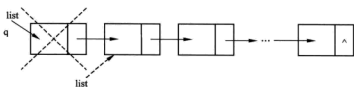


图 1-6 删除链表结点的第一种情形

当  $q$  所指向的结点的前驱结点的指针已知时（假设为  $r$ ），只需将  $q$  所指结点的指针域  $next$  的值赋值给  $r$  的指针域  $next$ ，再释放掉  $q$  所指结点即可。该过程如图 1-7 所示。

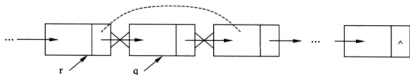


图 1-7 删除链表结点的第二种情形

以上两种情形可用下面这段代码来描述。

```
void delLink(LinkList *list, LinkList r, LinkList q) {
    if (q == *list)                /*删除链表结点的第一种情形*/
        *list = q->next;
    else
        r->next = q->next;         /*删除链表结点的第二种情形*/
    free(q);
}
```

当  $q$  所指向的结点的前驱结点的指针未知时，需要先通过链表头指针  $list$  遍历链表，找到  $q$  的前驱结点的指针，并将该指针赋值给指针变量  $r$ ，再按照第二种情形去做即可。

下面给出具体的代码描述。

```
void delLink(LinkList *list, LinkList q) {
    LinkList r;
    if (q == *list) {
        *list = q->next;
        free(q);
    }
    else {
        for (r = *list; r->next != q; r = r->next); /*遍历链表，找到 q 的前驱结点的指针*/
        if (r->next != NULL) {
            r->next = q->next;
            free(q);
        }
    }
}
```



### 1.3.4 销毁一个链表

在链表使用完毕后建议销毁它，因为链表本身会占用内存空间。如果一个系统中使用很多的链表，而使用完后又不及时地销毁它，那么这些垃圾空间积累过多，最终可能导致内存的泄漏甚至程序的崩溃。因此应当养成及时销毁不用的链表的习惯。

下面给出销毁一个链表 list 的代码描述：

```
void destroyLinkList(LinkList *list){
    LinkList p,q;
    p=*list;
    while(p){
        q=p->next;
        free(p);
        p=q;
    }
    *list=NULL;
}
```

函数 destroyLinkList()的作用是销毁一个链表 list，它包括以下步骤。

(1) 首先将 \*list 的内容赋值给 p，这样 p 也指向链表的第一个结点，成为了链表的表头。

(2) 然后判断只要 p 不为空 (NULL)，就将 p 指向的下一个结点的指针 (地址) 赋值给 q，并应用函数 free() 释放掉 p 所指向的结点，p 再指向下一个结点。如此循环，直到链表为空为止。

(3) 最后将 \*list 的内容置为 NULL，这样主函数中的链表 list 就为空了，防止了 list 变为野指针。而且链表在内存中也被完全地释放掉了。

### 1.3.5 实例与分析

**【实例 1-3】** 编写一个程序，要求：从终端输入一组整数 (大于 10 个数)，以 0 作为结束标志，将这一组整数存放在一个链表中 (结束标志 0 不包括在内)，打印出该链表中的值。然后删除该链表中的第 5 个元素，打印出删除后的结果。最后在内存中释放掉该链表。

下面给出本题的程序清单。

程序清单 1-3

```
/*----- 1-3.c -----*/
#include "stdio.h"

typedef int ElemType;

typedef struct node{
    ElemType data;           /*数据域*/
    struct node *next;      /*指针域*/
}LNode,*LinkList;

LinkList GreatLinkList(int n){ /*创建一个链表，包含 n 个结点*/
    LinkList p,r,list=NULL;
    ElemType e;
```

```

int i;
for(i=1;i<=n;i++){
    scanf("%d",&e); /*输入结点的内容*/
    p=(LinkedList)malloc(sizeof(LNode)); /*为新建的结点开辟内存空间*/
    p->data=e; /*元素赋值*/
    p->next=NULL;
    if(!list)
        list=p; /*赋值链表头指针*/
    else
        r->next=p; /*将结点连入链表*/
    r=p;
}
return list; /*返回链表头指针*/
}

void insertList(LinkedList *list,LinkedList q,ElemType e){ /*向链表中插入结点*/
    LinkedList p;
    p=(LinkedList)malloc(sizeof(LNode)); /*为新建的结点开辟内存空间*/
    p->data=e; /*元素赋值*/
    if(!*list){
        *list=p; /*赋值链表头指针*/
        p->next=NULL;
    }
    else{ /*将结点连入链表*/
        p->next=q->next;
        q->next=p;
    }
}

void delLink(LinkedList *list ,LinkedList q){ /*删除链表的某结点*/
    LinkedList r;
    if(q==list){ /*如果删除第一个结点*/
        *list=q->next;
        free(q);
    }
    else{ /*删除其他结点*/
        for(r=*list;r->next!=q;r=r->next);
        if(r->next!=NULL){
            r->next=q->next;
            free(q);
        }
    }
}

void destroyLinkedList(LinkedList *list){ /*销毁一个链表*/
    LinkedList p,q;
    p=*list;
    while(p){ /*循环释放掉每一个链表结点*/
        q=p->next;
        free(p);
        p=q;
    }
    *list=NULL;
}

main()
{

```

```

int e,i;
LinkedList l,q;
q=l=GreatLinkedList(l);          /*创建一个链表结点, q 和 l 指向该结点*/
scanf("%d",&e);
while(e)                          /*循环地输入数据, 同时插入新生成的结点*/
{
    insertList(&l,q,e);
    q=q->next;
    scanf("%d",&e);
}
q=l;
printf("The content of the linklist\n");
while(q)                          /*输出链表中的内容*/
{
    printf("%d ",q->data);
    q=q->next;
}
q=l;
printf("\nDelete the fifth element\n");
for(i=0;i<4;i++)                  /*将指针 q 指向链表第 5 个元素*/
{
    q=q->next;
}
delLink(&l,q);                    /*删除 q 所指的结点*/
q=l;
while(q)                          /*打印出删除后的结果*/
{
    printf("%d ",q->data);
    q=q->next;
}
destroyLinkedList(&l);            /*释放掉该链表*/
getche();
}

```

(1) 应用函数 GreatLinkedList() 创建一个只含有 1 个结点的链表, 并向该结点中输入数据。

(2) 然后通过函数 insertList() 向链表中插入新的结点, 在插入结点的同时输入数据, 直到输入 0 为止。数据从键盘终端输入。

(3) 然后打印出该链表中的数据。再通过循环使指针 q 指向该链表的第 5 个元素, 调用函数 delLink() 删除 q 所指的结点, 打印出删除元素后链表中的值。

(4) 最后调用 destroyLinkedList() 函数释放掉链表所占用的内存空间。

本程序的运行结果如图 1-8 所示。

```

1 2 3 4 5 6 7 8 9 0
The content of the linklist
1 2 3 4 5 6 7 8 9
Delete the fifth element
1 2 3 4 6 7 8 9

```

图 1-8 例 1-3 的运行结果

## 1.4 栈

栈是一种重要的线性结构。可以这样讲, 栈是前面讲过的线性表的一种具体形式。也就是说, 栈必须通过顺序表或者链表来实现。顺序表或者链表既可以像前面介绍的那样独立存在, 组织和操作数据, 同时它们也是一些特殊的数据结构(栈、队列……)实现的基

础，它们的概念更宽泛一些。下面通过具体的介绍来深入理解栈的概念。

### 1.4.1 栈的定义

栈(stack)是一个后进先出(last in first out, LIFO)的线性表,它要求只在表尾进行删除和插入等操作。也就是说,所谓栈其实就是一个线性表(顺序表、链表),但是它在操作上有一些特殊的要求和限制。首先,栈的元素必须先进后出,这与一般的顺序表不同。其次,栈的操作只能限定在这个顺序表的表尾进行。这就是栈的独特之处。对于栈来说,这个顺序表或者链表的表尾(也就是进行删除和插入等操作的地方)称为栈的栈顶(top),相应的表头称为栈底(bottom)。

栈形如一个子弹膛,最先压进去的子弹一定最后弹出,如图1-9所示。

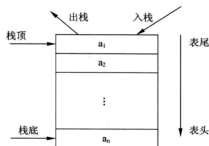


图 1-9 栈的示意

如图1-9所示,栈的本质就是一个线性表,只不过数据不能像一般的顺序表那样操作自如。数据必须从栈顶进入,还必须从栈顶取出,先进入栈中的数据在后进入栈中的数据的下面。最开始栈中不含有任何数据,叫做空栈,此时栈顶就是栈底。然后数据从栈顶进入,栈顶栈底分离,整个栈的当前容量变大。数据出栈时从栈顶弹出,栈顶下移,整个栈的当前容量变小。

上面只给出了栈及其操作的形象的描述。下面具体介绍如何定义一个栈,并在随后的小节中介绍栈的操作。

线性表有两种存储形式,即顺序表存储和链表存储。一般的栈都是用顺序表存储形式实现的,因此在这里只介绍顺序栈。

可以用以下的方式定义一个顺序栈。

```
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;
```

这里定义了一个顺序栈 sqStack 类型,它包含3个数据项: base、top、stacksize。其中,base 是指向栈底的指针变量。top 是指向栈顶的指针变量,它们指向的数据类型为 ElemType,这是压入栈中的数据的类型(可以是 int、char 等)。stacksize 指示栈的当前可使用的最大容量。掌握了这3个数据,就相当于掌握了一个完整的栈。可以通过 base 和 top

对栈进行各种操作, 通过 `stacksize` 判断栈的空间分配情况。

## 1.4.2 创建一个栈

创建一个栈有两个任务: 一是在内存中开辟一段连续的空间, 用做栈的物理存储空间; 二是将栈顶、栈底地址赋值给 `sqStack` 类型变量(对象)的 `top` 和 `base` 域, 并设置 `stacksize` 值, 以便通过这个变量(对象)对栈进行各种操作。创建一个栈的代码如下:

```
#define STACK_INIT_SIZE 100
initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间, 首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0);           /*分配空间失败*/
    s->top = s->base;                 /*最开始, 栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}
```

通过以上代码可以创建一个空栈, 其步骤如下所述。

(1) 首先通过 `malloc()` 函数开辟一段内存空间, 大小为预定义的存储空间初始分配量 `STACK_INIT_SIZE` 与每个栈元素类型 `ElemType` 大小的乘积。将创建的存储空间的首地址赋值给 `s->base`, `s` 是指向 `sqStack` 类型变量的指针。

(2) 由于最开始栈中没有任何内容, 因此是一个空栈, 所以栈顶与栈底相同, 即 `s->top = s->base`。同时这个新创建的栈的可用空间的大小为 `STACK_INIT_SIZE`, 即 `s->stacksize = STACK_INIT_SIZE`。此时, 该创建好的栈的状态如图 1-10 所示。



图 1-10 栈的初始化状态

**注意:** 要区分栈的最大容量和栈的当前容量这两个概念的不同。对于上面这个栈, 其最大容量为 100 个 `ElemType` 类型空间大小, 但是它是一个空栈, 因为它里面没有任何内容, 因此栈顶等于栈底。

## 1.4.3 入栈操作

入栈操作又叫压栈操作, 就是向栈中存放数据。入栈操作要在栈顶进行, 每向栈中压入一个数据, `top` 指针就增加 1, 直到栈满为止。可以通过以下代码实现入栈操作。

```

#define STACKINCREMENT 10
Push(sqStack *s, ElemType e){
    if(s->top - s->base >= s->stacksize){
        /*栈满，追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0); /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
    s->top++;
}

```

通过上面的代码可以向  $s$  指向的栈中压入一个  $\text{ElemType}$  类型的数据，具体步骤如下：

(1) 首先要判断栈是否已满。判断的方法是计算  $s \rightarrow \text{top}$  和  $s \rightarrow \text{base}$  的差，看它是否大于等于  $s \rightarrow \text{stacksize}$ 。因为  $s \rightarrow \text{top}$  和  $s \rightarrow \text{base}$  的差表示该栈的当前的实际容量（这里注意  $\text{top}$  指向的空间始终为栈顶元素的上一个空间）。如果该栈的当前的实际容量大于等于该栈的最大容量，则说明该栈已满，也就是说不能再存放数据了，因此需要追加一段内存空间。这里用  $\text{realloc}()$  函数进行内存的追加，内存追加空间在原有的  $\text{stacksize}$  的基础上增加  $\text{STACKINCREMENT}$  个存储单元。然后将  $s \rightarrow \text{top}$  指针指向  $s \rightarrow \text{base} + s \rightarrow \text{stacksize}$  的位置，将  $s \rightarrow \text{stacksize}$  的值再追加  $\text{STACKINCREMENT}$ 。

(2) 将待存放到栈中的数据  $e$  存放到栈顶，然后  $\text{top}$  自增 1，也就是栈顶指针向上移 1。不需要追加空间的入栈操作过程如图 1-11 所示。



图 1-11 不需要追加空间的入栈操作过程

字符  $D$  入栈后指针  $\text{top}$  增 1，整个栈的容量没有变化，注意  $\text{top}$  始终指向栈顶元素的上一个空间。

需要追加空间的入栈操作过程如图 1-12 所示。

字符  $G$  入栈之前  $\text{top}$  已经指向栈空间以外的单元，因此无法再进行入栈操作，需要追加一段内存单元，然后再进行入栈操作。

#### 1.4.4 出栈操作

出栈操作就是在栈顶取出数据，栈顶指针随之下移的操作。每当从栈内弹出一个数据，栈的当前容量就减少 1。可以重复出栈操作，直到该栈变为空栈为止。可以通过以下代码实现出栈操作。

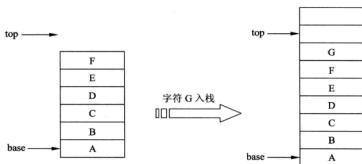


图 1-12 需要追加空间的入栈操作过程

```
Pop(sqStack *s, ElemType *e){
    if(s->top == s->base) return;
    *e = *(--(s->top));
}
```

出栈操作相对简单，首先判断栈顶指针是否与栈底指针相等，即  $s->top$  是否等于  $s->base$ 。如果相等则说明栈空，因此无法执行出栈操作。如果不等，则可以执行出栈操作。出栈操作是先将指针  $s->top$  减 1，再取出指针指向的内容，赋值给  $e$ 。

出栈操作过程如图 1-13 所示。

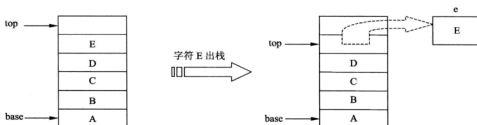


图 1-13 出栈操作过程

### 1.4.5 栈的其他操作

除了以上介绍的创建栈、入栈、出栈等操作外，对栈还有一些其他的操作。例如：清空一个栈、销毁一个栈、计算栈的当前容量等。其实程序员完全可以根据实际编程的需要来设计这些操作。

#### 1. 清空一个栈

所谓清空一个栈，就是希望将栈中的元素全部作废，而栈本身的物理空间并不一定发生改变。因此只要将  $s->top$  的内容赋值为  $s->base$  即可，这样  $s->base$  等于  $s->top$ ，也就表明这个栈是空的了。清空一个栈的代码如下：

```
ClearStack(sqStack *s){
```

```

s->top = s->base;
}

```

这样，栈顶指针  $s \rightarrow top$  与栈底指针  $s \rightarrow base$  就同时指向了栈底，表示该栈为空。

## 2. 销毁一个栈

销毁一个栈与清空一个栈不同。销毁一个栈是要释放掉该栈所占据的物理内存空间，因此不要把销毁一个栈与清空一个栈这两种操作混淆。销毁一个栈的代码如下：

```

DestroyStack(sqStack *s){
    int i,len;
    len = s->stacksize;
    for(i=0 ; i<len; i++){
        free(s->base);
        s->base++;
    }
    s->base = s->top = NULL;
    s->stacksize = 0;
}

```

因为在创建一个栈时，用的是 `malloc()` 函数在内存的动态区中开辟的一段内存空间，因此销毁一个栈要用 `free()` 函数将该段空间释放掉。然后将  $s \rightarrow base$  和  $s \rightarrow top$  置为 `NULL`，将  $s \rightarrow stacksize$  置为 0。

## 3. 计算栈的当前容量

计算栈的当前容量也就是计算栈中元素的个数，因此只要返回  $s \rightarrow top - s \rightarrow base$  即可。计算栈的当前容量代码如下：

```

int StackLen(sqStack s){
    return (s.top - s.base) ;
}

```

要注意栈的最大容量是指该栈占据内存空间的大小，其值为 `s.stacksize`，它与栈的当前容量不是一个概念。

其他的栈操作由读者自己编写。下面通过实例进一步理解栈的创建以及栈的各种操作。

## 1.4.6 实例与分析

**【实例 1-4】** 利用栈的数据结构，将二进制数转换为十进制数。

**【分析】**

二进制数是计算机中数据的存储形式。它是由一串 0/1 编码组成。每个二进制数都可以转换成为相应的十进制数，转换的方法如下：

$$(x_n x_{n-1} \cdots x_3 x_2 x_1)_2 = x_1 \cdot 2^0 + x_2 \cdot 2^1 + x_3 \cdot 2^2 + \cdots + x_n \cdot 2^{n-1}$$

一个二进制数要转换为相应的十进制数，就是从最低位起用每一位去乘以对应位的积，也就是说用第  $i$  位去乘以  $2^{i-1}$ ，然后再将每一位的乘积累加，就得到原二进制数对应的十进制表达。



由于栈具有后进先出的特性,因此可以用栈很方便地实现二进制转换为十进制。具体做法是,将一串二进制的 0/1 码从高位到低位顺序入栈,然后再逐一从栈顶取出元素,取出的第  $i$  个元素乘以  $2^{i-1}$ ,再逐一累加在一起,最终得到该二进制数的十进制表达。

程序清单 1-4

```

/*----- 1-4.c -----*/
#include "stdio.h"
#include "math.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10

typedef char ElemType;
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;

initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间,首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0); /*分配空间失败*/
    s->top = s->base; /*最开始,栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}

Push(sqStack *s, ElemType e){
    if(s->top - s->base >= s->stacksize){
        /*栈满,追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0); /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
    s->top++;
}

Pop(sqStack *s, ElemType *e){
    if(s->top == s->base) return;
    *e = *(s->top);
}

int StackLen(sqStack s){
    return (s.top - s.base);
}

main()
{
    ElemType c;
    sqStack s;
    int len, i, sum = 0;
    printf("Please input a Binary digit\n");

    initStack(&s); /*创建一个栈,用来存放二进制字符串*/
    /*输入 0/1 字符表示的二进制数,以#结束*/
}

```

```

scanf("%c",&c);
while(c!='#')
{
    Push(&s,c);
    scanf("%c",&c);
}
getchar();
len = StackLen(s);           /*得到栈中的元素个数,即二进制数的长度*/

for(i=0;i<len;i++){
    Pop(&s,&c);
    sum = sum + (c-48) * pow(2,i); /*转换为十进制*/
}
printf("Decimal is %d\n",sum);
getche();
}

```

本程序的作用是把从终端输入的一串 0/1 字符串所表示的二进制数转换为它对应的十进制数。程序的执行过程如下:

- (1) 首先程序用 `initStack()` 函数创建一个空栈, 最开始该栈的大小为 20 B。
  - (2) 然后通过一个循环语句输入二进制 0/1 码, 以 “#” 作为结束标志。在输入的过程中, 用 `Push()` 函数将该二进制字符串从高位到低位顺序压栈。
  - (3) 输入完毕, 用 `StackLen()` 函数得到该二进制数的长度。然后通过一个循环逐一从栈顶取数, 用 `Pop()` 函数实现, 取出的字符 (0 或 1) 存放在变量 `c` 中。`c-48` 是为了得到该字符对应的 0/1 值, 这是因为字符 0 的 ASCII 码为 48。函数 `pow(x,y)` 是计算  $x^y$ 。通过语句 `sum = sum + (c-48) * pow(2,i);` 达到将二进制的第  $i$  位元素乘以  $2^{i-1}$ , 再逐一累加在一起的目的。累加的结果存放在变量 `sum` 中。
  - (4) 最终 `sum` 就是输入的二进制数对应的十进制数, 将其输出到屏幕上。
- 本程序的运行结果如图 1-14 所示。



```

Please input a Binary digit
100101011
Decimal is 149

```

图 1-14 例 1-4 的运行结果

## 1.5 队 列

### 1.5.1 队列的定义

队列 (queue) 也是一种重要的线性结构。与栈相同, 实现一个队列同样需要顺序表或者链表作为基础。也就是说, 可以用链表或者顺序表来构造一个队列。但是与栈不同的是, 队列是一种先进先出 (first in first out, FIFO) 的线性表。它要求所有的数据从队列的一端进入, 从队列的另一端离开。在队列中, 允许插入数据的一端叫做队尾 (rear), 允许数据离开的一端叫做队头 (front)。如图 1-15 所示。

如图 1-15 所示, 一个队列其实就是一个线性表, 它既可以是一个顺序表, 又可以是一个链表。只是在操作上有特殊的限制: 数据只能从队尾进入队列, 只能从队头取出队列。

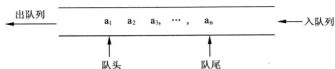


图 1-15 队列的示意图

队列既可以用链表实现，也可以用顺序表实现。这里重点介绍用链表实现一个队列，这样的队列简称为链队列。可以通过以下代码定义一个队列结构。

```
typedef struct QNode{
    QelemType data;
    struct QNode *next;
} QNode, *QueuePtr;
typedef struct{
    QueuePtr front;    //队头指针
    QueuePtr rear;     //队尾指针
}LinkQueue;
```

上面这段代码定义了一个完整的队列。首先定义一个 QNode 类，它是一个队列元素类。也就是说，队列中每个元素的类型都是 QNode。因为要定义一个链队列，因此每个队列元素不但要包括数据本身，还必须包括一个指针域，用来存放后继元素的指针（地址）。因此 QNode 是一个结构体，包含数据域 data 和指针域 next。这种定义方式与前面所讲到的定义链表的结点类十分类似，因为它们的本质是一样的。QNode 为队列元素的类型，QueuePtr 为指向 QNode 类型元素的指针类型，它等价于 QNode \*。再定义一个 LinkQueue 类型，该类型也是一个结构体类型，包含两个域。front 域用来指向队列的头，即用来存放队头元素的地址。它是一个 QueuePtr 类型的变量，也就是指向 QNode 类型元素的指针变量。QueuePtr front; 等价于 QNode \*front;。同理，rear 域用来指向队列的尾。

由于这里定义的队列是一个链队列，队列元素之间有指针相连，因此只要掌握了队列的队头指针和队尾指针，就可以对该队列进行各种操作。因此队列类型 LinkQueue 主要包含的两个域就是队头指针 front 和队尾指针 rear。

## 1.5.2 创建一个队列

创建一个队列要完成两个任务：一是在内存中创建一个头结点，但是该头结点不是用来存放数据的，而是为了操作方便人为添加的。当然也可以不定义这个头结点。二是将队列的头指针和尾指针都指向这个生成的头结点，此时队列中没有任何队列元素，该队列为空队列。不难看出应用这种方式创建的队列，队列为空的判定条件就是头指针 front 和尾指针 rear 都同时指向头结点。创建一个空队列的代码如下：

```
initQueue(LinkQueue *q){
    /*初始化一个空队列*/
    q->front = q->rear = (QueuePtr)malloc(sizeof(QNode));
    /*创建一个头结点，队头队尾指针指向该结点*/
    if(!q->front) exit(0);    /*创建头结点失败*/
    q->front->next = NULL;    /*头结点指针域置 NULL*/
}
```

通过上面这段代码可以创建一个空队列。

(1) 首先通过 `malloc()` 函数创建一个 `QNode` 类型的结点，将 `q->front` 和 `q->rear` 指针分别指向该头结点。

(2) 然后将该头结点的指针域 `next` 置为空 `NULL`。此时，这个创建好的空队列的状态如图 1-16 所示。



图 1-16 空队列的状态

### 1.5.3 入队列操作

入队列操作就是将一个 `QNode` 类型的元素从队列的尾部进入队列。每当将一个队列元素插入队列，队列的尾指针都要进行修改（因为元素从队列的尾部进入队列），队头的指针不发生改变。入队列操作的代码描述如下：

```
EnQueue(LinkQueue *q, ElemType e){
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode));           /*创建一个队列元素结点*/
    if(!q->front) exit(0);                          /*创建头结点失败*/
    p->data = e;
    p->next = NULL;
    q->rear->next = p;
    q->rear = p;
}
```

代码首先用 `malloc()` 函数创建一个 `QNode` 类型的元素结点，并用 `QueuePtr` 类型变量 `p`（也就是指向 `QNode` 类型的指针变量）指向该结点。然后将数据 `e` 赋值给该元素结点的数据域 `p->data`，将该元素结点的指针域 `next` 置空。最后通过语句：

```
q->rear->next = p;
q->rear = p;
```

将该元素结点从队列的尾部插入队列。

入队列的操作过程如图 1-17 所示。

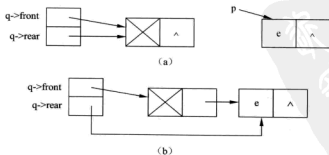


图 1-17 入队列的操作过程演示

从图 1-17 中可以看出,图(a)表示代码中前 5 句的执行结果,图(b)表示后 2 句将元素结点从队列的尾部插入队列的执行结果。

### 1.5.4 出队列操作

出队列操作是将队列中的元素从队列的头部移出。每当从队列中移出数据时,队头指针不发生改变,但是头结点的 `next` 指针发生改变。队尾指针只有在原队列中只有一个元素,即队头等于队尾的情况下才会改变,否则也不改变。出队列操作的代码描述如下:

```
DeQueue(LinkQueue *q, ElemType *e){
    /*如果队列 q 不为空,删除 q 的队头元素,用 e 返回其值*/
    if(q->front == q->rear) return; /*队列为空,返回*/
    p = q->front->next;
    *e = p->data;
    q->front->next = p->next;
    if(q->rear == p) q->rear = q->front; /*如果队头就是队尾,则修改队尾指针*/
    free(p);
}
```

程序首先判断队列是否为空,如果是空队列则返回,程序终止执行,否则继续执行。先将队头元素(`q->front->next` 指向的队列元素结点)的指针赋值给 `p`,这样 `p` 就指向了该队列的第一个元素。然后将 `p->data` 赋值给变量 `*e`,这样通过指针变量 `e` 将队列的队头元素返回。然后通过语句:

```
q->front->next = p->next;
if(q->rear == p) q->rear = q->front; /*如果队头就是队尾,则修改队尾指针*/
free(p);
```

将队列的队头元素删除。这里要注意一点,如果原队列中队头就是队尾,则说明删除队头元素后该队列为空,因此队尾指针 `q->rear` 必须修改,应当赋值为 `q->front`,此时队列变为空,队头队尾指针同时指向头结点。

原队列中有不止 1 个元素的情况下出队列的操作过程如图 1-18 所示。

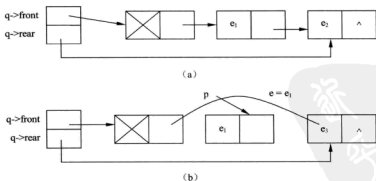


图 1-18 出队列操作的第一种情形

如图所示,图(a)是未删除队首元素前的队列的状态。图(b)是删除队首元素后的

情形, 变量  $e$  将队首元素中的数据  $e1$  返回,  $p$  指向的队首元素可以用  $\text{free}()$  函数将其释放掉。此时指针  $q \rightarrow \text{front}$  和  $q \rightarrow \text{rear}$  并没有发生改变。

原队列中只有 1 个元素的情况下出队列的操作过程如图 1-19 所示。

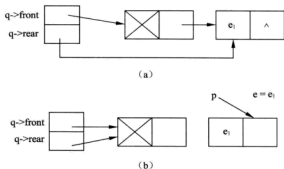


图 1-19 出队列操作的第二种情形

如图所示, 图 (a) 是未删除队首元素前的队列的状态。图 (b) 是删除队首元素后的情形, 变量  $e$  将队首元素中的数据  $e1$  返回,  $p$  指向的队首元素可以用  $\text{free}()$  函数将其释放掉。此时由于队列为空, 因此  $q \rightarrow \text{rear}$  指针必须进行修改, 将  $q \rightarrow \text{front}$  和  $q \rightarrow \text{rear}$  同时指向头结点。

### 1.5.5 销毁一个队列

由于链队列建立在内存的动态区, 因此当一个队列不再有用时应当把它及时销毁掉, 以免过多地占用内存空间。销毁一个队列的方法与销毁一个链表的方法类似, 代码如下:

```
DestroyQueue(LinkQueue *q) {
    while(q->front) {
        q->rear = q->front->next;
        free(q->front);
        q->front = q->rear;
    }
}
```

通过上面的代码可以完整地销毁一个队列, 最终  $q \rightarrow \text{rear}$  和  $q \rightarrow \text{front}$  都为空。

### 1.5.6 循环队列的概念

还有一种用顺序表实现的队列叫做循环队列。所谓循环队列顾名思义就是该队列与传统的链队列不同, 队列的空间是可以循环使用的。循环队列一般有固定的容量, 与传统的队列相同, 队列元素必须从队尾进入队列, 必须从队头出队列。如果在使用队列的过程中, 不断地有元素入队列, 同时又不断地有元素出队列, 那么对于一般的链队列, 只要队列不为空, 其队头指针  $\text{front}$  和队尾指针  $\text{rear}$  都不会发生改变, 只是头结点的  $\text{next}$  指针和队尾的前一个结点的  $\text{next}$  指针会发生变化, 而且链队列的长度也会随着入出队列元素而不断变化。而对于循环队列, 它的容量是固定的, 并且它的队头指针和队尾指针都可以随着元素入出队列而发生改变, 这样循环队列逻辑上就好像是一个环形的存储空间, 只要队列中还

有空单元未使用,就可以向队列中存放元素。由于循环队列的这一特性,循环队列可作为缓冲池存储结构来存放数据。图 1-20 为循环队列的逻辑结构。

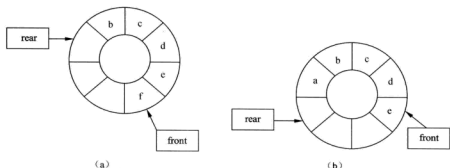


图 1-20 循环队列的逻辑结构

如图所示,图(a)为一个循环队列,该队列总容量为 8 B,实际长度为 5 B。为了方便操作,这里约定循环队列队头指针 front 始终指向队头元素,队尾指针 rear 始终指向队尾元素的下一个空间。因此这里队头元素为 f,队尾元素为 b,该队列的实际可用空间为 7 B。图(b)为将队列图(a)的队首元素 f 出队列,将字符 a 入队列后,该队列的情形。从图(b)中可以看出,对于循环队列,入队列操作就是向 rear 指向的空间赋值,然后指针 rear 再指向队尾元素的下一个空间。出队列操作就是将队头指针 front 向上移一个单元。整个循环队列逻辑上就是一个首尾相接的环形缓冲区。

### 1.5.7 循环队列的实现

在实际的内存当中,不可能有像图 1-20 那样的环形存储区,只有线性的存储单元。因此,循环队列实际上是用顺序表模拟出来的逻辑上循环,物理存储空间线性的队列数据结构。下面通过如图 1-21 所示的循环队列的几种状态来理解循环队列的实现方法和基本操作。

图(a)表示初始化一个循环队列,在内存中开辟一个 7 B 大小的连续存储单元。这就标志着该循环队列最多存放 6 B 元素,因为 rear 指针始终指向队尾元素的下一个单元。最初的循环队列队头指针 front 和队尾指针 rear 都指向 0 号单元(最低的地址)。

图(b)表示向队列中存放了两个元素 a、b。字符 a、b 入队列的过程是:先将字符 a 存放到队尾指针 rear 指向的空间,rear 指针加 1,这样字符 a 入队列;同样的过程,将字符 b 存放到队尾指针 rear 指向的空间,rear 指针再加 1,这样字符 b 入队列。队尾指针 rear 始终指向队尾元素的下一个单元。

图(c)表示图(b)所示的队列将队首元素 a 出队列后队列的状态。队首元素 a 出队列的过程为:将 front 所指的元素返回,再将指针 front 加 1。队头指针 front 始终指向队头元素。

图(d)表示在图(c)所示队列的基础上,将 c、d、e、f、g 这 5 个字符元素入队列后队列的状态。此时队尾指针 rear 重新指向 0 号单元,这是因为 rear 不断加 1 后超出了循环队列的地址范围,于是采取取模运算处理的结果。因此,无论是入队列的 rear 加 1 操作,

还是出队列的  $\text{front} + 1$  操作, 实际上是模加操作, 即  $(\text{rear} + 1) \% 7$  和  $(\text{front} + 1) \% 7$ 。正是因为这样才能在线性的内存空间上模拟出逻辑上循环的队列。

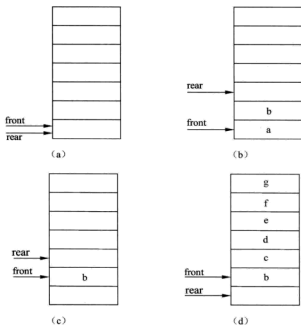


图 1-21 循环队列的几种状态

下面给出创建循环队列以及循环队列基本操作的代码描述。

### 1. 定义一个循环队列

```
#define MAXQSIZE 100          /*定义循环队列最大容量*/
typedef struct{
    ElemType *base             /*循环队列的内存分配基地址*/
    int front;                 /*队头*/
    int rear;                  /*队尾*/
}cycleQueue;                  /*定义循环队列类 cycleQueue*/
```

### 2. 初始化一个循环队列

```
initQueue(cycleQueue *q){
    q->base = (ElemType *)malloc(MAXQSIZE*sizeof(ElemType));
    /*创建 MAXQSIZE 个单元的顺序表作为循环队列的存储空间*/
    if (!q->base) exit(0);      /*内存分配失败*/
    q->front = q->rear = 0;     /*空队列, front 和 rear 都指向 0 号单元*/
}
```

### 3. 入队列操作

```
EnQueue(cycleQueue *q, ElemType e){
    if ((q->rear + 1) % MAXQSIZE == q->front) return; /*循环队列已满*/
    q->base[q->rear] = e; /*将元素 e 入队列, q->base 为顺序表的首地址*/
```



```

    q->rear = (q->rear + 1) % MAXSIZE;          /*队尾指针加1*/
}

```

#### 4. 出队列操作

```

DeQueue(cycleQueue *q, ElemType *e){
    if(q->front == q->rear) return;          /*队列为空*/
    *e = q->base[q->front];                  /*取出队头元素, 用*e 返回*/
    q->front = (q->front + 1) % MAXQSIZE;    /*队头指针加1*/
}

```

### 1.5.8 实例与分析

**【实例 1-5】** 实现一个链队列, 任意输入一串字符, 以@为结束标志, 然后将队列中的元素逐一取出, 打印在屏幕上。

#### 【分析】

这个题目很简单, 主要就是考查创建一个链队列的方法, 以及队列的基本操作。由于是链队列, 队列建立在内存的动态区上, 因此可以输入任意个字符, 以@作为结束标志。

程序清单 1-5

```

/*----- 1-5.c -----*/
#include "stdio.h"
typedef char ElemType;
typedef struct QNode{
    ElemType data;
    struct QNode *next;
} QNode, *QueuePtr;
typedef struct{
    QueuePtr front;    //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;

initQueue(LinkQueue *q){
    /*初始化一个空队列*/
    q->front = q->rear = (QueuePtr)malloc(sizeof(QNode));
    /*创建一个头结点, 队头队尾指针指向该结点*/
    if( !q->front) exit(0);    /*创建头结点失败*/
    q->front->next = NULL;    /*头结点指针域置 NULL*/
}

EnQueue(LinkQueue *q, ElemType e){
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode));    /*创建一个队列元素结点*/
    if( !q->front) exit(0);    /*创建头结点失败*/
    p->data = e;
    p->next = NULL;
    q->rear->next = p;
    q->rear = p;
}

DeQueue(LinkQueue *q, ElemType *e){
    /*如果队列 q 不为空, 删除 q 的队头元素, 用 e 返回其值*/
    QueuePtr p;
    if(q->front == q->rear) return;    /*队列为空, 返回*/
}

```

```

    p = q->front->next;
    *e = p->data;
    q->front->next = p->next;
    if(q->rear == p) q->rear = q->front; /*如果队头就是队尾，则修改队尾指针*/
    free(p);
}
/*测试函数*/
main()
{
    ElemType e;
    LinkQueue q;
    initQueue(&q); /*初始化一个队列 q*/
    printf("Please input a string into a queue\n");
    scanf("%c",&e);
    while(e!='@'){
        EnQueue(&q,e); /*向队列中输入字符串，以@表示结束*/
        scanf("%c",&e);
    }
    printf("The string into the queue is\n");
    while(q.front != q.rear){ /*将队列中的元素出队列，并打印在屏幕上*/
        DeQueue(&q,&e);
        printf("%c",e);
    }
    printf("\n");
    getch();
}

```

- (1) 在该程序中首先应用 initQueue()函数初始化一个队列。
- (2) 然后通过 EnQueue()函数将输入的字符顺序插入队列之中，直到输入字符@为止。
- (3) 然后再通过 DeQueue()函数将队列中的元素输出，并打印出来。最终队列的头指针与尾指针都指向头结点，因此队列为空。该程序的运行结果如图 1-22 所示。

```

Please input a string into a queue
This is a test: Hello world!@
The string into the queue is
This is a test: Hello world!

```

图 1-22 例 1-5 的运行结果

队列是一种很有用的线性结构，所有的大型软件系统（例如操作系统、数据库系统等）的实现几乎都离不开队列。例如操作系统中进程的管理，服务器中的进程线程管理等都需要应用队列这种数据结构的支持。因此队列的用途十分广泛，应当认真学好。

## 1.6 树 结 构

在程序设计中，树结构也是一种经常用到的数据结构。与线性结构不同，树结构采用的是非线性结构组织数据。在实际应用中，许多问题采用非线性的结构来进行描述会更加简单、方便。

直观地看，树结构是以分支关系定义的一种层次结构。也就是说，应用树结构组织起来的数据应当具有层次关系。而具有这类特性的数据在计算机中应用是十分广泛的。例如：

操作系统中的文件管理、网络系统中的域名管理、数据库系统中的索引、编译系统中的语法树等数据都是用树形结构组织的。

### 1.6.1 树的概念

用形式化的语言描述，树的定义是这样的：树是由  $n$  ( $n \geq 0$ ) 个结点组成的有穷集合。在任意的一棵非空树中：

- 有且仅有一个称为根 (Root) 的结点；
- 当  $n > 1$  时，其余的结点分为  $m$  ( $m > 0$ ) 个互不相交的有限集， $T_1, T_2, \dots, T_m$ 。其中，每一个集合本身又是一棵树，并称为根的子树 (SubTree)。

直观地讲，树的结构如图 1-23 所示。

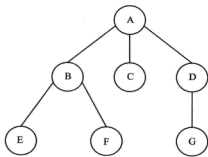


图 1-23 树的结构

图 1-23 为树结构的示意，图中该树共有 7 个结点。其中，A 为整个树结构的根结点，即为该树的根；B、C、D 结点为根结点 A 的“孩子” (Child)，以 B、C、D 为根结点又构成 3 棵树，这 3 棵树为根结点 A 的子树 (SubTree)，其中子树 C 只有一个根结点。

### 1.6.2 树结构的计算机存储形式

树结构在计算机中的存储形式很多，这里只介绍最为简单的一种树的存储形式——多重链表表示。在多重链表中，每个结点由一个数据域和若干个指针域组成，其中，每一个指针域的指针指向该结点的一个孩子结点。多重链表的结点类型可描述如下：

```

#define MaxChild 10
typedef struct node
{
    dataType data;
    struct node *child[MaxChild];
}
  
```

将树的每个结点都定义成如上所示的多重链表的结点。其中，data 为结点的数据域，存放结点的信息，child 为指向孩子结点指针数组，通过这个指针数组将父结点与子结点联系起来。可用图 1-24 形象地描述树的存储结构。

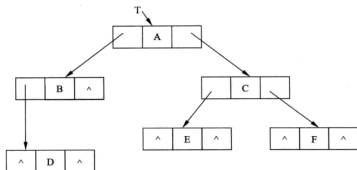


图 1-24 树的存储结构

图 1-24 中描述的是一棵二叉树(每个结点至多有两棵子树的树结构)的内部存储结构。从图中可以看出,每个结点包含一个数据域和两个指针域。数据域用来存放结点中的数据信息,例如:A、B、C…指针域用来存放指向孩子结点的指针,图中^表示空指针,表明无子结点。对于二叉树来说,一个结点至多可以有左右两个孩子,或者只有一个孩子,后者没有孩子。T 是一个指针,指向二叉树的根结点。只有得到了 T 这个指针才能访问整个树结构。

当然树结构的存储形式不止这一种。在利用多重链表表示树结构时还分为定长结点表示和不定长结点表示两种。上面的例子是最为简单的定长结点表示,即每个结点的指针域个数固定(例中都是 2 个,因为是二叉树)。还有不定长的结点表示,即不同结点的指针域个数不同。显然这种表示方法最大的优点是节省内存资源,但操作起来相对麻烦。

### 1.6.3 二叉树的定义

由于二叉树使用的范围最广,最具有代表意义,并且通过一定的方法可以将一棵多叉树转化为二叉树,因此本书重点讨论二叉树。

二叉树是一种特殊形式的树结构。前面已经提到,二叉树的特点是每个结点最多有两棵子树,下面给出二叉树更为严格的定义。

二叉树(Binary Tree)是这样的树结构:它或者为空,或者由一个根结点加上两棵分别称为左子树和右子树的互不相交的二叉树组成。显然这个定义是递归形式的。

如同前面所讲的那样,二叉树的一般存储结构采用链式存储结构。直观地讲就是将二叉树的各个结点(根结点、叶子结点等)用链表的形式连接在一起。这样通过特定的算法可以对二叉树中的每个结点进行操作。链式存储的二叉树结点的内存结构如图 1-25 所示。

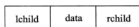


图 1-25 二叉树结点

图中二叉树结点有 3 个域,其中 lchild 和 rchild 为指针域,用来指向该结点的左孩子和右孩子。data 是数据域,用来存放该结点中包含的数据。可以用下面的代码定义二叉树

的结点。

```
typedef struct BiTNode{
    ElemType data;           /*结点的数据域*/
    struct BiTNode *lchild, *rchild; /*指向左孩子和右孩子*/
} BiTNode, *BiTree;
```

上述代码定义了一个二叉树的结点类 BiTNode，也就是说二叉树的每一个结点都属于 BiTNode 类型。另外定义了 BiTree 类型，它是一个指向 BiTNode 类型数据（对象）的指针类型，变量声明

```
BiTree t;
```

等价于

```
BiTNode *t;
```

通过 BiTree 类型的变量 t 就可以访问二叉树中的结点。一般用二叉树的根结点的指针代表一棵二叉树，它也是 BiTree 类型的。这是因为只要得到了二叉树的根结点的指针（地址），就可以通过二叉树的链结构访问到二叉树中的每一个结点。

## 1.6.4 二叉树的遍历

假设现在已经创建了一棵二叉树，在应用这个二叉树结构解决实际问题时，往往要求在二叉树中找到具有某些特征的二叉树结点，或者要求访问二叉树中的每一个结点。这就需要设一个算法，针对二叉树的数据结构特性来访问二叉树的每一个结点。这个过程被称做二叉树的遍历（Traversing binary tree）。所谓二叉树的遍历，说得直白一点就是从二叉树的一个结点（一般为根结点）出发，按照一定的规律访问到该二叉树的全部结点，每个结点只访问 1 次。这里的“访问”要依情况而定，它可能是输出结点中的数据，可能是对结点的元素进行处理等。前面讲过，二叉树的定义是一种递归形式的定义，因此可以巧妙地应用二叉树的这种递归的逻辑结构，采用递归的方法遍历二叉树。

从二叉树的定义可知，二叉树宏观上由 3 部分组成，即根结点、左子树、右子树。因此只要完整地遍历了这 3 部分，就等于遍历了整个二叉树。根结点很好访问，因为它就是一个结点。关键是如何遍历左子树和右子树。可以把左子树和右子树看成两棵独立的树，因为它们也都是由根结点、左子树、右子树 3 部分组成，因此遍历它们的方式与遍历原先那棵二叉树的方式是一样的。这样就构成了递归形式的二叉树遍历方法。根据二叉树遍历顺序的不同，对二叉树的遍历有 3 种方案：先序遍历、中序遍历、后续遍历。

### 1. 先序遍历

先序遍历二叉树的操作定义为：（1）访问根结点；（2）先序遍历左子树；（3）先序遍历右子树。其代码描述如下：

```
PreOrderTraverse(BiTree T){
    if(T){ /*递归结束条件，T为空*/
        visit(T->data); /*访问根结点*/
        PreOrderTraverse(T->lchild); /*先序遍历T的左子树*/
        PreOrderTraverse(T->rchild); /*先序遍历T的右子树*/
    }
```

```

    }
}

```

函数 `PreOrderTraverse()` 的参数是一个 `BiTree` 类型的变量，它是指向二叉树结点的指针。最开始调用该函数时，`T` 应为指向二叉树根结点的指针。`Visit()` 函数的作用要依具体情况具体分析。

## 2. 中序遍历

中序遍历二叉树的操作定义为：(1) 中序遍历左子树；(2) 访问根结点；(3) 中序遍历右子树。其代码描述如下：

```

InOrderTraverse(BiTree T){
    if(T){
        InOrderTraverse(T->lchild); /*递归结束条件，T为空*/
        visit(T->data);             /*中序遍历T的左子树*/
        InOrderTraverse(T->rchild); /*访问根结点*/
    }
}

```

## 3. 后序遍历

后序遍历二叉树的操作定义为：(1) 后序遍历左子树；(2) 后序遍历右子树；(3) 访问根结点。其代码描述如下：

```

PosOrderTraverse(BiTree T){
    if(T){
        PosOrderTraverse(T->lchild); /*递归结束条件，T为空*/
        PosOrderTraverse(T->rchild); /*后序遍历T的左子树*/
        visit(T->data);             /*后序遍历T的右子树*/
    }
}

```

## 1.6.5 创建二叉树

前面讲过了二叉树的遍历，遍历是二叉树最基本的操作。在已知一棵二叉树的根结点指针的前提下，可以通过二叉树遍历的操作访问二叉树中任何一个结点，求任何一个结点的孩子结点，求任何一个结点的双亲结点等。同样，通过二叉树的遍历操作也可以生成结点，从而通过二叉树的遍历创建一棵二叉树。

下面介绍按先序序列创建一棵二叉树。如图 1-26 所示为一棵二叉树的结构示意。

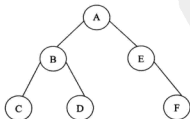


图 1-26 二叉树结构

该二叉树的先序序列是：A、B、C、D、E、F。如果要按先序序列生成这样一棵二叉树，那么生成结点的次序也应是A、B、C、D、E、F。如果要生成如图所示的这样一棵二叉树，每个结点中存放一个字符（A~F），可以通过下面这段代码创建该二叉树。

```
typedef struct BiTNode{
    char data;                                /*结点的数据域*/
    struct BiTNode *lchild , *rchild;        /*指向左孩子和右孩子*/
} BiTNode , *BiTree;
/*先序序列创建一棵二叉树*/
CreatBiTree(BiTree *T){
    char c;
    scanf("%c",&c);
    if(c == ' ') *T = NULL;
    else{
        *T = (BiTNode *)malloc(sizeof(BiTNode)); /*创建根结点*/
        (*T)->data = c;                          /*向根结点中输入数据*/
        CreatBiTree(&((*T)->lchild));             /*递归地创建左子树*/
        CreatBiTree(&((*T)->rchild));             /*递归地创建右子树*/
    }
}
```

在执行本段程序时，要输入每个结点的内容。则应当按照下列顺序读入字符：

A B C @ @ D @ @ E @ F @ @

其中@为空格符。空格是上述代码递归运算结束的标志。当创建到叶子结点时，因为叶子结点的左右子树都为空，因此要输入空格表示结束。在创建二叉树的过程中，程序总是按照：创建根结点——创建左子树——创建右子树的顺序进行。

## 1.6.6 实例与分析

**【实例 1-6】** 用先序序列创建一棵如图 1-26 所示的二叉树，并输出字符 D 位于二叉树的层数。

**【分析】**

解决这个问题首先可以用 1.6.5 节中介绍的方法创建一棵二叉树，然后通过二叉树的遍历访问每一个结点，找到包含字符 D 的结点，并输出它位于二叉树的层数。可以参考下面的代码。

程序清单 1-6

```
/*----- 1-6.c -----*/
#include "stdio.h"

typedef struct BiTNode{
    char data;                                /*结点的数据域*/
    struct BiTNode *lchild , *rchild;        /*指向左孩子和右孩子*/
} BiTNode , *BiTree;
/*创建一棵二叉树*/
CreatBiTree(BiTree *T){
    char c;
    scanf("%c",&c);
    if(c == ' ') *T = NULL;
    else{
```

```

    *T = (BiTNode *)malloc(sizeof(BiTNode));    /*创建根结点*/
    (*T)->data = c;                            /*向根结点中输入数据*/
    CreatBiTree(&((*T)->lchild));              /*递归地创建左子树*/
    CreatBiTree(&((*T)->rchild));              /*递归地创建右子树*/
}
/*访问二叉树结点，输出包含D字符结点位于二叉树中的层数*/
visit(char c,int level){
    if(c == 'D')
        printf("%c is at %d lever of BiTree\n",c,level);
}
/*遍历二叉树*/
PreOrderTraverse(BiTree T,int level){
    if(T){
        visit(T->data,level);                /*递归结束条件，T为空*/
        PreOrderTraverse(T->lchild,level+1); /*访问根结点*/
        PreOrderTraverse(T->rchild,level+1); /*先序遍历T的左子树*/
        PreOrderTraverse(T->rchild,level+1); /*先序遍历T的右子树*/
    }
}
main()
{
    int level = 1;
    BiTree T = NULL;    /*最开始T指向空*/
    CreatBiTree(&T);    /*创建二叉树*/
    PreOrderTraverse(T,level);/*遍历二叉树，找到包含D字符结点位于二叉树中的层数*/
    getch();
}

```

在源程序中将先序遍历的算法加以改造。函数 `PreOrderTraverse()` 包含两个参数，一个参数是指向二叉树根结点的指针，另一个参数是一个整型变量 `level`，它的作用是用来记录当前递归操作的层数，也就是记录当前遍历到了二叉树结点的层数。`level` 作为一个局部变量每次作为参数传递时都加 1，因此递归深入一层，变量 `level` 的值就自动增 1。而一旦递归从下一层返回到上一层时，上一层的 `level` 值保持原来的值不变，因为在这里变量 `level` 的传递仅仅是单值传递，并不是指针的传递，它在每层中的局部值都不会受到下一层任何操作的影响。函数 `visit()` 负责监视每个二叉树结点中的数据，如果访问到数据 `D`，就将该结点所在二叉树中的层数输出。该程序的运行结果如图 1-27 所示。

```

DRC D E F E F
D is at 3 lever of BiTree

```

图 1-27 例 1-6 的运行结果

树结构是一种非常重要的数据结构，特别是解决一些比较复杂的问题，很多地方都要应用到树结构作为数据的存储方式，例如多媒体技术中的哈夫曼树、人工智能领域的决策树等。特别是二叉树的结构更为重要，这不但因为二叉树的应用在树结构中是最为普遍的，而且其他类型的树结构也可以通过特定的算法与二叉树结构相互转化。因此掌握二叉树就显得更加重要。

## 1.7 图 结 构

图是一种更为复杂的数据结构。在实际的程序设计中，数据元素之间存在着 3 种关系：一种是“先行后续”的关系，一个数据元素有一个直接前驱和一个直接后继，这种数据的



组织结构叫做线性结构；一种是明显的层次关系，每一层上的数据元素可能和下一层中的多个数据元素（孩子）相关，但只和上一层中的一个数据元素（双亲）相关，这种数据的组织结构叫做树结构；还有一种是数据元素之间存在“一对多”或者“多对一”的关系，也就是任意的两个数据元素之间都可以存在着关系，这种数据的组织结构叫做图结构。

### 1.7.1 图的概念

图（graph）是由顶点的非空有限集合  $V$ （由  $N>0$  个顶点组成）与边的集合  $E$ （顶点之间的关系）所构成的。若图  $G$  中每一条边都没有方向，则称  $G$  为无向图；若图  $G$  中每一条边都有方向，则称  $G$  为有向图。

直观地讲，无向图如图 1-28 所示。

如图 1-28 所示，该无向图由 4 个顶点和 4 条边构成。每个顶点之间都有一条边相连接，而且这些边都没有方向。

有向图如图 1-29 所示。

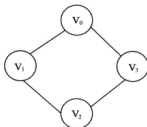


图 1-28 无向图

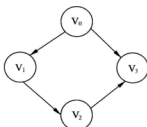


图 1-29 有向图

如图 1-29 所示，该有向图由 4 个顶点和 4 条边构成。每个顶点之间都有一条边相连接，而且这些边都存在着方向。

### 1.7.2 图的存储形式

最为常见的图的存储方法有两种：邻接矩阵存储方法和邻接表存储方法。

邻接矩阵存储方法也称数组存储方法，其核心思想是：利用两个数组来存储一个图。这两个数组一个是—维数组，用来存放图中的数据，一个是二维数组，用来表示图中顶点之间的相互关系，称为邻接矩阵。具体地，一个具有  $n$  个顶点的图  $G$ ，定义一个数组  $vertex[0,1,\dots,n-1]$ ，将该图中顶点的数据信息分别存放在该数组中对应的数组元素上。例如，顶点  $v_0$  的数据信息存放在  $vertex[0]$  中，……，顶点  $v_i$  的数据信息存放在  $vertex[i]$  中。当然数组  $vertex$  的类型要与图中顶点元素的类型一致。再定义一个二维数组  $A[0\cdots n-1][0\cdots n-1]$ ， $A$  称为邻接矩阵，它存放顶点之间的关系信息。 $A[i][j]$  定义为：

$$A[i][j] = \begin{cases} 1 & \text{当顶点 } i \text{ 与顶点 } j \text{ 之间有边} \\ 0 & \text{当顶点 } i \text{ 与顶点 } j \text{ 之间无边} \end{cases}$$

例如图 1-29 所示的有向图的邻接矩阵可表示为：

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

这样通过一个简单的邻接矩阵就可以把一个复杂的图关系表现出来。根据邻接矩阵的信息，可以操纵数组 **vertex** 的元素，从而对整个图进行操纵。

另一种图的存储形式是利用邻接表对图进行存储。邻接矩阵存储方法不适于存储稀疏图（边的数目很少的图），可以用邻接表存储这类图。

图的邻接表存储方法是一种顺序分配与链式分配相结合的存储方法。它由链表和顺序数组组成。链表用来存放边的信息，数组用来存放顶点的数据信息。具体地，对于图中的每一个顶点分别建立一个链表，如果一个图具有  $n$  个顶点，其邻接表就含有  $n$  个线性链表。每个链表前面设置一个头结点，称为顶点结点，顶点结点的结构如图 1-30 所示。



图 1-30 顶点结点的结构

顶点域 **vertex** 用来存放顶点的数据信息，指针域 **next** 指向依附于顶点 **vertex** 的第一条边。通常将一个图的  $n$  个顶点结点放到一个统一的数组中进行管理，并用该数组的下标表示顶点在图中的位置。

而在每一个链表中，链表的每一个结点称之为边结点，它表示依附于对应的顶点结点的一条边。边结点的结构如图 1-31 所示。



图 1-31 边结点的结构

如图 1-31 所示，**adjvex** 域存放该边的另一端顶点在顶点数组中的位置（数组下标）；**weight** 存放边的权重，对于无权重的图，此项省略；**next** 是指针域，它将第  $i$  个链表中的所有边结点连接成一个链表，最后一个边结点的 **next** 域为 **NULL**。

例如图 1-29 所示的有向图的邻接表可表示为图 1-32。

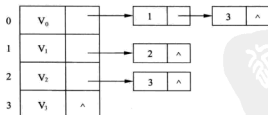


图 1-32 图 1-29 的邻接表存储形式

除了上面介绍的使用邻接矩阵和邻接表作为图的存储形式之外，还有其他形式的存储方法，例如十字链表存储法、多重邻接表存储法等，在这里就不一一介绍了。有兴趣的读者可参看专业的《数据结构》等书籍。本书重点介绍应用邻接表作为图的存储结构。

### 1.7.3 邻接表的定义

前面已经介绍了邻接表的结构特点。从前面的介绍中可以知道，一个邻接表是由一个顺序表和一组链表构成的。顺序表中存放图的顶点信息，链表中存放图的边的信息。第  $i$  个单链表中的结点表示依附于顶点  $v_i$  的边，对于有向图来说，是以顶点  $v_i$  为尾的边。如图 1-32 所示，就是图 1-29 表示的有向图的邻接表存储形式。其中顶点  $v_0$  有两条边，其出度（射出的弧的条数）为 2，一条边指向顶点  $v_1$ ，另一条边指向顶点  $v_3$ 。而顶点  $v_3$  也有两条边，但其出度为 0，因此指向的单链表为空。其余的顶点（ $v_1, v_2$ ）所指向的单链表也是这样的构成规律。对于无向图来说，每个顶点指向的单链表就表示依附于该顶点边，不考虑边的指向。图 1-33 所示为无向图 1-28 的邻接表表示。

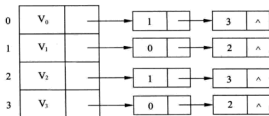


图 1-33 图 1-28 的邻接表存储形式

根据上述对邻接表结构的介绍，可以通过如下的代码定义一个邻接表。

```
#define MAX_VERTEX_NUM 20
typedef struct ArcNode{
    /*单链表中的结点的类型*/
    int adjvex;                /*该边指向的顶点在顺序表中的位置*/
    struct ArcNode *next;      /*下一条边*/
    infoType * weight;         /*边上的权重，可省略*/
}ArcNode;

typedef struct VNode{
    /*顶点类型*/
    VertexType data;           /*顶点中的数据信息*/
    ArcNode *firstarc;          /*指向单链表，即指向第一条边*/
}VNode;

VNode G[MAX_VERTEX_NUM];      /* VNode 类型的数组 G，它是图的存储容器*/
```

### 1.7.4 图的创建

清楚了图的邻接表存储结构，以及如何用代码定义一个邻接表结构，下面就可以应用这些知识创建一个基于邻接表存储结构的图结构。

在创建一个图之前，必须先设计好图的逻辑结构。因为图的创建过程比较复杂，因此建议程序员先在纸上画出要创建的图的逻辑结构，再使用邻接表在计算机中创建出图本身。

例如要应用邻接表创建一个如图 1-34 所示的有向图结构。

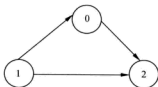


图 1-34 要创建的有向图

如图所示，该有向图包含 3 个顶点，内容分别为 0、1、2。不妨设顶点  $v_0=0$ ； $v_1=1$ ； $v_2=2$ 。有了图的逻辑结构，就不难得到该图应用邻接表存储的具体形式。图 1-34 所对应的邻接表存储形式如图 1-35 所示。

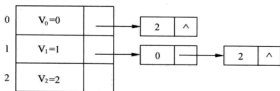


图 1-35 图 1-34 所对应的邻接表存储形式

有了上面这些准备就可以比较轻松地创建出图的邻接表存储结构了。创建一个邻接表的过程分为两个步骤。

- (1) 创建图的顶点，也就是创建“存储图中顶点的”顺序表。
- (2) 创建顶点之间的边，也就是创建单链表。可以通过下面这段代码创建一个图的邻接表结构。

下面给出创建一个邻接表存储结构的图的代码描述。

```

CreatGraph(int n , VNode G[] ){
    int i,e;
    ArcNode *p , *q;
    printf("Input the information of the vertex\n");
    for(i=0;i<n;i++){
        Getdata(G[i]);                                /*得到每个顶点中的数据*/
        G[i].firstarc = NULL;                          /*初始化第一条边为空*/
    }
    for(i=0;i<n;i++){
        printf("Creat the edges for the %dth vertex\n",i) ;
        scanf("%d",&e);
        while(e!=-1){
            p = (ArcNode *)malloc(sizeof(ArcNode));    /*创建一条边*/
            p->next = NULL;
            p->adjvex = e;
            if(G[i].firstarc == NULL) G[i].firstarc = p; /*i 结点的第一条边*/
            else q->next = p;                             /*下一条边*/
            q = p;
            scanf("%d",&e);
        }
    }
}
    
```

因为整个图的顶点都存放在邻接表的顺序表中,因此可以用 VNode 类型的一个数组来表示一个图。该数组中的每个元素都是 VNode 类型,包括顶点的数据域和指向第一条边的指针域。函数 CreatGraph()包含两个参数:  $n$  表示要创建的图的顶点的个数,  $G[]$  为一个数组指针引用,通过该数组指针修改 VNode 类型数组  $G$  中的内容,从而创建邻接表。

(1) 首先向顶点中赋值。通过函数 Getdata()得到每个顶点中的数据,并将它赋值到  $G[i].data$  中去。每个顶点的 firstarc 域要初始化为空 (NULL)。

(2) 然后创建顶点之间的边。在创建顶点之间的边时,应当严格按照最开始所设计的图的逻辑结构进行边的输入。创建边的过程与创建链表的过程类似。

如果要创建如图 1-34 所示的邻接表结构,应当通过下面的代码实现。

```
main()
{
    VNode G[3];
    CreatGraph(3,G);
}
```

首先定义 1 个包含 3 个顶点的图 (VNode 类型的数组  $G[3]$ ),再调用 CreatGraph()函数创建该图的邻接表。为了方便起见,在函数 CreatGraph()中获得顶点数据函数 Getdata()简单地定义为 scanf,即直接从终端接收一个字符。每个顶点中的数据都为整数,分别为  $v_0=0$ ;  $v_1=1$ ;  $v_2=2$ 。程序运行界面如图 1-36 所示。

```
Input the information of the vertex
0 1 2
Creat the edges for the 0th vertex
2 -1
Creat the edges for the 1th vertex
0 2 -1
Creat the edges for the 2th vertex
-1
```

图 1-36 创建邻接表

如图 1-36 所示,最开始程序提示输入顶点的数据,在此输入整数 0, 1, 2。这样顶点  $v_0=0$  存放在  $G[0]$  中; 顶点  $v_1=1$  存放在  $G[1]$  中; 顶点  $v_2=2$  存放在  $G[2]$  中。然后程序提示创建顶点  $v_0$  的边。由于该有向图中顶点  $v_0$  只有一条边射向  $v_2$ , 而顶点  $v_2$  实际存放在顺序表的  $G[2]$  中,因此在这里输入 2,表示顶点  $G[0]$  和顶点  $G[1]$  之间存在一条边,并且以顶点  $G[2]$  为尾,以顶点  $G[0]$  为头。再输入 -1 表示结束。以下输入类似。

通过上面的一系列操作,就可以将图 1-34 所示的有向图用邻接表的形式进行存储,该图对应的邻接表存储形式如图 1-35 所示。VNode 类型的数组  $G$  是该图的唯一存储容器。

## 1.7.5 图的遍历 (1) ——深度优先搜索

创建了一个图之后就要利用它解决实际问题。同树的遍历一样,图的遍历也是对图的一种最基本的操作。所谓图的遍历就是从图中的某一顶点出发,访遍图中其余顶点,且使每一个顶点只被访问一次。图的遍历操作是求解图的连通性问题,进行拓补排序,求解最短路径,求解关键路径等运算的基础。

目前普遍应用的图的遍历方法有两种。一种是本节要介绍的深度优先搜索遍历,一种是下一节介绍的广度优先搜索遍历。

深度优先搜索的基本思想是：从图中的某个顶点  $v$  出发，访问该顶点  $v$ ，然后再依次从  $v$  的未被访问过的邻接点出发，继续深度优先遍历该图，直到图中与顶点  $v$  路径相通的所有顶点都被访问到为止。由于一个图结构未必是连通的，因此一次的深度优先搜索不一定可以遍历到图中所有的顶点，若此时仍然有图中的顶点未被访问，就另选图中的一个没有被访问到的顶点作为起始点，继续深度优先搜索。重复上述的操作，直到图中的所有顶点都被访问到为止。显然深度优先搜索的过程是一个递归过程，因为深度优先搜索每次都是重复“访问顶点  $v$ ，再依次从  $v$  的未被访问的邻接点出发继续深度优先搜索”这个操作。

深度优先搜索的代码描述如下：

```
/*深度优先搜索一个连通图*/
void DFS(VNode G[],int v){
    int w;
    visit(v);           /*访问当前顶点*/
    visited[v] = 1;      /*将顶点v对应的访问标记置1*/
    w = FirstAdj(G,v);   /*找到顶点v的第一个邻接点，若无邻接点，返回-1*/
    while(w != -1){
        if(visited[w] == 0) /*该顶点未被访问*/
            DFS(G,w);      /*递归地进行深度优先搜索*/
        w = NextAdj(G,v); /*找到顶点v的下一个邻接点，若无邻接点，返回-1*/
    }
}
/*对图G=(V,E)进行深度优先搜索的主算法*/
void Travel_DFS(VNode G[], int visited[], int n){
    int i;
    for(i=0;i<n;i++){
        visited[i] = 0;    /*将标记数组初始化为0*/
    }
    for(i=0;i<n;i++){
        if(visited[i] == 0) /*若有顶点未被访问，从该顶点开始继续深度优先搜索*/
            DFS(G,i);
    }
}
```

通过上述代码可遍历一个包含  $n$  个顶点的图  $G$ 。在代码中设置了一个“访问标志数组”  $visited[n]$ ，该数组中有  $n$  个元素，在遍历过程中约定： $visited[i]=1$ ，表明图中的第  $i$  个顶点被访问过； $visited[i]=0$ ，表明图中的第  $i$  个顶点尚未被访问。

首先调用主算法函数  $Travel\_DFS()$ 。该函数包含 3 个参数： $G[]$  表示存储图结构的容器，这里可以理解为邻接表； $visited[]$  为设置的访问标志数组，用来标记图中被访问过的顶点； $n$  表示图中顶点的个数。现将标记数组  $visited[]$  初始化为 0，因为开始时图中的任何定点都没有被访问。然后从第一个没有被访问的顶点开始（即满足  $visited[i]=0$  条件的顶点  $i$ ）调用递归函数  $DFS()$ ，从该顶点开始深度优先遍历整个图。

函数  $DFS()$  是一个递归定义的函数。正如前面叙述的深度优先搜索的过程那样，函数  $DFS()$  首先访问当前顶点  $v$ ，这里用  $visit()$  函数进行顶点的访问，它需要根据具体需要由程序员自己定义。然后将顶点  $v$  对应的访问标记置 1，表明该顶点已被访问。接下来通过函数  $FirstAdj()$  得到当前顶点  $v$  的第一个邻接点，将其标号赋值给  $w$ ，如果顶点  $v$  无邻接点，返回 -1。然后通过一个循环从顶点  $v$  的第一个邻接点  $w$  开始深度优先搜索。每搜索完  $v$  的一个邻接点，就用函数  $NextAdj()$  得到  $v$  的下一个邻接点，将其标号赋值给  $w$ ，继续从  $w$  开始深度优先搜索，直到  $NextAdj()$  的返回值为 -1，表明顶点  $v$  已经没有邻接点了。这样与顶点  $v$  相通的所有图中的顶点都会被访问到。

之所以不能仅仅通过一个递归函数 DFS() 来遍历整个图, 是因为 DFS() 只能遍历到从起始顶点  $v$  开始所有与  $v$  相通的图中的顶点。如果这个图不是连通的, 如图 1-37 所示, 仅通过函数 DFS() 是无法遍历完全的。

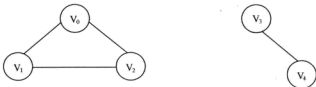


图 1-37 非连通图

如果用上述的深度优先搜索算法遍历图 1-37, 其遍历过程如下:

假设以  $v_0$  作为遍历的起点, 访问标志数组初始值为  $visited[5]=\{0,0,0,0,0\}$ 。

- (1) 访问顶点  $v_0$ ,  $visited[0]$  置 1, 这样  $visited[5]=\{1,0,0,0,0\}$ 。
- (2) 用函数 FirstAdj() 得到  $v_0$  的第一个邻接点, 例如  $v_1$ 。
- (3) 如果不返回-1, 即有邻接点:
  - (3.1) 访问顶点  $v_1$ ,  $visited[1]$  置 1, 这样  $visited[5]=\{1,1,0,0,0\}$ 。
  - (3.2) 用函数 FirstAdj() 得到  $v_1$  的第一个邻接点  $v_2$ , 因为  $v_0$  已被访问过。
  - (3.3) 如果不返回-1, 即有邻接点:
    - (3.3.1) 访问顶点  $v_2$ ,  $visited[2]$  置 1, 这样  $visited[5]=\{1,1,1,0,0\}$ 。
    - (3.3.2) 用函数 FirstAdj() 得到  $v_1$  的第一个邻接点, 返回-1, 因为都被访问过。
  - (3.4) 用函数 NextAdj() 得到  $v_1$  的下一个邻接点, 返回-1, 因为都被访问过。
- (4) 用函数 NextAdj() 得到  $v_0$  的下一个邻接点, 返回-1, 因为都被访问过。

DFS() 结束。

上述过程是深度优先搜索图 1-37 的包含顶点  $\{v_0, v_1, v_2\}$  的连通分量, 因此最终访问标志数组  $visited[5]=\{1,1,1,0,0\}$ , 表明只访问到了图中的 3 个顶点。因此还要通过函数 Travel\_DFS() 调用 DFS() 去遍历该图的另一个分支。在这里, 另一个分支的遍历过程省略, 留给读者自己思考。注意上述遍历过程的层次结构, 相同等级的标号 (例如: 1, 2, 3; 1.1, 1.2) 为同一层的递归调用, 这里是将递归调用过程展开来书写的。

## 1.7.6 图的遍历 (2) ——广度优先搜索

广度优先搜索的基本思想是: 从图中的指定顶点  $v$  出发, 先访问顶点  $v$ , 然后再依次访问  $v$  的各个未被访问的邻接点, 然后从这些邻接点出发, 按照同样的原则依次访问它们的未被访问的邻接点, 如此循环, 直到图中的所有与  $v$  相通的邻接点都被访问。与深度优先搜索一样, 若此时仍然有图中的顶点未被访问, 就另选图中的一个没有被访问到的顶点作为起始点, 继续广度优先搜索, 直到图中的所有顶点都被访问到为止。

通过上面的叙述可以体会到, 广度优先搜索的思想与深度优先搜索的思想不同。正如它们的名字, 深度优先搜索是从一个顶点  $v_0$  开始, 按照顺序: 访问  $v_0$ , 深度优先搜索  $v_0$  的第 1 个邻接点, 深度优先搜索  $v_0$  的第 2 个邻接点……深度优先搜索  $v_0$  的第  $n$  个邻接点。

它的特点是：从一点深入下去，深入到不能再深入为止，再从另一点进行深入下去。而广度优先搜索则是一种按层次搜索的方法，即先访问离顶点  $v_0$  最近的顶点  $v_1$ 、 $v_2$ 、 $v_3$ …再逐一访问离  $v_1$ 、 $v_2$ 、 $v_3$ …最近的顶点……

广度优先搜索的代码描述如下：

```
/*广度优先搜索一个连通图*/
void BFS(VNode G[],int v){
    int w;
    visit(v);           /*访问顶点v*/
    visited[v] = 1;      /*将顶点v对应的访问标记置1*/
    EnQueue(q,v);        /*顶点v入队列*/
    while(!emptyQ(q)){
        DeQueue(&q, &v); /*出队列，元素由v返回*/
        w = FirstAdj(G, v); /*找到顶点v的第一个邻接点，若无邻接点，返回-1*/
        while(w != -1){
            if(visited[w] == 0){
                visit(w);
                EnQueue(q,w); /*顶点w入队列*/
                visited[w] = 1;
            }
            w = NextAdj(G, v); /*找到顶点v的下一个邻接点，若无邻接点，返回-1*/
        }
    }
}

/*对图G=(V,E)进行广度优先搜索的主算法*/
void Travel_BFS(VNode G[], int visited[], int n){
    int i;
    for(i=0;i<n;i++){
        visited[i] = 0; /*将标记数组初始化为0*/
    }
    for(i=0;i<n;i++){
        if(visited[i] == 0) /*若有顶点未被访问，从该顶点开始继续广度优先搜索*/
            BFS(G,i);
    }
}
```

函数 `BFS()` 实现广度优先遍历一个连通的图。参数 `G[]` 表示图的存储容器，这里可以想象成邻接表，参数 `v` 表示广度优先遍历的访问起点。首先通过函数 `visit()` 访问顶点 `v`，每访问到一个顶点，都将对应的访问标记置 1。然后将已被访问过的顶点 `v` 入队列，接下来循环执行以下操作：

- (1) 从队列中取出队头元素（顶点 `v`）。
- (2) 用函数 `FirstAdj()` 得到该顶点 `v` 的第 1 个邻接点。
- (3) 如果该邻接点还未被访问，则用函数 `visit()` 访问该邻接点，并将该邻接点入队列，对应的访问标记置 1。
- (4) 用函数 `NextAdj()` 求顶点 `v` 的下一个邻接点，如果存在邻接点，跳回到步骤 (3)，如果不存在邻接点，跳回到步骤 (1)。

循环地执行上述操作，直到队列取空为止。一旦队列取空则表明最后访问到的顶点已经没有未访问到的邻接点了。如果应用上述广度优先搜索算法遍历如图 1-38 所示的图结构。

它的执行步骤如下：

- ① 访问  $v_0$ ，`visited[5]={1,0,0,0,0}`，并将顶点  $v_0$  入队列，此时队列中只有  $v_0$ 。
- ② 将  $v_0$  出队列，求  $v_0$  的第一个邻接点，例如  $v_1$ ，此时队列为空。



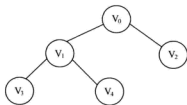


图 1-38 应用广度优先搜索的图

③ 由于  $v_1$  未被访问，因此访问  $v_1$ ， $\text{visited}[5]=\{1,1,0,0,0\}$ ，并将顶点  $v_1$  入队列，此时队列中只有  $v_1$ 。

④ 用  $\text{NextAdj}$  求顶点  $v_0$  的下一个邻接点，得到  $v_2$ 。

⑤ 由于  $v_2$  未被访问，因此访问  $v_2$ ， $\text{visited}[5]=\{1,1,1,0,0\}$ ，并将顶点  $v_2$  入队列，此时队列中有  $v_1$ 、 $v_2$ 。

⑥ 用  $\text{NextAdj}$  求顶点  $v_0$  的下一个邻接点，返回-1。

⑦ 将  $v_1$  出队列，求  $v_1$  的第一个邻接点，例如  $v_3$ ，此时队列为  $v_2$ 。

⑧ 由于  $v_3$  未被访问，因此访问  $v_3$ ， $\text{visited}[5]=\{1,1,1,1,0\}$ ，将顶点  $v_3$  入队列，此时队列中有  $v_2$ 、 $v_3$ 。

⑨ 用  $\text{NextAdj}$  求顶点  $v_1$  的下一个邻接点，得到  $v_4$ 。

⑩ 由于  $v_4$  未被访问，因此访问  $v_4$ ， $\text{visited}[5]=\{1,1,1,1,1\}$ ，将顶点  $v_4$  入队列，此时队列中有  $v_2$ 、 $v_3$ 、 $v_4$ 。

⑪ 用  $\text{NextAdj}$  求顶点  $v_1$  的下一个邻接点，返回-1。

⑫ 将  $v_2$  出队列，求  $v_2$  的第一个邻接点，返回-1，此时队列  $v_3$ 、 $v_4$ 。

⑬ 将  $v_3$  出队列，求  $v_3$  的第一个邻接点，返回-1，此时队列  $v_4$ 。

⑭ 将  $v_4$  出队列，求  $v_4$  的第一个邻接点，返回-1，此时队列空。

程序结束。

以上步骤就是对图 1-38 所示的图结构进行广度优先遍历的具体执行操作。其中用粗体字标注的步骤表示顶点元素出队列的操作。参看本节中给出的代码以及图 1-38，结合上述执行步骤的详细描述，读者就不难掌握广度优先搜索算法的执行过程。

其实图 1-38 所示的数据结构就是一个（二叉）树结构（树结构是图结构的一种特例），因此可知广度优先搜索算法同样适用于对树结构的遍历，因为树结构是标准的层次结构。

## 1.7.7 实例与分析

**【实例 1-7】** 用邻接表存储的形式创建一棵如图 1-39 所示的无向图，并应用深度优先搜索的方法遍历该图中的每个顶点，打印出每个顶点中包含的数据。

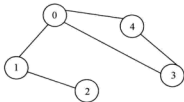


图 1-39 无向图

## 【分析】

首先要应用第 1.7.3 节中介绍的算法建立该图的邻接表。令图中顶点  $v_0=0$ 、 $v_1=1$ 、 $v_2=2$ 、 $v_3=3$ 、 $v_4=4$ 。然后从顶点  $v_0$  开始进行图的深度优先搜索。每访问到一个顶点，输出该顶点中的数据。在第 1.7.4 节中介绍了图的深度优先搜索算法，但这个算法不一定是针对图的邻接表的存储形式的。因此要将算法具体化，也就是针对邻接表的存储结构，具体化 FirstAdj() 和 NextAdj() 两个函数。具体的代码如下：

程序清单 1-7

```

/*----- 1-7.c -----*/
#include "stdio.h"

typedef struct ArcNode{
/*单链表中的结点的类型*/
int adjvex; /*该边指向的顶点在顺序表中的位置*/
struct ArcNode *next; /*下一条边*/
}ArcNode;

typedef struct VNode{
/*顶点类型*/
int data; /*顶点中的数据信息*/
ArcNode *firstarc; /*指向单链表，即指向第一条边*/
}VNode;

int visited[5]={0,0,0,0,0};

CreatGraph(int n, VNode G[]){
int i,e;
ArcNode *p, *q;
printf("Input the information of the vertex\n");
for(i=0;i<n;i++){
scanf("%d",&G[i]);
G[i].firstarc = NULL; /*初始化第一条边为空*/
}
for(i=0;i<n;i++){
printf("Creat the edges for the %dth vertex\n",i);
scanf("%d",&e);
while(e!=-1){
p = (ArcNode *)malloc(sizeof(ArcNode)); /*创建一条边*/
p->next = NULL;
p->adjvex = e;
if(G[i].firstarc == NULL) G[i].firstarc = p; /*1 结点的第一条边*/
else q->next = p; /*下一条边*/
q = p;
scanf("%d",&e);
}
}
}

int FirstAdj(VNode G[],int v){
if(G[v].firstarc != NULL) return (G[v].firstarc)->adjvex;
return -1;
}

int NextAdj(VNode G[],int v){
ArcNode *p;

```

```

    p = G[v].firstarc;
    while( p!= NULL){
        if(visited[p->adjvex]) p = p->next;
        else return p->adjvex;
    }
    return -1;
}

void DFS(VNode G[],int v){
    int w;
    printf("%d ",G[v]);          /*访问当前顶点,打印出该顶点中的数据信息*/
    visited[v] = 1;              /*将顶点 v 对应的访问标记置 1*/
    w = FirstAdj(G,v);           /*找到顶点 v 的第一个邻接点, 若无邻接点, 返回-1*/
    while(w != -1){
        if(visited[w] == 0)       /*该顶点未被访问*/
            DFS(G,w);            /*递归地进行深度优先搜索*/
        w = NextAdj(G,v);        /*找到顶点 v 的下一个邻接点, 若无邻接点, 返回-1*/
    }
}

main()
{
    VNode G[5];
    CreatGraph(5,G);
    DFS(G,0);
    getch();
}

```

在这段代码中, 通过函数 CreatGraph() 可以生成一个如图 1-39 所示的无向图的邻接表结构。然后通过函数 DFS() 深度优先遍历该图。这里没有用到像第 1.7.4 节中描述的算法那样通过一个 Travel\_DFS() 函数调用 DFS() 递归过程, 而是直接调用了 DFS()。这是因为事先已经知道该图是无向图, 而且又是一个连通图, 因此没有必要再像 1.7.4 节中描述的算法那样。读者在编写程序时也应该灵活掌握算法。为了操作的方便, 在本程序中将 visited 访问标志数组设置为全局数组。正如前面提到的那样, 由于本程序使用的是邻接表存储结构, 因此在设计 DFS() 函数时, 必须按照邻接表的结构特性具体化 FirstAdj() 和 NextAdj() 两个函数。

函数 FirstAdj() 定义为:

```

int FirstAdj(VNode G[],int v){
    if(G[v].firstarc != NULL) return (G[v].firstarc->adjvex;
    return -1;
}

```

该函数的功能是返回顶点 v 的第一个邻接点在数组 G 中的下标。如果该顶点无邻接点, 则返回-1。

函数 NextAdj() 定义如下:

```

int NextAdj(VNode G[],int v){
    ArcNode *p;
    p = G[v].firstarc;
    while( p!= NULL){
        if(visited[p->adjvex]) p = p->next; /*该顶点已被访问, 继续向下查找*/
        else return p->adjvex;             /*返回 v 的下一个邻接点在数组 G 中的下标*/
    }
}

```

```
return -1; /*已没有下一个邻接点*/  
}
```

该函数的功能是返回顶点  $v$  的下一个邻接点在数组  $G$  中的下标。也就是该顶点的下一个未被访问过的邻接点的数组下标。如果已没有下一个邻接点（该顶点所有的邻接点都被访问或者该顶点无邻接点），则返回-1。

该程序的运行结果如图 1-40 所示。

```
Input the information of the vertex  
0 1 2 3 4  
Creat the edges for the 0th vertex  
1 3 4 -1  
Creat the edges for the 1th vertex  
0 2 -1  
Creat the edges for the 2th vertex  
1 -1  
Creat the edges for the 3th vertex  
0 4 -1  
Creat the edges for the 4th vertex  
0 3 -1  
DFS Travel this undirected graph  
0 1 2 3 4 _
```

图 1-40 例 1-7 的运行结果

## 第2章 常用的查找与排序方法

在文件系统中，经常要对文件的记录进行各种各样的操作，主要包括以下内容：

- ❑ 文件的查找：对用户指定的文件中的记录进行查找，也称为检索。
- ❑ 插入记录：将一个新的记录插入到文件的指定位置。
- ❑ 记录的删除：从文件中删除指定的记录。
- ❑ 文件的排序：对文件中的记录按指定的关键字的顺序进行排序。
- ❑ 修改文件的记录：修改文件中记录的内容，并更新。

而这些操作都要放在计算机的内存中进行。于是就要设计一些特定的算法对内存中的文件记录进行上述的操作，然后再将文件的内容写回磁盘中。这个过程如图 2-1 所示。

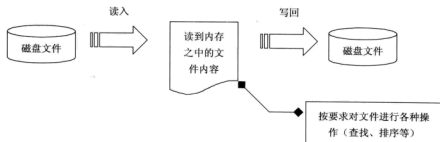


图 2-1 文件操作的基本过程

本章主要介绍一些常用的记录的查找算法和排序算法。

### 2.1 顺序查找

如果一个文件具有  $n$  个连续的记录，可将该文件读到内存中的一个顺序表中进行各种操作。顺序查找就是在文件的关键字集合  $\text{key}[1, 2, \dots, n]$  中找出与给定的关键字  $\text{key}$  相等的文件记录。如图 2-2 所示，磁盘中有这样一个顺序文件。

000001	小赵
000003	小钱
000002	小孙
000004	小李

图 2-2 顺序文件

该顺序文件是一个学生信息记录文件，其中包含4条记录。所谓关键字就是每个文件记录的唯一标识，在该文件中学号就可充当记录的关键字。在一个文件中，两个记录的关键字是不能相同的，所以该学生记录文件的关键字可以设定为学号。对文件中记录的查找就是对其关键字的查找，找到了关键字就找到了该关键字对应的那条记录。

如果要查找关键字为000002号同学的记录，就可以将该文件读到内存中的顺序表里，通过顺序查找的方法找到关键字key=000002的文件记录，并读取记录中的其他信息。

这种顺序查找的方法非常简单直观，可以描述为以下步骤。

- (1) 从文件的第一个记录开始，将每个记录的关键字与给定的关键字key进行比较。
- (2) 如果查找到某个记录的关键字等于key，则查找成功，返回该记录的地址。如果所有记录的关键字都与key进行了比较，但都未匹配，则本次查找失败，返回失败标志-1。其算法描述如下：

```
int sq_search(keytype key[], int n, keytype key)
{
    int i;
    for(i=0; i<n; i++)
        if(key[i] == key) /*查找成功*/
            return i;
    return -1; /*查找失败*/
}
```

在算法中， $n$ 表示记录的个数。key表示要查找记录的关键字。key[]为关键字顺序表，每个元素都是对应记录的关键字。例如key[0]为第0个记录的关键字。如果每条记录的信息与它的关键字都存放在一个记录中，如下定义：

```
typedef struct {
    keytype key; /* keytype 类型的关键字 key*/
    datatype data; /*记录中的其他信息*/
}RecordType;
```

每个记录中包含一个关键字域key和一个数据域data。这种记录的顺序查找算法可描述为：

```
int sq_search(RecordType r[], int n, keytype key)
{
    int i;
    for(i=0; i<n; i++)
        if( r[i].key == key ) /*查找成功*/
            return i; /*查找失败*/
    return -1;
}
```

顺序查找的方法的优点在于简单直观，对于被查找的记录在文件中的排列顺序没有限制，因此比较适合顺序文件的查找。同时这种查找思想也适合于对顺序表数据结构和链表数据结构中的元素进行查找。它的缺点在于平均查找长度过大，查找效率较低。

**【实例 2-1】** 一个结构体数组中存放的是学生的记录，每条记录包括：学号、姓名、成绩。该结构体数组中的内容如表 2-1 所示。

表 2-1 学生记录

学 号	姓 名	成 绩
1004	TOM	100
1002	LILY	95
1001	ANN	93
1003	LUCY	98

编写一个程序，要求输出 1001 编号同学的具体信息。

#### 【分析】

首先要定义结构体类型：

```
typedef struct student{
    int id;           /*学生编号*/
    char name[10];    /*学生姓名*/
    float score;      /*成绩*/
}Student;
```

然后初始化结构体数组。为了方便起见，这里直接初始化给结构体数组赋值。在实际的应用中，记录的获得往往都是从文件中读取的。

接下来采用顺序查找的方法找到 1001 编号同学的具体信息。这里的关键字为学生的学号，因为在学生记录中，只有学号能唯一标识学生的身份，姓名和成绩都不能（因为有重名的情况和成绩相同的情况）。

程序清单 2-1

```
/*----- 2-1.c -----*/
#include "stdio.h"
typedef struct student{
    int id;           /*学生编号*/
    char name[10];    /*学生姓名*/
    float score;      /*成绩*/
}Student;

int search(Student stu[],int n,int key){
    int i;
    for(i=0; i<n; i++){
        if( stu[i].id == key )           /*查找成功*/
            return i;
        return -1;                       /*查找失败*/
    }
}

main()
{
    Student stu[4] = {{1004,"TOM",100} ,
                      {1002,"LILY",95},
                      {1001,"ANN",93},
                      {1003,"LUCY",98}
    };                                     /*初始化结构体数组*/

    int addr;                             /*要查找的记录的地址*/
    addr = search(stu,4,1001);
    printf("Student ID:  %d\n",stu[addr].id); /*输出查找到的记录的信息*/
    printf("Student name: %s\n",stu[addr].name);
    printf("Student score: %f\n",stu[addr].score);
}
```

```

getche();
}

```

本程序的运行结果如图 2-3 所示。

```

Student ID: 1001
Student name: ANN
Student score: 93.000000

```

图 2-3 例 2-1 的运行结果

## 2.2 折半查找

如果从文件中读取的数据记录的关键字是有序排列的，则可以用一种效率更高的查找方法来查找文件中的记录，这就是折半查找法，又称为二分搜索。

像图 2-2 所示的顺序文件就不能采用折半查找的方法查找文件中的记录，这是因为它的关键字不是有序的，即关键字的排列不是递增或递减的。只有在顺序文件的关键字的排列是有序的（递增的或是递减的）情况下，才能应用折半查找的方法查找记录。

折半查找的基本思想是：减小查找序列的长度，分而治之地进行关键字的查找。它的查找过程是：先确定待查找记录的所在的范围，然后逐渐缩小查找的范围，直至找到该记录为止（也可能查找失败）。例如文件记录的关键字序列为：

(1, 3, 5, 6, 9, 12, 13, 17, 21, 28, 30)

该序列包含 11 个元素，而且关键字单调递增。现在想查找关键字 key 为 28 的记录。如果应用顺序查找法进行查找，需要将 28 之前的所有关键字与 key 进行比较，共需比较 10 次。如果用折半查找可以这样做。

设指针 low 和 high 分别指向关键字序列的上界和下界，即 low = 0, high = 10。指针 mid 指向序列的中间位置，即  $mid = \lfloor (low + high) / 2 \rfloor = 5$ 。在这里 low 指向关键字 1, high 指向关键字 30, mid 指向关键字 12。

(1, 3, 5, 6, 9, 12, 13, 17, 21, 28, 30)

↑
↑
↑  
 low                      mid                      high

(1) 首先将 mid 所指向的元素与 key 进行比较，因为 key=28，大于 12，这就说明待查找的关键字一定位于 mid 和 high 之间。这是因为原关键字序列是有序递增的。因此下面的查找工作只需在 [mid+1, high] 中进行。于是令指针 low 指向 mid+1 的位置，即 low = 6，也就是指向关键字 13，并将 mid 调整到指向关键字 21，即 mid = 8。

(1, 3, 5, 6, 9, 12, 13, 17, 21, 28, 30)

↑
↑
↑  
 low                      mid                      high

(2) 然后再将 mid 所指的元素与 key 进行比较，因为 key=28，大于 21，说明待查找的关键字一定位于 mid 和 high 之间。所以下面的查找工作仍然只需在 [mid+1, high] 中进行。于是令指针 low 指向 mid+1 的位置，即 low = 9，也就是指向关键字 28，并将 mid 调整到指向关键字 28，即 mid = 9。high 保持不变。



(1, 3, 5, 6, 9, 12, 13, 17, 21, 28, 30)

↑    ↑  
mid high  
low

(3) 接下来仍然将 mid 所指的元素与 key 进行比较, 比较相等, 查找成功, 返回 mid 的值 9。

假设要查找的关键字 key 为 29, 那么上述的查找还要继续下去。由于当前 mid 所指的元素为 28, 小于 29, 因此下面的查找工作仍然只需在 [mid+1, high] 中进行。将指针 low 指向 mid+1 的位置, 并调整指针 mid 的位置。这时指针 low、mid 与 high 三者重合, 都指向关键字 30, 它们的值都为 10。

(1, 3, 5, 6, 9, 12, 13, 17, 21, 28, 30)

↑  
high  
mid  
low

再将 mid 所指的元素与 key 进行比较, 因为 key=29, 小于 30, 说明待查找的关键字一定位于 low 和 mid 之间。所以下面的查找工作仍然只需在 [low, mid-1] 中进行。于是令指针 high 指向 mid-1 的位置, 即 high=9, 也就是指向关键字 28。这时指针 high 小于指针 low, 这表明本次查找失败。

折半查找的算法如下:

```
int bin_search(keytype key[], int n, keytype k)
{
    int low = 0, high = n-1, mid;
    while(low <= high){
        mid = (low+high)/2;
        if(key[mid] == k)
            return mid;          /*查找成功, 返回mid*/
        if(k > key[mid])
            low = mid + 1;        /*在后半序列中查找*/
        else
            high = mid - 1;       /*在前半序列中查找*/
    }
    return -1;                   /*查找失败, 返回-1*/
}
```

在算法中,  $n$  表示记录的个数。key 表示要查找记录的关键字。key[] 为关键字顺序表, 每个元素都是对应记录的关键字。例如 key[0] 为第 0 个记录的关键字。如果每条记录的信息与它的关键字都存放在一个记录中, 如下定义:

```
typedef struct {
    keytype key;          /* keytype 类型的关键字 key */
    datatype data;        /* 记录中的其他信息 */
} RecordType;
```

每个记录中包含一个关键字域 key 和一个数据域 data。这种记录的折半查找算法可描述为:

```
int bin_search(RecordType r[], int n, keytype k)
{
```

```

int low = 0, high = n-1, mid;
while(low<=high){
    mid = (low+high)/2;    /*设置 mid 指针的值*/
    if( r[mid].key == k)
        return mid;      /*查找成功, 返回 mid*/
    if(k > r[mid].key)
        low = mid + 1;    /*在后半序列中查找*/
    else
        high = mid - 1;   /*在前半序列中查找*/
    }
    return -1;            /*查找失败, 返回-1*/
}

```

这里要注意, 以上的算法只适用于关键字顺序递增的有序表查找。如果该顺序表是关键字递减的, 则算法需要改动, 只是在 low 指针与 high 指针的修改上对调即可, 读者可自己完成。

折半查找只能应用于关键字有序的顺序表的查找, 也就是说先要将文件记录存放在内存中定义的顺序表中, 再进行指定记录的查找。折半查找一般不适用于对链表中记录的查找。

折半查找的效率比顺序查找的效率要高很多。如果一个顺序表中有 1000 个关键字, 应用顺序查找方法查找指定的关键字, 平均要比较 500 次。而应用折半查找方法查找指定的关键字, 平均只要比较 9 次。因此对于有序的顺序文件记录的查找, 应当使用折半查找方法。

**【实例 2-2】** 有一个数组 A[10], 里面存放了 10 个整数, 顺序递增。

A[10] = {2, 3, 5, 7, 8, 10, 12, 15, 19, 21}

**【分析】**

任意输入一个用数字  $n$ , 用折半查找法找到  $n$  位于数组中的位置。如果  $n$  不属于数组 A, 显示错误提示。

这是一个折半查找法的简单应用。折半查找不仅可以通过查找关键字对文件记录进行查找, 也可应用于简单的数组, 顺序表结构等查找。在这里同样要求数组中的元素有序。

程序清单 2-2

```

/*----- 2-2.c -----*/
#include "stdio.h"
bin_search(int A[],int n,int key){
    int low,high,mid;
    low = 0;
    high = n-1;
    while(low<=high)
    {
        mid = (low + high)/2;
        if(A[mid]==key)return mid;    /*查找成功, 返回 mid*/
        if(A[mid]<key){
            low = mid + 1;            /*在后半序列中查找*/
        }
        if(A[mid]>key){
            high = mid - 1;          /*在前半序列中查找*/
        }
    }
}

```

```

    return -1;                                /*查找失败，返回-1*/
}
main()
{
    int A[10] = {2,3,5,7,8,10,12,15,19,21},i,n ,addr;
    printf("The contents of the Array A[10] are\n");
    for(i=0;i<10;i++)
        printf("%d ",A[i]);                /*显示数组 A 中的内容*/
    printf("\nPlease input a interger for search\n");
    scanf("%d",&n);                        /*输入待查找的元素*/
    addr = bin_search(A,10,n);              /*折半查找，返回该元素在数组中的下标*/
    if(-1 != addr)                          /*查找成功*/
        printf("%d is at the %dth unit is array A\n",n,addr);
    else printf("There is no %d in array A\n",n); /*查找失败*/
    getch();
}

```

本程序的运行结果如图 2-4 所示。

```

The contents of the Array A[10] are
2 3 5 7 8 10 12 15 19 21
Please input a interger for search
7
7 is at the 3th unit is array A

```

图 2-4 例 2-2 的运行结果

**注意：**在这里数组的下标是从 0 开始的，因此 7 位于数组 A 的第 3 单元。

## 2.3 排序的概述

在前面的两节中简要地介绍了顺序文件的查找算法（顺序查找和折半查找）。其实文件记录的排序在文件的操作中地位更加重要。例如，如果想应用工作效率很高的折半查找算法检索文件中的某一条记录，一个先决条件就是文件记录必须按关键字有序排列，否则是无法使用折半查找的。这就需要将文件读入内存之后先进行一步排序的操作，使得文件的记录按照关键字的顺序有序排列，再进行折半查找。除此之外，排序的意义不仅限于对文件记录的排序，在计算机计算和处理加工数据时，经常会直接或间接地涉及数据的排序问题。因此掌握一些排序算法是很有用的。

对于文件而言，排序可以理解为：根据文件记录的关键字的值的递增或者递减关系将文件的记录的次序进行重新排列的过程。排序后的文件记录一定是按关键字值有序排列的。例如，最开始从磁盘中读出的文件如图 2-5 所示。

关键字	其他信息
3	.....
2	.....
1	.....
4	.....
5	.....

图 2-5 无序的文件

该文件的关键字显然是无序排列的，这样的文件无法应用折半查找的方法检索某一条记录。因此可以将该文件进行排序操作，也就是在内存中将文件的记录按关键字值的顺序进行调整。如果将排序后的文件再写回磁盘，那么以后该文件就是有序的了。按关键字递增的顺序将该文件排序后如图 2-6 所示。

关键字	其他信息
1	.....
2	.....
3	.....
4	.....
5	.....

图 2-6 有序的文件

其实将排序的概念推广之后，排序的对象并不只限于文件的记录。可以简单地将排序操作理解为：将一个按值无序的数据序列转换成为一个按值有序的数据序列的过程。例如将一个无序的数组  $A[5]=\{7, 5, 8, 2, 1\}$  排列成有序的数组  $A[5]=\{1, 2, 5, 7, 8\}$  的过程当然也是一个排序过程。文件的排序过程只是排序过程的一个特例。在后面介绍的各种排序算法，都是针对这种数据元素序列进行的，并不限于对文件的排序。读者只要掌握了每一种排序方法的基本思想，就很容易将它应用到实际的工作中。

## 2.4 直接插入排序

直接插入排序 (Straight Insertion Sort) 是一种最为简单的排序方法，因此也被称为简单插入排序。

直接插入排序的基本思想是：第  $i$  趟排序将序列中的第  $i+1$  个元素  $k_{i+1}$  插入到一个已经按值有序的子序列  $(k_1, k_2, \dots, k_i)$  中的合适的位置，使得插入后的序列仍然保持按值有序。

下面通过一个例子来介绍直接插入排序的执行过程。

有这样一个数据元素序列  $\{3, 6, 4, 2, 11, 10, 6'\}$ ，其中  $6'$  表示该元素与本序列中其他元素有重复，在这里加以区分。

首先该序列中已存在一个有序的子序列。

$\{ (3, 6), 4, 2, 11, 10, 6' \}$

接下来要将 4 插入到这个子序列中，得到一个含有 3 个元素的有序的子序列。首先要判断 4 应当插入的位置，然后才能进行插入。具体方法是，从元素 6 开始向左查找。因为 4 小于 6，因此要将元素 6 向后移动；又因为元素 4 大于 3，因此将 4 插入到 3 和 6 之间。这样在原序列中得到一个新的按值有序的子序列。

$\{ (3, 4, 6), 2, 11, 10, 6' \}$

上述这个插入过程称为一趟直接插入排序。

按照这种插入的方法，可将后续的 4 个元素逐一插入到前面的子序列中，形成新的子序列。当子序列与原序列长度一样时，插入排序结束。

在进行第1趟的插入排序时,可将第1个元素看成长度为1,按值有序的子序列,然后将第2个元素插入到这个子序列之中。依次类推,第*i*趟排序将序列中的第*i*+1个元素 $k_{i+1}$ 插入到一个已经按值有序的子序列 $(k_1, k_2, \dots, k_i)$ 中的合适的位置,使得插入后的序列仍然保持按值有序。

上述的元素序列{3, 6, 4, 2, 11, 10, 6'}的直接插入排序过程如图2-7所示。

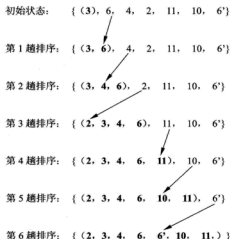


图2-7 直接插入排序

从这个插入排序的过程中可以看出:一个包含有*n*个元素的序列,需要*n*-1趟的直接插入排序就可以将原序列排列有序。直接插入排序的算法描述如下:

```
void insertsort(keytype k[],int n){
    int i,j;
    keytype tmp;
    for(i=2;i<=n;i++){
        tmp = k[i];           /*将k[i]保存在临时变量tmp中*/
        j = i-1;
        while(j>0 && tmp<k[j]) /*找到tmp的插入位置*/
            k[j+1] = k[j--];   /*将k[j]后移,再将j减1*/
        k[j+1] = tmp;         /*将元素tmp插入到指定位置,第i-1趟插入完成*/
    }
}
```

通过上述算法可以看出,将元素 $k[i]=tmp$ 插入到子序列 $k[1] \sim k[i-1]$ 中时要进行一些列的比较,将比 $k[i]$ 大的元素向后移动,直到找到比 $k[i]$ 小的第一个元素为止,将 $k[i]=tmp$ 插入到这个元素的前面。因此本算法的功能是将序列从小到大递增排列。也可以模仿该算法设计出从大到小的排序算法。

在本算法中,数据序列用一个keytype类型的数组k[]存放,第一个单元k[0]没有使用。因此在算法中可以省却掉临时变量tmp,用k[0]存储每次待插入的元素 $k[i]$ ,这样就节省了系统的空间开销。因为keytype类型的大小是不一定的。

**【实例2-3】**编写一个C程序,实现数据序列{2, 5, 6, 3, 7, 8, 0, 9, 12, 1}的直接插入排序,要求从大到小排列,并输出排序后的数列元素。

## 【分析】

该数据序列包含 10 个元素，因此可将它存放到一个含有 11 个单元的数组中，第 0 号单元作为存放每次待插入的元素  $A[j]$  的空间。在这里不能直接照搬前面给出的算法，前面的算法只能实现序列的从小到大排序，在这里要加以改造。

程序清单 2-3

```

/*----- 2-3.c -----*/
#include "stdio.h"
insertsort(int a[],int n)          /*直接插入排序*/
{
    int i,j;
    for(i=2;i<=n;i++)
    {
        a[0] = a[i];
        j = i - 1;
        while(j>0 && a[0]>a[j])    /*改变判断条件，实现从大到小地排列*/
            a[j+1] = a[j--];
        a[j+1] = a[0];            /*将元素 a[0]插入指定位置*/
    }
}

main()
{
    int i,a[11] = {-11,2,5,6,3,7,8,0,9,12,1};
    printf("The original data array is\n");
    for(i=1;i<=10;i++)            /*显示原序列之中的元素*/
        printf("%d ",a[i]);
    insertsort(a,10);              /*插入排序*/
    printf("\nThe result of insertion sorting for the array is\n");
    for(i=1;i<=10;i++)
        printf("%d ",a[i]);      /*输出排序后的结果*/
    getch();
}

```

本程序的运行结果如图 2-8 所示。

```

The original data array is
2 5 6 3 7 8 0 9 12 1
The result of insertion sorting for the array is
12 9 8 7 6 5 3 2 1 0 _

```

图 2-8 例 2-3 的运行结果

## 2.5 选择排序

选择排序 (selection sort) 也是一种比较常见的排序方法。它的基本思想是：第  $i$  趟排序从序列的后  $n-i+1$  ( $i=1, 2, \dots, n-1$ ) 个元素中选择一个最小的元素，与该  $n-i+1$  个元素的最前面那个元素进行位置交换，也就是与第  $i$  个位置上的元素进行交换，直到  $i=n-1$ 。直观地讲，每一趟的选择排序就是从序列中未排好顺序的元素中选择一个最小的元素，将该元素与这些未排好顺序的元素的一个元素交换位置。

下面通过一个例子来介绍选择排序的执行过程。

有这样一个数据元素序列{3, 6, 4, 2, 11, 10, 6'}，其中 6'表示该元素与本序列中其他元素有重复，在这里加以区分。

第1趟排序，从后7个元素（即全部元素）中选择一个最小的元素，并将它与第1个元素交换位置。全部序列中最小的元素是2，因此将2与第1个元素3交换位置，得到序列：

{ (2), 6, 4, 3, 11, 10, 6' }

第2趟排序，从后6个元素中选择一个最小的元素，并将它与第2个元素交换位置。后6个元素序列中最小的元素是3，因此将3与第2个元素6交换位置，得到序列：

{ (2, 3), 4, 6, 11, 10, 6' }

第3趟排序，从后5个元素中选择一个最小的元素，并将它与第3个元素交换位置。后5个元素序列中最小的元素是4，因此本次选择不需要交换元素位置，得到序列：

{ (2, 3, 4), 6, 11, 10, 6' }

接下来的每一趟操作都是按照这样的“选择-交换”方法进行的。不难看出，选择排序的第*i*趟排序就是从序列的后  $n-i+1$  ( $i=1,2,\dots,n-1$ ) 个元素中选择一个最小的元素，并与第*i*个位置上的元素进行交换的过程。一个包含  $n$  个元素的数据序列需要  $n-1$  趟的选择排序。因为第  $n$  个元素不需要再选择，它一定是最大的。

上述的元素序列{3, 6, 4, 2, 11, 10, 6'}的选择过程如图2-9所示。

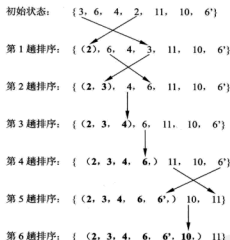


图2-9 选择排序的过程

选择排序的算法描述如下：

```
void selectsort(keytype k[],int n)
{
    int i,j,min;
    keytype tmp;
    for(i=1;i<=n-1;i++){
        min = i;
        for(j=i+1;j<=n;j++)
            if(k[j] < k[min])
                /*在后 n-i+1 个元素中找到最小的元素位置*/
    }
}
```

```

        min = j;                /*用 min 记录下最小元素的位置*/
        if(min != i){           /*如果最小的元素不位于后 n-i+1 个元素档第 1 个*/
            tmp = k[min];
            k[min] = k[i];       /*元素的交换*/
            k[i] = tmp;
        }
    }
}

```

在本算法中, tmp 为一个临时变量,用来实现序列中元素的交换。也可以省略掉这个临时变量使用 k[0]代替,因为在这里第一个单元 k[0]没有被使用。此外本算法依然只实现序列的从小到大的排序。如果将该算法稍加改造就可实现序列的从大到小排序。

**【实例 2-4】**编写一个 C 程序,实现数据序列 {2, 5, 6, 3, 7, 8, 0, 9, 12, 1} 的选择排序,要求从大到小排列,并输出排序后的数列元素。

#### 【分析】

该数据序列包含 10 个元素,因此可将它存放到一个含有 11 个单元的数组中,第 0 号单元可作为数据位置交换的临时空间。在这里不能直接照搬前面给出的算法,前面的算法只能实现序列的从小到大排序,在这里要加以改造。

程序清单 2-4

```

/*----- 2-4.c -----*/
#include "stdio.h"
void selectsort(int k[],int n) /*选择排序*/
{
    int i,j,max;
    for(i=1;i<=n-1;i++){
        max = i;
        for(j=i+1;j<=n;j++) /*在后 n-i+1 个元素中找到最小的元素位置*/
            if(k[j] > k[max]) /*用 min 记录下最小元素的位置*/
                max = j; /*如果最小的元素不位于后 n-i+1 个元素档第 1 个*/
        if(max != i){
            k[0] = k[max]; /*元素的交换*/
            k[max] = k[i];
            k[i] = k[0];
        }
    }
}

main()
{
    int i,a[11] = {-111,2,5,6,3,7,8,0,9,12,1}; /*初始化序列, a[0]可任意置数*/
    printf("The original data array is\n");
    for(i=1;i<=10;i++) /*显示原序列之中的元素*/
        printf("%d ",a[i]);
    selectsort(a,10); /*执行选择排序*/
    printf("\nThe result of selection sorting for the array is\n");
    for(i=1;i<=10;i++)
        printf("%d ",a[i]); /*输出排序后的结果*/
    getch();
}

```

本程序的运行结果如图 2-10 所示。



```

The original data array is
2 5 6 3 7 8 0 9 12 1
The result of selection sorting for the array is
12 9 8 7 6 5 3 2 1 0

```

图 2-10 例 2-4 的运行结果

## 2.6 冒泡排序

冒泡排序 (bubble sort) 是最为常用的一种排序方法。它是一类具有“交换”性质的排序方法。冒泡排序的基本思想可描述如下:

将序列中的第 1 个元素与第 2 个元素进行比较, 若前者大于后者, 则将第 1 个元素与第 2 个元素进行位置交换, 否则不交换。

再将第 2 个元素与第 3 个元素进行比较, 同样若前者大于后者, 则将第 2 个元素与第 3 个元素进行位置交换, 否则不交换。

依此类推, 直到将第  $n-1$  个元素与第  $n$  个元素进行比较为止。

这个过程称为第 1 趟冒泡排序。经过第 1 趟冒泡排序后, 将长度为  $n$  的序列中最大的元素置于了序列的尾部, 即第  $n$  个位置上。

然后再对剩下的  $n-1$  个元素作同样的操作, 这叫做第 2 趟冒泡排序。经过了第 2 趟的冒泡排序, 将剩下的  $n-1$  个元素中最大的元素置于了序列的  $n-1$  的位置上。

如此进行下去, 当执行完第  $n-1$  趟的冒泡排序后, 就可以将序列中剩下 2 个元素中的最大的元素置于了序列的第 2 个位置上。第 1 个位置上的元素就是该序列中最小的元素。这样冒泡排序完成。

元素序列 {3, 6, 4, 2, 11, 10, 6'} 的冒泡排序的过程如图 2-11 所示。



图 2-11 冒泡排序的过程

如图 2-11 所示, 一个包含  $n$  个元素的序列要进行  $n-1$  趟的冒泡排序, 第  $i$  趟冒泡排序

将第  $1 \sim n-i+1$  个元素中最大的元素交换到第  $n-i+1$  个位置上。因此第  $n-1$  趟冒泡排序就是将第  $1 \sim 2$  个元素中最大的元素交换到第 2 个位置上。因此不难想象,这种排序方法就是通过序列中邻接元素之间的交换,使较小的元素逐步从序列的后端移到序列的前端,使较大的元素从序列的前端移到后端。这就像水底的气泡不断向上“冒”一样,因此人们形象地称这种排序方法为“冒泡排序”法。

冒泡排序法的算法描述如下:

```
void bubblesort(keytype k[],int n){
    int i,j,tmp;
    for(i=1;i<=n-1;i++)          /*执行 n-1 趟排序*/
        for(j=1;j<=n-i;j++){
            if(k[j]>k[j+1]){        /*数据交换*/
                tmp = k[j+1];
                k[j+1] = k[j];
                k[j] = tmp;
            }
        }
}
```

该算法仍然只对数据进行从小到大的排列。 $k[]$  的 0 号单元不存放内容,因此也可省略中间变量  $tmp$ , 而用  $k[0]$  作为数据交换的临时缓冲区。

但是上述的算法并不是一个优秀的冒泡排序算法。细心的读者会发现在图 2-11 所示的冒泡排序的过程中,从第 4 趟排序开始,序列本身就没有再发生任何变化。只是相邻元素的比较,并没有发生相邻元素的交换。因此可以想到,从第 4 趟排序操作往下的比较运算其实都可以不再进行。不难理解,如果某一趟排序过程中只有元素之间的比较操作,而没有发生元素的位置交换,那就说明到本趟排序为止,序列中的元素已经按值有序,因此不需要再进行下一趟排序了,排序可以结束。认识到这一点就可以将原来的冒泡排序算法加以改进,使它的排序效率更高。

在算法中可以设置一个标志变量  $flag$ 。规定当  $flag$  为 1 时,说明本趟排序中仍有元素交换的动作,因此还需进行下一趟的比较。当  $flag$  为 0 时,说明本趟排序中已经没有了元素的交换,只有元素的比较,因此表明该序列已经按值有序,排序可以停止。

改进后的冒泡排序算法如下:

```
void bubblesort(keytype k[],int n){
    int i,j,tmp,flag = 1;
    for(i=1;i<=n-1 && flag == 1;i++){
        flag = 0;
        for(j=1;j<=n-i;j++){
            if(k[j]>k[j+1]){
                tmp = k[j+1];
                k[j+1] = k[j];
                k[j] = tmp;
                flag = 1;          /*发生数据交换 flag 置 1*/
            }
        }
    }
}
```

使用改进后的算法可以减少排序时的比较次数,提高冒泡排序的效率。

**【实例 2-5】** 编写一个 C 程序,实现数据序列 {2, 5, 6, 3, 7, 8, 0, 9, 12, 1} 的冒

泡排序，要求从大到小排列，并输出排序后的数列元素。

### 【分析】

该数据序列包含 10 个元素，因此可将它存放到一个含有 11 个单元的数组中，第 0 号单元可作为数据位置交换的临时空间。在这里不能直接照搬前面给出的算法，前面的算法只能实现序列的从小到大排序，在这里要加以改造。

程序清单 2-5

```

/*----- 2-5.c -----*/
#include "stdio.h"
void bubblesort(int k[],int n){          /*冒泡排序*/
    int i,j,tmp ,flag = 1;
    for(i=1;i<=n-1 && flag == 1;i++){    /*执行 n-1 趟排序*/

        flag = 0;
        for(j=1;j<=n-i;j++){
            if(k[j]<k[j+1]){              /*数据交换*/
                tmp = k[j+1];
                k[j+1] = k[j];
                k[j] = tmp;
                flag = 1;
            }
        }
    }
}

main()
{
    int i,a[11] = {-111,2,5,6,3,7,8,0,9,12,1};
    printf("The original data array is\n"); /*初始化序列，a[0]可任意置数*/
    for(i=1;i<=10;i++)                     /*显示原序列之中的元素*/
        printf("%d ",a[i]);
    bubblesort(a,10);                       /*执行冒泡排序*/
    printf("\nThe result of bubble sorting for the array is\n");
    for(i=1;i<=10;i++)
        printf("%d ",a[i]);                /*输出排序后的结果*/
    getch();
}

```

本程序的运行结果如图 2-12 所示。

```

The original data array is
2 5 6 3 7 8 0 9 12 1
The result of bubble sorting for the array is
12 9 8 7 6 5 3 2 1 0

```

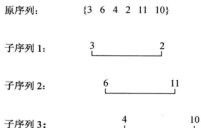
图 2-12 例 2-5 的运行结果

## 2.7 希尔排序

希尔排序 (Shell's sort) 又称为“缩小增量排序” (Diminishing Increment Sort)，是

由希尔在1959年提出的。希尔排序是对插入排序的一种改进,在效率上较前面所讲的几种排序方法有较大改进。

希尔排序的基本思想是:设定一个元素间隔增量  $gap$ ,将参加排序的序列按这个间隔数  $gap$  从第1个元素开始依次分成若干个子序列。例如最开始  $gap=3$ ,下面的序列可这样划分:



将序列 {3, 6, 4, 2, 11, 10} 划分为3个子序列。在子序列中采用其他排序方法(例如冒泡排序)。然后缩小增量  $gap$ ,重新将整个序列按照新的间隔数  $gap$  进行划分,再分别对每个子序列进行排序。如此将“缩小增量  $gap$ —划分序列—将每个子序列排序”操作进行下去,直到间隔数增量  $gap=1$  为止。

由于排序时每一趟都是以不同的间隔数对子序列进行排序,因此元素的移动在子序列中是跳跃式的。间隔数  $gap$  越大,跳跃的跨度就越大。一般情况下,当增量  $gap$  缩小到1时,序列大都几乎已经按值有序,不需要进行较多的元素移动就能达到排序的目的。

上述的元素序列 {3, 6, 4, 12, 11, 10, 8, 9} 的希尔过程如图2-13所示。

初始状态: {3, 6, 4, 12, 11, 10, 8, 9}

第1趟排序,  $gap=4$ : {3, 6, 4, 9, 11, 10, 8, 12}

第2趟排序,  $gap=2$ : {3, 6, 4, 9, 8, 10, 11, 12}

第3趟排序,  $gap=1$ : {3, 4, 6, 8, 9, 10, 11, 12}

图2-13 希尔排序

希尔排序的算法描述如下:

```
void shellsort(keytype k[],int n)
{
    int i, j, flag, gap = n;
    keytype tmp;
    while(gap > 1){
        gap = gap/2;          /*增量缩小, 每次减半*/
        do{                  /*子序列应用冒泡排序*/
            flag = 0;
            for(i=1; i<=n-gap; i++){
                j = i + gap;
                if(k[i]>k[j]){
                    tmp = k[i];
                    k[i] = k[j];
                    k[j] = tmp;
                    flag = 1;
                }
            }
        }
    }
}
```

```

        }while(flag !=0);
    }
}

```

上述算法描述希尔排序的过程。参数  $n$  为序列中元素的个数。将  $n$  赋值给变量  $gap$ ，当  $gap$  大于 1 时，循环地执行希尔排序的每一趟排序操作。在执行每一趟排序操作之前， $gap$  的值都要减半，增量缩小。在每趟的排序中使用的是冒泡法排序，当然这里也可以使用其他的排序方法。

希尔排序算法的速度是一系列增量  $gap$  的函数，要对希尔排序的算法做出准确地分析并不容易，如何选取最合适的间隔数序列才能达到最优的排序效果是至今尚未解决的数学难题。

**【实例 2-6】** 编写一个 C 程序，实现数据序列 {2, 5, 6, 3, 7, 8, 0, 9, 12, 1} 的希尔排序，要求从大到小排列，并输出排序后的数列元素。

#### 【分析】

本序列中包含 10 个元素，因此间隔数  $gap$  的初始值为  $10/2=5$ 。以后在进行每一趟排序之前， $gap$  的值都要减半，直到  $gap$  等于 1 为止。在这里要注意，本题要求将序列从大到小排列，因此要将每一趟的排序方法（上述算法中采用的是冒泡排序法）作适当的调整。

程序清单 2-6

```

/*----- 2-6.c -----*/
#include "stdio.h"
void shellsort(int k[],int n)
{
    int i, j, flag ,gap = n;
    int tmp;
    while(gap > 1){
        gap = gap/2;          /*增量缩小，每次减半*/
        do{                  /*子序列应用冒泡排序*/
            flag = 0;
            for(i=1;i<=n-gap;i++){
                j = i + gap;
                if(k[i]<k[j]){
                    tmp = k[i];
                    k[i] = k[j];
                    k[j] = tmp;
                    flag = 1;
                }
            }
        }while(flag !=0);
    }
}

main()
{
    int i,a[11] = {-111,2,5,6,3,7,8,0,9,12,1};
    /*初始化序列，a[0]可任意置数*/
    printf("The original data array is\n");
    for(i=1;i<=10;i++){
        printf("%d ",a[i]);
        /*显示原序列之中的元素*/
    }
    shellsort(a,10);
    /*执行希尔排序*/
    printf("\nThe result of Shell's sorting for the array is\n");
}

```

```

for(i=1;i<=10;i++)
    printf("%d ",a[i]);          /*输出排序后的结果*/
getche();
}

```

本程序的运行结果如图 2-14 所示。

```

The original data array is
2 5 6 3 7 8 0 9 12 1
The result of Shell's sorting for the array is
12 9 8 7 6 5 3 2 1 0

```

图 2-14 例 2-6 的运行结果

## 2.8 快速排序

快速排序 (quick sort) 是由 C.A.R Hoarse 提出的一种排序算法, 它是冒泡排序的一种改进算法。由于快速排序算法元素之间的比较次数较少, 速度较快, 因而得名快速排序。在各种内部排序方法中, 快速排序被认为是目前最好的一种排序方法。

快速排序算法的基本思想是: 在当前的排序序列 ( $k_1, k_2, \dots, k_n$ ) 中任意选取一个元素, 将该元素称为基准元素或支点, 把小于等于基准元素的所有元素都移到基准元素的前面, 把大于基准元素的所有元素都移到基准元素的后面, 这样使得基准元素所处的位置恰好就是排序的最终位置, 并且把当前参加排序的序列划分为前后两个子序列。其中, 前面的子序列中的元素都小于等于基准元素, 后面的子序列的元素都大于基准元素。接下来分别对这两个子序列重复上述的排序操作 (如果子序列的长度大于 1), 直到使得所有元素都被移动到排序后它们应处的最终的位置上。

快速排序方法之所以效率较高, 是因为每一次元素的移动都是跳跃式的。因为每趟的排序都要指定一个基准点, 把小于等于基准元素的所有元素都移到基准元素的前面, 把大于基准元素的所有元素都移到基准元素的后面, 这样每次元素的移动就不会像冒泡排序那样只能在相邻元素之间进行, 元素移动的间隔距离较大, 因此总的比较和移动次数减少, 排序的速度自然提高。

在排序的过程中, 每次按照基准元素将原序列划分为前后两个子序列的过程称为一次划分操作。一次划分操作的过程如下:

假设原序列为: {5, 7, 4, 2, 11, 10, 6}。

首先设置两个变量  $i$  和  $j$ 。  $i=1$ , 指向元素 5;  $j=7$ , 指向元素 6。设定基准元素为  $i$  指向的元素 5, 如图 2-15 所示。

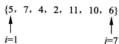


图 2-15 序列的初始状态

- (1) 反复执行  $i=i+1$  的操作, 直到  $i$  指向的元素大于等于基准元素 5, 或者  $i$  指向序列

尾部, 即  $i=7$  为止。然后反复执行  $j=j-1$  的操作, 直到  $j$  指向的元素小于等于基准元素 5, 或者  $j$  指向序列的首部, 即  $j=1$  为止。此时序列的状态如图 2-16 所示。

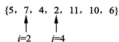


图 2-16 步骤 (1) 后序列的状态

(2) 若此时  $i < j$ , 则将  $i$  与  $j$  指向的元素进行交换, 如图 2-17 所示。

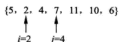


图 2-17 实现第一次交换

然后重复执行步骤 (1)、(2) 或 (3)。执行完步骤 (2) 后, 序列的状态如图 2-18 所示。

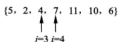


图 2-18 执行步骤 (2) 后序列的状态

(3) 若此时  $i \geq j$ , 则将基准元素与  $j$  指向的元素交换位置, 如图 2-19 所示。

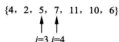


图 2-19 完成原序列的第一次划分

至此完成了原序列的第一次划分, 接下来分别递归地对基准点 5 前后的子序列中长度大于 1 的子序列重复执行上述操作, 直到整个序列排序结束。

序列的每一次划分操作都是按照: 步骤 (1) → 步骤 (2) → 步骤 (1) … 步骤 (2) → 步骤 (1) → 步骤 (3) → 结束, 这样的过程进行的。也就是说, 只有当  $i \geq j$  时才将本次划分的基准元素放置到它最终的位置上。对于本例来说, 本次划分的基准元素为 5, 它的排序后最终位置就是位于序列的第 3 个位置上。这样序列中元素 5 前面的元素都小于 5, 元素 5 后面的元素都大于 5。

进行完序列的一次划分之后, 再对基准元素 5 前后的两个子序列分别重复进行上述操作。对于每个子序列的操作又是一次划分的过程, 每次划分过程的基准元素仍可设定为该子序列的第一个元素。

归纳起来, 快速排序的递归算法如下:

```
void quicksort(keytype k[], int s, int t)
{
    int i, j;
    if (s < t) {
        i = s;
```

```

    j = t+1;
    while(1){
        do i++;
        while(!(k[s]<=k[i] || i==t)); /*重复执行 i++操作*/
        do j--;
        while(!(k[s]>=k[j] || j==s)); /*重复执行 j--操作*/
        if(i<j)
            swap(k[i],k[j]); /*交换 k[i]和 k[j]的位置*/
        else
            break;
    }
    swap(k[s],k[j]); /*交换基准元素与 k[j]的位置,完成一次划分*/
    quick(k,s,j-1); /*递归排序基准元素前面的子序列*/
    quick(k,j+1,t); /*递归排序基准元素后面的子序列*/
}

```

该算法中,子算法 swap(k1,k2)的功能是交换 k1 和 k2 的位置。

由于快速排序的算法特性的约束,快速排序一般适用于顺序表线性结构或数组序列的排序,它并不适合于在链表结构上实现排序。

**【实例 2-7】**编写一个 C 程序,实现数据序列{2, 5, 6, 3, 7, 8, 0, 9, 12, 1}的快速排序。要求从大到小排列,并输出排序后的数列元素。

#### 【分析】

前面介绍的快速排序算法只适用于序列的从小到大的排序,在这里要实现序列的从大到小的排序,因此算法上要稍加修改。

程序清单 2-7

```

/*----- 2-7.c -----*/
#include "stdio.h"

void swap(int *a,int *b)
{
    /*序列中元素位置的交换*/
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}

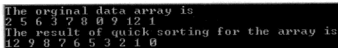
void quicksort(int k[], int s,int t)
{
    /*快速排序*/
    int i,j;
    if(s<t){
        i = s;
        j = t+1;
        while(1){
            do i++;
            while(!(k[s]>=k[i] || i==t)); /*重复执行 i++操作*/
            do j--;
            while(!(k[s]<=k[j] || j==s)); /*重复执行 j--操作*/
            if(i<j)
                swap(&k[i],&k[j]); /*交换 k[i]和 k[j]的位置*/
            else
                break;
        }
    }
}

```



```
    }  
    swap(&k[s], &k[j]);          /*交换基准元素与 k[j] 的位置*/  
    quicksort(k, s, j-1);        /*递归排序基准元素前面的子序列*/  
    quicksort(k, j+1, t);        /*递归排序基准元素后面的子序列*/  
}  
}  
  
main()  
{  
    int k[10]={2,5,6,3,7,8,0,9,12,1}, i;  
    printf("The original data array is\n");  
    for(i=0; i<10; i++)          /*显示原序列之中的元素*/  
        printf("%d ", k[i]);  
    quicksort(k, 0, 9);          /*快速排序*/  
    printf("\nThe result of quick sorting for the array is\n");  
    for(i=0; i<10; i++)          /*显示排序后的结果*/  
        printf("%d ", k[i]);  
    getch();  
}
```

本程序的运行结果如图 2-20 所示。



```
The original data array is  
2 5 6 3 7 8 0 9 12 1  
The result of quick sorting for the array is  
12 9 8 7 6 5 3 2 1 0
```

图 2-20 例 2-7 的运行结果



## 第3章 常用的算法思想

对于计算机科学而言,算法(Algorithm)是一个非常重要的概念。它是程序设计的灵魂,它是将实际问题同解决该问题的计算机程序建立起联系的桥梁。可以这样讲,在编写任何一个计算机程序时(无论使用什么编程语言),都不可回避地要进行算法的设计。本章将重点介绍算法的基本概念,以及一些常用的算法思想。

### 3.1 什么是算法

一个程序往往要包含两个方面的描述:一是对数据组织的描述;一是对程序操作流程的描述。对数据组织的描述主要是指数据的类型和数据的组织形式(例如数组),称做数据结构(Data Structure),在3.2节将会讲到。对程序操作流程的描述就是程序的操作步骤,也本节所要介绍的所谓算法(algorithm)。正如Nikiklaus Wirth提出的公式:

数据结构+算法=程序

一样,算法是一个程序中不可缺少的一部分。如果把一个可运行的程序比喻成一个具有生命的人,那么数据结构就是这个人的躯体,而算法则是这个人的灵魂或者精神。

所谓算法,广义地讲就是解决问题的方法和过程。例如洗衣服的过程就可描述为以下几步:

- (1) 用盆接足量的清水;
- (2) 将要洗的衣物浸入水中;
- (3) 放入洗衣粉进行清洗;
- (4) 用清水漂洗;
- (5) 拧干衣物进行晾晒。

那么上述5步就可以叫做完成洗衣服这项工作的算法。

在计算机领域,算法更为严格的定义是若干条指令组成的有穷序列,如果它满足以下几条性质。

- (1) 输入:有零个或多个又外部提供的值作为算法的输入。
  - (2) 输出:产生至少一个量作为输出。
  - (3) 确定性:组成算法的每条指令确定无二义。
  - (4) 有限性:算法中每条指令执行的次数是有限的,每条指令的执行时间也是有限的。
- 以下讨论的算法都是指这种在计算机领域用于解决计算机要处理的问题的算法。

下面从几个方面对算法进行讨论。

## 3.2 算法的分类表示及测评

“算法”只是一个笼统的叫法。在计算机科学领域，将解决不同性质问题的算法划分成为不同的类型。另外，算法本身也存在优劣之分，因此必须有一套完整的评价体系去测评一个算法的性能，从而给出算法一个客观的评价。本节将对算法的分类、表示以及算法的测评简要的介绍。

### 3.2.1 算法的分类

计算机算法可分为两大类，一类叫做数值算法，它主要是解决一些工程上的数值计算问题，例如数值积分、数值求解微分方程等，有一门叫做《数值分析》或《计算方法》的课程是专门来讨论这种数值算法的。还有一类叫做非数值算法，它主要用于解决那些非数值的计算机问题。

### 3.2.2 算法的表示

宽泛地讲，不管用哪种形式描述一个解题过程，只要它逻辑清晰，结果正确，哪怕就是在脑子里构思的算法也是好的算法。但是在解决实际问题时，问题往往比较复杂，并不是可以直接用大脑就想得清楚的，因此就需要用一种简单、清晰的“描述语言”来辅助大脑构建解题过程。这也就是算法的表示形式。

算法的表示形式很多。有以下几种表示形式可供参考。

#### 1. 用自然语言描述

如同上面描述的洗衣服的过程那样就是一个用自然语言描述的算法。除了很简单的问题，一般不用自然语言表示算法，因为自然语言存在二义性。

#### 2. 用流程图表示

这是一种使用最为普遍的算法表示方法，其主要优点是简单直观，便于理解。如图 3-1 所示给出了流程图的图元表示方法。

下面通过一个例子来理解流程图表示的算法。

**【实例 3-1】**用流程图描述判断一个数  $i$  是否为素数的算法。

如图 3-2 所示为“判断一个数  $i$  是否为素数”算法的流程图。

#### 3. 用 NS 流程图表示

1973 年美国学者提出了一种新型流程图：N-S 流程图。N-S 流程图表示方法如图 3-3

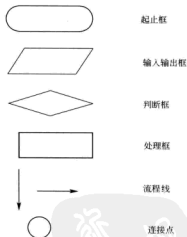


图 3-1 流程图的图元表示方法

表示。

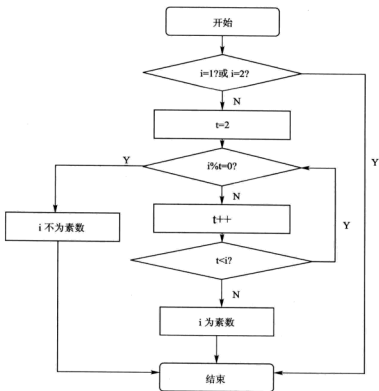


图 3-2 流程图的算法描述

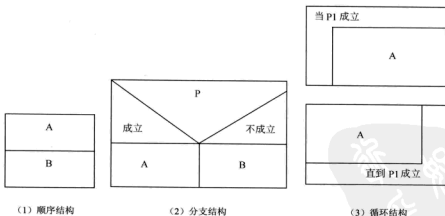


图 3-3 NS 流程图的表示方法

#### 4. 用伪代码表示算法

所谓伪码 (pseudocode) 是帮助程序员制定算法的智能化信息语言。伪代码使用介于

自然语言和计算机语言之间的文字和符号来描述算法。伪代码程序并不能在计算机上运行，它只作为程序员设计程序之前的一种辅助工具。因此伪代码并没有固定的语法和格式，常根据程序员的习惯而定，随意性很大。

**【实例 3-2】**用伪代码描述判断一个数  $i$  是否为素数的算法。

```

If i=1 or i=2
Then i 为素数，程序结束
Else
t←2
Repeat:
    If i mod t =0
        Then i 不为素数，程序结束
    Else t←t+1
Until t>=i
i 为素数，程序结束

```

以上这段描述就是判断一个数  $i$  是否为素数的伪代码算法描述。可以看出，伪代码描述较流程图算法描述更加接近程序代码形式，因此也更加容易转换为实际的程序。

### 3.2.3 算法性能的测评

既然算法是解决实际问题的方法，那么算法就必然存在着好坏优劣之分。一个算法的好坏是有指标能够加以测评的。这个指标通常称为算法的复杂度。

算法的复杂度体现在运行该算法时所需要的系统资源开销上。如果计算机执行一个算法时所需要的系统资源开销很大，就说这个算法复杂度很高。相反，如果计算机执行一个算法时所需要的系统资源开销很小，就说这个算法复杂度很低。在设计算法时自然是希望算法的复杂度越低越好。

由于计算机最重要的资源就是时间和空间资源，因此，算法的复杂度分为时间复杂度和空间复杂度。要想更深一步地探讨算法的复杂度问题，可以参看《算法分析》等书籍。

## 3.3 穷举法思想

穷举法 (Exhaustive Attack method)，又称为强力法 (Brute-force method)，它是一种最为直接，实现最为简单，同时又最为耗时的一种解决实际问题的算法思想。本节将详细介绍穷举法的算法思想。

### 3.3.1 基本概念

穷举法算法的基本思想是：在可能的解空间中穷举出每一种可能的解，并对每一个可能解进行判断，从中得到问题的答案。

使用穷举法思想实际问题，最关键的步骤是划定问题的解空间，并在该解空间中一一枚举每一个可能的解。这里有两点需要注意。一是解空间的划定必须保证覆盖问题的全部解。如果解空间集合用  $H$  表示，问题的解集用  $h$  表示，那么只有当  $h \subset H$  时，才能使用穷举法求解。二是解空间集合及问题的解集一定是离散的集合，也

就是说集合中的元素是可列的、有限的。

穷举法用时间上的牺牲换来了解的全面性保证，因此穷举法的优势在于确保得到问题的全部解，而瓶颈在于运算效率十分低下。但是穷举法算法思想简单，易于实现，在解决一些规模不是很大的问题上，使用穷举法不失为一种很好的选择。另外，随着计算机硬件性能的不断改善，CPU 运算速度的不断提高，以及多处理器并行计算技术的发展，穷举法的形象已经不再是最低等和原始的无奈之举，它将越来越为人们所重视。

下面通过具体的实例来理解穷举法思想。

### 3.3.2 寻找给定区间的素数

**【实例 3-3】**寻找[1, 100]之间的素数。

**【分析】**

解决这个问题最简便的方法就是使用穷举法。在[1, 100]中对每一个整数进行判断，看它是不是素数。在这里，问题的解空间自然就是[1, 100]中的全部整数，因为不会有任一个解超出这个范围，同时该解空间构成的集合元素是可列有限的。算法描述如下：

```
for(i=1;i<=100;i++)
    if (i是素数) 输出 i ;
```

判断一个数是否是素数的算法描述在 3.2 节中已详细介绍，这里不再赘述。

程序清单 3-1

```
/*----- 3-1.c -----*/
#include "stdio.h"
int isPrime(int n)
{
    /*判断 n 是否是素数，是则返回 1，不是则返回 0*/
    int i;
    for(i=2;i<n;i++)
    {
        if(n%i==0) return 0;
    }
    return 1;
}

getPrime(int low,int high)
{
    /*寻找[low,high]之间的素数*/
    int i;
    for(i=low;i<=high;i++)
        if(isPrime(i))
            printf("%d ",i);
}

main()
{
    int low,high;
    printf("Please input the domain for searching prime\n");
    printf("low limitation:");
    scanf("%d",&low);
    printf("high limitation:");
    scanf("%d",&high);
```

```

printf("The whole primes in this domain are\n");
getPrime(low,high);
getche();
}

```

本程序的运行结果如图 3-4 所示。

```

Please input the domain for searching prime
low limitation:1
high limitation:100
The whole primes in this domain are
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

```

图 3-4 例 3-3 的运行结果

### 3.3.3 TOM 的借书方案

**【实例 3-4】**TOM 共有 5 本新书，要借给 A、B、C 3 位同学，每人只能借 1 本书，则 TOM 可以有多少种不同的借书方法。

**【分析】**

这个问题仍然可以用穷举法轻松地解决。假设 TOM 的 5 本书编号为 {1, 2, 3, 4, 5}，每个同学可能借到的书的范围就限定在 {1, 2, 3, 4, 5} 之中。因此 TOM 借书给 3 位同学的组合方案不可能超过  $5^3=125$  种。由这 125 种借书方案构成的解空间可描述为  $\{(x_1, x_2, x_3) | 1 \leq x_i \leq 5, \text{ 且 } x_i \in \mathbb{R}\}$ ，该解空间是由 3 个各包含 5 个元素（图书编号）的集合排列组合而成。应用穷举法在该空间中搜索答案。

当然问题的解不可能就是这 125 种借书方案，因为在这 125 种借书方案中包含着不符合实际要求的排列组合。由于 1 本书一次只能借给 1 位同学，因此在每一种借书方案中，元素有重复的排列组合一定不会是问题的答案。于是可以得出 TOM 借书方案的解集为： $\{(x_1, x_2, x_3) | 1 \leq x_i \leq 5, \text{ 且 } x_i \in \mathbb{R}, x_1 \neq x_2 \neq x_3\}$ 。这个解集可通过穷举解空间集合中的每一个元素，并加判断条件  $x_1 \neq x_2 \neq x_3$  获得。

综合上述，该问题算法描述如下：

```

for (i=1; i<=5; i++)
    for (j=1; j<=5; j++)
        for (k=1; k<=5; k++)
            if (i!=j && j!=k && i!=k) 得到一种借书方案;

```

该算法中，变量  $i$ 、 $j$ 、 $k$  表示图书的编号。

程序清单 3-2

```

/*----- 3-2.c -----*/
#include "stdio.h"

main()
{
    int i, j, k;
    printf("There are different methods for TOM to distribute his book to A, B, C\n");
    for (i=1; i<=5; i++)
        for (j=1; j<=5; j++)

```

```

for(k=1;k<=5;k++)
    if(i!=j && j!=k && i!=k){
        printf("(%d,%d,%d) ",i,j,k);
    }
getche();
}

```

本程序的运行结果如图 3-5 所示。

```

There are different methods for TOM to distribute his book to n.B.C
<1,2,3> <1,2,4> <1,2,5> <1,3,2> <1,3,4> <1,3,5> <1,4,2> <1,4,3> <1,4,5> <1,5,2>
<1,5,3> <1,5,4> <2,1,3> <2,1,4> <2,1,5> <2,3,1> <2,3,4> <2,3,5> <2,4,1> <2,4,3>
<2,4,5> <2,5,1> <2,5,3> <2,5,4> <3,1,2> <3,1,4> <3,1,5> <3,2,1> <3,2,4> <3,2,5>
<3,4,1> <3,4,2> <3,4,5> <3,5,1> <3,5,2> <3,5,4> <4,1,2> <4,1,3> <4,1,5> <4,2,1>
<4,2,3> <4,2,5> <4,3,1> <4,3,2> <4,3,5> <4,5,1> <4,5,2> <4,5,3> <5,1,2> <5,1,3>
<5,1,4> <5,2,1> <5,2,3> <5,2,4> <5,3,1> <5,3,2> <5,3,4> <5,4,1> <5,4,2> <5,4,3>

```

图 3-5 例 3-4 的运行结果

## 3.4 递归与分治思想

递归与分治的算法思想往往是相伴而生的，它们在各类算法中使用非常频繁，应用递归和分治的算法思想有时可以设计出代码简洁且比较高效的算法来。本章将详细介绍递归与分治的算法思想。

### 3.4.1 基本概念

在解决一些比较复杂的问题，特别是解决一些规模较大的问题时，常常将问题进行分解。具体来说，就是将一个规模较大的问题分割成规模较小的同类问题，然后将这些小的子问题逐个加以解决，最终也就将整个大的问题解决了。这种分而治之的思想称做分治的思想。在解决一些问题比较复杂、计算量庞大的问题时经常被用到。

一个最为经典的使用分治思想设计的算法就是第 2 章中介绍过的“折半查找算法”。折半查找算法利用了元素之间的顺序关系（有序序列），采用分而治之的策略，不断缩小问题的规模，每次都把问题的规模减小至上一次的 1/2。采用顺序查找的方法对关键字进行搜索的时间复杂度为  $O(n)$ ，而采用折半查找方法的时间复杂度仅为  $O(\log_2 n)$ 。

递归的思想也是一种常见的算法设计思想。所谓递归算法，就是一种直接或间接地调用原算法本身的一种算法。可以通过一个具体的例子来理解递归的算法思想。

**【实例 3-5】** 计算  $n$  的阶乘  $n!$ 。

这是一个再简单不过的问题了。只要学过程序设计语言的人都可以很容易地编程解决此题。一般采用循环累乘的算法求解，算法如下：

```

int factorial(n)
{
    int i, res = 1;
    for(i=1;i<=n;i++)
        res = res * i;
    return res;
}

```



其实阶乘的数学定义可以用递归函数来简单地描述:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

这样的函数称为递归函数, 因为该函数本身直接或间接地调用了该函数本身。基于阶乘的递归函数的描述, 就不难设计出计算  $n$  的阶乘  $n!$  的递归算法。

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

可以看出, 使用递归算法解决阶乘问题形式上更加简洁, 更易于人们理解。

在设计递归算法时要注意以下几点。

(1) 每个递归函数都必须有一个非递归定义的初始值, 作为递归结束标志, 或递归结束的出口。就像实例 3-5 所描述的递归算法中的 `if(n==0) return 1;`。如果一个递归算法中没有这个非递归定义的初始值, 那么该递归调用是无法计算出具体值的(或无法得到结果), 同时该递归调用也无法结束。

(2) 在设计递归算法时, 要解决的问题需具有递归性。例如要计算  $n$  的阶乘  $n!$ ,  $n!$  的定义本身具有递归性。这种所谓的递归性实际上就是一种反复调用自身过程的特性。

(3) 虽然采用递归算法解决问题, 特别是一些复杂问题, 更加方便且容易实现, 但是递归方法的运行较低, 时间和空间复杂度都比较高, 因此对于一些对时间和空间要求较高的程序, 建议使用非递归算法设计。

在实际的算法设计中, 递归与分治如同一对兄弟, 经常结合在一起使用。这是因为, 由分治的方法产生的子问题往往都是原问题的更小规模。反复使用分治的手段, 可使子问题与原问题类型一致, 但规模不断缩小, 最终使子问题比较容易求解。既然子问题与原问题的类型一致, 这就具有了所谓的递归性, 因此可以使用递归的方法去解决原问题的算法去解决同类型的子问题。在第 2 章中介绍的折半查找算法只是单纯地使用了分治的策略, 在下面的实例分析中将使用递归与分治思想相结合的方法进行折半查找算法的设计。

### 3.4.2 计算整数的划分数

**【实例 3-6】** 将一个正整数  $n$  表示成一系列的正整数之和:

$$n = n_1 + n_2 + \cdots + n_k \quad (n_1 \geq n_2 \geq \cdots \geq n_k \geq 1, k \geq 1)$$

被称做正整数  $n$  的一个划分。一个正整数  $n$  可能存在着不同的划分, 例如正整数 6 的全部的划分为:

6=6

6=5+1

6=4+2    6=4+1+1

6=3+3    6=3+2+1    6=3+1+1+1

6=2+2+2    6=2+2+1+1    6=2+1+1+1+1

6=1+1+1+1+1+1

## 【分析】

正整数  $n$  的不同的划分的个数称为该正整数  $n$  的划分数。例如正整数 6 的划分数为 11。  
编写一个程序，计算输入的正整数  $n$  的划分数。

根据正整数划分的定义，可以总结出以下规律：

设标记  $P(n, m)$  表示正整数  $n$  的所有不同划分中，最大加数不大于  $m$  的划分个数。例如  $P(6, 2) = 4$ ，因为在正整数 6 的全部划分中，最大加数不大于 2 的只有：

$$6=2+2+2 \quad 6=2+2+1+1 \quad 6=2+1+1+1+1$$

$$6=1+1+1+1+1+1$$

这 4 个划分。

可以建立起如下递归关系：

$$(1) P(n, 1) = 1, n \geq 1;$$

这是因为任何正整数  $n$  的划分中，加数不大于 1 的划分有且仅有 1 种，即  $n = \overbrace{1+1+\cdots+1}^n$ 。

$$(2) P(n, m) = P(n, n), m \geq n;$$

这是因为任何正整数  $n$  的划分中，只存在一种划分即  $n=n$ ，其最大加数等于  $n$ 。不存在最大加数大于  $n$  的情况。（根据划分的定义，任何加数必须大于 1）。

$$(3) P(n, n) = P(n, n-1) + 1;$$

因为最大加数为  $n$  的划分只有 1 种，即  $n=n$ 。这里要注意，“ $n$  的最大加数为  $n$  的划分数”不同于“ $n$  的最大加数不大于  $n$  的划分数”，即  $P(n, n)$ ，如图 3-6 所示。

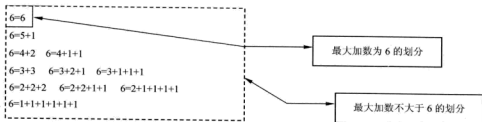


图 3-6 最大加数为 6 的划分以及最大加数不大于 6 的划分

因此最大加数不大于  $n$  的划分数  $P(n, n)$  等于最大加数不大于  $n-1$  的划分数  $P(n, n-1)$  与最大加数为  $n$  的划分数 1 之和。

$$(4) P(n, m) = P(n, m-1) + P(n-m, m), n > m > 1;$$

“正整数  $n$  的最大加数不大于  $m$  的划分数 ( $n > m > 1$ )”等于“ $n$  的最大加数不大于  $m-1$  的划分数  $P(n, m-1)$ ”与“ $n$  的最大加数为  $m$  的划分数”之和，如图 3-7 所示。

如图 3-7 所示，“正整数 6 的最大加数不大于 4 的划分数  $P(6, 4)$ ”等于“6 的最大加数不大于 3 的划分数  $P(6, 3)$ ”与“6 的最大加数为 4 的划分数”之和。而“6 的最大加数为 4 的划分数”为 2，其实它等于“6-4=2 的最大加数不大于 4 的划分数”，即等于  $P(6-4, 4) = P(2, 4) = 2$ 。

有些读者可能会提出问题来，可不可以说  $n$  的最大加数为  $m$  的划分数等于  $P(n-m, n-m)$  呢？因为  $P(6-4, 4) = P(2, 4) = P(2, 2)$ 。为什么这里要写成  $P(n-m, m)$ ，而不能写成  $P(n-m, n-m)$  呢？答案是否定的。因为  $P(6-4, 4) = P(2, 4) = P(2, 2)$  只是一个特例。如果求 6 的最大加数为 2 的划分数，那么它等于  $P(6-2, 2) = P(4, 2) = 3$  而不等于  $P(6-2, 6-2) = P(4, 4) = 5$ ，这一点应当注意。因此可以得出结论： $P(n, m) = P(n, m-1) + P(n-m, m), n > m > 1$ 。

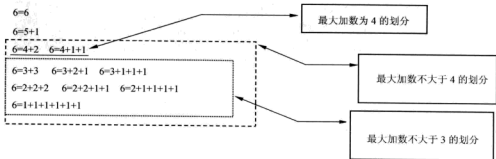


图 3-7 6 的最大加数不大于 4 的划分组成

根据以上归纳出的递归关系，可以得出计算  $P(n, m)$  的递归函数式：

$$P(n, m) = \begin{cases} 1 & m = 1 \\ P(n, n) & n < m \\ 1 + P(n, n-1) & n = m \\ P(n, m-1) + P(n-m, m) & n > m > 1 \end{cases}$$

很显然，这是递归函数。下面可以由此递归函数设计出求解正整数  $n$  的划分数值的递归算法。其实正整数  $n$  的划分数即为  $P(n, n)$ 。

```
int P(int n, int m)
{
    if (m == 1) return 1;
    if (m > n) return P(n, n);
    if (m == n) return 1 + P(n, m-1);
    return P(n, m-1) + P(n-m, m);
}
```

下面给出完整的程序清单。

程序清单 3-3

```
/*----- 3-3.c -----*/
#include "stdio.h"
int P(int n, int m)
{
    if (m == 1 || n == 1) return 1;
    if (m > n) return P(n, n);
    if (m == n) return 1 + P(n, m-1);
    return P(n, m-1) + P(n-m, m);
}

main()
{
    int n, s;
    printf("Please input a integer for getting the number of division\n");
    scanf("%d", &n);          /*输入正整数 n*/
    s = P(n, n);              /*求出正整数 n 的划分数*/
    printf("The number of division of %d is %d\n", n, s);
    getch();
}
```

本程序的运行结果如图 3-8 所示。

```
Please input a integer for getting the number of division
6
The number of division of 6 is 11
```

图 3-8 例 3-6 的运行结果

### 3.4.3 递归的折半查找算法

**【实例 3-7】**有一个数组 A[10]，里面存放了 10 个整数，顺序递增。

A[10] = {2,3,5,7,8,10,12,15,19,21}

任意输入一个用数字  $n$ ，用折半查找法找到  $n$  位于数组中的位置。如果  $n$  不属于数组 A，显示错误提示。要求用递归的方法实现折半查找。

**【分析】**

在第 2 章中曾详细地介绍过折半查找的算法，它是一种针对有序序列（或文件记录）的高效的查找算法。折半查找的基本思想是：减小查找序列的长度。它的查找过程是：先确定待查找记录的所在的范围，然后逐渐缩小查找的范围，直至找到该记录为止（也可能查找失败）。因此它也是一种基于分治的算法思想设计出来的查找算法。折半查找的算法如下：

```
int bin_search(keytype key[],int n,keytype k)
{
    int low = 0, high = n-1, mid;
    while(low<=high){
        mid = (low+high)/2;
        if(key[mid] == k)
            return mid;          /*查找成功，返回mid*/
        if(k > key[mid])
            low = mid + 1;        /*在后半序列中查找*/
        else
            high = mid - 1;       /*在前半序列中查找*/
    }
    return -1;                   /*查找失败，返回-1*/
}
```

从上述的算法中不难看出，折半查找的过程实际上也是一个递归的过程。因为在折半查找的过程中，每次都都将问题的规模减小至原问题的一半，而缩小后的子问题与原问题类型保持一致。因此，解决缩小后的子问题的方法与解决原问题的方法是一样的。这就说明折半查找的思想具有递归的性质。因而比较容易写出折半查找的递归算法。

```
int bin_search(keytype key[],int low, int high,keytype k)
{
    int mid;
    if(low>high)
        return -1;
    else{
        mid = (low+high) / 2;
        if(key[mid]==k)
            return mid;
        if(k>key[mid])
            return bin_search(key, mid+1, high, k);
        else
            return bin_search(key, low, mid-1, k);
    }
}
```

```

        return bin_search(key,mid+1,high,k);    /*在序列的后半部分查找*/
    else
        return bin_search(key,low,mid-1,k);    /*在序列的前半部分查找*/
}

```

这里要注意一点，只有原序列中的元素是以递增不减的顺序排列时，才能使用上述的算法进行元素的查找。使用该算法进行折半查找，若查找成功，返回元素记录在原序列中的位置；若查找不成功，返回-1。

下面给出完整的程序清单。

程序清单 3-4


```

/*----- 3-4.c -----*/
#include "stdio.h"
int bin_search(int key[],int low, int high,int k)
{
    int mid;
    if(low>high)
        return -1;
    else{
        mid = (low+high) / 2;
        if(key[mid]==k)
            return mid;
        if(k>key[mid])
            return bin_search(key,mid+1,high,k);    /*在序列的后半部分查找*/
        else
            return bin_search(key,low,mid-1,k);    /*在序列的前半部分查找*/
    }
}

main()
{
    int n , i , addr;
    int A[10] = {2,3,5,7,8,10,12,15,19,21};    /*初始化数组序列 A*/
    printf("The contents of the Array A[10] are\n");
    for(i=0;i<10;i++)
        printf("%d ",A[i]);    /*显示数组 A 中的内容*/
    printf("\nPlease input a interger for search\n");
    scanf("%d",&n);    /*输入待查找的元素*/
    addr = bin_search(A,0,9,n);
    if(-1 != addr)    /*查找成功*/
        printf("%d is at the %dth unit is array A\n",n,addr);
    else printf("There is no %d in array A\n",n);    /*查找失败*/
    getch();
}

```

本程序的运行结果如图 3-9 所示。

 **注意：**在这里数组的下标是从 0 开始的，因此 12 位于数组 A 的第 6 单元。

```

The contents of the Array A[10] are
2 3 5 7 8 10 12 15 19 21
Please input a interger for search
12
12 is at the 6th unit is array A

```

图 3-9 例 3-7 的运行结果

## 3.5 贪心算法思想

贪心算法的思想非常简单而且算法效率很高,在一些问题的解决上有着明显的优势。本章将详细介绍贪心算法的基本思想。

### 3.5.1 基本概念

先看一个找零钱的问题。假设有3种硬币,面值分别为1元、5角、1角。这3种硬币各自的数量不限,现在要找给顾客2元7角钱,请问怎样找钱才能使得找给顾客的硬币数量最少呢?你也许会不假思索地说出答案:找给顾客2枚1元硬币,1枚5角硬币,2枚1角硬币。其实找给顾客5枚5角硬币,2枚1角硬币也能凑够2元7角,但是这种找零钱的方法硬币数量较多,不符合题目的要求。在这里,我们下意识地应用了所谓贪心算法解决找零钱的问题。

所谓贪心算法,就是总是做出在当前看来是最好的选择的一种方法。以上述的找零钱为例,为了找给顾客的硬币数量最少,在选择硬币的面值时,当然是尽可能地选择面值大的硬币。因此要找给顾客2元7角钱,下意识地遵循了以下方案。

- (1) 首先找出一个面值不超过2元7角的最大硬币,即1元硬币。
- (2) 然后从2元7角中减去1元,得到1元7角,再找出一个面值不超过1元7角的最大硬币,即1元硬币。
- (3) 然后从1元7角减去1元,得到7角,再找出一个面值不超过7角的最大硬币,即5角硬币。
- (4) 然后从7角中减去5角,得到2角,再找出一个面值不超过2角的最大硬币,即1角硬币。
- (5) 然后从2角中减去1角,得到1角,再找出一个面值不超过1角的最大硬币,即1角硬币。
- (6) 找零钱过程结束。

这个找钱过程实际应用的就是一种典型的贪心算法思想。

因此不难看出,应用贪心算法求解问题,并不从问题的整体最优上加以考虑,它所作出的每一步选择只是在某种意义上的局部最优选择。因此严格意义上讲,要使用贪心算法求解问题,该问题应当具备以下性质。

#### (1) 贪心选择性质

所谓贪心选择性质就是指所求解的问题的整体最优解可以通过一系列的局部最优解得到。所谓局部最优解,就是指在当前的状态下做出的最好选择。例如上述的找零钱问题,当前状态下的最好选择就是使找过硬币后剩余的(应当找给顾客的)零钱数最接近于0。因此每次找钱都要找面值尽可能大的硬币。

#### (2) 最优子结构性

当一个问题的最优解包含着它的子问题的最优解时,就称此问题具有最优子结构性

质。找零钱问题本身就具有最优子结构性质。

但是,要判断一个问题是否具备以上两种性质,也就是说判断一个问题是否可以通过贪心算法得到最优解,是一件比较困难事情。这里需要比较复杂而严格的数学证明。因此虽然贪心算法简单易实现,并且效率很高,但是使用贪心算法之前必须对问题本身进行深入而透彻地分析和证明,以保证使用贪心算法得到最优解。

其实,虽然贪心算法不是对所有问题都能得到整体的最优解,但是实际应用中的许多问题都可以使用贪心算法得到最优解。与此同时,即使使用贪心算法不能产生出问题的最优解,但最终结果也是最优解的很好的近似解。因此在解决一般性问题时(并不一定要得到最优解),我们大可不必过分要求使用贪心算法一定要得到最优解,也没有必要进行严格地推理证明,使用贪心算法是一种不错的选择。

我们经常使用的哈夫曼(Huffman Tree)编码算法,求解最小生成树的克鲁斯卡尔(Kruskal)算法和普里姆(Prim)算法,求解图的单源最短路径的迪克斯特拉(Dijkstra)算法等都是基于贪心算法的思想设计的。

### 3.5.2 最优装船问题

**【实例 3-8】**有一批集装箱要装入一个载质量为  $C$  的货船中,每个集装箱的质量由用户自己输入指定,在货船的装载体积无限的前提下,如何装载集装箱才能尽可能多地将集装箱装入货船中。

#### 【分析】

这个问题可以用贪心算法求得最优解。只要每次装船时,采取质量最轻的集装箱先装船的策略,就可以得到最优装船问题的一个最优解。

在设计算法时,可用一个数组  $w[]$  存放每个集装箱的质量,例如  $w[2]=5$ ,就表示第 2 个集装箱的质量为 5。再设置一个向量数组  $x[]$  存放集装箱的取舍状态。其中,  $x[i]=1$  表示将第  $i$  个集装箱放入船中,  $x[i]=0$  表示第  $i$  个集装箱不装入货船。最终输出的结果应当是向量数组  $x[]$  的下标,如果输出结果为  $\{0, 1, 4, 5\}$  表示将第 0, 1, 4, 5 个集装箱装入货船中。也就是输出数组  $x[]$  中所有  $x[i]=1$  的下标  $i$ 。

应用贪心算法求解,每次都要选出数组  $w[]$  中当前质量最轻的(即  $w[i]$  值最小的)集装箱,并将  $x[i]$  置 1,同时  $c=c-w[i]$ ,即变量  $c$  中存放货船的剩余载质量,  $c$  的初始值为货船载质量  $C$ 。循环执行上述操作,直到  $w[i]>c$ ,表明货船已达到最大载质量,或者集装箱全部装完为止。然后输出数组  $x[]$  中所有  $x[i]=1$  的下标  $i$ 。可用下面这段代码描述这个算法过程。

```
Loading(int x[],int w[],int c,int n)
{
    int i, s = 0;
    int *t = (int *)malloc(sizeof(int) * n);
    /*动态开辟一个临时数组,存放 w[] 的下标,如果 t[i],t[j], i<j, 则 w[i]≤w[j]*/
    sort(w,t,n);
    /*排序,用数组 t[] 存放 w[] 的下标*/
    for(i = 0;i<n;i++)
        x[i] = 0;
    /*初始化数组 x[]*/
    for(i=0;i<n && w[t[i]]<=c; i++){
        x[t[i]] = 1;
        /*将第 t[i] 个集装箱装入货船中*/
        c = c - w[t[i]];
        /*变量 c 中存放货船的剩余载质量*/
    }
}
```

上述算法中,用一个临时数组t来存放数组w[]的下标,使得如果 $i < j$ ,则 $w[t[i]] \leq w[t[j]]$ 。例如 $w[3]=\{5,3,2\}$ ,那么 $t[3]=\{2,1,0\}$ ,它表明集装箱 $w[t[0]]$ 即 $w[2]$ 的质量小于集装箱的 $w[t[1]]$ ,即 $w[1]$ 的质量小于集装箱 $w[t[0]]$ 即 $w[0]$ 的质量。这个工作由函数sort来完成。函数sort的算法描述如下:

```
sort(int w[], int t[], int n)
{
    int i,j;
    int w_tmp= (int *)malloc(sizeof(int) * n);
                                /*动态开辟一个临时数组,存放w[]中的内容,用于排序*/
    for(i=0;i<n;i++)
        t[i] = i;                /*初始化数组t*/
    for(i=0;i<n;i++)
        w_tmp[i] = w[i];        /*复制数组w[]到w_tmp[]*/
    for(i=0;i<n-1;i++)
        for(j=0;j<n-i-1;j++) /*冒泡排序实现*/
            if(w_tmp[j]>w_tmp[j+1])
            {
                tmp = w_tmp[j];
                w_tmp[j] = w_tmp[j+1];
                w_tmp[j+1] = tmp;
                tmp = t[j];
                t[j] = t[j+1];
                t[j+1] = tmp;
            }
}
```

下面给出完整的程序清单。

程序清单 3-5

```
/*----- 3-5.c -----*/
#include "stdio.h"

void sort(int w[], int t[], int n)
{
    int i,j,tmp;
    int *w_tmp= (int *)malloc(sizeof(int) * n);
                                /*动态开辟一个临时数组,存放w[]中的内容,用于排序*/
    for(i=0;i<n;i++)
        t[i] = i;                /*初始化数组t*/
    for(i=0;i<n;i++)
        w_tmp[i] = w[i];
    for(i=0;i<n-1;i++)
        for(j=0;j<n-i-1;j++) /*冒泡排序实现*/
            if(w_tmp[j]>w_tmp[j+1])
            {
                tmp = w_tmp[j];
                w_tmp[j] = w_tmp[j+1];
                w_tmp[j+1] = tmp;
                tmp = t[j];
                t[j] = t[j+1];
                t[j+1] = tmp;
            }
}
```



```

}

void Loading(int x[],int w[],int c,int n)
{
    int i, s = 0;
    int *t = (int *)malloc(sizeof(int) * n);
    /*动态开辟一个临时数组,存放w[]的下标,如果t[i]、t[j]、i<j,则w[i]≤w[j]*/
    sort(w,t,n); /*排序,用数组t[]存放w[]的下标*/
    for(i = 0;i<n;i++)
        x[i] = 0; /*初始化数组x[]*/
    for(i=0;i<n && w[t[i]]<=c; i++){
        x[t[i]] = 1; /*将第t[i]个集装箱装入货船中*/
        c = c - w[t[i]]; /*变量c中存放货船的剩余载质量*/
    }
}

main()
{
    int x[5],w[5],c,i;
    printf("Please input the maximum loading of the sheep\n");
    scanf("%d",&c); /*输入货船的最大载质量*/
    printf("Please input the weight of FIVE box\n");
    for(i=0;i<5;i++) /*输入每个集装箱的质量*/
        scanf("%d",&w[i]);
    Loading(x,w,c,5); /*进行最优装载*/
    printf("The following boxes will be loaded\n");
    for(i=0;i<5;i++) /*输出结果*/
    {
        if(x[i] == 1) printf("BOX:%d ",i) ;
    }
    getch();
}

```

本程序的运行结果如图 3-10 所示。

**注意：**本次运行程序，设定船的最大载质量为 13，5 个集装箱分别重 {5, 7, 6, 3, 2}，得到的最佳装载方案是：装载集装箱 0，质量为 5；装载集装箱 3，质量为 3；装载集装箱 4，质量为 2。总的装载质量为 10，小于最大载质量 13，装载的集装箱数为 3 个。

```

Please input the maximum loading of the ship
13
Please input the weight of FIVE box
5 7 6 3 2
The following boxes will be loaded
BOX:0 BOX:3 BOX:4

```

图 3-10 例 3-8 的运行结果

### 3.6 回溯法

回溯法是一种非常有效，适用范围相当广泛的算法设计思想。许多复杂的问题，规模较大的问题都可以使用回溯法求解。因此回溯法又有“通用解题方法”的美称。本章将详细介绍回溯法的算法思想。

### 3.6.1 基本概念

回溯法的基本思想是：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去；如果该结点不包含问题的解，那就说明以该结点为根结点的子树一定不包含问题的最终解，因此要跳过对以该结点为根的子树的系统探索，逐层向其祖先结点回溯。这个过程叫做解空间树的“剪枝”操作。

如果应用回溯法求解问题的所有解，要回溯到解空间树的树根，这样根结点的所有子树都被探索到才结束。如果只要求解问题的一个解，那么在探索解空间树时，只要搜索到问题的一个解就可以结束了。

许多复杂的问题都可以用回溯法的思想来求解，例如经典的 N 皇后问题。

N 皇后问题的描述为：求解如何在一个  $N \times N$  的棋盘上无冲突地摆放 N 个皇后棋子。在国际象棋里，皇后的移动方式为横竖交叉的，因此在任意一个皇后所在位置的水平、竖直，以及  $45^\circ$  斜线上都不能出现皇后的棋子，如图 3-11 所示，就是八皇后问题一种解。

N 皇后问题的解法很多，可以用回溯法解决 N 皇后问题。以四皇后问题为例，可以构建出一棵解空间树，通过探索这棵解空间树，可以得到四皇后问题的一种或几种解。这样的解空间树共有 4 棵。如图 3-12 所示为四皇后问题的一种解空间树。

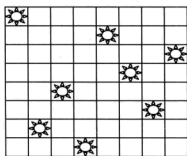


图 3-11 八皇后问题的一种解

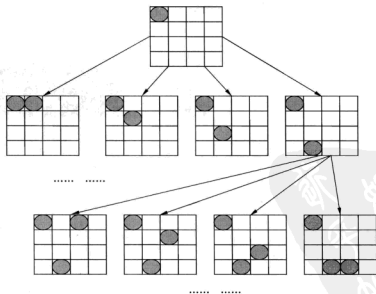


图 3-12 四皇后问题的一种解空间树

由于版面的限制,该解空间树并不完整。该解空间树的根结点为第一个皇后的一种摆法,它还有另外3种摆法,因此一共可以构造出4棵解空间树。通过探索上述这棵解空间树,可以搜索出由根结点这种棋面所产生的所有四皇后的解(如果有解)。

如图3-12所示,根结点派生出4个子结点。每个结点都示意出前两个皇后可能摆放的位置。每个子结点又可派生出4个子结点,每个结点都示意出前3个皇后可能摆放的位置……整个解空间树为一棵四叉的满树,包含85个结点。

应用回溯法求解四皇后问题,从根结点出发,深度优先搜索整个解空间树。当访问到根结点的第一个孩子时,发现该结点不包含问题的解。也就是说,该结点所示的皇后的摆法不符合四皇后问题的要求,于是停止向下探索,回溯到根结点,继续探索根结点的下一个孩子结点。这就是所谓的剪枝操作,这样可以减少搜索的步数,以尽快找到问题的答案。实践证明,如图3-12所示的解空间树中不包含四皇后问题的解。于是需要探索第二棵解空间树,如图3-13所示。

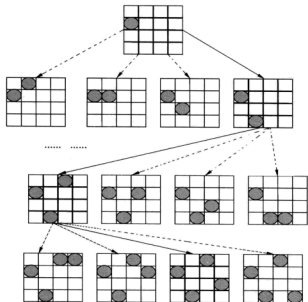


图3-13 四皇后问题的另一种解空间树

按照上述的探索过程深度优先搜索如图3-13所示的解空间树,最终可以搜索出四皇后问题的一个解,它的搜索路径在图中用粗体表示,叶结点为四皇后问题的一种解。图中用虚线描绘的结点之间的连线表示在此执行剪枝操作。

下面通过实例加深对回溯法的理解。

### 3.6.2 四皇后问题求解

**【实例3-9】**应用回溯法的思想求解四皇后问题。

**【分析】**

3.6.1节中已经详细介绍了回溯法解决四皇后问题的基本过程。在这里将给出具体的算

法描述和程序清单。

其实在解决四皇后问题时，并不一定要真的构建出这样一棵解空间树，它完全可以过一个递归回溯来模拟。所谓解空间树只是一个逻辑上的抽象。当然也可以用树结构真实地创建出一棵解空间树，不过那样会比较浪费空间资源。

解决四皇后问题的算法描述如下：

```
void Queen(int j,int (*Q)[4]){
    int i , k;
    if(j==4)
    {
        for(i=0;i<4;i++){
            {
                /*得到了一个解，在屏幕上输出结果*/
                for(k=0;k<4;k++){
                    printf("%d ",Q[i][k]);
                    printf("\n");
                }
                printf("\n");
                getche();
                return;
            }
        }
        for(i=0;i<4;i++){
            {
                if(isCorrect(i,j,Q)) /*如果 Q[i][j] 可以放置皇后*/
                {
                    Q[i][j] = 1; /*放置皇后*/
                    Queen(j+1,Q); /*递归深度优先搜索解空间树*/
                    Q[i][j] = 0;
                }
            }
        }
    }
}
```

在该算法中，用一个二维数组  $Q[4][4]$  存放棋盘布局。 $Q[i][j]=0$  表示不放置皇后， $Q[i][j]=1$  表示放置皇后。在这里采用了递归回溯的方法深度优先搜索解空间树，可以将四皇后问题的全部解找到并输出。函数  $Queen()$  包括两个参数，参数  $j$  为放置的皇后所在棋盘的列数，最开始调用  $Queen()$  时， $j$  的初始值为 0，由它可派生出第一棵解空间树。 $j$  的取值范围是 0~3，对应着 4 棵解空间树。当  $j$  的值等于 4 时，表明已得到了一个四皇后的解，程序返回。参数  $(*Q)[4]$  为指向二维数组每一行的指针。

在该算法中调用了子函数  $isCorrect(i,j,Q)$ ，它的功能是判断棋盘中  $Q[i][j]$  的位置是否可以放置一个皇后。它的算法描述如下：

```
int isCorrect(int i,int j,int (*Q)[4])
{
    int s,t;
    for(s=i,t=0;t<4;t++){
        if(Q[s][t]==1 && t!=j) return 0; /*判断行*/

        for(t=j,s=0;s<4;s++){
            if(Q[s][t]==1 && s!=i) return 0; /*判断列*/

            for(s=i-1,t=j-1;s>=0&&t>=0;s--,t--)
                if(Q[s][t] == 1) return 0; /*判断左上方*/
        }
    }
}
```

```

for(s=i+1,t=j+1;s<4&&t<4;s++,t++)
if(Q[s][t] == 1) return 0;          /*判断右下方*/

for(s=i-1,t=j+1;s>=0&&t<4;s--,t++)
if(Q[s][t] == 1) return 0;          /*判断右上方*/

for(s=i+1,t=j-1;s<4&&t>=0;s++,t--)
if(Q[s][t] == 1) return 0;          /*判断左下方*/

return 1;                            /*否则返回1*/
}

```

该算法的设计思想是，以  $Q[i][j]$  为中心，分别判断二维数组  $Q$  的行、列、左上方、右下方、右上方、左下方的状态。如果存在 1（有皇后棋子），则表明  $Q[i][j]$  的位置不能放置皇后，返回 0；否则可以放置皇后，返回 1，如图 3-14 所示。

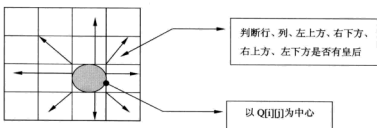


图 3-14 判断  $Q[i][j]$  是否可以放置皇后

下面给出四皇后问题完整的程序清单。

#### 程序清单 3-6

```

/*----- 3-6.c -----*/
#include "stdio.h"
int count=0;                      /*记录四皇后问题解的个数*/
int isCorrect(int i,int j,int (*Q)[4])
{
    int s,t;
    for(s=i,t=0;t<4;t++)
        if(Q[s][t]==1 && t!=j) return 0;    /*判断行*/

    for(t=j,s=0;s<4;s++)
        if(Q[s][t]==1 && s!=i) return 0;    /*判断列*/

    for(s=i-1,t=j-1;s>=0&&t>=0;s--,t--)
        if(Q[s][t] == 1) return 0;          /*判断左上方*/

    for(s=i+1,t=j+1;s<4&&t<4;s++,t++)
        if(Q[s][t] == 1) return 0;          /*判断右下方*/

    for(s=i-1,t=j+1;s>=0&&t<4;s--,t++)
        if(Q[s][t] == 1) return 0;          /*判断右上方*/

    for(s=i+1,t=j-1;s<4&&t>=0;s++,t--)
        if(Q[s][t] == 1) return 0;          /*判断左下方*/
}

```

```

    return 1;                                /*否则返回1*/
}

void Queen(int j,int (*Q)[4]){
    int i , k;
    if(j==4)
    {
        for(i=0;i<4;i++)
        {
            for(k=0;k<4;k++)
                printf("%d ",Q[i][k]);
            printf("\n");
        }
        printf("\n");
        getche();
        count++;
        return;
    }
    for(i=0;i<4;i++)
    {
        if(isCorrect(i,j,Q))
        {
            Q[i][j] = 1;
            Queen(j+1,Q);
            Q[i][j] = 0;
        }
    }
}

main()
{
    int Q[4][4];
    int i,j;
    for( i=0;i<4;i++)
        for( j=0;j<4;j++)
            Q[i][j] = 0;
    Queen(0,Q);
    printf("The number of the answers of FOUR_QUEEN are %d",count);
    getche();
}

```

本程序的运行结果如图 3-15 所示。

```

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

The number of the answers of FOUR_QUEEN are 2

```

图 3-15 例 3-9 的运行结果

由四皇后问题很容易扩展成为八皇后问题或N皇后问题，在此不再详述。

## 3.7 数值概率算法

在解决实际问题时，有时会用到所谓的概率算法。概率算法允许在执行过程中随机地选择下一步的计算步骤，因此使用概率算法有时会大大地提高算法的效率，但有时也可能得不到问题的全部答案。

### 3.7.1 基本概念

概率算法大致分为4类：数值概率算法、蒙特卡洛（Monte Carlo）算法、拉斯维加斯（Las Vegas）算法和舍伍德（Sherwood）算法。这里只介绍最为基础的数值概率算法。

数值概率算法常应用于解决数值计算的问题。应用数值概率算法往往只能得到问题的近似解，并且该近似解的精度一般随着计算时间的增加而不断提高。因为在一些数值问题中，不可能也没有必要计算出问题的精确解（例如：计算无理数 $\pi$ 的取值等）。因此，在解决一些数值计算的问题时，数值概率算法常能派上用场。下面通过实例来认识数值概率算法。

### 3.7.2 计算定积分

【实例3-10】设 $f(x)=1-x^2$ ，计算定积分：

$$I = \int_0^1 f(x) dx \text{ 的值。}$$

分析

函数 $f(x)=1-x^2$ 的在 $[0, 1]$ 上的图像如图3-16所示。

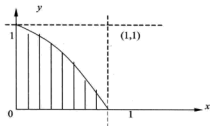


图3-16 函数 $f(x)=1-x^2$ 的在 $[0, 1]$ 上的图像

要计算的定积分值的几何含义就是图中阴影部分的面积。可以试想，如果随机地向图中虚线与 $x, y$ 坐标轴所围成的正方形中投点，那么根据几何概率的知识可知，随机点落入阴影区域的概率即为阴影部分的面积与虚线与 $x, y$ 坐标轴所围成的正方形的面积之比。而由定积分的意义又可知：

$$P_{\text{阴影}} = \frac{S_{\text{阴影}}}{S_{\text{正方形}}} = \frac{\int_0^1 (1-x^2) dx}{\int_0^1 1 dx} = \int_0^1 (1-x^2) dx$$

因此，只要求出随机点落入阴影区域的概率就求出了定积分 $I = \int_0^1 f(x) dx$ 的近似值。

假设向单位正方形中随机投入 $n$ 个点 $(x_i, y_i)$ ， $i=1, 2, 3, \dots, n$ 。随机点 $(x_i, y_i)$ 是否落入阴影区域内，可由 $y_i \leq f(x_i) = 1 - x_i^2$ 来判断。如果有 $m$ 个点落入阴影区域内，则概率 $P \approx m/n$ 。

算法描述如下：

```
double Darts(int n)
{
    double x, y;
```

```

time_t t;
int i, count = 0;
srand((unsigned)time(&t));
for(i=0; i<n; i++)
{
    x = rand()%100/100.0;
    y = rand()%100/100.0;
    if(y<=1 - pow(x,2))
        count++;
}
return (double)count/(double)n;
}

```

本算法中应用函数 `srand()` 和 `rand()` 产生随机数，这样每次产生的随机数都不一样。因为函数 `rand()` 产生的随机数返回值为整数，因此在这里产生 100 以内的随机数 (`rand()%100`)，再将它除以 100.0，得到双精度的  $[0, 1]$  内的随机数点。

程序清单 3-7

```


/*----- 3-7.c -----*/
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"

double Darts(int n)
{
    double x, y;
    time_t t;
    int i, count = 0;
    srand((unsigned)time(&t));
    for(i=0; i<n; i++)
    {
        x = rand()%100/100.0;
        y = rand()%100/100.0;
        if(y<=1 - pow(x,2))
            count++;
    }
    return (double)count/(double)n; /*返回落入阴影区域的点数与总点数 n 的比值*/
}

main()
{
    int n;
    printf("Please input the accuracy\n"); /*输入精度，即投点数*/
    scanf("%d", &n);
    printf("The result is about\n"); /*输出计算结果*/
    printf("%f\n", Darts(n));
    getch();
}

```

本程序的运行结果如图 3-17 所示。

 注意： $I = \int_0^1 (1-x^2)dx$  的计算结果应为  $2/3 \approx 0.667$ 。

```

Please input the accuracy
10000
The result is about
0.672900

```

图 3-17 例 3-10 的运行结果



## 第2部分 编程实例解析

- » 第4章 编程基本功
- » 第5章 数学趣题（一）
- » 第6章 数学趣题（二）
- » 第7章 数据结构趣题
- » 第8章 数值计算问题
- » 第9章 综合题
- » 第10章 算法设计与数据结构面试题精粹

新  
学  
社  
PDG

## 第4章 编程基本功

想要成为一名优秀的程序员，就必须在实践中一点一滴地去积累编程的经验，多编程，敢编程，而且不应急于求成，从最基本的程序编起。作为初学者，最重要的就应当是夯实编程的基本功，充分掌握程序语言的基本知识，掌握编程的基本技巧，这样一点点积累起来最终成为一个编程高手。

本章就是特意为基础一般的读者编写的。它通过一些比较基础的编程实例揭示了如何编写一个地道的C程序。同时帮助读者复习巩固已有的C语言知识和编程技巧。如果你是一个学过C语言但是编程实践不多的人，或是很久没有编程的人士，那么建议你学习本章。

### 4.1 字符类型统计器

#### 【题目要求】

请编写一个C程序，在终端用键盘输入字符串，以Ctrl+Z组合键表示输入完毕，统计输入的字符串中空格符、制表符、换行符的个数，并显示统计的结果。

#### 【题目分析】

这是一类在面试中或计算机考试中常考的题目，具有一定的普遍性。本例题涉及的知识点是字符串的输入输出方法以及字符的判断等知识。程序设计的关键是如何辨认出从终端输入的字符哪个是空格符，哪个是制表符，哪个是换行符。然后在字符串输入的过程中，通过不同的变量记录下来每一种字符的个数即可。解决字符的分类问题可以通过字符的ASCII码进行判断，因为不同的字符对应不同的ASCII码。通过查表可知空格符的ASCII码为32，制表符的ASCII码为9，换行符的ASCII码为10。可以通过它们不同的ASCII码来区分出它们。此外，Ctrl+Z的组合键输入的字符在计算机中对应的是EOF结束标志。

程序清单 4-1

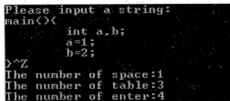
```
/*----- 4-1.c -----*/
#include <string.h>
#include <stdio.h>
main()
{
    char c;
    int space=0,table=0,enter=0;
    printf("Please input a string:\n");
    scanf("%c",&c);
    while(c!=EOF)
    {
        switch(c){
            case 32:space++;break;
            case 9: table++;break;
```

```

        case 10:enter++;break;
        default:break ;
    }
    scanf("%c",&c);
}
printf("The number of space:%d\n",space);
printf("The number of table:%d\n",table);
printf("The number of enter:%d\n",enter);
getchar();
return 0;
}

```

程序的运行结果如图 4-1 所示。



```

Please input a string:
main()>
    int a,b;
    a=1;
    b=2;
>^Z
The number of space:1
The number of table:3
The number of enter:4

```

图 4-1 程序 4-1 的运行结果

## 4.2 计算字符的 ASCII 码

### 【题目要求】

编写一个程序，在终端输入一个字符，输出它的 ASCII 码。

### 【题目分析】

解决本题的关键在于理解字符在内存中的存储方式。通过 C 语言的学习我们知道，一个字符在内存中的存放形式是以它的 ASCII 码形式存放的，大小为 8 位（bits），一个字节。例如空格符的 ASCII 码为 32，那么在内存中 32 对应的 8 位二进制数 100000 就代表一个空格符。根据这个道理就不难解决上述问题。只要变换一种输出的形式就可以显示出该字符的 ASCII 码。

通常用我们输出一个字符一般用 `printf("%c",x);` 的形式，因为输出格式规定的是“%c”，因此表示以字符的形式输出，所以我们看到的是 x 对应的 ASCII 码的字符形式。例如执行 `printf("%c", 97);` 语句等价于执行 `printf("%c", 'a');` 语句。屏幕上输出的只是一个字符 a。如果换一种输出形式，例如 `printf("%d", x);`，那么输出的就是 x 对应的 ASCII 码的整数形式。这样用户通过这个输出就可以知道字符 x 对应的 ASCII 码了。

程序清单 4-2

```

/*----- 4-2.c -----*/
#include "stdio.h"
main()
{
    char c;
    printf("Please input a character\n");
}

```

```
scanf("%c",&c);
getchar();
printf("The ASCII of %c is %d\n",c,c);
}
```

#### 【程序说明】

注意最后一条语句，输出的内容是同样的变量 *c*，输出的格式前者是“%c”，后者是“%d”，输出的内容因此而不同，前者是字符本身，后者是字符对应的 ASCII 码。

程序的运行结果如图 4-2 所示。

```
Please input a character
d
The ASCII of d is 100
```

图 4-2 程序 4-2 的运行结果

## 4.3 嵌套 if-else 语句的妙用

#### 【题目要求】

学校进行成绩分级管理，取消分数制，改为成绩分级评定。具体办法是：小于 60 分为 E 类；60 分至 70 分（不含 70 分）为 D 类；70 分至 80 分（不含）为 C 类；80 分至 90 分（不含）为 B 类；90 分以上为 A 类。设计一个程序，对输入的成绩进行等级划分。

#### 【题目分析】

读者可能很快就能想到这个问题的解决方法，通过一个 if-else if 语句就可以轻松地实现题目的要求。可以这样设计算法：

```
if score<60 then belong to class "E"
else if score<70 then belong to class "D"
else if score<80 then belong to class "C"
else if score<90 then belong to class "B"
else belong to class "A"
```

但是如果更深入地考虑一下就知道这并不是最好的解决方案。可以先看一下上述解决方案对应的流程图，如图 4-3 所示。

通过该程序执行的流程图可以看出，如果一个学生的成绩为 95 分，那他至少要在该程序中进行 4 次比较才能最终得到结果。而根据一般的统计规律，一个班的成绩一般服从正态分布，也就是说平均成绩一般集中在 70~80 分之间，因此应用这个程序进行成绩等级划分的效率是不高的。解决的方法是成绩减少比较的平均搜索长度，提高 70 分和 80 分的比较优先级。可以设计这样一个程序执行的流程，如图 4-4 所示。

从上述流程图中可以看出，应用这种划分成绩的流程可以减少程序的平均比较次数。因为它最多只要比较 3 次就能得到最终结果，而且将 70 分和 80 分的比较优先级提高了。应用这个方法来进行成绩等级划分的效率显然要比上一个高。如果是在大型系统中，需要比较的次数更多，那这个程序的优势将更加明显。

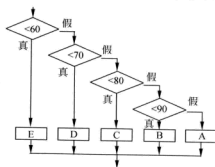


图 4-3 第一种解决方案

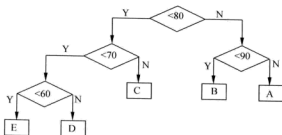


图 4-4 改进的解决方案

其实只要使用例如

```

if(...)
    if(...)
        else ...
    else...
  
```

的 if-else 嵌套形式就可以实现这样的功能。但是使用 if-else 嵌套语句时必须注意 else 与 if 要配对正确，否则可能出现程序的逻辑混乱，以至不能得到正确结果。

程序清单 4-3

```

/*----- 4-3.c -----*/
#include <stdio.h>
int main(void)
{
    int score;
    printf("Please input the score\n");
    scanf("%d",&score);
    if(score<80)
        if(score<70)
            if(score<60)
                printf("E\n");
            else
                printf("D\n");
        else
            printf("C\n");
    else
        if(score<90)
            printf("B\n");
        else
            printf("A\n");
    getchar();
    return 0;
}
  
```

程序的运行结果如图 4-5 所示。

```

Please input the score
75
C
  
```

图 4-5 程序 4-3 的运行结果

## 4.4 基于 switch 语句的译码器

### 【题目要求】

给定一个前缀码表如下：

a	b	c
1	01	001

又知有一个 0/1 字符串为：“001011101001011001”，编写一个 C 程序，按照给定的前缀码表为该字符串译码。

### 【题目分析】

前缀码是一种无二义性的编码，因此可以作为一种编码体制被广泛应用。一些编码，例如：

a: 001 b:0011 c:1

这就会出现所谓的译码时的二义性，也就是说当翻译到 0011 时无法确定是 ac 还是 b。因此如果要设计长短不一的编码，必须使用前缀码。应用广泛的哈夫曼编码就是一种前缀码。

现要求编写一个程序来为已知的这串 0/1 代码译码。译码的方法很多，现在介绍一种最为直观，最为简单的译码方式：应用 switch 语句的嵌套进行译码。

根据 switch 语句的特点可以设计以下算法实现这个译码器。

```
Repeat:
Switch (当前 0/1 码)
Case 1: 输出 a; break;          ---a:1
Case 0:
    指针指向下一个 0/1 码
    Switch (当前 0/1 码)
    Case 1: 输出 b; break;      ---b:01
    Case 0:
        指针指向下一个 0/1 码
        Switch (当前 0/1 码)
        Case 1: 输出 c; break;  ---c:001
    指针指向下一个 0/1 码
Until 待翻译的 0/1 码串结束
```

0/1 字符串流从左至右进入译码器，每翻译一个编码都要执行以下的判断过程。

- (1) 如果是 1，则译码为 a；否则进入状态 (2)。
- (2) 如果是 1，则译码为 b；否则进入状态 (3)。
- (3) 如果是 1，则译码为 c，否则不属于预定义的编码。

然后再进入下一个编码的翻译，直到 0/1 字符串流结束。

应用 switch 嵌套语句实现的译码算法，可以只扫描一遍 0/1 代码串就可以翻译出全部内容。算法的具体结构要依赖于码表的定义。应用 switch 嵌套语句来实现该译码算法，优点在于简单直观，很容易理解，且翻译效率较高。缺点在于代码量较大，如果码表庞大，代码量也会随之增大，而且比较机械，缺乏灵活性。这里旨在用以说明 switch 语句的嵌套

使用。

程序清单 4-4

```

/*----- 4-4.c -----*/
#include "stdio.h"
void Decode(char *str,int n);
main()
{
    char str[18]="001011101001011001";
    Decode(str,18);
    getchar();
}
void Decode(char *str,int n)
{
    int i=0;
    while(i<n)
    {
        switch(str[i])
        {
            case '1':printf("a");break;
            case '0':
            {
                i++;
                switch(str[i])
                {
                    case '1':printf("b");break;
                    case '0':
                    {
                        i++;
                        switch(str[i])
                        {
                            case '1':printf("c");break;
                        }
                        break;
                    }
                }
            }
            break;
        }
        i++;
    }
}

```

cbaabcbac\_

程序的运行结果如图 4-6 所示。

图 4-6 程序 4-4 的运行结果

## 4.5 判断闰年

### 【题目要求】

输入一个年份，判断该年是否是闰年。

### 【题目分析】

这是一道公司面试和各种 C 语言考试中常考的题目。虽然这道题目十分简单，但是作为一个专业的程序员这是必备的常识。所谓闰年，是要符合下面两个条件之一。

- 该年份能被 4 整除，但不能被 100 整除。

□ 该年份能被4整除，又能被400整除。

只要符合以上的两条件之一的年份都是闰年。最简单的判断闰年的方法是用条件判断语句（if-else 语句）配合逻辑表达式进行判断，充分地利用逻辑表达式进行判断可使程序的可读性更好，效率更高。

程序清单 4-5

```

/*----- 4-5.c -----*/
#include "stdio.h"
main()
{
    int year;
    printf("Please a year:\n");
    scanf("%d",&year);
    /*判断是否是闰年*/
    if((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
        printf("%d is leap year!\n",year);
    else
        printf("%d is not leap year!\n",year);
    getch();
}

```

#### 【程序说明】

源程序中 if 语句的判断条件是：

```
(year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
```

而不是：

```
(year % 4 == 0 && year % 100 != 0) || (year % 4
== 0 && year % 400 == 0)
```

是因为如果 year 能被 400 整除必然能被 4 整除，因此可以省略 year % 4 == 0 的判断。

程序的运行结果如图 4-7 所示。

```

Please a year:
2008
2008 is leap year!

```

图 4-7 程序 4-5 的运行结果

## 4.6 指针变量作参数

#### 【题目要求】

编写一个函数 inputArray，该函数被主函数调用，通过该函数实现向主函数中定义的数组输入数据。

#### 【题目分析】

程序设计结构化要求将具有特定功能的程序块编写成函数，这样程序结构清晰，可读性强，易于调试，代码的复用性强。因此把向数组中输入数据这样的程序块单独编写成为一个函数，通过主函数调用实现其功能，这样符合结构化程序设计的要求。

但是这里重要的一点是如何通过被调函数改变主函数中的值。我们知道，函数的调用是在内存的堆栈中实现的。因此，一旦函数调用完毕，该函数内部的所有局部变量都会被释放掉。本题目要求对主函数中的数组值进行修改，因此无法通过函数的返回值来实现该功能。所以要想通过被调函数修改主函数中的数组值，只能将数组的首地址（指针）作为



函数的参数进行传递,被调函数通过主函数传递来的数组地址对数组值进行修改。

程序清单 4-6

```

/*----- 4-6.c -----*/
#include "stdio.h"
void inputArray(int *array,int len)
{
    int i;
    for(i=0;i<len;i++)
        scanf("%d",array+i);
}
main()
{
    int i, array[10];
    printf("Please input data for array\n");
    inputArray(array,10);
    for(i=0;i<10;i++)
        printf("%d ",array[i]);
    printf("\n");
}

```

### 【程序说明】

本程序中,函数 `inputArray()` 的实参 `array` 是数组名,它是一个地址常量。形参的 `array` 是一个指针变量,用来接收实参传递过来的数组的首地址。对于本题,函数 `inputArray()` 的形参也可定义成 `int array[]` 的形式,但是必须注意这种形式的形参只限于数组首地址的传递。对于一般的指针传递,形参还是定义为指针变量。在语句 `scanf` 中,参数 `array+i` 是分别指向第  $i$  个数组元素的指针,也就是数组第  $i$  个元素的地址,它等价于 `&array[i]`。

程序的运行结果如图 4-8 所示。

```

Please input data for array
1 2 3 4 5 6 7 8 9 0
1 2 3 4 5 6 7 8 9 0

```

图 4-8 程序 4-6 的运行结果

## 4.7 矩阵的转置运算

### 【题目要求】

用键盘从终端输入一个 3 行 4 列的矩阵,编写一个函数对该矩阵进行转置操作。

### 【题目分析】

矩阵的转置运算是线性代数中的一个基本运算。例如矩阵  $A_{ij}$  如下:

$$A_{ij} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$A_{ij}$  为一个 3 行 3 列的矩阵,其中矩阵中第  $i$  行第  $j$  列的元素是  $a_{ij}$ 。定义矩阵  $A_{ij}$  的转置矩阵为  $B_{ij}$ ,则矩阵  $A^T_{ij}$  为:

$$A_{ij}^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

在矩阵  $A_{ij}^T$  中, 第  $i$  行第  $j$  列的元素为矩阵  $A_{ij}$  中第  $j$  行第  $i$  列的元素。把  $A_{ij}^T$  这样的矩阵称为矩阵  $A_{ij}$  的转置矩阵。显然, 一个  $m$  行  $n$  列的矩阵经过转置运算后就变成了一个  $n$  行  $m$  列的矩阵。

这个问题的解决关键是要解决两个问题: (1) 数据在内存中的存储问题, 也就是数据结构的问题; (2) 如何通过函数来实现矩阵的转置运算。显然, 可以用一个二维数组存储矩阵的数据, 通过将二维数组的指针作为函数的参数进行传递, 来实现矩阵转置函数的功能。

程序清单 4-7

```

/*----- 4-7.c -----*/
#include "stdio.h"
#include <stdio.h>
void InputMatrix(int (*a)[4],int ,int );
void OutputMatrix(int (*b)[3],int ,int );
void MatrixTranspose(int (*a)[4],int (*b)[3]);

int main(void)
{
    int a[3][4],b[4][3];          /*a 存放原矩阵, b存放 a 矩阵的转置矩阵*/
    printf("Please input 3x4 matrix\n");
    InputMatrix(a,3,4);
    MatrixTranspose(a,b);
    printf("The Transpose Matrix is\n");
    OutputMatrix(b,4,3);
    getchar();
    return 0;
}

void InputMatrix(int (*a)[4],int n,int m)
{
    /*输入 n*m 阶的矩阵 */
    int i,j;
    for(i=0;i<n;i++)
        for(j=0;j<m;j++)
            scanf("%d",&a[i][j]);
}

void OutputMatrix(int (*b)[3],int n,int m)
{
    /* 输出 n*m 阶矩阵的值 */
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
            printf("%d ",*(b+i)[j]);
        printf("\n");
    }
}

void MatrixTranspose(int (*a)[4],int (*b)[3])
{
    /*矩阵的转置运算*/
    int i,j;

```

```

for(i=0;i<4;i++)
    for(j=0;j<3;j++)
        b[i][j]=a[j][i];
}

```

#### 【程序说明】

本程序中，通过 `InputMatrix()` 函数向主函数中定义的矩阵 `a` 输入数据。通过函数 `MatrixTranspose()` 实现矩阵的转置，即将矩阵 `a` 进行转置操作，并将结果存放在矩阵 `b` 中。通过函数 `OutputMatrix()` 输出转置矩阵 `b` 中的内容。在这些函数的参数中，形参都包含了一个指向二维数组的指针变量，例如：

```

int (*a)[4]
int (*b)[3]

```

因为这里的被调函数要对主函数中定义的二维数组进行修改，因此不能像 4.6 节中介绍的那样用 `int *a`，或者 `int a[]` 的形式作为形参。这是因为对于一个二维数组：

```
a[m][n]
```

数组名 `a` 指的是指向二维数组第一行的指针（地址），`a+1` 则指的是指向二维数组第 2 行的指针。因此 `a+i` 指的是指向二维数组的第  $i+1$  行的指针。它并不是指向一个整型变量，而是指向一个整型的一维数组。因此作为二维数组指针的传递，实参可以是数组名，但是形参一定是如 `(*a)[n]` 的形式。其中， $n$  表示该二维数组每行的元素个数，也就是列数。因此，

```
int (*a)[4]
```

表示 `a` 指向一个包含 4 个元素的一维数组。

在被调函数中，`a[i][j]`、`b[i][j]` 都表示主函数中二维数组 `a` 和 `b` 的对应的值。也就是说，可以通过 `a[i][j]`、`b[i][j]` 的形式直接引用主函数中的数组 `a` 和 `b` 的元素。

程序的运行结果如图 4-9 所示。

```

Please input 3X4 matrix
1 2 3 4
5 6 7 8
9 0 1 2
The Transpose Matrix is
1 5 9
2 6 0
3 7 1
4 8 2

```

图 4-9 程序 4-7 的运行结果

## 4.8 矩阵的乘法运算

#### 【题目要求】

有两个矩阵  $A_1$  和  $A_2$ ，分别如下：

$$A_1 = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad A_2 = \begin{bmatrix} 1 & 0 & 2 & 3 \\ 4 & 1 & 5 & 6 \\ 6 & 8 & 9 & 0 \end{bmatrix}$$

编写一个程序，实现这两个矩阵的乘积。

### 【题目分析】

先来讨论矩阵乘法的运算法则。一个  $i \times j$  的矩阵同 一个  $j \times k$  的矩阵相乘，结果是得到一个  $i \times k$  的矩阵。矩阵的乘法按照下列公式的法则进行：

$$\begin{pmatrix} a_{11} & \cdots & a_{1j} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ij} \end{pmatrix} \bullet \begin{pmatrix} b_{11} & \cdots & b_{1k} \\ \vdots & \ddots & \vdots \\ b_{j1} & \cdots & b_{jk} \end{pmatrix} = \begin{pmatrix} \sum_{t=1}^j a_{1t} \cdot b_{t1} & \cdots & \sum_{t=1}^j a_{1t} \cdot b_{tk} \\ \vdots & \ddots & \vdots \\ \sum_{t=1}^j a_{it} \cdot b_{t1} & \cdots & \sum_{t=1}^j a_{it} \cdot b_{tk} \end{pmatrix}$$

从上面的公式中可以看出，一个  $i$  行  $j$  列的矩阵  $A$  与一个  $j$  行  $k$  列的矩阵  $B$  相乘，得到一个  $i$  行  $k$  列的积矩阵  $C$ ，该矩阵中的元素  $c_{ij}$  为矩阵  $A$  的第  $i$  行与矩阵  $B$  的第  $j$  列对应元素的乘积的和。因此  $A$  矩阵的列数必须与  $B$  矩阵的行数相等。

从上面的计算法则中可以看出，要计算两个矩阵的乘积需要三重循环。即：

- $A$  矩阵的第  $m$  行和  $B$  矩阵的第  $n$  列各元素相乘，得到  $c_{mn}$ 。
- $A$  矩阵的第  $m$  行分别与  $B$  矩阵的第  $1 \sim k$  列相乘，得到  $c_{m1}, c_{m2}, \dots, c_{mk}$ 。
- $A$  矩阵的第  $1 \sim i$  行分别与矩阵的  $n$  列相乘，得到最终结果。

因此，解决矩阵相乘的问题的一种比较简单的方法就是用三重循环嵌套语句。

程序清单 4-8

```

/*----- 4-8.c -----*/
#include <stdio.h>
int main(void)
{
    int A[2][3]={{1,2,3},{4,5,6}};           /*初始化矩阵 A*/
    int B[3][4]={{1,0,2,3},{4,1,5,6},{6,8,9,0}}; /*初始化矩阵 B*/
    int C[2][4]={{0,0,0,0},{0,0,0,0}};
    int i,j,k;
    for(i=0;i<2;i++)
        for(j=0;j<4;j++)
            for(k=0;k<3;k++)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
    printf("The result is\n");
    for(i=0;i<2;i++) {
        for(j=0;j<4;j++)
            printf("%d ",C[i][j]);
        printf("\n");
    }
    return 0;
}

```

### 【程序说明】

程序中应用到三重 for 循环语句，每一层循环的作用如下。

(1) 最内层的 for 语句循环 3 次, 实现  $A$  矩阵的第  $m$  行和  $B$  矩阵的第  $n$  列对应元素相乘求和, 得到  $C$  矩阵中的元素  $c_{mn}$ 。

(2) 第二层循环执行 4 次, 实现  $A$  矩阵的第  $m$  行分别和  $B$  矩阵的第 1~4 列相乘。

(3) 最外层循环执行 2 次, 实现  $A$  矩阵的第 1~2 行分别与  $B$  矩阵的  $n$  列相乘, 得到最终结果。

程序的运行结果如图 4-10 所示。

```
The result is
27 26 39 15
60 53 87 42
```

图 4-10 程序 4-8 的运行结果

## 4.9 巧用位运算

### 【题目要求】

用位运算操作实现两个整数的交换。例如  $x1=5$ ,  $x2=10$ ; 交换后  $x1=10$ ,  $x2=5$ 。

### 【题目分析】

一般情况下要实现两个变量  $a$ 、 $b$  的内容交换, 多采用设置一个临时变量  $t$ , 通过

```
t=a;
a=b;
b=t;
```

的方法实现。其实还有一种更好更有效的方法实现这个功能, 就是采用位运算的方法。

具体的作法是:

```
a=a^b;
b=b^a;
a=a^b;
```

其中 $\wedge$ 为按位异或运算。这样同样可以达到  $a$ 、 $b$  两值互换的目的。

下面给出这种方法的具体证明:

因为: 令  $a'=a^b$ ; ( $a'$  为中间值), 所以:  $b=b^a'$ ; 即  $b=b^b^a=b^b^a^b=b^b^a=0^a=a$ 。

同理:  $a=a'^b=a^b^a=a^b^a^b=a^b^0^b=b$ 。

因此实现了  $a$  与  $b$  值的交换。

在这个证明中, 用到了异或运算的结合律和交换律。

巧妙地运用按位异或运算的方法, 可以省掉这个临时变量, 从而减少了程序运算的空间复杂度。

程序清单 4-9

```
/*----- 4-9.c -----*/
#include "stdio.h"
main()
{
    int a,b;
    a=5;
    b=10;
    printf("a=%d,b=%d\n",a,b);
    a=a^b;
```

```

b=b^a;
a=a^b;
printf("a=%d,b=%d\n",a,b);
getche();
}

```

```

a=5,b=10
a=10,b=5

```

程序的运行结果如图 4-11 所示。

图 4-11 程序 4-9 的运行结果

## 4.10 文件的读写

### 【题目要求】

创建一个后缀名为 txt 的文件，并向该文件中写入一个字符串，保存起来。再打开该文件，读出文件中的内容。

### 【题目分析】

本题主要考查对文件的读写操作。C 库函数中提供了一组对文件操作的函数，即 I/O 函数，通过调用这些函数程序员可以编写程序操纵磁盘上的文件，来实现软件特定的功能。

I/O 函数也叫输入输出函数，是 C 标准库函数中十分重要的一类函数。所有 I/O 函数都定义在 <stdio.h> 的头文件中。因此，在编写程序时要将头文件 <stdio.h> 包含在源程序的文档中。如下所示：

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

按照本题目的要求，需要用到以下 I/O 函数：

```

FILE *fopen(char *filename, char *type);           /*打开指定路径的文件*/
int fclose(FILE *fp);                             /*关闭文件*/
int fread(void *buf, int size, int count, FILE *fp); /*读文件函数*/
int fwrite(void *buf, int size, int count, FILE *fp); /*写文件函数*/

```

程序清单 4-10

```

/*----- 4-10.c -----*/
#include "stdio.h"
main()
{
    FILE *fp;
    char pathName[20], txt1[20]={'\0'}, txt2[20]={'\0'};
    int fileLen;
    /*打开文件*/
    printf("Please type the path name of the file\n");
    scanf("%s", pathName);
    fp=fopen(pathName, "w");
    /*将字符串写入文件*/
    printf("Please input a string to this file\n");
    scanf("%s", txt1);
    fileLen=strlen(txt1);
    fwrite(txt1, fileLen, 1, fp);
    fclose(fp);
    printf("The file has been saved\n");
    printf("The content of the file: %s is\n", pathName);
    fp=fopen(pathName, "r");
}

```

```

fread(txt2,fileLen,1,fp);
printf("%s\n",txt2);
getche();
}

```

### 【程序说明】

源程序中 `fp` 是一个 `FILE` 类型的变量，用它来保存一个文件的指针。`pathName` 是一个字符型的数组首地址，用来保存文件的路径。`txt1` 和 `txt2` 也是字符型的数组首地址，用来保存输入的字符串以及从文件中读出的字符串。将它们都赋初值 `{'\0'}`，`'\0'` 是 C 语言中默认的字符串结束标志。`fileLen` 用来存储输入的字符串的长度。

程序执行的过程如下。

(1) 首先通过 `scanf` 从终端输入一个字符串，这里用到 `%s` 的输入格式，将字符串直接输入到 `pathName` 指向的缓冲区中。在这里要将文件所在的路径以及文件名输入进去，例如：“`C:\\a.txt`”。由于程序本身的限制，以及 `%s` 输入格式的要求，这里必须注意几点：文件的路径名长度不能超过 20 bytes，盘符信息要以 “`C:\\`” 的形式输入，整个路径名中不能出现空格符。

(2) 然后通过 `fopen()` 函数打开指定的文件。如果磁盘上不存在该文件，则系统会自动生成一个同样名称的空文件。参数 `w` 是指以写方式打开一个文本文件。再通过 `scanf` 语句，以 `%s` 的格式输入一个字符串。并用 `fileLen` 记录下该字符串的长度。

(3) 然后用 `fwrite()` 函数将该字符串从 `txt` 缓冲区写入到指定的文件之中，最后关闭该文件。至此就会在指定的文件中输入了一串字符串，可以在指定的路径下找到该文件。

(4) 接下来再以读方式 `r` 打开该文件，并用 `fread()` 函数将该文件中的字符串读到 `txt2` 指向的缓冲区中，并通过 `%s` 的格式输出到屏幕上。

程序的运行结果如图 4-12 所示。

```

Please type the path name of the file
c:\\test.txt
Please input a string to this file
hello_world!
The file has been saved
The content of the file: c:\\test.txt is
hello_world!

```

图 4-12 程序 4-10 的运行结果

## 4.11 计算文件的大小

### 【题目要求】

编写一个 C 程序，用来计算指定文件的大小。

### 【题目分析】

计算指定文件大小的方法很多。最直观的方法是通过扫描整个文件计算出文件的字节数。但是这种方法对系统的开销很大，比较浪费时间。可以巧妙地利用 I/O 库中提供的函数来进行文件大小的判定。

在 I/O 函数中，有两个经常被使用的函数 `fseek()` 和 `ftell()`，它们的功能为如下。

- ❑ `fseek(FILE *fp, long offset, int base)`: 重定位流上的文件指针，即将 `fp` 指向的文件的位置指针移向以 `base` 为基准，以 `offset` 为偏移量的位置。
- ❑ `ftell(FILE *fp)`: 返回当前文件指针的位置。这个位置是指当前文件指针相对于文件开头的位移量。

因此可以先通过函数 `fseek()` 将文件的指针定位到文件的最后 `SEEK_END`，然后通过函数 `ftell()` 返回当前文件指针相对于文件开头的位移量，也就是文件的长度。

程序清单 4-11


```
/*----- 4-11.c -----*/
main()
{
    FILE *myf;
    long f;
    myf=fopen("C:\\test.txt","r");
    fseek(myf,0,SEEK_END);
    f=ftell(myf);
    fclose(myf);
    printf("The length of the file is %d bytes\n",f);
    getch();
}
```

#### 【程序说明】

本程序中要测试大小的文件是在 4.10 节中生成的文件 `C:\test.txt`，其执行过程如下：

- (1) 首先用 `fseek()` 函数重定位文件的指针，将文件的指针 `myf` 指向文件的尾部。
- (2) 然后通过 `ftell()` 函数将指针相对于文件开头的位移量，也就是文件大小返回，赋值给变量 `f`。

在 4.10 节向文件 `C:\test.txt` 写入的字符串为 “hello\_world! ”，因此大小为 12B。程序的运行结果如图 4-13 所示。



```
The length of the file is 12 bytes
Press any key to continue_
```

图 4-13 程序 4-11 的运行结果

## 4.12 记录程序的运行时间

#### 【题目要求】

任意编写一段程序，要求纪录并输出该段程序执行的时间。

#### 【题目分析】

在调试程序，分析代码和算法的性能，查找系统瓶颈等时候，经常要统计一段代码或者一个模块执行的时间，通过观察代码执行的时间来分析代码的复杂度，效率和性能。因此知道如何记录程序运行的时间是一个程序员的基本功。

其实纪录一段代码执行的时间的方法非常简单，只要了解 `<time.h>` 库文件中定义的几



个函数就可以轻松地解决这个问题。

首先要了解 `clock_t` 类型。`clock_t` 类型是 `<time.h>` 库文件中定义的表示时间值的算术类型。也就是说应用 `clock_t` 类型的变量可以记录存储一个系统时间值。

其次要掌握一个库函数 `clock()`。`clock()` 函数的作用是返回从程序运行开始到调用 `clock()` 函数时所花费的处理器时间。返回值的类型是 `clock_t` 类型。

应用 `clock()` 函数就可以计算出程序运行的时间。

程序清单 4-12

```

/*----- 4-12.c -----*/
#include <time.h>
#include <stdio.h>
#include <dos.h>
int main()
{
    clock_t start, end;
    /*程序运行到现在的时间*/
    start = clock();
    /*间隔1秒*/
    sleep(1);
    /*程序运行到现在的时间*/
    end = clock();
    printf("The time was: %f\n", (end - start) / CLK_TCK);
    return 0;
}

```

#### 【程序说明】

程序的执行过程如下。

(1) 首先用 `clock()` 函数记录下程序从开始运行到当前所用的时间，并将该时间存放到 `clock_t` 类型的变量 `start` 中。

(2) 然后程序挂起 1s，再用 `clock()` 函数记录下程序运行到当前所用的时间，并将该时间存放到 `clock_t` 类型的变量 `end` 中。

(3) 最后计算出时间差 (`end-start`)，这个时间差就是程序执行 `sleep()` 函数所耗费的时间。用  $(end - start) / CLK\_TCK$  是将该时间差转换为以秒为单位。

The time was: 1.043956

程序的运行结果如图 4-14 所示。

图 4-14 程序 4-12 的运行结果

**注意：**由于系统上存在误差，因此这里记录的时间为 1.043956s，而并非精确的 1s。

## 4.13 十进制/二进制转化器

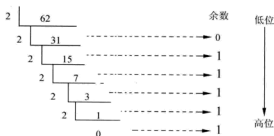
#### 【题目要求】

编写一个程序，将输入的十进制数转化为二进制表示。例如：输入十进制数 64，输出二进制数表示 1000000。

#### 【题目分析】

将一个十进制数转化为二进制数一般采取“除 2 取余”的方法。例如将十进制数 62

转化为二进制表示，方法如下：



“除2取余”法的步骤是：将十进制数不断除以2，并记录下每次除以2后得到的余数，先得到的余数为二进制数的低位，后得到的余数为二进制数的高位。该运算直到商为0为止。因此十进制数62的二进制表示为111110。

“除2取余”法的算法描述如下：

输入一个十进制数 a

Repeat:

    r = a/2;               /\*a 整除 2, 商给变量 r\*/

    s = a%2;             /\*a 整除 2, 余数给变量 s\*/

    将 s 入栈 Stack 保存

    If r!=0 then a←r

    Else 结束循环

将栈 Stack 的数据依次读出并打印在屏幕上，即为十进制 a 的二进制形式

该算法中要用到栈的操作。本节主要是为了介绍数据进制的转化，因此在这里只用一个静态的数组来模拟栈的操作，应用栈结构实现的进制转化程序可参见后续章节。

程序清单 4-13

```

/*----- 4-13.c -----*/
#include "stdio.h"
deTob1(int a){
    /*将十进制数 a 转化为二进制表示，并打印在屏幕上*/
    int i = 0, stack[10], r, s;
    do{
        r = a/2;          /*商*/
        s = a%2;          /*余数*/
        stack[i] = s;
        if(r!=0)
        {
            i++;
            a = r;        /*将 a 整除 2 的结果作为下一个整除 2 的对象*/
        }
    } while(r);          /*循环直到商 r 为 0 为止*/
    for(i>=0;i--)
        printf("%d",stack[i]);
    printf("\n");
}

main()
{
    int a;

```

```

printf("Please input a Decimal digit\n");
scanf("%d", &a);
deTobi(a);
getche();
}

```

### 【程序说明】

本程序实现将输入的十进制数转化为二进制表示。这里用一个函数 `deTobi()` 来实现数制的转化。该函数的参数是待转化的十进制数，函数的功能是将这个十进制数转化为二进制数，并打印出来。正如算法中描述的那样，循环执行  $a/2$  和  $a\%2$  两个操作。将  $a$  整除 2 的结果作为下一个整除 2 的对象。将  $a$  模 2 的结果入栈保存，因为先得到的结果是二进制数的低位，在屏幕上最后打印出来，因此应当先入栈保存。

为了简化程序，这里的栈结构是用一个整型数组 `stack[]` 代替的，因此它最多存放 10 个 0/1 数字，所以待转化的十进制数据  $a$  的大小不能超过 1023。这也正是本程序为了简化而带来的局限。最后将数组 `stack` 中的 0/1 数据按照栈操作的要求反向打印出来，得到的结果即是十进制数据  $a$  的二进制表示。

程序的运行结果如图 4-15 所示。

```

Please input a Decimal digit
116
1110100

```

图 4-15 程序 4-13 的运行结果

## 4.14 打印特殊图案

### 【题目要求】

在应用 C 语言开发程序时，有时为了程序运行界面的美观，需要在屏幕上用字符构成一些特殊的图案用以装饰。请设计一个 C 程序，实现在屏幕上输出一个类似于下面的图案。

```

      *
     * * *
    * * * * *

```

### 题目分析

这种关于特定格式输出的问题，最重要的一点是理解输出格式的规律。只要理解了输出格式的规律，通过简单的循环语句就可以显示出希望得到的图案。

现在来分析一下上面这个图案输出格式的规律。仔细分析上面的图案不难发现，上面的图案是由星号“\*”按照一定的规律逐行打印形成的一个三角形。共打印了 3 行“\*”字符，第一行 1 个“\*”，第二行 3 个“\*”，第三行 5 个“\*”。其中第一行的“\*”空出 2 个“\*”的位置，在第三个位置上打印出来，而且只打印 1 个；第二行的“\*”空出一个“\*”的位置，从第二个位置开始打印，共打印 3 个“\*”，第三行的“\*”就从起始位置开始打印，共打印 5 个“\*”。由此可以抽象出这样的规律：设共打印  $n$  行这样的“\*”构成一个等腰三角形，第  $i$  行的“\*”从第  $n-i+1$  的位置开始打印，共打印  $2i-1$  个“\*”即可，其中

$i=1,2,3\cdots$ 用这个规律来验证上面的图案显然也是符合的。

有了上述规律，不但可以打印出像上面这样的由三行“\*”字符构成的等腰三角形，还可以打印出更多行的“\*”字符构成的等腰三角形，其行数可由程序来控制。

打印该特殊图案的算法描述如下：

输入要打印等腰三角形图案的“\*”字符的行数  $n$ ：

```
i ← 1
repeat:
  打印 (n-i) 个 space
  打印 (2i-1) 个 "*"
  打印 1 个 "\n"
  i ← i+1
until i > n
```

程序清单 4-14

```
/*----- 4-14.c -----*/
#include <string.h>
#include <stdio.h>
void PrintTriangle(int n);
int main()
{
    int n;
    printf("How many rows of * for triangle\n");
    scanf("%d", &n);
    PrintTriangle(n);
    getchar();
    return 0;
}
void PrintTriangle(int n)
{
    int i, j;
    for (i=1; i<=n; i++)
    {
        for (j=0; j<n-i; j++)
            printf(" ");           /*打印 (n-i) 个 space*/
        for (j=0; j<2*i-1; j++)
            printf("*");           /*打印 (2i-1) 个 "*"*/
        printf("\n");
    }
}
```

#### 【程序说明】

在程序中定义函数 PrintTriangle() 实现在屏幕上打印出由“\*”构成的等腰三角图案的功能。参数  $n$  规定了该等腰三角形的高度（“\*”的行数）。

当  $n=3$  时，输出结果如图 4-16 所示。

```
How many rows of * for triangle
3
  *
 ***
*****
```

图 4-16  $n=3$  时的运行结果

当  $n=8$  时, 输出结果如图 4-17 所示。

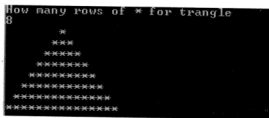


图 4-17  $n=8$  时的运行结果

## 4.15 打印杨辉三角

### 【题目要求】

在屏幕上打印出一个 6 阶杨辉三角。

### 【题目分析】

杨辉三角又称为贾宪三角, 是我国北宋数学家贾宪于 1050 年首先发现并使用的。而后南宋数学家杨辉在《详解九章算法》一书中记载并保存了“贾宪三角”。因此贾宪三角又被称为杨辉三角。杨辉三角的发现是我国数学史上光辉灿烂的一页, 它要比法国数学家帕斯卡发现的“帕斯卡三角”(即杨辉三角)早 600 多年。

所谓杨辉三角就是一个由数字排列成的三角形数表, 一般形式如下:

N=0	1
N=1	1 1
N=2	1 2 1
N=3	1 3 3 1
N=4	1 4 6 4 1
N=5	1 5 10 10 5 1
N=6	1 6 15 20 15 6 1
⋮	⋮

这里规定杨辉三角从第 0 行开始, 它的第  $N$  行就是二项式  $(a+b)^N$  的展开式的系数。

例如:

$$(a+b)^0 = 1$$

$$(a+b)^1 = a + b$$

$$(a+b)^2 = a^2 + 2ab + b^2$$

$$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$$

⋮

因此掌握了杨辉三角可以简化二项式的运算, 方便记忆, 在实际当中很有用处。

要想在屏幕上打印出一个杨辉三角, 首先要分析一下杨辉三角构成的规律。仔细观察上面的杨辉三角不难发现其规律: (1) 杨辉三角的两腰都为 1; (2) 杨辉三角

除腰上的1以外的各数，都等于它“肩上”的两数之和。例如，当 $N=2$ 时，中间的数字2等于它“肩上”的两数1和1的和；当 $N=4$ 时，数字4等于它“肩上”的两数1和3的和，6等于它“肩上”的两数3和3的和。

因此可以形式化地抽象出这样的规律：杨辉三角中的第 $i$ 行第 $j$ 个数据的值（其中 $i \neq 0, 1$ ； $j \neq 1, i+1$ ）等于第 $i-1$ 行的第 $j-1$ 和第 $j$ 个数据值之和；第 $i$ 行第 $j$ 个数据的值（其中 $i=0$ 或1； $j=1$ 或 $i+1$ ）都等于1。

从杨辉三角自身的排列规律易知，要打印一个杨辉三角要比打印一个4.14小节中所介绍的“\*”三角形难许多。因为这里在打印每一行的数列时都要知道它上一行的数，也就是说要打印第 $i$ 行的第 $j$ 个数据，必须知道第 $i-1$ 行中的第 $j-1$ 和第 $j$ 个数据。因此这里需要一个专门的缓冲区来存放第 $i-1$ 行的数据，而不可能像打印“\*”三角形那样直接打印“\*”就可以了。

对于本题，6阶杨辉三角规定为 $N=6$ ，即该杨辉三角为一个7行的等边三角形。打印6阶杨辉三角（ $N=6$ ）的算法描述如下：

```
开辟一个整型数组buf[7]，它可存放7个整数；          /*N=6时杨辉三角中数据为7个*/
i=0
Repeat:
    If (i == 0) 打印1
    If (i == 1) 打印两个1，并将两个1存放到buf[0]，buf[1]中
    Else
                                                /*打印第2-6行的数据*/
        j=1
        Repeat:
            If (j == 1 or j == i+1) 打印1
            Else 打印出buf[j-2] + buf[j-1]的值
            j←j+1
        until j>i+1
        将第i行的数据存放到buf[0] ~buf[i]中
        i←i+1
until i>6
```

程序清单 4-15

```
/*----- 4-15.c -----*/
#include "stdio.h"
main()
{
    int i,j,k, buf[7],tmp[7];
    for(i=0;i<=6;i++)
    {
        if(i==0)printf("%d",1);          /*打印第一行*/
        else if(i==1) {
            printf("\n%d %d\n",1,1);      /*打印第二行*/
            buf[0] = 1;
            buf[1] = 1;
        }
        else
        {
            for(j=1;j<=i+1;j++)
            { /*打印一行*/
                if(j==1 || j==i+1){
                    printf("%d ",1);
```

```

        tmp[j-1] = 1;
    }
    else{
        printf("%d ",buf[j-2] + buf[j-1]);
        tmp[j-1] = buf[j-2] + buf[j-1];
    }
}
printf("\n");
for(k=0;k<7;k++)
    buf[k] = tmp[k]; /*将第 i 行的数据存放到 buf[0] ~buf[i]中*/
}
}
}

```

**【程序说明】**

代码中通过两层的循环打印一个 6 阶杨辉三角，具体操作如下。

外层循环控制打印的行数，内层循环负责打印出每一行的内容。其中，杨辉三角的第 0 行和第 1 行直接打印出来，第 2 行至第 6 行由循环打印产生。在本程序中设置了一个数组 tmp 用来临时存放第  $i$  行的数据内容，待第  $i$  行打印完毕后再将 tmp 的内容赋值给数组 buf 中。这样数组 buf 中始终存放的是当前打印行的前一行内容。

本程序的运行结果如图 4-18 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

```

图 4-18 程序 4-15 的运行结果

**注意：**本程序中并没有严格控制杨辉三角的输出格式，这是为了读者看得更清楚明白。

其实只要在本程序的基础上稍加修改，就可以打印出一个对称的杨辉三角图案来，有兴趣的读者可以自己尝试。

## 4.16 复杂级数的前 $n$ 项和

**【题目要求】**

求级数  $\sum_{n=1}^{\infty} (1/2)^n n!$  的前 10 项和  $S_{10}$ 。

**【题目分析】**

本题要求级数  $\sum_{n=1}^{\infty} (1/2)^n n!$  的前 10 项和  $S_{10}$ ，将级数展开，实际就是求  $S_{10} = (1/2) + (1/2)^2 2! + \dots +$

$(1/2)^{10} 10!$ 。直观地看，最简单的解决方法就是应用循环。最外层的循环控制每一项的相加运算；内层包含两个循环体，一个是计算  $1/2$  的  $n$  次方，一个是计算  $n$  的阶乘。按照这样的思路，可以这样设计算法。

```

m = 1; /*记录 (1/2)^n*/
n = 1; /*记录 n!*/
s = 0; /*前 10 项和*/
for(i=1;i<=10;i++){ /*最外层循环，控制加法运算*/

```

```

for(j=0;j<i;j++)
    m = m * (1/2);          /*计算 (1/2)^n */
for(j=1;j<=i;j++)
    n = n * j;              /*计算 n!*/
s = s + m*n;               /*累加*/
m = 1;
n = 1;
}

```

在这里值得注意的是,在实际的编程中要注意变量类型的选取。存放 $(1/2)^n$ 的变量应当是浮点型或者双精度类型,因为它得到的不是一个整数,而是浮点数;存放 $n!$ 的变量应当是长整型,浮点型或双精度类型,因为它得到的可能会是一个很大的数,如果用整型变量存放结果会发生溢出。因此综合考虑, $S_{10}$ 的计算结果应存放到 `float` 类型的变量中比较合适。同时在实际的编程中, $1/2$  应书写成 `0.5`, 否则因为  $1/2$  是一个整除运算,它的结果是 `0`,而不是 `0.5`。

程序清单 4-16

```

/*----- 4-16.c -----*/
#include "stdio.h"
main()
{
    float m = 1.0;          /*记录 (1/2)^n*/
    float n = 1.0;          /*记录 n!*/
    float s = 0.0;
    int i,j;                /*前 10 项和*/
    for(i=1;i<=5;i++){     /*最外层循环,控制加法运算*/
        for(j=0;j<i;j++)
            m = m * 0.5;    /*计算 (1/2)^n */
        for(j=1;j<=i;j++)
            n = n * j;      /*计算 n!*/
        s = s + m*n;        /*累加*/
        m = 1.0;
        n = 1.0;
    }
    printf("Sn = %f\n",s);  /*输出结果*/
    getch();
}

```

Sn = 4467.625000

本程序的运行结果如图 4-19 所示。

图 4-19 程序 4-16 的运行结果

## 4.17 寻找矩阵中的“鞍点”

### 【题目要求】

在一个矩阵中,可能会有这样的元素:它在该行中最大,而在该列中最小。我们把这样的元素称为“鞍点”。一个矩阵中也可能没有鞍点。任意输入一个  $5 \times 5$  的矩阵,寻找该矩阵中的鞍点,并将它在矩阵中的位置(行,列)输出。

### 【题目分析】

仔细分析鞍点的定义可知,在一个矩阵中,最多出现一个鞍点。这是因为假设一个矩



阵  $A$  有两个鞍点  $a_{ij}$  和  $a_{(i+s)(j+t)}$ :

$$A = \begin{pmatrix} a_{ij} & b & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ c & a_{(i+s)(j+t)} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{pmatrix}$$

根据鞍点的定义可知:  $a_{ij} > b$ ,  $a_{ij} < c$ ; 因此  $c > b$ ; 同理  $a_{(i+s)(j+t)} > c$ ,  $a_{(i+s)(j+t)} < b$ ; 因此  $b > c$ ; 这样就产生了矛盾, 因此假设错误。所以一个矩阵中要存在鞍点也只能有一个。

因此在设计寻找鞍点的算法时, 可以逐行寻找鞍点。先找出某行中最大的元素 (要求同行中只能有一个最大的元素), 再将该元素与同列中的其他元素进行比较。如果该元素为同列中最小的元素, 则该元素即为该矩阵的鞍点, 于是返回它在矩阵中的位置, 程序结束; 否则继续对下一行进行上述的操作。算法描述如下:

```

对于一个 m*n 的矩阵
i ← 0;
Repeat:
    找出第 i 行中最大的元素 A[i][t];
    If (本行中有与元素 A[i][t] 的值相等的元素) then 本行中没有鞍点, 跳出 Repeat 循环;
    Else 将元素 A[i][t] 与第 t 列中的每个元素逐一进行比较,
        If (存在小于或等于 A[i][t] 的元素, 则 A[i][t] 不是该矩阵的鞍点) then 跳出 Repeat 循环;
    Else A[i][t] 是鞍点, 返回该元素在矩阵中的位置 (i, t), 程序结束。
    i ← i+1
until i >= m
返回 0, 该矩阵中无鞍点。

```

#### 程序清单 4-17

```

/*----- 4-17.c -----*/
#include "stdio.h"
int getSaddlePoint (int *x,int *y,int (*A)[5],int m,int n)
{
    int max , i , j , k , flag;
    for(i=0;i<m;i++) {
        /*逐行扫描寻找鞍点*/
        max = 0;
        /*max 记录第 i 行中最大值的列数*/
        flag = 1;
        for(j=1;j<n;j++)
            /*找出第 i 行中的最大元素的位置 A[i][max]*/
            {
                if(A[i][j]>A[i][max]) max = j;
            }
        for(k=0;k<n;k++)
            if(A[i][max] == A[i][k] && max != k) {flag = 0;break;}
        if(flag == 1)
            for(k=0;k<m;k++)
                /*判断 A[i][max] 是否是本列中最小的*/
                {
                    if(A[k][max]<=A[i][max] && k!=i)
                        /*不是最小元素, 因此 A[i][max] 不是鞍点*/
                        {
                            flag = 0;
                            break;
                        }
                }
            if(flag == 1)
                /*找到鞍点, 用 x,y 返回, 程序结束返回 1*/

```

```

    {
        *x = i;
        *y = max;
        return 1;
    }
}

return 0; /*没有找到鞍点, 返回 0*/
}

main()
{
    int A[5][5], i, j, x, y;
    printf("Please input some digit into the 5*5 matrix\n");
    for(i=0; i<5; i++)
        for(j=0; j<5; j++)
            scanf("%d", &A[i][j]);
    if(getSaddlePoint (&x, &y, A, 5, 5))
        printf("\nSaddlePoint is at (%d, %d)", x+1, y+1);
    else
        printf("\nThere is no in the matrix\n");
    getch();
}

```

本程序的运行结果如图 4-20 所示。

```

Please input some digit into the 5*5 matrix
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
5 6 7 8 9
SaddlePoint is at <1,5>

```

图 4-20 程序 4-17 的运行结果

## 4.18 $n$ 阶勒让德多项式求解

### 【题目要求】

$n$  阶勒让德多项式定义为:

$$P_n(x) = \begin{cases} 1 & (n=0) \\ x & (n=1) \\ ((2n-1) \cdot x - P_{n-1}(x) - (n-1) \cdot P_{n-2}(x)) / n & (n \geq 1) \end{cases}$$

编写程序, 输入正整数  $n$  和任意数  $x$ , 求出勒让德多项式的值  $P_n(x)$ 。

### 【题目分析】

显然勒让德多项式的定义是一种递归形式的定义。因此要求出  $n$  阶勒让德多项式的值, 最简单的方法就是编写一个递归函数实现。算法如下:

```

float Rand(int n, float x) {
    if (n==0) return 1;

```

```

    else if(n==1) return x;
    else return ((2*n-1)*x-Rand(n-1,x)-(n-1)*Rand(n-2,x))/n ;
}

```

函数 Rand() 的参数为  $n$  和  $x$ ,  $n$  表示  $n$  阶勒让德多项式, 随着函数 Rand() 的递归调用,  $n$  的值会逐层减少, 直到  $n$  等于 0 或 1, 递归结束。  $x$  为多项式函数的自变量。函数 Rand() 的返回值类型为 float。

程序清单 4-18

```

/*----- 4-18.c -----*/
#include "stdio.h"

float Rand(int n, float x) {           /*递归函数, 求勒让德多项式*/
    if (n==0) return 1;
    else if(n==1) return x;
    else return ((2*n-1)*x-Rand(n-1,x)-(n-1)*Rand(n-2,x))/n ;
}

main()
{
    int n;
    float x, p;
    printf("Please input n\n");
    scanf("%d", &n);                    /*输入勒让德多项式的阶*/
    printf("Please input x\n");
    scanf("%f", &x);                    /*输入自变量 x*/
    p = Rand(n, x);
    printf("The result of P%d(%f) = %f\n", n, x, p); /*输出结果*/
    getch();
}

```

本程序的运行结果如图 4-21 所示。

```

Please input n
5
Please input x
1.6
The result of P5(1.600000) = 1.560000

```

图 4-21 程序 4-18 的运行结果

## 4.19 递归反向输出字符串

### 【题目要求】

编写一个递归函数, 实现将输入的任意长度的字符串反向输出的功能。例如输入字符串: ABCD, 输出字符串: DCBA。

### 【题目分析】

应用递归的思想有时可以很轻松地实现一些看似不太容易实现的功能。本题就是利用递归方法解决这类问题的一个代表。要将一个字符串反向地输出, 我们一般采用的方法是将该字符串存储到一个数组中, 然后将数组元素反向地输出即可。这样需要一个存储空间,

而且字符串的长度无法自由掌握,因为数组是一种静态数据结构。如果选用动态生成的顺序表或者链表来存放字符串,那么实现起来会比较麻烦,特别是如果把字符串存放到单链表中,反向输出是非常困难的。

如何才能输入任意长度的字符串,然后将其反向输出呢?可以通过一个递归的方法巧妙地实现这个功能。算法描述如下:

```
Print()
{
    输入字符串的一个字符 a;
    If(a != '#') then Print();
    If(a != '#') then 输出该字符 a;
}
```

在该算法中,字符串的结束标志为#,并且#不作为字符串中的内容。首先输入字符串的一个字符,存放到变量 a 中;然后递归地调用函数 Print(),重复上述操作,直到输入字符串结束标志#为止;然后输出字符串中的字符。

假设从屏幕上输入字符串 ABC,并以#作为结束标志,函数 Print()的递归过程如图 4-22 所示。

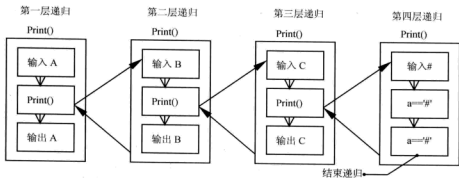


图 4-22 函数 Print()的递归过程

如图所示,从屏幕上输入字符串 ABC,并以#作为结束标志,然后反向输出该字符串,整个流程沿图中箭头方向执行。因此输出的字符串与输入的字符串方向相反。

程序清单 4-19

```
/*----- 4-19.c -----*/
#include "stdio.h"

Print()
{
    char a;
    scanf("%c",&a);
    if(a != '#') Print();
    if(a != '#')
        printf("%c",a);
}

main()
{
    printf("Please input a string ending for '#'\n");
```

```
Print();
getche();
}
```

本程序的运行结果如图 4-23 所示。

```
Please input a string ending for '#'
abcdefghijklmnopqrstuvwxyz_
```

图 4-23 程序 4-19 的运行结果

## 4.20 一年中的第几天

### 【题目要求】

输入某年某月某日，判断这一天是这一年的第几天？

### 【题目分析】

要判断某年某月某日是这一年中的第几天，就要计算出该月以前几个月的天数总和，再加上本月的日期。例如：要计算 3 月 6 日是本年中的第几天，就要计算出 1 月和 2 月的天数总和，再加上 3 月的日期，即 6，这样就可以得出 3 月 6 日是本年中的第几天。但是根据常识我们知道，一年中每个月的天数基本都是固定的，除了 2 月。在平年中，2 月的天数为 28 天；在闰年中，2 月的天数为 29 天。因此在设计程序时要考虑到这一点，应当首先用年份来判断该年是平年还是闰年，再进行计算。计算某年某月某日是这一年的第几天的算法描述如下：

```
int Day(int year, int month, int date)
{
    int months[13] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}, i, days = 0;
    if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
        /*判断是否是闰年*/
        months[2] = 29;
    else
        months[2] = 28;
    for (i = 1; i < month; i++)
        /*计算天数*/
        days = days + months[i];
    days = days + date;
    return days;
}
```

函数 Day() 包含 3 个参数，分别为年份 (year)、月份 (month) 和日期 (date)。函数 Day() 的作用是返回该日期在这一年中的天数。函数中定义了一个数组 months，用来存放每月的天数。为了使数组 months 的下标与月份相对应，本数组第 1 个元素 months[0] 置 0，从 months[1] 开始真正有效。这样 months[i] 表示第 i 月的天数 (除 2 月)。

(1) 程序首先判断 year 是否为闰年，判断闰年的方法在本章第 5 节中已有介绍。如果 year 为闰年，将 months[2] 赋值为 29，否则赋值为 28。

(2) 然后通过一个循环操作计算第 month 月以前几个月的天数总和，即 1 月 ~ (month-1) 月的天数总和，再执行 days = days + date; 语句，加上本月的日期。

(3) 最终 days 为第 month 月第 date 日在 year 年中的天数。

程序清单 4-20

```
/*----- 4-20.c -----*/
#include "stdio.h"
int Day(int year ,int month,int date)
{
    int months[13]={0,31,0,31,30,31,30,31,31,30,31,30,31} ,i,days=0;
    /*判断是否是闰年*/
    if((year % 4 == 0 && year % 100 !=0)|| (year % 400 ==0 ))
        months[2]=29;
    else
        months[2]=28;
    for(i=1;i<month;i++)
        days = days + months[i];
    days = days + date;
    return days;
}

main()
{
    printf("The days of 6th Match 2009 is %d\n",Day(2009,3,6));
    getch();
}
```

本程序的运行结果如图 4-24 所示。

The days of 6th Match 2009 is 65

图 4-24 程序 4-20 的运行结果

## 第5章 数学趣题（一）

在现实生活中有许多有趣的数学问题。经常有意识地寻找并解决这些问题可以增强我们的逻辑思维能力，进而开发我们的大脑，提高我们的智力水平，同时使生活变得丰富多彩。计算机就是帮助我们解决这些问题的强有力的工具。经常练习通过编写程序解决数学难题，可以减少我们在解题时遇到的繁琐而复杂的计算，把精力集中在解决具体问题的方法上，从而锻炼我们的思考能力，逻辑思维水平，同时提高自身的编程水平和应用计算机解决实际问题的能力。

本章将讲解如何通过程序设计来解决一些有趣的数学问题。通过本章的学习，读者不但可以了解一些有趣的数学问题的求解方法，而且可以通过举一反三扩大知识面，提高应用计算机编程解决实际问题的能力。

### 5.1 舍罕王的失算

#### 【题目要求】

舍罕是古印度的国王，据说他十分好玩，宰相达依尔为讨好国王，发明了现今的国际象棋献给国王。舍罕非常喜欢这项游戏，于是决定嘉奖达依尔，许诺可以满足达依尔提出的任何要求。达依尔指着舍罕王前面的棋盘提出了要求：“陛下，请您按棋盘的格子赏赐我一点麦子吧，第1个小格赏我一粒麦子，第2个小格赏我两粒，第3个小格赏四粒，以后每一小格都比前一个小格赏的麦粒数增加一倍，只要把棋盘上全部64个小格按这样的方法得到的麦粒都赏赐给我，我就心满意足了。”舍罕王听了达依尔这个“小小”的要求，想都没想就满口答应下来。

结果在给达依尔麦子时舍罕惊奇地发现它要给达依尔的麦子比自己想象的要多得多，于是他进行了计算，结果令他大惊失色。问题是：舍罕王的计算结果是多少粒麦子？

#### 【题目分析】

其实这个问题解决起来非常的简单。虽然题目叙述很多，但是我们应当先抽象出它的数学模型。

达依尔要求：第1个格子赏1粒麦子，第2个格子赏2粒麦子，第3个格子赏4粒麦子，……以后每一小格都比前一个小格赏的麦粒数增加一倍，直到赏够64个小格子为止。因此不难得出这样的结论：达依尔要求的是第*i*个格子赏 $2^{i-1}$ 粒麦子（ $i=1,2,\dots,64$ ）；而达依尔希望得到麦粒的总数便是： $\sum_{i=1}^{64} 2^{i-1}$ 。因此， $\sum_{i=1}^{64} 2^{i-1}$ 就是舍罕王失算的结果。

程序清单 5-1

/\*----- 5-1.c -----\*/

```
#include <stdio.h>
#include "math.h"
main()
{
    double sum = 0;           /*定义双精度变量 sum 存放结果*/
    int i;
    for(i=1;i<=64;i++)
        sum = sum + pow(2,i-1);    /*累加和*/
    printf("The number of the grain is %nf\n",sum);    /*打印出结果*/
}
```

**【程序说明】**

程序中定义 sum 为 double 类型是因为该题目的运算结果是一个 20 位十进制的大数，在 C 语言的基本数据类型中，只有 double 类型和 long double 类型的数据可以容纳。另外函数 pow(x,y)的作用是计算  $x^y$ ，函数的返回值为 double 类型。

程序的运行结果如图 5-1 所示。

```
The number of the grain is
18446744073709552000.000000
```

图 5-1 程序 5-1 的运行结果

## 5.2 求两个数的最大公约数和最小公倍数

**【题目要求】**

编写一个程序计算两个正整数的最大公约数和最小公倍数。

**【题目分析】**

所谓两个数最大公约数就是指两个数  $a$ 、 $b$  的公共因数中最大的那一个。例如：4 和 8，两个数的公共因数分别为 1、2、4，其中 4 为 4 和 8 的最大公约数。

因此要计算出两个数的最大公约数，最简单的方法就是从两个数中较小的那个开始依次递减，得到的第一个这两个数的公因子数即为这两个数的最大公约数。

如果一个数  $i$  为  $a$  和  $b$  的公共因数，那么一定满足  $a\%i$  等于 0，并且  $b\%i$  等于 0。所以，设计算法时只需从  $i=\min(a,b)$  开始依次递减 1，并逐一判断  $i$  是否为  $a$  和  $b$  的公共因数，得到的第一个公因数就是  $a$  和  $b$  的最大公因数。

所谓两个数的最小公倍数就是指两个数  $a$ 、 $b$  的公共倍数中最小的那一个。例如：5 和 3，两个数的公倍数可以是 15、30、45……，其中 15 最小，因此 15 是 5 和 3 的最小公倍数。

因此要计算两个数的最小公倍数，最简单的方法就是从两个数中最大的那个数开始依次加 1，得到的第一个公共倍数就为这两个数的最小公倍数。

如果一个数  $i$  为  $a$  和  $b$  的公共倍数，那么一定满足  $i\%a$  等于 0，并且  $i\%b$  等于 0。所以，设计算法时只需从  $i=\max(a,b)$  开始依次加 1，并逐一判断  $i$  是否为  $a$  和  $b$  的公共倍数，得到的第一个公倍数就是  $a$  和  $b$  的最小公倍数。

**程序清单 5-2**

```
/*----- 5-2.c -----*/
```



```

#include "stdio.h"
int gcd(int a,int b){
/*最大公约数*/
    int min;
    if(a<=0||b<=0) return -1;
    if(a>b) min = b;           /*找到 a、b 中的较小的一个赋值给 min*/
    else min = a;
    while(min){
        if(a%min == 0 && b%min == 0) /*判断公因数*/
            return min;           /*找到最大公约数, 返回*/
        min--;                   /*没有找到最大公约数, min 减 1*/
    }
    return -1;
}

int lcm(int a,int b){
/*最小公倍数*/
    int max;
    if(a<=0||b<=0) return -1;
    if(a>b) max = a;           /*找到 a、b 中的较大的一个赋值给 max*/
    else max = b;
    while(max){
        if(max%a == 0 && max%b == 0) /*判断公倍数*/
            return max;           /*找到最小公倍数, 返回*/
        max++;                   /*没有找到最小公倍数, max 加 1*/
    }
    return -1;
}

main()
{
    int a,b;
    printf("Please input two digit for getting GCD and LCM\n");
    scanf("%d %d",&a,&b);
    printf("The GCD of %d and %d is %d\n",a,b,gcd(a,b));
    /*打印出 a、b 的最大公约数*/
    printf("The LCM of %d and %d is %d\n",a,b,lcm(a,b));
    /*打印出 a、b 的最小公倍数*/
    getch();
}

```

程序的运行结果如图 5-2 所示。

```

Please input two digit for getting GCD and LCM
4 10
The GCD of 4 and 10 is 2
The LCM of 4 and 10 is 20

```

图 5-2 程序 5-2 的运行结果

### 5.3 歌德巴赫猜想的近似证明

#### 【题目要求】

众所周知, 歌德巴赫猜想的证明是一个世界性的数学难题, 至尽未能完全解决。我国著名数学家陈景润为歌德巴赫猜想的证明作出过杰出的贡献。

所谓歌德巴赫猜想是说任何一个大于2的偶数都能表示成为两个素数之和。应用计算机工具可以很快地在一定范围内验证歌德巴赫猜想的正确性。请编写一个C程序，验证指定范围内歌德巴赫猜想的正确性，也就是近似证明歌德巴赫猜想（因为不可能用计算机穷举出所有正偶数）。

### 【题目分析】

可以把问题归结为在指定范围内（例如：1~2000 内）验证其中每一个偶数是否满足歌德巴赫猜想的论断，即是否能表示为两个素数之和。如果发现一个偶数不能表示为两个素数之和，即不满足歌德巴赫猜想的论断，则意味着举出了反例，从而可以否定歌德巴赫猜想。

可以应用枚举的方法枚举出指定范围内的每一个偶数，然后判断它是否满足歌德巴赫猜想的论断，一旦发现有不满足歌德巴赫猜想的数据，则可以跳出循环，并做出否定的结论；否则如果集合内的数据都满足歌德巴赫猜想的论断，则可以说明在该指定范围内，歌德巴赫猜想是正确的。

这一过程的伪码算法描述为：

```
low      ;范围下界
high     ;范围上界
a←low
repeat:
    if a 为偶数并且 a>2 then
        if a 满足歌德巴赫猜想 then
            输出一种结论
        else
            设置标志，跳出循环
        endif
    endif
a←a+1
until a>high
if 设置标志 then 歌德巴赫猜想不成立
else 在[low,high]内歌德巴赫猜想成立
```

现在问题的核心变为如何验证一个偶数  $a$  是否满足歌德巴赫猜想，即偶数  $a$  能否表示为两个素数之和。可以这样考虑这个问题：

一个正偶数  $a$  一定可以表示成为  $a/2$  种正整数相加的形式。这是因为  $a=1+(a-1)$ ； $a=2+(a-2)$ ；……； $a=a/2-1+a/2+1$ ； $a=a/2+a/2$ ；共  $a/2$  种。后面还有  $a/2-1$  种表示形式与前面  $a/2-1$  种表示形式相同，因此可以不考虑。那么，在这  $a/2$  种正整数相加的形式中，只要存在一种形式  $a=i+j$ ，其中  $i$  和  $j$  均为素数，则就可以断定该偶数  $a$  满足歌德巴赫猜想。因此，判断一个大于2的偶数  $a$  是否满足歌德巴赫猜想的伪码算法描述为：

```
i←1
repeat:
    if i 是素数 and a-i 也是素数 then
        设置标志，跳出循环
    endif
    i←i+1
until i>a/2
if 设置标志 then a 满足歌德巴赫猜想
else a 不满足歌德巴赫猜想
```

## 程序清单 5-3

```

/*----- 5-3.c -----*/
#include <string.h>
#include <stdio.h>

int isGoldbach(int a);
int TestifyGB_Guess(int low,int high);
int isPrime(int i);
void main()
{
    /*验证1~100以内的歌德巴赫猜想*/
    printf("Now testify Goldbach Guess in the range of 1~100\n\n");
    if(TestifyGB_Guess(1,100))
        printf("\nIn the range of 1~100,Goldbach Guess is right.\n");
    else printf("\nGoldbach Guess is wrong.\n");
    getchar();
}

int TestifyGB_Guess (int low,int high)
{
    /*在 low 和 high 的范围内验证歌德巴赫猜想*/
    int i,j=0;
    int flag=0;
    for(i=low;i<=high;i++)
        if(i%2==0&& i>2)
            if(isGoldbach(i)){
                /*偶数 i 符合歌德巴赫猜想*/
                j++;
                /*j 用来控制输出格式*/
                if(j==5){
                    printf("\n");
                    j=0;
                }
            }
            else
                {flag=1;break;}
    if(flag==0)
        return 1;
    else
        return 0;
}

/*在 low 和 high 的范围内歌德巴赫猜想正确返回 1*/
/*在 low 和 high 的范围内歌德巴赫猜想不正确返回 0*/

int isGoldbach(int a)
{
    /*判断偶数 a 是否符合歌德巴赫猜想*/
    int i,flag=0;
    for(i=1;i<=a/2;i++)
    {
        if(isPrime(i)&& isPrime(a-i))
            /*如果 i 和 a-i 都为素数, 则符合歌德巴赫猜想*/
            {
                flag=1;
                printf("%d=%d+%d ",a,i,a-i);
                break;
            }
    }
    if(flag==1)
        return 1;
    else
        return 0;
}
/*a 符合歌德巴赫猜想返回 1*/
/*a 不符合歌德巴赫猜想返回 0*/

```

```

int isPrime(int i)
{
    /*判断 i 是否是素数*/
    int n, flag=1;
    if(1==i) flag=0;          /*1 不是素数，素数都要大于 1*/
    for(n=2; n<i; n++)
        if(i%n==0) {flag=0; break;}
    /*如果在 2~i-1 之间 i 有其他因子，则 i 不是素数，flag 置 0*/

    if(flag==1)
        return 1;           /*i 是素数返回 1*/
    else
        return 0;           /*i 不是素返回 0*/
}

```

### 【程序说明】

本程序将歌德巴赫猜想验证的范围规定在 100 以内。主函数调用 3 个函数：TestifyGB\_Guess() 函数有两个参数，分别指定歌德巴赫猜想验证的范围的下界 low 和上界 high，该函数的作用是在 [low, high] 的范围内验证歌德巴赫猜想的正确性。函数 isGoldbach() 有一个参数 a，它的作用是判断 a 是否满足歌德巴赫猜想，也就是偶数 a 是否可表示成两个素数之和。函数 isPrime() 有一个参数 i，判断 i 是否是素数，这里要注意，严格地讲 1 不是素数。

程序的运行结果如图 5-3 所示。

```

Now testify Goldbach Guess in the range of 1~100
4=2+2 6=3+3 8=3+5 10=3+7 12=5+7
14=3+11 16=3+13 18=5+13 20=3+17 22=3+19
24=5+19 26=3+23 28=5+23 30=7+23 32=3+29
34=3+31 36=5+31 38=7+31 40=3+37 42=5+37
44=3+41 46=3+43 48=5+43 50=3+47 52=5+47
54=7+47 56=3+53 58=5+53 60=7+53 62=3+59
64=3+61 66=5+61 68=7+61 70=3+67 72=5+67
74=3+71 76=3+73 78=5+73 80=7+73 82=3+79
84=5+79 86=3+83 88=5+83 90=7+83 92=3+89
94=5+89 96=7+89 98=19+79 100=3+97
In the range of 1~100, Goldbach Guess is right.

```

图 5-3 程序 5-3 的运行结果

## 5.4 三色球问题

### 【题目要求】

有红、黄、绿三种颜色的球，其中红球 3 个，黄球 3 个，绿球 6 个。现将这 12 个球混放在一个盒子中，从中任意摸出 8 个球，编程计算摸出球的各种颜色搭配。

### 【题目分析】

这是一道排列组合的问题。从 12 个球中任意摸出 8 个球，求颜色搭配的种类。解决这类问题的一种比较简单直观的方法是应用穷举法，在可能的解空间中找出所有的搭配，然后再根据约束条件加以排除，最终筛选出正确的答案。

针对本题，由于是任意地从 12 个球中摸取，一切都是随机事件，因此每种颜色的球被摸到的可能的个数如下表 5-1 所示。

表 5-1 每种颜色的球被摸到的可能的个数

红 球	黄 球	绿 球
0, 1, 2, 3	0, 1, 2, 3	2, 3, 4, 5, 6

显然绿球不可能被摸到 0 个或者 1 个。因为假设只摸到 1 个绿球, 那么摸到的红球和黄球的总数一定为 7, 而红球与黄球全部被摸到的总数才为 6, 因此假设是不可能成立的。同理, 绿球也不可能为 0 个。

下面就要在表 4-1 所划定的可能解空间的范围内寻找答案了。如果将红黄绿三色球可能被摸到的个数排列组合到一起构成解空间, 那么解空间的大小为  $4 \times 4 \times 5 = 80$  种颜色搭配组合。但是在这 80 种颜色搭配组合中, 只有满足“红球数+黄球数+绿球数=8”的才是真正的答案, 其余的搭配组合都不能满足题目的要求。解决三色球问题的算法描述如下:

```

Red = 0
Repeat:
  Yellow = 0
  Repeat:
    Green = 2
    Repeat:
      If Red + Yellow + Green = 8 Then 输出这组 (Red, Yellow, Green)
      Green ← Green + 1
    Until Green = 8
  Yellow ← Yellow + 1
  Until Yellow = 3
Red ← Red + 1
Until Red = 3

```

程序清单 5-4

```

/*----- 5-4.c -----*/
#include "stdio.h"
/*三色球问题求解*/
main()
{
    int red, yellow, green;
    printf("red yellow green\n");
    for(red=0; red<=3; red++)
        for(yellow=0; yellow<=3; yellow++)
            for(green=2; green<=6; green++)
                if(red+yellow+green == 8)
                    printf("%d %d %d\n", red, yellow, green);
}

```

程序的运行结果如图 5-4 所示。

```

red yellow green
0 2 6
0 3 5
1 1 6
1 2 5
1 3 4
2 0 6
2 1 5
2 2 4
2 3 3
3 0 5
3 1 4
3 2 3
3 3 2

```

图 5-4 程序 5-4 的运行结果

## 5.5 百钱买百鸡问题

### 【题目要求】

我国古代数学家张丘建在《算经》一书中曾提出过著名的“百钱买百鸡”问题。该问题叙述如下：鸡翁一，值钱五；鸡母一，值钱三；鸡雏三，值钱一；百钱买百鸡，则翁、母、雏各几何？请编写 C 程序，解决“百钱买百鸡”问题。

### 【题目分析】

如果用数学的方法解决百钱买百鸡问题，可将该问题抽象成如下的方程组：

设鸡翁  $x$  只，鸡母  $y$  只，鸡雏  $z$  只。

则

$$\begin{cases} 5x + 3y + \frac{1}{3}z = 100 \\ x + y + z = 100 \end{cases}$$

显然该方程组应有无数多解，因为根据代数知识，该方程组必有一个自由未知数，因此其解是无数组的。但是作为一个实际问题，其解是有限的。因为这里有约束条件：所有解（ $x$ 、 $y$ 、 $z$  取值）必须为正整数。所以，这就把该问题的解集放到了一个有限的空间内来进行讨论。

因此，解决百钱买百鸡问题最为直观的方法就是在（ $x$ 、 $y$ 、 $z$ ）的取值空间上穷举出所有可能的取值，只要（ $x$ 、 $y$ 、 $z$ ）的取值满足上述两个方程，就一定是百钱买百鸡问题的解。

现在讨论（ $x$ 、 $y$ 、 $z$ ）的取值空间。最简单的划分方法就是： $x$ 、 $y$ 、 $z \in \mathbb{R}$  且  $0 \leq x \leq 100$ ； $0 \leq y \leq 100$ ； $0 \leq z \leq 100$ 。这是因为无论  $x$ 、 $y$ 、 $z$  都不可能超过 100 或小于 0，且  $x$ 、 $y$ 、 $z$  均为正整数。因此，百钱买百鸡问题的解空间大小为  $101^3$ ，即就是要穷举出这  $101^3$  个可能的解，并在这  $101^3$  个可能的解中找出百钱买百鸡问题的真正的解。

百钱买百鸡的算法描述如下：

```
repeat:
    j ← 0
    repeat:
        k ← 0
        repeat:
            if i, j, k 的值符合上述两个方程 then
                (i, j, k) 为一个解向量，输出之。
            endif
            k ← k + 1
        until k > 100
        j ← j + 1
    until j > 100
    i ← i + 1
until i > 100
```

上述算法利用了三重循环遍历了整个（ $x$ 、 $y$ 、 $z$ ）的取值空间，然后从中找出该问题的解。

程序清单 5-5

```

/*----- 5-5.c -----*/
#include <string.h>
#include <stdio.h>

int accord(int i,int j,int k) ;

void main()
{
    int i,j,k;
    printf("The possible plans for buying 100 fowls with 100 yuan are:\n\n");
    for(i=0;i<=100;i++)
        for(j=0;j<=100;j++)
            for(k=0;k<=100;k++)
                if(accord(i,j,k))
                    printf("cock=%d,hen=%d,chicken=%d\n",i,j,k);
    getchar();
}

int accord(int i,int j,int k)
{
    if (5*i+3*j+k/3==100&&k%3==0&&i+j+k==100)    /*显然 k 必为 3 的整数倍*/
        return 1;                                /*符合百千百鸡要求返回 1*/

    else
        return 0;                                /*不符合百千百鸡要求返回 0*/
}

```

**【程序说明】**

程序中判断  $i$ 、 $j$ 、 $k$  是否满足方程组时，加  $k\%3=0$  约束条件的原因是  $k$  为整型数据，因此  $k/3$  的结果会自动舍弃小数部分，这样会造成误差（例如  $78/3=26$ ， $80/3=26$ ），影响最终结果。而这里显然  $k$  必为 3 的整数倍，因此增加约束条件  $k\%3=0$ 。

程序的运行结果如图 5-5 所示。

```

The possible plans for buying 100 fowls with 100 yuan are:
cock=0,hen=25,chicken=25
cock=4,hen=18,chicken=20
cock=8,hen=11,chicken=81
cock=12,hen=4,chicken=84

```

图 5-5 程序 5-5 的运行结果

## 5.6 判断回文数字

**【题目要求】**

有这样一类数字，它们顺着看和倒着看是相同的数，例如 121、656、2332 等，这样的数字叫做回文数字。编写一个程序，判断从键盘接收的数字是否为回文数字。

**【题目分析】**

要想判断一个数是否是回文数字，必须从回文数字的特点入手。因为回文数字顺着看和倒着看是相同的数，所以可以通过这个特点来判断一个数字是否是回文数字。

显然可以通过将一个十进制数“倒置”的办法来判断它是否是回文数字。所谓倒置就是计算该十进制数倒过来后的结果。例如一个数是 123，它的倒置结果为 321，因为 123 不等于 321，所以 123 不是回文数字。同理，一个数是 121，它的倒置结果也为 121，所以 121 是回文数字。

这一过程的伪码算法描述为：

```
输入一个十进制数 i
计算 i 的倒置数 j
if i=j then i 是回文数字
else i 不是回文数字
endif
```

接下来就是本题的关键，如何计算一个十进制数  $i$  的“倒置数” $j$ 。

假设一个十进制数为  $10^2a_1+10a_2+a_3$ ，显然它的倒置数应该是  $10^2a_3+10a_2+a_1$ 。因此关键是得到每一位上的数字  $a_i$ 。例如十进制数  $10^2a_1+10a_2+a_3$ ，求它的倒置数  $10^2a_3+10a_2+a_1$  的过程可描述为：

(1) 将十进制数  $10^2a_1+10a_2+a_3$  模 10，得到个位数字  $a_3$ ，并将十进制数  $10^2a_1+10a_2+a_3$  整除 10，得到新的十进制数字  $10a_1+a_2$ 。

(2) 将十进制数字  $10a_1+a_2$  模 10，得到个位数字  $a_2$ ，并将十进制数  $10a_1+a_2$  整除 10，得到新的十进制数字  $a_1$ 。同时将  $a_3$  乘以 10 加上  $a_2$ ，得到  $10a_3+a_2$ 。

(3) 将十进制数字  $a_1$  模 10，得到个位数字  $a_1$ ，同时将  $10a_3+a_2$  乘以 10 加上  $a_1$ ，从而得到  $10^2a_3+10a_2+a_1$ ，即为倒置数。

因此，可以抽象出下面的算法来计算  $i$  的倒置数。

```
m ← i
j ← 0
repeat:
    j ← j*10 + (m MOD 10)
    m ← m 整除 10
until m ≤ 0
```

最后可将上述两个算法映射成为两个函数，通过函数的调用实现该功能。

程序清单 5-6

```
/*----- 5-6.c -----*/
#include <string.h>
#include <stdio.h>

int isCircle(int n);          /*判断 n 是否是回文数字*/
int reverse(int i);          /*计算 i 的倒置数*/

void main()
{
    int n;
    printf("Type a integer for judging is Circle:\n");
    scanf("%d", &n);          /*从屏幕输入一个数*/
    if(isCircle(n))            /*判断是回文数字*/
        printf("%d is Circle\n", n);
    else
        printf("%d is not Circle\n", n); /*判断不是回文数字*/
}
```



```

    getchar();
}

int isCircle(int n)
{ /*函数 isCircle()判断 n 是否是回文数字*/
    int m;
    m= reverse(n);
    if(m==n)
        return 1;
    else
        return 0;
}

int reverse(int i)
{ /*求 i 的倒置数*/
    int m,j=0;
    m=i;
    while(m){
        j=j*10+m%10;
        m=m/10;
    }
    return j; /*返回 i 的倒置数 j*/
}

```

程序的运行结果如图 5-6 所示。

```

Type a integer for judging Circle or not:
131
131 is Circle

```

图 5-6 程序 5-6 的运行结果

## 5.7 填数字游戏求解

### 【题目要求】

有这样一个算式：

$$\begin{array}{r}
 ABCD \\
 \times \quad E \\
 \hline
 DCBA
 \end{array}$$

其中 ABCDE 代表的数字各不相同。编写一个程序，计算出 ABCDE 各代表什么数字。

### 【题目分析】

这道题的实质就是求这样一个 4 位数 ABCD 和 1 位数 E, 要求它们的乘积等于 DCBA。同时 A、B、C、D、E 互不相等。也就是在 4 位的整数集合[1000, 9999]和 1 位的整数集合[1, 9]中找到符合上述算式条件的 4 位数 ABCD 和 1 位数 E。因此不难想到应用穷举法可以方便地找到答案。其算法描述如下：

```

ABCD←1000
Repeat:
    E←1
    Repeat:

```

```

        If ABCD * E == DCBA and A,B,C,D,E互不相等 Then
            输出这个答案
            E←E+1
        Until E>9
        ABCD←ABCD + 1
    Until ABCD>9999

```

通过上述的算法一定可以找到满足题目要求的4位数ABCD和1位数E,因为算法将4位数空间的每一个数与1位数空间的每一个数都进行了搭配比较,答案不会超出这个范围。下面最关键的问题就集中在如何判断 $ABCD \times E = DCBA$ 和如何判断A、B、C、D、E互不相等。

判断 $ABCD \times E$ 是否等于DCBA的方法可以参考4.6节中判断回文数字的方法。只要求出ABCD的“倒置数”DCBA来,再判断 $ABCD \times E$ 是否等于DCBA即可。判断A、B、C、D、E是否互不相等的方法是将4位数ABCD的每一位分解出来,然后再进行A、B、C、D、E的逐一比较。这里必须注意,在比较A、B、C、D、E时既不要漏掉任何一对的比较,也不要重复地比较。需要比较的数对分别是:(A,B)(A,C)(A,D)(A,E)(B,C)(B,D)(B,E)(C,D)(C,E)(D,E)10对。

程序清单 5-7

```

/*----- 5-7.c -----*/
#include "stdio.h"
int reverse(i)
{
    /*求i的倒置数*/
    int r = 0;
    while(i)
    {
        r = r * 10 + i % 10;
        i = i / 10;
    }
    return r;
}

int fun(int i, int j)
{
    /*判断ABCDE 5个数字是否相同*/
    int buf[4], k = 0;
    while(i){
        buf[k] = i % 10;
        i = i / 10;
        k++;
    }
    if(buf[0] == buf[1]) return 0; /* A跟B比较 */
    if(buf[0] == buf[2]) return 0; /* A跟C比较 */
    if(buf[0] == buf[3]) return 0; /* A跟D比较 */
    if(buf[0] == j) return 0; /* A跟E比较 */
    if(buf[1] == buf[2]) return 0; /* B跟C比较 */
    if(buf[1] == buf[3]) return 0; /* B跟D比较 */
    if(buf[1] == j) return 0; /* B跟E比较 */
    if(buf[2] == buf[3]) return 0; /* C跟D比较 */
    if(buf[2] == j) return 0; /* C跟E比较 */
    if(buf[3] == j) return 0; /* D跟E比较 */
}

```

```

    return 1;
}

main()
{
    int i,j;
    for(i=1000;i<10000;i++)           /*遍历4位数空间[1000, 9999]*/
        for(j=1;j<10;j++)             /*遍历1位数空间[1, 9]*/
            if(i * j == reverse(i) && fun(i,j)) /*如果 i、j 满足题目的要求*/
            {
                printf("%d\n",i);
                printf("* %d\n",j);
                printf("-----\n");
                printf("%d\n\n",i*j);
            }
}

```

程序的运行结果如图 5-7 所示。

```

The result is
2178
* 4
-----
8712

```

图 5-7 程序 5-7 的运行结果

## 5.8 新郎和新娘

### 【题目要求】

3 对新婚夫妇参加婚礼, 3 个新郎为 A、B、C, 3 个新娘为 X、Y、Z。有人不知道谁和谁结婚, 于是询问了 6 位新人中的 3 位, 但听到的回答是这样的: A 说他将与 X 结婚; X 说她的未婚夫是 C; C 说他将与 Z 结婚。这人听后知道他们在开玩笑, 全是假话。请编程找出谁将和谁结婚。

### 【题目分析】

如果“乱点鸳鸯谱”的话, 3 个新郎 A、B、C 和 3 个新娘 X、Y、Z 共有 6 种配对组合方式。这是因为不能出现两个新郎(新娘)和一个新娘(新郎)结婚的状况, 所以只可能有  $3 \times 2 \times 1 = 6$  种配对方案。因此只要穷举出这 6 种配对方案, 再应用问题中给出的约束条件就可以筛选出正确的答案来。

现在的关键是如何找出这 6 种配对方案。可以这样考虑解决问题: 假定新郎 A、B、C 的顺序是不变的, 不断调换新娘 X、Y、Z 的位置。具体地, 新郎用 `husband[3] = {'A','B','C'}` 表示, 新娘用 `wife[3] = {'X','Y','Z'}` 表示。用 i、j、k 3 个变量不断调整 wife 与 husband 的配对方式, 如表 5-2 所示。

表 5-2 wife[i]、wife[j]、wife[k]与husband[0]、husband[1]、husband[2]的配对关系

husband[0]	wife[i]
husband[1]	wife[j]
husband[2]	wife[k]

规定 wife[i] 为 husband[0] 的新娘, wife[j] 为 husband[1] 的新娘, wife[k] 为 husband[2] 的新娘。i、j、k 的值在 0~2 中不断调整变化, 例如: i=0;j=1;k=2 或者 i=1;j=0;k=2……。这样随着 i、j、k 的每一次调整, 就得到一种配对方案。这里必须注意  $i \neq j \neq k$ , 否则就会出现“2 个新郎配 1 个新娘”的情况。变量 i、j、k 的调换可以用下面的代码实现。

```
for(i=0;i<3;i++)           /*新郎 A 的配对*/
    for(j=0;j<3;j++)       /*新郎 B 的配对*/
        for(k=0;k<3;k++)   /*新郎 C 的配对*/
            if(i!=j && j!=k && i!=k) /*不能 1 个新娘配 2 个新郎*/
            {
                ...
            }                /*得到一种配对方式*/
```

通过上述代码, 就可以得到全部 6 种配对方案, 下面就是如何根据题目的叙述筛选出符合要求的答案。由于如表 5-2 的配对顺序, 又由于题目所述: “A 说他将与 X 结婚; X 说她的未婚夫是 C; C 说他将与 Z 结婚。这人听后知道他们在开玩笑, 全是假话”。可以得到下面一段代码:

```
int match(int i,int j,int k,char wife[])
{
    /*A 不和 X 结婚*/
    if(wife[i] == 'X') return 0;
    /*X 不和 C 结婚*/
    if(wife[k] == 'X') return 0;
    /*C 不和 Z 结婚*/
    if(wife[k] == 'Z') return 0;
    return 1;
}
```

函数 match() 判断 wife[i]、wife[j]、wife[k] 是否可以与 husband[0]、husband[1]、husband[2] 实现真正的配对。如果 wife[i] = 'X', 则不符合 A 不和 X 结婚的要求; wife[k] = 'X', 则不符合 X 不和 C 结婚的要求; 如果 wife[k] = 'Z', 则不符合 C 不和 Z 结婚的要求。否则就符合题目的要求了。

程序清单 5-8

```
/*----- 5-8.c -----*/
#include "stdio.h"

main()
{
    char husband[3] = {'A','B','C'}, wife[3] = {'X','Y','Z'};
    int i, j, k;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            for(k=0;k<3;k++)
                if(i!=j && j!=k && i!=k) /*不能 1 个新娘配 2 个新郎*/
                {
                    /*得到一种配对方式*/
                    if (match(i,j,k,wife))
                    {
                        printf("husband wife\n");
                    }
                }
}
```

```

        /*这种配对方式符合题目要求*/
        printf("A-----%c\n",wife[i]);
        printf("B-----%c\n",wife[j]);
        printf("C-----%c\n",wife[k]);
    }
    getch();
}

int match(int i,int j,int k,char wife[])
{
    /*A不和X结婚*/
    if(wife[i] == 'X') return 0;
    /*X不和C结婚*/
    if(wife[k] == 'X') return 0;
    /*C不和Z结婚*/
    if(wife[k] == 'Z') return 0;
    return 1;
}

```

**【程序说明】**

函数 match()实现配对方案的筛选, 由于数组 wife 定义在主函数中, 这里将数组的首地址作为函数的参数进行传递。

程序的运行结果如图 5-8 所示。

```

husband  wife
A-----Z
B-----X
C-----Y

```

图 5-8 程序 5-8 的运行结果

## 5.9 爱因斯坦的阶梯问题

**【题目要求】**

爱因斯坦曾出过这样一道有趣的数学题: 有一个长阶梯, 若每步上 2 阶, 最后剩 1 阶; 若每步上 3 阶, 最后剩 2 阶; 若每步上 5 阶, 最后剩 4 阶; 若每步上 6 阶, 最后剩 5 阶; 只有每步上 7 阶, 最后刚好一阶也不剩。请问该阶梯至少有多少阶。编写一个 C 程序解决该问题。

**【题目分析】**

把这个问题抽象成数学表达就是求这样一个数  $x$ ,  $x$  满足:

```

x mod 2=1
x mod 3=2
x mod 5=4
x mod 6=5
x mod 7=0

```

显然  $x$  的取值应该有无穷多个, 但这里要求取最小的那个解。

从上面这一串表达式中不难发现  $x$  一定是 7 的整数被, 这是因为  $x \bmod 7=0$ 。从这一

点出发就不难得到解决该问题的一个简单而直观的方法：可以依次递增地求出 7 的整数倍的值 ( $7 \cdot 1$ 、 $7 \cdot 2$ 、 $7 \cdot 3$ ……)，每求出一值，就用该值与 2、3、5、6 进行取模运算，最先得到的满足上述 5 个方程式的  $x$  值即为本题的答案。

解决该问题的伪码算法如下：

```
x ← 7
repeat
    if (x mod 2=1) and (x mod 3=2) and (x mod 5=4) and (x mod 6=5) then
        x 为所求答案，保存答案
        设置标志，跳出循环
    endif
until 超出循环范围
if 设置了标志 then 输出计算结果
else 输出在指定范围内没有找到答案
endif
```

程序清单 5-9

```
/*----- 5-9.c -----*/
#include <string.h>
#include <stdio.h>

void main()
{
    int x=7,i,res,flag=0;
    for(i=1;i<=100;i++) /*将循环次数定为 100*/
    {
        if((x%2==1)&&(x%3==2)&&(x%5==4)&&(x%6==5)) /*如果 x 符合题目叙述的要求*/
        {
            res=x;
            flag=1;
            break; /*跳出循环，不再进行比较*/
        }
        x=7*(i+1);
    }
    if(flag)
        printf("The result of Einstein's question is %d",res); /*输出答案*/
    else
        printf("In this range cannot get result\n "); /*在程序限定的范围内找不到答案*/
}
```

程序的运行结果如图 5-9 所示。

```
The result of Einstein's question is
119
```

图 5-9 程序 5-9 的运行结果

## 5.10 寻找水仙花数

### 【题目要求】

如果一个 3 位数等于其各位数字的立方和, 则称这个数为水仙花数。例如:  $407=4^3+0^3+7^3$ , 因此 407 就是一个水仙花数。编写一个程序, 找出全部的水仙花数。

### 【题目分析】

水仙花数是 3 位数, 只要应用穷举法穷举出 100~999 闭区间中的每一个数字(正整数), 然后对每一个正整数进行判断, 看它是不是水仙花数。如果是水仙花数, 则将该数输出, 如果不是水仙花数, 则不输出该数。可以用下面的算法进行描述。

```
i←100
repeat:
if i 是水仙花数 then 输出 i endif
i←i+1
until i>=1000
```

于是, 问题归结到如何判断一个数是否是水仙花数上。根据水仙花数的定义易知, 要判断一个数是否是水仙花数, 只需判断该数是否等于其各位数字的立方和。又因为如果一个数为  $a$ , 则  $a\%10$  即为该数的个位数字,  $a/10\%10$  即为该数的十位数字,  $a/100\%10$  即为该数的百位数字……依次类推, 因此, 判断一个数  $a$  是否是水仙花数的伪码算法可描述为:

```
sum←0
tmp←a
repeat:
sum←sum+(tmp mod 10)3
tmp←tmp/10
until tmp<=0
if sum 等于 a then
a 是水仙花数
else a 不是水仙花数
endif
```

程序清单 5-10

```
/*----- 5-10.c -----*/
int IsNarcissus(int a);
void Narcissus();
void main()
{
printf("The Narcissus numbers below are\n");
Narcissus();
getche();
}

void Narcissus()
{ /*寻找 100~999 之间的水仙花数*/
int i;
for(i=100;i<=999;i++)
if(IsNarcissus(i))
printf("%d ",i);
```

```

}

int IsNarcissus(int a)
{
    /*判断是否是水仙花数，是则返回1，不是返回0*/
    int sum=0, tmp;
    tmp=a;
    while(tmp>0)
    {
        sum=sum+(tmp%10)*(tmp%10)*(tmp%10);
        tmp=tmp/10;
    }
    if(sum==a)
        return 1;      /*a 是水仙花数*/
    else
        return 0;      /* a 不是水仙花数*/
}

```

程序的运行结果如图 5-10 所示。

```

The Narcissus numbers below are
153 370 371 407

```

图 5-10 程序 5-10 的运行结果

## 5.11 猴子吃桃问题

### 【题目要求】

有一只猴子第一天摘下若干个桃子，当即吃掉了一半，又多吃了一个；第二天又将剩下的桃子吃掉一半，又多吃一个；按照这样的吃法每天都吃前一天剩下的桃子的一半又一个。到了第十天，就只剩下一个桃子。问题：这只猴子第一天摘了多少个桃子。

### 【题目分析】

解决这道题目应从第 10 天的桃子数入手。因为猴子吃桃子的规律是“每天都吃前一天剩下的桃子的一半又一个”，因此第 9 天的桃子数应该为 4 个，只有这样猴子吃掉一半（2 个桃子）又一个（1 个桃子）后才会给第 10 天只剩下 1 个桃子。依次类推，可得到第 1 天的桃子数。可以抽象出这样的结论：如果第  $i$  天猴子的桃子数为  $a_i=n$ ，则第  $i-1$  天猴子的桃子数为  $a_{i-1}=2(n+1)$ ，即得到递推公式： $a_{i-1}=2(a_i+1)$ 。其中， $a_i$  为第  $i$  天的桃子数， $a_{i-1}$  为第  $i-1$  天的桃子数。

现在已知  $a_{10}=1$ ，根据递推公式可求出  $a_9$ ，同理由  $a_9$  可通过递推公式求出  $a_8$ ……最终由  $a_2$  通过递推公式求出  $a_1$ ，即得到答案。

### 程序清单 5-11

```

/*----- 5-11.c -----*/
#include "stdio.h"
main()
{
    int sum = 1, i;          /*sum 初始值为 1，表示第十天的桃子数*/

```



```

for(i=9;i>=1;i--)
    sum = (sum + 1) * 2 ;           /*每次循环都得出第1天的桃子数*/
printf("The number of peach are %d\n",sum);
getche();
}

```

**【程序说明】**

本题的解法很简单，解决本题的关键在于找出第  $i$  天桃子数  $a_i$  和第  $i-1$  天桃子数  $a_{i-1}$  的关系，这样通过第 10 天的桃子数 1 就可以递推出第 1 天的桃子数来。

程序的运行结果如图 5-11 所示。

The number of peach are 1534

图 5-11 程序 5-11 的运行结果

## 5.12 兔子产仔问题

**【题目要求】**

13 世纪意大利数学家斐波那契他的《算盘书》中提出这样一个问题：有人想知道一年一对兔子可繁殖成多少对，便筑了一道围墙把一对新生的兔子关在里面。已知一对两个月大的兔子以后每个月都可以生一对小兔子，而一对新生的兔子出生两个月后才可以生小兔子（例如：1 月份出生，3 月份才可产仔）。假如一年内没有发生死亡，则一年内共能繁殖成多少对？

**【题目分析】**

兔子产仔问题是一个经典而古老的数学问题。在解这道题目之前，首先要排除一些常识的干扰，你不要考虑兔子如何配对的问题，也就是说按照题目的要求：一对新生的兔子出生两个月后就可以生小兔子。因为数学家斐波那契提出这道兔子产仔问题就是为了揭示它背后的数学规律。因此读者没有必要追究它的实际性。

解决兔子产仔问题可以从围墙中的第一对兔子产仔开始研究，然后逐步总结归纳出兔子的数量变化的规律。可以这样归纳兔子产仔的规律：

1 月：1 对新生的兔子(A1,A2)。

2 月：1 对兔子(A1,A2)；因为兔子对(A1,A2)出生两个月后才可以生小兔子，第 2 月还没有繁殖能力。

3 月：1 对兔子(A1,A2) + 1 对新生的兔子(B1,B2)；因为兔子对(A1,A2)出生两个月后可以产仔。

4 月：1 对兔子(A1,A2) + 1 对兔子(B1,B2) + 1 对新生的兔子(C1,C2)；因为兔子对(A1,A2)继续产仔，兔子对(B1,B2)还没有繁殖能力。

.....

通过上面的规律可以看出：从第 3 个月开始，每个月的兔子总对数 = 前两个月兔子数的总和。因此可以抽象出这样结论：

设  $F_i$  为第  $i$  月份围墙内兔子的总数，则

$$F_i = \begin{cases} 1 & i=1 \\ 1 & i=2 \\ F_{i-1} + F_{i-2} & i \geq 3 \end{cases}$$

这就是著名的斐波那契数列，该数列的特点是从数列的第3项开始，每一项都等于前两项之和，而数列的第一项和第二项都为1。于是兔子产仔问题就转化为求  $F_{12}=?$ 。

很明显解决这个问题最简单的办法就是应用函数的递归调用，因为斐波那契数列本身就是递归定义的。算法描述如下：

```
F(n)
If n ==1 or n ==2 Then return 1;
Else return F(n-1) + F(n-2);
```

程序清单 5-12

```
/*----- 5-12.c -----*/
#include "stdio.h"
Fibonacci(n){ /*递归函数*/
    if (n==1 || n==2) return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2); /*递归调用函数 Fibonacci */
}
main()
{
    printf("There are %d pairs of rabbits 1 year later",Fibonacci(12));
    getch();
}
```

程序的运行结果如图 5-12 所示。

There are 144 pairs of rabbits 1 year later\_

图 5-12 程序 5-12 的运行结果

## 5.13 分解质因数

### 【题目要求】

根据数论的知识可知任何一个合数都可以写成几个质数相乘的形式，这几个质数都叫做这个合数的质因数。例如  $24=2 \times 2 \times 2 \times 3$ 。把一个合数写成几个质数相乘的形式表示，叫做分解质因数。对于一个质数，它的质因数可定义为它本身。编写一个程序实现分解质因数。

### 【题目分析】

所谓合数就是除了1和它本身之外还存在其他因数的数字，例如24除了1和24之外还有因数2等。因此可以这样考虑对  $n$  分解质因数：

在  $2 \sim n-1$  之间找出  $n$  的两个因数（不一定是质因数） $i$  和  $j$ ，即  $i \times j = n$ 。

如果  $i$  是质数，则  $i$  一定是  $n$  的一个质因数，否则继续对  $i$  进行质因数分解。

如果  $j$  是质数，则  $j$  一定是  $n$  的一个质因数，否则继续对  $j$  进行质因数分解。

很显然,这是一种递归的方法。在这个递归的过程中,如果对 $k$ 进行质因数分解,只有当 $k$ 的质因数 $s$ 、 $t$ 全部找到后,这一层的调用才会结束,返回上一层的调用中。否则会将 $s$ 或 $t$ 中非质数的那个因数作为参数,继续调用这个递归过程进行分解。因此,一旦执行完这个递归的调用,一定能够求出 $n$ 的全部质因数。

另外还必须注意两点:

(1) 如果从2开始到 $n-1$ 顺序地查找 $n$ 的因数,那么第一个找到的因数 $i$ 一定是质因数。例如: $24=2 \times 12$ 。

2是质因数。可用反证法证明:假设 $i$ 不是质因数,则 $i$ 除了1和 $i$ 还会有其他的因数,即存在 $p, q \in [2, n-1]$ 使得 $pq=i$ 。因此 $pq(n/i)=n$ , 因此 $p$ 和 $q$ 也是 $n$ 的因数。但是我们是从小开始递增求 $n$ 的因数的,又因为 $i$ 是第一个找到的因数,所以在 $i$ 之前不会有其他的因数,所以结论与题设产生矛盾。因此假设错误, $i$ 一定是质因数。

(2) 一个合数的质因数分解的结果是唯一的,因此与分解的顺序无关,如图5-13所示。

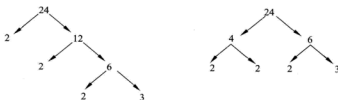


图 5-13 24 的质因子分解过程

虽然分解的次序不同,但是结果是一样的,24的质因子都是2、2、2、3。

根据以上的叙述,可以得到分解质因数的递归算法:

```
Factorization(n)
i ← 2
Repeat:
    If (n % i == 0)      /*i 是因数*/
    Then {
        i 是质因数, 输出 i
        If ( (n/i) 是质数 )
            Then 输出 n/i
        Else Factorization(n/i)
    }
    i ← i + 1
until i = n-1
```

程序清单 5-13

```
/*----- 5-13.c -----*/
#include "stdio.h"

int isPrime(int a)
{ /*判断a是否是质数, 是质数返回1, 不是质数返回0*/
    int i;
    for(i=2; i<=a-1; i++)
        if(a % i == 0)
            return 0;      /*不是质数*/
    return 1;
}
```

```

    return 1;                                /*是质数*/
}

void PrimeFactor(int n)
{ /*对参数 n 分解质因数*/
    int i;
    if(isPrime(n)) printf("%d ",n);
    else
    {
        for(i=2;i<=n-1;i++)
            if(n % i == 0)
            {
                printf("%d ",i);              /*第一个因数一定是质因数*/
                if(isPrime(n/i)) {             /*判断第二个因数是否是质数*/
                    printf("%d ",n/i);
                    break;                      /*找到全部质因子*/
                }
                else
                    PrimeFactor(n/i);          /*递归地调用 PrimeFactor 分解 n/i */
                break;
            }
    }
}

main()
{
    int n;
    printf("Please input a integer for getting Prime factor\n") ;
    scanf("%d",&n);
    PrimeFactor(n);    /*对 n 分解质因数*/
    getch();
}

```

**【程序说明】**

程序中通过递归函数 PrimeFactor()对参数 n 进行质因数分解。在函数 PrimeFactor()中,调用函数 isPrime()判断参数是否是质数。如果参数  $n/i$  为质数,则说明在本层中找到了全部质因数,递归结束。

程序的运行结果如图 5-14 所示。

```

Please input a integer for getting Prime factor
1155
3 5 7 11

```

图 5-14 程序 5-14 的运行结果

## 5.14 常胜将军

**【题目要求】**

现有 21 根火柴,两人轮流取,每人每次可以取走 1 至 4 根,不可多取,也不能不取,谁取最后一根火柴谁输。请编写一个程序进行人机对弈,要求人先取,计算机后取;计算

机一方为“常胜将军”。

### 【题目分析】

可以这样思考这个问题：要想让计算机是“常胜将军”，也就是要让人取到最后一根火柴。这样只有一种可能，那就是让计算机给人只剩下1根火柴，因为此时人至少取1根火柴。别的情况都不能保证计算机常胜。

于是问题转化为“有20根火柴，两人轮流取，每人每次可以取走1至4根，不可多取，也不能不取，要求人先取，计算机后取，谁取最后一根火柴谁赢”。为了让计算机取到最后一根火柴，就要保证最后一轮的抽取（人先取一次，计算机再取一次）之前剩下5根火柴。因为只有这样，无论人怎样取火柴，计算机都能将其余的火柴全部取走。

于是问题又转化为“15根火柴，两人轮流取，每人每次可以取走1至4根，不可多取，也不能不取，要求人先取，计算机后取，保证计算机取到最后一根火柴”。同样道理，为了让计算机取到最后一根火柴，就要保证最后一轮的抽取（人先取一次，计算机再取一次）之前剩下5根火柴。

于是问题又转化为10根火柴的问题……，依次递推。

最后可以得出这样的结论：21根火柴，在人先取，计算机后取，每次取1~4根的前提下，只要保证每一轮的抽取（人先取一次，计算机再取一次）人抽到的火柴数与计算机抽到的火柴数之和为5就可以实现计算机的常胜不败。

程序清单 5-14

```
/*----- 5-14.c -----*/
#include "stdio.h"
main()
{
    int computer , people , spare = 21;
    printf(" -----\n");
    printf(" ----- 你不能战胜我，不信试试 -----\n");
    printf(" -----\n\n");
    printf("Game begin:\n\n");
    while(1)
    {
        printf(" ----- 目前还有火柴 %d 根 -----\n",spare);
        printf("People:");
        scanf("%d",&people);
        if(people<1 || people>4 || people>spare)
        {printf("你违规了，你取的火柴数有问题!\n\n");continue;}
        spare = spare - people;
        if(spare == 0){printf("\nComputer win! Game Over!\n"); break;}
        computer = 5 - people;
        spare = spare - computer;
        printf("Computer:%d \n",computer);
        if(spare == 0){printf("\nPeople win! Game Over!\n"); break;}
    }
}
```

程序的运行结果如图 5-15 所示。

```

=====
你不能再胜我，不信试试
=====
Game begin!
-----
People:1 目前还有火柴 21 根
Computer:2
-----
People:2 目前还有火柴 16 根
Computer:2
-----
People:3 目前还有火柴 11 根
Computer:2
-----
People:5 你违规了，你取的火柴数有问题！
-----
People:2 目前还有火柴 11 根
Computer:3
-----
People:1 目前还有火柴 6 根
Computer:4
-----
People:1 目前还有火柴 1 根
Computer:4
-----
Computer win! Game Over!

```

图 5-15 程序 5-15 的运行结果

## 5.15 求 $\pi$ 的近似值

### 【题目要求】

编写一个 C 程序，用来求出  $\pi$  的近似值。

### 【题目分析】

求  $\pi$  的方法很多，这里两种最为常用的求  $\pi$  的方法。

方法 1：利用“正多边形逼近”法求  $\pi$ 。

“正多边形逼近”法求  $\pi$  的核心思想是极限的思想。假设一个直径  $d$  为 1 的圆，只要求出该圆的周长  $C$ ，就可以通过  $\pi=C/d$  的方法求出  $\pi$  的值。所以关键是求出该圆的周长  $C$ 。这里用“正多边形逼近”的方法求圆的周长。

早在我国古代，数学家们就知道应用正多边形逼近的方法近似地求解圆的周长，称其为“割圆术”。其核心思想是：当一个圆的内接正多边形边数越多时，其边长就越接近外接的圆周长，如图 5-16 所示。

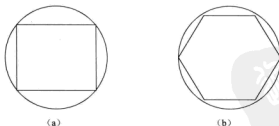


图 5-16 正多边形逼近法

显然图 (b) 中的内接正六边形的周长较图 (a) 中的内接正四边形的周长更加接近于其外接圆的周长。

现在有这样的迭代关系：设单位圆的内接多边形的边长为  $b$ ，边数为  $i$ ，则多边形边数

加倍后,新多边形边长为:  $x = \sqrt{2 - 2\sqrt{1 - b^2}}/2$ 。其中  $\sqrt{\phantom{x}}$  为开方运算。这样,新多边形的周长就为:  $C = 2ix$ , 而原先多边形的周长为:  $C = bi$ 。

如果最初单位圆的内接多边形为正四边形,其边长为  $\sqrt{2}/2$ , 于是  $b = \sqrt{2}/2, i = 4$ 。那么边数加倍后,新八边形的边长为  $\sqrt{2 - 2\sqrt{1 - (\sqrt{2}/2)^2}}/2, i = 8$ 。如此迭代,可求出十六边形,三十二边形……的边长,再乘以相应的边数得到周长  $C$ , 用这个周长  $C$  来近似代替其外接圆的周长,进而求出  $\pi$  的近似值。

利用“正多边形逼近”法求  $\pi$  的伪码算法如下:

```
n ← 要迭代的次数 (迭代次数越多越精确)
div ← 0
b ← sqrt(2)/2, i ← 4 ; 单位圆内接正四边形的初始化
repeat:
    x ← sqrt(2 - 2sqrt(1 - b^2))/2
    b ← x
    i ← i * 2
    div ← div + 1
until div > n
C = ix
π ≈ C / 1
```

方法2: 应用数值概率算法求  $\pi$ 。

数值概率算法又叫做随机数方法。它是利用概率论的思想解决实际问题的。

利用数值概率算法求  $\pi$  的核心思想是: 在一个边长为  $r$  的正方形中, 以一个顶点为圆心,  $r$  为半径作一个  $1/4$  圆, 随机向该正方形中投入点, 其中落入该  $1/4$  圆内的点的概率的4倍就是  $\pi$  的近似值。如图 5-17 所示。

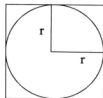


图 5-17 数值概率算法

显然根据几何概率的知识, 随机点落入  $1/4$  圆中概率为  $1/4$  圆面积与边长为  $r$  的正方形面积的比值, 即  $1/4\pi r^2 : r^2$ , 结果为  $1/4\pi$ 。因此, 此概率的4倍就是  $\pi$  的近似值。

应用数值概率算法方法求  $\pi$  的伪码算法如下:

```
inCircle ← 0 ; 记录落入 1/4 圆内的点数
count ← 随机点数 ; 向正方形内随机点的数目, 越多越精确
repeat:
    设置随机点的横坐标 x (0 ~ 100 内)
    设置随机点的纵坐标 y (0 ~ 100 内)
    if x^2 + y^2 ≤ 10000 then
        (x, y) 落入半径为 100 的 1/4 圆中, inCircle ← inCircle + 1.
    endif
    count ← count - 1
until count ≤ 0
π ≈ (inCircle / 随机点数) * 4
```

这里注意, 圆的半径选取与计算结果无关。

下面给出上述两种方法的程序实现。

#### 程序清单 5-15

(1) 用“正多边形逼近”法计算  $\pi$  的近似值。

```

/*----- 5-15.c -----*/
#include <string.h>
#include <stdio.h>
#include <math.h>

double getPI(int n);

void main()
{
    int n;
    double PI;
    printf("Please enter accuracy\n");
    scanf("%d",&n);
    PI=getPI(n);
    printf("The similar value of PI is\n%f\n",PI);
    getchar();
}

double getPI(int n)
{
    int div,i=4;
    double b=sqrt(2)/2.0;
    double c=0.0;
    for(div=0;div<n;div++)
    {
        b=sqrt(2.0-2.0*sqrt(1.0-b*b))*0.5;
        i=i*2;
    }
    c=b*i;
    return c;
}

```

程序的运行结果如图 5-18 所示。

迭代次数为 3

```

Please enter accuracy
3
The similar value of PI is
3.136548

```

迭代次数为 6

```

Please enter accuracy
6
The similar value of PI is
3.141514

```

迭代次数为 11

```

Please enter accuracy
11
The similar value of PI is
3.141593

```

图 5-18 程序 5-15 的运行结果

从运行结果上可以看出，迭代次数越多，所求的  $\pi$  值越精确。

#### 程序清单 5-16

(2) 应用数值概率算法方法求  $\pi$  的近似值。

```

/*----- 5-16.c -----*/
#include <string.h>

```



```

#include <stdio.h>
#include <stdlib.h>

double getPI(int n);

void main()
{
    double PI;
    int n;
    printf("Please enter the number of random points for test\n");
    scanf("%d",&n);
    PI=getPI(n);
    printf("The similar value of PI is\n%f\n",PI);
    getchar();
}

double getPI(int n)
{
    int inCircle=0;
    float x,y;
    int count=n;
    while(count)
    {
        x=random(101);
        y=random(101);
        if(x*x+y*y<=10000)
            inCircle++;
        count--;
    }
    return 4.0*inCircle/n;
}

```

程序的运行结果如图 5-19 所示。

```

Please enter the number of random points for test
10000
The similar value of PI is
3.135600

```

图 5-19 程序 5-16 的运行结果

**注意：**因为数值概率算法方法具有随机性，理论上当随机点数目越大时，所求的结果就越逼近  $\pi$  的真实值，但实际计算时结果精确性并不一定随着随机点数目的增加而一定更加精确。

## 5.16 魔幻方阵

### 【题目要求】

有一种方阵被称为“魔幻方阵”。所谓魔幻方阵是指在  $n \times n$  的矩阵中填写  $1 \sim n^2$  这  $n^2$  个数字，使得它的每一行、每一列以及两个对角线之和均相等。例如三阶魔幻方阵如下：

$$\begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix}$$

它的每一行、每一列以及两个对角线之和均为15。编写一个程序，打印出一种三阶的魔方方阵。

### 【题目分析】

求解魔方阵的一种最直观，最简便的方法是应用穷举法。要求解一个三阶的魔方阵，就是在  $3 \times 3$  的方阵中填写 1~9 这 9 个数字，使得它满足魔方阵的要求。根据排列组合的知识不难理解，如果任意地将 1~9 这 9 个数字填写到  $3 \times 3$  的方阵格子中，共有 9! 种排列方式。而在这 9! 种排列方式中，一定包含着满足魔方阵要求的排列方式。只要找到一种这样的排列方式，就找到了该问题的答案。因此，解决魔方阵问题的算法设计思想是：穷举出  $3 \times 3$  的方阵格子中的 9! 种排列方式，找到一种满足魔方阵要求的排列方式将其输出。

关键在于如何穷举出这 9! 种排列方式。可以这样思考这个问题，如果将  $3 \times 3$  的方阵的每个格子中都设置一个固定的变量，如图 5-20 所示。

<i>i</i>	<i>j</i>	<i>k</i>
<i>l</i>	<i>m</i>	<i>n</i>
<i>o</i>	<i>p</i>	<i>q</i>

图 5-20 将  $3 \times 3$  的方阵的每个格子中都设置一个固定的变量

如果通过一个 9 重的循环，使得每个变量都能够在 1~9 之间有序地变化，并且在变量  $i \sim q$  都互不相等的条件下，就可以穷举出 9! 种排列方式。其实，只要在穷举的过程中找到了一种满足魔方阵要求的排列方式（即每一行、每一列以及两个对角线之和均相等），该穷举运算就可以结束了，并不一定要穷举完，因为这里只要输出一种三阶的魔方阵的即可。其算法描述如下：

```
for(i=1; i<=9; i++)
    for(j=1; j<=9; j++)
        .....
        for(q=1; q<=9; q++)
        {
            if(变量 i~q 互不相等){
                /* 找到 9! 种排列方式之中的 1 种排列方式；*/
                if(该排列方式满足魔方阵要求)
                    输出这种排列方式，并返回该程序；
            }
        }
```

该算法巧妙地应用了 9 个变量  $i \sim q$ ，通过一个 9 重循环和一个“变量  $i \sim q$  互不相等”的条件判断，筛选这 9! 种排列方式，并从中找出一种满足魔方阵要求的排列方式。然后只要按照

$$\begin{pmatrix} i & j & k \\ l & m & n \\ o & p & q \end{pmatrix}$$

的格式将这 9 个变量输出，就求解出了一个三阶的魔方阵。

该算法的优点在于简单直观，易于实现，缺点是时间复杂度较高，效率较低。

### 程序清单 5-17

```
/*----- 5-17.c -----*/
#include "stdio.h"
int match(int i,int j,int k,int l,int m,int n,int o,int p,int q)
```

```

{
    /*判断变量 i~q 是否互不相等, 是则返回 1, 不是则返回 0*/
    if (i!=j&&i!=k&&i!=l&&i!=m&&i!=n&&i!=o&&i!=p&&i!=q
        &&j!=k&&j!=l&&j!=m&&j!=n&&j!=o&&j!=p&&j!=q
        &&k!=l&&k!=m&&k!=n&&k!=o&&k!=p&&k!=q
        &&l!=m&&l!=n&&l!=o&&l!=p&&l!=q
        &&m!=n&&m!=o&&m!=p&&m!=q
        &&n!=o&&n!=p&&n!=q
        &&o!=p&&o!=q
        &&p!=q)

        return 1 ;
    else return 0;
}

int justic(int i,int j,int k,int l,int m,int n,int o,int p,int q)
{
    /*判断变量 i~q 的这种排列是否满足魔幻方阵的要求, 满足返回 1, 不满足返回 0*/
    if (i+j+k == l+m+n && i+j+k == o+p+q &&
        i+l+o == j+m+p && i+l+o == k+n+q
        && i+m+q == k+m+o) return 1;
    else return 0;
}

void getMatrix() {
    int i,j,k,l,m,n,o,p,q;
    for (i=1;i<=9;i++)
        for (j=1;j<=9;j++)
            for (k=1;k<=9;k++)
                for (l=1;l<=9;l++)
                    for (m=1;m<=9;m++)
                        for (n=1;n<=9;n++)
                            for (o=1;o<=9;o++)
                                for (p=1;p<=9;p++)
                                    for (q=1;q<=9;q++)
                                        {
                                            if (match(i,j,k,l,m,n,o,p,q))
                                                if (justic(i,j,k,l,m,n,o,p,q))
                                                {
                                                    printf("%d %d %d\n",i,j,k);
                                                    /*输出结果*/
                                                    printf("%d %d %d\n",l,m,n);
                                                    printf("%d %d %d\n",o,p,q);
                                                    printf("\n");
                                                    return;
                                                }
                                        }
}

main()
{
    getMatrix();
    getch();
}
/*输出一个三阶魔幻方阵*/

```

程序的运行结果如图 5-21 所示。

```

2 7 6
9 5 4
8 3 1

```

图 5-21 程序 5-17 的运行结果

## 5.17 移数字游戏

### 【题目要求】

有这样一个包含 9 个圆圈的数阵，如图 5-22 所示。

将 1~8 这 8 个数随机地填写到该数阵的外层的圆圈中，只剩下中间的一个空圆圈。规定每个数字只能按照数阵中的直线从一个圆圈移动到另一个空的圆圈中。通过若干步的移动，要求将该数阵中的数字移动成为如图 5-23 所示的状态。



图 5-22 包含 9 个圆圈的数阵

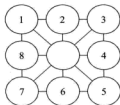


图 5-23 数阵的最终状态

编写一个程序，输出数字每一步的移动过程。

### 【题目分析】

要想清楚地输出数字的每一步的移动步骤，首先需要给这个数阵中的圆圈编号。可以按照图 5-24 所示的方式进行编号。

其中 0# 圆圈初始为空，本题目就是要通过这个空圈，结合一系列的数字移动操作，将数阵中的数字 1~8 排列成如图 5-23 所示的那样。假设要将第 1 号圆圈中的数字移动到第 0 号的空圈中，可输出(1# → 0#)表示。

如何移动该数阵中的数字才能将其排列成如图 5-23 所示的那样呢？最巧妙的方法是：将外圈的数据看成一个“链”。最开始时，由于是任意地向 1# 圈~8# 圈中输入了 1~8 这 8 个数字，因此该链中最初是 8 个无序的数据。按照题目的要求，最终将数字排成图 5-23 所示的状态。这个过程其实就如同一个数列排序的过程，也就是将 1#~8# 单元中的数字按照从小到大的顺序排列。

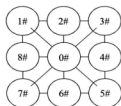


图 5-24 数阵的编号

但是这里需要注意，前面的章节中曾介绍过许多数列的排序算法，而在这里只能使用带有交换性质的排序方法（例如冒泡排序等）实现。因为在本题中，要想将数字最终排成图 5-23 所示的状态，就必须通过数阵中的空圈进行邻接两个数字的交换，这个过程与交换排序的过程是类似的。

除此之外，与一般的数列排序不同的是，本题并不是要得到数列排序的最终结果，而是要通过数列排序的过程模拟出数阵中数字移动的过程。以冒泡排序为例，在冒泡排序中，每次都要进行邻接的两个数据之间的交换。例如将变量 a 和 b 中的内容交换，一般要用到一个临时变量 tmp，通过代码：

```
tmp = a;
a = b;
b = tmp;
```

实现。在这里可以通过代码:

```
printf("(#a → -#0)");
printf("(#b → #a)");
printf("(#0 → #b)");
```

来模拟数阵中数字的移动步骤, 0#单元就相当于冒泡排序中的临时变量 tmp。

本题应用冒泡排序的思想来实现数阵中数字的移动过程。冒泡排序的思想在前面的章节中已有了详细的介绍, 这里不再赘述。

程序清单 5-18

```
/*----- 5-18.c -----*/
#include "stdio.h"

getStep(int m[]) /*显示数阵中数字移动的步骤*/
{
    int i, j, tmp;
    for(i=0; i<7; i++) /*冒泡排序*/
        for(j=0; j<7-i; j++)
            if(m[j]>=m[j+1])
            {
                tmp = m[j];
                m[j] = m[j+1];
                m[j+1] = tmp; /*数据的交换*/
                printf("(#d# --> 0#)\n", j+1);
                printf("(#d# --> #d#)\n", j+2, j+1);
                printf("(0# --> #d#)\n", j+2); /*输出移动步骤*/
            }
    printf("\n");
}

Print(int m[]) /*打印出当前数阵的状态*/
{
    printf(" [%d]--[&d]--[&d]\n", m[0], m[1], m[2]);
    printf(" | %c | %c | \n", 92, 47);
    printf(" [%d]--[ ]--[&d]\n", m[7], m[3]);
    printf(" | %c | %c | \n", 47, 92);
    printf(" [%d]--[&d]--[&d]\n", m[6], m[5], m[4]);
}

main()
{
    int i, m[8];
    printf("Please input 8 integer (1~8) to arrange this matrix\n");
    for(i=0; i<8; i++)
        scanf("%d", &m[i]);
    printf("The initial data matrix is like\n");
    Print(m);
    printf("\nMove Step:\n");
    getStep(m);
    printf("The result of moving is\n");
    Print(m);
    getch();
}
```

**【程序说明】**

本程序中应用函数 `getStep()` 输出数字在数阵中的移动步骤, 数阵的编号如图 5-24 所示。函数 `Print()` 的功能是打印出数阵当前的状态。

程序的运行结果如图 5-25 所示。

```

Please input 8 integer (1~8) to arrange this matrix
1 2 4 3 6 5 7 8
The initial data matrix is like
(1)---(2)---(4)
  | \ | / |
(8)---(  )---(3)
  | / | \ |
(7)---(5)---(6)

Move Step:
<3# --> 8#>
<4# --> 3#>
<8# --> 4#>
<5# --> 8#>
<6# --> 5#>
<8# --> 6#>

The result of moving is
(1)---(2)---(3)
  | \ | / |
(8)---(  )---(4)
  | / | \ |
(7)---(6)---(5)
  
```

图 5-25 程序 5-18 的运行结果

## 5.18 数字的全排列

**【题目要求】**

输入一个数字序列  $\{a_1, a_2, \dots, a_n\}$ , 将该序列进行排列, 并输出每一种排列方式。例如: 输入数字序列: 1, 3, 5。该数列共有 6 种排列方式, 分别为:

1, 3, 5  
1, 5, 3  
3, 1, 5  
3, 5, 1  
5, 1, 3  
5, 3, 1

编写一个 C 程序, 实现数字序列的全排列。

**【题目分析】**

一种简便的方法解决数列的全排列问题就是应用递归的思想。可以将数列的全排列形式化地定义为:

设数列  $R = \{r_1, r_2, \dots, r_n\}$ , 包含  $n$  个元素。定义符号  $R_i = R - \{r_i\}$ 。数列  $R$  的全排列  $\text{Perm}(R)$  定义为:

当  $n=1$  时,  $\text{Perm}(R) = \{r\}$ 。其中,  $r$  为数列  $R$  中的唯一元素。

当  $n>1$  时,  $\text{Perm}(R)$  由一组排列:  $(r_1)\text{Perm}(R_1), (r_2)\text{Perm}(R_2), \dots, (r_n)\text{Perm}(R_n)$  构成。这显然是一个递归的定义, 依此递归定义, 可以设计出算法 `Perm` 如下:

```

Perm(int a[], int n, int s, int r[], int m){
    int i, j, k, flag=0;
    int b[max];          /*数组 b 存放子序列*/
  
```

```

for(i=0;i<n;i++){
    flag = 1;
    r[s] = a[i];          /*复制数列 a 中第 i 个元素到数组 r*/
    j = 0;
    for(k=0;k<n;k++){    /*产生子序列 b*/
        if(i != k){
            b[j] = a[k];
            j++;
        }
        Perm(b,n-1,s+1,r,m); /*全排列子序列 b*/
    }
    if(flag == 0){        /*得到一种排列结果*/
        for(k=0;k<m;k++){
            printf("%d",r[k]); /*输出存放排列结果的数组 r 中的内容*/
            printf("\n");
        }
    }
}

```

在这个算法中, 参数  $a[]$  为待排列的数组; 参数  $n$  为当前数组中的元素的个数, 这个数会随着递归的调用发生变化。确切地说,  $n$  为当前子序列中元素的个数。参数  $r[]$  为排列后的数组, 即它存放数组  $a$  的每一种排列结果。参数  $s$  为数组  $r$  的下标。参数  $m$  为原始数组  $a$  中的元素的个数, 这个数不会发生改变。

在算法中, 数组  $b[]$  的作用是存放数列  $a$  中除去  $a[i]$  的其他元素, 也就是每一层递归操作生成的子序列。将数组  $b$  作为递归调用  $Perm$  的实参进行传递, 此时序列中元素的个数  $n$  减 1, 存放排列结果的数组  $r$  的下标加 1。变量  $flag$  为一个标志变量。当  $flag$  为 0 时, 表明已经得出数列  $a$  的一种排列结果, 存放在数组  $r$  中, 因此输出  $r$  中的结果 ( $m$  个元素)。当  $flag$  为 1 时, 表明尚未得到排列结果, 需要继续递归求解。

#### 程序清单 5-19

```

/*----- 5-19.c -----*/
#include "stdio.h"
#define max 100

Perm(int a[], int n, int s, int r[], int m){
    int i, j, k, flag=0;
    int b[max];          /*数组 b 存放子序列*/
    for(i=0;i<n;i++){
        flag = 1;
        r[s] = a[i];      /*复制数列 a 中第 i 个元素到数组 r*/
        j = 0;
        for(k=0;k<n;k++){ /*产生子序列 b*/
            if(i != k){
                b[j] = a[k];
                j++;
            }
            Perm(b,n-1,s+1,r,m); /*全排列子序列 b*/
        }
        if(flag == 0){    /*得到一种排列结果*/
            printf("\n");
            for(k=0;k<m;k++){
                printf("%d",r[k]); /*输出存放排列结果的数组 r 中的内容*/
                printf("\n");
            }
        }
    }
}

```

```

    }
}

main()
{
    int a[max] , r[max];
    int i,n;
    printf("Please input the number of digit in the array\n");
    scanf("%d",&n);          /*输入待排列的数列中元素的个数*/
    printf("Please input a string for array\n");
    for(i=0;i<n;i++)          /*输入数列中的元素（整数）*/
        scanf("%d",&a[i]);
    Perm(a,n,0,r,n);          /*全排列操作*/
    getch();
}

```

### 【程序说明】

为了实现方便，这里规定数组  $a$  的最大容量为  $\text{max}=100$  个整数，也就是说输入的数字序列的元素个数最多不能超过 100 个。

程序的运行结果如图 5-26 所示。

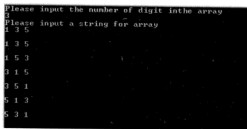


图 5-26 程序 5-19 的运行结果

## 5.19 完全数

### 【题目要求】

如果一个数恰好等于它的因子之和，那么这个数就被称为完全数。例如 6 的因子为 1, 2, 3，而  $6=1+2+3$ ，因此 6 是一个完全数。求出 1000 以内的完全数。

### 【题目分析】

本题最为直接的解法就是利用穷举法在 1~1000 以内判断每个数是否是完全数，如果是完全数就输出之。问题的关键就是如何判断一个数  $a$  是否是完全数。它要分为两个步骤完成：

(1) 求出  $a$  的所有因子  $a_1, a_2 \dots a_n$  的和  $\text{sum}$ 。

(2) 判断  $a_1+a_2+\dots+a_n$  的和  $\text{sum}$  是否等于  $a$ ，如果  $\text{sum}$  等于  $a$  则  $a$  是完全数，否则  $a$  不是完全数。

求  $a$  的因子和  $\text{sum}$  的方法也可以应用穷举法。在  $1 \sim a-1$  之间穷举出每一个整数，判断它是否是  $a$  的因子，即判断它是否可以整除  $a$ ，再通过一个变量  $\text{sum}$  将  $a$  的所有因子累加求和，计算出  $a$  的所有因子的和。其算法如下：



```

int factorSum(a)           /*求 a 的因子和*/
{
    int i, sum = 0;
    for(i=1; i<a; i++)
        if(a%i == 0)      /*i 是 a 的一个因子*/
            sum = sum + i; /*通过变量 sum 累加和*/
    return sum;           /*返回 a 的因子的和*/
}

```

如果  $a$  的所有因子的和等于  $a$  本身, 则  $a$  是完全数, 否则  $a$  不是完全数。

程序清单 5-20

```

/*----- 5-20.c -----*/
#include "stdio.h"

int factorSum(a)           /*求 a 的因子和*/
{
    int i, sum = 0;
    for(i=1; i<a; i++)
        if(a%i == 0)      /*i 是 a 的一个因子*/
            sum = sum + i; /*累加和*/
    return sum;           /*返回 a 的因子的和*/
}

int perfextnumber(int a)   /*判断 a 是否是完全数*/
{
    if(a == factorSum(a)) return 1;
    else return 0;
}

main()
{
    int a;
    printf("There are following perfect numbers 1~1000 are:\n");
    for(a=1; a<=1000; a++)
    {
        if(perfextnumber(a)) /*寻找 1~1000 以内的完全数*/
            printf("%d ", a);
    }
    getch();
}

```

程序的运行结果如图 5-27 所示。



```

There are following perfect numbers 1~1000 are:
6 28 496

```

图 5-27 程序 5-20 的运行结果

## 5.20 亲 密 数

### 【题目要求】

如果整数  $A$  的全部因子 (包括 1, 不包括  $A$  本身) 之和等于  $B$ , 并且整数  $B$  的全部因子 (包括 1, 不包括  $B$  本身) 之和等于  $A$ , 则称整数  $A$  和  $B$  为亲密数。求解 3000 以内的

全部亲密数。

### 【题目分析】

本题目与上一题有些类似，都需要求出某一个数的因子之和。然而，寻找亲密数的过程要比寻找完全数的过程复杂一些。这是因为判断一个数是否是完全数，只要看它的因子之和是否等于该数本身即可，如果相等，则该数是完全数，否则不是完全数。这样应用穷举法在1~1000之间对每个数都进行一次判断，即可找出全部的完全数。然而，亲密数是两个数之间的关系。也就是说在使用穷举法寻找亲密数时，不能立刻判断一个整数 $a$ 是否有亲密数，更不知道它的亲密数是多少。

可以采用一种具有选择性质的算法来查找一定范围内的亲密数，这种方法效率不一定很高，但是简单直观。

在一个数集 $\{x_1, x_2, x_3, \dots, x_n\}$ 中寻找亲密数，可以针对某一个元素，在其他的元素中寻找该元素的亲密数。为了使寻找亲密数的效率更高，可以逻辑上划分两个子集A和B，最初数集中所有的元素都存放在子集B中，如图5-28所示。

然后在集合B中任选一个元素放入集合A中，同时在集合B中删去该元素。然后在集合B中的其他元素中寻找刚刚放入集合A中的那个元素的亲密数，如图5-29所示。

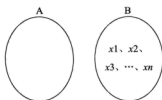


图 5-28 子集A和B的初始状态

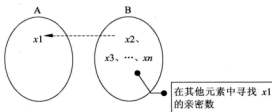


图 5-29 将 $x_1$ 放入集合A，并寻找 $x_1$ 的亲密数

如果在集合B中找到了刚刚放入集合A中的那个元素的亲密数，则将该亲密数（不妨设为 $x_i$ ）也放入集合A中，同时在集合B中删去 $x_i$ ，如图5-30所示。

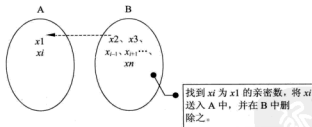


图 5-30 找到亲密数 $x_i$

然后重复上述操作，在集合B中任选一个元素，放入集合A中，在集合B中寻找它的亲密数……。

如果在集合B中没有找到刚刚放入集合A中的那个元素的亲密数，则重复上述操作，在集合B中任选一个元素，放入集合A中，在集合B中寻找它的亲密数……。

上述操作直到子集B中的元素全部放到子集A中为止。

这样就可以避免许多不必要的查询和判断。这是因为由亲密数的定义不难理解, 亲密数配对是唯一的。另外如果一个元素  $x_i$  在集合  $\{x_1, x_2, x_3, \dots, x_n\}$  中没有亲密数, 那么任何一个其他的元素  $x_j (j \neq i)$ , 的亲密数都不会是  $x_i$ 。这样就保证了上述寻找亲密数的方法的正确性。

基于上述思想, 在一个有限的范围  $\{1, 2, \dots, 3000\}$  中寻找所有的亲密数的算法可以描述如下:

```
将 1, 2, ..., 3000 各元素存放在 x[1...3000] 中;
for(i=1; i<=3000; i++)
    if(x[i] 没有找到其亲密数, 即 x[i] 仍在集合 B 中) {
        for(j=i+1; j<=n; j++)
            if(x[j] 为 x[i] 的亲密数)
                输出亲密数 (x[i], x[j]), 并记录 x[j] 已经找到其亲密数;
    }
```

在该算法中, 可将集合  $\{1, 2, \dots, 3000\}$  中的每个元素存放到一个数组  $x[n]$  中, 然后顺序查找。如何判断  $x[j]$  是否为  $x[i]$  的亲密数呢? 可以分别求出它们的因子之和, 然后比较  $x[j]$  的因子之和是否等于  $x[i]$ ;  $x[i]$  的因子之和是否等于  $x[j]$ 。

但是这样是很消耗时间的, 因为按照这种方法比较亲密数, 寻找 3000 以内的亲密数大约需要计算九百万次的因子和, 计算因子和的过程是比较费时的, 而绝大多数的计算又都是重复和多余的, 这样的时间消耗是难以令人接受的。因此合理的做法是: 事先全部计算出  $1 \sim 3000$  以内所有整数的因子的和, 存放数组  $x[]$  中, 这样  $x[i]$  中存放的是  $i$  的因子之和, 寻找  $\{1, 2, \dots, 3000\}$  范围中所有的亲密数的算法就变为:

```
将 1, 2, ..., 3000 各元素的因子之和存放在 x[1...3000] 中;
for(i=1; i<=3000; i++)
    if(i 没有找到其亲密数, 即 i 仍在集合 B 中) {
        for(j=i+1; j<=n; j++)
            if( j 为 i 的亲密数 )
                输出亲密数 (i, j), 并记录 j 已经找到其亲密数;
    }
```

其中判断  $j$  和  $i$  是否为亲密数的方法是: 判断  $x[j]$  是否等于  $i$ , 以及判断  $x[i]$  是否等于  $j$ 。这样就省去了每次都要重复计算每个数的因子之和的麻烦。

程序清单 5-21

```
/*----- 5-21.c -----*/
#include "stdio.h"

int factorSum(a)          /*求 a 的因子和*/
{
    int i, sum = 0;
    for(i=1; i<a; i++)
        if(a%i == 0)      /*i 是 a 的一个因子*/
            sum = sum + i; /*通过变量 sum 累加和*/
    return sum;           /*返回 a 的因子的和*/
}
```

```

int isfriend(int a,int b,int i,int j)
    /*判断 a 与 b 是否是亲密数，是亲密数返回 1，否则返回 0*/
{
    if(a==j && b== i )return 1;
    else return 0;
}

friendly()          /*寻找 1~3000 范围内的亲密数*/
{
    int i , j, x[3001];
    for(i=1;i<=3000;i++)
        x[i] = factorSum(i);
    for(i=1;i<=3000;i++)
    {
        if(x[i]!=-111){
            for(j=i+1;j<=3000;j++)
                if(isfriend(x[i],x[j],i,j))
                {
                    printf("(%d,%d) ",i,j);
                    x[j] = -111;          /*表示 j 已经找到亲密数*/
                }
        }
    }
}

main()
{
    printf("There are following friendly numbers from 1 to 3000\n");
    friendly();
    getche();
}

```

程序的运行结果如图 5-31 所示。

```

There are following friendly numbers from 1 to 3000
<220,284> <1184,1210> <2620,2924>

```

图 5-31 程序 5-21 的运行结果

## 5.21 数字翻译器

### 【题目要求】

输入一个正整数 N，输出它的英文表达。例如：输入 1，输出 one；输入 12，输出 twelve；输入 135，输出 one hundred thirty five…。编写程序实现之。

### 【题目分析】

解决这道题的关键是弄清阿拉伯数字与英文表达的对应关系。

首先数字 1~19 中每个数都对应一个英文的单词。

数字 20~99，每个数字对应的英文单词都是该数的十位数表达加（十单位）上个位数表达的组，例如 21 的英文表示为 twenty one，99 的英文表示为 ninety nine…。

数字 100~999，每个数对应的英文单词都是该数的百位数表达（百单位）加上上面所

述的1~99的表达方式,例如101的英文表示为 one hundred one, 999 的英文表示为 nine hundred ninety nine...

数字1000~9999,每个数对应的英文单词都是该数的千位数表达(千单位)加上上面所述的1~999的表达方式,例如1001的英文表示为 one thousand one, 9999 的英文表示为 nine thousand nine hundred ninety nine...

数字10000~999999,因为英文中没有“万”和“十万”的表达方式(单位),因此要将它们划归为“千”的表示形式(单位)。具体做法是将这个范围内的数整除1000,得到的商一定在10~999之间,将这个商用上述的英文表示方法表示,再加上 thousand,后面的数用上述的1~999的表达方式表达。例如123456的英文表示为: one hundred twenty three thousand four hundred fifty six。

后面的数字表达仍然存在类似上面的规律,为了简化题目的难度,这里仅以999999以内的整数为例,实现数字的翻译,读者可以在此基础上考虑更大的数字翻译。

找到了阿拉伯数字与英文表达的对应关系的规律,就不难设计出解决该题的算法。这里要做的是将输入的数据按千位表达(thousand)、百位表达(hundred)、十位和个位表达进行划分,然后从高位到低位逐级翻译成英文表达。算法描述如下:

输入一个数(1~999999的整数)N;

过程A: 翻译千位数

a=N/1000;

if(a != 0) 则a一定在10~999之间,调用子过程B按规则翻译a,并打印thousand;

a=N%1000;

if(a != 0) 则a一定在1~999之间,调用子过程B按规则翻译a。

过程B: 翻译百位数

b=a/100;

if(b != 0) 调用子过程C按规则翻译b,并打印hundred;

b=a%100;

f(b != 0) 调用子过程C按规则翻译b。

else 程序结束

过程C: 翻译十位数和个位数

if(b<=19) 直接翻译c;

else c=b/10,按规则翻译十位数c,再按规则翻译个位数b%10。

#### 程序清单 5-22

```
/*----- 5-22.c -----*/
#include "stdio.h"
char data_1[19][10]={"one","two","three","four",
                    "five","six","seven","eight",
                    "nine","ten","eleven","twelve",
                    "thirteen","fourteen","fifteen","sixteen",
                    "seventeen","eighteen","nineteen"};

char data_2[8][7]={"twenty","thirty","forty","fifty",
                  "sixty","seventy","eighty","ninty"};

translation_A(long N)
{
    /*翻译千位数*/
    long a;
    if(N==0) {printf("Zero\n");return;}
    a = N/1000;
    if(a!=0) {
```

```

        translation_B(a);
        printf("thousand ");
    }
    a = N%1000;
    if(a!=0)
        translation_B(a);
}

translation_B(long a)
{
    long b;
    b=a/100;
    if(b!=0){
        translation_C(b);
        printf("hundred ");
    }
    b = a%100;
    if(b!=0)
        translation_C(b);
}

translation_C(long b)
{
    long c;
    if(b<=19)
        printf("%s ",data_1[b-1]);
    else{
        c = b/10;
        printf("%s ",data_2[c-2]);
        c=b%10;
        if(c!=0)
            printf("%s ",data_1[c-1]);
    }
}

main()
{
    long N;
    printf("Please input a longeger from 0~999999\n");
    scanf("%ld",&N);
    translation_A(N);
    getch();
}

```

程序的运行结果如图 5-32 所示。

```

Please input a longeger from 0~999999
250061
two hundred fifty thousand sixty one

```

图 5-32 程序 5-22 的运行结果

## 5.22 递归实现数制转换

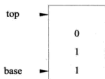
### 【题目要求】

应用递归的方法实现一个数制转换器，它可以将输入的二进制数转换为十进制表达。

## 【题目分析】

问题的关键在于如何应用递归的方法实现数制转换的功能。一般实现二进制数转换为十进制数的功能,通常的做法是应用一个栈数据结构。具体的做法是,先将二进制数从高位到低位依次入栈,再从栈顶取数分别与  $2^0, 2^1 \dots 2^n$  相乘,并累加求和,直到将栈中的数据取完为止。最终得到的累加和就是原二进制数的十进制形式。例如将二进制数 110 转换为十进制数表示,应用栈实现的过程如图 5-33 所示。

(1) 将二进制数 110 从高位到低位依次入栈



(2) 再从栈顶取数分别与  $2^0, 2^1 \dots 2^n$  相乘,并累加求和

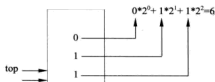


图 5-33 应用栈实现数制转换

这里之所以要用到栈结构,就是利用了它先进后出的特性。因为要将二进制数转换为十进制数,一般都要从低位开始进行转换。

其实栈的操作往往可以用递归的方法进行等价。使用栈操作效率比较高,但是实现起来比较麻烦。在对程序的运行效率要求不高的情况下,使用递归的方法代替栈操作会更加易于实现。

二进制数转换为十进制表达的递归算法如下:

```

biTOde(int n,int *sum,int *m)
{
    char c;
    scanf("%c",&c);
    if(c!='\n'){
        *m = *m + 1;
        biTOde(n+1,&(*sum),&(*m));
    }
    if(c == '1') *sum = *sum + pow(2,(*m)-n-1);
    /*从二进制串的低位开始累加和*/
}

```

算法中参数  $sum$  用来记录累加和。参数  $m$  随着递归的深入不断加 1,并且递归结束  $m$  的值也不会发生改变。因为参数  $sum$  和  $m$  都要在递归中保持值的不变,因此采用指针传递方式。参数  $n$  的值随着递归的深入不断加 1,但是每一层的递归  $n$  的取值都不相同。不难理解,在将二进制数转换为十进制表示时,2 的指数应为  $(*m)-n-1$ 。输入的二进制数以字

符形式存储,当输入字符#时,表示二进制字符串输入结束。在调用该算法时,参数的初值都应当为0。应用该算法将二进制数110转换为十进制数的过程如图5-34所示。

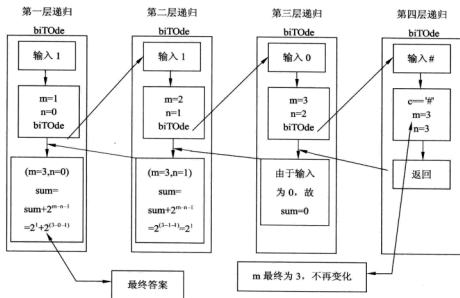


图 5-34 递归方法将二进制数 110 转换为十进制数的过程

如图 5-34 所示,程序按照箭头所指的方向执行,最终得到  $sum=2^2+2^1$  即为二进制数 110 的十进制数值。

程序清单 5-23

```

/*----- 5-23.c -----*/
#include "stdio.h"
#include "math.h"

biTode(int n,int *sum,int *m)
{
    char c;
    scanf("%c",&c);
    if(c!='#'){
        *m = *m + 1;
        biTode(n+1,&(*sum),&(*m));
    }
    if(c == '1') *sum = *sum + pow(2,(*m)-n-1);
}

main()
{
    int sum = 0,m=0;
    printf("Please input a binary number ending for '#'\n");
    biTode(0,&sum,&m);
    printf("The Decimal Decimal digit is\n");
}

```



```
printf("%d ",sum);
getche();
}
```

程序的运行结果如图 5-35 所示。

```
Please input a binary number ending for '#'
11101#
The Decimal Decimal digit is
29
```

图 5-35 程序 5-23 的运行结果

**注意：**不难看出，利用递归方法解决进制转换问题相比应用栈的方法要更加易于实现。

## 5.23 谁在说谎

### 【题目要求】

3 个嫌疑犯在法官面前各执一词，甲说：乙在说谎；乙说：丙在说谎；丙说：甲乙两人都在说谎。法官为了难，甲乙丙三人到底谁在说谎，谁说的是真话？

### 【题目分析】

这是一道十分有趣的逻辑推理问题。解决这类逻辑推理问题最简单直观的方法是使用穷举法。甲乙丙三人中任何人所说的话无外乎有两种可能，即真和假。如果用 1 表示真，用 0 表示假，甲乙丙三人所说的话的真假情况限定在以下范围内。

甲	乙	丙
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

以上的 8 种可能构成了本题的解空间。也就是说，如果本题有解，那么该解一定存在于以上这 8 种解之中。下面的任务就是在这 8 种解中搜索出真正的答案。

以上 8 种可能的解哪个才是真正的答案呢？关键是看它们是否符合说话者所说的话的逻辑。

设甲乙丙三人所说的话的真假用变量  $a$ 、 $b$ 、 $c$  表示。 $a$ 、 $b$ 、 $c$  的取值为 1 或 0。

按照甲乙丙所说的话，存在如下逻辑关系：

如果  $a=1$ ，则  $b=0$ 。

如果  $a=0$ ，则  $b=1$ 。

如果  $b=1$ , 则  $c=0$ 。

如果  $b=0$ , 则  $c=1$ 。

如果  $c=1$ , 则  $a+b=0$ 。

如果  $c=0$ , 则  $a+b \neq 0$ 。

因此, 要判断 3 个人所说的话的真伪, 必须使他们的话符合以上逻辑。即满足以下关系:

$$[(a=1 \&\& b=0) \vee (a=0 \&\& b=1)] \&\& [(b=1 \&\& c=0) \vee (b=0 \&\& c=1)] \&\& [(c=1 \&\& a+b=0) \vee (c=0 \&\& a+b \neq 0)]$$

$$\Rightarrow (a \&\& !b \vee !a \&\& b) \&\& (b \&\& !c \vee !b \&\& c) \&\& (c \&\& a+b=0 \vee !c \&\& a+b \neq 0)$$

凡是不满足以上的逻辑表达式判断的组合都不是正确的答案, 只有所得结果为真的组合才是本题的正确答案。

程序清单 5-24

```

/*----- 5-24.c -----*/
#include "stdio.h"
main()
{
    int a,b,c;
    for(a=0;a<=1;a++)
        for(b=0;b<=1;b++)
            for(c=0;c<=1;c++)
                if((a&&!b || !a&&b) && (b&&!c || !b&&c) && (c&&a+b==0
                || !c&&a+b!=0))
                {
                    printf("甲 told a %s\n",a?"truth":"lie");
                    printf("乙 told a %s\n",b?"truth":"lie");
                    printf("丙 told a %s\n",c?"truth":"lie");
                }
    getch();
}

```

程序的运行结果如图 5-36 所示。

```

甲 told a lie
乙 told a truth
丙 told a lie

```

图 5-36 程序 5-24 的运行结果

## 第6章 数学趣题（二）

本章继续第5章的内容，向读者介绍一些有趣的数学问题，同时给出详尽的分析过程和程序源代码。

### 6.1 连续整数固定和问题

#### 【题目要求】

编写一个程序，找出一个数的全部连续整数固定和。所谓一个数  $n$  的连续整数固定和，就是指存在  $a_1, a_2, \dots, a_n$ ，其中  $a_{i+1}$  比  $a_i$  大 1，使得  $a_1 + a_2 + \dots + a_n = n$ 。这样  $a_1, a_2, \dots, a_n$  称为  $n$  的一个连续整数固定和。例如 27 的全部连续整数固定和有 3 组： $2+3+\dots+7=27$ ； $8+9+10=27$ ； $13+14=27$ 。本题就是要找出任意输入的整数  $n$  的全部连续整数固定和。

注意，有些数也可能不存在连续整数固定和，例如 4 和 16 等。

#### 【题目分析】

这道题最直观的解法是应用穷举法。由于是要找出  $n$  的连续整数固定和，因此它的搜索范围可以锁定在数列  $1 \sim n$  之间。也就是说，如果整数  $n$  存在连续整数固定和，它的连续整数固定和一定是数字串  $1, 2, \dots, n$  的子串，如图 6-1 所示。



图 6-1 连续整数固定和示意

如图 6-1 所示，子串之间也可能发生重叠。因此只要穷举出集合  $\{1, 2, \dots, n\}$  的所有连续子集合，再判断连续子集合的每个元素的和是否等于  $n$ ，就可以得到整数  $n$  的全部连续整数固定和。

关键的问题在于如何进行穷举搜索。我们可以采取固定起点，扫描数列元素的方法进行连续整数固定和的搜索。具体做法是：

首先在数列  $1, 2, 3, \dots, n$  中确定一个起点  $a_1$ ，然后顺次向后累加，即  $a_1 + a_2 + \dots$ ，直到所加的总和  $(a_1 + a_2 + \dots + a_i)$  大于或等于整数  $n$ （要求连续整数固定和的整数）为止。如果所加的总和恰好等于整数  $n$ ，则从  $a_1$  到  $a_i$  的累加和即为整数  $n$  的一个连续整数固定和，否则说明以  $a_1$  为起点的连续整数串不存在整数  $n$  的连续整数固定和。不难理解，只要判断了

起点为 1, 2, ...,  $n-1$  的全部数列子串, 就可以从中得出  $n$  的全部连续整数固定和。这个算法思想的代码描述如下:

```
void cntnsIntSum(int n)
{
    int i, sum=0, j;
    for(i=1; i<n; i++)          /*控制起点的选择, 从1到n-1*/
    {
        j = i-1;
        while(sum<n)           /*从起点向后顺序累加*/
        {
            j++;
            sum = sum + j;
        }
        if(sum == n)
        {
            printf("%d+...+%d = %d\n", i, j, n);
        }
        sum = 0;
    }
}
```

外层的 for 循环控制起点的选择, 即起点的设置从 1 开始一直到  $n-1$  为止。内层的 while 循环是从起点开始, 顺序向后累加, 直到总和大于或等于整数  $n$  为止。该函数可将  $n$  的全部连续整数固定和显示在屏幕上。

程序清单 6-1

```
/*----- 6-1.c -----*/
#include "stdio.h"

void cntnsIntSum(int n)
{
    int i, sum=0, j;
    for(i=1; i<n; i++)
    {
        j = i-1;
        while(sum<n)
        {
            j++;
            sum = sum + j;
        }
        if(sum == n)
        {
            printf("%d+...+%d = %d\n", i, j, n);
        }
        sum = 0;
    }
}

main()
{
    int n;
    printf("Please input a integer\n");
    scanf("%d", &n);
    cntnsIntSum(n);
    getche();
}
```

本程序的运行结果如图 6-2 所示。

```
Please input a integer
27
2+...+7 = 27
8+...+10 = 27
13+...+14 = 27
```

图 6-2 程序 6-1 的运行结果

## 6.2 表示成两个数的平方和

### 【题目要求】

已知一个正整数  $N$ ，编写一个程序，找出所有满足  $X^2+Y^2=N$  的正整数对  $X$  和  $Y$ 。

### 【题目分析】

这道题可以用穷举法来求解。只要规定好正整数  $X$  和  $Y$  的范围，然后在这个范围内任意搭配  $X$  和  $Y$ ，判断  $X^2+Y^2$  是否等于  $N$ ，就可以筛选出所有符合要求的正整数对  $X$  和  $Y$ 。关键的问题在于确定正整数  $X$  和  $Y$  的范围。一种最为保险的办法是将  $X$  和  $Y$  的范围确定在  $[1, N]$  中的全部正整数，因为无论如何， $[1, N] \times [1, N]$  这  $N^2$  种数对组合也一定能够覆盖全部的解。但是这样做会产生大量的冗余的计算，使得程序的效率很低。因为这种穷举法的时间复杂度为  $O(n^2)$ ，因此如果  $N$  的值较大，算法会非常耗时。不难理解，无论正整数  $X$  的取值，还是  $Y$  的取值，它们的大小都不可能超过  $\sqrt{N}$ 。因此，可将  $X$  和  $Y$  的范围确定在  $[1, \sqrt{N}]$  的全部正整数中，这样  $X$  和  $Y$  最多有  $N$  种组合形式，因此最多判断  $N$  次  $X^2+Y^2$  是否等于  $N$  就可以找出所有符合要求的正整数对  $X$  和  $Y$  来。下面给出具体的算法描述：

```
void getXY(int N)
{
    int x, y;
    for(x=1; x<sqrt(N); x++)
        for(y=1; y<sqrt(N); y++)
        {
            if(x*x+y*y == N)
            {
                printf("%d^2+%d^2=%d\n", x, y, N);
            }
        }
}
```

算法中外层循环控制变量  $X$  的搜索，内层循环控制变量  $Y$  的搜索，最多循环  $N$  次便可找到所有满足  $X^2+Y^2=N$  的正整数对  $X$  和  $Y$ 。但是这个算法是将  $X$  和  $Y$  分别看作两个独立的变量而进行搜索的。例如使用该算法求解满足  $X^2+Y^2=100$  的  $X$  和  $Y$ ，会得到两对答案  $X=6, Y=8$  和  $X=8, Y=6$ 。但是如果将  $(X, Y)$  看作一个整数对，那么  $X=6, Y=8$  和  $X=8, Y=6$  其实是等价的，也就是说正整数对  $(6, 8)$  满足  $X^2+Y^2=100$ 。如果要实现这样的输出结果，也就是将  $(X, Y)$  看作一个整数对输出，可将上述算法稍加修改如下：

```
void getXY(int N)
```

```

{
    int x,y;
    for(x=1;x<sqrt(N);x++)
        for(y=x;y<sqrt(N);y++)          /*y 从 x 开始向后搜索*/
        {
            if(x*x+y*y == N)
            {
                printf("%d^2+%d^2=%d\n",x,y,N);
            }
        }
}

```

其实  $X$  和  $Y$  的范围没有必要都确定在  $[1, \sqrt{N}]$  的全部正整数中, 因为那样做仍会产生重复。像上述算法中所示, 外层循环控制变量  $X$ , 它的搜索范围为  $1 \sim \sqrt{N}$ , 而内层循环控制的变量  $Y$  的搜索范围为  $X \sim \sqrt{N}$ , 这样就会避免上述的重复现象。可以通过下面这个矩阵式来解释这个算法的原理。

$$\begin{pmatrix}
 (1,1) & (1,2) & (1,3) & (1,4) & (1,5) \\
 (2,1) & (2,2) & (2,3) & (2,4) & (2,5) \\
 (3,1) & (3,2) & (3,3) & (3,4) & (3,5) \\
 (4,1) & (4,2) & (4,3) & (4,4) & (4,5) \\
 (5,1) & (5,2) & (5,3) & (5,4) & (5,5)
 \end{pmatrix}$$

通过这个矩阵可以看出, 如果外层循环和内层循环都是  $1 \sim 5$ , 那么  $X$  和  $Y$  的组合形式为 25 种, 即矩阵中的每一个元素。但是该矩阵为一个对称的矩阵, 因此若将  $(X, Y)$  看作一个整数对来搜索, 只需要搜索该矩阵的上三角或下三角 (包含对角线) 就可以了, 也就是说  $X$  和  $Y$  的组合形式为 15 种。如果外层循环控制搜索该矩阵的“行”, 内层循环就负责控制搜索该矩阵的“列”, 那么内层循环的起始位置应为当前外层循环的值。即如果外层变量  $X$  的搜索范围为  $1 \sim \sqrt{N}$ , 则内层变量  $Y$  的搜索范围为  $X \sim \sqrt{N}$ 。

程序清单 6-2

```

/*----- 6-2.c -----*/
#include "stdio.h"
#include "math.h"

void getXY(int N)
{
    int x,y;
    for(x=1;x<sqrt(N);x++)
        for(y=x;y<sqrt(N);y++)
        {
            if(x*x+y*y == N)
            {
                printf("%d^2+%d^2=%d\n",x,y,N);
            }
        }
}

main()
{
    int N;
    printf("Please input a integer N\n");
}

```

```
scanf("%d",&N);
getXY(N);
getche();
}
```

本程序的运行结果如图 6-3 所示。

```
Please input a integer N
100
6^2+8^2=100
```

图 6-3 程序 6-2 的运行结果

### 6.3 具有特殊性质的数

#### 【题目要求】

有这样一个 4 位数  $abcd$ ，它具有这样的性质  $abcd=(ab+cd)^2$ 。其中， $ab$  和  $cd$  为两个 2 位数，求这个 4 位数  $abcd$ 。

#### 【题目分析】

一个 4 位数  $abcd$  是由它各个位上的数字组成的，除了最高位数字  $a$  以外，每个数字的取值范围都是  $0\sim 9$ 。 $a$  的取值范围为  $1\sim 9$ 。因此不难想象，将个、十、百三位数每位数字从 0 遍历到 9，将千位数字从 1 遍历到 9，由此组合而成的 9000 个 4 位数就是该题的解空间。也就是说，如果存在这样一个 4 位数  $abcd$ ，使得  $abcd=(ab+cd)^2$ ，那么该数一定存在于  $1000\sim 9999$  之间。

在设计算法时，可以使用 4 个变量  $a$ 、 $b$ 、 $c$ 、 $d$  分别存放该 4 位数的 4 个数字。然后通过一个四重循环搜索整个解空间，从中找出符合题目要求的答案。算法描述如下：

```
void func()
{
    int a,b,c,d;
    for(a=1;a<=9;a++)
        for(b=0;b<=9;b++)
            for(c=0;c<=9;c++)
                for(d=0;d<=9;d++)
                {
                    if(a*1000+b*100+c*10+d == ((a*10+b) + (c*10+d))*
                        ((a*10+b) + (c*10+d)))
                        printf("%d%d%d%d\t",a,b,c,d);
                }
}
```

变量  $a$  为最高位的千位上的数字，它的搜索范围是  $[1, 9]$ ，内层循环分别控制百位数字  $b$ ，十位数字  $c$  和个位数字  $d$  的搜索。只要满足条件  $a*1000+b*100+c*10+d == ((a*10+b) + (c*10+d))*((a*10+b) + (c*10+d))$  的，即符合题目要求  $abcd=(ab+cd)^2$ ，在程序中将其输出。

当然本题还有另外一种解决方法，即通过搜索  $1000\sim 9999$  这 9000 个数，从中寻找出符合题目要求的 4 位数来，这样做也是可以的，它只需要一重循环即可。但是该方法需要将每个数字“平分二段”，例如将整数 1234 平分分为两个 2 位数 12 和 34，然后才能进行条

件判断, 因此不如本算法来得直观方便。有兴趣的读者可以尝试另外一种方法。

程序清单 6-3

```

/*----- 6-3.c -----*/
#include "stdio.h"

void func()
{
    int a,b,c,d;
    for(a=1;a<=9;a++)
        for(b=0;b<=9;b++)
            for(c=0;c<=9;c++)
                for(d=0;d<=9;d++)
                {
                    if(a*1000+b*100+c*10+d == ((a*10+b) + (c*10+d))*
                        ((a*10+b) + (c*10+d)))
                        printf("%d%d%d%d\t",a,b,c,d);
                }
}

main()
{
    printf("There are following numbers according with the condition\n");
    func();
    getch();
}

```

本程序的运行结果如图 6-4 所示。

```

There are following numbers according with the condition
2025    3025    9801

```

图 6-4 程序 6-3 的运行结果

## 6.4 验证角谷猜想

### 【题目要求】

角谷猜想的内容为: 任意给定一个自然数, 若它为偶数则除以 2, 若它为奇数则乘 3 加 1, 得到一个新的自然数, 按照这样的计算方法计算下去, 若干次后得到的结果必然为 1。编写程序对角谷猜想的正确性加以验证。

### 【题目分析】

验证角谷猜想的方法很简单, 可以按照题目中给定的角谷猜想的内容直接编写程序, 对任意给定的自然数通过一个循环反复进行判断操作 (若它为偶数则除以 2, 若它为奇数则乘 3 加 1, 得到一个新的自然数), 直到结果为 1 为止。在这个过程中, 可以设定一个阈值以规定循环的上限次数, 如果循环在阈值范围内结束, 即说明该数符合角谷猜想; 如果循环超过了预先设定的阈值, 则说明要调整阈值的选取。这种方法只能验证角谷猜想的正确性, 而不能验证其错误性, 因为无论怎样选取阈值, 该阈值都是有穷大的数。设置循环上限 (阈值) 的目的只是为了防止由于验证内容的不正确而导致程序陷入死循环当中。



验证角谷猜想的算法描述如下:

```

proveJiaoGu(int n)
{
    int count=1;
    while(n!=1 && count<=1000){          /*阈值设为1000*/
        if(n%2==0)                        /*n为偶数*/
        {
            printf("%d/2=%d\n",n,n/2);
            n = n/2;
        }
        else
        {
            printf("%d*3+1=%d\n",n,n*3+1); /*n为奇数*/
            n=n*3+1;
        }
        count++;
    }
    if(count<1000 && n==1)
        printf("This natural number is according to JiaoGu Guess\n");
}

```

该算法中输入的参数为  $n$ , 然后按照角谷猜想的规律循环地对  $n$  进行各种操作, 并将每一步操作显示在屏幕上, 直到  $n$  等于 1, 或者循环次数  $\text{count}$  超过规定的上限阈值 1000 为止。最终, 如果  $n$  等于 1, 并且循环的次数  $\text{count}$  小于 1000, 则说明自然数  $n$  符合角谷猜想。否则要重新调整阈值上限, 继续验证。该算法只能验证角谷猜想的正确性。

程序清单 6-4

```

/*----- 6-4.c -----*/
#include "stdio.h"
proveJiaoGu(int n)
{
    int count=1;
    while(n!=1 && count<=1000){          /*阈值设为1000*/
        if(n%2==0)                        /*n为偶数*/
        {
            printf("%d/2=%d\n",n,n/2);
            n = n/2;
        }
        else
        {
            printf("%d*3+1=%d\n",n,n*3+1); /*n为奇数*/
            n=n*3+1;
        }
        count++;
    }
    if(count<1000 && n==1)
        printf("This natural number is according to JiaoGu Guess\n");
}

main()
{
    int n;
    printf("Please input a number to verify\n");
    scanf("%d",&n);
    printf("----- Step of Verification-----\n");
}

```

```

    proveJiaoGu(n);
    getche();
}

```

本程序的运行结果如图 6-5 所示。

```

Please input a number to verify
5
----- Step of Verification -----
5*3+1=16
16/2=8
8/2=4
4/2=2
2/2=1
This natural number is according to JiaoGu Guess

```

图 6-5 程序 6-4 的运行结果

## 6.5 验证四方定理

### 【题目要求】

四方定理是数论中的重要定理，它可以叙述为：所有自然数最多只要 4 个数的平方和就可以表示，编写一个程序验证四方定理。

### 【题目分析】

由于四方定理已被证明了的数论定理，因此所谓验证四方定理，实质上就是要把任意输入的一个自然数表示为至多 4 个数的平方和的形式。例如  $100=6^2+8^2$ ， $29=2^2+3^2+4^2$ ， $134=5^2+3^2+6^2+8^2$ ……。该问题可以这样想：对于任意输入的自然数  $N$ ，如果它可以表示成为一个数  $a$  的平方，即  $N=a^2$ ；或者它可以表示成为  $a$  和  $b$  的平方和，即  $N=a^2+b^2$ ；或者它可以表示成为  $a,b,c$  的平方和，即  $N=a^2+b^2+c^2$ ；或者它可以表示成为  $a,b,c,d$  的平方和，即  $N=a^2+b^2+c^2+d^2$ ，则都表明满足四方定理的要求。因此要把任意输入的一个自然数表示为至多 4 个数的平方和的形式，就是要寻找满足以上 4 个条件 ( $N=a^2$  或者  $N=a^2+b^2$  或者  $N=a^2+b^2+c^2$  或者  $N=a^2+b^2+c^2+d^2$ ) 的等式，只要找到其中的一个，就表明对于输入的自然数  $N$  验证了四方定理的正确性。

可以这样设计解题算法：

```

void proveFourSquares(int N)
{
    if(mode_1(N))           /* N=a^2 */
        printf("It has verified Four Squares theorem");
    else if(mode_2(N))       /* N=a^2+b^2 */
        printf("It has verified Four Squares theorem ");
    else if(mode_3(N))       /* N=a^2+b^2+c^2 */
        printf("It has verified Four Squares theorem ");
    else if(mode_4(N))       /* N=a^2+b^2+c^2+d^2 */
        printf("It has verified Four Squares theorem ");
    else printf("Error\n");
}

```

以上的算法描述为一个简单的分支语句，其中函数 `mode_1~mode_4` 为 4 个判断函数，分别判断自然数  $N$  是否可以表示成为  $N=a^2$ 、 $N=a^2+b^2$ 、 $N=a^2+b^2+c^2$ 、 $N=a^2+b^2+c^2+d^2$  的形式。如果能够表示成为上述形式则返回 1，否则返回 0。

由于本题是要验证四方定理, 因此可以保证一定能够输出正确的结果。在最后加上语句 `printf("Error\n");` 只是为了保证程序的完整性或防止意外的出现。

下面的关键问题是如何判断自然数  $N$  是否可以表示成为  $N=a^2$ 、 $N=a^2+b^2$ 、 $N=a^2+b^2+c^2$ 、 $N=a^2+b^2+c^2+d^2$  的形式, 即如何设计函数 `mode_1()`、`mode_2()`、`mode_3()`、`mode_4()`。它们的算法描述如下:

mode\_1

```
int mode_1(N)
{
    if((int)sqrt(N)*(int)sqrt(N)==N)
    {
        printf("%d*d=%d\n", (int)sqrt(N), (int)sqrt(N), N);
        return 1;
    }
    else return 0;
}
```

函数 `mode_1()` 的作用是判断自然数  $N$  是否可以表示成为  $N=a^2$  的形式。这里用关系表达式 `(int)sqrt(N)*(int)sqrt(N)==N` 作为判断的条件。因为函数 `sqrt()` 的返回值为浮点型数据, 因此要做 `(int)` 的强制转换将其取整。这样只有完全平方数  $N$  才能判断为真。如果自然数  $N$  是否可以表示成为  $N=a^2$  的形式, 该函数输出其表达式, 并返回 1; 否则返回 0。

mode\_2

```
int mode_2(N)
{
    int x, y;
    for(x=1; x<sqrt(N); x++)
        for(y=x; y<sqrt(N); y++)
        {
            if(x*x+y*y == N)
            {
                printf("%d^2+%d^2=%d\n", x, y, N);
                return 1;
            }
        }
    return 0;
}
```

函数 `mode_2()` 的作用是判断自然数  $N$  是否可以表示成  $N=a^2+b^2$  的形式。该函数与前面所讲的内容相似, 这里不再赘述。如果自然数  $N$  是否可以表示成为  $N=a^2+b^2$  的形式, 该函数输出其表达式, 并返回 1; 否则返回 0。

与函数 `mode_2()` 同样道理可以写出函数 `mode_3()` 和 `mode_4()`。详细可参看程序清单。

程序清单 6-5

```
/*----- 6-5.c -----*/
#include "stdio.h"
#include "math.h"

int mode_1(N) /*判断自然数 N 是否可以表示成为 N=a^2 的形式*/
{
    if((int)sqrt(N)*(int)sqrt(N)==N)
```

```

    {
        printf("%d*%d=%d\n", (int) sqrt(N), (int) sqrt(N), N);
        return 1;
    }
    else return 0;
}

int mode_2(N)          /*判断自然数N是否可以表示成 $N=a^2+b^2$ 的形式*/
{
    int x,y;
    for(x=1;x<sqrt(N);x++)
        for(y=x;y<sqrt(N);y++)
        {
            if(x*x+y*y == N)
            {
                printf("%d^2+%d^2=%d\n",x,y,N);
                return 1;
            }
        }
    return 0;
}

int mode_3(N)          /*判断自然数N是否可以表示成 $N=a^2+b^2+c^2$ 的形式*/
{
    int x,y,z;
    for(x=1;x<sqrt(N);x++)
        for(y=x;y<sqrt(N);y++)
            for(z=y;z<sqrt(N);z++)
            {
                if(x*x+y*y+z*z == N)
                {
                    printf("%d^2+%d^2+%d^2=%d\n",x,y,z,N);
                    return 1;
                }
            }
    return 0;
}

int mode_4(N)          /*判断自然数N是否可以表示成 $N=a^2+b^2+c^2+d^2$ 的形式*/
{
    int x,y,z,t;
    for(x=1;x<sqrt(N);x++)
        for(y=x;y<sqrt(N);y++)
            for(z=y;z<sqrt(N);z++)
                for(t=z;t<sqrt(N);t++)
                {
                    if(x*x+y*y+z*z+t*t == N)
                    {
                        printf("%d^2+%d^2+%d^2+%d^2=%d\n",x,y,z,t,N);
                        return 1;
                    }
                }
    return 0;
}

void proveFourSquares(int N)
{
    if(mode_1(N))
        printf("It has verified Four Squares");
}

```

```

else if(mode_2(N))
printf("It has verified Four Squares");
else if(mode_3(N))
printf("It has verified Four Squares");
else if(mode_4(N))
printf("It has verified Four Squares");
}

main()
{
    int N;
    printf("Please input a natural number\n");
    scanf("%d",&N);
    printf("----- Step of Verification-----\n");
    proveFourSquares(N);
    getch();
}

```

本程序的运行结果如图 6-6 所示。

```

Please input a natural number
156
----- Step of Verification-----
1^2+3^2+5^2+11^2=156
It has verified Four Squares

```

图 6-6 程序 6-5 的运行结果

## 6.6 递归法寻找最小值

### 【题目要求】

编写一个程序,要求从一个整数序列中找出最小的元素,并用递归的方法实现。

### 【题目分析】

在一个整数序列中找出最小值,最常用的方法是扫描整个序列,记录下最小的元素并返回。但是这里要求用递归的方法实现,因此必须构造这个问题递归解。可以这样理解:如果要从一个整数序列  $A=\{a_1, a_2, a_3, \dots, a_n\}$  中找出其最大的元素,实际上就是要从  $A$  的子序列  $A_1=\{a_1, a_2, \dots, a_{n/2}\}$  中找出最小的元素  $a_i$ ,再从  $A$  的子序列  $A_2=\{a_{n/2+1}, a_{n/2+2}, \dots, a_n\}$  中找出最小的元素  $a_j$ ,比较元素  $a_i$  和  $a_j$  的大小。如果  $a_i < a_j$ ,则  $a_i$  是序列  $A$  中的最小的元素,否则  $a_j$  是序列  $A$  中的最小元素。不难理解,这个寻找序列中最大值的过程是一个递归的过程,因为寻找子序列  $A_1$  和  $A_2$  中的最小值的过程也是重复上述过程的。因此可以总结出从一个整数序列中找出最小的元素的递归算法,描述如下:

```

int getMin(int array[],int n)
{
    int val1,val2,val3;
    if(n == 1) return array[0];
    if(n%2 == 0) /*序列长度为 2 的倍数*/
    {
        val1 = getMin(array, n/2); /*得到前半序列中的最小元素 val1*/
        val2 = getMin(array+n/2, n/2); /*得到后半序列中的最小元素 val2*/
    }
}

```

```

        if(val1<val2)
            return val1;                /*val1<val2, 返回 val1*/
        else
            return val2;                /*val1>=val2, 返回 val2*/
    }
    if(n%2 != 0)                        /*序列长度为奇数*/
    {
        val1 = getMin(array, n/2);      /*得到中位元素之前的子序列中的最小值*/
        val2 = getMin(array+n/2+1,n/2); /*得到中位元素之后的子序列中的最小值*/
        val3 = array[n/2];
        if(val1<val2)
        {
            if(val1<val3) return val1;
            else return val3;
        }
        else
        {
            if(val2<val3) return val2;
            else return val3;
        }
    }
}

```

在设计上述算法时,考虑得比前面所讲的递归法的解题思想要更加具体和明确一些。考虑到整数序列的长度可能为偶数也可能为奇数,在比较大小时需要分情况加以讨论。

(1) 如果整数序列的长度为偶数,则将该整数序列二等分,调用函数 `getMin(array, n/2)` 获得整数序列的前半部分(子序列)的最小值;再调用函数 `getMin(array+n/2, n/2)` 获得整数序列的后半部分(子序列)的最小值;再将得到的两个局部最小值进行比较得到整个序列的最小值。其中,函数 `getMin()` 为递归函数,在子序列中查找最小值的过程仍然重复上述过程(如果子序列中的元素个数也为偶数个,否则子序列的查找过程如2)。

(2) 如果整数序列的长度为奇数,则将该整数序列分为三部分:中位元素,中位元素之前的子序列,中位元素之后的子序列。调用函数 `getMin(array, n/2)` 获得中位元素之前的子序列中的最小值;调用函数 `getMin(array+n/2+1, n/2)` 获得中位元素之后的子序列中的最小值;再将这两个局部最小值与中位元素 `array[n/2]` 相互比较,求出整个序列的最小值。其中函数 `getMin` 为递归函数,在子序列中查找最小值的过程仍然重复上述过程(如果子序列中的元素个数也为奇数个,否则子序列的查找过程如1)。

下面给出完整的程序清单。

程序清单 6-6

```

/*----- 6-6.c -----*/
#include "stdio.h"

int getMin(int array[],int n)
{
    int val1,val2,val3;
    if(n == 1) return array[0];
    if(n%2 == 0)
    {
        val1 = getMin(array, n/2);
        val2 = getMin(array+n/2, n/2);
        if(val1<val2)
            return val1;
    }
}

```

```

        else
            return val2;
    }
    if(n%2 != 0)
    {
        val1 = getMin(array, n/2);
        val2 = getMin(array+n/2+1,n/2);
        val3 = array[n/2];
        if(val1<val2)
        {
            if(val1<val3) return val1;
            else return val3;
        }
        else
        {
            if(val2<val3) return val2;
            else return val3;
        }
    }
}

main()
{
    int array[9]={11,13,23,56,8,23,11,23,111},val;
    val = getMin(array,9);
    printf("%d",val);
    printf("The minum element of this array is %d \n",val);
}

```

本程序的运行结果如图 6-7 所示。

The minum element of this array is 8

图 6-7 程序 6-6 的运行结果

## 6.7 寻找同构数

### 【题目要求】

正整数  $n$  若是它平方数的尾部, 则称  $n$  为同构数。例如, 6 是其平方数 36 的尾部, 76 是其平方数 5776 的尾部, 因此 6 与 76 都是同构数。编写一个程序, 找出 1000 以内的同构数。

### 【题目分析】

本题最直观的解法是使用穷举法, 在 1~1000 的正整数中进行搜索, 判断每一个数是否是同构数。如果是则将其输出, 如果不是则跳过此数, 继续向下寻找, 直到判断完这 999 个数为止。该过程可用下面的算法描述:

```

void gettonggou()
{
    int i;
    for(i=1;i<1000;i++)
        /*在 1~1000 中寻找同构数*/
}

```

```

{
    if(func(i))                /*如果 i 是同构数，则将其输出*/
        printf("%d ",i);
}

```

这个算法再简单不过了，它通过一个简单的循环得到[1, 1000]中的每一个正整数，并调用函数 `func(i)` 来判断  $i$  是否是同构数。如果  $i$  是同构数，则函数 `func()` 返回 1，否则返回 0。

下面关键要解决的问题是如何判断一个正整数是否是同构数，即如何设计函数 `func()`。如果一个数  $i$  是同构数，则  $i$  满足性质： $i^2$  的尾部等于  $i$ 。尾部是一个人为规定的概念，在本题目中， $i^2$  的尾部就是指  $i^2$  中从最低位起与  $i$  位数相同的部分。例如  $4^2$  的尾部为 6，因为  $4^2=16$ ，而在 16 中，从最低位起与 4 位数相同的部分的数为 6（4 与 6 都是个位数），因此  $4^2$  的尾部为 6； $11^2$  的尾部为 21，因为  $11^2=121$ ，而 121 中从最低位起与 11 的位数相同的部分的数为 21（11 和 21 都是十位数），因此  $11^2$  的尾部为 21。如果彻底搞清楚了尾部的概念，就不难设计出算法去判断一个数是否是同构数。

要判断一个正整数  $i$  是否是同构数，首先要判断  $i$  的位数，即判断  $i$  是个位数，还是十位数，还是百位数……。因为这里要寻找 1000 以内的同构数，因此  $i$  的位数不可能超过百位数。

如果  $i$  是个位数，则  $i^2$  的尾部可通过  $i^2 \% 10$  来获得。只要判断  $i^2 \% 10$  是否等于  $i$  就可以判断出  $i$  是否是同构数。如果  $i^2 \% 10$  等于  $i$ ，则  $i$  是同构数，否则  $i$  不是同构数。

如果  $i$  是十位数，则  $i^2$  的尾部可通过  $i^2 \% 100$  来获得。只要判断  $i^2 \% 100$  是否等于  $i$  就可以判断出  $i$  是否是同构数。如果  $i^2 \% 100$  等于  $i$ ，则  $i$  是同构数，否则  $i$  不是同构数。

依次类推，可以判断  $i$  是否是同构数。判断  $i$  是否是同构数的算法描述如下：

```

int func(int i)
{
    int j;
    for(j=10;j<1000;j=j*10)
    {
        if(i/j == 0)                /*获得 i 的位数 j*/
            break;
    }
    if((i*i)%j == i)                /*判断 i 是否为同构数*/
        return 1;
    else
        return 0;
}

```

首先通过一个循环来判断参数  $i$  的位数。因为本题中  $i$  一定小于 1000，因此  $j$  的取值只可能为 10、100、1000 这 3 个值。具体地，如果  $i$  是个位数（0~9）， $j$  的值为 10，如果  $i$  是十位数（10~99）， $j$  的值为 100，如果  $i$  是百位数（100~999）， $j$  的值为 1000。然后通过条件语句 `if((i*i)%j == i)` 来判断  $i$  是否是同构数。

下面给出完整的程序清单。

程序清单 6-7

```

/*----- 6-7.c -----*/
#include "stdio.h"

```



```

int func(int i)
{
    int j;
    for(j=10;j<1000;j=j*10)
    {
        if(i/j == 0)
            break;
    }
    if((i*i)%j == i)
        return 1;
    else
        return 0;
}

void gettonggou()
{
    int i;
    for(i=1;i<=1000;i++)
    {
        if(func(i))
            printf("%d ",i);
    }
}

main()
{
    printf("The Tonggoushu bellow 1000 are\n");
    gettonggou();
    printf("\n");
}

```

本程序的运行结果如图 6-8 所示。

```

The Tonggoushu bellow 1000 are
1 5 6 25 76 376 625

```

图 6-8 程序 6-7 的运行结果

## 6.8 验证尼科彻斯定理

### 【题目要求】

尼科彻斯定理可以叙述为：任何一个整数的立方都可以表示成一串连续奇数的和。这里要注意，这些奇数一定是要连续的，例如 1、3、5、7、9…。

### 【题目分析】

要验证尼科彻斯定理的正确性，实质上就是要对任意输入的整数  $N$ ，找出一串连续的奇数，使得这串连续的奇数的和等于  $N^3$ 。因此本题可以换一种叙述的方式：对任意输入的整数  $N$ ，找出一串连续的奇数  $i$ 、 $i+2$ 、 $i+4$ 、…、 $i+2n$ ，使得  $N^3 = \sum_{k=i}^{i+2n} k$ ，其中  $n$  是未知量。

因为这里要求奇数序列是连续的，所以可以采用设定奇数序列起点的办法扫描奇数空间，找出符合要求的一串奇数序列。具体的做法如下。

如果输入的整数  $N$  的立方  $N^3$  恰好为一个奇数，则奇数空间可设为  $F\{1, 3, 5, \dots, N^3\}$ ，也就是说所要求的奇数序列  $S$  一定是  $F$  的一个连续子串。同理不难理解，如果输入的整数  $N$  的立方  $N^3$  恰好为一个偶数，则奇数空间可设为  $F\{1, 3, 5, \dots, N^3-1\}$ 。当然严格地讲待搜索的奇数空间并没有这么大，但是为了避免更多的数学推导，还是这样规定奇数空间。

下面的工作就是在奇数空间  $F$  中顺次设定起点，然后沿集中的元素累加下去，直到累加和超过了  $N^3$  或者等于  $N^3$  为止。如果本次累加和等于  $N^3$ ，则说明已找到了该连续的奇数序列串，因此程序可以跳出返回；否则如果累加和超过了  $N^3$ ，则说明本次起点设置的不对，顺次将下一个元素设置为起点，重复上述的操作。由于符合题目要求的奇数序列串是由它的起点元素决定的，因此只要起点选取正确，就一定能够找到该奇数序列。该算法描述如下：

```
void Nicoqish(int N)
{
    int i, j, sum = 0;
    for(i=1; i<N*N*N; i=i+2)                /*i 为起点*/
        for(j=i; j<N*N*N; j=j+2)            /*j 控制从 i 向后顺次累加*/
        {
            sum = sum + j;
            if(sum == N*N*N)                  /*如果奇数序列的和等于  $N^3$ */
            {
                printf("d=%d+d...+d\n", N*N*N, i, i+2, j);
                /*输出结果，程序返回*/
                return;
            }
            if(sum>N*N*N)                     /*如果奇数序列的和大于  $N$ ，跳出本次循环*/
            {
                sum = 0;
                break;
            }
        }
    printf("Erro\n");
}
```

函数 `Nicoqish()` 的作用是验证参数  $N$  的尼科彻斯性质。通过二重循环找出符合要求的连续奇数序列。外层循环通过变量  $i$  控制扫描奇数空间的起点，内层循环由变量  $j$  控制从起点  $i$  开始向后顺次累加，直到累加和超过了  $N^3$  或者等于  $N^3$  为止。如果在内层循环中找到符合要求的连续奇数序列，则输出结果，并通过 `return` 语句返回该函数；如果累加和超过了  $N^3$  而仍没有找到符合要求的连续奇数序列，则用 `break` 语句跳出本次循环，由外层循环重新设定起点。

由于本题是要验证尼科彻斯定理，因此可以保证一定能够输出正确的结果。在最后加上语句 `printf("Erro\n");` 只是为了保证程序的完整性或防止意外的出现。

程序清单 6-8

```
/*----- 6-8.c -----*/
#include "stdio.h"

void Nicoqish(int N)
{
    int i, j, sum = 0;
```

```

for(i=1;i<N*N*N;i=i+2)          /*i 为起点*/
    for(j=i;j<N*N*N;j=j+2)      /*j 控制从 i 向后顺次累加*/
    {
        sum = sum + j;
        if(sum == N*N*N)
        {
            printf("%d=%d+%d...+%d\n",N*N*N,i,i+2,j);
            return;
        }
        if(sum>N*N*N)
        {
            sum = 0;
            break;
        }
    }
}

main()
{
    int N;
    printf("Please input a integer to verify Nicoqish Law\n");
    scanf("%d",&N);
    Nicoqish(N);
    getch();
}

```

本程序的运行结果如图 6-9 所示。

```

Please input a integer to verify Nicoqish Law
5
125=21+23...+29

```

图 6-9 程序 6-8 的运行结果

## 6.9 三重回文数字

### 【题目要求】

找出 11~999 之间的所有的三重回文数字。所谓三重回文数  $a$  就是指  $a$ 、 $a^2$ 、 $a^3$  都是回文数字。

### 【题目分析】

前面已经详细介绍了如何判断一个数是否是回文数，本题是上一题的扩展，要求找出 11~999 之间的三重回文数字。因此可以采用穷举法，求出 11~999 之间每一个数的一次方，二次方和三次方，并分别判断它们是否都是回文数字，如果都是回文数字，则该数就是三重回文数字；否则该数不是三重回文数字。算法描述如下：

```

palindrome(long low,long high)
{
    long i;
    for(i=low;i<=high;i++)
    {
        if(isCircle(i) && isCircle(i*i) && isCircle(i*i*i))
            printf("%d ",i);
    }
}

```

在本算法中用函数 `isCircle()` 来判断一个数是否是回文数字。这里分别判断 `i`, `i*i` 和 `i*i*i` 是否是回文数字, 如果它们都是回文数字, 则 `i` 是三重回文数字, 并输出之; 否则 `i` 不是三重回文数字。`isCircle()` 的算法描述在前面的内容里已有详细的讲解, 这里不再赘述。

下面给出完整的程序清单。

程序清单 6-9

```

/*----- 6-9.c -----*/
#include "stdio.h"

long reverse(long i)
{
    long m, j=0;          /*求 i 的倒置数*/
    m=i;
    while(m){
        j=j*10+m%10;
        m=m/10;
    }
    return j;             /*返回 i 的倒置数 j*/
}

long isCircle(long n)
{
    /*函数 isCircle 判断 n 是否是回文数字*/
    long m;
    m= reverse(n);
    if(m==n)
        return 1;
    else
        return 0;
}

palindrome(long low, long high)
{
    long i;
    for(i=low; i<=high; i++)
    {
        if(isCircle(i) && isCircle(i*i) && isCircle(i*i*i))
            printf("%d ", i);
    }
}

main()
{
    printf("The palindrome numbers between 11~999 are\n");
    palindrome(11, 999);
    getch();
}

```

本程序的运行结果如图 6-10 所示。



```

The palindrome numbers between 11~999 are
11 101 111

```

图 6-10 程序 6-9 的运行结果

**注意:** 在本程序中函数 `isCircle()` 和函

数 `reverse()` 中定义的变量都是 `long` 型。这是因为要判断 11~999 返回内的每个整数是否是三重回文数字, 最多要计算到每个数的三次方, 例如 999 的三次方  $999^3 = 997002999$ , 它的范围超出了 `int` 类型数据所能表示的返回。因此这里的变量都设为 `long` 类型。

## 6.10 马克思手稿中的数学题

## 【题目要求】

伟大的思想家马克思的手稿中有这样一道有趣的数学题：有 30 个人，其中有男人、女人和小孩。他们在一家饭馆中吃饭，共花费 50 先令。如果每个男人吃饭要花 3 先令，每个女人要花 2 先令，每个小孩要花 1 先令，问男人、女人、小孩各多少人？

## 【题目分析】

可以设男人的人数为  $x$ ，女人的人数为  $y$ ，小孩的人数为  $z$ 。按照题目的已知，可列出一个三元一次方程组：

$$\begin{cases} 3x + 2y + z = 50 \\ x + y + z = 30 \end{cases}$$

该方程组的解就是本题目的答案。但是由代数学的知识我们知道，该方程组的解是不止一组的，因为该方程组必有一个自由未知数，其解为无数组。但是作为一个实际问题，该题的解是存在约束条件的，即所有解 ( $x$ 、 $y$ 、 $z$  取值) 必须为正整数。因此，本题可以在一个有限的解空间中通过穷举法进行求解。

为了方便起见，不妨设 ( $x$ 、 $y$ 、 $z$ ) 的解空间为  $(1, 1, 1) \sim (30, 30, 30)$ ，即  $x, y, z \in \mathbb{R}$  且  $1 \leq x \leq 30$ ； $1 \leq y \leq 30$ ； $1 \leq z \leq 30$ 。这样本题的解空间大小为  $30^3$ ，本题的答案也必定可以在这  $30^3$  组解中获得。算法描述如下：

```
Marx()
{
    int x,y,z;
    for(x=1;x<30;x++)
        for(y=1;y<30;y++)
            for(z=1;z<30;z++)
            {
                if(3*x+2*y+z==50 && x+y+z==30) /*如果符合条件*/
                    printf("%5d %5d %5d\n",x,y,z); /*输出结果*/
            }
}
```

本算法利用一个三重循环在  $(1, 1, 1) \sim (30, 30, 30)$  的解空间中寻找符合题目要求的答案，一旦找出一组解 ( $x$ 、 $y$ 、 $z$ ) 就将其输出。

下面给出完整的程序清单。

程序清单 6-10

```
/*----- 6-10.c -----*/
#include "stdio.h"

Marx()
{
    int x,y,z;
    for(x=1;x<30;x++)
```

```

        for(y=1;y<30;y++)
            for(z=1;z<30;z++)
            {
                if(3*x+2*y+z==50 && x+y+z==30)
                    printf("%5d %5d %5d\n",x,y,z);
            }
    }

main()
{
    printf(" The solutions of Marx's topic\n");
    printf(" Men Women Children\n");
    Marx();
    getch();
}

```

本程序的运行结果如图 6-11 所示。

The solutions of Marx's topic		
Men	Women	Children
1	18	11
2	16	12
3	14	13
4	12	14
5	10	15
6	8	16
7	6	17
8	4	18
9	2	19

图 6-11 程序 6-10 的运行结果

## 6.11 渔夫捕鱼问题

### 【题目要求】

A、B、C、D、E 5 个渔夫夜间合伙捕鱼，凌晨时都疲倦不堪，各自在河边的树丛中找地方睡着了。待日上三竿，渔夫 A 第一个醒来，他将鱼分作 5 份，把多余的一条扔回河中，拿自己的一份回家去了。渔夫 B 第二个醒来，也将鱼分作 5 份，扔掉多余的一条，拿走自己的一份，接着 C、D、E 依次醒来，也都按同样的办法分鱼，问 5 个渔夫至少合伙捕了多少条鱼？试编程序算出。

### 【题目分析】

可以直观地想到，假设 5 位渔夫捕得的鱼的数量不够多，那么当某个渔夫醒来后，他会发现剩下的鱼会不够分作五份的。例如第  $n$  个渔夫发现剩下的鱼为 6 条，于是他按照规则将其分为 5 份，并把多余的一条扔回河中，拿自己的一份回家去。这样剩下的鱼为 4 条。当第  $n+1$  个渔夫醒来后，由于剩下的鱼数为 4 条，因此就无法按照规则分鱼了。

因此要保证鱼的数量足够五名渔夫分，就要保证第 5 个渔夫 E 醒来分鱼时，剩下的鱼至少为 6 条。当然按照分鱼的规则，此时剩下的鱼的数目也可以是 11 条、16 条、21 条……，它的通式为  $5n+1$ ， $n \geq 1$ ， $n \in \mathbb{R}$ 。本题中要求计算 5 个渔夫至少合伙捕了多少条鱼，因此这里取最小值 6 即可。在知道了第 5 个渔夫 E 的分鱼总数后，就可以依次推导出第 4 个渔夫的分鱼总数，第 3 个渔夫的分鱼总数……，直到第 1 个渔夫的分鱼总数，也就是 5 位渔夫

的最少捕鱼量。设第 $n-1$ 位渔夫的分鱼总数为 $S_{n-1}$ ,第 $n$ 位渔夫的分鱼总数为 $S_n$ ,其中 $2 \leq n \leq 5$ , $n \in \mathbb{R}$ 。则有递推公式 $S_{n-1} = 5S_n + 1$ 。前面已经分析了 $S_5 = 6$ ,由该递推公式可求出 $S_4$ 、 $S_3$ 、 $S_2$ 、 $S_1$ 。 $S_1$ 即为要求的答案。下面给出本题的算法描述。

```
int func(int init,int n)
{
    int s = init;
    while(n)
    {
        s = f(s) ;
        n--;
    }
    return s;
}
```

本算法适用于一般的递推公式求解。函数func()包含两个参数,参数init为递推初值,即上述递推公式 $S_n$ 。参数n为迭代次数,即要执行多少次递推公式。函数f(s)为具体的递推公式。对于本题,递推初值init应为6,迭代次数n应为4,函数f(s)应为 $f(s) = 5s + 1$ 。

下面给出完整的程序清单。

程序清单 6-11

```
/*----- 6-11.c -----*/
#include "stdio.h"
int getfish(int init,int n)
{
    int s = init;
    while(n)
    {
        s = 5*s+1 ;
        n--;
    }
    return s;
}

main()
{
    printf("Fish which were gotten by fishers at least are %d",getfish(6,4));
    getch();
}
```

本程序的运行结果如图6-12所示。

Fish which were gotten by fishers at least are 3906

图6-12 程序6-11的运行结果

## 6.12 寻找假币

### 【题目要求】

一个国王要赏赐给一个大臣 30 枚金币,但是其中有一枚是假币。国王提出要求:只能用—个天平作为测量工具,并用尽量少的比较次数找出这枚假币,那么余下的 29 枚金币

就赏赐给这个大臣；否则这个大臣将得不到赏赐。已知假币要比真币的分量略轻一些。聪明的大臣思考片刻，很快用天平找出了这枚假币，于是得到了剩下的 29 枚金币。你知道这位大臣是如何找到假币的吗？请编写一个程序模拟找假币的过程，注意用尽量少的比较次数找出这枚假币。

### 【题目分析】

如果你是这位大臣，你能用什么办法找出假币呢？一种最简单的办法是应用天平进行两两比较。首先给硬币编上序号（1~30），然后按照下面的步骤进行两两比较：

（1）用天平比较第 1 枚硬币和第 2 枚硬币的重量，如果天平两边平衡，则进行第 2 步操作，否则较轻的一边的硬币为假币。

（2）用天平比较第 3 枚硬币和第 4 枚硬币的重量，如果天平两边平衡，则进行第 3 步操作，否则较轻的一边的硬币为假币。

（3）……

最坏的情况是当比较到第 29 枚硬币和第 30 枚硬币时才得出结果，也就是说第 29 或第 30 枚硬币是假币的情况。

应用这种方法进行硬币的两两比较，最终筛选出假币的序号虽然实施起来简单，易于理解，但是效率是比较低的，没有达到国王提出的“并用尽量少的比较次数找出这枚假币”的要求。我们来分析一下这种找假币的方法的效率。

如果 30 枚硬币中第 1 枚或第 2 枚硬币是假币，那么就十分幸运了，因为只要用天平比较一次就可以找出假币来。如果第 3 枚或第 4 枚硬币是假币，则要用天平比较 2 次才能找出假币来。依次类推，如果第 29 枚或第 30 枚硬币是假币，则要用天平比较 15 次才能找出假币来。而最初为硬币编号时是随机编号的，因此假币的序号也是随机的。如果将这 30 枚硬币编为 15 组，第 1 枚硬币和第 2 枚硬币为一组，第 3 枚硬币和第 4 枚硬币为一组，……，第 29 枚硬币和第 30 枚硬币为一组，那么假币落入某一组的概率都是 1/15。因此可以得出使用这种两两比较的方法找假币的平均比较次数：

$$ACT = \frac{1}{15} \sum_{i=1}^{15} i = 8$$

上式中 ACT 表示平均比较次数。也就是说，使用这种两两比较的方法在 30 枚硬币中寻找假币，平均需要用天平测 8 次才能找出假币来。

这里推荐一种更为高效的找假币的方法——利用递归分治的策略寻找假币。可以先将所有的硬币等分为两份，放在天平的两边。因为假币的分量较轻，因此天平较轻的一侧中一定包含假币。再将较轻的一侧中的硬币等分为两份，重复上述的做法。直到剩下 2 枚硬币，可用天平直接找出假币来。在这个比较的过程中，可能会出现硬币无法等分的情况，也就是硬币的个数为奇数的情况。在这种情况下，不妨挑出一枚硬币，将其余的硬币等分后放入天平测量，如果天平平衡，则表明刚才挑出的那枚硬币一定是假币；否则将较轻的一侧中的硬币等分为两份，重复上述的做法。这个过程就是一个标准的分治递归的过程，我们先给出它的算法描述，再来分析它的效率（比较次数）。

```
int getFalseCoin(int coin[],int low,int high)
{
    int i,sum1 = 0,sum2 = 0,sum3 = 0;
    if(low+1==high)
    {
```



```

    if(coin[low]<coin[high]) return low+1;      /*返回假币的编号*/
    else return high+1;
}
if((high-low+1)%2 == 0)                      /*偶数个硬币*/
{
    for(i=low;i<=low+(high-low)/2;i++)
        sum1= sum1 + coin[i];                /*前半段和*/
    for(i=low+(high-low)/2+1;i<=high;i++)
        sum2 = sum2 + coin[i];                /*后半段和*/
    /*在数组 coin 的 low~high 范围的前半段寻找假币*/
    if(sum1 < sum2) return getFalseCoin(coin,low,low+(high-low)/2);
    /*在数组 coin 的 low~high 范围的后半段寻找假币*/
    if(sum1 > sum2) return getFalseCoin(coin,low+(high-low)/2+1,high);
}
if((high-low+1)%2 !=0)                      /*奇数个硬币*/
{
    for(i=low;i<=low+(high-low)/2-1;i++)
        sum1= sum1 + coin[i];                /*前半段和*/
    for(i=low+(high-low)/2+1;i<=high;i++)
        sum2 = sum2 + coin[i];                /*后半段和*/
    sum3 = coin[low+(high-low)/2];
    if(sum1< sum2) /*在数组 coin 的 low~high 范围的前半段寻找假币*/
        return getFalseCoin(coin,low,low+(high-low)/2-1);
    if(sum1 > sum2) /*在数组 coin 的 low~high 范围的后半段寻找假币*/
        return getFalseCoin(coin,low+(high-low)/2+1,high);
    if(sum1+sum3 == sum2+sum3) return low+(high-low)/2+1;
    /*返回假币的编号*/
}
}

```

因为要用程序模拟找假币的过程,所以这里用一个数组 `coin[]` 来存放每枚假币的重量。当然这里要求假币的重量比真币轻,并且真币的重量都相同。函数 `getFalseCoin()` 的作用是找出数组 `coin[]` 中元素较小的那一个(即假币),并返回它所在数组中的位置。也就是说返回假币的序号。函数 `getFalseCoin()` 包含 3 个参数, `coin[]` 为存放硬币重量的数组的首地址, `low` 为数组下标的下限, `high` 为数组下标的上限。因为每一层的递归都可能在数组 `coin[]` 中的不同区段内进行操作,所以为了操作方便,添加 `low` 和 `high` 两个参数。为了方便阐释这段代码的含义,以一个实例为对象进行讲解。

假设现有 10 枚硬币,真币重量为 2,假币重量为 1,用数组 `coin[]` 记录下它们的重量: `coin[10] = {2,2,2,2,1,2,2,2,2,2}`。很显然,编号为 5 的硬币是假币,其余的硬币是真币。

首先要在整个数组 `coin[]` 的范围内进行查找,因此 `low=0`, `high=9`。因为是偶数个硬币,所以进入第 2 个 `if` 语句。用变量 `sum1` 记录下数组 `coin` 的前半段和,等于 9;用变量 `sum2` 记录下数组 `coin` 的后半段和,等于 10。所以假币一定在 `coin[0]~coin[4]` 中。递归调用函数 `getFalseCoin()`,在数组 `coin[]` 的 `low=0` 到 `high=4` 的范围内寻找假币。

因为本次递归调用中硬币的个数为奇数(5 枚硬币),因此进入第 3 个 `if` 语句。用变量 `sum1` 记录下数组 `coin` 的前半段和,等于 4;用变量 `sum2` 记录下数组 `coin` 的后半段和,等于 3;用变量 `sum3` 记录中间值 `coin[2]=2`。因为 `sum1>sum2`,因此假币一定在 `coin[3]~coin[4]` 中。如果 `sum1` 等于 `sum2`,则说明中间序号的硬币 `sum3` 是假币。递归调用函数 `getFalseCoin()`,在数组 `coin[]` 的 `low=3` 到 `high=4` 的范围内寻找假币。

因为本次递归调用中 `low` 的值加 1 等于 `high` 的值,这表明本次递归调用只对 2 枚硬币

进行比较,因此只需返回较小的硬币序号下标即可。这里要注意返回值为数组 `coin[]` 的下标值加1,这是因为数组 `coin[]` 的下标是从0开始计起的,而硬币的编号则是从1计起。

通过上述具体实例的介绍,我们就不难理解函数 `getFalseCoin()` 的实现原理和执行过程了。下面来分析一下该算法的效率。

如果在30枚硬币中寻找假币,第一次比较可将假币搜索的范围缩小至15枚,第二次比较可将假币的搜索范围缩小至7枚,第三次比较可将假币的搜索范围缩小至3枚,第4次比较可将假币的搜索范围缩小至2枚,第5次比较即可得到结果。这说明在最坏的情况下,应用上述递归分治策略寻找30枚硬币中的假币,仅需要比较5次即可找出假币。其实在第二次比较、第三次比较、第四次比较中都有可能直接找出假币的序号来(如果假币的序号位于中间值),因此应用该算法在这30枚硬币中寻找假币,平均需要用天平比较的次数要小于5次。很显然比较的次数要比上面介绍的“两两比较”的方法少一些。

可以证明搜索的硬币数量越多,与两两比较的算法相比,使用递归分治算法寻找假币的效率越高,优势越明显。有兴趣的读者可以自己验证。

下面给出完整的程序清单供读者参考。

程序清单 6-12

```

/*----- 6-12.c -----*/
#include "stdio.h"

int getFalseCoin(int coin[],int low,int high)
{
    int i,sum1 = 0,sum2 = 0,sum3 = 0;
    if(low+1==high)
    {
        if(coin[low]<coin[high]) return low+1;
        else return high+1;
    }
    if((high-low+1)%2 == 0) /*n是偶数*/
    {
        for(i=low;i<=low+(high-low)/2;i++)
            sum1= sum1 + coin[i]; /*前半段和*/
        for(i=low+(high-low)/2+1;i<=high;i++)
            sum2 = sum2 + coin[i]; /*后半段和*/
        if(sum1 < sum2) return getFalseCoin(coin,low,low+(high-low)/2);
        if(sum1 > sum2) return getFalseCoin(coin,low+(high-low)/2+1,high);
    }

    if((high-low+1)%2 !=0)
    {
        for(i=low;i<=low+(high-low)/2-1;i++)
            sum1= sum1 + coin[i]; /*前半段和*/
        for(i=low+(high-low)/2+1;i<=high;i++)
            sum2 = sum2 + coin[i]; /*后半段和*/
        sum3 = coin[low+(high-low)/2];
        if(sum1< sum2)
            return getFalseCoin(coin,low,low+(high-low)/2-1);
        if(sum1 > sum2)
            return getFalseCoin(coin,low+(high-low)/2+1,high);
        if(sum1+sum3 == sum2+sum3) return low+(high-low)/2+1;
    }
}

```

```

    }
}
main()
{
    int coin[30] = {2,2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,2, 2,2,2,2,2,
    2,2,1,2,2};
    printf("The %dth coin is false\n",getFalseCoin(coin,0,29));
    getch();
}

```

本程序的运行结果如图 6-13 所示。

The 28th coin is false

图 6-13 程序 6-12 的运行结果

Ⓐ注意：程序中数组  $\text{coin}[30]=\{2,2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,2,2,2, 2,2,2,2,2,2,2,1,2,2\}$ ；表示编号为 28 的硬币为假币。然后调用函数  $\text{getFalseCoin}()$  确定假币的编号，返回值为 28。

## 6.13 计算组合数

### 【题目要求】

编写一个程序，计算组合数  $C_m^n$ 。

### 【题目分析】

组合数是概率统计中的一个重要概念。组合数  $C_m^n$  的概率意义是从  $m$  个事物中任意选取  $n$  个事物的选法。例如从标号为 1~5 的小球中任意选取 2 个，则其不同选法的种类即为一个组合数，记作  $C_5^2$ 。这里要区分排列和组合概念上的不同。所谓组合只是考虑选取的方案种类，而并不考虑每个方案中内部的排列次序。因此  $C_5^2=10$ ，包含的组合分别为：(1, 2)、(1, 3)、(1, 4)、(1, 5)、(2, 3)、(2, 4)、(2, 5)、(3, 4)、(3, 5)、(4, 5) 10 种。也就是说，从标号为 1~5 的小球中任意选取 2 个，共有上述 10 种选取方案。

计算组合数的方法很多，这里介绍应用递归法计算组合数。为了叙述得更加清楚了，仍然以上述选取小球为实例。

首先可以规定只从标号为 1~4 的小球中任意选取 2 个小球，而不选取第 5 号小球，那么选取的方案为  $C_4^2$  种。然后规定必须选取第 5 号小球，那么也就是说只能从标号为 1~4 的小球中任意选取 1 个小球，因此选取的方案为  $C_4^1$  种。不难理解，所谓的从标号为 1~5 的小球中任意选取 2 个小球，其选取的方案恰好是上述的两种情形的和。也就是说，将“从标号为 1~5 的小球中任意选取 2 个小球”的事件划分为两个子事件，事件一：“不选取第 5 号小球，从标号为 1~4 的小球中任意选取 2 个小球”；事件二：“选取第 5 号小球，再从标号为 1~4 的小球中任意选取 1 个小球”。以是否选取第 5 号小球为事件的划分点（这种划分是完备的，因为它将是否选取第 5 号小球的情况都考虑了进去，不存在第三个子事件）。事件一与事件二是不相容的（不存在交集），因此事件一的组合数与事件二的组合

数之和一定等于其和事件的组合数,即  $C_3^2 = C_4^2 + C_4^1$ 。显然这里构成了一个递归形式的表达式,因为在计算  $C_4^2$  和  $C_4^1$  时,仍可以沿用上述的划分和事件,求解子事件组合数之和的方法。这样就可以得出计算组合数的递归公式:

$$C_m^n = \begin{cases} 1 & n=m \text{ or } n=0 \\ C_{m-1}^n + C_{m-1}^{n-1} & 1 < n < m \end{cases}$$

当  $n$  等于  $m$  或  $n$  等于 0 时,  $C_m^n$  的值为 1, 这作为递归调用的结束条件。因此就不难写出计算组合数的递归算法:

```
int cnr(int m,int n)
{
    if(m == n || n == 0)
        return 1;
    else
        return cnr(m-1,n) + cnr(m-1,n-1);
}
```

该算法参数为  $m$  和  $n$ , 返回  $C_m^n$  的值。

下面给出完整的程序清单供读者参考。

程序清单 6-13

```
/*----- 6-13.c -----*/
#include "stdio.h"
int cnr(int m,int n)
{
    if(m == n || n == 0)
        return 1;
    else
        return cnr(m-1,n) + cnr(m-1,n-1);
}

main()
{
    int m,n;
    printf("Please input m and n for C(m,n)\n");
    scanf("%d %d",&m,&n);
    printf("C(%d,%d)=%d",m,n,cnr(m,n));
    getch();
}
```

本程序的运行结果如图 6-14 所示。

```
Please input m and n for C(m,n)
5 2
C(5,2)=10
```

图 6-14 程序 6-13 的运行结果

## 6.14 递归法求幂

### 【题目要求】

请编写一个递归算法, 计算  $m^n$ 。

## 【题目分析】

一般情况下计算  $m^n$  时多采用循环连乘的方法, 即把  $m$  连乘  $n$  次。这种方法的效率是很低的。现在介绍一种更为有效的算法来计算整数的幂——递归法。

参看下面的公式:

$$m^n = \begin{cases} 1 & n=0 \\ (m^k)^2 & n=2k \\ m \cdot m^{2k} & n=2k+1 \end{cases}$$

其实  $m^n$  完全可以如上述公式这样定义。当指数  $n$  等于 0 时,  $m^n$  的值一定为 1; 当指数  $n$  为偶数时, 即指数  $n$  可以表示成  $2k$ , 这样  $m^n$  的值等于  $m^{2k}=(m^k)^2$ ; 当指数  $n$  为奇数时, 即指数  $n$  可以表示成  $2k+1$ , 这样  $m^n$  的值等于  $m \cdot m^{2k}$ 。这样便构成了计算整数幂的递归结构。因为在计算  $m^k$  和  $m^{2k}$  时仍需要重复上述过程。

可以将上述公式改写成为下列递归函数:

$$\text{pow}(m, n) = \begin{cases} 1 & n=0 \\ m & n=1 \\ \text{pow}(m, k) \cdot \text{pow}(m, k) & n=2k, k=1, 2, 3 \cdots \\ m \cdot \text{pow}(m, 2k) & n=2k+1, k=1, 2, 3 \cdots \end{cases}$$

函数  $\text{pow}(m, n)$  表示计算  $m$  的  $n$  次幂  $m^n$  的值。递归结束的条件是  $n$  等于 1 和  $n$  等于 0。这样就不难写出其递归算法。

```
unsigned long myPow(int m, int n)
{
    unsigned long tmp;
    if(n == 0) return 1;
    if(n == 1) return m;
    if(n % 2 == 0) /*指数 n 为偶数的情况*/
    {
        tmp = myPow(m, n/2);
        return tmp * tmp;
    }
    if(n % 2 != 0) /*指数 n 为奇数的情况*/
        return m * myPow(m, n-1);
}
```

应用这种递归算法计算整数的幂要比循环连乘的方法高效得多。以计算  $m^8$  为例, 应用循环连乘的方法计算需要进行 7 次乘法, 而如果采用这种递归算法来求解, 只需要进行 3 次乘法。理论上可以证明使用这种递归算法计算整数的幂所需的乘法次数为  $O(\log(n))$ , 相比之下, 使用循环连乘的方法所需的乘法次数为  $O(n)$ 。因此递归算法的效率有了明显的提高, 特别是在指数  $n$  很大的情况下, 这种提高的效果更加明显。

有两点特别值得注意。第一, 在本算法中, 当  $n\%2$  等于 0 时, 用到了一个临时变量  $\text{tmp}$  来保存递归函数  $\text{myPow}()$  返回的值, 然后再返回  $\text{tmp} \cdot \text{tmp}$  的值。只有这样做才能真正体现该算法的优势, 真正提高算法的效率。如果直接返回  $\text{myPow}(m, n/2) \cdot \text{myPow}(m, n/2)$ , 则要调用两次递归函数  $\text{myPow}()$ , 并没有减少任何乘法运算, 这样的效率与循环连乘的效率是一样的, 失去了算法改进的意义。第二, 幂运算的增长速度是在所有类型的运算中最快的, 看似不起眼的  $5^5$ , 其运算结果为 3125。因此计算结果的返回值类型应当足够的长, 防止数

据越界溢出,必要时可使用字符串等数据结构作为数据的存储容器模拟进行幂运算。

下面给出完整的程序清单供读者参考。

程序清单 6-14

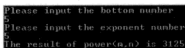
```

/*----- 6-14.c -----*/
#include "stdio.h"
unsigned long myPow(int m,int n)
{
    unsigned long tmp;
    if(n == 0) return 1;
    if(n == 1) return m;
    if(n % 2 == 0){
        tmp = myPow(m,n/2);
        return tmp * tmp;
    }
    if(n % 2 != 0)
        return m * myPow(m,n-1);
}

main()
{
    int m,n;
    printf("Please input the bottom number\n");
    scanf("%d",&m); /*输入底数 m*/
    printf("Please input the exponent number\n");
    scanf("%d",&n); /*输入指数 n*/
    printf("The result of power(m,n) is %ld\n",myPow(m,n));
    /*输出计算结果 m^n*/
    getche();
}

```

本程序的运行结果如图 6-15 所示。



```

Please input the bottom number
5
Please input the exponent number
5
The result of power(5,5) is 3125

```

图 6-15 程序 6-14 的运行结果

## 6.15 汉诺 Hanoi 塔

### 【题目要求】

一块板上有三根针：A、B、C。A 针上套有 64 个大小不等的圆盘，大的在下，小的在上，如图 6-16 所示。要把这 64 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。求移动的步骤。

### 【题目分析】

这是一个古老的数学问题，也是一个经典的递归问题。如果要满足上述的要求：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。可以这样考虑移动的步骤：

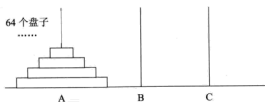


图 6-16 Hanoi 塔示意

- (1) 先将第 1~63 个盘子移到 B 针上, 要保证大盘在下小盘在上。
- (2) 再将最底下的最大盘子移到 C 针上。
- (3) 最后将 B 针上的 63 个盘子移到 C 针上。

这样问题就解决了。但是关键在于第(1)步和第(3)步如何执行。由于每次只能移动一个圆盘, 所以在移动的过程中显然要借助另外一根针, 即第(1)步将第 1~63 个盘子借助 C 针移到 B 针上; 第(3)步将 B 针上的 63 个盘子借助 A 针移到 C 针上。这显然又成为两个新的 Hanoi 塔问题了。

问题一: 将 A 针上的第 1~63 个盘子借助 C 针移到 B 针上。

问题二: 将 B 针上的 63 个盘子借助 A 针移到 C 针上。

解决上述两个问题依然用前面的方法。问题一的圆盘移动步骤为:

- (1) 将 A 针上的第 1~62 个盘子借助 B 针移到 C 针上, 要保证大盘在下小盘在上。
- (2) 再将第 63 个盘子移到 B 针上。
- (3) 最后将 C 针上的 62 个盘子借助 A 针移到 B 针上。

问题二的圆盘移动步骤为:

- (1) 将 B 针上的第 1~62 个盘子借助 C 针移到 A 针上, 要保证大盘在下小盘在上。
- (2) 再将第 63 个盘子移到 C 针上。
- (3) 最后将 A 针上的 62 个盘子借助 B 针移到 C 针上。

其中, 第(1)步和第(3)步的解决又是新的 Hanoi 塔问题, 解决的方案与上面一样, 直到第(1)步和第(3)步所移动的盘子个数为 1 时为止。这显然是个递归的问题, 也就是 Hanoi 塔问题中有 Hanoi 塔问题。

在程序的设计上, 设函数 `move(int n, int x, int y, int z)`, 表示的意思是: 将  $n$  个盘子从  $x$  针借助  $y$  针移到  $z$  针上。可以这样定义该函数:

```
move(int n, char x, char y, char z)
{
    if (n==1)
        printf("%c-->%c\n", x, z);
    else
    {
        move(n-1, x, z, y); *
        printf("%c-->%c\n", x, z);
        move(n-1, y, x, z);
    }
}
```

递归的结束条件是: ( $n=1$ ), 也就是这时只需要移动一个盘子, 那么无需借助  $y$  针, 直接显示移动过程  $x \rightarrow z$ 。

否则, 递归地调用函数 `move()`, 按照上面所讲的移动步骤先将  $n-1$  个盘子从  $x$  针借助

$z$  针移到  $y$  针。然后将第  $n$  个盘子直接移到  $z$  针，也就是显示移动过程  $x \rightarrow z$ 。

最后将  $y$  上的  $n-1$  个盘子通过  $x$  针移到  $z$  针。

而每次的所谓从“某针借助某针移到某针”都是重新调用 `move()` 函数的过程，也就是重新执行上述步骤的过程。直到遇到结束条件  $n=1$ ，这时直接显示移动过程，并且不再递归地调用 `move()` 函数。

程序清单 6-15

```
/*----- 6-15.c -----*/
#include <stdio.h>
move(int n,char x,char y,char z)
{
    if(n==1)
        printf("%c-->%c\n",x,z);
    else
    {
        move(n-1,x,z,y);
        printf("%c-->%c\n",x,z);
        move(n-1,y,x,z);
    }
}
int main(void)
{
    int n;
    printf("input disks number:\n");
    scanf("%d",&n); /*输入盘子数目 n*/
    printf("The step to moving %d disks:\n",n);
    move(n,'A','B','C'); /*递归调用 move(), 求解盘子的搬运过程*/
    getch();
    return 0;
}
```

这里仅以 3 阶 Hanoi 塔问题为例，最终得到盘子的移动步骤，如图 6-17 所示。

```
input disks number:
3
The step to moving 3 disks:
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C
```

图 6-17 程序 6-15 的运行结果

## 6.16 选美比赛

### 【题目要求】

在选美比赛的现场，有一批选手参加比赛，比赛的规则是最后得分越高，名次越低。当比赛结束时，要在现场按照选手的出场顺序（即选手的序号）宣布最后得分和最后名次，获得相同分数的选手具有相同的名次，名次连续编号，不用考虑同名次的选手人数。例如：

选手序号为： 1, 2, 3, 4, 5, 6, 7



选手得分为: 5, 3, 4, 7, 3, 5, 6

则输出名次为: 3, 1, 2, 5, 1, 3, 4

请编程帮助大赛组委会完成比赛的评分和排名工作。

### 【题目分析】

读者可能不假思索地说出解决此问题的方法, 将选手的得分存储到一个数组中, 然后从小到大进行排列, 排列靠前的名次靠前, 排列靠后的名次靠后。但是问题并非像想象的那么简单。首先题目要求按照选手的出场顺序(即选手的序号)宣布最后得分, 也就是说并不是按照名次的前后输出选手信息, 而是按照选手的序号输出最后的得分和名次。其次, 题目要求获得相同分数的选手具有相同的名次, 名次连续编号, 不用考虑同名次的选手人数, 因此不存在并列名次占位的情况, 即如果存在并列第一名, 那么下一名的名次是第二名, 而不是第三名。另外, 如果只是简单地对选手的得分序列进行排序, 那么选手的得分与选手的序号就不能构成一一对应的关系, 那么这样的排序也就没有意义了。出于这几点考虑, 此问题并非只是一个简单的排序运算就可以解决的。

一种比较直观的解决方法是应用结构体。将每个选手的信息(包括序号、得分、名次)存放在一个结构体变量中, 然后组成一个结构体数组。该结构体可定义为:

```
struct player{
    int num;
    int score;
    int rand;
};
```

最开始每个结构体变量中只存放选手的序号和得分的信息, 然后以选手的得分为比较对象, 从小到大进行排序。算法描述如下:

```
void sortScore(struct player psn[],int n)
{
    int i,j;
    struct player tmp;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-1-i;j++)
        {
            if(psn[j].score>psn[j+1].score)
            {
                tmp = psn[j];
                psn[j] = psn[j+1];
                psn[j+1] = tmp;
            }
        }
}
```

这里应用冒泡排序法对含有  $n$  个元素的结构体数组 `psn` 中的元素按照 `score` 从小到大的顺序进行排序。排序后的数组 `psn` 按照 `score` 的值从小到大排列。

然后指定每一位选手的名次。因为此时结构体数组 `psn` 已按照 `score` 从小到大排列, 因此就比较容易设定每一位选手的名次了。该算法描述如下:

```
void setRand(struct player psn[],int n)
{
    int i,j=2;
    psn[0].rand=1;
    for(i=1;i<n;i++)
```

```

    {
        if(psn[i].score!=psn[i-1].score )
        {
            psn[i].rand=j;
            j++;
        }
        else
            psn[i].rand=psn[i-1].rand;
    }
}

```

首先给第一位选手psn[0]的名次设定为1,因为它的得分是最少的。然后依次给psn[2]~psn[n-1]设定名次。如果psn[i].score 不等于psn[i-1].score, 说明psn[i]的名次要落后一名;否则psn[i]的名次与psn[i-1]的名次相同。

最后再按照选手的序号重新排序,以便能够按照选手的序号输出结果。该算法描述如下:

```

void sortNum(struct player psn[],int n)
{
    int i,j;
    struct player tmp;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-1-i;j++)
        {
            if(psn[j].num>psn[j+1].num)
            {
                tmp = psn[j];
                psn[j] = psn[j+1];
                psn[j+1] = tmp;
            }
        }
}

```

程序清单 6-16

```

/*----- 6-16.c -----*/
#include "stdio.h"
struct player{
    int num;
    int score;
    int rand;
};

void sortScore(struct player psn[],int n)
{
    int i,j;
    struct player tmp;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-1-i;j++)
        {
            if(psn[j].score>psn[j+1].score)
            {
                tmp = psn[j];
                psn[j] = psn[j+1];
                psn[j+1] = tmp;
            }
        }
}

```

```

    }
}

void setRand(struct player psn[],int n)
{
    int i,j=2;
    psn[0].rand=1;
    for(i=1;i<n;i++)
    {
        if(psn[i].score!=psn[i-1].score )
        {
            psn[i].rand=j;
            j++;
        }
        else
            psn[i].rand=psn[i-1].rand;
    }
}

void sortNum(struct player psn[],int n)
{
    int i,j;
    struct player tmp;
    for(i=0;i<n-1;i++)
        for(j=0;j<n-1-i;j++)
        {
            if(psn[j].num>psn[j+1].num)
            {
                tmp = psn[j];
                psn[j] = psn[j+1];
                psn[j+1] = tmp;
            }
        }
}

void sortRand(struct player psn[],int n)
{
    sortScore(psn,n);          /*以分数为关键字排序*/
    setRand(psn,n);           /*按照分数排名次*/
    sortNum(psn,n);           /*按照序号重新排序*/
}

main()
{
    struct player psn[7]={ {1,5,0},{2,3,0},{3,4,0},{4,7,0},
        {5,3,0},{6,5,0},{7,6,0}};
    int i;
    sortRand(psn,7);
    printf("num  score rand  \n");
    for(i=0;i<7;i++)
    {
        printf("%d\t%d\t%d\n",psn[i].num,psn[i].score,psn[i].rand);
    }
}

```

```
getche();  
}
```

本程序的运行结果如图 6-18 所示。

num	score	rand
1	5	3
2	3	1
3	4	2
4	7	5
5	3	1
6	5	3
7	6	4

图 6-18 程序 6-16 的运行结果

## 第7章 数据结构趣题

掌握数据结构是编写出地道、复杂程序的基础。很难想象一个不懂数据结构的人，或是不够精通数据结构的人会是一个出色的程序员。因此多用数据结构的知识编写一些程序对培养人们的编程水平以及用计算机处理复杂问题的能力都大有裨益。本章将通过一些具体的实例介绍数据结构在程序设计中的应用。

### 7.1 顺序表的就地逆置

#### 【题目要求】

编写一个函数，实现顺序表的就地逆置，也就是说利用原表的存储空间将顺序表  $(a_1, a_2, \dots, a_n)$  逆置为  $(a_n, a_{n-1}, \dots, a_1)$ 。

#### 【题目分析】

本题主要考查顺序表线性结构的应用。前面的章节中已经讲过，顺序表的基本操作包括顺序表的创建、插入数据、删除数据等。但是在实际的应用中，对顺序表的操作并不仅限于上述这几种操作。因此学习顺序表也不能仅限于学懂前面所讲的几种操作而已，应当灵活掌握顺序表结构，并能够熟练地操纵顺序表。

要实现长度为  $\text{length}$  的顺序表的就地逆置，需要利用一个临时变量  $\text{buf}$  作为数据的缓冲区，同时要设置两个指针变量  $\text{low}$  和  $\text{high}$  分别指向顺序表的首尾。操作的过程如下：

- (1) 先将  $\text{low}$  指向的内容存放到临时缓冲区  $\text{buf}$  中。
  - (2) 再将  $\text{high}$  指向的内容存放到  $\text{low}$  指向的缓冲区  $\text{buf}$  中。
  - (3) 将缓冲区中的内容存放到  $\text{high}$  指向的缓冲区中  $\text{buf}$ 。
  - (4)  $\text{low}++$ ,  $\text{high}--$ ，重复 (1) (2) (3) 的操作，共重复执行  $\lceil \text{length}/2 \rceil$  次
- 通过上面的算法描述就可以实现一个长度为  $\text{length}$  的顺序表的就地逆置。

程序清单 7-1

```
/*----- 7-1.c -----*/
#include "stdio.h"
#define MAXSIZE 10 /*静态顺序表的最大空间*/
typedef struct {
    int * base;
    int length;
}sqlist; /*定义一个顺序表类型*/

reverseSQ(sqlist *l) {
    /*实现顺序表 l 的就地逆置*/
    int low = 0, high = l->length - 1; /*low 和 high 分别指向顺序表的首尾*/
```

```

int buf , i;
for(i=0;i<l->length/2;i++)
{
    /*循环 length/2 次, 实现数据逆置*/
    buf = l->base[low] ;
    l->base[low] = l->base[high];
    l->base[high] = buf;
    low++;
    high--;
}
}

main()
{
    slist l;
    int a , i = 0;
    /*创建一个顺序表*/
    l.base = (int *)malloc(sizeof(int)*MAXSIZE);
    l.length = 0;

    /*输入数据*/
    printf("Please input below 10 integer into the slist\n");
    printf("Type -1 for stopping input\n");
    scanf("%d",&a);
    while(a != -1 && i<=9)
    {
        l.base[i] = a;
        l.length++;
        i++;
        scanf("%d",&a);
    }

    /*输出原顺序表中的数据*/
    printf("The contents of the slist are\n");
    for(i=0;i<l.length;i++)
        printf("%d ",l.base[i]);
    printf("\n");

    reverseSQ(&l);    /*就地逆置顺序表*/

    /*输出逆置后的顺序表中的数据*/
    printf("The contents of the reversed slist are\n");
    for(i=0;i<l.length;i++)
        printf("%d ",l.base[i]);
    getche();
}

```

### 【程序说明】

为了简化程序的复杂度,把重点放在顺序表的逆置上,这里对顺序表的其他操作(创建顺序表、插入元素等)都放在了主函数中,在实际的应用中建议按照规范把每个特定的操作都定义在不同的函数中,这样更加符合结构化程序设计的要求。

本程序应用静态的顺序表实现顺序表逆置,因此输出的数据个数不能超过 10 个。如果输入的数据超过 10 个,那么程序会自动按照顺序表中仅有 10 个数据的情况处理。如果在输入数据的过程中输入-1,则标志着数据输入的结束,因此本程序中-1 不能作为输入的数据,只能作为结束的标志。

程序的运行结果如图 7-1 所示。

```

Please input below 10 integer into the sqliist
Type -1 for stopping input
1 2 3 4 5 6 -1
The contents of the sqliist are
1 2 3 4 5 6
The contents of the reversed sqliist are
6 5 4 3 2 1

```

图 7-1 程序 7-1 的运行结果

## 7.2 动态数列排序

### 【题目要求】

编写一个 C 程序，实现这样的功能：从键盘输入任意个整数，以 0 作为结束标志，对这个整数序列从小到大排序，并输出排序后的结果。

### 【题目分析】

要实现动态数列排序首先要选择好数据的存储结构。如果采取静态的线性存储结构，例如数组、静态顺序表等就无法实现动态数列排序的功能。因为静态线性存储结构的内存空间开辟在内存的静态区，它是在程序编译时就分配好了的，因此无法在程序运行时改变空间的大小，也就无法实现动态数列排序的功能。因此可以选择动态的顺序表或者链表作为数据的存储结构。

在这里应用链表作为数据的存储结构。因为链表的存储空间是分配在系统的动态存储区的，因此可以在程序执行时动态地分配内存。这样就可以轻松地解决动态的数列排序问题了。

再一个需要考虑的就是数列排序的算法。在前面的章节中已就各种排序算法进行了详细的介绍，在这里就不再赘述。本题采用经典的冒泡排序。

程序清单 7-2

```

/*----- 7-2.c -----*/
#include <string.h>
#include <stdio.h>
#include <malloc.h>

/*定义 int 为 ElemType 类型*/
typedef int ElemType;

/*定义链表的结点类型*/
typedef struct node{
    ElemType data;        /*数据域*/
    struct node *next;    /*指针域*/
}LNode,*LinkList;

/*创建一个长度为 n 的链表，并输入数据*/
LinkList GreatLinkList(int n){
    LinkList p,r,list=NULL;
    ElemType e;
    int i;

```

```

    for(i=1;i<=n;i++){
        scanf("%d",&e);
        p=(LinkedList)malloc(sizeof(LNode));
        p->data=e;
        p->next=NULL;
        if(!list)
            list=p;
        else
            r->next=p;
        r=p;
    }
    return list;
}

/*向链表中插入结点,并向该结点的数据域中存放数据e*/
void insertList(LinkedList *list,LinkedList q,ElemType e){
    LinkedList p;
    p=(LinkedList)malloc(sizeof(LNode));
    p->data=e;
    if(!*list){
        *list=p;
        p->next=NULL;
    }
    else{
        p->next=q->next;
        q->next=p;
    }
}

/*基于链表的冒泡排序算法*/
void Sort(LinkedList q)
{
    LNode *p=q;
    int t,i,j,k=0;
    while(p){k++; p=p->next;}
    p=q;
    for(i=0;i<k-1;i++)
    {
        for(j=0;j<k-i-1;j++)
        {
            if(p->data>p->next->data)
            {
                t=p->data;
                p->data=p->next->data;
                p->next->data=t;
            }
            p=p->next;
        }
        p=q;
    }
}

/*打印出排序后的新链表中的内容*/
void Print(LinkedList q)
{
    while(q)
    {
        printf("%d ",q->data);
        q=q->next;
    }
}

```



```

}

/*主函数*/
main()
{
    ElemType e;
    LinkList l,q;           /*定义一个链表l*/
    printf("Please input some integer digit and type 0 for end\n");
    q=l=GreatLinkList(l);   /*创建一个链表结点, q和l指向该结点*/

    scanf("%d",&e);
    while(e)                 /*循环地输入数据, 同时插入新生成的结点*/
    {
        insertList(&l,q,e);
        q=q->next;
        scanf("%d",&e);
    }
    Sort(l);
    Print(l);
    getche();
}

```

### 【程序说明】

本程序的执行步骤如下。

(1) 程序首先通过函数 GreatLinkList()创建一个链表的头结点, 并向头结点中输入数据(该链表的头结点存放数据)。

(2) 然后再通过函数 insertList()向链表中插入数据, 在插入数据的同时生成链表本身, 直到输入 0 为止。由于程序是要边输入数据, 边生成链表的结点, 因此应用函数 GreatLinkList()只创建一个结点, 后面的结点用函数 insertList()向链表的尾部插入。

第2章中对链表的 GreatLinkList()和 insertList()操作有详细的说明。其实也可以省略函数 GreatLinkList(), 只通过 insertList()函数生成该链表, 代码如下:

```

l = NULL;
q = NULL;
scanf("%d",&e);           /*输入1个数据*/
insertList(&l ,q, e );     /*生成1个头结点*/
q = l;                     /*q指向链表的第一个结点*/
scanf("%d",&e);
while(e)                   /*循环地输入数据, 同时插入新生成的结点*/
{
    insertList(&l,q,e);
    q=q->next;
    scanf("%d",&e);
}

```

有兴趣的读者可以研究这段代码并替换使用这段代码, 同样可以实现程序 6-2 的功能。

(3) 然后程序通过函数 Sort()实现数列的冒泡排序。

(4) 最后用函数 Print()打印出排序后的新链表中的内容。

本程序的运行结果如图 7-2 所示。

```

Please input some integer digit and type 0 for end
8 3 5 2 7 6 1 0
1 2 3 5 6 7 8

```

图 7-2 程序 7-2 的运行结果

### 7.3 在原表空间进行链表的归并

#### 【题目要求】

有两个按元素值递增有序排列的链表  $l_1$  和  $l_2$ ，编写一个程序将  $l_1$  表和  $l_2$  表归并成一个按元素值递增有序的链表  $l_3$ 。要求（1）链表中允许有相同元素，只要链表  $l_1$ 、 $l_2$ 、 $l_3$  单调不减即可；（2）要利用原表空间（即  $l_1$  表和  $l_2$  表）的结点空间构造表  $l_3$ 。

#### 【题目分析】

首先要创建两个链表  $l_1$  和  $l_2$ ，在输入数据时要按照题目的要求单调不减地输入（例如：1, 1, 2, 3, 3, 4, 5, 6），并用 0 作为输入数据的结束标志。这些操作可按照 5.2 节中介绍的那样通过 GreatLinkList() 和 insertList() 函数边输入数据边生成链表。最终在内存中创建了两个递增有序的链表  $l_1$  和  $l_2$ 。关键的问题是在  $l_1$  表和  $l_2$  表的原空间上有序归并这两个链表。可以通过下面这段代码实现两个链表的递增有序归并：

```
void MergeLink(LinkList l1, LinkList l2, LinkList *l3)
{
    /*将链表 l1, l2 有序归并, l3 指向归并后的新链表*/
    LinkList p, q1, q2;
    q1 = q2 = l2;    /*l3 指向 l2*/
    p = l1;
    while(p != NULL && q1 != NULL) {
        if(p->data >= q1->data) {
            q2 = q1;
            q1 = q1->next;
        }
        else {
            l1 = l1->next;
            insertNode(&q1, &q2, &p, &l2); /*将 p 指向的结点插到 q2 之前 q1 之后*/
            p = l1;
        }
    }
    if(q1 == NULL) q2->next = p;
    *l3 = l2;
}
```

这段代码的功能是将链表  $l_1$  和  $l_2$  进行原空间的递增有序归并，用  $l_3$  指向归并好的链表首地址。因为要修改  $l_3$  的值，因此这里  $l_3$  为指针的传递。本段代码实现链表归并的过程如图 7-3 所示。

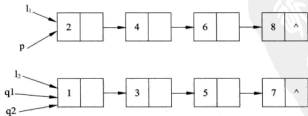


图 7-3 链表的最初状态

如图 7-3 所示,最初内存中有两个链表  $l_1$  和  $l_2$ ,链表中的值都是单调递增的。将指针  $p$  指向  $l_1$ ,将指针  $q_1$ 、 $q_2$  同时指向  $l_2$ 。

归并开始,首先比较  $p \rightarrow data$  与  $q_1 \rightarrow data$  的值。如果  $p \rightarrow data \geq q_1 \rightarrow data$  (如图 7-3 所示),说明  $p$  指向的结点应插入到  $q_1$  指向的结点的后面,但是还不能确定插入的具体位置(因为不能保证  $p$  指向的结点一定小于  $q_1$  的下一个结点),因此执行操作  $q_2 = q_1$ ;  $q_1 = q_1 \rightarrow next$ ;将  $q_1$  指针后移,  $q_2$  指针始终指向  $q_1$  的前一个结点,如图 7-4 所示。

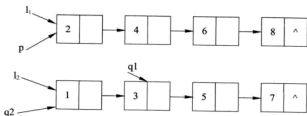


图 7-4  $q_1$ 、 $q_2$  指针后移 (本次操作  $q_2$  未改动)

再次重复上面的  $p \rightarrow data$  与  $q_1 \rightarrow data$  值的比较,直到比较到  $p \rightarrow data < q_1 \rightarrow data$  为止(如图 7-4 所示)。然后通过函数  $insertNode()$  将  $p$  指向的结点插入到  $q_2$  与  $q_1$  结点之间,同时修改指针  $p$  的值。图 7-4 所示的链表合并完  $l_1$  的第一个结点后的状态如图 7-5 所示。

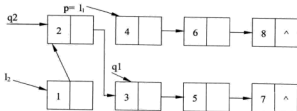


图 7-5 合并完第一个结点后链表的状态

如图 7-5 所示,应用函数  $insertNode()$  将 2 插入到 1 和 3 中间后,  $p$  和  $l_1$  同时指向了原链表  $l_1$  的下一个结点,  $q_2$  指向了新插入的结点,  $q_1$  指向  $q_2 \rightarrow next$  结点,同样  $q_2$  指针始终指向  $q_1$  的前一个结点。

因此不难理解,函数  $MergeLink()$  的作用是将链表  $l_1$  归并 (插入) 到链表  $l_2$  中。指针  $p$  始终指向链表  $l_1$  的头结点用来向链表  $l_2$  插入结点,指针  $q_1, q_2$  作为函数  $insertNode()$  的参数,用来执行将  $p$  指向的结点插入到  $q_1, q_2$  之间的操作。然后再比较  $p \rightarrow data$  与  $q_1 \rightarrow data$  的值,重复上述的操作,直到指针  $p$  为空或者  $q_1$  为空为止。如果  $p$  指针为空,说明  $l_1$  已经完全归并到  $l_2$  中,因此归并操作可以结束,将  $l_2$  的指针赋值给  $l_3$  即可。如果  $q_1$  的指针为空,说明链表  $l_1$  后面还有一段有序链表没有并入到  $l_2$  中,此时只需进行  $q_2 \rightarrow next = p$  操作就可将  $l_2$  完全归并到  $l_1$  中,再将  $l_2$  的指针赋值给  $l_3$  即可。

下面讨论函数  $insertNode()$ 。函数  $insertNode()$  定义如下:

```
void insertNode(LinkList *q1, LinkList *q2, LinkList *p, LinkList *l2) {
    if (*q1 == *q2)
    {
        (*p) -> next = *q2;
```

```

    *l2 = *q2 = *q1 = *p;
}
else
{
    (*q2)->next = *p;
    (*p)->next = *q1;
    (*q2) = (*q2)->next;
}
}

```

它的作用是将  $p$  指针指向的结点插入到  $q1$ 、 $q2$  指向的结点中间。因为函数可能修改指针  $l2$ 、 $q2$ 、 $q1$  本身的值，因此这里使用指针作为参数传递。但是这里有一点例外，如果链表  $l1$  的第一个结点内容小于链表  $l2$  的第一个结点内容，那么在执行 MergeLink() 时，第一步就是将  $p$  指向的结点插入到  $q1$ 、 $q2$  指向的结点中间。但是此时  $q1$ 、 $q2$  都指向  $l2$  的头结点，所以这种情况的插入方式有些特殊。如图 7-6 所示：

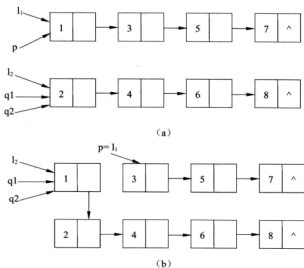


图 7-6 特殊情况的插入方式

在上面的函数 insertNode() 的代码中，if(\*q1 == \*q2) 分支执行的插入操作就是这种特殊的插入操作。在结点插入之前，链表的状态如图 7-6 (a) 所示。将  $p$  指向的结点插入到  $l2$  的头结点前面以后，链表的状态如图 7-6 (b) 所示。else 分支执行的插入操作就是一般的将  $p$  指向的结点插入到  $q1$ 、 $q2$  指向的结点中间的操作，然后将  $q2$  的指针后移。

总之，函数 MergeLink() 实现将  $l1$  归并到  $l2$  中，在函数 MergeLink() 中调用函数 insertNode() 实现结点的具体插入。这样就实现了链表  $l1$  和  $l2$  的原空间递增有序归并。

程序清单 7-3

```

/*----- 7-3.c -----*/
#include "stdio.h"

/*定义 int 为 ElemType 类型*/
typedef int ElemType;

```

```

/*定义链表的结点类型*/
typedef struct node{
    ElemType data;          /*数据域*/
    struct node *next;      /*指针域*/
} LNode, *LinkList;

/*创建一个长度为 n 的链表，并输入数据。此函数用来生成链表的头结点*/
LinkList GreatLinkList(int n){
    LinkList p, r, list=NULL;
    ElemType e;
    int i;
    for(i=1; i<=n; i++){
        scanf("%d", &e);
        p=(LinkList)malloc(sizeof(LNode));
        p->data=e;
        p->next=NULL;
        if(!list)
            list=p;
        else
            r->next=p;
        r=p;
    }
    return list;
}

/*向链表中插入结点，并向该结点的数据域中存放数据 e，此函数用来生成链表时使用*/
void insertList(LinkList *list, LinkList q, ElemType e){
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode));
    p->data=e;
    if(!*list){
        *list=p;
        p->next=NULL;
    }
    else{
        p->next=q->next;
        q->next=p;
    }
}

/*将 p 指向的结点插入到 q1、q2 所指向的结点中间*/
void insertNode(LinkList *q1, LinkList *q2, LinkList *p, LinkList *l2){
    if(*q1 == *q2)
    {
        /*插入结点的特殊情况*/
        (*p)->next = *q2;
        *l2 = *q2 = *q1 = *p;
    }
    else
    {
        /*实现将 p 指向的结点插入到 q1、q2 所指向的结点中间*/
        (*q2)->next = *p;
        (*p)->next = *q1;
        (*q2) = (*q2)->next; /*修改指针 q2，使得 q2 始终指向 q1 的前一个结点*/
    }
}

/*将链表 l1, l2 原空间有序归并，用 l3 返回*/
void MergeLink(LinkList l1, LinkList l2, LinkList *l3)

```

```

{
    /*将链表 l1, l2 有序归并, l3 指向归并后的新链表*/
    LinkList p, q1, q2;
    q1 = q2 = l2; /*l3 指向 l2*/
    p = l1;
    while(p!=NULL && q1!=NULL) {
        if(p->data >= q1->data) {
            q2 = q1;
            q1 = q1->next;
        }
        else {
            l1 = l1->next; /*链表 l1 删除头结点*/
            insertNode(&q1, &q2, &p, &l2); /*将 p 指向的结点插到 q2 之前 q1 之后*/
            p = l1; /*p 指向下一个结点*/
        }
    }

    if(q1 == NULL) q2->next = p; /*将 l1 的剩余尾链归并到 l2 中*/

    *l3 = l2;
}

main()
{
    ElemType e;
    LinkList l1, l2, l3, q;

    printf("Please input the contents of Link1\n");
    /*生成链表 l1*/
    q=l1=GreatLinkList(1); /*创建一个链表结点, q 和 l1 指向该结点*/

    scanf("%d", &e);
    while(e) /*循环地输入数据, 同时插入新生成的结点*/
    {
        insertList(&l1, q, e);
        q=q->next;
        scanf("%d", &e);
    }

    printf("Please input the contents of Link2\n");
    /*生成链表 l2*/
    q=l2=GreatLinkList(1); /*创建一个链表结点, q 和 l2 指向该结点*/

    scanf("%d", &e);
    while(e) /*循环地输入数据, 同时插入新生成的结点*/
    {
        insertList(&l2, q, e);
        q=q->next;
        scanf("%d", &e);
    }

    MergeLink(l1, l2, &l3); /*合并 l2, l2 并用 l3 指向合并后的链表*/

    q = l3;
    printf("The merge of link1 and link 2 is\n");
    while(q) /*打印出合并后的链表*/
    {
        printf("%d ", q->data);
    }
}

```

```

    q = q->next;
}
getche();
}

```

### 【程序说明】

本题的难点在于实现原空间的链表归并,因此需要非常灵活地操纵链表  $L_1$  和  $L_2$  的指针,需要非常清楚链表操作的各个细节,所以本题是有一定难度的。但是如果掌握了本题,会使读者对指针的理解水平、链表的操作能力等都得到提高。因此建议有余力的读者认真分析此题,研究程序源代码,从中得到收获。

本程序的运行结果如图 7-7 所示。

```

Please input the contents of Link1
1 2 4 6 8 9 0
Please input the contents of Link2
2 3 5 6 9 11 0
The merge of link1 and link 2 is
1 2 2 3 4 5 6 6 8 9 9 11 _

```

图 7-7 程序 7-3 的运行结果

## 7.4 约瑟夫环

### 【题目要求】

编号为 1, 2, 3...,  $n$  的  $n$  个人按顺时针方向围坐一圈,每个人手中持有一个密码。开始时任选一个正整数作为报数的上限  $m$ ,从第一个人开始按顺时针方向自 1 开始顺序报数,报到  $m$  停止。报  $m$  的人出列,将他手中的密码作为新的报数上限  $m$ ,从他在顺时针方向上的下一个人开始重新从 1 报数,如此循环报数下去,求最后剩下的那个人的最初编号是多少。

### 【题目分析】

解决约瑟夫环问题,最关键的是要选取好存放数据的数据结构。最简单的方法是使用循环链表作为存储结构,通过链表的删除操作实现报数人的“出列”,通过对链表的循环遍历,实现顺时针“报数”。

链表的结点定义如下:

```

/*链表结点定义*/
typedef struct node{
    int number;      /*编号*/
    int psw;         /*个人密码*/
    struct node *next;
} LNode,*LinkList;

```

定义链表的结点为 LNode 类型, LinkList 为指向链表结点的指针类型。链表中每个结点包含 2 个数据域和 1 个指针域。数据域中分别存放每个人的编号和个人密码。

然后通过函数 CreatJoseph() 创建一个约瑟夫环。代码如下:

```

CreatJoseph(LinkList *jsp, int n)

```

```

{
    LinkList q = NULL, list = NULL;
    int i, e;
    printf("Please input the password for people in the Joseph circle\n");
    for(i=0; i<n; i++){
        scanf("%d", &e);
        insertList(&list, q, i+1, e);          /*向 q 指向的结点后面插入新的结点*/
        if(i == 0) q = list;                  /*第一次只生成头结点, q 也指向头结点*/
        else q = q->next;                     /*q 指向下一结点*/
    }
    q->next = list; /*形成循环链表*/

    *jsp = list;    //返回
}

```

函数 CreatJoseph()的作用是生成一个有  $n$  个结点的约瑟夫环, 并把它的首地址赋值给 jsp。因为在函数中要修改 jsp, 因此这里用指针传递。这里调用 insertList()函数生成一个链表, 它是利用不断向链表尾部插入新结点的方法创建链表的。每输入一个密码就创建一个链表结点, 而每个结点的编号由程序自动生成(从 1, 2 …)。最后通过  $q \rightarrow next = list$  将该单向链表改造成一个循环链表。然后将链表的首地址(指向编号为 1 的结点)首地址赋值给 jsp 返回。创建好的约瑟夫环如图 7-8 所示。

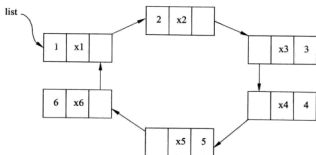


图 7-8 约瑟夫环的初始状态

如图 7-8 所示,  $x_1 \sim x_6$  为要输入的密码, 程序会在用户输入密码时创建每个结点。list 指向编号为 1 的结点。

接下来就是要对这个循环链表从 list 指向的结点开始顺时针地进行“报数-出列”操作。整个操作将不断删除链表中的结点, 最终链表中只剩下 1 个结点, 这个结点的编号就是所要求解的。这个过程由函数 exJoseph()完成。代码如下:

```

exJoseph(LinkList *jsp, int m)
{
    LinkList p, q;
    int i;
    q = p = *jsp;
    while(q->next != p) q = q->next; /*q 指向 p 的前一个结点*/

    printf("The order of a column is\n");
    while(p->next != p){
        for(i=0; i<m-1; i++){
            /*p 指向要删除的结点, q 指向 p 的前一个结点*/

```



```

    q = p;
    p = p->next;
}
q->next = p->next;
printf("%d ",p->number);
m = p->psw;
free(p);
p = q->next;
}
printf("\nThe last person in the circle is %d\n",p->number);
/*打印出最后留在队中的人的编号*/
}

```

函数 `exJoseph()` 有两个参数, `jsp` 是应用 `CreatJoseph` 生成的循环链表的头指针, 指向编号为 1 的单元, `m` 是第一次的报数上限, 以后每次的报数上限由“出列”人的手中的密码决定。整个操作由两个指向链表结点的指针 `p`、`q` 完成。首先将 `p`、`q` 都指向编号为 1 的结点, 然后通过 `while(q->next != p)` `q=q->next`; 循环将 `q` 指针置于 `p` 指针的前一个位置。然后通过二重循环执行约瑟夫环的操作。

外层循环 `while(p->next != p)` 控制循环结束的条件: 直到链表中还剩下一个结点为止。内层循环 `for(i=0; i<m-1; i++)` 按照每次得到的密码规定的循环次数进行, 当循环到指定的位置时, 通过前后两个指针 `q` 和 `p` 将 `p` 指向的结点删除。每删除一个结点, 程序都将其编号输出, 作为“出列”的顺序显示出来, 并将删除的结点的 `psw` 域的值作为下一次的报数上限 `m`。最终链表中只剩下 1 个结点, 其状态如图 7-9 所示。

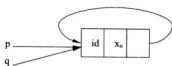


图 7-9 约瑟夫环最终状态

如图 7-9 所示, 指针 `p`、`q` 都指向最后一个结点, 该结点自身指针域中的指针也指向自身。

程序清单 7-4

```

/*----- 7-4.c -----*/
#include "stdio.h"

/*链表结点定义*/
typedef struct node{
    int number; /*编号*/
    int psw; /*个人密码*/
    struct node *next;
} LNode, *LinkList;

/*向链表 list 中 q 指向的结点后面插入一个新结点, 数据域值为 e1、e2*/
void insertList(LinkList *list, LinkList q, int e1, int e2){
    LinkList p;
    p=(LinkList)malloc(sizeof(LNode)); /*生成一个结点*/
    p->number = e1; /*输入数据 1*/
    p->psw = e2; /*输入数据 2*/
}

```

```

    if(!*list){          /*创建头结点*/
        *list=p;
        p->next=NULL;
    }
    else{                 /*插入其他结点*/
        p->next=q->next;
        q->next=p;
    }
}

CreatJoseph(LinkList *jsp, int n)
{
    LinkList q = NULL, list = NULL;
    int i, e2;
    printf("Please input the password for people in the Joseph circle\n");
    for(i=0;i<n;i++){
        scanf("%d",&e);
        insertList(&list,q,i+1,e);    /*向 q 指向的结点后面插入新的结点*/
        if(i == 0) q = list;          /*第一次之生成头结点, q 也指向头结点*/
        else q = q->next;             /*q 指向下一结点*/
    }
    q->next = list;                  /*形成循环链表*/

    *jsp = list;                    /*返回*/
}

exJoseph(LinkList *jsp, int m)
{
    LinkList p, q;
    int i;
    q = p = *jsp;
    while(q->next != p) q=q->next;    /*q 指向 p 的前一个结点*/
    printf("The order of a column is\n");
    while(p->next != p){
        for(i=0;i<m-1;i++){
            /*p 指向要删除的结点, q 指向 p 的前一个结点*/
            q = p;
            p = p->next;
        }
        q->next = p->next;            /*删除 p 指向的结点*/
        printf("%d ",p->number);      /*打印出“出列”的顺序*/
        m = p->psw;                  /*重置报数上限 m*/
        free(p);                    /*释放掉 p 指向的结点*/
        p = q->next;                /*p 指向 q 的下一个结点*/
    }
    printf("\nThe last person in the circle is %d\n",p->number);
    /*打印出最后留在队中的人的编号*/
}

main()
{
    LinkList jsp;
    int n, m;
    printf("Please input number of the people in the Joseph circle\n");
    scanf("%d",&n);                /*输入约瑟夫环的人数*/
    CreatJoseph(&jsp, n);           /*创建约瑟夫环*/
    printf("Please input the first maximum number\n");

```

```

scanf("%d",&m);          /*输入第一次的报数上限m*/
exJoseph(&jsp,m);        /*执行“报数-出列”操作*/
getche();
}

```

本程序的运行结果如图 7-10 所示。

```

Please input number of the people in the Joseph circle
7
Please input the password for people in the Joseph circle
3 1 7 2 4 8 4
Please input the first maximum number
6
The order of a column is
6 1 4 7 2 3
The last person in the circle is 5

```

图 7-10 程序 7-4 的运行结果

## 7.5 二进制/八进制转换器

### 【题目要求】

编写一个程序，要求从终端输入一串 0/1 表示的二进制数，输出它的八进制表示形式。

### 【题目分析】

进行制数转换这类运算最简单的办法是使用栈的数据结构。可以按照如下步骤做：

(1) 首先在输入二进制的 0/1 字符串时，将每次输入的 0 或 1 压入栈 A 中保存。这样栈底存放的是二进制的最高位，栈顶存放的是二进制的最低位。

(2) 在进行二进制/八进制的转换时，要将二进制数的每三位转换成一个八进制数表示，因此可以从栈 A 顶开始逐一将二进制的 0/1 码取出，每取出三位的 0/1 码，就将它们转换成一个对应的八进制数。因为先得到的是八进制数的低位，因此要将该八进制数保存到一个新栈 B 中，直到将栈 A 中的元素取完为止。

(3) 最后将栈 B 中的八进制表示逐一取出显示，就得到了原二进制数的八进制表示。

因此，只要通过对两个动态栈的基本操作（初始化，入栈，出栈等），就可以比较方便地解决二进制/八进制转换的问题。

可以通过下面这样一段代码生成一个空栈，并向栈中输入一个二进制的 0/1 串。

```

typedef char ElemType;
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;

ElemType c;
sqStack s1;
initStack(&s1); /*创建一个栈 s1，用来存放二进制字符串*/
/*输入 0/1 字符串表示的二进制数，以#结束*/
scanf("%c",&c);

```

```

while(c!='#')
{
    if(c=='0' || c=='1')
        Push(&s1,c);
    scanf("%c",&c);
}

```

在源程序中，将 `char` 定义为 `ElemType` 类型。同时定义了一个栈类型 `sqStack`，它是一个结构体类型，包含两个指针域 `base`, `top` 分别指向栈顶和栈底，数据域 `stacksize` 指示栈的当前可使用的最大容量。定义一个 `ElemType` 类型（`char` 类型）变量 `c`，用来接收输入的 0/1 字符。定义一个栈对象 `s1`，用来存放 0/1 串。

先应用函数 `initStack()` 初始化一个空栈，然后从终端输入一个字符（0 或 1），如果该字符为 # 表示输入结束，否则将输入的字符通过 `Push` 操作压入栈中，再输入下一个字符（0 或 1）……直到用户输入字符 # 为止。在输入字符的过程中，只能输入字符 0 或 1，其他字符均为非法输入，不能入栈。假设输入的 0/1 字符串为 10011011，输入完毕后栈的状态如图 7-11 所示。

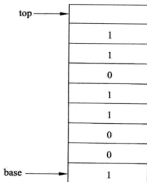


图 7-11 输入完毕后栈 `s1` 的状态

如图 7-11 所示，二进制的 0/1 字符串的高位处于栈底，低位处于栈顶，栈顶指针 `top` 始终指向栈顶元素的上一个空间。

然后通过下面的这样一段代码将栈中 0/1 字符串表示的二进制数转换为八进制表示。

```

sqStack s2;
ElemType c;
initStack(&s2);           /*创建一个栈 s2，用来存放八进制字符串*/
len = StackLen(s1);       /*得到栈中的元素个数，即二进制数的长度*/
for(i=0; i<len; i=i+3){
    for(j=0; j<3; j++){
        Pop(&s1, &c);      /*取出栈顶元素*/
        sum = sum + (c-48) * pow(2, j); /*转换为八进制数*/
        if(s1.base == s1.top) break;
    }
    Push(&s2, sum+48);      /*将八进制数以字符形式压入栈中*/
    sum = 0;
}
while(s2.base != s2.top){ /*输出八进制栈的内容*/
    Pop(&s2, &c);
}

```

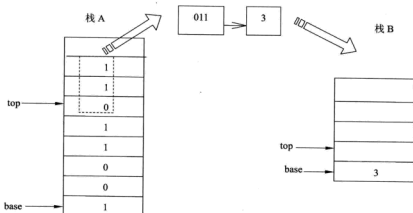
```
printf("%c",c);
}
```

(1) 程序首先创建一个新栈 s2, s2 用来存放八进制的字符串。

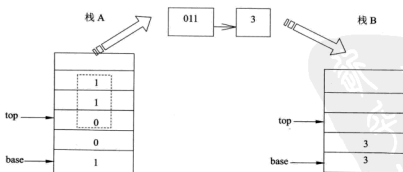
(2) 然后通过一个二重循环将二进制栈 s1 中的内容取出, 转换为八进制字符, 再存入八进制的栈 s2 中。内层的循环每次循环 3 遍, 每次都从二进制栈中的 3 个 0/1 字符取出, 通过  $\text{sum} = \text{sum} + (\text{c}-48) * \text{pow}(2, i);$  语句将这 3 个 0/1 字符串转换为八进制数, 例如将二进制数 “110” 转换为 6。在这里要加上一个判断  $\text{if}(\text{s1.base} == \text{s1.top})$ , 这是因为二进制栈 s1 中的 0/1 字符的个数不一定就是 3 的整数倍, 因此只要判断  $\text{s1.base} == \text{s1.top}$ , 即栈 s1 为空, 就应跳出本循环。

(3) 接下来将转换成的八进制数通过语句  $\text{Push}(\&\text{s2}, \text{sum}+48);$  以字符的形式压入栈 s2 中, 其中参数  $\text{sum}+48$  为八进制数 sum 的字符形式的 ASCII 码。由外层循环控制重复上面的操作, 一共执行  $(1/3) \text{len}$  次, len 为栈 s1 的长度, 也就是刚才输入的 0/1 字符串的长度。

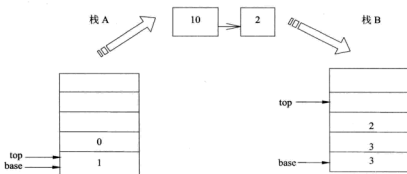
(4) 最后通过一条循环语句将栈 s2 中的内容取出并打印出来, 得到的就是原二进制数的八进制形式。上述过程如图 7-12 所示。



(a) 将栈 A 的 011 转换为 3, 存放到栈 B 中



(b) 将栈 A 的 011 转换为 3, 存放到栈 B 中



(c) 将栈 A 的 10 转换为 2，存放到栈 B 中

图 7-12 二进制转换为八进制的过程

通过如图 7-12 所示的一系列操作，可将二进制栈 A 中的内容取出，转换为十六进制字符，再存放到十六进制栈 B 中。栈 B 中的十六进制数是低位在栈底，高位在栈顶。

程序清单 7-5

```

/*----- 7-5.c -----*/
#include "stdio.h"
#include "math.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10

typedef char ElemType;
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;
/*初始化栈*/
void initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间，首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0); /*分配空间失败*/
    s->top = s->base; /*最开始，栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}
/*入栈操作，将 e 压入栈中*/
void Push(sqStack *s, ElemType e){
    if(s->top - s->base >= s->stacksize){
        /*栈满，追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0); /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
}

```

```

        s->top++;
    }
    /*出栈操作, 用 e 将栈顶元素返回*/
    void Pop(sqStack *s , ElemType *e){
        if(s->top == s->base) return;
        *e = *(s->top);
    }

    /*计算栈 s 的当前长度*/
    int StackLen(sqStack s){
        return (s.top - s.base) ;
    }

    main()
    {
        ElemType c;
        sqStack s1;
        sqStack s2;
        int len,i,j,sum = 0;
        initStack(&s1);          /*创建一个栈 s1, 用来存放二进制字符串*/

        /*输入 0/1 字符表示的二进制数, 以#结束*/
        scanf("%c",&c);
        while(c!='#')
        {
            if(c=='0' || c=='1')
                Push(&s1,c);
            scanf("%c",&c);
        }
        initStack(&s2);          /*创建一个栈 s2, 用来存放八进制字符串*/
        len = StackLen(s1);      /*得到栈中的元素个数, 即二进制数的长度*/
        for(i=0;i<len;i=i+3){
            for(j=0;j<3;j++){
                Pop(&s1,&c);      /*取出栈顶元素*/
                sum = sum + (c-48) * pow(2,j); /*转换为八进制数*/
                if(s1.base == s1.top) break;
            }
            Push(&s2,sum+48);      /*将八进制数以字符形式压入栈中*/
            sum = 0;
        }
        while(s2.base != s2.top ){ /*输出八进制栈的内容*/
            Pop(&s2,&c);
            printf("%c",c);
        }
        getch();
    }

```

本程序的运行结果如图 7-13 所示。

```

Please input a binary number and type '#' for end
10011011#
The Octal from is
233_

```

图 7-13 程序 7-5 的运行结果

**注意：**由于本程序中采用的是动态栈，即堆栈的容量可以随着用户的输入而自动增加，因此它可以实现任意位数的二进制/八进制数制转换。

## 7.6 回文字符串的判定

### 【题目要求】

有一种字符序列正读和反读都相同，这种字符序列被称为“回文”。例如：abba 就是回文。编写一个程序，从键盘输入一个任意长度的字符串，以@作为结束标志，判断该字符串是否是回文。

### 【题目分析】

解决本题的方法很多，可以用一种最为直观、简便的方法，通过一个链队列和一个动态栈来完成。步骤如下：

(1) 在输入字符序列时，把每次输入的字符都分别存放到队列  $q$  中和栈  $s$  中，直到输入字符@为止。

(2) 逐一从栈  $s$  和队列  $q$  中取出字符。由于从栈中取出的字符串与输入的顺序相反，从队列中取出的字符串与输入的顺序相同，因此只要比较每次从栈  $s$  中取出的字符与队列  $q$  中取出的字符是否相等即可。

(3) 重复出栈操作和出队列操作  $\text{len}/2$  次，进行字符的比较。在这个过程中如果出现字符不等的情况，则说明输入的字符串不是回文；如果比较完  $\text{len}/2$  次，且每次比较字符都相等，则说明输入的字符串是回文。

之所以只需要比较栈  $s$  和队列  $q$  的前半部分，是因为队列中的内容与栈中的内容实际上是正好相反的。因此比较队列和栈的前半部分就相当于比较该字符序列的前半部分和后半部分。因此没有必要将整个栈中的内容与队列中的内容全部进行比较。图 7-14 可以说明这一点。

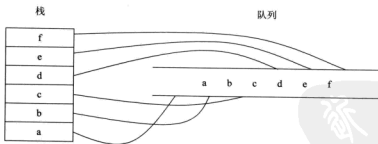


图 7-14 栈和队列中内容的对应关系

如图 7-14 所示，逐一比较栈中的前 3 个元素 (f, a)、(e, b)、(d, c)，就相当于比较原字符序列的前半部分和后半部分，即：第 1 个与第 6 个比较，第 2 个与第 5 个比较，第 3 个与第 4 个比较。这样就可以判断字符序列是否是回文。

可以通过下面的这段代码输入一个字符序列：



```

sqStack s;
LinkQueue q;
ElemType e, r1, r2;
int flag = 1, i, len;
initQueue(&q);
initStack(&s);
printf("Please input a string ,type '@' for end\n");
scanf("%c", &e);
while(e != '@'){
    Push(&s, e);
    EnQueue(&q, e);
    scanf("%c", &e);
}

```

首先通过函数 `initQueue()` 和 `initStack()` 初始化一个空队列 `s` 和一个空栈 `s`。然后通过循环输入一个字符串。每输入一个字符，都将其通过函数 `Push()` 压入栈中，通过函数 `EnQueue()` 进入队列之中，直到输入字符 `@` 为止。

然后通过下面这样一段代码判断刚才输入的字符序列是否是回文。

```

len = StackLen(s)/2;
for(i=0; i<len; i++)
{
    Pop(&s, &r1);
    DeQueue(&q, &r2);
    if(r1 != r2) { flag = 0; break; }
}
if(flag == 1) printf("It is a circle string.\n");
else printf("It is not a circle string.\n");

```

首先应用函数 `StackLen` 计算栈 `s` 的长度。实际上这个长度就是刚才输入的字符串的长度。将这个长度的  $1/2$  赋值给变量 `len` 作为循环次数。接下来通过一个 `for` 循环将栈 `s` 中的元素以及队列 `q` 中的元素依次取出，分别存放到字符变量 `r1` 和 `r2` 中。然后比较 `r1` 和 `r2` 是否相等。一旦 `r1` 不等于 `r2`，则跳出该循环，说明该字符序列不是回文。

以上的代码中调用的函数 `Push()`、`Pop()`、`EnQueue()`、`DeQueue()`、等都是栈和队列结构的基本操作，在前面的章节中都有详细的介绍，这里不再赘述。

程序清单 7-6

```

/*----- 7-6.c -----*/
#include "stdio.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10
typedef char ElemType; /*将 char 类型定义为 ElemType*/

typedef struct QNode { /*定义队列结点类*/
    ElemType data;
    struct QNode *next;
} QNode, *QueuePtr;

typedef struct { /*定义一个链队列*/
    QueuePtr front; /*队头指针*/
    QueuePtr rear; /*队尾指针*/
} LinkQueue;

```

```

typedef struct{                               /*定义一个栈类型*/
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;

initQueue(LinkQueue *q){
    /*初始化一个空队列*/
    q->front = q->rear = (QueuePtr)malloc(sizeof(QNode));
    /*创建一个头结点, 队头队尾指针指向该结点*/
    if( !q->front) exit(0);                  /*创建头结点失败*/
    q->front->next = NULL;                    /*头结点指针域置 NULL*/
}

EnQueue(LinkQueue *q, ElemType e){           /*入队列操作*/
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode));     /*创建一个队列元素结点*/
    if( !q->front) exit(0);                  /*创建头结点失败*/
    p->data = e;                             /*将元素 e 入队列*/
    p->next = NULL;                          /*修改队尾指针*/
    q->rear->next = p;
    q->rear = p;
}

DeQueue(LinkQueue *q, ElemType *e){
    /*如果队列 q 不为空, 删除 q 的队头元素, 用 e 返回其值*/
    QueuePtr p;
    if(q->front == q->rear) return;           /*队列为空, 返回*/
    p = q->front->next;
    *e = p->data;
    q->front->next = p->next;
    if(q->rear == p) q->rear = q->front;      /*如果队头就是队尾, 则修改队尾指针*/
    free(p);
}

initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间, 首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0);                    /*分配空间失败*/
    s->top = s->base;                          /*最开始, 栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE;          /*最大容量为 STACK_INIT_SIZE */
}

Push(sqStack *s, ElemType e){               /*入栈操作*/
    if(s->top - s->base >= s->stacksize){
        /*栈满, 追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0);                /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
}

```

```

    s->top++;
}

Pop(sqStack *s, ElemType *e){          /*出栈操作*/
    if(s->top == s->base) return;        /*将栈顶元素弹出*/
    *e = *(s->top);                      /*修改栈顶指针*/
}

int StackLen(sqStack s){                /*获得栈s的大小*/
    return (s.top - s.base);
}

main()
{
    sqStack s;
    LinkQueue q;
    ElemType e, r1, r2;
    int flag = 1, i, len;
    initQueue(&q);                       /*初始化一个队列*/
    initStack(&s);                       /*初始化一个栈*/
    printf("Please input a string ,type '@' for end\n");
    scanf("%c", &e);
    while(e != '@'){                     /*输入待判断的字符序列*/
        Push(&s, e);
        EnQueue(&q, e);
        scanf("%c", &e);
    }
    len = StackLen(s)/2;                  /*获得字符序列的长度*/
    for(i=0; i<len; i++){
        Pop(&s, &r1);                     /*出栈操作, 由r1将栈顶元素返回*/
        DeQueue(&q, &r2);                 /*出队列操作, 由r2将队头元素返回*/
        if(r1 != r2) { flag = 0; break; }
    }
    if(flag == 1) printf("It is a circle string.\n");
    else printf("It is not a circle string.\n");
    getch();
}

```

本程序的运行结果如图 7-15 所示。

```

Please input a string ,type '@' for end
abbccebba@
It is a circle string.

```

图 7-15 程序 7-6 的运行结果

**注意：**本程序看似代码量不小，实际上大多数都是对栈和队列的基本操作，例如：Push()、Pop()、EnQueue()、DeQueue()等。其实真正需要自己设计思考的并不多。如果把这些基本操作事先定义好，存放到头文件中，每次编程时可以直接引用，这样会大大提高代码的复用率。另外，由于本程序中采用的是动态栈和链队列，因此输入的字符串可以为任意长度。

## 7.7 括号匹配

### 【题目要求】

假设表达式中只允许两种括号：圆括号和方括号，它们可以任意的嵌套，例如[O(O)]都是合法的。但是要求括号必须成对出现，像[( ] )或者( [ ] )的形式都是非法的。编写一个程序，从终端输入一组括号，以字符#作为结束标志，判断输入的括号是否匹配合法。

### 【题目分析】

从题目的要求中不难看出输入的括号可以分为两类：左括号 “[” 和 “(”，和右括号 “)” 和 “]”。由于允许括号的嵌套输入，因此要判断一组括号是否匹配合法，就必须从最里层的括号开始判断。可以应用上面的这些特点设计出括号匹配的算法，这里要用到栈的数据结构。请参看下面这段代码：

```
sqStack s;
char c, e;
initStack( &s );           /*初始化一个空栈*/
scanf("%c",&c);            /*输入第一个字符*/
while(c!='#'){             /*'#'为输入的结束标志*/
    if(!StackLen(s))
        Push(&s,c);        /*如果栈为空，则说明输入的是第一个字符，因此保存在栈中*/
    else
    {
        Pop(&s,&e);         /*取出栈顶元素*/
        if(!match(e,c)){    /*将输入的元素与取出的栈顶进行比较，如果匹配不成功*/
            Push(&s,e);      /*先将原栈顶元素重新入栈*/
            Push(&s,c);      /*再将输入的括号字符入栈*/
        }
        scanf("%c",&c);      /*输入下一个字符*/
    }
}
if(!StackLen(s)) printf("The brackets are matched\n");
/*如果栈s为空，则括号完全匹配*/
else printf("The brackets are not matched\n");
/*如果栈s不为空，则括号不完全匹配*/
```

上面这段代码实现了在输入括号字符序列的过程中，判断括号是否匹配的功能。其步骤如下。

(1) 首先输入1个字符，当该字符不是#时，程序进入到1个while循环之中。

(2) 进入循环中后，首先判断栈是否为空。如果栈为空，即 StackLen(s)等于0，则说明用户输入的括号字符是第一个，因此将它保存在栈中。否则要先将栈顶元素取出，通过函数 match()判断栈顶的元素是否与刚才输入的括号字符相匹配。如果匹配（函数 match()返回1），则不进行其他操作，继续输入下一个字符；如果不匹配（函数 match()返回0），则必须先将刚才取出的栈顶元素 e 重新入栈，再将刚才输入的括号字符 c 也入栈，然后再输入下一个字符。

(3) 重复上面的操作，直到输入结束标志字符#为止。

可以想象得到通过这种方法输入括号，如果括号是完全合法匹配的，最终的栈应当为空。假设输入一组括号 “[ ( ) ]”，按照上面代码描述的算法进行括号输入的过程如图 7-16 所示。

如图 7-16 所示，图 (a) 为初始化一个空栈；图 (b) 表示输入了第一个括号字符 “[”，系统将其入栈；图 (c) 表示用户输入第二个字符 “(”，系统将栈顶元素取出并通过函数 match() 与输入的 “(” 进行匹配比较，匹配不成功；图 (d) 为将原先的栈顶元素 “[” 和输入的字符 “(” 一同入栈；图 (e) 表示用户输入第三个字符 “)” ，系统将栈顶元素取出并通过函数 match() 与输入的 “)” 进行匹配比较，匹配成功；图 (f) 表示用户输入第四个字符 “]”，系统将栈顶元素取出并通过函数 match() 与输入的 “]” 进行匹配比较，匹配成功。

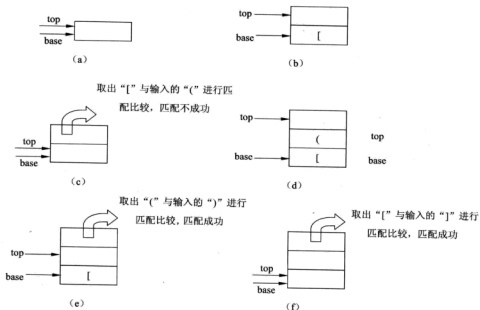


图 7-16 括号输入的过程

在这里函数 match() 的功能是判断栈顶的符号元素与输入的括号是否匹配。这个判断非常简单，即 “(” 与 “)” 相匹配； “[” 与 “]” 相匹配，其他不匹配。算法描述如下：

```
int match(char e, char c) {
    /* 比较栈顶元素 e 和输入的括号字符 c 是否匹配 */
    if (e == '(' && c == ')') return 1;
    if (e == '[' && c == ']') return 1;
    return 0;
}
```

从整个操作过程不难看出，如果输入的括号都可以匹配成功，那么最终栈是为空的。相反，如果输入的括号不能完全匹配，最终栈中一定还有元素未被取出。

#### 程序清单 7-7

```
/*----- 7-7.c -----*/
```

```

#include "stdio.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10
typedef char ElemType;          /*将 char 类型定义为 ElemType*/
typedef struct                  /*定义一个栈类型*/
{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;

initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间, 首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0);        /*分配空间失败*/
    s->top = s->base;              /*最开始, 栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}

Push(sqStack *s, ElemType e){    /*入栈操作*/
    if(s->top - s->base >= s->stacksize){
        /*栈满, 追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0);    /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e;                /*放入数据*/
    s->top++;
}

Pop(sqStack *s, ElemType *e){    /*出栈操作*/
    if(s->top == s->base) return; /*将栈顶元素弹出*/
    *e = *(s->top);               /*修改栈顶指针*/
}

int StackLen(sqStack s){         /*获得栈 s 的大小*/
    return (s.top - s.base);
}

int match(char e, char c){       /*比较栈顶元素 e 和输入的括号字符 c 是否匹配*/
    if(e=='(' && c==')') return 1;
    if(e=='[' && c==']') return 1;
    return 0;
}

main()
{
    sqStack s;
    char c, e;
    initStack(&s);                /*初始化一个空栈*/
    scanf("%c", &c);              /*输入第一个字符*/
    while(c!='#'){
        if(!StackLen(s))
            Push(&s, c);          /*如果栈为空, 则说明输入的是第一个字符, 因此保存在栈中*/
    }
}

```

```

else
{
    Pop(&s,&e);          /*取出栈顶元素*/
    if(!match(e,c)){ /*将输入的元素与取出的栈顶进行比较, 如果匹配不成功*/
        Push(&s,e);      /*先将原栈顶元素重新入栈*/
        Push(&s,c);      /*再将输入的括号字符入栈*/
    }
    scanf("%c",&c);      /*输入下一个字符*/
}
if(!StackLen(s)) printf("The brackets are matched\n");
/*如果栈 s 为空, 则括号完全匹配*/
else printf("The brackets are not matched\n");
/*如果栈 s 不为空, 则括号不完全匹配*/
getche();
}

```

本程序的运行结果如下。图 7-17 为输入的括号相匹配的情况。

```

[[<>]][]<>]#
The brackets are matched

```

图 7-17 输入的括号相匹配

图 7-18 为输入的括号不相匹配的情况。

```

[[<<]]>]#
The brackets are not matched

```

图 7-18 输入的括号不相匹配

## 7.8 魔王语言翻译

### 【题目要求】

传说有一个魔王使用自己的语言说话, 没人能够听得懂他的语言。后来从国外来了一位智者, 他发现魔王的语言可以逐步翻译成人能听懂的语言, 因为他的语言是由以下两种形式的规则由人的语言逐步抽象上去的。

$$(1) \alpha \rightarrow \beta_1 \beta_2 \beta_3 \cdots \beta$$

$$(2) (\theta \delta_1 \delta_2 \cdots \delta_n) \rightarrow \theta \delta_n \theta \delta_{n-1} \cdots \theta \delta \theta$$

上面的规则中, 从左到右表示将魔王语言翻译成人类的语言。魔王语言和人类语言按照该语法规则进行转换。设大写字母表示魔王语言词汇, 小写字母表示人类语言词汇。上述的希腊文法式中, 希腊字母表示可以用大写字母或小写字母代换的变量。魔王语言可以包含人类的词汇。

$$(1) B \rightarrow tAdA$$

$$(2) A \rightarrow sac$$

编写一个魔王语言的翻译系统, 把魔王的话翻译成人类的语言。

## 【题目分析】

首先要搞懂魔王语言的语法规则。根据题目可知，魔王语言的组成是这样的：由大写字母组成；由小写字母组成；由大写小写字母混合组成；由包含括号的字母组成。其中大写字母与小写字母的对应关系为：B→tAdA；A→sae。因此魔王语言中的大写字母只能包含A和B两个。例如下面的魔王语言对应的翻译如下。

B 翻译为：tsaedsae；

AB 翻译为：saetsaedsae；

(abc)翻译为：acaba；

BabcA 翻译为：tsaedsaeabcsae；

AB(esacd)B 翻译为：saetsaedsaeedeceaeetsaedsae；

A(Aasc)B 翻译为：saesaecsaessaesaetsaedsae。

理解了魔王的语法规则，就可以设计一个魔王语言编译器来解释魔王语言，将它翻译为只有小写字母组成的人类语言。

可以这样考虑魔王语言的翻译。将输入的魔王语言从右至左进栈，每次从栈顶弹出一个元素（大写字母、小写字母、括号），对该元素进行如下处理：

输入：一个符合魔王语言语法的元素，包括小写字母、大写字母‘A’或‘B’、括号‘(’。

If 小写字母 then 直接输出之

Else If 大写字母 then

    If 'A' then 输出 sae

    Else If 'B' then 输出 tsaedsae

Else If '(' then 将括号里面的内容从栈中取出，按语法规则的顺序从右至左进入一个新栈，再每次从栈顶弹出一个元素，递归调用该处理过程。

这样不断地从栈顶取出元素进行上面的递归操作，最终可将栈中的魔王语言全部取出翻译。上述算法可通过下面这段代码实现。

```
void translate(ElemType e, sqStack *s){
    ElemType c, a;
    sqStack ssl;
    if(e>=97 && e<=122) printf("%c",e);
    else if(e == 'A') printf("%s","sae");
    else if(e == 'B') printf("%s","tsaedsae");
    else if(e == '(')
    {
        initStack(&ssl);          /*初始化栈 ssl，用来将括号里面的内容从栈中取出*/
        Pop(&(*s),&c);            /* 注意这里的*s相当于主函数中的s。&(*s)相当于主函数
                                   中的&s。请读者自己考虑其中的原因 */
        a = c;                    /*保留括号后的第一个元素*/
        Pop(&(*s),&c);
        while(c!=''){
            Push(&ssl,a);
            Push(&ssl,c);
            Pop(&(*s),&c);
        }
        Push(&ssl,a);              /*按语法规则排列从右至左进入一个新栈 ssl*/
        while(StackLen(ssl))      /*翻译括号里的内容*/
        {
            Pop(&ssl,&c);          /*取出 ssl 中的元素 c*/
```



```

    translate(c, &ss1);    /*递归地调用函数 translate 对元素 c 进行翻译*/
  }
}

```

函数 `translate` 是一个递归函数，实现将参数 `e` 表示的魔王语言翻译成为人类语言。这里重点讲解当 `e='('` 时程序的执行过程。

当 `e='('`，说明接下来括号里的内容要按照

$$(\theta\delta_1\delta_2\cdots\delta_n) \rightarrow \theta\delta_n\theta\delta_{n-1}\cdots\theta\delta_1\theta$$

的规则进行翻译。

程序首先创建一个新栈 `ss1`，取出括号后面的第一个元素，保存到变量 `a` 中，如图 7-19 所示。

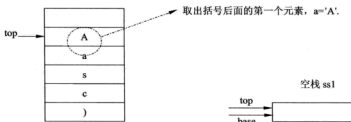


图 7-19 取出括号后面的第一个元素，并创建新栈 `ss1`

然后程序通过一个循环操作将括号中的字符按语法规则排列从右至左进入新栈 `ss1` 中。该循环直到取出的元素是 `)` 为止，如图 7-20 所示。

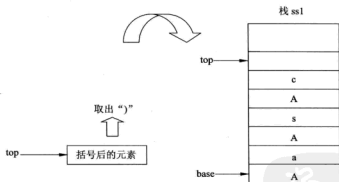


图 7-20 将括号中的的字符按语法规则排列从右至左进入新栈 `ss1` 中

在这个过程中，每从栈 `s` 中取出一个元素就压入栈 `ss1` 中，再将保存在变量 `a` 中的括号后面的第一个元素压入栈中，交替进行。

按照语法规则，最后还要将括号后面的第一个元素再次压入栈中。这样再从栈 `ss1` 的栈顶取出元素，递归地调用函数 `translate()`，进行括号中的魔王语言的翻译。

之所以采用这种递归的方法翻译魔王语言，原因在于魔王语言本身的语法规则。由于

$$(\theta\delta_1\delta_2\cdots\delta_n)\rightarrow\theta\delta_n\theta\delta_{n-1}\cdots\theta\delta_1\theta$$

中的希腊字母表示可以用大写字母或小写字母代换的变量，因此如果括号中存在大写字母，要对大写字母进行翻译，它本身就是再次调用 `translate()` 函数的过程。如果不使用递归，而是直接对字母 A 或 B 进行判断翻译也是可以的，但是代码量要大一些，程序会显得繁冗。

程序清单 7-8

```

/*----- 7-8.c -----*/
#include "stdio.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10

typedef char ElemType; /*将 char 类型定义为 ElemType*/
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;

initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间，首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0); /*分配空间失败*/
    s->top = s->base; /*最开始，栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}

Push(sqStack *s, ElemType e){ /*入栈操作*/
    if(s->top - s->base >= s->stacksize){
        /*栈满，追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0); /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
    s->top++;
}

Pop(sqStack *s, ElemType *e){ /*出栈操作*/
    if(s->top == s->base) return; /*将栈顶元素弹出*/
    *e = --(s->top); /*修改栈顶指针*/
}

int StackLen(sqStack s){ /*获得栈 s 的大小*/
    return (s.top - s.base);
}

void translate(ElemType e, sqStack *s){

```

```

ElemType c, a;
sqStack ssl;
if(e>=97 && e<=122) printf("%c",e);
else if(e == 'A') printf("%s", "sae");
else if(e == 'B') printf("%s", "tsaedsae");
else if(e == '(')
{
    initStack(&ssl);          /*初始化栈 ssl, 用来将括号里面的内容从栈中取出*/
    Pop(&(*s), &c);           /*注意*s 相当于主函数中的 s*/
    a = c;                    /*保留括号后的第一个元素*/
    Pop(&(*s), &c);
    while(c!=' '){
        Push(&ssl, a);
        Push(&ssl, c);
        Pop(&(*s), &c);
    }
    Push(&ssl, a);             /*并按语法规则排列后从右至左进入一个新栈 ssl*/
    while(StackLen(ssl)) /*翻译括号里的内容*/
    {
        Pop(&ssl, &c);         /*取出 ssl 中的元素 c*/
        translate(c, &ssl);    /*递归地调用函数 translate 对元素 c 进行翻译*/
    }
}
}

main()
{
    ElemType e;
    sqStack s1, s2;
    initStack(&s1);            /*初始化栈 s1*/
    printf("Please input Devil language:\n");
    scanf("%c", &e);
    while(e != '#')
    {
        if(e=='A' || e=='B' || (e>=97 && e<=122) || e=='(' || e==' '){
            /*输入的魔王语言合法*/
            Push(&s1, e);       /*入栈*/
        }
        scanf("%c", &e);
    }

    initStack(&s2);            /*初始化栈 s2*/
    while(StackLen(s1))
    {
        Pop(&s1, &e);
        //printf("%c\n", e);
        Push(&s2, e);          /*将魔王语言从右至左入栈 s2*/
    }
    printf("The mankind language is:\n");
    /** 最终输入的魔王语言存放在栈 s2 中, 处理栈顶元素进行翻译*/
    while(StackLen(s2))
    {
        Pop(&s2, &e);
        translate(e, &s2);
    }
    getch();
}

```

本程序的运行结果如图 7-21 所示。

```
Please input Devil language:
a(Asc)EW
The mankind language is:
saesaecsaessaeasaetsaedsae
```

图 7-21 程序 7-8 的运行结果

## 7.9 动态双向链表的应用

### 【题目要求】

设有一个双向循环链表，每个结点中除了有 `prior`、`data` 和 `next` 3 个域外还有一个访问频度域 `freq`。在该双向链表启用之前，频度域 `freq` 的值全部初始化为 0，每当对链表进行一次访问时，被访问的结点的频度域 `freq` 就自动增加 1，同时调整链表中结点之间的次序，使其按照访问频度非递增的次序顺序排列，以便始终保持被访问的结点总是靠近表头结点。编写一个程序实现上述功能。

### 【题目分析】

首先必须知道什么是双向链表，如图 7-22 所示为一个双向链表。



图 7-22 双向链表示意

如图 7-22 所示双向链表中的每个结点都至少包含 3 个域：数据域、`next` 指针域、`prior` 指针域。其中，`next` 指针域存放链表后继结点的指针，`prior` 指针域存放链表的前序结点的指针。因此对双向链表的结点访问更加灵活，因为只要得到链表中一个结点的指针，就可以访问到整个链表的所有结点。对本题而言，可以这样定义一个双向链表。

```
typedef struct node{
    int data;
    int freq;
    struct node *prior;
    struct node *next;
}dbLNode,*dbLinkedList;
```

定义 `dbLNode` 为链表结点的类型，定义 `dbLinkedList` 为指向该双向链表结点的类型。每个结点中包含 4 个域：`data` 域存放数据，`freq` 域存放该结点的访问频度，`prior` 指向前序结点，`next` 指向后继结点。

首先要创建一个双向链表。为了方便起见，约定创建一个带头结点的双向链表，即双向链表的第一个结点中不存放任何内容，从第二个结点开始才正式被使用。可以通过下面这段代码创建一个双向链表。

```
/*创建一个双向链表，返回它的头指针*/
dbLinkedList GreatdbLinkedList(int n){
    dbLinkedList p,r,list=NULL;
```

```

int e;
int i;
list = (dbLinkedList)malloc(sizeof(dbLNode));
/*创建头结点 head, 头结点中不存放内容*/
list->next = list->prior = NULL;
for(i=1; i<=n; i++){
    scanf("%d", &e);
    p = (dbLinkedList)malloc(sizeof(dbLNode));
    p->data = e;
    p->freq = 0;
    p->next = NULL;
    p->prior = NULL;
    if(!list->next) {
        list->next = p; /*第二个结点*/
        p->prior = list;
    }
    else{
        r->next = p; /*双向连接下面的结点*/
        p->prior = r;
    }
    r = p;
}
return list;
}

```

通过上面这段代码可以创建一个包含  $n$  个元素, 加上头结点共  $n+1$  个结点的双向链表。程序首先创建一个头结点, 并用 `list` 指向该结点, `list` 最终要作为返回值返回。头结点的 `next` 域和 `prior` 域都置为 `NULL`。然后通过  $n$  次的循环创建每一个结点, 每个结点的数据 `data` 由用户输入, `freq` 中的值自动初始化为 0。与创建单链表有所不同, 每次创建一个结点都用 `p` 指向该结点, `r` 指向前一个结点, `r` 指向结点的 `next` 域要存放 `p` 的值 (新创建的结点的地址), 同时 `p` 指向结点的 `prior` 域也要存放 `r` 的值 (前一个结点的地址)。这样才能实现双向链表的结构。最终将双向链表的头指针 `list` 返回。图 7-23 为通过上述代码创建的双向链表的示意。



图 7-23 创建好的双向链表

根据题目的要求, 需要编写一个函数 `visit()`, 通过函数 `visit()` 访问该双向链表中的指定结点, 并修改该结点中的 `freq` 值, 同时调整该结点在链表中的次序, 使之更靠近表头。可以用下面这段代码描述函数 `visit()`。

```

/*访问双向链表中指定的结点, 并调整结点次序*/
visit(dbLinkedList *l, int x){
    dbLinkedList p = *l, q, r, s;
    p = p->next; /*p 指向第二个结点*/
    while(p->data != x) p = p->next; /*通过循环找到指定的结点*/
    if(p == NULL) { printf("Input error!\n"); return; } /*找不到指定的结点, 程序报错*/
    p->freq++;
}

```

```

printf("Visiting this node\n");
while((p->prior)!=*l && p->freq > p->prior->freq)
{
    /*实现双向链表结点的交换*/
    q = p->prior;
    r = p->next;
    s = p->prior->prior;
    p->prior = q->prior;
    p->next = q;
    q->prior = p;
    q->next = r;
    r->prior = q;
    s->next = p;
}
}

```

首先通过一个循环操作找到指定的结点，该结点中的数据内容要等于函数 `visit()` 的参数 `x`。因此本程序要求创建双向链表时输入的数据都应当各不相同。如果找不到指定的结点，程序报错，`visit` 操作结束，需要重新输入 `x` 的值。找到该结点后，则通过语句 `p->freq++`；修改 `freq` 中的访问频度值，通过 `printf("Visiting this node\n");` 标志已访问该结点。在实际的程序开发中，或许要对该结点进行更为复杂的访问操作，但这里只用一个输出语句代替。接下来就是本程序中的一个难点：依据结点的访问频度，调整结点在双向链表中的位置。在代码中通过一个循环操作实现，每次的循环操作都将 `p` 指向的结点与它的前序结点进行交换。

首先必须清楚该循环的执行条件：`while((p->prior)!=*l && p->freq > p->prior->freq)`。第一个条件是 `(p->prior)!=*l`。如果 `(p->prior) == *l`，其状态如图 7-24 所示。

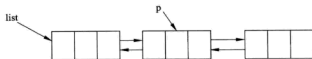


图 7-24 状态 `(p->prior) == *l`

如图 7-24 所示，在该状态下循环调整就应当结束。因为此时 `p->prior` 等于 `list`，而 `list` 指向的头结点并不存放任何内容，因此没有比较的意义。所以，第一个循环执行条件是 `(p->prior)!=*l`。

第二个循环执行条件是 `p->freq > p->prior->freq`，这个很好理解，就是如果 `p` 指向的前序结点的 `freq` 小于 `p` 指向的当前结点的 `freq` 值。也就是说，`p` 指向的当前结点的访问次数已经超过了它的前一个结点，因此要进行结点的交换。这里要注意一点，每次调整结点的次序，结点只需跟它的前序结点中的 `freq` 比较即可，无需考虑它的后继结点的 `freq` 值。这是因为该结点的后继结点的 `freq` 值（即访问频度）始终小于或等于该结点。

在清楚了循环的执行条件之后，就要看如何实现双向链表结点的交换了。

如果仅靠指针 `p` 实现两结点的交换是相当困难的，很容易混乱，这是因为看似简单的双向链表两结点交换，实际上要修改 6 个指针域的值，如图 7-25 所示。

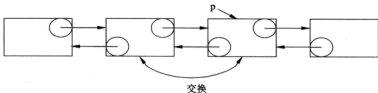


图 7-25 双向链表结点交换要修改的指针

如图 7-25 所示，“圆圈”标志的就是需要修改的结点的指针域。虽然只要进行两个结点（ $p$  指向的结点和它的前序结点）的位置交换，但是需要修改指针域的结点多达 6 个。因此如果仅靠指针  $p$  实现两结点的交换是相当困难的。最简单的办法就是多设几个指针分别指向这 4 个结点，再进行指针的调换，这样问题就简单了许多。在本程序中，设指针  $q$  指向  $p$  结点的前序结点，指针  $r$  指向  $p$  结点的后继结点，指针  $s$  指向  $q$  结点的前序结点。通过如代码所示的指针操作就可以实现两结点的交换。

执行循环操作，直到不满足循环执行条件为止，这样就能实现将  $\text{freq}$  值大的（访问频度高的）结点置于双向链表的头部，即始终保持被访问的结点总是靠近表头结点。下面给出完整的程序清单。

程序清单 7-9

```

/*----- 7-9.c -----*/
#include "stdio.h"
/*定义双向链表类型*/
typedef struct node{
    int data;
    int freq;
    struct node *prior;
    struct node *next;
}dbLNode,*dbLinkList;

/*创建一个双向链表，返回它的头指针*/
dbLinkList GreatdbLinkList(int n){
    dbLinkList p,r,list=NULL;
    int e;
    int i;
    list = (dbLinkList)malloc(sizeof(dbLNode));
    list->next = list->prior = NULL;
    for(i=1;i<=n;i++){
        scanf("%d",&e);
        p = (dbLinkList)malloc(sizeof(dbLNode));
        p->data = e;
        p->freq = 0;
        p->next = NULL;
        p->prior = NULL;
        if(!list->next){
            list->next = p; /*第一个结点*/
            p->prior = list;
        }
        else{
            r->next = p; /*双向连接下面的结点*/
            p->prior = r;
        }
    }
}

```

```

        }
        r = p;
    }
    return list;
}

/*访问双向链表中指定的结点,并调整结点次序*/
visit(dbLinkList *l, int x){
    dbLinkList p = *l, q, r, s;
    p = p->next;    /*p 指向第一个结点*/
    while(p->data != x) p = p->next;    /*通过循环找到指定的结点*/
    if(p == NULL) { printf("Input error!\n"); return; }
    /*找不到指定的结点,程序报错*/

    p->freq++;
    printf("Visiting this node\n");
    while((p->prior)!=*l && p->freq > p->prior->freq)
    {
        /*实现双向链表结点的交换*/
        q = p->prior;
        r = p->next;
        s = p->prior->prior;
        p->prior = q->prior;
        p->next = q;
        q->prior = p;
        q->next = r;
        r->prior = q;
        s->next = p;
    }
}

TransdbLinkList(dbLinkList l)
{
    /*遍历整个双向链表,并打印出每个结点中的数据和访问频度*/
    l = l->next;
    while(l!=NULL){
        printf("(data:%d ,freq:%d)--> ", l->data, l->freq);
        l = l->next;
    }
    printf("X\n\n");
}

main()
{
    dbLinkList l;
    int d;
    printf("Input five integer to creat a doubly link list\n");
    l = GreatdbLinkList(5); /*创建一个包含5个元素,6个结点的双向链表*/
    TransdbLinkList(l);    /*遍历双向链表,打印出每个结点中的数据和访问频度*/
    printf("Please input the data that you want to vist\n");
    scanf("%d",&d);
    while(d!=0){
        visit(&l,d);
        TransdbLinkList(l);
        scanf("%d",&d);
    }
    getche();
}

```

### 【程序说明】

本程序中创建一个包含 5 个元素, 6 个结点的双向链表, 通过函数 TransdbLinkList()



遍历整个双向链表，并打印出每个结点中的数据和访问频度。再通过函数 visit()访问指定的结点。所谓指定的结点就是该结点中的 data 值等于 visit 的参数 d。每访问一次指定的结点，就调用 TransdbLinkedList()函数显示当前双向链表中结点的次序，从而验证结点调整的正确性。循环执行上述操作，直到输入 0 为止。

本程序的运行结果如图 7-26 所示。

```

Input five integer to create a doubly link list
1 2 3 4 5
(data:1 ,freq:0)--> (data:2 ,freq:0)--> (data:3 ,freq:0)--> (data:4 ,freq:0)-->
(data:5 ,freq:0)--> K
Please input the data that you want to visit
2
Visiting this node
(data:2 ,freq:1)--> (data:1 ,freq:0)--> (data:3 ,freq:0)--> (data:4 ,freq:0)-->
(data:5 ,freq:0)--> K
3
Visiting this node
(data:2 ,freq:1)--> (data:3 ,freq:1)--> (data:1 ,freq:0)--> (data:4 ,freq:0)-->
(data:5 ,freq:0)--> K
4
Visiting this node
(data:2 ,freq:2)--> (data:3 ,freq:1)--> (data:1 ,freq:0)--> (data:4 ,freq:0)-->
(data:5 ,freq:0)--> K
5
Visiting this node
(data:2 ,freq:2)--> (data:3 ,freq:1)--> (data:1 ,freq:0)--> (data:4 ,freq:0)-->
(data:5 ,freq:1)--> K
0

```

图 7-26 程序 7-9 的运行结果

**注意：**

双向链表中的内容 1, 2, 3, 4, 5;  
访问 2, 链表调整为 2, 1, 3, 4, 5;  
访问 3, 链表调整为 2, 3, 1, 4, 5;  
访问 3, 链表调整为 3, 2, 1, 4, 5。

## 7.10 判断完全二叉树

### 【题目要求】

完全二叉树的定义是这样的：深度为  $k$  的，有  $n$  个结点的二叉树，当且仅当其每一个结点都与深度为  $k$  的满二叉树的中编号为  $1 \sim n$  的结点相对应，则它被称为完全二叉树。所谓满二叉树就是：深度为  $k$  且有  $2^k - 1$  个结点的二叉树。其中，结点的编号约定为：编号从根结点起从上至下，自左向右地顺序编号。例如图 7-27 为一棵满二叉树。

如图 7-27 所示，该二叉树共有 3 层，共包含  $2^3 - 1 = 7$  个结点，因此它是满二叉树。直观地讲，满二叉树就是除了叶子结点外，其他全部结点都有左孩子和右孩子结点的二叉树。那么图 7-28 就是一棵完全二叉树。

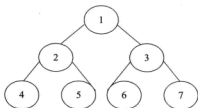


图 7-27 含 7 个结点的满二叉树



图 7-28 含 6 个结点的完全二叉树

因为该二叉树的6个结点编号与同层数满二叉树结点编号一一对应,因此它是一棵完全二叉树。而图7-29所示的二叉树不是完全二叉树。

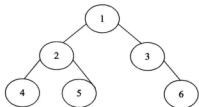


图 7-29 含 6 个结点的非完全二叉树

因为该二叉树中的6个结点编号与同层数满二叉树结点编号不能一一对应,因此它不是完全二叉树。

编写一个程序,判断一棵二叉树是否是完全二叉树。

#### 【题目分析】

要判断一棵二叉树是否是完全二叉树,必须找到完全二叉树的判定方法。满足什么条件的二叉树才是完全二叉树呢?通过上面介绍的二叉树的定义,以及观察完全二叉树结构的特点不难发现,一棵完全二叉树必须具备以下结构上的条件。

(1) 如果一棵完全二叉树的深度(从根结点到最底层的叶结点的层数)为 $k$ ,那么第 $k-1$ 层中的结点中,一旦有一个结点的右孩子为空,则同层中后继编号结点的左右孩子都为空,如图7-30(a)所示;一旦有一个结点的左孩子为空,该结点的右孩子,以及同层中后继编号结点的左右孩子都为空,如图7-30(b)所示。

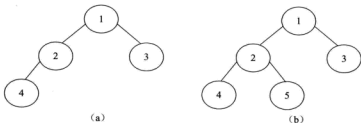


图 7-30 完全二叉树的特点

(2) 如果一棵完全二叉树的深度为 $k$ ,那么第1层至第 $k-2$ 层中的全部结点都必须既有左孩子又有右孩子,即第1层至第 $k-1$ 层中的全部结点构成一棵满二叉树。如图7-31所示的二叉树就不是完全二叉树。

条件(1)、(2)构成判定完全二叉树的充分必要条件。因此可以将条件(1)、(2)设计成算法用来判断一棵二叉树是否是完全二叉树。可以通过遍历二叉树的方法来判断第 $k-1$ 层的结点是否满足条件(1),同时判断第 $1 \sim (k-2)$ 层的结点是否满足条件(2)。下面这段代码实现了上述功能。

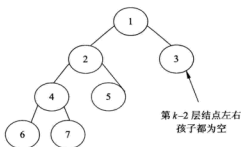


图 7-31 非完全二叉树的特点

```

int JusticBiTree(BiTree T,int level ,int n,int flag){
    if(!T){
        return 1;
    }
    if(level == n)
    {
        /*当前访问的结点处于二叉树T中的k-1层*/
        if(T->lchild == NULL && T->rchild != NULL) return 0;

        if(flag == 0){/*同层的前面的结点无空指针*/
            if(T->rchild == NULL) flag = 1; /*出现空指针*/
        }
        else if(flag == 1){ /*同层的前面的结点有空指针*/
            if(T->lchild!=NULL || T->rchild!=NULL) return 0;
        }
    }
    if(level != n && level !=n+1)
    {
        if(T->lchild == NULL || T->rchild == NULL) return 0;
    }
    if(!JusticBiTree(T->lchild,level+1,n,flag)) return 0;
    if(!JusticBiTree(T->rchild,level+1,n,flag)) return 0;
    return 1;
}

```

这段代码实际是通过遍历二叉树来判断结点指针域的。函数 JusticBiTree() 包括 4 个参数: T 为二叉树根结点指针, level 为当前访问的结点所在二叉树中的层数, n 的值为  $k-1$ , 其中  $k$  为二叉树的深度, flag 为一个标志变量, 用来记录第  $k-1$  层的结点的右孩子是否为空。函数 JusticBiTree() 的返回值类型为 int, 返回 1 表示该二叉树是完全二叉树, 返回 0 表示不是完全二叉树。

首先判断二叉树是否为空树, 如果是空树, 返回 1。否则:

如果 level 等于 n, 即当前访问的结点处于二叉树 T 中的  $k-1$  层, 则对该结点的左孩子和右孩子进行判断: (1) 如果该结点左孩子为空, 右孩子不为空, 则该二叉树一定不是完全二叉树, 因为它违背判定条件 (1)。(2) 在 flag 等于 0, 即同层 (第  $k-1$  层) 的前面的结点无空指针 (都既有左孩子又有右孩子) 的条件下, 如果当前访问的结点右孩子为空, 则 flag 置 1, 表明第  $k-1$  层中存在着包含空指针的结点。(3) 在 flag 等于 1, 即同层 (第  $k-1$  层) 的前面的结点有空指针 (存在着包含空指针的结点) 的条件下, 如果当前访问的结点左孩子不为空或者右孩子不为空, 则说明该二叉树一定不是完全二叉树, 因为它违背判定条件 (1)。

如果 level 不等于 n, 也不等于 n+1, 即不是第 k 层和第 k-1 层, 则说明当前访问的结点是第 1~(k-2) 层的结点, 根据判定条件 (2), 如果该结点的左孩子或右孩子有一个为空, 则该二叉树一定不是完全二叉树。

接下来递归地判断后续结点 (包括第 k-1 层后续的结点以及第 1~(k-2) 层的后续结点)。如果在判断过程中发现某个结点的左 (右) 孩子不符合相应的判定条件, 则程序立刻返回 (return 0), 递归结束。如果程序顺利地遍历完整棵二叉树, 则说明该二叉树是完全二叉树。

在这个判断完全二叉树的过程中需要一个重要的参数 n, 即 k-1、k 为二叉树的深度。因此在创建二叉树时就应当记录下该二叉树的深度, 作为函数 JusticBiTree() 的一个参数。可通过以下方式创建二叉树。

```

CreatBiTree(BiTree *T, int *level1, int level2){
    char c;
    scanf("%c",&c);
    if(c == ' ') *T = NULL;
    else{
        *T = (BiTreeNode *)malloc(sizeof(BiTreeNode));    /*创建根结点*/
        (*T)->data = c;    /*向根结点中输入数据*/
        if(*level1 < level2) {*level1 = level2;}
        CreatBiTree(&((*T)->lchild), &(*level1), level2+1);    /*递归地创建左子树*/
        CreatBiTree(&((*T)->rchild), &(*level1), level2+1);    /*递归地创建右子树*/
    }
}

```

在这里用变量 level1 记录二叉树的深度, 变量 level2 随着遍历过程不断改变, 它始终记录的是当前访问的结点在二叉树中的深度。因此 level1 一定是 level2 中最大的那一个。由于 level1 是要返回的参数, 因此要用指针传递的方法作为参数进行传递。

#### 程序清单 7-10

```

/*----- 7-10.c -----*/
#include "stdio.h"

typedef struct BiTreeNode{
    char data;    /*结点的数据域*/
    struct BiTreeNode *lchild, *rchild;    /*指向左孩子和右孩子*/
} BiTreeNode, *BiTree;

/*创建一棵二叉树*/
CreatBiTree(BiTree *T, int *level1, int level2){
    char c;
    scanf("%c",&c);
    if(c == ' ') *T = NULL;
    else{
        *T = (BiTreeNode *)malloc(sizeof(BiTreeNode));    /*创建根结点*/
        (*T)->data = c;    /*向根结点中输入数据*/
        if(*level1 < level2) {*level1 = level2;}
        CreatBiTree(&((*T)->lchild), &(*level1), level2+1);    /*递归地创建左子树*/
    }
}

```

```

CreatBiTree(&((*T)->rchild),&(*level1),level2+1);
/*递归地创建右子树*/
}
/* 判断是否是完全二叉树 */
/* 是: 返回1 */
/* 不是: 返回0 */
int JusticCompleteBiTree(BiTree T,int level ,int n,int flag){
    if(!T){
        return 1;
    }
    if(level == n)
    {
        if(T->lchild == NULL && T->rchild != NULL) return 0;
        if(flag == 0){ /*同层的前面的结点无空指针*/
            if(T->rchild == NULL) flag = 1; /*出现空指针*/
        }
        else if(flag == 1){ /*同层的前面的结点有空指针*/
            if(T->lchild!=NULL || T->rchild!=NULL) return 0;
        }
    }
    if(level != n && level !=n+1)
    {
        if(T->lchild == NULL || T->rchild == NULL) return 0;
    }
    if(!JusticCompleteBiTree (T->lchild,level+1,n,flag)) return 0;
    if(!JusticCompleteBiTree (T->rchild,level+1,n,flag)) return 0;
    return 1;
}

main()
{
    BiTree T;
    int level1 = 0;
    printf("Please type some character to creat a binary tree\n");
    CreatBiTree(&T, &level1,0); /*创建一棵二叉树 T, level1 返回它的深度*/
    if(JusticCompleteBiTree(T,0,level1-1,0)) printf("It is a complete
binary tree\n"); /*输出判断结果*/
    else printf("It is NOT a complete binary tree\n");
    getche();
}

```

如果构造的二叉树如图 7-32 所示。

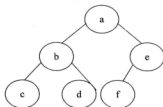


图 7-32 程序中构造的二叉树示意

本程序的运行结果如图 7-33 所示。

```
Please type some character to creat a binary tree
abc d ef
It is a complete binary tree
```

图 7-33 程序 7-10 的运行结果

## 7.11 动画模拟创建二叉树

### 【题目要求】

编写一个 C 程序，要求在 C 程序图形界面下动态模拟显示二叉树的先序创建过程。

### 【题目分析】

高级 C 语言也提供了 DOS 环境下的图形界面功能。在基于 Windows 的图形界面程序开发问世之前，早期的程序开发都是基于 DOS 环境下的字符界面或者是图形界面，因此开发周期较长，开发工作比较困难。本题借用高级 C 程序设计中的基于 DOS 下的图形界面开发，目的并不在于使读者学习 C 程序的图形界面开发，而是通过设计简单的图形界面程序，对二叉树的性质有更深入的理解，对应用递归方法解决实际问题的能力有更大的提高，同时增强读者的学习兴趣。

创建二叉树的方法在前面的章节中都有详细的介绍。它可以按照输入数据的先序序列进行结点的创建，最终生成一棵二叉树。这里不再作更多的赘述。下面重点介绍如何使用高级 C 语言的图形界面功能完成题目的要求。

为了动态地显示二叉树的先序创建过程，首先要在内存中生成一棵真正的二叉树，然后对该二叉树进行先序遍历，每遍历到一个结点就将该结点的内容输出到界面上，打印出一个结点的图案。在遍历二叉树的过程中，要根据二叉树结点所在该二叉树中位置（层数、左孩子、右孩子）的不同，设置图形界面中对应结点的图案的位置。例如，整棵二叉树的根结点应当显示在屏幕的中央上方等。另外还要在结点与结点之间划出一条连线，表明两结点之间的父子关系。这样动态地按二叉树先序遍历的顺序逐一地将每个结点显示出来，也就是该二叉树先序创建的过程。可以通过下面的代码实现这个功能。

```
printTree(BiTree T,int x,int y){
    char e[2] = {'\0','\0'};

    if(T){ /*递归结束条件，T为空*/

        { /*画出一个结点，程序暂停2秒*/
            e[0] = T->data ;
            circle(x,y,8) ;
            settextstyle(DEFAULT_FONT,HORIZ_DIR,2);
            outtextxy(x,y,e);
            sleep(2);
        }

        if(printTree(T->lchild,,y+20)) /*先序遍历T的左子树*/
            line(x,y,x+(200-y),y+20); /*画出结点之间的连线*/
        if(printTree(T->rchild,x+(200-y),y+20)) /*先序遍历T的右子树*/
            line(x,y,x+(200-y),y+20); /*画出结点之间的连线*/
    }
}
```

```

    return 1;
}
return 0;
}

```

函数 `printTree()` 是递归定义的。它通过先序遍历二叉树 `T`，将二叉树的先序创建过程显示在屏幕上。按照先序遍历的顺序，程序首先访问每一棵子树的根结点。在访问根结点时要做的“动作”就是将该结点的内容在屏幕上输出，并且画出结点的图案。函数 `circle()` 的功能是画一个以  $(x,y)$  为圆心，`e` 为半径的圆。函数 `settextstyle()` 的作用是设置输出字符的字体，大小等参数。函数 `outtextxy()` 的作用是在屏幕的  $(x,y)$  的坐标位置上输出结点中包含的字符。`sleep(2)` 函数的作用是将程序挂起 2 秒，以使画面动态显示。其中  $x$ 、 $y$  为递归函数 `printTree()` 的参数，它们的作用是控制图形界面中结点图案的位置。 $x$  控制图案在屏幕上的横坐标， $y$  控制纵坐标。在接下来的递归调用中，要将  $x$ 、 $y$  的值修改后传递，这样在下一层的结点的遍历中，结点图案的位置会发生改变。坐标  $(x,y)$  的变化规律如图 7-34 所示。

如图 7-34 所示，每次递归调用函数 `printTree` 时，结点的纵坐标  $y$  竖直方向增加 20 个像素；结点的横坐标  $x$ ，左孩子方向变为  $x-(200-y)$ ，右孩子方向变为  $x+(200-y)$ ，这样可以保证画出的结点图案不发生重合。当然这些值的增量并不是一定的，读者也可以根据自己的界面设计情况修改这些值的增量。

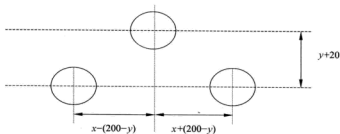


图 7-34 坐标  $(x,y)$  的变化规律

函数 `line()` 的作用是画出结点之间的连线。它包含 4 个参数，前两个参数规定线段的起点坐标，后两个参数规定线段的终点坐标。画线的条件是：根结点和子树同时存在。因此这里要用 `if(printTree(T->lchild,y+20))` 和 `if(printTree(T->rchild,x+(200-y),y+20))` 进行二叉树的遍历，只有当 `if` 语句的返回值为 1 时，即结点 `T` 存在左孩子（或右孩子），才能进行画线操作。因此可以看出本程序的绘图规律是：画出根结点→画出左子树→画出根结点与左子树之间的连线→画出右子树→画出根结点与右子树之间的连线。每棵子树也是按照这样的规律绘制。

程序清单 7-11

```

/*----- 7-11.c -----*/
#include "stdio.h"
#include <conio.h>
#include <graphics.h>

typedef struct BiTNode{
    char data;                /*结点的数据域*/

```

```

    struct BitNode *lchild , *rchild;    /*指向左孩子和右孩子*/
} BitNode , *BiTree;

CreatBiTree(BiTree *T){
    char c;
    gotoxy(5,5) ;
    printf("Please input chareter to creat a BiTree\n");
    scanf("%c",&c);
    if(c == ' ' ) *T = NULL;
    else{
        *T = (BitNode * )malloc(sizeof(BitNode));    /*创建根结点*/
        (*T)->data = c;    /*向根结点中输入数据*/
        CreatBiTree(&((*T)->lchild));    /*递归地创建左子树*/
        CreatBiTree(&((*T)->rchild));    /*递归地创建右子树*/
    }
}

printTree(BiTree T,int x,int y){
    char e[2] = {'\0','\0'};

    if(T){    /*递归结束条件, T为空*/
        {    /*画出一个结点, 程序暂停2秒*/
            e[0] = T->data ;
            circle(x,y,8) ;
            setttextstyle(DEFAULT_FONT,HORIZ_DIR,2);
            outtextxy(x,y,e);
            sleep(2);
        }
        if(printTree(T->lchild,x-(200-y),y+20))    /*先序遍历T的左子树*/
            line(x,y,x-(200-y),y+20);
        if(printTree(T->rchild,x+(200-y),y+20))    /*先序遍历T的右子树*/
            line(x,y,x+(200-y),y+20) ;
        return 1;
    }
    return 0;
}

main()
{
    BiTree T = NULL;
    int y = 100 ,x = 350;    /*设置二叉树根结点坐标*/
    int GraphDriver = DETECT;
    int GraphMode;
    int color;
    color = RED;
    initgraph( &GraphDriver, &GraphMode, "\\tc\\bgi" );    /* 初始化图形系统,
    驱动文件"\\tc\\bgi"的路径不能设置错误*/
    setcolor(color);    /*设置图形颜色: 红色*/
    CreatBiTree(&T);    /*创建二叉树*/
    printTree(T,x,y);    /*动态显示二叉树创建过程*/
    getch();
    closegraph();
}

```



**【程序说明】**

本程序中包含一些 C 语言中的图形函数，这里只做出必要的注释，不进行深入的介绍，因为它们的实用性不强。有兴趣的读者可以参看《C 语言高级程序设计》等书籍。

本程序的运行结果如图 7-35 所示。



图 7-35 程序 7-11 的运行结果

## 7.12 打印符号三角形

**【题目要求】**

规定这样一种形状的三角形，如图 7-36 所示为一个符号三角形。



图 7-36 符号三角形

这种三角形的特点是只由“+”和“-”组成，2个同号下面都是“+”。两个异号下面都是“-”。因此这种符号三角形的第一行决定整个符号三角形的“+”和“-”的个数及排列状态。编写一个程序，任意输入符号三角形的第一行（ $n$  个符号），打印出符合规则要求的符号三角形。例如：输入（++-+-++），就能够打印出如图 7-36 所示的符号三角形。

**【题目分析】**

实现本题目的要求方法很多，这里介绍一种比较直观方便而灵活的方法，应用队列的数据结构实现。

因为是任意地输入第一行  $n$  个符号，因此可以在输入符号时，将这  $n$  个符号入队列，然后从队首依次将符号取出。在取出每一行的过程中，当取出第  $i$  个符号时，要判断第  $i+1$  个符号的类型。如果第  $i+1$  个符号与第  $i$  个符号同号，则将“+”从队尾入队列；如果第  $i+1$  个符号与第  $i$  个符号异号，则将“-”入队列。在打印下一行时，依然重复上述操作，总共打印  $n$  行。

可以通过下面的代码完成上述操作。

```

printTriangle(int n){
    LinkQueue q;
    char e,a,b ;
    int i,j;
    initQueue(&q);                                /*初始化队列*/
    for(i=0;i<n;i++){                             /*输入第一行的符号，并入队列*/
        scanf("%c",&e);
        EnQueue(&q,e);
    }
    for(i=0;i<n;i++){                             /*控制符号三角的行数*/
        for(j=0;j<i;j++){
            printf(" ");                          /*控制输出格式，打印成倒三角形形状*/
        }
        DeQueue(&q,&a);                             /*出队列，打印每行的第一个符号*/
        printf("%c ",a);
        for(j=0;j<n-i-1;j++){                     /*控制输出每一行，第 i 行输出 n-i 个符号*/
            {
                DeQueue(&q,&b);                     /*出队列*/
                printf("%c ",b);
                if(a == b) EnQueue(&q,'+');        /*向队尾插入新元素*/
                else EnQueue(&q,'-');
                a = b;
            }
        }
        printf("\n");                             /*控制输出格式，打印成倒三角形形状*/
    }
}

```

打印一个符号三角形的操作由函数 `printTriangle()` 实现。该函数包含一个参数 `n`，它指定符号三角形中第一行的元素个数。

首先初始化一个队列  $q$ ，通过循环操作向队列中输入第一行的符号。然后通过一个二重循环打印符号三角形（暂不考虑控制输出格式的语句）。

外层循环控制符号三角形输出的行数，因为它是一个等边三角形，因此三角形的行数（高度）与第一行的元素个数  $n$  相同。

接下来是打印符号三角形中其中一行的符号，这个过程比较复杂，因此以一个具体实例加以说明其过程。假设队列中存放的第一行符号元素如图 7-37 所示。



图 7-37 队列中存放的第一行符号元素

首先通过 DeQueue()取出队首元素“+”，放入变量 a 中，并打印之。

然后通过一个循环输出第 2 个至最后一个元素，其过程如下。

(1) 通过 `DeQueue()` 取出第一个元素, 这里是 “+”, 放入变量 `b` 中。

(2) 如果变量  $a$  与  $b$  中的元素值相等, 即同号, 则三角形中  $a, b$  下面的符号一定为 “+”, 因此将 “+” 从队尾插入, 否则将 “-” 插入队尾。这里将 “+” 插入队尾, 如图 7-38 所示。

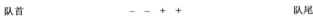


图 7-38 取出第 1, 2 个元素, 并将 “+” 插入队尾

(3) 再将  $b$  赋值给  $a$ , 重复 (1) 的操作。

该循环操作直到本行中的全部符号都取出打印后才停止。此时队列中存放的就是下一行中应当输出的符号了。对于本例，此时队列中的状态如图 7-39 所示。

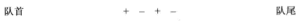


图 7-39 输出第 1 行后队列的状态

然后外层循环  $i++$ ，继续上述操作，打印下一行符号。直到把  $n$  行的符号三角形全部打印出来。

在本题的程序设计中要用到队列的基本操作，包括：

- ❑ 初始化队列: `initQueue()`;
- ❑ 入队列操作: `EnQueue()`;
- ❑ 出队列操作: `DeQueue()`。

这些操作的功能与具体实现在前面的章节中都有详细的介绍，这里不再赘述。

程序清单 7-12

```

7-12.c
#include "stdio.h"
typedef struct QNode { /*定义队列结点类*/
    char data;
    struct QNode *next;
} QNode , *QueuePtr;

typedef struct { /*定义一个链队列*/
    QueuePtr front; /*队头指针*/
    QueuePtr rear; /*队尾指针*/
} LinkQueue;

initQueue(LinkQueue *q) {
    /*初始化一个空队列*/
    q->front = q->rear = (QueuePtr)malloc(sizeof(QNode));
    /*创建一个头结点，队头队尾指针指向该结点*/
    if( !q->front) exit(0); /*创建头结点失败*/
    q->front->next = NULL; /*头结点指针域置 NULL*/
}

EnQueue(LinkQueue *q, char e) { /*入队列操作*/
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode)); /*创建一个队列元素结点*/
    if( !q->front) exit(0); /*创建头结点失败*/
    p->data = e; /*将元素 e 入队列*/
    p->next = NULL; /*修改队尾指针*/
    q->rear ->next = p;
}

```

```

    q->rear = p;
}
DeQueue(LinkQueue *q, char *e){
    /*如果队列q不为空, 删除q的队头元素, 用e返回其值*/
    QueuePtr p;
    if(q->front == q->rear) return;          /*队列为空, 返回*/
    p = q->front->next;
    *e = p->data;
    q->front->next = p->next;
    if(q->rear == p) q->rear = q->front; /*如果队头就是队尾, 则修改队尾指针*/
    free(p);
}

printTriangle(int n){
    LinkQueue q;
    char e,a,b;
    int i,j;
    initQueue(&q);                          /*初始化队列*/
    printf("Input the the charecter(+/-) of the row.\n");
    for(i=0;i<n;i++){                        /*输入第一行的符号, 并入队列*/
        scanf("%c",&e);
        EnQueue(&q,e);
    }
    printf("The charecter triangle is like\n");
    for(i=0;i<n;i++){                        /*控制符号三角形的行数*/
        for(j=0;j<i;j++){
            printf(" ");                    /*控制输出格式, 打印成倒三角形*/
            DeQueue(&q,&a);                  /*出队列, 打印每行的第一个符号*/
            printf("%c ",a);
            for(j=0;j<n-i-1;j++){            /*控制输出每一行, 第i行输出n-i个符号*/
                DeQueue(&q,&b);              /*出队列*/
                printf("%c ",b);
                if(a == b) EnQueue(&q,'+'); /*向队尾插入新元素*/
                else EnQueue(&q,'-');
                a = b;
            }
            printf("\n");                    /*控制输出格式, 打印成倒三角形*/
        }
    }
}

main()
{
    int n;
    printf("Please input the number of '+' or '-' of the row.\n");
    scanf("%d",&n);                          /*输入符号三角形中第一行的元素个数*/
    getchar();
    printTriangle(n);                          /*打印符号三角形*/
    getche();
}

```

### 【程序说明】

在程序中, 首先要输入符号三角形中第一行的元素个数, 然后程序提示输入符号三角形的第一行的元素。在输入符号的过程中不要按“空格”键将符号分开, 直接输入每一个符号即可, 最后以 Enter 结束。

本程序的运行结果如图 7-40 所示。

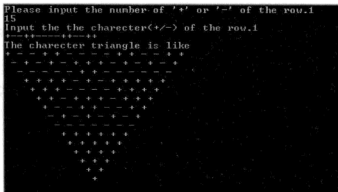


图 7-40 程序 7-12 的运行结果

### 7.13 递归函数的非递归求解

**【题目要求】**

已知  $n$  是大于等于 0 的整数, 请编写非递归的程序计算下列递归函数  $f(n)$ 。

$$f(n) = \begin{cases} n+1 & n=0 \\ nf(\lfloor n/2 \rfloor) & n \neq 0 \end{cases}$$

【题目分析】

递归方法固然是一种非常有效,易于实现的好方法,但是并非所有的程序设计语言都支持递归的功能。另外,递归方法本身也存在着系统开销过大,空间复杂度较高等缺点。因此有些时候需要把递归过程转换为非递归过程加以求解。

如果用递归方法求解此问题，其算法可描述为：

```
int f(int n)
{
    if(n==0)
        return n+1;
    else
        return n*f(n/2);
}
```

这个算法描述非常简单清楚，几乎与原递归函数一一对应。但是如果此题要用非递归方法求解就要复杂许多。

最常用的求解递归问题的非递归方法是应用栈的数据结构。我们知道，在计算机内部调用函数时，实际就应用到栈的数据结构（操作系统提供的功能，对于程序员来说是透明的）。在这个过程中，堆栈的主要作用是函数动态分配内存空间。当函数调用完毕后，堆栈空间随即被释放掉。我们也可以模仿操作系统使用堆栈进行函数调用这一特性来使用堆栈的数据结构解决递归问题。

递归函数的特点是“自己调用自己”，其实质就是嵌套地函数调用。我们可以利用堆栈结构来保存每一次递归调用的“现场”，这样每当一层递归调用结束后，都可以回到上一层递归调用的调用现场，得到刚才保留下的值。为了更加具体地说明如何使用堆栈保存

现场，以计算 $f(5)$ 为例，跟踪其过程。

首先建立一个空栈，该空栈最好是动态的，这样可以节省空间并具有很强的灵活性，如图 7-41 所示。

因为 5 不等于 0，因此 $f(5)=5f(2)$ ，这样就调用了函数 $f(2)$ 。于是在堆栈中保存本次函数调用的现场，即 $n=5$ 。这是因为目前还无法计算出 $f(2)$ 的值是多少，即无法明确 $f(5)$ 的值，需要继续调用函数 $f(2)$ 来进一步求解。但是此时的系数 5 将在后面的计算中用到，所以这里必须保留系数 5 的值。因此将 $n=5$ 压入栈中，如图 7-42 所示。



图 7-41 初始化一个空栈

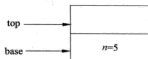


图 7-42 保留现场 (1)

再进一步计算 $f(2)$ 的值。因为 2 不等于 0，因此 $f(2)=2f(1)$ ，这样就调用了函数 $f(1)$ 。于是在堆栈中保存本次函数调用的现场，即 $n=2$ 。因此将 $n=2$ 压入栈中，如图 7-43 所示。

再进一步计算 $f(1)$ 的值。因为 1 不等于 0，因此 $f(1)=1f(0)$ ，这样就调用了函数 $f(0)$ 。于是在堆栈中保存本次函数调用的现场，即 $n=1$ 。因此将 $n=1$ 压入栈中，如图 7-44 所示。

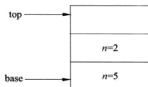


图 7-43 保留现场 (2)

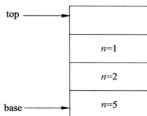


图 7-44 保留现场 (3)

再进一步计算 $f(0)$ 的值，因为 $f(0)=0+1=1$ ，因此函数的递归调用结束，于是 $f(0)$ 返回 1。接下来的操作堆栈就要派上用场了。因为在这之前堆栈中已存放了每次调用函数 $f()$ 的乘积系数 $n$ ，因此需要移动 top 指针将栈顶元素一一取出。

首先计算 $f(1)$ ，因为 $f(1)=1f(0)$ ，而 $f(0)$ 的乘积系数即保留在栈顶，于是取出栈顶元素 $n=1$ ，并与返回 $f(0)$ 相乘，得到 $f(1)=1$ 。

再计算 $f(2)$ ，因为 $f(2)=2f(1)$ ，而 $f(1)$ 的乘积系数此时也存放在栈顶，于是取出栈顶元素 $n=2$ ，并与返回 $f(1)$ 相乘，得到 $f(2)=2$ 。

再计算 $f(5)$ ，因为 $f(5)=5f(2)$ ，而 $f(2)$ 的乘积系数此时也存放在栈顶，于是取出栈顶元素 $n=5$ ，并与返回 $f(2)$ 相乘，得到 $f(5)=10$ 。

此时堆栈中为空，表明递归过程结束。

可以用下面的代码描述本题的非递归算法：

```
int f(int n)
{
    int r = 1, e;
    sqStack stack;
    initStack(&stack);
    /*初始化栈*/
}
```

```

while(n!=0){
    Push(&stack,n);          /*保存现场 n*/
    n = n/2;
}
while(stack.top!=stack.base)
{
    Pop(&stack,&e);
    r = r * e;                /*计算 f(0), f(1), f(2), ..., f(n)*/
}
return r;                    /*返回 f(n)*/
}

```

上述代码描述的执行过程如下。

(1) 首先初始化一个栈结构，然后通过一个循环保留每次调用函数  $f$  的现场值  $n$ 。所谓的调用函数  $f()$  在这里并非真的调用，如果真去调用函数  $f()$ ，就是递归解法了。这里只将系数  $n$  顺序入栈，并做  $n=n/2$  的操作，直到  $n$  等于 0 为止。这样就模拟出每次调用函数  $f$  时保存现场参数的操作了。

(2) 然后再通过一个循环顺序取出刚才保存的系数  $n$ ，并与  $r$  迭代相乘，直到将堆栈中的元素取完为止。其中  $r$  的初值为 1，它表示  $f(0)$  的值为 1。每进行一次相乘，都对应着结束一层的递归调用并返回，最终得到的值就是欲求的  $f(n)$ 。

所谓递归函数的非递归求解，就是利用栈的数据结构集中保存每一次递归调用时现场参数（本题目中为  $f(n)$  的系数），以此来模拟真实的函数调用。堆栈的入栈操作对应的是函数的调用操作（或者递归调用），堆栈的出栈操作对应的是函数调用的结束，并返回到上一层函数（或者递归调用结束，返回上一层递归调用）。

从递归算法到非递归算法的转换十分有用，特别是在一些特定的场合（例如嵌入式系统的开发、系统资源紧缺、时间复杂度要求较高等），递归算法是不适宜应用的。因此掌握从递归算法到非递归算法的转换方法具有实际意义，希望读者认真思考本节内容，并提出更好的解决方案。

下面给出完整的程序清单。

程序清单 7-13

```

/*----- 7-13.c -----*/
#include "stdio.h"
#include "math.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10

typedef int ElemType;
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;
/*初始化栈*/
void initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间，首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0);          /*分配空间失败*/
    s->top = s->base;                /*最开始，栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}

```

```

/*入栈操作, 将e压入栈中*/
void Push(sqStack *s, ElemType e){
    if(s->top - s->base >= s->stacksize){
        /*栈满, 追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
        STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0); /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
    s->top++;
}
/*出栈操作, 用e将栈顶元素返回*/
void Pop(sqStack *s, ElemType *e){
    if(s->top == s->base) return;
    *e = *(s->top--);
}

int f(int n)
{
    int r = 1, e;
    sqStack stack;
    initStack(&stack); /*初始化栈*/
    while(n!=0){
        Push(&stack,n); /*保存现场n*/
        n = n/2;
    }
    while(stack.top!=stack.base)
    {
        Pop(&stack,&e);
        r = r * e;
    }
    return r;
}

main()
{
    printf("The result for conversion of recursion to non recursion is\n");
    printf("f(5)=%d \n",f(5));
    getch();
}

```

本程序的运行结果如图 7-45 所示。

```

The result for conversion of recursion to non recursion is
f(5)=10

```

图 7-45 程序 7-13 的运行结果

**注意：**有关栈结构的基本操作（例如：初始化堆栈 `initStack`，入栈操作 `Push`，出栈操作 `Pop` 等）在第 1 章中已有详细的讲解，在这里直接引用。

## 7.14 任意长度整数加法

### 【题目要求】

编写一个程序实现任意长度的整数加法。



## 【题目分析】

在C语言中,无论采用哪种数据类型(int、float、long等)存储数据,它的长度都是有限的,超过这个长度的范围就会发生溢出。因此想通过定义数据类型的方式实现任意长度的整数加法是不可能的。

可以利用字符串表示任意长度的整数。将任意长度的整数以字符串的形式存储在一个动态的线性表中,这样无论多长的整数都可以轻松地表示了。在进行两个整数相加时,只需将每个字符串中对应的字符所表示的数字进行相加,并作相应的进位处理即可。最后将计算的结果以字符串的形式存储在一个新的线性表中。

我们需要考虑的是应用什么数据结构来作为字符串的存储容器。一种比较高效而直观的方法是采用动态栈结构来存放整数字符串。因为两个整数相加时都是从最低位加起的,因此如果采用顺序表结构存储整数字符串,必须从字符串的尾部开始取出元素进行操作,这样不够方便。如果使用堆栈结构,先输入的高位数据一定处于栈底,后输入的低位数据处于栈顶,因此只需从栈顶取元素。另外,还要将每次相加的结果以字符的形式存放到另外一个结果栈中。这是因为结果的产生也是从低位向高位进行的,而高位的数据必然先输出,因此这里采用堆栈存储运算的结果字符串也是十分合理的。

堆栈的定义如下代码:

```
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10
typedef char ElemType;
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;
```

定义 sqStack 为堆栈类,结构中包含堆栈的栈顶指针、栈底指针和堆栈的最大容量。假设用户已经创建了3个堆栈,分别为 s1, s2, s3。其中堆栈 s1 和 s2 中存放了两个加数的字符串,堆栈 s3 为一个空栈,用来存放加数的和,如图 7-46 所示。

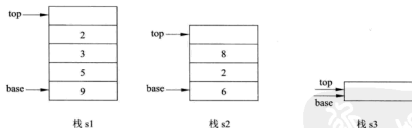


图 7-46 堆栈 s1, s2, s3 的初始状态

如图 7-46 所示,栈 s1 中存放的整数为 9532,栈 s2 中存放的数据为 628,现要将它们的和以字符串的形式反向存入到堆栈 s3 中。可以通过下面的算法实现这个功能。

```
void ADD(sqStack *s1,sqStack *s2,sqStack *s3)
{
    char a1,a2,a3,c=0;          /*a1, a2 分别存放从堆栈 s1, s2 中取出的(数据)元素,
                                a3=a1+a2,c 中存放进位*/
```

```

while(StackLen(*s1)!=0 && StackLen(*s2)!=0)
{
    Pop(s1,&a1);           /*取出 s1 栈的栈顶元素给 a1*/
    Pop(s2,&a2);           /*取出 s2 栈的栈顶元素给 a2*/
    a3 = (a1-48) + (a2-48) + c + 48; /*相加*/
    if(a3>'9')
    {
        a3 = a3 - '9' + 47; /*产生进位的情况*/
        c = 1;
    }
    else
        c = 0; /*不产生进位*/
    Push(s3,a3); /*将结果入栈 s3*/
}
if(StackLen(*s1)!=0) /*栈 s1 不为空的情况*/
{
    while(StackLen(*s1)!=0)
    {
        Pop(s1,&a1); /*取出 s1 栈的栈顶元素给 a1*/
        a3 = a1 + c; /*与进位标志 c 相加*/
        if(a3>'9')
        {
            a3 = a3 - '9' + 47; /*产生进位的情况*/
            c = 1;
        }
        else
            c = 0; /*不产生进位*/
        Push(s3,a3); /*将结果入栈 s3*/
    }
}
else if(StackLen(*s2)!=0) /*栈 s1 不为空的情况*/
{
    while(StackLen(*s2)!=0)
    {
        Pop(s2,&a2); /*取出 s1 栈的栈顶元素给 a1*/
        a3 = a2 + c; /*与进位标志 c 相加*/
        if(a3>'9')
        {
            a3 = a3 - '9' + 47; /*产生进位的情况*/
            c = 1;
        }
        else
            c = 0; /*不产生进位*/
        Push(s3,a3); /*栈 s1 不为空的情况*/
    }
}
if(c==1)
    Push(s3,'1'); /*如果最后有进位，将字符'1'入栈 s3*/
}

```

上述算法的功能是将堆栈 s1 和 s2 中的内容（整数）相加，并反向存入堆栈 s3 中。

首先考虑每一位的相加。在算术中，严格地讲每一位数字的相加都要包含 3 个部分的内容，即两个加数和一个进位。因此在本算法中，每次都从堆栈 s1 和 s2 的栈顶取出元素，分别存放到 a1 和 a2 中，再通过语句  $a3 = (a1-48) + (a2-48) + c + 48$ ；将其相加，并以字符的形式存放到堆栈 s3 中。其中 a1-48 为字符 a1 对应的十进制整数，a2-48 为字符 a2 对应的十进制整数，c 为进位标志（1 或 0）。将它们相加后再加上 48 即得到对应的字符码。

算法中通过一个循环不断从栈 s1 和 s2 的栈顶取出元素（数字的字符码），相加后再以字符的形式存放到栈 s3 中。这个过程只处理两个加数中较短的加数的长度部分。例如

12345+123 的计算在上述过程中只处理低 3 位的运算。以图 7-46 所示的整数相加为例, 上述过程中的每一步操作如图 7-47 所示。

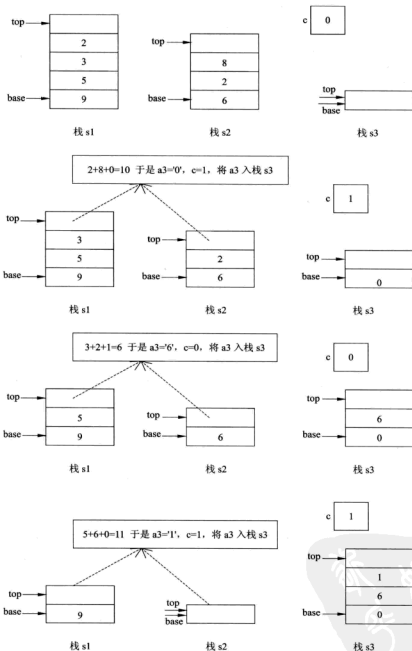


图 7-47 计算步骤(一)

如图 7-47 所示, 经过一系列的出栈入栈等操作, 将 9532 与 628 的低 3 位相加, 得到

160 并存放到堆栈 s3 中。这里要注意两点：一是在堆栈 s3 中，栈底存放计算结果的低位，栈顶存放计算结果的高位（高位在上，低位在下），这与栈 s1 和 s2 是正好相反的。二是此时进位标志 c 的值为 1，后续的计算还要用到它。

接下来就要将不为空的堆栈里的内容取出，并以字符串的形式存放到堆栈 s3 中。这里的操作仍然通过一个循环来完成。每次循环都取出剩余堆栈中的栈顶元素，存放到 a1 或 a2 中（如果是从栈 s1 中取数，则存放到 a1 中，否则存放到 a2 中），再通过语句  $a3 = a2 + c$ ；或者  $a3 = a2 + c$ ；对取出的元素进行处理，这是因为这个过程中仍然有进位的可能（例如图例中所示）。最后将计算的结果以字符的形式存放到堆栈 s3 中。上述过程的每一步操作如图 7-48 所示。

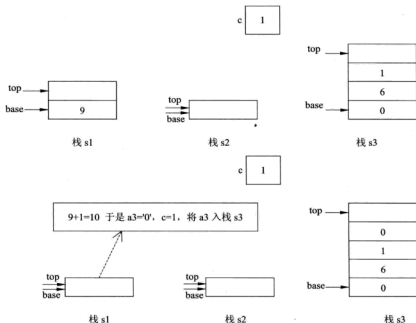


图 7-48 计算步骤（二）

如图 7-48 所示，此时堆栈 s1 和 s2 都已为空，于是退出循环。最后还要做一步操作，即判断 c 是否为 1。如果进位标志 c 等于 1，表明两个整数相加后最高位产生了进位，因此还要将字符 '1' 存放到栈 s3 的栈顶。如果进位标志 c 等于 0，表明两个整数相加后最高位没有产生进位，此时堆栈 s3 中存放的字符串即为计算结果。对于本例，还要将字符 1 存放到栈 s3 的栈顶，如图 7-49 所示。

最终堆栈 s3 中存放的就是相加后的结果字符串，高位处于栈顶，低位处于栈底，因此只要顺序出栈并打印出栈中的每一个元素（字符），就可以输出最终的计算结果。本例中输出 s3 后得到的计算结果为 10160，计算结果正确。

下面给出完整的程序清单供读者参考。

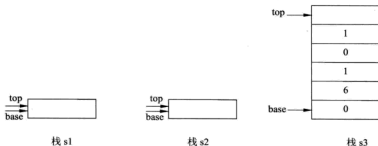


图 7-49 计算步骤 (三)

程序清单 7-14

```

/*----- 7-14.c -----*/
#include "stdio.h"
#include "math.h"
#define STACK_INIT_SIZE 20
#define STACKINCREMENT 10
/*定义堆栈*/
typedef char ElemType;
typedef struct{
    ElemType *base;
    ElemType *top;
    int stacksize;
}sqStack;
/*初始化栈*/
void initStack(sqStack *s)
{
    /*内存中开辟一段连续空间作为栈空间，首地址赋值给 s->base*/
    s->base = (ElemType *)malloc(STACK_INIT_SIZE * sizeof(ElemType));
    if(!s->base) exit(0); /*分配空间失败*/
    s->top = s->base; /*最开始，栈顶就是栈底*/
    s->stacksize = STACK_INIT_SIZE; /*最大容量为 STACK_INIT_SIZE */
}
/*入栈操作，将e压入栈中*/
void Push(sqStack *s, ElemType e){
    if(s->top - s->base >= s->stacksize){
        /*栈满，追加空间*/
        s->base = (ElemType *)realloc(s->base, (s->stacksize +
            STACKINCREMENT)*sizeof(ElemType));
        if(!s->base) exit(0); /*存储分配失败*/
        s->top = s->base + s->stacksize;
        s->stacksize = s->stacksize + STACKINCREMENT; /*设置栈的最大容量*/
    }
    *(s->top) = e; /*放入数据*/
    s->top++;
}
/*出栈操作，用e将栈顶元素返回*/
void Pop(sqStack *s, ElemType *e){
    if(s->top == s->base) return;
    *e = --(s->top);
}

```

```

/*计算堆栈 s 当前的长度*/
int StackLen(sqStack s){
    return (s.top - s.base) ;
}

void ADD(sqStack *s1,sqStack *s2,sqStack *s3)
{
    char a1,a2,a3,c=0;          /*a1, a2 分别存放从堆栈 s1, s2 中取出的 (数据) 元素,
                                a3=a1+a2, c 中存放进位*/
    while(StackLen(*s1)!=0 && StackLen(*s2)!=0)
    {
        Pop(s1,&a1);              /*取出 s1 栈的栈顶元素给 a1*/
        Pop(s2,&a2);              /*取出 s2 栈的栈顶元素给 a2*/
        a3 = (a1-48) + (a2-48) + c + 48; /*相加*/
        if(a3>'9')
        {
            a3 = a3 - '9' + 47;    /*产生进位的情况*/
            c = 1;
        }
        else
            c = 0;                /*不产生进位*/
        Push(s3,a3);              /*将结果入栈 s3*/
    }
    if(StackLen(*s1)!=0)          /*栈 s1 不为空的情况*/
    {
        while(StackLen(*s1)!=0)
        {
            Pop(s1,&a1);          /*取出 s1 栈的栈顶元素给 a1*/
            a3 = a1 + c ;         /*与进位标志 c 相加*/
            if(a3>'9')
            {
                a3 = a3 - '9' + 47; /*产生进位的情况*/
                c = 1;
            }
            else
                c = 0;            /*不产生进位*/
            Push(s3,a3);          /*将结果入栈 s3*/
        }
    }
    else if(StackLen(*s2)!=0)     /*栈 s1 不为空的情况*/
    {
        while(StackLen(*s2)!=0)
        {
            Pop(s2,&a2);          /*取出 s1 栈的栈顶元素给 a1*/
            a3 = a2 + c;         /*与进位标志 c 相加*/
            if(a3>'9')
            {
                a3 = a3 - '9' + 47; /*产生进位的情况*/
                c = 1;
            }
            else
                c = 0;            /*不产生进位*/
            Push(s3,a3);          /*栈 s1 不为空的情况*/
        }
    }
}

```

```

    if (c==1)
        Push(s3, '1');          /*如果最后有进位，将字符'1'入栈 s3*/
    }

main()
{
    char e;
    sqStack s1, s2, s3;
    initStack(&s1);              /*初始化堆栈 s1，存放加数*/
    initStack(&s2);              /*初始化堆栈 s2，存放加数*/
    initStack(&s3);              /*初始化堆栈 s3，存放结果*/
    printf("Please input the first integer\n");
                                /*输入第一个任意长整数，按#结尾*/
    scanf("%c", &e);
    while (e!='#')
    {
        Push(&s1, e);            /*将加数（字符串）入栈 s1*/
        scanf("%c", &e);
    }
    getchar();                  /*接收回车符*/
    printf("Please input the second integer\n");
                                /*输入第二个任意长整数，按#结尾*/
    scanf("%c", &e);
    while (e!='#')
    {
        Push(&s2, e);            /*将加数（字符串）入栈 s2*/
        scanf("%c", &e);
    }
    ADD(&s1, &s2, &s3);           /*加法运算，将结果存放在 s3 中*/
    printf("The result is\n");
    while (StackLen(s3) != 0)    /*输出结果，打印在屏幕上*/
    {
        Pop(&s3, &e);
        printf("%c", e);
    }
    getche();
}

```

本程序的运行结果如图 7-50 所示。

```

Please input the first integer
12345678987654321#
Please input the second integer
98765432123456789#
The result is
11111111111111110

```

图 7-50 程序 7-14 的运行结果

## 第8章 数值计算问题

数值计算问题是计算机科学的一个重要分支。电子计算机的设计初衷也是为了计算一些规模庞大的,无法通过人工手算解决的数值问题。如何使用计算机来计算一些数学中碰到的计算问题(例如:解方程、计算积分值等)是数值计算要解决的目标。而通过已有的数学知识来设计数值算法是解决数值计算问题的途径。本章将围绕着“数值积分”、“方程求根”、“常微分方程求解”和“解线性方程组”这4类常见的数值计算问题展开讨论,向大家介绍一些常用的数值算法。

### 8.1 递推化梯形法求解定积分

#### 【题目要求】

计算定积分:  $I = \int_0^1 \frac{\sin x}{x} dx$  的值。

#### 【题目分析】

计算定积分是数值计算领域的一个重要内容。在微积分中计算定积分常用的方法是使用牛顿-莱布尼兹公式。但有时被积函数很难得到其原函数(例如:  $f(x)=\sin x/x$ ),因此人们考虑使用计算机来帮助计算这类定积分,这就是积分的数值计算问题。

一种比较常用的计算定积分的数值方法是递推化的梯形法。它的思想源于简单的梯形公式。

定积分的几何意义就是由积分上限  $x=a$ , 积分下限  $x=b$ , 被积函数  $f(x)$  和  $x$  轴围成区域的面积。例如定积分  $I = \int_a^b f(x) dx$  的几何意义如图 8-1 所示。

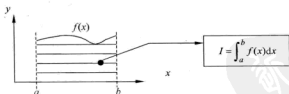


图 8-1 定积分的几何意义

如图 8-1 所示,阴影部分的面积即为定积分的值。最初的设想是应用梯形的面积来近似代替积分值。即  $\int_a^b f(x) dx \approx T_1 = \frac{b-a}{2} [f(a) + f(b)]$ , 它的几何意义如图 8-2 所示。



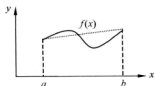


图 8-2 梯形公式的几何意义

如图 8-2 所示, 用图中所示的梯形面积来代替定积分值。这样的近似计算未免太过粗糙, 显然, 很难保证应用这种梯形公式计算出来的积分值的精准度。但是应用梯形公式可以保证 1 次代数精度。也就是说, 如果被积函数为一个一次函数 (例如:  $f(x)=2x+1$  等), 是可以应用梯形公式计算出准确值的, 这一点十分显然。但是如果计算  $I = \int_0^1 \frac{\sin x}{x} dx$ , 应用梯形公式显然不适合。

考虑将求积区域二分, 构成两个梯形, 如图 8-3 所示。

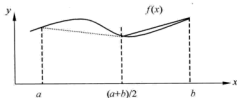


图 8-3 二分求积区域

这样再在区域  $[a, (a+b)/2]$  上和区域  $[(a+b)/2, b]$  上使用梯形公式法计算梯形面积, 得到的面积为  $T_2 = \frac{h}{4}[f(a) + 2f(\frac{a+b}{2}) + f(b)]$ , 其中  $h=b-a$ 。显然应用  $T_2$  代替近似积分值  $I$  比  $T_1$  更加精确, 而  $T_2$  与  $T_1$  之间存在着以下关系:

$$T_2 = \frac{1}{2}T_1 + \frac{b-a}{2}f\left(\frac{a+b}{2}\right)$$

将这种思想加以推广, 将  $[a, b]$  分为  $n$  等分, 一共有  $n+1$  个等分点  $x_k=a+kh$ 。其中  $h=(b-a)/n$ ,  $k=0, 1, 2, \dots, n$ 。用  $T_n$  表示将  $[a, b]$   $n$  等分后的梯形面积总和, 用  $T_{2n}$  表示将  $[a, b]$   $2n$  等分后的梯形面积总和, 则可得到递推公式:

$$T_{2n} = \frac{1}{2}T_n + \frac{h}{2} \sum_{k=0}^{n-1} f\left(\frac{1}{2}(x_k + x_{k+1})\right)$$

显然, 当  $n=1$  时有:

$$\begin{aligned} T_2 &= \frac{1}{2}T_1 + \frac{h}{2}f\left(\frac{x_0 + x_1}{2}\right) \\ \Rightarrow T_2 &= \frac{1}{2}T_1 + \frac{b-a}{2}f\left(\frac{a + (a+h)}{2}\right) \\ \Rightarrow T_2 &= \frac{1}{2}T_1 + \frac{b-a}{2}f\left(\frac{a+b}{2}\right) \end{aligned}$$

通过上述递推公式就可以将积分区域  $[a, b]$  不断地二分, 这样得到的积分值将会越来越精确。在求积分时, 可以设置一个精度  $\epsilon$ , 当  $|T_{2n} - T_n| < \epsilon$  时迭代结束。

递推化的梯形法求积分的算法描述如下:

```

输入: 积分上下限 a 和 b, 以及精度 e;
h ← b - a
 $T_1 \leftarrow \frac{h}{2} [f(a) + f(b)]$ 
repeat1:
    S ← 0
    x ← a + h / 2
    repeat2:
        S ← S + f(x)
        x ← x + h
        if (x < b)
            repeat2;
        else
             $T_2 \leftarrow \frac{T_1}{2} + \frac{h}{2} S;$ 
            if (|T2 - T1| < e)
                output T2 end
            else
                (h ← h / 2; T1 ← T2; repeat1;)

```

这里注意, 使用该算法求解定积分, 无论积分上下限  $a$ ,  $b$  的取值的正负都同样适用, 但要保证积分下限  $b$  小于积分上限  $a$ 。

下面给出递推梯形法计算  $I = \int_0^1 \frac{\sin x}{x} dx$  的源程序清单。

程序清单 8-1

```

/*----- 8-1.c -----*/
#include "stdio.h"
#include "math.h"

double func(double x)
{
    if(x!=0)
        return sin(x)/x;
    else
        return 1.0;
}

double ING(double a,double b,double e)
{
    double T1=0.0,T2=0.0,S=0.0,h,x;
    int flag;
    h = b - a;
    T1 = h/2*(func(a)+func(b));
    do{
        S = 0;
        x = a + h/2;
        while(x<b){
            S = S + func(x);
            x = x + h;
        }
        T2 = T1/2 + h/2*S;
        if(fabs(T1-T2)>=e){
            h = h/2;

```

```

        T1 = T2;
        flag = 1;
    }
    else flag = 0;
}
while(flag);
return T2;
}

main()
{
    double a,b,e;
    printf("Please input the low & high limitation and the accuracy\n");
    printf("Low limitation:");
    scanf("%lf",&a);
    printf("High limitation:");
    scanf("%lf",&b);
    printf("The accuracy:");
    scanf("%lf",&e);
    printf("The result of integration is %lf",ING(a,b,e));
    getche();
}

```

### 【程序说明】

本程序严格按照上面所示的算法编写。其中函数 func()的作用是计算  $\sin x/x$  的值，即返回被积函数  $f(x)$  的值。因此可根据被积函数的不同修改函数 func() 的表达。由于  $\sin 0/0$  的值不存在，但根据极限定理可知， $\lim_{x \rightarrow 0} \frac{\sin x}{x} = 1$ ，因此  $\frac{\sin 0}{0} = 1$ 。

本程序的运行结果如图 8-4 所示。

```

Please input the low & high limitation and the accuracy
Low limitation:0.
High limitation:1.
The accuracy:0.000001
The result of integration is 0.946083

```

图 8-4 程序 8-1 的运行结果

注意： $I = \int_0^1 \frac{\sin x}{x} dx = 0.9460831$

## 8.2 求解低阶定积分

### 【题目要求】

计算下列定积分的值：

$$I_1 = \int_0^1 (2x+3) dx$$

$$I_2 = \int_0^1 (x^3 + 2x - 1) dx$$

$$I_3 = \int_0^1 (x^4 + x^3 + 1) dx$$

## 【题目分析】

不难发现以上3个积分式的被积函数都是一些低阶的函数。求解这类低阶定积分时，当然可以应用8.1节中介绍的递推化梯形法，但是其实没有必要那么麻烦，有一些现成的计算公式可以直接对低阶的定积分求解。

前面已经介绍过，对于1阶的定积分（被积函数为1阶函数），可以用梯形公式直接求解。即：

$$I = \int_a^b f(x)dx = \frac{b-a}{2} [f(a) + f(b)]$$

计算3阶以下的定积分时，可以应用辛甫生（Simpson）公式求解，即：

$$I = \int_a^b f(x)dx = \frac{b-a}{6} \left[ f(a) + 4f\left(\frac{a+b}{2}\right) + f(b) \right]$$

计算4阶以下的定积分时，可以用柯特斯(Cotes)公式求解，即：

$$I = \int_a^b f(x)dx = \frac{b-a}{90} [7f(x_0) + 32f(x_1) + 12f(x_2) + 32f(x_3) + 7f(x_4)]$$

$$\text{其中 } x_k = a + kh (k=0,1,2,3,4), h = \frac{b-a}{2}$$

这里注意，使用该算法求解定积分，无论积分上下限  $a$ 、 $b$  的取值的正负都同样适用，但要保证积分下限  $b$  小于积分上限  $a$ 。

应用以上这些公式，可以非常方便地求解出低阶定积分的值，而且效率很高。如果在规定的阶数范围内使用上述公式求解定积分，可以得到定积分的精确解。如果超出了规定的阶数范围，只能求得近似解。上述公式的推导在此省略，有兴趣的读者可参看《数值分析》等方面的书籍。

下面分别给出计算定积分  $I_1$ 、 $I_2$ 、 $I_3$  的源程序清单。

程序清单 8-2 （计算定积分  $I_1 = \int_0^1 (2x+3)dx$ ）

```

/*----- 8-2.c -----*/
#include "stdio.h"
float func(float x)
{
    return 2*x+3;
}

float ING(float a,float b)
{
    return (b-a)/2*(func(a)+func(b));
}

main()
{
    float a,b;
    printf("Please input the low & high limitation and the accuracy\n");
    printf("Low limitation:");
    scanf("%f",&a);
    printf("High limitation:");
    scanf("%f",&b);
    printf("The result of integration is %f",ING(a,b));
    getche();
}

```

本程序的运行结果如图 8-5 所示。

```
Please input the low & high limitation and the accuracy
Low limitation:0
High limitation:1
The result of integration is 4.000000_
```

图 8-5 程序 8-2 的运行结果

程序清单 8-3 (计算定积分  $I_2 = \int_0^1 (x^3 + 2x - 1)dx$ )

```
/*----- 8-3.c -----*/
#include "stdio.h"
#include "math.h"
float func(float x)
{
    return pow(x,3)+2*x-1;
}

float ING(float a,float b)
{
    return ((b-a)/6)*(func(a)+4*func((a+b)/2)+func(b));
}

main()
{
    float a,b;
    printf("Please input the low & high limitation and the accuracy\n");
    printf("Low limitation:");
    scanf("%f",&a);
    printf("High limitation:");
    scanf("%f",&b);
    printf("The result of integration is %f",ING(a,b));
    getch();
}
```

本程序的运行结果如图 8-6 所示。

```
Please input the low & high limitation and the accuracy
Low limitation:0
High limitation:1
The result of integration is 0.250000_
```

图 8-6 程序 8-3 的运行结果

程序清单 8-4 (计算定积分  $I_3 = \int_0^1 (x^4 + x^3 + 1)dx$ )

```
/*----- 8-4.c -----*/
#include "stdio.h"
#include "math.h"
float func(float x)
{
    return pow(x,4)+pow(x,3)+1;
}

float x(float a,float b,int k)
```

```

{
    return a+k*(b-a)/4 ;
}

float ING(float a,float b)
{
    return ((b-a)/90)*(7*func(x(a,b,0))+32*func(x(a,b,1))+
    12*func(x(a,b,2))+32*func(x(a,b,3))+7*func(x(a,b,4)));
}

main()
{
    float a,b;
    printf("Please input the low & high limitation and the accuracy\n");
    printf("Low limitation:");
    scanf("%f",&a);
    printf("High limitation:");
    scanf("%f",&b);
    printf("The result of integration is %f",ING(a,b));
    getch();
}

```

本程序的运行结果如图 8-7 所示。

```

Please input the low & high limitation and the accuracy
Low limitation:0
High limitation:1
The result of integration is 1.450000

```

图 8-7 程序 8-4 的运行结果

用计算机计算定积分时，可以应用以上两节中介绍的方法进行求解。如果要计算一些高次而复杂的定积分，可以使用递推化梯形法求解；如果计算的定积分阶数较低，可以直接使用本节中介绍的公式进行计算。在第 3 章中曾介绍过一种数值概率算法计算定积分，那也不是不错的方法，读者可以灵活掌握。另外，数值积分的算法并不止这些，这里介绍的只是最为常用的、比较简单的几种。如果读者想要更多地了解数值积分算法，可以参看《数值分析》等书籍。

### 8.3 迭代法开平方运算

#### 【题目要求】

用迭代法求  $x = \sqrt{a}$ 。已知求平方根的迭代运算公式为：

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

要求前后两次求出的  $x$  的差的绝对值小于  $10^{-5}$ 。

#### 【题目分析】

这是一个数值计算中迭代法求解平方根的问题。它的理论基础是应用迭代法求方程的根，下面作一简要介绍。

应用计算机求解方程：

$$x = \phi(x)$$

的根，一般采用迭代法逐次逼近方程的根。首先要构造一个迭代公式：

$$x_{n+1} = \phi(x_n)$$

然后选定一个所谓的迭代初值  $x_0$ ，将该迭代初值代入上面的迭代公式中得到  $x_1$ ：

$$x_1 = \phi(x_0)$$

再将  $x_1$  代入上面的迭代公式中得到  $x_2$ ：

$$x_2 = \phi(x_1)$$

依次类推。直到将  $x_k$  代入迭代公式中得到  $x_{k+1}$ ，此时  $|x_{k+1} - x_k|$  的值等于 0，或者  $|x_{k+1} - x_k|$  的值处在一个规定的区间范围内，迭代结束。如果  $|x_{k+1} - x_k|$  的值等于 0，那么得到的  $x_{k+1}$  就是该方程的精确根。如果  $|x_{k+1} - x_k|$  的值处在一个规定的区间范围内，那么得到的  $x_{k+1}$  就是该方程的近似根。

这种迭代法求解方程根的理论基础是迭代的收敛性。可以用图 8-8 形象地显示这种迭代的过程。

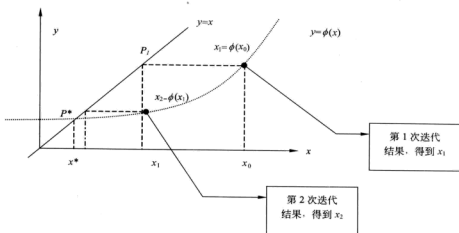


图 8-8 迭代过程

如图 8-8 所示，方程  $x = \phi(x)$  的根实际上就是坐标系中曲线  $y = \phi(x)$  与曲线  $y = x$  的交点  $P^*$  的横坐标  $x^*$ 。首先由迭代初值  $x_0$  在曲线  $y = \phi(x)$  得到  $x_1$ ，通过坐标  $(x_0, x_1)$  作与  $x$  轴平行的直线交曲线  $y = x$  于点  $P_1$ ，那么点  $P_1$  的横坐标就是  $x_1$ ，纵坐标的值也为  $x_1$ 。过点  $P_1$  作平行于  $y$  轴的直线交  $x$  轴于点  $x_1$ ，那么这个  $x_1$  就是第一次迭代后的结果。可以看到  $x_1$  较迭代初值  $x_0$  更加靠近  $x^*$ ，也就是说  $x_1$  比  $x_0$  更加接近方程的根。按照相同的做法继续做下去，如果横坐标分别依迭代公式  $x_{k+1} = \phi(x_k)$  收敛，则得到的  $x_1, x_2, x_3, \dots$  将越来越靠近  $x^*$ ，最终得到方程的根或近似根。

因此这里要求应用迭代法求解的方程  $x = \phi(x)$  必须保证其迭代公式  $x_{k+1} = \phi(x_k)$  是迭代收敛的，即  $x^* = \lim_{k \rightarrow \infty} x_k$ 。

本题中给定的计算平方根的迭代公式也是根据“迭代法求解方程根”的理论得来的。因为求  $a$  的平方根  $x = \sqrt{a}$ ，实际上就是要求解方程：

$$x = \frac{1}{2} \left( x + \frac{a}{x} \right)$$

的根。而构造出的迭代公式:

$$x_{n+1} = \frac{1}{2} \left( x_n + \frac{a}{x_n} \right)$$

本身就是迭代收敛的, 因此可以用迭代法求解正数  $a$  的平方根。

了解了迭代公式的含义, 就不难设计出求解正数  $a$  的平方根的迭代算法。其算法描述如下:

任意给定一个迭代初值  $x_n$ , 不妨令  $x_n = a$ :

Repeat:

将  $x_n$  代入迭代公式中, 求出  $x_{n+1}$ ;

If  $(|x_{n+1} - x_n| < \text{规定的精度要求} \text{ or } x_n == x_{n+1})$  then 迭代结束, 返回  $x_{n+1}$ ;

Else  $x_n \leftarrow x_{n+1}$ ;

Until 迭代次数超过了预先规定的上界

If (迭代次数超过了预先规定的上界) then 该迭代公式有可能不收敛, 返回 0。

在这里要设定一个迭代上限, 这样如果迭代公式不收敛, 程序能够保证在规定时间内结束, 不至于陷入死循环。对于本题, 由于已知求平方根的迭代公式是迭代收敛的, 因此可以省略掉这个迭代上限。

#### 程序清单 8-5

```

/*----- 8-5.c -----*/
#include "stdio.h"
#include "math.h"
double Sqrt(double a){          /*迭代法开方运算*/
    double xx = a, x = 0.0;     /*迭代初值*/
    while(fabs(xx - x)>0.00001){
        x = xx;
        xx = 0.5*(x + a / x);
    }
    return xx;                  /*返回迭代结果*/
}

main()
{
    double a, r;
    printf("Please input a digit for sqrt\n");
    scanf("%lf", &a);
    r = Sqrt(a);
    printf("Sqrt(%f) = %f\n", a, r);
    getche();
}

```

本程序的运行结果如图 8-9 所示。

```

Please input a digit for sqrt
7
Sqrt<7.000000> = 2.645751

```

图 8-9 程序 8-5 的运行结果



## 8.4 牛顿法解方程

## 【题目要求】

应用牛顿法解方程： $xe^x - 1 = 0$ 。

## 【题目分析】

在上一节中介绍了迭代法解方程的基本原理，以及基于迭代法解方程的思想计算平方根的方法。使用迭代法解方程  $f(x)=0$ ，必须先将原方程改写成  $x=\phi(x)$  的形式，然后总结出它的迭代公式。但是这样得到的迭代公式不一定会收敛，也可能收敛速度很慢，这样就需要重新调整迭代公式。因此，用迭代法求解方程的根并不很容易。

这里介绍一个所谓的牛顿公式，应用牛顿公式可以很快地求出方程的解。

设原方程为  $f(x)=0$ ；

已知方程的近似根为  $x_k$ ，函数  $f(x)$  在点  $x_k$  附近可展开成为一阶泰勒公式：

$$p(x) = f(x_k) + f'(x_k)(x - x_k)$$

也就是说在  $x_k$  附近可以用  $p(x)$  近似代替  $f(x)$ 。因此方程  $f(x)=0$  可以近似地表示为  $p(x)=0$ 。

取方程  $p(x)=0$  的根作为方程  $f(x)=0$  的新的近似根，记作  $x_{k+1}$ ，这样就有：

$$\begin{aligned} f(x_k) + f'(x_k)(x - x_k) &= 0 \\ \Rightarrow \frac{f(x_k)}{f'(x_k)} + x - x_k &= 0 \\ \Rightarrow x &= x_k - \frac{f(x_k)}{f'(x_k)} \\ \Rightarrow x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} \end{aligned}$$

这样就得到了一个迭代公式：

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

我们称之为牛顿公式。应用牛顿公式可以高效快速地求出方程  $f(x)=0$  的根。其实上一节中介绍的开方公式就是方程  $x^2 - c = 0$  对应的牛顿公式。牛顿法解方程是迭代法解方程的一个具体实例。

牛顿法解方程的思想是将非线性方程  $f(x)=0$  线性化为  $p(x)=0$ 。它的几何意义是很明显的，如图 8-10 所示。

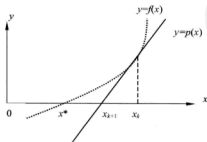


图 8-10 牛顿法的几何意义

如图所示,  $x^*$  是方程  $f(x)=0$  的精确解, 函数  $p(x)$  为函数  $f(x)$  在  $x_k$  附近的线性展开。线性方程  $p(x)=0$  的解为  $x_{k+1}$ ,  $x_{k+1}$  更接近方程  $f(x)=0$  的精确解  $x^*$ 。可以想象, 如果过  $(x_{k+1}, f(x_{k+1}))$  点再作  $f(x)$  的一条切线  $p(x)$ , 得到的解  $x_{k+2}$  会更接近精确解  $x^*$ 。最终, 牛顿公式收敛于方程的精确解。由于这种直观的几何意义, 牛顿法也称为切线法。

结合本题, 先确定方程  $xe^x - 1 = 0$  的牛顿公式。

$$x_{k+1} = x_k - \frac{x_k - e^{-x_k}}{1 + x_k}$$

然后就可以比较容易地写出解决本题的算法。算法描述如下:

```
double NewtonFun(double x)
{
    /*用牛顿公式计算出  $x_{k+1}$  的值*/
    return x - (x - exp(-x)) / (1+x);
}

double getResult(double x1, double accuracy)
{
    /*牛顿法解方程, 输入迭代初值 x1 和迭代精度 accuracy*/
    double x2 = 0.0;
    x2 = NewtonFun(x1);
    while(fabs(x2-x1) >= accuracy)
    {
        x1 = x2;
        x2 = NewtonFun(x1);
    }
    return x2;
    /*返回精度范围内的近似根*/
}
```

下面给出完整的代码源程序清单。

#### 程序清单 8-6

```
/*----- 8-6.c -----*/
#include "stdio.h"
#include "math.h"

double NewtonFun(double x)
{
    return x - (x - exp(-x)) / (1+x);
}

double getResult(double x1, double accuracy)
{
    double x2 = 0.0;
    x2 = NewtonFun(x1);
    while(fabs(x2-x1) >= accuracy)
    {
        x1 = x2;
        x2 = NewtonFun(x1);
    }
    return x2;
}

main()
{
    double x1, accuracy;
    printf("Please input the initial value\n");
    scanf("%lf", &x1);
    /*输入迭代初值*/
}
```

```

printf("Please input accuracy\n");          /*输入迭代精度*/
scanf("%lf",&accuracy);
printf("The result of function is %lf\n",getResult(x1,accuracy));
getche();
}

```

本程序的运行结果如图 8-11 所示。

```

Please input the initial value
0.5
Please input accuracy
0.0001
The result of function is 0.567143

```

图 8-11 程序 8-6 的运行结果

## 8.5 欧拉方法求解微分方程

### 【题目要求】

已知微分方程的初值问题如下：

$$\begin{cases} y' = y - \frac{2x}{y} & (0 < x \leq 1) \\ y(0) = 1 \end{cases}$$

求  $y(1)$  的值。

### 【题目分析】

在一些工程计算中，常遇到求解微分方程的问题。但与我们在微积分中遇到的微分方程问题不同，工程中的解微分方程并不一定希望得到原方程的解析解，即  $y=f(x)$  形式的解，往往只是希望得到  $x_n$  对应的某一点的值  $y(x_n)$  或者  $y_n$ 。因此这里讨论的微分方程求解都是这类的已知初值问题的求解。

应用计算机求解这类微分方程一般采用的方法都是“以差分代微分”的近似方法。其中最为简单的一种方法就是所谓的欧拉方法。

若在点  $x_n$  处列出方程：

$$y'(x_n) = f(x_n, y(x_n))$$

不难想象，可以近似地用差商： $\frac{y(x_{n+1}) - y(x_n)}{h}$  代替微商  $y'(x_n)$ ，于是得到：

$$\begin{aligned} \frac{y(x_{n+1}) - y(x_n)}{h} &\approx f(x_n, y(x_n)) \\ y(x_{n+1}) &\approx y(x_n) + hf(x_n, y(x_n)) \end{aligned}$$

将  $y(x_n)$  的近似值  $y_n$  代入方程的右边，并用  $y_{n+1}$  代替  $y(x_{n+1})$ ，这样可以推导出计算公式：

$$y_{n+1} = y_n + hf(x_n, y_n) \quad n = 0, 1, 2, \dots$$

这个公式也称为欧拉公式。当已知微分初值  $y_0$  时，可以通过该公式求出  $y_1, y_2, \dots$ 。它们分别是  $y(x_1), y(x_2), \dots$  的近似值。而  $x_n$  与  $x_{n+1}$  的差，即  $|x_n - x_{n+1}|$  的值取决于步长值  $h$ 。

虽然应用这种方法计算出来的微分方程的某一点的解  $y_n$  与精确解  $y(x_n)$  的偏差较大，但

是这种方法简单且易于实现，在精度要求不高的情况下是可以使用的。

下面给出应用欧拉方法求解微分方程的算法描述：

```
float getResult(float x0,float y0,float h,float xn)
{
    float yn;
    while(x0<xn)
    {
        yn = y0+h*func(x0,y0);      /*计算  $y_{n+1}=y_n+h f(x_n, y_n)$  */
        y0 = yn;                    /*  $y_{n+1}=y_n$  */
        x0 = x0+h;                  /*  $x_{n+1}=x_n+h$  */
    }
    return yn;
}
```

其中函数 func() 要根据具体的微分方程式来设定。对于本题，得到的欧拉公式为：

$$y_{n+1} = y_n + h(y_n - \frac{2x_n}{y_n})$$

因此函数 func() 可表示为：

```
float func(float x,float y)
{
    return y - 2*x / y;
}
```

本题的源代码程序清单如下：

程序清单 8-7

```
/*----- 8-7.c -----*/
#include "stdio.h"

float func(float x,float y)
{
    return y - 2*x / y;
}

float getResult(float x0,float y0,float h,float xn)
{
    float yn;
    while(x0<xn)
    {
        yn = y0+h*func(x0,y0);
        y0 = yn;
        x0 = x0+h;
    }
    return yn;
}

main()
{
    float x0,y0,h,xn;
    printf("Please input the initial value x0 & y0\n");
    scanf("%f %f",&x0,&y0);          /*输入初始值 x0 和 y0*/
    printf("Please input the step length\n");
    scanf("%f",&h);                  /*输入步长值 h*/
}
```

```

printf("Please input Xn\n");
scanf("%f",&xn);
printf("The result of the function is %f\n",getResult(x0,y0,h,xn));
getche();
}

```

/\*输入要求的  $y(x_n)$  的  $x_n$ \*/

本程序的运行结果如图 8-12 所示。

```

Please input the initial value x0 & y0
0 1
Please input the step length
0.1
Please input Xn
1.0
The result of the function is 1.784771

```

图 8-12 程序 8-7 的运行结果 (1)

本题的解析解为  $y = \sqrt{1+2x}$ ，因此  $y(1) = \sqrt{3} \approx 1.7321$ 。可以看出本题的误差还是比较大的。解决的方法是如果缩小步长值  $h$ ，如果  $h=0.0001$ ，则本程序的运行结果如图 8-13 所示。

```

Please input the initial value x0 & y0
0 1
Please input the step length
0.0001
Please input Xn
1.0
The result of the function is 1.732112

```

图 8-13 程序 8-7 的运行结果 (2)

显然这个结果与精确解更加接近。

## 8.6 改进的欧拉方法求解微分方程

### 【题目要求】

已知微分方程的初值问题如下：

$$\begin{cases} y' = y - \frac{2x}{y} & (0 < x \leq 1) \\ y(0) = 1 \end{cases}$$

求  $y(1)$  的值。

### 【题目分析】

8.5 节中介绍了应用欧拉方法求解微分方程的问题。虽然通过缩小步长值的方法，可以改进欧拉方法求解的精度，但是这样一来求解的步数就会增加，程序的效率也会随之降低。本节介绍一种改进版的欧拉方法，它可以用较少的计算步骤得出精度更高的解。

设微分方程：

$$y' = f(x, y)$$

将该方程两端从  $x_n$  到  $x_{n+1}$  求积分, 得到:

$$\begin{aligned}\int_{x_n}^{x_{n+1}} y' dx &= \int_{x_n}^{x_{n+1}} f(x, y) dx \\ y(x_{n+1}) - y(x_n) &= \int_{x_n}^{x_{n+1}} f(x, y(x)) dx \\ y(x_{n+1}) &= y(x_n) + \int_{x_n}^{x_{n+1}} f(x, y(x)) dx\end{aligned}$$

这样, 只要求出定积分  $\int_{x_n}^{x_{n+1}} f(x, y(x)) dx$  的值, 就可以得到  $y(x_{n+1})$  的计算公式。如果应用第 8.2 节中介绍的矩形公式计算定积分的值, 那么有

$$\int_{x_n}^{x_{n+1}} f(x, y(x)) dx \approx hf(x_n, y(x_n))$$

于是得到欧拉公式:

$$y(x_{n+1}) \approx y(x_n) + hf(x_n, y(x_n))$$

为了提高计算的精度, 可以用精度更高的梯形公式来代替矩形公式计算定积分。梯形公式为:

$$\int_{x_n}^{x_{n+1}} f(x, y(x)) dx \approx \frac{h}{2} [f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))]$$

这样计算  $y(x_{n+1})$  可表示为

$$y(x_{n+1}) \approx y(x_n) + \frac{h}{2} [f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))]$$

用  $y_n$  近似代替  $y(x_n)$ , 用  $y_{n+1}$  代替  $y(x_{n+1})$  可得到计算公式:

$$y_{n+1} = y_n + \frac{h}{2} [f(x_n, y_n) + f(x_{n+1}, y_{n+1})]$$

这个公式一般称为求解微分方程的梯形格式。但是要想应用梯形格式计算微分方程的解并不是一件容易的事情, 因为梯形格式本身是一种隐式算法, 它需要借助迭代过程求解, 因此计算量较大。

因此可以综合欧拉方法和梯形格式这两种方法, 先应用欧拉公式计算出  $y_{n+1}$  的近似值, 记作  $y'_{n+1}$ , 将其称为预报值; 再应用梯形格式, 将梯形格式右边的  $y_{n+1}$  用  $y'_{n+1}$  代替计算, 得到  $y_{n+1}$  (左边), 这个值称为校正值。因此建立起求解微分方程的预报/校正系统:

$$\begin{cases} \text{预报: } y'_{n+1} = y_n + hf(x_n, y_n) \\ \text{校正: } y_{n+1} = y_n + \frac{h}{2} [f(x_n, y_n) + f(x_{n+1}, y'_{n+1})] \end{cases}$$

该公式等价于以下的平均化形式:

$$\begin{cases} y_p = y_n + hf(x_n, y_n) \\ y_c = y_n + hf(x_{n+1}, y_p) \\ y_{n+1} = 1/2(y_p + y_c) \end{cases}$$

应用这种预报/校正系统求解微分方程的解可以提高计算的效率, 改善精度。其算法描述如下:

```
float getResult(float x0, float y0, float h, float xn)
{
    float yp, yc;
```

```

while(x0<xn)
{
    yp = y0 + h*func(x0,y0);           /*yp=yn+hf(xn,yn)*/
    yc = y0 + h*func(x0+h,yp);         /*yc=yn+hf(xn+1,yp)*/
    y0 = 1.0/2.0*(yp + yc);            /*yn+1=1/2 (yp+yc)*/
    x0 = x0 + h;                       /*xn+1=xn+h*/
}
return y0;
}

```

其中函数 func()要根据具体的微分方程式来设定。因此函数 func()可表示为:

```

float func(float x,float y)
{
    return y - 2*x / y;
}

```

本题的源代码程序清单如下。

程序清单 8-8

```

/*----- 8-8.c -----*/
#include "stdio.h"

float func(float x,float y)
{
    return y - 2*x / y;
}

float getResult(float x0,float y0,float h,float xn)
{
    float yp,yc;
    while(x0<xn)
    {
        yp = y0 + h*func(x0,y0);
        yc = y0 + h*func(x0+h,yp);
        y0 = 1.0/2.0*(yp + yc);
        x0 = x0 + h;
    }
    return y0;
}

main()
{
    float x0,y0,h,xn;
    printf("Please input the initial value x0 & y0\n");
    scanf("%f %f",&x0,&y0);           /*输入初始值 x0 和 y0*/
    printf("Please input the step length\n");
    scanf("%f",&h);                   /*输入步长值 h*/
    printf("Please input Xn\n");
    scanf("%f",&xn);                  /*输入要求的 y(xn) 的 xn*/
    printf("The result of the function is %f\n",getResult(x0,y0,h,xn));
    getch();
}

```


本程序的运行结果如图 8-14 所示。

```

Please input the initial value x0 & y0
0 1
Please input the step length
0.1
Please input Xn
1.0
The result of the function is 1.737867

```

图 8-14 程序 8-8 的运行结果

 **注意：**很显然与欧拉公式相比，在相同的步长值的前提下，预报/校正系统求解微分方程的精度要高很多。

## 8.7 雅可比迭代公式求解线性方程组

### 【题目要求】

求解线性方程组：

$$\begin{cases} 10x_1 - x_2 - 2x_3 = 7.2 \\ -x_1 + 10x_2 - 2x_3 = 8.3 \\ -x_1 - x_2 + 5x_3 = 4.2 \end{cases}$$

### 【题目分析】

在工程计算中，经常要遇到求解线性方程组的问题。在这里介绍一种比较简单的方法——雅可比迭代公式法求解线性方程组。

对于本题，要先将该方程组中的  $x_1, x_2, x_3$  分离出来，得到

$$\begin{cases} x_1 = 0.1x_2 + 0.2x_3 + 0.72 \\ x_2 = 0.1x_1 + 0.2x_3 + 0.83 \\ x_3 = 0.2x_1 + 0.2x_2 + 0.84 \end{cases}$$

再构造出迭代公式

$$\begin{cases} x_1(n+1) = 0.1x_2(n) + 0.2x_3(n) + 0.72 \\ x_2(n+1) = 0.1x_1(n) + 0.2x_3(n) + 0.83 \\ x_3(n+1) = 0.2x_1(n) + 0.2x_2(n) + 0.84 \end{cases}$$

在计算方程组的解时，先要给出迭代初值  $x_1(0), x_2(0), x_3(0)$ 。例如令  $x_1(0)=x_2(0)=x_3(0)=0$ 。然后代入上述迭代公式。当迭代次数  $n$  不断增加时，得到的迭代值  $x_1(n), x_2(n), x_3(n)$  会越来越接近方程组的解。

因此，对于一般形式的线性方程组

$$\sum_{j=1}^n a_{ij}x_j = b_i, \quad i=1, 2, \dots, n$$

分解出每一个未知数  $x_i$  得到

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j), \quad i=1, 2, \dots, n$$



从而得到每一个解的迭代公式

$$x_i(n+1) = \frac{1}{a_{ii}}(b_i - \sum_{j=1, j \neq i}^n a_{ij}x_j(n)), \quad i=1,2,\dots,n$$

该公式称为雅可比迭代公式。

应用雅可比迭代公式求解线性方程组的前提条件是要保证每个解的迭代公式都迭代收敛。因此雅可比迭代公式不一定适用于任何线性方程组的求解。所以在设计算法时，应当设置一个迭代上限 L，当迭代公式不收敛，或者收敛速度很慢时，可以停止程序执行。

下面给出求解本题的算法描述。

```
int getResult(float *x1,float *x2,float *x3 ,int L,float accuracy)
{
    float new_x1=*x1,new_x2=*x2,new_x3=*x3;           /*新值初始化*/
    float old_x1=100,old_x2=100,old_x3=100;           /*老值初始化*/
    int count=0;
    while((fabs(new_x1-old_x1)>accuracy || fabs(new_x2-old_x2)>accuracy
        || fabs(new_x3-old_x3)>accuracy)&&count<L)
    {
        /*当 x1, x2, x3 都达到精度要求，或迭代次数超过迭代上限，循环终止*/
        old_x1 = new_x1;           /*将新值赋值给老值，即 x(n)=x(n+1)*/
        old_x2 = new_x2;
        old_x3 = new_x3;
        new_x1 = 0.1*old_x2+0.2*old_x3+0.72;   /*由老值 x(n) 得到新值 x(n+1)*/
        new_x2 = 0.1*old_x1+0.2*old_x3+0.83;
        new_x3 = 0.2*old_x1+0.2*old_x2+0.84;
        count++;
    }
    if((fabs(new_x1-old_x1)>accuracy && fabs(new_x2-old_x2)>accuracy
        && fabs(new_x3-old_x3)>accuracy))           /*未达到精度要求，返回 0*/
        return 0;
    else
    {
        /*达到精度要求，返回 x1, x2, x3 和 1*/
        *x1 = new_x1;
        *x2 = new_x2;
        *x3 = new_x3;
        return 1;
    }
}
```

本算法只针对本题的方程组求解，如果要应用雅可比迭代公式计算其他方程组的解，可以在本算法的基础上修改参数的个数和迭代公式的具体内容。下面给出完整的程序清单。

程序清单 8-9

```
/*----- 8-9.c -----*/
#include "stdio.h"
#include "math.h"

int getResult(float *x1,float *x2,float *x3 ,int L,float accuracy)
{
    float new_x1=*x1,new_x2=*x2,new_x3=*x3;
    float old_x1=100,old_x2=100,old_x3=100;
    int count=0;
    while((fabs(new_x1-old_x1)>accuracy || fabs(new_x2-old_x2)>accuracy
```

```

        || fabs(new_x3-old_x3)>accuracy)&&count<L)
    {
        old_x1 = new_x1;
        old_x2 = new_x2;
        old_x3 = new_x3;
        new_x1 = 0.1*old_x2+0.2*old_x3+0.72;
        new_x2 = 0.1*old_x1+0.2*old_x3+0.83;
        new_x3 = 0.2*old_x1+0.2*old_x2+0.84;
        count++;
    }
    if((fabs(new_x1-old_x1)>accuracy && fabs(new_x2-old_x2)>accuracy
        && fabs(new_x3-old_x3)>accuracy))
        return 0;
    else
    {
        *x1 = new_x1;
        *x2 = new_x2;
        *x3 = new_x3;
        return 1;
    }
}

main()
{
    float x1,x2,x3,accuracy;
    int L;
    printf("Please input initial value of x1,x2,x3\n"); /*输入迭代初值*/
    scanf("%f %f %f",&x1,&x2,&x3);
    printf("Please input the accuracy\n"); /*输入迭代精度*/
    scanf("%f",&accuracy);
    printf("Please input the limitation for counting\n"); /*输入迭代上限*/
    scanf("%d",&L);
    if(getResult(&x1,&x2,&x3,L,accuracy))
        printf("x1 = %f;x2 = %f;x3 = %f\n",x1,x2,x3); /*输出结果*/
    else
        printf("Please adjust the accuracy or limitation\n");
        /*本次计算失败*/
    getch();
}

```

本程序的运行结果如图 8-15 所示。

```

Please input initial value of x1,x2,x3
0 0 0
Please input the accuracy
0.0001
Please input the limitation for counting
10
x1 = 1.099979;x2 = 1.199979;x3 = 1.299975

```

图 8-15 程序 8-9 的运行结果

已知原方程的精确解为：

$$\begin{cases} x_1 = 1.1 \\ x_2 = 1.2 \\ x_3 = 1.3 \end{cases}$$

可以通过缩小迭代精度和增大迭代上限的方法改善解的精度。如果将迭代精度缩小为 0.000001，将迭代上限增加为 20，可得到精确解，如图 8-16 所示。

```
Please input initial value of x1,x2,x3
0 0 0
Please input the accuracy
0.000001
Please input the limitation for counting
20
x1 = 1.100000;x2 = 1.200000;x3 = 1.300000
```

图 8-16 得到精确解



## 第9章 综合题

在前面几章中介绍了一些数学趣味问题、数据结构问题、数值计算问题等程序设计实例。每一类问题的侧重点不同，解决的方法和思路也有所差异。其实我们关键要掌握的还是如何建立起用计算机解决问题的思维，用程序设计的思想解决实际生活中遇到的问题。为了巩固读者业已掌握的编程技巧，加深读者对常用算法的理解程度，提高读者应用计算机解决和处理综合复杂问题的能力，本章将列举一些经典的综合题编程实例。这些题目生动有趣，同时具有一定的难度，因此作者尽量做到讲解深入浅出，把问题讲透彻，讲清楚。同时希望读者能从中得到启发，启迪思维，提高自身的编程水平。

### 9.1 破碎的砝码

#### 【题目要求】

法国数学家梅齐亚克在他所著的《数字组合游戏》中提出了这样一个问题：一个商人有一个质量为 40 磅的砝码，一天他不小心将该砝码摔成了 4 块。商人发现每块砝码的质量都是整磅数，而且每块砝码的质量各不相同，并且发现这 4 块砝码碎片可以在天平上称出 1~40 磅之间的任意质量（整磅数，即 1 磅，2 磅……）。问这 4 块砝码碎片的质量各是多少？

#### 【题目分析】

这是一道很有趣的题目。题目要求求出 4 块砝码碎片的质量各是多少，给定的条件如下：

- ☐ 砝码碎片的质量各不相同。
- ☐ 4 块砝码碎片可以在天平上称出 1~40 磅之间的任意质量（整磅数）。

因此可以想到，问题的求解空间是有限的、可列的，使用穷举法是一种比较简便的方法。

设 4 块砝码碎片的质量分别为  $a$ 、 $b$ 、 $c$ 、 $d$ ，只要穷举出这样的组合  $\{a, b, c, d\}$ ，其中  $a$ 、 $b$ 、 $c$ 、 $d$  互不相等， $a+b+c+d=40$ ，那么组合  $\{a, b, c, d\}$  就是该问题的可能解。如果存在这样一个组合  $\{a_i, b_i, c_i, d_i\}$ ，使得应用  $\{a_i, b_i, c_i, d_i\}$  可以在天平上称出 1~40 磅之间的任意质量（整磅数），那么这个组合自然就是该问题的答案。算法如下：

```
Repeat: 穷举出组合  $\{a, b, c, d\}$ ，其中  $a, b, c, d$  互不相等，并且  $a+b+c+d=40$ ；  
    If 组合  $\{a_i, b_i, c_i, d_i\}$  可以在天平上称出 1~40 磅之间的任意质量（整磅数）  
    Then 输出该组合方式（答案），返回  
    Else 继续进行穷举操作  
Until: 穷举完毕
```

现在就要解决两个问题:

(1) 如何穷举出所有的组合  $\{a, b, c, d\}$ , 其中  $a, b, c, d$  互不相等,  $a+b+c+d=40$ 。

(2) 如何判断组合  $\{a_i, b_i, c_i, d_i\}$  是否可以在天平上称出 1~40 磅之间的任意质量 (整数磅数)。

对于问题 (1) 有些读者可能会轻易地得出这样的方案, 用下面的算法穷举出组合  $\{a, b, c, d\}$ 。

```
for(i=1; i<40; i++)
    for(j=1; j<40; j++)
        for(k=1; k<40; k++)
            for(s=1; s<40; s++)
            {
                if(i, j, k, s 互不相等 && i+j+k+s==40)
                    得到一种组合{a, b, c, d}, 为该问题的可能解;
            }
```

这样的算法是低效而冗余的。因为这里所讲的组合  $\{a, b, c, d\}$  只是一种不考虑排列方式的组合, 即组合  $\{1, 2, 3, 4\}$  与组合  $\{4, 3, 2, 1\}$  其实是一样的。这是因为对于砝码碎片的质量而言, 每一片的质量是没有先后之分的, 这里所考虑的只是碎片的质量各是多少, 而并没有为每块砝码碎片编号。如果应用上述算法对可能解组合进行穷举, 会产生大量的冗余, 例如组合  $\{1, 2, 3, 34\}$  和组合  $\{34, 3, 2, 1\}$  都可以通过上述算法求出, 但是这两种组合方式其实是一回事。

这里介绍一种经典穷举的算法, 应用该算法可以有效地穷举出这种只考虑组合方式, 而不考虑排列方式的集合。算法如下:

```
for(i=1; i<=40; i++)
    for(j=i+1; j<=40-i; j++)
        for(k=j+1; k<=40-j-i; k++)
            for(s=k+1; s<=40-k-j-i; s++)
                if(i+j+k+s==40)
                {
                    得到一种组合{a, b, c, d}, 为该问题的可能解;
                }
```

上述算法的基本思想是: 确保内层循环搜索的范围小于外层循环搜索的范围。这样每次得到的组合方式的 4 个元素都不会相等, 同时也不会出现“相同组合, 不同排列”的情况。要从理论上解释或证明上述算法的正确性是有一定难度的, 这里不再详述, 有兴趣的读者可以研究。但是这个算法确实很实用, 也很经典, 建议读者熟记。

接下来的问题是如何判断组合  $\{a_i, b_i, c_i, d_i\}$  是否可以在天平上称出 1~40 磅之间的任意质量 (整数磅数)。如果将其形式化为数学符号, 它其实就是要判断方程:

$$x_1 a_i + x_2 b_i + x_3 c_i + x_4 d_i = W \quad x_1, x_2, x_3, x_4 \in \{-1, 0, 1\}$$

在  $W=1, 2, 3, \dots, 40$  时是否都有解。注意, 该方程的解  $x_1, x_2, x_3, x_4$  只能从  $\{-1, 0, 1\}$  中取值。这个道理很简单, 假设存在这样一组解  $\{x_1, x_2, x_3, x_4\} = \{1, 0, -1, 1\}$  使得方程

$$x_1 a_i + x_2 b_i + x_3 c_i + x_4 d_i = 1$$

有解, 则有:

$$\begin{aligned} 1 a_i + 0 b_i + (-1) c_i + 1 d_i &= 1 \\ \Rightarrow a_i + d_i &= c_i + 1 \end{aligned}$$

这样天平的一边放上碎片  $a_i$  和  $d_i$ ，另一边放上碎片  $c_i$  和一个 1 磅的砝码，天平就平衡了。这说明碎片  $\{a_i, b_i, c_i, d_i\}$  可以称出 1 磅的质量。同理，如果方程

$$x_1 a_i + x_2 b_i + x_3 c_i + x_4 d_i = W \quad x_1, x_2, x_3, x_4 \in -1, 0, 1$$

在  $W=1, 2, 3, \dots, 40$  时都有解（注意不是方程组，每组解可以不同），那就表明碎片  $\{a_i, b_i, c_i, d_i\}$  可以称出 1~40 磅之间的任意质量。

因此可以得出下面的算法判断组合  $\{a_i, b_i, c_i, d_i\}$  是否可以在天平上称出 1~40 磅之间的任意质量（整磅数）。

```
int justify(int i,int j,int k,int s)
{
    int weight;
    for(weight=1;weight<=40;weight++)
    {
        if(!getWeight(i,j,k,s,weight))    /*判断组合{i,j,k,s}是否可以称出
                                                1~40 磅之间的任意质量*/
            return 0;
    }
    return 1;
}
```

该算法通过一个循环（weight 从 1~40）来判断组合  $\{i,j,k,s\}$  是否可以称出 1~40 磅之间的任意质量。如果组合  $\{i,j,k,s\}$  可以称出 1~40 磅之间的任意质量，函数返回 1，否则返回 0。

其中函数 getWeight() 为一个子过程，它包含 5 个参数，参数 i、j、k、s 为组合  $\{i,j,k,s\}$  的元素，即砝码碎片的质量，weight 为 1~40 之间的某一个数。如果组合  $\{i,j,k,s\}$  能称出 weight 所表示的质量，则函数 getWeight() 返回 1，否则返回 0。getWeight() 的算法描述如下。

```
int getWeight(int i,int j,int k,int s,int weight)
{
    int x1,x2,x3,x4;
    for(x1=-1;x1<=1;x1++)
        for(x2=-1;x2<=1;x2++)
            for(x3=-1;x3<=1;x3++)
                for(x4=-1;x4<=1;x4++)
                    if(x1*i+x2*j+x3*k+x4*s == weight) /*存在一组解*/
                        return 1;
    return 0;
}
```

上述的算法描述仍然应用穷举法思想。

程序清单 9-1

```
/*----- 9-1.c -----*/
#include "stdio.h"

int getWeight(int i,int j,int k,int s,int weight)
{
    int x1,x2,x3,x4;
    for(x1=-1;x1<=1;x1++)
        for(x2=-1;x2<=1;x2++)
            for(x3=-1;x3<=1;x3++)
                for(x4=-1;x4<=1;x4++)
                    if(x1*i+x2*j+x3*k+x4*s == weight) //存在一组解
                        return 1;
    return 0;
}
```

```

        return 1;
    }
    return 0;
}

int justify(int i,int j,int k,int s)
{
    int weight;
    for(weight=1;weight<=40;weight++)
    {
        if(!getWeight(i,j,k,s,weight))
            return 0;
    }
    return 1;
}

pieces(){
    int i,j,k,s;

    for(i=1;i<=40;i++)
        for(j=i+1;j<=40-i;j++)
            for(k=j+1;k<=40-j-i;k++)
                for(s=k+1;s<=40-k-j-i;s++)
                    if(i+j+k+s==40)
                    {
                        if(justify(i,j,k,s)){
                            printf("The weight is broken up 4 pieces:\n");
                            printf("%d,%d,%d,%d",i,j,k,s);
                        }
                    }
}

main()
{
    pieces();
    getche();
}

```

本程序的运行结果如图 9-1 所示。

```

The weight is broken up 4 pieces:
1.3.9.2?

```

图 9-1 程序 9-1 的运行结果

## 9.2 计算 24 的问题

### 【题目要求】

在屏幕上输入 1~10 范围内的 4 个整数（可以有重复），对它们进行加、减、乘、除四则运算后（可以任意的加括号限定计算的优先级），寻找计算结果等于 24 的表达式。

例如输入 4 个整数 4, 5, 6, 7, 可得到表达式:  $4*((5-6)+7)=24$ 。这只是一个解，本题目要求输出全部的解。要求表达式中数字的顺序不能改变。

### 【题目分析】

本题最简便的解法是应用穷举法搜索整个解空间，筛选出符合题目要求的全部解。关键的问题是如何确定该题的解空间。假设输入的 4 个整数为 A、B、C、D，如果不考虑括

号优先级的情况，仅用四则运算符将它们连接，例如： $A+B \cdot C/D \cdots$ ，共有  $4^3=64$  种表达式情形。如果考虑加括号的情况，暂不考虑运算符，共有以下 5 种情形：

$((A \square B) \square C) \square D$   
 $(A \square (B \square C)) \square D$   
 $A \square (B \square (C \square D))$   
 $A \square ((B \square C) \square D)$   
 $(A \square B) \square (C \square D)$

其中  $\square$  代表  $(+,-,\cdot, /)$  任意的运算符。将上面两种情况综合起来考虑，每输入 4 个整数，其构成的解空间为  $64 \cdot 5=320$  种表达式。也就是说，每输入 4 个整数，无论以什么方式或优先级进行四则运算，其结果都会在这 320 种答案之中。我们的任务就是在这 320 种表达式中寻找出计算结果为 24 的表达式。

以上是应用穷举法解决本题目的基本思想。下面介绍具体的实施办法。

可以将不同位置上的运算符设置成为不同的变量： $op1$ 、 $op2$ 、 $op3$ 。规定  $op1$  为整数 A 与 B 之间的运算符； $op2$  为整数 B 与 C 之间的运算符； $op3$  为整数 C 与 D 之间的运算符。

A  $op1$  B  $op2$  C  $op3$  D

又规定变量  $op1$ 、 $op2$ 、 $op3$  取值范围为 1、2、3、4，分别表示加、减、乘、除四种运算。如表 xx 所示。

表 9-1 变量  $op1$ 、 $op2$ 、 $op3$  与运算符的对应情况

$op1$ 、 $op2$ 、 $op3$ 变量值	表示的运算	$op1$ 、 $op2$ 、 $op3$ 变量值	表示的运算
1	+	3	*
2	-	4	/

这样通过一个 4 重循环就可以枚举出不考虑括号情况的 64 种表达式类型。算法如下：

```

for (op1=1; op1<=4; op1++)
    for (op2=1; op2<=4; op2++)
        for (op3=1; op3<=4; op3++)
            {得到一种不含括号的表达式情形: A op1 B op2 C op3 D}

```

下面的问题就是考虑如何在表达式中添加括号，以及如何通过每种表达式的状态计算出对应的表达式的值。我们分别来介绍。

首先，上述算法得到的每一种表达式都可能具有 5 种添加括号的方式。而这 5 种添加括号的方式实际上涵盖了该表达式的所有优先级可能的运算。例如表达式： $A+B-C \cdot D$  的 5 种添加括号的方式为：

$((A+B)-C) \cdot D$   
 $(A+(B-C)) \cdot D$   
 $A+(B-(C \cdot D))$   
 $A+((B-C) \cdot D)$   
 $(A+B)-(C \cdot D)$

实际上，对表达式  $A+B-C \cdot D$  以任何优先级方式运算，都包含在这 5 种表达式之中。

例如不添加任何括号的表达式： $A+B-C \cdot D$  等价于表达式  $(A+B)-(C \cdot D)$ 。表达式： $A+(B-C) \cdot D$  等价于表达式： $A+((B-C) \cdot D)$ 。也就是说，上述 5 种表达式覆盖了  $(+,-,\cdot, /)$



这种运算符组合构成的表达式的全部解（以任何优先级进行计算）。因此就应当针对这 5 种添加括号的方式，分别计算出每一种表达式的值，看它是否等于 24。

由于每一种表达式的计算顺序（优先级）不尽相同，为了避免表达式运算的麻烦，可以设置 5 个函数，分别对应每一种类型表达式的计算。算法如下：

```
float calculate_model1(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    /*对应表达式类型: ((A□B)□C)□D*/
    float r1,r2,r3;
    r1 = cal(i,j,op1);
    r2 = cal(r1,k,op2);
    r3 = cal(r2,t,op3);
    return r3;
}

float calculate_model2(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    /*对应表达式类型: (A□(B□C))□D*/
    float r1,r2,r3;
    r1 = cal(j,k,op2);
    r2 = cal(i,r1,op1);
    r3 = cal(r2,t,op3);
    return r3;
}

float calculate_model3(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    /*对应表达式类型: A□(B□(C□D))*/
    float r1,r2,r3 ;
    r1 = cal(k,t,op3);
    r2 = cal(j,r1,op2);
    r3 = cal(i,r2,op1);
    return r3;
}

float calculate_model4(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    /*对应表达式类型: A□((B□C)□D)*/
    float r1,r2,r3;
    r1 = cal(j,k,op2);
    r2 = cal(r1,t,op3);
    r3 = cal(i,r2,op1);
    return r3;
}

float calculate_model5(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    /*对应表达式类型: (A□B)□(C□D)*/
    float r1,r2,r3 ;
    r1 = cal(i,j,op1);
    r2 = cal(k,t,op3);
    r3 = cal(r1,r2,op2);
    return r3;
}
```

上述算法中，每一个函数对应一种表达式的计算，其返回值为表达式的值，原表达式

的形式为  $i\ op1\ j\ op2\ k\ op3\ t$ ，不同的表达式类型，根据其运算时的优先级的不同进行不同的运算。其中函数 `cal()` 的作用是通过每种表达式的状态计算出对应的表达式的值。

函数 `cal()` 包括 3 个参数，第 3 个参数为运算符变量，它标志着不同种类的运算符（如表 xx 所示）。前两个参数为运算数。该函数的作用是将前两个参数指定的运算数进行第 3 个参数指定的运算，并返回其运算结果。例如函数调用 `cal(2,3,1)` 的作用是计算  $2+3=5$ ，并返回 5。函数 `cal()` 的算法描述如下：

```
float cal(float x,float y,int op)
{
    switch(op)
    {
        case 1: return x+y;      /*op 等于 1, 加法运算*/
        case 2: return x-y;      /*op 等于 2, 减法运算*/
        case 3: return x*y;      /*op 等于 3, 乘法运算*/
        case 4: return x/y;      /*op 等于 4, 除法运算*/
    }
}
```

将上面所讲的算法结合起来，就可以遍历由 4 个运算数（范围：1~10），3 个运算符（取值：+、-、\*、/），以任何优先级进行运算所构成的 320 种表达式，从而寻找其中答案为 24 的表达式。

如果更加细致深入地思考本题，本题的算法还可以改善。其实由 4 个整数和 4 种运算符（+、-、\*、/）以不同的优先级运算方式构成的表达式并不一定就是 320 种，有些表达式可能是完全等价的，例如

$$((A+B)-C)*D$$

$$(A+(B-C))*D$$

上面这两个表达式是完全等价的，无论 ABCD 为何值，它们的计算结果都是一样的。因此可以将上面这两个表达式合并成为一个表达式：

$$(A+B-C)*D$$

这样就缩小了搜索的空间，提高了查询的效率。但是这样做要考虑的问题就会多些，有兴趣的读者可以自己尝试完成。

程序清单 9-2

```
/*----- 9-2.c -----*/
#include "stdio.h"
char op[5]={'#','+','-','*','/'};

float cal(float x,float y,int op)
{
    switch(op)
    {
        case 1: return x+y;
        case 2: return x-y;
        case 3: return x*y;
        case 4: return x/y;
    }
}
```

```

float calculate_model1(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    float r1,r2,r3;
    r1 = cal(i,j,op1);
    r2 = cal(r1,k,op2);
    r3 = cal(r2,t,op3);
    return r3;
}

float calculate_model2(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    float r1,r2,r3;
    r1 = cal(j,k,op2);
    r2 = cal(i,r1,op1);
    r3 = cal(r2,t,op3);
    return r3;
}

float calculate_model3(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    float r1,r2,r3 ;
    r1 = cal(k,t,op3);
    r2 = cal(j,r1,op2);
    r3 = cal(i,r2,op1);
    return r3;
}

float calculate_model4(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    float r1,r2,r3;
    r1 = cal(j,k,op2);
    r2 = cal(r1,t,op3);
    r3 = cal(i,r2,op1);
    return r3;
}

float calculate_model5(float i,float j,float k,float t,int op1,int op2,int
op3)
{
    float r1,r2,r3 ;
    r1 = cal(i,j,op1);
    r2 = cal(k,t,op3);
    r3 = cal(r1,r2,op2);
    return r3;
}

get24(int i,int j,int k,int t)
{
    int op1,op2,op3;
    int flag=0;
    for(op1=1;op1<=4;op1++)
        for(op2=1;op2<=4;op2++)
            for(op3=1;op3<=4;op3++)
                if(calculate_model1(i,j,k,t,op1,op2,op3)==24)

```

```

        {printf("( (%d%c%d)%c%d)%c%d=24\n",i,op[op1],j,op[op2],k,op
[op3],t);flag = 1;}
        if(calculate_model2(i,j,k,t,op1,op2,op3)==24)
        {printf("( (%d%c(%d%c%d))%c%d=24\n",i,op[op1],j,op[op2],k,op
[op3],t);flag = 1;}
        if(calculate_model3(i,j,k,t,op1,op2,op3)==24)
        {printf("( (%d%c((%d%c%d))%c%d=24\n",i,op[op1],j,op[op2],k,op
[op3],t);flag = 1;}
        if(calculate_model4(i,j,k,t,op1,op2,op3)==24)
        {printf("( (%d%c(%d%c(%d%c%d))%c%d=24\n",i,op[op1],j,op[op2],k,op
[op3],t);flag = 1;}
        if(calculate_model5(i,j,k,t,op1,op2,op3)==24)
        {printf("( (%d%c(%d%c(%d%c%d))%c%d=24\n",i,op[op1],j,op[op2],k,op
[op3],t);flag = 1;}
    }
    return flag;
}
main()
{
    int i,j,k,t;
    printf("Please input four integer (1~10)\n");
loop:   scanf("%d %d %d %d",&i,&j,&k,&t);
    if(i<1||i>10 || j<1||j>10 || k<1||k>10 || t<1||t>10){
        printf("Input illege, Please input again\n");
        goto loop;
    }
    if(get24(i,j,k,t));
    else printf("Sorry, the four integer cannot be calculated to get
24\n");
    getch();
}

```

### 【程序说明】

本程序中，从主函数输入4个1~10的整数，调用函数get24()寻找符合要求（计算结果为24）的表达式。如果找到结果等于24的表达式，函数get24()将其输出，并返回1；如果没有找到结果等于24的表达式，函数get24()返回0。

本程序的运行结果如图9-2和图9-3所示。

(1) 找到结果等于24的表达式。

```

Please input four integer (1~10)
5 4 3 8
<<(5+4)/3> *8=24
(5+4)/(3/8)=24
<<(5-4)*3> *8=24
(5-4)<3*8>=24

```

图9-2 程序9-2的运行结果(1)

(2) 没有找到结果等于24的表达式。

```

Please input four integer (1~10)
3 2 6 7
Sorry, the four integer cannot be calculated to get 24

```

图9-3 程序9-2的运行结果(2)

## 9.3 马踏棋盘

### 【题目要求】

国际象棋的棋盘为  $8 \times 8$  的方格棋盘。现将“马”放在任意指定的方格中，按照“马”走棋的规则将“马”进行移动。要求每个方格只能进入一次，最终使得“马”走遍棋盘的 64 个方格。编写一个 C 程序，实现马踏棋盘操作，要求用 1~64 这 64 个数字标注马移动的路径，也就是按照求出的行走路线，将数字 1, 2, ……64 依次填入棋盘的方格中，并输出。

国际象棋中，“马”的移动规则如图 9-4 所示。

如图 9-4 所示，图中实心的圆圈代表“马”的位置，它下一步可移动到图中空心圆圈标注的 8 个位置上，该规则叫做“马走日”。但是如果“马”位于棋盘的边界附近，它下一步可移动到的位置就不一定有 8 个了，因为要保证“马”每一步都走在棋盘上。

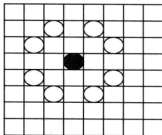


图 9-4 “马”的移动规则

### 【题目分析】

马踏棋盘的问题其实就是要将 1, 2, …, 64 填入到一个  $8 \times 8$  的矩阵中，要求相邻的两个数按照“马”的移动规则放置在矩阵中。例如数字  $a$  放置在矩阵的  $(i, j)$  位置上，数字  $a+1$  只能放置在矩阵的  $(i-2, j+1)$ ,  $(i-1, j+2)$ ,  $(i+1, j+2)$ ,  $(i+2, j+1)$ ,  $(i+2, j-1)$ ,  $(i+1, j-2)$ ,  $(i-1, j-2)$ ,  $(i-2, j-1)$  之中的一个位置上。将矩阵填满并输出。这样在矩阵中从 1, 2…遍历到 64，就得到了马踏棋盘的行走路线。因此本题的最终目的是输出一个  $8 \times 8$  的矩阵，在该矩阵中填有 1, 2…64 这 64 个数字，相邻数字之间遵照“马走日”的规则。

解决马踏棋盘问题的一种比较容易理解的方法是应用递归的深度优先搜索的思想。因为“马”每走一步都是盲目的，它并不能判断当前的走步一定正确，而只能保证当前这一步是可走的。“马”走的每一步棋都是从它当前位置出发，向下一步的 8 个位置中的 1 个行走（在它下一步有 8 个位置可走的情况下）。因此“马”当前所走的路径并不一定正确，因为它可能还有剩下的可选路径没有尝试，如图 9-5 所示。

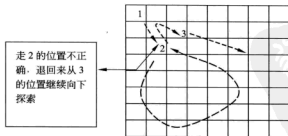


图 9-5 “马”的可选路径

如图 9-5 所示，假设最开始“马”位于棋盘的  $(0,0)$  的位置，接下来“马”有两处位置

可走，即(1,2)和(2,1)。这时“马”是无法确定走2的位置最终是正确的，还是走3的位置最终是正确的。因此“马”只能任意先从一个路径走下去（例如从2的位置）。如果这条路是正确的，那当然是幸运的，如果不正确，则“马”要退回到第一步，继续从3的位置走下去。以后“马”走的每一步行走都遵循这个规则。这个过程就是一种深度搜索的过程，同时也是一种具有重复性操作的递归过程。可以用一棵“探索树”来描述该深度优先搜索过程，如图9-6所示。

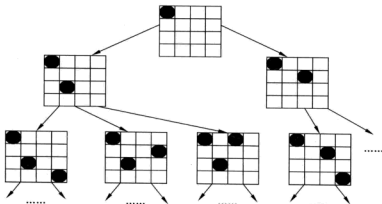


图9-6 深度优先搜索过程

“马”的行走过程实际上就是一个深度探索的过程。如图9-6所示，“探索树”的根结点为“马”在棋盘中的初始位置（这里用4\*4的棋盘示意）。接下来“马”有两种行走方式，于是根结点派生出两个分支。而再往下一步行走，根结点的两个孩子又能够分别派生出其他不同的“行走路线”分支，如此派生下去，就得到了“马”的所有可能的走步状态。可以想见，该探索树的叶子结点只可能有两种状态：一是该结点不能再生出其他的“走步”分支了，也就是“马”走不通了；二是棋盘中的每个方格都被走到，即“马”踏遍棋盘。于是从该探索树的根结点到第二种情况的叶结点构成的路径就是马踏棋盘的行走过程。

如何才能通过搜索这棵探索树找到这条马踏棋盘的行走路径呢？可以采用深度优先搜索的方法以先序的方式访问树中的各个结点，直到访问到叶结点。如果叶结点是第二种情况的叶结点，则搜索过程可以结束，因为找到了马踏棋盘的行走路径；如果叶结点为第一种情况的叶结点，即走不通了，则需要返回到上一层的结点，顺着该结点的下一条分支继续进行深度优先搜索下去。

因此在设计“马踏棋盘”的算法时可以借鉴前面讲过的图的深度优先遍历算法和二叉树的先序遍历算法。但是在这里并不需要真正地构建这样一棵探索树，我们只需要借用探索树的思想。在实际的操作过程中，所谓的探索树实际就是深度优先搜索的探索路径，每个结点实际就是当前的棋盘状态，而所谓的叶结点要么就是在当前棋盘状态下，“马”无法再进行下一步行走；要么就是马踏棋盘成功。该算法描述可如下：

```
int TravelChess Board (int x,int y,int tag)
{
    chess[x][y] = tag;
    if(tag == 64) { return 1;}
    找到“马”的下一个行走坐标 (x1,y1)，如果找到返回 flag=1,否则返回 flag=0;
```

```

while(flag ){
    if(TravelChessBoard (x1,y1,tag+1))return 1; /*递归调用 TravelChess ,
                                                从 x1,y1 向下搜索; 如果从
                                                x1,y1 往下马踏棋盘成功, 返
                                                回 1*/
    else
        继续找到“马”的下一个行走坐标(x1,y1), 如果找到返回 flag=1, 否则返回 flag=0;
    }
    if(flag == 0)
        chess[x][y] = 0;
    return 0;
}

```

该算法中通过函数 TravelChess() 递归地搜索“马”的每一种走法。其中参数  $x, y$  指定“马”当前走到棋盘中的位置,  $tag$  是标记变量, 每走一个棋盘方格,  $tag$  自动增 1, 它标识着马踏棋盘的行走路线。

算法首先将当前“马”处在棋盘中的位置上添加标记  $tag$ , 然后判断  $tag$  是否等于 64, 如果等于 64, 说明这是马踏棋盘的最后一步, 因此搜索成功, 程序应当结束, 返回 1。否则, 找到“马”下一步可以走到的位置( $x1, y1$ ), 如果找到这个位置坐标,  $flag$  置 1, 否则  $flag$  置 0。

下面在  $flag$  为 1 的条件下 (即找到  $x1, y1$ ), 递归地调用函数 TravelChess()。也就是从  $x1, y1$  指定的棋盘中的位置继续向下深度搜索。如果从  $x1, y1$  向下搜索成功, 即程序一直执行下去, 直到  $tag$  等于 64 返回 1, 那就说明“马”已经踏遍棋盘 (马踏棋盘的过程是: 先走到棋盘的( $x, y$ ) 位置, 再从( $x1, y1$ ) 向下深度搜索, 走遍全棋盘), 于是搜索结束, 返回 1; 否则继续找到“马”的下一个可以行走的坐标( $x1, y1$ ), 如果找到这个位置坐标,  $flag$  置 1, 并从( $x1, y1$ ) 向下重复上述的递归搜索, 否则  $flag$  置 0, 本次递归结束。

如果找遍当前位置( $x, y$ ) 的下一个坐标( $x1, y1$ ) (一般是 8 种), 但是从( $x1, y1$ ) 向下继续深度优先搜索都不能成功地“马踏棋盘” (此时  $flag$  等于 0), 则表明当前所处的状态并不处于马踏棋盘的“行走路径”上, 也就是说“马”本不应该走到( $x, y$ ) 的位置上, 因此将  $chess[x][y]$  置 0, 表明棋盘中该位置未被走过 (擦掉足迹), 同时返回 0, 程序退到上一层的探索状态。

这里应当知道, 所谓当前位置( $x, y$ ) 的下一个坐标( $x1, y1$ ) 是指“马”下一步可以走到的地方, 用坐标( $x1, y1$ ) 返回。在探索树中它处在( $x, y$ ) 所在的结点的子结点中, 如图 9-7 所示。

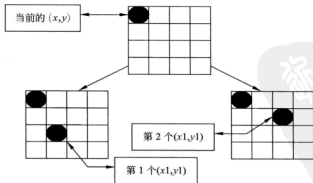


图 9-7 ( $x, y$ ) 的下一个坐标( $x1, y1$ )

当前位置(x,y)的下一个坐标(x1,y1)的可选个数由当前棋盘的局面决定。一般情况下是有8种可走的位置(如图9-7所示),但是如果“马”位于棋盘的边缘(如图9-7所示的探索树的根结点)或者8个可选位置中有的已被“马”前面的足迹所占据,在这两种情况下(x1,y1)的可选个数就不是8个了。

上述搜索算法相当于先序遍历探索树,只不过它不一定是将探索树完整地遍历,而是当tag等于64时,也就是棋盘被全部“踏遍”时就停止继续搜索了。

下面给出完整的程序清单供读者参考。

程序清单 9-3

```

/*----- 9-3.c -----*/
#include "stdio.h"
#define X 8
#define Y 8
int chess[X][Y];

int nextxy(int *x,int *y,int count) /*找到基于 x、y 位置的下一个可走的位置*/
{
    switch(count)
    {
        case 0: if(*x+2<=X-1 && *y-1>=0 && chess[*x+2][*y-1]==0) {*x = *x+2; *y = *y-1; return 1; } break;
        case 1: if(*x+2<=X-1 && *y+1<=Y-1 && chess[*x+2][*y+1]==0) {*x = *x+2; *y = *y+1; return 1; } break;
        case 2: if(*x+1<=X-1 && *y-2>=0 && chess[*x+1][*y-2]==0) {*x = *x+1; *y = *y-2; return 1; } break;
        case 3: if(*x+1<=X-1 && *y+2<=Y-1 && chess[*x+1][*y+2]==0) {*x = *x+1; *y = *y+2; return 1; } break;
        case 4: if(*x-2>=0 && *y-1>=0 && chess[*x-2][*y-1]==0) {*x = *x-2; *y = *y-1; return 1; } break;
        case 5: if(*x-2>=0 && *y+1<=Y-1 && chess[*x-2][*y+1]==0) {*x = *x-2; *y = *y+1; return 1; } break;
        case 6: if(*x-1>=0 && *y-2>=0 && chess[*x-1][*y-2]==0) {*x = *x-1; *y = *y-2; return 1; } break;
        case 7: if(*x-1>=0 && *y+2<=Y-1 && chess[*x-1][*y+2]==0) {*x = *x-1; *y = *y+2; return 1; } break;
        default: break;
    }
    return 0;
}

int TravelChessBoard(int x,int y,int tag) /*深度优先搜索地“马踏棋盘”*/
{
    int x1 = x, y1 = y, flag = 0, a, b, count = 0;
    chess[x][y] = tag;
    if(tag == X*Y) { return 1; }
    flag = nextxy(&x1,&y1,count);
    while(flag == 0 && count < 7) {
        count = count + 1;
        flag = nextxy(&x1,&y1,count);
    }
    while(flag) {

```



```

    if(TravelChessBoard(x1,y1,tag+1))return 1;
    x1 = x;y1 = y;
    count = count +1;
    flag = nextxy(&x1,&y1,count);    /*寻找下一个(x,y)*/
    while(flag == 0 && count <7){    /*循环地寻找下一个(x,y)*/
        count = count + 1;
        flag = nextxy(&x1,&y1,count);
    }
}
if(flag == 0)
chess[x][y] = 0;
return 0;
}
main()
{
    int i,j;
    for(i=0;i<X;i++)
        for(j=0;j<Y;j++)
            chess[i][j] = 0;
    if(TravelChessBoard(2,0,1)) {
        for(i=0;i<X;i++) {
            for(j=0;j<Y;j++)
                printf("%d ",chess[i][j]);
            printf("\n");
        }
        printf("The horse has travelled the chess board\n");
    }
    else
        printf("The horse cannot travel the chess board\n");
    getch();
}

```

### 【程序说明】

本程序中应用二维数组 chess[8][8]作为 8\*8 的棋盘，“马”每走到棋盘的(i,j)处，就将当前的步数 tag 赋值给 chess[i][j]。为了方便起见，将数组 chess 设置为外部变量。另外定义字符常量 X, Y, 它规定棋盘的大小。本程序清单中将 X 和 Y 都设置为 8。

函数 TravelChessBoard()是上述马踏棋盘算法的具体实现。本程序中初始的(x,y)位置为(2,0)，说明“马”从 chess[8][8]的 chess[2][0]位置开始进行“马踏棋盘”的。在函数 TravelChessBoard()中要调用函数 nextxy()。

```
int nextxy(int *x,int *y,int count)
```

函数 nextxy()包含 3 个参数：x 和 y 为传递下来的“马”当前踏到棋盘上的位置，它是指针变量。变量 count 的作用是标记调用 nextxy()过程的次数，根据 count 值的不同，返回基于(x,y)的下一个不同的坐标，以此来避免返回同一个坐标，真正实现寻找“针对当前位置(x,y)的下一个位置”的目的。函数 nextxy()的作用是找到“马”当前的位置(x,y)的下一个可以行走的位置，并通过指针修改 x、y 的内容，将下一个位置坐标用变量 x、y 返回。

“寻找(x,y)的下一个位置坐标”的操作通过代码：

```

x1 = x; y1 = y;    /*将坐标(x1,y1)初始化为当前访问的坐标
                    (x,y)，因为(x,y)不能被改动*/
flag = nextxy(&x1,&y1,count);    /*寻找(x,y)下一个位置坐标(x1,y1)*/
while(flag == 0 && count <7){    /*循环地寻找(x,y)的下一个坐标*/
    count = count + 1;
}

```

```
flag = nextxy(&x1,&y1,count);
}
```

实现。程序最开始将 $(x1,y1)$ 初始化为当前坐标 $(x,y)$ ，因为“马”的当前位置坐标 $(x,y)$ 不能被轻易改动。然后将 $\&x1$ 、 $\&y1$ 作为函数`nextxy`的参数传递，并通过参数 $\&x1$ 和 $\&y1$ 返回当前坐标 $(x,y)$ 的第`count`个下一个坐标 $(x1,y1)$ 。只有当`flag`为1时，才表明找到了下一个坐标，于是循环可以停止。但是如果`count`的值超过了7，则说明无法找到 $(x,y)$ 的下一个坐标，也就说明棋走不通了。所谓当前位置 $(x,y)$ 的“下一个位置”，如图9-8所示。

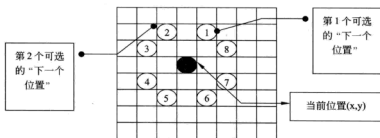


图9-8  $(x,y)$ 的“下一个位置”示意

图中实心的圆圈的坐标为 $(x,y)$ ，它周围的8个空心的圆圈1~8，为当前坐标 $(x,y)$ 的可选的“下一个”坐标 $(x1,y1)$ 。每次调用`nextxy()`都可能得到这8个坐标其中的一个，当参数`count`为 $i$ 时，返回圆圈 $i$ 所处的坐标。这样就保证了不返回同一个坐标。

如果像本程序这样将字符常量`X`和`Y`都设置为8，那么程序就执行 $8*8$ 的马踏棋盘。但是这样一来会非常费时，实践证明应用本程序运算出 $8*8$ 的马踏棋盘的结果要花几分钟的时间。因此读者在调试程序时可以通过设置`X`和`Y`的值减小棋盘的规格，从而更快地得到结果。

本程序的运行结果如图9-9所示。

```
43  50  47  38  57  52  61  32
48  37  44  51  46  33  58  53
1   42  49  56  39  60  31  62
36  15  40  45  34  29  54  59
41  2   35  16  55  24  63  30
14  5   12  9   22  19  28  25
3   10  7   20  17  26  23  64
6   13  4   11  8   21  18  27
The horse has travelled the chess board
```

图9-9 程序9-3的运行结果

## 9.4 0-1 背包问题

### 【题目要求】

给定 $n$ 种物品和一个背包。物品 $i$ 的质量为 $w_i$ ，其价值为 $v_i$ ，背包的最大载重量为 $c$ 。编写一个程序，求解如何装载背包中的物品，使得装入背包中的物品总价值最大。

## 【题目分析】

0-1 背包问题是一个经典的算法问题。解决 0-1 背包的方法很多，一般常用的方法包括：动态规划、分支限界、回溯法等。在这里介绍一种比较容易理解的方法——回溯法。

如果把这  $n$  种物品的取舍状态用一个向量表示  $\{x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_n\}$ ，其中  $x_i$  只有两种可能的取值 1 或 0。当  $x_i=1$  时，表明将第  $i$  个物品装入背包，当  $x_i=0$  时，表明不将第  $i$  个物品装入背包。那么这个 0-1 背包问题就可以形式化地描述为：

$$\max \sum_{i=1}^n v_i x_i$$

$$\text{结束条件} \begin{cases} \sum_{i=1}^n w_i x_i \leq c \\ x_i \in \{0, 1\}, 1 \leq i \leq n \end{cases}$$

也就是说 0-1 背包问题实质上就是求这样一个  $n$  维的向量  $\{x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_n\}$ ， $x_i \in \{0, 1\}$ ， $1 \leq i \leq n$ ，使得  $\sum_{i=1}^n v_i x_i$  的值最大，同时满足条件  $\sum_{i=1}^n w_i x_i \leq c$ 。

如何才能找到这样的向量呢？一种比较直观的方法是应用回溯法在两棵由 0-1 构成的解空间树中进行查找。解空间树如图 9-10 所示。

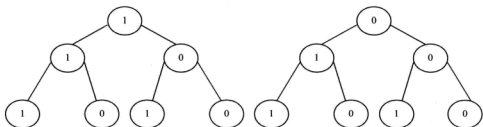


图 9-10 解空间树

如图 9-10 所示为向量  $\{x_1, x_2, x_3\}$  的解空间树。由概率论的知识可知，一个二值向量  $\{x_1, x_2, x_3\}$  的可能取值为  $2^3=8$  种，因此它的解空间相当于图 9-10 所示的两棵满二叉树，每一种可能的解都对应二叉树中从根结点到叶结点的一条路径。推而广之，一个包含  $n$  个元素的二值向量  $\{x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_n\}$  的可能取值为  $2^n$  种，它的解空间就是两棵由 0-1 构成的，包含  $n$  层的满二叉树。

不难想象，只要搜索这两个二叉树，找出向量  $\{x_1, x_2, \dots, x_i, x_{i+1}, \dots, x_n\}$  的所有的可能解（从根结点到叶结点的路径），再判断这些可能的解向量是否满足 0-1 背包的条件，找出符合条件的解向量，那么 0-1 背包问题就可以解决了。

在搜索解空间树寻找可能解的过程中，只要其孩子是一个可行结点，搜索就进入该孩子结点继续搜索下去，否则可以通过一个“剪枝”操作剪掉不可行结点以下的分支，以提高搜索的效率，减少搜索的次数。举例说明这一点。

例如有 3 件物品要装包，物品的重量分别为： $w_1=3, w_2=5, w_3=2$ ；物品的价值分别为： $v_1=9, v_2=10, v_3=7$ ；背包的载重量  $c=7$ 。在检索第一棵二叉树时，当搜索到根结点的左孩子结点时就会发现该结点是“不可行”的结点，如图 9-11 所示。

此时向量元素  $x_1$  的值为 1，因此它是一个不可行点，因为与向量  $X$  对应的物品重量向

量  $H=\{w_1, w_2, w_3\}$  的  $w_1+w_2>c$ , 因此这是不符合题目的要求的。如果沿着该结点继续搜索下去, 得到的可能解也一定不符合 0-1 背包题目的要求。因此不妨将该结点包含以下的分支全部剪掉, 转而沿着根结点的右孩子继续搜索下去, 如图 9-12 所示。

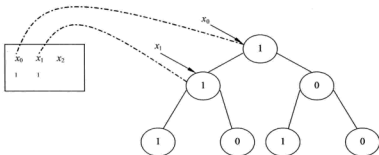


图 9-11 搜索到根结点的左孩子结点

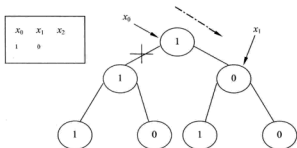


图 9-12 “剪枝”过程

因为要在解空间中筛选出价值总量最高的可能解, 而这样的解不一定只有一个 (0-1 背包问题可能有多个解), 因此需要在搜索 0-1 解空间树的过程中记录下搜索的路径, 然后比较哪 (些) 条路径对应的装包方案总价值量最高, 这一 (些) 条搜索路径对应的装包方案就是最终的解。但是这样做的空间复杂度较高, 因为每搜索到一条可能解路径都要将它记录下来, 对于  $n$  个物品的 0-1 背包问题, 可能要存储  $2^n$  条搜索路径, 因此空间占用量太大。

一种比较简单的方法是分两次回溯搜索解空间树, 第一次搜索计算出背包可装载物品的最大价值量, 第二次搜索计算出满足这个最大价值量的全部装包方案。这样做虽然有些费时, 但是简化了程序的设计, 减少了空间的使用。

在实际的算法设计中, 并不需要真的建立所谓的解空间树, 可以通过一个递归过程模拟这个搜索过程。算法描述如下:

```
int x[100]; //外部数组向量, 存放 n 种物品的取舍状态 (0-1) */
knap_1(int n, int flag, int c, int *price){ //找到物品包装的最高价值量*/
    int i, j, p;
    if(isOverLoad(n, c)) return; //超过背包的装载上限 c, 剪枝*/
    if(n == flag){ //搜索完一条可能的解路径*/
        p = getVal(n); //获得这种装包方案的物品的总价值量*/
    }
```

```

        if(*price< p) *price = p; /*price 存储各种装包方案中最高价值量*/
        return;
    }
    for(i=0;i<=1;i++)
    {
        x[n] = i; /*生成可能解路径*/
        knap_1(n+1,flag,c,price); /*递归调用 knap 过程*/
    }
}
knap_2(int n , int flag ,int c,int price){ /*找到物品装包的最高价值量 price*/
    int i,j,p;
    if(isOverLoad(n,c))return ; /*超过背包的装载上限 c, 剪枝*/
    if(n == flag){
        p =getVal(n);
        if(price == p) {
            /*如果该路径对应装包方案的价值总量等于 price*/
            输出该装包方案;
            return ; /*回溯继续搜索*/
        };
        return ; /*回溯继续搜索*/
    }
    for(i=0;i<=1;i++)
    {
        x[n] = i; /*生成可能解路径*/
        knap_2(n+1,flag,c,price); /*递归调用 knap 过程*/
    }
}

```

该算法包含两个函数 knap\_1()和 knap\_2()。

knap\_1()的作用是递归回溯地搜索解空间 ( $2^n$  种向量  $X$  的 0-1 排列)，计算出背包可装载物品的最大价值量。其中参数  $n$  的作用是作为数组  $x$  的下标，每递归调用 knap\_1() 一次， $n$  的值都会加 1 传递，通过这种方法在数组  $x$  中记录搜索的路径。flag 为物品的数量，当  $n$  等于 flag 时表明找到一条可能解的路径，即找到了一种可行的背包装载方案，但不一定是最佳的。参数  $c$  是装载上界，函数 isOverLoad()的作用是判断当前  $x$  中记录的搜索路径对应的物品重量是否超过装载上界  $c$ ，一旦物品的当前重量已经超过  $c$ ，则本次返回递归调用，相当于搜索二叉树时的剪枝操作，这样后续的分支就不再搜索了。参数 \*price 为一个指针型的变量，它的作用是返回背包可装载物品的最大价值量。当  $n$  等于 flag 时表明找到了一种可行的背包装载方案，这时调用函数 getVal 获得此背包装载方案对应的物品价值总量，如果该值大于 \*price，则将该价值总量赋值给 \*price，因此 \*price 中始终存放最大的价值量。

knap\_2()的作用是重新遍历搜索解空间，找到并输出与最大价值量 price 相等的装包方案。它的操作与 knap\_1()大致相同，只是在  $n$  等于 flag 时判断先计算好的最大价值量 price 是否等于 getVal 获得的背包装载方案对应的物品总价值总量。如果相等，则找到了一种背包装载方案，并输出该装包方案；否则继续回溯搜索。因为 0-1 背包问题可能有多个解，所以要搜索遍整个解空间，寻找出物品价值总量等于 price 的背包装载方案。

程序清单 9-4

```

/*----- 9-4.c -----*/
#include "stdio.h"
int x[100]; /*数组 x 用来存放路径*/

```

```

int val[100] = {-1};          /*存放物品的单价*/
int weight[100] = {-1};       /*存放物品的重量*/

int isOverLoad(int n,int c){
    int i,w = 0;
    for(i=0;i<n;i++){
        w = w + x[i] * weight[i];
        if(w > c) return 1;    /*超重*/
        else return 0;
    }
}

int getVal(int n){
    int i,v = 0;
    for(i=0;i<n;i++){
        v = v + x[i] * val[i];
    }
    return v;
}

knap_1(int n , int flag ,int c,int *price){ /*找到物品装包的最高价值量*/
    int i,j,p;

    if(isOverLoad(n,c))return; /*剪枝*/

    if(n == flag){
        p =getVal(n);
        if(*price< p) *price = p;
        return;
    }
    for(i=0;i<=1;i++){
        {
            x[n] = i;
            knap_1(n+1,flag,c,price);
        }
    }
}

int knap_2(int n , int flag ,int c,int price){ /*找到物品装包的最佳方案*/
    int i,j,p;
    if(isOverLoad(n,c))return; /*剪枝*/
    if(n == flag){
        p =getVal(n);
        if(price == p) {
            printf("-----bag-----\n");
            for(j=0;j<n;j++){
                if(x[j]==1){
                    printf(" |P%d:      | \n",j);
                    printf(" |weight:%2d kg| \n",weight[j]);
                    printf(" |price: %2d $ | \n\n",val[j]);
                }
            }
            printf("-----\n\n");
            getch();
            return ;
        }
    }
    return ;
}
for(i=0;i<=1;i++){
    {
        x[n] = i;
        knap_2(n+1,flag,c,price);
    }
}

```

```

}
main()
{
    int price=0, n ,c,i;
    printf("Input the number of products\n");
    scanf("%d",&n);
    printf("Input the weight of each product\n");
    for(i=0;i<n;i++)
        scanf("%d",&weight[i]);
    printf("Input the price of each product\n");
    for(i=0;i<n;i++)
        scanf("%d",&val[i]);
    printf("Input the limit weight the bag can overload\n");
    scanf("%d",&c);
    knap_1(0,n,c,&price);
    knap_2(0,n,c,price);
    printf("The grass price :%d $",price);
    getch();
}

```

### 【程序说明】

为了方便起见，本程序中将存放物品取舍状态的数组  $x$ ，存放每个物品价值的数组  $val$  和存放每个物品重量的数组  $weight$  都设置为外部变量，初始大小为 100，因此应用本程序最多计算 100 个物品的 0-1 背包问题。

函数  $isOverLoad()$  的作用是判断当前的搜索路径（数组  $x$  中的当前状态）对应的物品重量的总和是否超过装载上界  $c$ ，如果超重返回 1，否则返回 0。数组  $x$  存储的内容为物品取舍的状态，因而它对应着装入背包中的物品重量。假设数组  $x$  的当前状态为 {1, 0, 1, 0, 0, 0, 1, 0}，其对应的物品总重量为  $w = w_0 + w_2 + w_6$ ，如图 9-13 所示。

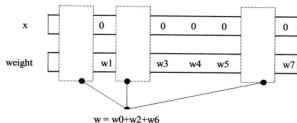


图 9-13 数组  $x$  对应的物品总重量

函数  $getVal()$  的作用是返回当前背包装载方案对应的物品总价值总量。假设数组  $x$  的当前状态为 {1, 0, 1, 0, 0, 0, 1, 0}，其对应的物品总价值为  $v = v_0 + v_2 + v_6$ ，如图 9-14 所示。

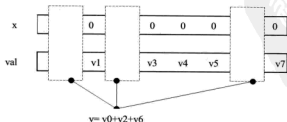


图 9-14 数组  $x$  对应的物品总价值量

本程序的运行结果如图 9-15 所示。

```

Input the number of products:
6
Input the weight of each product:
3 5 6 4 7 5
Input the price of each product:
8 9 6 7 5 6
Input the limit weight the bag can overload:
13
-----bag-----
:P0:                :
:weight: 3 kg:      :
:price: 8 $:        :
:P1:                :
:weight: 5 kg:      :
:price: 9 $:        :
:P2:                :
:weight: 4 kg:      :
:price: 7 $:        :
-----
The gross price :24 $
  
```

图 9-15 程序 9-4 的运行结果

**注意：**本程序运行时，输入 6 件物品，其价值和重量如表 9-2 所示，并输入背包最大载重量 13。

表 9-2 输入的物品信息

Weight	value	weight	value
3	8	4	7
5	9	7	5
6	6	5	6

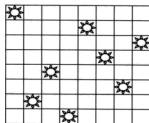
输入的运算结果为：

装包策略：将物品（weight: 3; value: 8）（weight: 5; value: 9）（weight: 4; value: 7）装入包中，总重量为 12，小于背包的最大载重量 13。总价值为 24。

## 9.5 八皇后问题求解

### 【题目要求】

八皇后问题是一道有趣而经典的数学问题。问题描述为：求解如何在一个 8\*8 的棋盘上无冲突地摆放 8 个皇后棋子。在国际象棋里，皇后的移动方式为横竖交叉的，因此在任意一个皇后所在位置的水平、竖直，以及 45 度斜线上都不能出现皇后的棋子。例如：





就是一种解。求出所有符合要求的摆放方法（即：求出所有的解）。

### 【题目分析】

八皇后问题是一道非常经典的数学问题。它的计算机解法很多，在第3章中曾介绍了应用回溯法解决四皇后问题，因此八皇后问题的一种常见的解法是使用回溯法。在这里将介绍另外一种方法求解八皇后问题——递归法。

要使用递归法求解八皇后问题，首先要构造八皇后问题的递归结构。其实我们可以把八皇后问题的解法简单地看成以下的两步。

- (1) 在棋盘的某行某列放置一枚皇后棋子。
- (2) 在其余的棋盘上构造出符合要求的八皇后局面。

结构如图9-16所示。

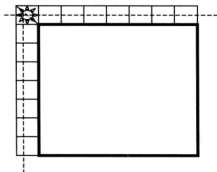


图9-16 八皇后问题的递归结构示意图(1)

如图9-16所示，假设棋盘的第一行第一列上放置了一枚皇后棋子，那么只要在除了第一行，第一列以及棋盘对角线的其余棋盘上放置下另外的7枚棋子，并且要求这7枚棋子同样符合八皇后问题的棋子摆放要求，则这便得出了八皇后问题的一个解。余下的7枚棋子都要放置到图中粗体边框所包围的棋盘区域内，并且这7枚棋子都不能在棋盘的对角线上。

那么如何在其余棋盘上放置下另外的7枚棋子呢？可以如图9-17这样摆放。

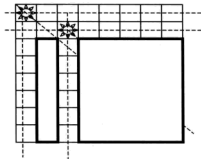


图9-17 八皇后问题的递归结构示意图(2)

如图9-17所示，在棋盘的第三行第二列上放置了一枚皇后棋子（它不处在棋盘的第一

行,第一列及对角线上),然后只要在其余的棋盘上(粗体边框所包围的棋盘区域内,不包括虚线穿过的方格)按要求放置另外的6枚棋子就可以构成一个符合摆放要求的7枚棋子的局面。很显然这个过程是重复执行了上述过程的,只是运算规模缩小了。这其实就是八皇后问题解法的递归结构,可以用下面这段伪码加以描述:

```
j=1
repeat:
在n*n的棋盘上第1行第j列放置一个皇后;
在剩余的棋盘上(非第1行第j列以及该皇后45度斜线上)构成一个符合要求的n-1皇后布局;
j←j+1
until j>n
```

按照上述算法所描述的步骤进行求解可得到八皇后问题的全部解。从宏观的角度来考虑这个问题,如果8\*8的棋盘上第一行第一列放置了一枚棋子,以此为基础可以产生出 $n_1$ 种不同的八皇后局面;如果棋盘上第一行第二列放置了一枚棋子,以此为基础又可以产生出 $n_2$ 种不同的八皇后局面……如果棋盘上第一行第八列放置了一枚棋子,以此为基础又可以产生出 $n_8$ 种不同的八皇后局面。这样八皇后问题的全部解即为 $n_1+n_2+\dots+n_8$ 种不同的局面。

以上只给出了八皇后问题递归解法的简略概述,具体的代码实现可参看下面的程序清单。

程序清单 9-5

```
/*----- 9-5.c -----*/
#include <string.h>
#include <stdio.h>
int count=0; /*计数变量,用来记录八皇后问题的解的个数,它是一个全局变量*/

/******
/*notEqual 函数为 EightQueen 函数所调用 */
/*目的是用来判断棋盘的 row 行第 j 列能否*/
/*摆放一个皇后,如果能够摆放返回1,否 */
/*则返回0. */
/******

int notEqual(int row,int j,int (*chess)[8])
{
    int i,k,flag1=0,flag2=0,flag3=0,flag4=0,flag5=0;
    for(i=0;i<8;i++)
        if((*chess+i)[j]!=0)
        {
            flag1=1;
            break;
        } /*判断列方向*/

    for(i=row,k=j;i>0 && k>0;i--,k--)
        if((*chess+i)[k]!=0)
        {
            flag2=1;
            break;
        } /*判断左上方*/

    for(i=row,k=j;i<8 && k<8;i++,k++)
        if((*chess+i)[k]!=0)
```

```

    {
        flag3=1;
        break;
    }/*判断右下方向*/

    for(i=row,k=j;i>=0 && k<8;i--,k++)
        if(*(chess+i+k)!=0)
        {
            flag4=1;
            break;
        }/*判断右上方向*/

    for(i=row,k=j;i<8 && k>=0;i++,k--)
        if(*(chess+i+k)!=0)
        {
            flag5=1;
            break;
        }/*判断左下方向*/

    if(flag1==1||flag2==1||flag3==1
       ||flag4==1||flag5==1)return 0; /*如果有一个方向不符合要求, 则返回 0. */
    else return 1; /*表明第 row 行第 j 列不能摆放皇后*/
    /*否则返回 1*/
}

/*****
/*EightQueen 函数实现八皇后问题的递归 */
/*求解。当形成符合要求的八皇后棋盘局面 */
/*时, 打印出棋盘的布局, 用一个 0-1 矩阵 */
/*表示棋盘, 0 代表空, 1 代表皇后。 */
/*参数: row 表示起始行; n 表示列数等于 8 */
/*(*chess)[8] 为指向棋盘每一行的指针 */
*****/

EightQueen(int row,int n,int (*chess)[8])
{
    int j;
    int k,t;
    int chess2[8][8];
    for(int i=0;i<8;i++)
        for(j=0;j<8;j++)
            chess2[i][j]=*(chess+i+j); /*复制棋盘, 用作递归使用*/

    if(row==8) /*递归结束条件, 形成符合要求的八皇后棋盘局面*/
    {
        for(k=0;k<8;k++){
            for(t=0;t<8;t++)
                printf("%d ",*(chess2+k+t)); /*打印棋盘*/
            printf("\n");
        }
        printf("\n\n");
        getchar();
        count++; /*记录解的个数*/
    }
    else /*不符合递归结束条件, 继续进行递归运算*/

        for(j=0;j<n;j++){
            if(notEqual(row,j,chess)) /*判断棋盘的 row 行第 j 列能否摆放一个皇后*/

```

```

    {
        for(i=0;i<8;i++)
            *(*(chess2+row)+i)=0;
            *(*(chess2+row)+j)=1;          /*向棋盘的第row行第j列摆放皇后*/
            EightQueen (row+1,n,chess2);    /*递归地调用EightQueen()函数*/
        }/*endif*/
    }/*endfor*/
}/*end else*/
}

/*主函数*/
main()
{
    int chess[8][8],i,j;
    for(i=0;i<8;i++)
        for(j=0;j<8;j++)
            chess[i][j]=0;                /*初始化棋盘，全部置0*/

    EightQueen(0,8,chess);                /*调用 EightQueen() 函数，参数：n=0, row=8, */
                                           /*chess 为指向棋盘每一行的指针 */
    printf("The number of the answer for eightqueen is\n");
                                           /*输出八皇后问题的解的个数*/
    printf("%d\n\n",count);
}

```

由于八皇后问题的全部解的个数为 92 种，因此无法在这里一一展示，只给出第一种解和最后一种解的棋盘布局情况，并显示记录下来的解的个数。

第 1 种解的棋盘布局如图 9-18 所示。

第 92 种解的棋盘布局如图 9-19 所示。

```

1 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0

```

图 9-18 程序 9-5 的运行结果 (1)

```

0 0 0 0 0 0 0 1
0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0

```

图 9-19 程序 9-5 的运行结果 (2)

输出八皇后问题的解的个数，如图 9-20 所示。

```

The number of the answer for eightqueen is
92

```

图 9-20 程序 9-5 的运行结果 (3)

## 9.6 简易文件加密/解密系统

### 【题目要求】

用 C 语言实现一个简易的文件加密/解密系统。该系统采用对称加密体制，由用户指定

自己的密钥  $key$ 。加密函数可描述为:  $M=P+key$ ; 对应的解密函数可描述为:  $P=M-key$ 。其中  $P$  表示 8 位的明文数据,  $M$  表示 8 位的密文数据。系统要求提供一个字符菜单界面, 可以对任意格式的文件进行加密/解密操作。

### 【题目分析】

对称加密体制是一种传统而经典的加密体制策略。所谓对称加密体制即加密方 A 和解密方 B 共享一个密钥  $key$ 。加密方 A 使用该密钥  $key$  对要保密的文件进行加密操作, 从而生成密文; 解密方 B 同样使用该密钥  $key$  对加密方生成的密文实施解密操作, 从而生成明文, 如图 9-21 所示。

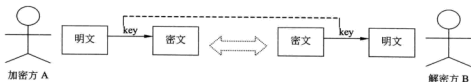


图 9-21 对称密码体制示意

如图 9-21 所示, 加密方和解密方共享同一个密钥  $key$ 。因此加密方与解密方的角色是对等的, 即加密方同样可以作为解密方, 使用密钥  $key$  对密文进行解密; 与此同时解密方也可以作为加密方使用密钥  $key$  对明文进行加密。这样的密码体制简单清晰, 易于实现。目前比较普遍使用的 DES、AES 等加密算法都是基于对称密码体制的加密算法。

本题要求实现一个简易的文件加密/解密系统, 并给出了自定义的加密解密算法(函数)。其中加密函数为  $M=P+key$ , 解密函数为  $P=M-key$ 。由于该系统基于对称加密体制, 因此加密函数与解密函数互为逆函数。只要将明文文件的一个字节代入加密函数中进行运算, 得到的结果即为一个字节的密文数据; 同理, 只要将密文文件的一个字节代入解密函数中进行运算, 得到的结果即为一个字节的明文数据。为了符合结构化程序设计的要求, 使得系统开发的流程更加规范, 下面列出该系统的设计过程。

### 一、概要设计

简易文件加密/解密系统的结构功能图如图 9-22 所示。



图 9-22 文件加密/解密系统的结构功能图

如图 9-22 所示, 该文件加密/解密系统总体上划分为 3 个模块——文件加密、文件解密以及系统菜单。按照自顶向下、逐步求精的原则, 又可将“文件加密”和“文件解密”两个子模块继续划分, 如图 9-23 所示。

如图 9-23 所示, 可将“文件加密”和“文件解密”两个子模块继续向下划分。其中读入明文是指将用户指定的磁盘上的明文文件读入到内存中的缓冲区中准备加密。加密明文是指将内存中的明文进行加密, 然后写回到内存中的缓冲区中, 这个过程需要用户指定密钥。保存密文是指将内存缓冲区中的密文数据以文件的形式保存在用户指定的目录下。“文

件解密”的三个子模块的功能与“文件加密”的功能恰好相反，但操作上是基本一致的，只是加密和解密的算法（函数）有所区别，因此可以最大限度地复用代码。



图 9-23 “文件加密”与“文件解密”子模块的划分

## 二、详细设计

对应上述划分的 6 个子模块，可将其分别映射为函数来调用。但是有些模块的操作是完全一致的，因此可以最大限度地复用代码。

将“读入明文”模块与“读入密文”模块合并为一个模块——“读入源文件”模块，因为他们的操作都是一样，都是将用户指定的磁盘上的文件（明文或密文）读入到内存中。读入源文件的函数定义如下：

```

int openSrcFile(char **buffer)
{
    FILE *myfile_src;           /*源文件指针*/
    char filename[20];          /*文件名数组*/
    long file_size;             /*记录文件的长度*/
    printf("Please input the path and filename of the file you want to process\n");
    scanf("%s", filename);      /*输入源文件的文件名*/
    if(! (myfile_src = fopen(filename, "rb")))
    {
        printf("ERROR!");
    }
    fseek(myfile_src, 0, SEEK_END); /*测试源文件的长度*/
    file_size = ftell(myfile_src);
    fseek(myfile_src, 0, SEEK_SET);
    *buffer = (char *)malloc(file_size);
    fread(*buffer, 1, file_size, myfile_src); /*读入文件到缓冲区 buffer*/
    fclose(myfile_src);          /*关闭源文件*/
    return file_size;           /*返回源文件的长度*/
}
  
```

在主调函数（调用函数 openSrcFile()的函数）中定义一个指向字符型变量的指针 char \*buffer，然后在函数 openSrcFile()中开辟内存缓冲区，将指定目录下的源文件（明文文件或密文文件）读入到该缓冲区中。该缓冲的首地址为 buffer，因此函数 openSrcFile()的入口参数为一个指向指针的指针 buffer，这样可以通过该参数直接修改主调函数中的指针变量 buffer。该函数的返回值为读入的文件（明文或密文）的长度。

文件加密函数定义如下：

```

void encryption(char buffer[], int file_size, int key)
{
    int i;
    for(i = 0; i < file_size; i++)
        buffer[i] = buffer[i] + key; /* M=P+key */
}
  
```

该函数将缓冲区 `buffer` 中存储的明文进行加密,并将密文存放到该缓冲区之中。参数 `file_size` 为文件的长度, `key` 为用户输入的密钥。

文件解密函数定义如下:

```
void decryption(char buffer[],int file_size,int key)
{
    int i;
    for( i = 0; i < file_size; i++)
        buffer[i] = buffer[i] - key;    /*P=M-key */
}
```

该函数将缓冲区 `buffer` 中存储的密文进行解密,并将明文存放到该缓冲区之中。参数 `file_size` 为文件的长度, `key` 为用户输入的密钥。

将“保存密文”模块与“保存明文”模块合并为一个模块——“保存目标文件”模块,因为它们的操作都是一样,都是将缓冲区 `buffer` 中的数据(明文数据或者密文数据)保存在用户指定的文件目录上。保存目标文件的函数定义如下:

```
void saveDstFile(char *buffer,long file_size)
{
    FILE *myfile_dst;           /*源文件指针*/
    char filename[20];          /*文件名数组*/
    printf("Please input the path and filename of the file you have\nprocessed\n");
    scanf("%s",filename);       /*输入目标文件的文件名*/
    if(!(myfile_dst = fopen(filename,"wb"))) /*打开目标文件*/
    {
        printf("ERROR!");
    }
    fwrite(buffer,1,file_size,myfile_dst); /*将 buffer 中的明文或密文写入文件*/
    printf("OK");
    fclose(myfile_dst);         /*关闭文件*/
}
```

调用函数 `saveDstFile()`时,主调函数(调用函数 `saveDstFile()`的函数)中定义的指针 `buffer` 指向的缓冲区中已存放了处理好的数据(明文数据或密文数据)。通过函数 `saveDstFile()`只是将缓冲区 `buffer` 中的内容写到用户指定的文件中。参数 `file_size` 为文件的长度,即缓冲区 `buffer` 的长度。

将上述 4 个函数通过一个函数 `Process()`整合在一起,函数 `Process()`由主函数 `main()`调用。函数 `Process()`的定义如下:

```
void Process(int a)
{
    char * buffer;
    int key;
    long file_size;             /*记录文件的长度*/
    file_size = openSrcFile(&buffer); /*读入源文件*/
    printf("Please input the key (a integer) for encryption or decryption\n");
    scanf("%d",&key);          /*用户输入密钥*/

    if(a == 0)
```

```

{
    encryption(buffer, file_size, key);    /*加密状态*/
}
else
{
    decryption(buffer, file_size, key);    /*解密状态*/
}
saveDstFile(buffer, file_size);           /*保存文件*/
}

```

函数 Process() 的参数由用户来指定, 当参数 a 等于 0 时为加密模式, 当 a 等于 1 时为解密模式。对文件(明文文件或密文文件)的读操作由函数 openSrcFile() 来完成, 对文件的写操作由函数 saveDstFile() 来完成。

主函数根据用户输入的不同命令调用函数 Process(), 并传递不同的参数, 对文件进行不同的处理。主函数 main() 的定义如下:

```

main()
{
    char flag;
    menu();                                /*显示系统菜单提示*/
    flag = getchar();
    getchar();
    while(flag != 'Q')                    /*根据用户输入的命令进行相应处理*/
    {
        switch(flag)
        {
            case 'E': Process(0); break;    /*加密文件*/
            case 'D': Process(1); break;    /*解密文件*/
            default: printf("Input Error!\n"); break; /*输入命令错误*/
        }
        flag = getchar();
        getchar();
    }
}

```

整个程序的调用关系结构如图 9-24 所示。

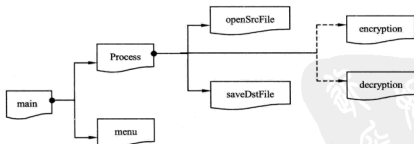


图 9-24 简易的文件加密/解密系统程序结构

有以下几点值得指出:

1. 这个“简易文件加密/解密系统”的设计过程遵循了“自顶向下的分析”和“自底



向上的设计”的原则，符合结构化程序设计的原则。

2. 题目中给定的加密函数  $M=P+key$  和解密函数  $P=M-key$  是不安全的，也不具有实用性，这里只是为了简化题目。在实际的加密/解密系统中，都是以更长的二进制位（64 位，128 位等）作为加密/解密的单元，不会像本题中只应用 8 位长的二进制位作为加密/解密的数据单元。同时密钥的长度也会更长。

3. 设置本题的目的在于向读者介绍“对称密码体制”的基本思想，并通过一个具体的实例简要地介绍一下结构化的软件开发方法。

下面给出“简易文件加密/解密系统”的完整的程序代码。

程序清单 9-6

```

/*----- 9-6.c -----*/
#include "stdio.h"

void encryption(char buffer[],int file_size,int key);
void decryption(char buffer[],int file_size,int key);
void Process(int a);
int openSrcFile(char **buffer);
void saveDstFile(char *buffer);
void menu();

main()
{
    char flag;
    menu();
    flag = getchar();
    getchar();
    while(flag != 'Q')
    {
        switch(flag)
        {
            case 'E':Process(0);break;
            case 'D':Process(1);break;
            default:printf("Input Error!\n");break;
        }
        flag = getchar();
        getchar();
    }
}

void menu()
{
    printf("*****\n");
    printf("-----A SIMPLE ENCRYPTION / DECRYPTION SYSTEM-----\n");
    printf("    ENCRYPTION press 'E'  DECRYPTION press 'D'  QUIT press 'Q'\n");
    printf("*****\n");
}

int openSrcFile(char **buffer)
{
    FILE *myfile_src;          /*源文件指针*/
    char filename[20];         /*文件名数组*/
    long file_size;            /*记录文件的长度*/
    printf("Please input the path and filename of the file you want to

```

```

process\n");
scanf("%s", filename);
if(!(myfile_src = fopen(filename, "rb")))
{
    printf("ERROR!");
}
fseek(myfile_src, 0, SEEK_END);
file_size = ftell(myfile_src);
fseek(myfile_src, 0, SEEK_SET);
*buffer = (char *)malloc(file_size);
fread(*buffer, 1, file_size, myfile_src);    /*读入文件*/
fclose(myfile_src);
return file_size;
}

void saveDstFile(char *buffer, long file_size)
{
    FILE *myfile_dst;                /*源文件指针*/
    char filename[20];               /*文件名数组*/
    printf("Please input the path and filename of the file you have
    processed\n");
    scanf("%s", filename);
    if(!(myfile_dst = fopen(filename, "wb")))
    {
        printf("ERROR!");
    }
    fwrite(buffer, 1, file_size, myfile_dst);
    printf("OK");
    fclose(myfile_dst);
}

void Process(int a)
{
    FILE *myfile_dst;
    char * buffer;
    int key;
    long file_size;                  /*记录文件的长度*/
    file_size = openSrcFile(&buffer); /*读入源文件*/
    printf("Please input the key (a integer) for encryption or decryp
    tion\n");
    scanf("%d", &key);               /*用户输入密钥*/

    if(a == 0)
    {
        /*加密状态*/
        encryption(buffer, file_size, key);
    }
    else
    {
        /*解密状态*/
        decryption(buffer, file_size, key);
    }
    saveDstFile(buffer, file_size);
}

void encryption(char buffer[], int file_size, int key)
{

```

```

int i;
for( i = 0; i < file_size; i++)
    buffer[i] = buffer[i] + key; /* M=2*(P+key) */
}

void decryption(char buffer[],int file_size,int key)
{
    int i;
    for( i = 0; i < file_size; i++)
        buffer[i] = buffer[i] - key;
}

```

程序的运行界面如图 9-25 所示。

```

===== A SIMPLE ENCRYPTION / DECRYPTION SYSTEM =====
ENCRYPTION press 'E'  DECRYPTION press 'D'  QUIT press 'Q'
=====
E
Please input the path and filename of the file you want to process
C:\test.txt
Please input the key (a integer) for encryption or decryption
5
Please input the path and filename of the file you have processed
C:\a.txt
OK
Input Error!
D
Please input the path and filename of the file you want to process
C:\a.txt
Please input the key (a integer) for encryption or decryption
5
Please input the path and filename of the file you have processed
C:\b.txt
OK

```

图 9-25 程序 9-6 的运行结果

如图 9-25 所示，在程序中输入字母 E 表示进入加密模式，按照提示语句首先输入明文的完整路径及文件名，然后输入密钥，最后输入密文保存的路径及文件名。当程序输出 OK 时表明加密成功。接下来输入字母 D 进入系统的解密模式，按照提示语句首先输入密文的完整路径及文件名，然后输入密钥（要求与加密时用的密钥一致），最后输入明文保存的路径及文件名。当程序输出 OK 时表明解密成功。当然加密操作和解密操作也可以在不同时刻进行、由不同用户完成，这里只要求加密时使用的密钥与解密时使用的密钥要一致，否则解密出的明文就是错误的。

加密前明文 C:\test.txt 的内容如图 9-26 所示。

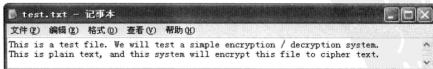


图 9-26 加密前的明文

加密后密文 C:\a.txt 的内容（乱码）如图 9-27 所示。

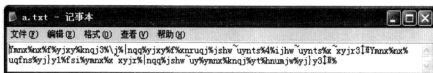


图 9-27 加密后的密文

解密后的明文 C:\b.txt 的内容如图 9-28 所示。

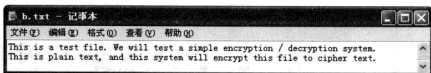


图 9-28 解密后的明文

## 第 10 章 算法设计与数据结构面试题精粹

在一些比较著名的公司的 C 语言相关面试中，往往会出一些基于 C 语言的算法设计和数据结构的题目。因为这类题目不但可以考查面试者最基本的编程功底，还可以考查面试者的综合素质，以及应用计算机解决实际问题的能力。另外，要想成为一名合格的程序设计人员，单纯地掌握编程语言的基本语法是远远不够的。熟练掌握一些常用算法的实现方法和常用的数据结构编程也是十分必要的。本章将介绍一些 C 语言算法设计和数据结构相关的题目。

### 10.1 常见的算法设计题

**【例 10-1】**输入一个字符串并将它输出，以 Ctrl+Z 组合键表示输入完毕，要求将输入的字符串中多于 1 个的连续空格符合并为 1 个。

例如输入字符串 Hello world!，输出为 Hello world!。

#### 【分析】

这类输入流操作的问题可以直接对输入的字符进行判断和处理，无需将输入的字符保存在数组或者其他数据结构中，这样会非常容易解决。

本题要求将输入的字符串中多于 1 个的空格符合并为 1 个，因此可以在输入每个字符时判断它是否是空格符。如果是空格符，则接下来的空格符不输出；否则输出接下来的任意字符。其算法描述如下：

```
inputString()
{
    char c, lastc = 'a';
    c = getchar();
    while(c!=EOF)
    {
        if(c!=' '){
            putchar(c);           /*输出该字符*/
            lastc = c;           /*保存该字符作为下一个输入字符的上一个字符*/
            c = getchar();       /*输入下一个字符*/
        }
        else{
            if(lastc!=' ')        /*如果上一个字符不为空格符*/
                putchar(c);      /*输出该空格符*/
            lastc = c;           /*保存该字符作为下一个输入字符的上一个字符*/
            c = getchar();       /*输入下一个字符*/
        }
    }
}
```

在该算法中,对输入字符 *c* 进行判断。(1)如果输入的字符不是空格符,则将它输出,并用变量 *lastc* 将其保存,作为下一个输入字符的参考,然后再输入下一个字符;(2)如果输入的字符是空格符,则要判断它的前一个字符 *lastc* 是否是空格符,仅当 *lastc* 不是空格符时才将 *c* 输出,否则不输出。同样这里要用变量 *lastc* 将该字符保存,作为下一个输入字符的参考,然后再输入下一个字符。重复(1)、(2)的操作,直到输入 Ctrl+Z 组合键(即 EOF)为止,循环结束。这样输出的字符串中多于 1 个的连续空格符就会被合并为 1 个输出。

由于第一个输入的字符无论是空格符都要被输出,因此 *lastc* 的初始值应为任意的非空格符字符,这里赋初值为字符 *a*。

这里要注意,只有当用户从终端输入回车符时,输入的字符串才作为一个输入流被 *getchar()* 函数一个一个地接收,再被函数 *putchar()* 一个一个地输出。

下面给出完整的测试程序,程序清单 10-1。

程序清单 10-1

```

/*----- 10-1.c -----*/
#include <stdio.h>
inputString()
{
    char c, lastc = 'a';
    c = getchar();
    while(c!=EOF)
    {
        if(c!=' '){
            putchar(c);
            lastc = c;
            c = getchar();
        }
        else{
            if(lastc!=' '){
                putchar(c);
                lastc = c;
                c = getchar();
            }
        }
    }
}
main()
{
    inputString();
    getch();
}

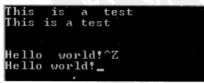
```

本程序的运行结果如图 10-1 所示。

**【例 10-2】**从终端输入 10 个整数,输出其中最大的数和次大的数。要求输入的 10 个整数互不相等。

#### 【分析】

解决这道题有两种考虑,一种考虑是将 10 个整数输入到一个数组中,然后从数组中筛选出最大的数和次大的数。但是题目并没有要求将 10 个整数保存,因此另一种考虑就是仿照上一题



```

This is a test
This is a test

Hello world!^Z
Hello world!_

```

图 10-1 程序 10-1 的运行结果

中的做法，在数据输入的过程中加以控制和比较。

可以设置两个变量 `maxVal` 和 `secondVal` 存储输入的数据。每次输入的数据 `a` 都与 `maxVal` 进行比较，如果 `a` 大于 `maxVal`，就将 `maxVal` 的值赋值给 `secondVal`，再将 `a` 赋值给变量 `maxVal`。如果 `a` 小于 `maxVal`，再将 `a` 与 `secondVal` 进行比较，如果 `a` 大于 `secondVal`，就将 `a` 赋值给 `secondVal`。这样就保证了 `maxVal` 中存放的是输入数据的最大值，`secondVal` 中存放的是输入数据的次大值。其算法可以描述如下：

```
int a,maxVal=-1000,secondVal=-1000,i;
/*maxVal 和 secondVal 初始化为绝对小值-1000*/
for(i=1;i<=10;i++)
{
    scanf("%d",&a);
    if(a>maxVal)
    {
        secondVal = maxVal; /*当输入的 a 大于 maxVal*/
        maxVal = a;
    }
    else /*当输入的 a 小于 maxVal*/
    {
        if(a>secondVal) /*当输入的 a 大于 secondVal*/
            secondVal = a;
    }
}
```

在这里将 `maxVal` 和 `secondVal` 赋初值为 `-1000`，认为它们是绝对小值，也就是输入的任何整数都要大于 `-1000`。

下面给出完整的测试程序，程序清单 10-2。

程序清单 10-2

```
/*----- 10-2.c -----*/
#include "stdio.h"
main()
{
    int a,maxVal=-1000,secondVal=-1000,i;
    for(i=1;i<=10;i++)
    {
        scanf("%d",&a);
        if(a>maxVal)
        {
            secondVal = maxVal;
            maxVal = a;
        }
        else
        {
            if(a>secondVal)
                secondVal = a;
        }
    }
    printf("The max value is   %d\n",maxVal); /*输出最大的数*/
    printf("The second value is %d\n",secondVal); /*输出次大的数*/
    getch();
}
```

本程序的运行结果如图 10-2 所示。

【例 10-3】编写一个程序，求分数序列 2, 3/2, 5/3, 8/5, 13/8... 的前 50 项和。

### 【分析】

解这类题的关键是找出数据序列的规律，然后通过循环语句累加求和。对于本题，仔细分析分数序列不难发现：序列的后一项分数的分母等于前一项分数的分子，序列后一项分数的分子等于前一项分数的分母与分子之和。根据这个规律，可以得到以下的算法。

```
1 3 5 7 9 2 4 6 8 0
The max value is 9
The second value is 8
```

图 10-2 程序 10-2 的运行结果

```
float Sum(int n)
{
    int i;
    float sum = 0, a=2, b=1, tmp;
    for(i=0; i<n; i++)
    {
        sum = sum + a/b;
        tmp = a + b;
        b = a;           /*后一项的分母等于前一项的分子*/
        a = tmp;         /*后一项的分子等于前一项分母与分子之和*/
    }
    return sum;
}
```

函数 Sum() 用于计算该分数序列的前  $n$  项和，参数  $n$  指定项数。其执行过程如下。

(1) 程序首先通过一个循环累加求和。用变量  $a$  表示每一项的分子，用变量  $b$  表示每一项的分母。按照分数序列每一项的变化规律，每次累加后都执行  $\text{tmp} = a + b;$  语句，记录分母与分子之和。

(2) 然后执行  $b = a;$  语句，使得分数序列后一项的分母等于前一项的分子；再执行  $a = \text{tmp};$  语句，使得分数序列后一项的分子等于前一项的分母与分子之和。共循环  $n$  次，可求出分数序列的前  $n$  项和。

(3) 最后返回累加后的结果  $\text{sum}$ 。

下面给出完整的测试程序，程序清单 10-3。

程序清单 10-3

```
/*-----10-3.c -----*/
#include "stdio.h"
float Sum(int n)
{
    int i;
    float sum = 0, a=2, b=1, tmp;
    for(i=0; i<n; i++)
    {
        sum = sum + a/b;
        tmp = a + b;
        b = a;           /*后一项的分母等于前一项的分子*/
        a = tmp;         /*后一项的分子等于前一项分母与分子之和*/
    }
    return sum;
}

main()
```



```

{
    printf("2+3/2+5/3+...=%f", Sum(50));    /*输出结算结果*/
    getch();
}

```

本程序的运行结果如图 10-3 所示。

**【例 10-4】**编写一个函数 `reverse(char *s)`，实现将字符串 `s` 的内容逆置。例如，原字符串的内容为 `abcd`，逆置后变为 `dcba`。要求不另外开辟字符串空间。

#### 【分析】

首先要将字符串保存在一个字符数组中，并将它的首地址作为函数 `reverse()` 的参数传递。在函数 `reverse()` 中，要实现将字符串内容的就地逆置（不另外开辟字符串空间），可以设置两个指针，分别指向字符串的首尾。然后将这两指针指向的位置互换，再将靠近字符串首的指针后移，靠近字符串尾的指针前移。重复上述操作，直到指针重合或者尾指针大于首指针（指针过界）为止。其算法描述如下：

```

void reverse(char *s)
{
    int len = strlen(s)-1, i=0;
    char tmp;
    while(i!=len && i<len)
    {
        tmp = s[i];           /*实现字符串中数据的逆置交换*/
        s[i] = s[len];
        s[len] = tmp;
        i++;
        len--;
    }
}

```

首先应用库函数 `strlen()` 计算出字符串 `s` 的长度，将 `len` 赋值为 `strlen(s)-1`，这样 `s[len]` 表示字符串 `s` 中最后一个有效字符。这是因为字符数组中最后一个元素为字符串结束标志，算法过程如图 10-4 所示。



图 10-3 程序 10-3 的运行结果

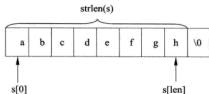


图 10-4 字符串长度及下标示意

然后通过一个循环交换 `s[i]` 与 `s[len]` 的内容，每次交换后执行 `i++` 和 `len--` 操作。循环的结束条件为 `i` 的值等于 `len` 的值（当字符串中字符的个数为奇数），或者 `i` 的值大于 `len` 的值（当字符串中字符的个数为偶数）。这样就可以实现一个字符串内容的就地逆置。

下面给出完整的测试程序，程序清单 10-4。

程序清单 10-4

```

/*----- 10-4.c -----*/
#include "stdio.h"
void reverse(char *s)
{

```

```

int len = strlen(s)-1,i=0;
char tmp;
while(i!=len && i<len)
{
    tmp = s[i];
    s[i] = s[len];
    s[len] = tmp;
    i++;
    len--;
}
}

main()
{
    char s[]="abcdefgh";
    printf("The string is %s\n",s);
    reverse(s);
    printf("The reversed string is %s\n",s);
    getch();
}

```

本程序的运行结果如图 10-5 所示。

**【例 10-5】**编写一个程序，将两个字符串连接，要求不破坏原有字符串。

```

The string is abcdefgh
The reversed string is hgfedcba

```

图 10-5 程序 10-4 的运行结果

#### 【分析】

很容易想到，将两个字符串连接到一起，可以将一个字符串的头部连接到另一个字符串的尾部。然后返回前面的那个字符串的首地址作为连接后字符串的首地址，该过程如图 10-6 所示。

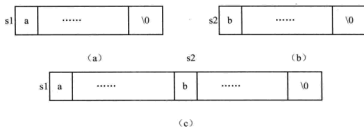


图 10-6 字符串 s1 和 s2 的连接

如图 10-6 所示，图 (a) 表示字符串 s1，图 (b) 表示字符串 s2，图 (c) 表示将字符串 s1 和 s2 连接到一起，形成一个新的字符串。

但是这样的解法并不符合题目的要求。因为题目中要求不破坏原有字符串，如果像上面介绍的方法进行字符串的连接操作，那么字符串 s1 和 s2 就都被破坏了。因此读者要审清题目。

正确的做法是另外开辟一个内存空间，它的长度为  $\text{strlen}(s1) + \text{strlen}(s2) + 1$ （最后一个存储单元存放字符串结束标志）。然后将字符串 s1 和 s2 按照指定的顺序复制到新的内存空间中去。这样就形成了一个新的字符串，它的内容是将原字符串 s1 和 s2 的内容连接在一起，同时也不破坏原有字符串。其算法描述如下：

```
char * cnnString(char *s1, char *s2)
```

```

{
    int len, len1, len2, i;
    char *s3;
    len1 = strlen(s1);          /*计算字符串的长度*/
    len2 = strlen(s2);
    len = len1 + len2 + 1;
    s3 = (char *)malloc(len);
    for(i=0; i<len1; i++)
        s3[i] = s1[i];          /*复制字符串 s1*/
    for(i=0; i<len2; i++)
        s3[len1+i] = s2[i];      /*复制字符串 s2*/
    s3[len-1] = '\0';           /*设置字符串 s3 的标识位*/
    return s3;                  /*返回新的字符串 s3*/
}

```

函数 `cnnString()` 实现将两个字符串 `s1` 和 `s2` 连接, 将 `s2` 连接到 `s1` 的尾部, 并返回连接后的新的字符串 `s3`。

- (1) 首先通过库函数 `strlen()` 计算字符串 `s1` 和 `s2` 的长度。
- (2) 然后动态开辟一段内存空间, 它的长度为 `strlen(s1)+strlen(s2)+1`, 并将这段内存空间的首地址赋值给指针变量 `s3`。这样 `s3` 指向新的字符串。
- (3) 然后通过两个循环语句分别将字符串 `s1` 和字符串 `s2` 顺序复制到字符串 `s3` 中。
- (4) 最后将 `s3` 的最后一个存储单元赋值为 `"\0"`, 作为字符串的结束标志。

下面给出完整的测试程序, 程序清单 10-5。

程序清单 10-5

```

/*----- 10-5.c -----*/
#include "stdio.h"
char * cnnString(char *s1, char *s2)
{
    int len, len1, len2, i;
    char *s3;
    len1 = strlen(s1);
    len2 = strlen(s2);
    len = len1 + len2 + 1;
    s3 = (char *)malloc(len);
    for(i=0; i<len1; i++)
        s3[i] = s1[i];          /*复制字符串 s1*/
    for(i=0; i<len2; i++)
        s3[len1+i] = s2[i];      /*复制字符串 s2*/
    s3[len-1] = '\0';
    return s3;
}
main()
{
    char s1[] = "This is a test ";
    char s2[] = "for connecting two string.";
    printf("%s\n", s1);          /*输出字符串 s1*/
    printf("%s\n", s2);          /*输出字符串 s2*/
    printf("%s\n", cnnString(s1, s2)); /*输出连接后的字符串 s3*/
    getch();
}

```

本程序的运行结果如图 10-7 所示。

```
This is a test
for connecting two string.
This is a test for connecting two string.
```

图 10-7 程序 10-5 的运行结果

【例 10-6】编写一个递归函数  $\text{sum}(\text{int } n)$ ，完成计算  $\sum_{i=1}^n i$ 。

#### 【分析】

设计递归算法是面试题中常见的一类问题。传统的计算累加求和的问题多用循环实现，但这里要使用递归方法。其实可以将题目中这个计算求和的函数  $\text{sum}()$  定义为递归函数如下：

$$\text{sum}(n) = \begin{cases} 1 & n=1 \\ n + \text{sum}(n-1) & n>1 \end{cases}$$

这是因为  $\text{sum}(n)$  表示计算  $1+2+\cdots+n$  的和。当  $n$  等于 1 时，很显然  $\text{sum}(n)$  等于 1，当  $n$  大于 1 时， $1+2+\cdots+n$  的和等于  $n+(1+2+\cdots+n-1)$ ，而  $1+2+\cdots+n-1$  的和可用  $\text{sum}(n-1)$  表示，因此构成了递归结构。

这样就可以方便地设计出  $\text{sum}$  的递归算法。

```
int sum(int n)
{
    if(n==1) return 1;
    else return n+sum(n-1);
}
```

下面给出完整的测试程序，程序清单 10-6。

程序清单 10-6

```
/*----- 10-6.c -----*/
#include "stdio.h"
int sum(int n) /*递归函数 sum() 求和*/
{
    if(n==1) return 1;
    else return n+sum(n-1);
}

main()
{
    int n;
    printf("Please input a integer for counting 1+2+...+n\n");
    scanf("%d",&n); /*输入求和数列的上限*/
    printf("The result of 1+2+...+n is\n",sum(n)); /*输出计算结果*/
    getch();
}
```

本程序的运行结果如图 10-8 所示。

```
Please input a integer for counting 1+2+...+n
100
The result of 1+2+...+100 is
5050
```

图 10-8 程序 10-6 的运行结果

**【例 10-7】**用递归方法编写一个程序，返回整型数组 `array` 中的最大值。数组 `array` 中的元素互不相等。

**【分析】**

要得到一个数组中的最大值最常见的方法是通过一个循环操作遍历数组中的元素，从而筛选出最大的元素。但是应用递归的方法也是一种很好的选择，因为寻找最大元素的过程本身具有递归特性。可以这样描述在一个数组中寻找最大元素的过程：

(1) 如果该数组中只有一个元素，那么该元素就是最大的值，因此将其返回。

(2) 如果该数组中的元素个数大于 1，那么将该数组中的第一个元素与后面其他元素中最大值进行比较。如果第一个元素大于后面其他元素中的最大值，则返回第一个元素，否则返回后面其他元素中的最大值。

那么计算“后面其他元素中的最大值”的过程又是重复执行 (1)、(2) 的过程，因此可以递归调用这个过程。其算法描述如下：

```
int arrayMaxVal(int array[],int n)
{
    if(n == 1) return array[0];
    if (array[0]>=arrayMaxVal(array+1,n-1))
        return array[0];
    else
        return arrayMaxVal(array+1,n-1);
}
```

函数 `arrayMaxVal()` 的作用是返回数组 `array` 中的最大值，参数 `n` 标志着数组 `array` 中元素的个数。首先判断 `n` 是否等于 1，如果 `n` 等于 1 表示当前数组中只有 1 个元素，因此返回 `array[0]`；否则比较 `array[0]` 和 `arrayMaxVal(array+1,n-1)` 的大小，并返回其中较大的值。`arrayMaxVal(array+1,n-1)` 是递归地调用函数 `arrayMaxVal()` 的过程，它返回当前数组中除第一个元素外，后面其他元素中的最大值。

下面给出完整的测试程序，程序清单 10-7。

程序清单 10-7

```
/*----- 10-7.c -----*/
#include <stdio.h>

int arrayMaxVal(int array[],int n)          /*递归获得一个数列中的最大值*/
{
    if(n == 1) return array[0];
    if (array[0]>=arrayMaxVal(array+1,n-1))
        return array[0];
    else
        return arrayMaxVal(array+1,n-1);
}

main()
{
    int array[]={1,2,13,11,7,9,3},max;
    max = arrayMaxVal(array,7);             /*获得数组 array 中的最大值*/
    printf("The max elem in the array is %d\n",max);    /*输出结果*/
    getche();
}
```

本程序的运行结果如图 10-9 所示。

The max elem in the array is 13

【例 10-8】任意给定输入一个小写英文字母串

图 10-9 程序 10-7 的运行结果

$a_1a_2a_3 \cdots a_{n-1}a_n$ 。输出字母串  $A_na_1A_{n-1}a_2A_{n-3}a_3 \cdots$

$A_2a_{n-1}A_1a_n$ ，其中  $A_i$  表示  $a_i$  的大写形式。例如输入 aybdx，输出为 XaDyBbYdAx。要求用递归方法实现。

### 【分析】

要应用递归的方法输出规定形式的字母串，就要找出按照一定规律重复的操作。可以这样描述这个过程：

假设小写英文字母串存放在数组  $str[]$  中，共包含  $n$  个元素  $str[0 \cdots n-1]$ 。按照输出的规律，首先输出  $upper(str[n-1])$ ，其中  $upper(c)$  表示字母  $c$  的大写形式，然后输出  $str[0]$ 。接下来输出  $upper(str[n-2])$ ，然后输出  $str[1] \cdots$ 。每一组都输出两个字母，它们的操作是一样的，只是数组  $str$  中的下标不同。因此可以构成一个递归过程，算法描述如下：

```
void func(char str[],int n,int i)
{
    if(i<n)
    {
        printf("%c%c",upper(str[n-i-1]),str[i]);
        func(str,n,i+1);
    }
}
```

函数  $func()$  包含 3 个参数， $str$  为数组名（数组的首地址）， $n$  表示数组中元素的个数， $i$  控制着输出的  $str$  数组元素的下标。当  $i$  小于  $n$  时，首先输出  $str[n-i-1]$  的大写形式，然后输出  $str[i]$ ，这样就完成了上面所讲的一组输出。然后递归地调用  $func()$  过程，参数  $i$  加 1 传递。直到  $i$  大于等于  $n$  时递归过程结束，函数一层一层地返回到第一层调用中。

下面给出完整的测试程序，程序清单 10-8。

程序清单 10-8

```
/*----- 10-8.c -----*/
#include "stdio.h"
char upper(char c) /*将小写字母转换为大写字母*/
{
    return(c-'a'+'A');
}

void func(char str[],int n,int i) /*递归法按要求格式输出字符串*/
{
    if(i<n)
    {
        printf("%c%c",upper(str[n-i-1]),str[i]);
        func(str,n,i+1);
    }
}

main()
{
    char str[5]={'a','b','c','d','e'}; /*初始化字符串*/
    printf("The string of transeration is\n");
    func(str,5,0); /*递归调用函数 func()*/
}
```

```

getche();
}

```

本程序的运行结果如图 10-10 所示。

**【例 10-9】**编写一个函数 `reverse(char *s)`，实现将字符串 `s` 的内容就地逆置。要求用递归方法实现。

```

The string of transference is
EaDhCcBdRe_

```

图 10-10 程序 10-8 的运行结果

**【分析】**

在例题 10.4 中介绍了 `reverse()` 的非递归解法，但这里要求使用递归方法实现。可以仿照上一例题的做法，设 `len=strlen(s)-1`，`i=0`。首先进行 `s[i]` 和 `s[len]` 的内容的置换，然后 `i` 自增 1，`len` 自减 1。重复上述操作，这个过程用递归方法来实现，直到 `i` 的值等于 `len` 的值（当字符串中字符的个数为奇数），或者 `i` 的值大于 `len` 的值（当字符串中字符的个数为偶数）为止。算法描述如下：

```

void reverser(char *s,int len,int i)
{
    char tmp;                                /*定义临时变量*/
    if(i<len)
    {
        tmp = s[len];
        s[len] = s[i];                        /*字符的置换*/
        s[i] = tmp;
        reverser(s,len-1,i+1);               /*递归调用函数 reverser()*/
    }
}

void reverse(char *s)                        /*保持接口一致，定义函数 reverse()*/
{
    int len = strlen(s);
    len--;
    reverser(s,len,0);                       /*调用递归函数 reverser()*/
}

```

函数 `reverse()` 是一个启动函数，它调用真正的递归函数 `reverser()`，这样符合题目的要求，同时与例题 10.4 中的 `reverse()` 过程接口一致，易于软件的移植。在递归函数 `reverser()` 中，当 `i` 小于 `len` 时，就将 `s[len]` 的内容与 `s[i]` 的内容交换位置，然后递归地调用 `reverser()` 函数，此时参数 `len` 减 1，参数 `i` 加 1。这个递归过程直到 `i ≥ len` 时结束。

下面给出完整的测试程序，程序清单 10-9。

程序清单 10-9

```

/*----- 10-9.c -----*/
#include "stdio.h"

void reverser(char *s,int len,int i)
{
    char tmp;                                /*定义临时变量*/
    if(i<len)
    {
        tmp = s[len];
        s[len] = s[i];                        /*字符的置换*/
        s[i] = tmp;
        reverser(s,len-1,i+1);               /*递归调用函数 reverser()*/
    }
}

```

```

    }
}

void reverse(char *s)           /*保持接口一致，定义函数 reverse()*/
{
    int len = strlen(s);
    len--;
    reverser(s, len, 0);        /*调用递归函数 reverser()*/
}

main()
{
    char s[]="abcdefg";
    printf("The original string is %s\n",s);
    reverse(s);
    printf("The reversed string is %s\n",s);
    getch();
}

```

本程序的运行结果如图 10-11 所示。

**【例 10-10】**有苹果、橘子、香蕉、菠萝、梨这 5 种水果，已知每个果盘中一定有 3 种水果，并且 3 种水果的种类各不相同。编程计算可以制作出多少种水果拼盘。

```

The original string is abcdefg
The reversed string is gfedcba

```

图 10-11 程序 10-9 的运行结果

#### 【分析】

本题最容易最直观的解法就是使用穷举法。如果用 3 个变量  $x$ 、 $y$ 、 $z$  表示每一种果盘中的 3 种水果，用常量 1~5 分别表示苹果、橘子、香蕉、菠萝、梨这 5 种水果，将 1~5 分别赋值给变量  $x$ 、 $y$ 、 $z$ ，每一种赋值表示一种装盘方法，那么不难想象共有  $5^3$  种装盘方案。这  $5^3$  种装盘方案构成了本题的解空间。但是这  $5^3$  种装盘方案并不全是答案，因为题目要求每个果盘中 3 种水果的种类各不相同，因此需要添加约束条件  $x \neq y \neq z$ 。这样就很容易得到解决本题的算法。

```

fruitPlate()
{
    int x,y,z;
    for(x=1;x<=5;x++)
        for(y=1;y<=5;y++)
            for(z=1;z<=5;z++)
            {
                if(x!=y && y!=z && x!=z)
                    输出一种拼盘方案，并计数
            }
}

```

通过上述算法 `fruitPlate()` 可以得到符合要求的拼盘方案。在这里使用一个三重循环，使得变量  $x$ 、 $y$ 、 $z$  在 1~5 之间有规律地改变，从而得到  $5^3$  种不同的组合形式。当  $x!=y$ ， $y!=z$ ， $x!=z$  时为符合题目要求的答案。

下面给出完整的测试程序，程序清单 10-10。

程序清单 10-10

```

----- 10-10.c -----*/
#include "stdio.h"

char fruit[][10]={"apple","orange","banana","pineapple","pear"};

```



```

int fruitPlate()
{
    int x,y,z,count=0;
    for(x=1;x<=5;x++)
        for(y=1;y<=5;y++)
            for(z=1;z<=5;z++)
            {
                if(x!=y && y!=z && x!=z)
                {
                    count++;
                    printf("%9s,%9s,%9s\n",fruit[x-1],fruit[y-1],
                        fruit[z-1]);
                }
            }
    return count;
}

main()
{
    printf("There are %d kinds of methods for arranging plate.\n",
        fruitPlate());
    getch();
}

```

本程序的运行结果如图 10-12 所示。

```

pineapple, apple, banana
pineapple, apple, pear
pineapple, orange, apple
pineapple, orange, banana
pineapple, orange, pear
pineapple, banana, apple
pineapple, banana, orange
pineapple, banana, pear
pineapple, pear, apple
pineapple, pear, orange
pineapple, pear, banana
pear, apple, orange
pear, apple, banana
pear, apple, pineapple
pear, orange, apple
pear, orange, banana
pear, orange, pineapple
pear, banana, apple
pear, banana, orange
pear, banana, pineapple
pear, pineapple, apple
pear, pineapple, orange
pear, pineapple, banana
There are 60 kinds of methods for arrangeing plate.

```

图 10-12 程序 10-10 的运行结果

【例 10-11】请在屏幕上输出一张乘法口诀表，形如：

```

1*1=1
1*2=2 2*2=4
1*3=3 2*3=6 3*3=9
1*4=4 2*4=8 3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

```

## 【分析】

解决这类特定格式输出的问题的关键是找出输出格式的规律，然后根据抽象出来的规律，应用简单的循环语句就可以显示出希望得到的结果。

现在来分析一下乘法口诀表的输出规律。如上图所示，一张乘法口诀表可以看作由 81 组算式构成的三角形数表。其中，每个算式都是一个乘法式，并给出计算结果（例如： $3*8=24$ ）。仔细观察不难发现，每个算式的被乘数都等于该算式所在口诀表中的列数，每个算式的乘数都等于该算式所在口诀表中的行数。即：如果一个算式为  $i*j=n$ ，那么该算式一定处在口诀表中的第  $j$  行，第  $i$  列， $n$  为  $i*j$  的计算结果。另外，口诀表的第  $j$  行中只输出  $j$  个乘法算式，共输出 9 行。

根据这个规律，就可以设计出输出一张乘法口诀表的算法。如果是按行输出该乘法口诀表，就可以应用一个二重循环语句。其中，外层循环控制行数，用变量  $j$  控制；内层循环控制列数，用变量  $i$  控制。那么每一个输出的算式自然就是  $i*j=n$ 。

下面给出解决该问题的伪码算法描述：

```
j←1, i←1
repeat:
    repeat:
        输出 i*j 以及其结果
        i←i+1
    until i>j
    打印 1 个 "\n"
j←j+1
until j>9
```

/\*实现输出乘法口诀表的第 j 行\*/

下面给出完整的测试程序，程序清单 10-11。

程序清单 10-11

```
/*----- 10-11.c -----*/
#include <string.h>
#include <stdio.h>

void PrintMulTab();

int main()
{
    PrintMulTab();
    getchar();
}

void PrintMulTab()
{
    int i, j;
    for (j=1; j<=9; j++)
    {
        for(i=1; i<=j; i++)
            printf("%d*%d=%d ", i, j, j*i);
        printf("\n");
    }
}
```

本程序的运行结果如图 10-13 所示。

```

i=1-1
i=2-2 2*2=4
i=3-3 2*3=6 3*3=9
i=4-4 2*4=8 3*4=12 4*4=16
i=5-5 2*5=10 3*5=15 4*5=20 5*5=25
i=6-6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
i=7-7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
i=8-8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
i=9-9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81

```

图 10-13 程序 10-11 的运行结果

【例 10-12】一个整数，它加上 100 后是一个完全平方数，再加上 168 又是一个完全平方数，编程计算该数是多少？

#### 【分析】

本题最简单的解法是应用穷举法，在规定的整数范围内逐一寻找符合条件的整数。一旦找到了这个整数，程序就停止继续寻找，返回该整数。关键在于如何确定这个整数范围。对于本题这个范围并不明确，只知道它是一个整数，因此可以试探性地规定这个范围。解决本题的算法描述如下：

```

func(int low,int high)
{
    int i;
    double x,y;
    for(i=low;i<=high;i++)
    {
        x=sqrt(i+100);           /*x 为加上 100 后开方后的结果*/
        y=sqrt(i+168);           /*y 为再加上 168 后开方后的结果*/
        if(x*x==i+100&&y*y==i+168) /*判断此数是否是完全平方数*/
            return i;
    }
    return 0;
}

```

函数 func()的功能是在 low 和 high 的整数范围内寻找满足条件的整数。如果找到该整数，则将其返回，否则返回 0。

下面给出完整的测试程序，程序清单 10-12。

程序清单 10-12

```

/*----- 10-12.c -----*/
#include "stdio.h"
#include "math.h"
func(int low,int high)
{
    int i;
    double x,y;
    for(i=low;i<=high;i++)
    {
        x=sqrt(i+100);           /*x 为加上 100 后开方后的结果*/
        y=sqrt(i+168);           /*y 为再加上 168 后开方后的结果*/
        if(x*x==i+100&&y*y==i+168) /*判断此数是否是完全平方数*/
            return i;
    }
}

```

```

    return 0;
}

main()
{
    int i;
    i = func(1,10000); /*在 0~10000 范围内寻找该数*/
    if(i)
        printf("%d add 100 or add 168 equal a perfect square number\n",i);
    else
        printf("There is no answer in the area of 1~10000\n");
    getch();
}

```

本程序的运行结果如图 10-14 所示。

```
156 add 100 or add 168 equal a perfect square number
```

图 10-14 程序 10-12 的运行结果

**【例 10-13】**编写算法，计算  $s=a+aa+aaa+aaaa+\cdots+aa...a$  的值，其中  $a$  是一个数字，该式共包含  $n$  项。

**【分析】**

本题可以通过一个循环操作完成。第  $i$  次操作在变量  $s$  上累加  $aa...a$ （由  $i$  个  $a$  组成），共循环  $n$  次。这样得到的  $s$  即为算式的结果。算法描述如下：

```

long Sum(int a,int n)
{
    int i;
    long s = 0,aa = 0;
    for(i=1;i<=n;i++)
    {
        aa = aa*10+a;
        s = s + aa;
    }
    return s;
}

```

函数 Sum()的功能是计算  $a+aa+aaa+\cdots+aa...a$  ( $n$  项) 的值，并将计算结果返回。变量  $aa$  是存放算式中每一项的值。每一次循环中，执行  $aa = aa*10+a$  语句，以计算每一项的值。变量  $s$  存放累加的结果。共循环  $n$  次，得到  $a+aa+aaa+\cdots+aa...a$  ( $n$  项) 的值，然后将计算结果返回。

下面给出完整的测试程序，程序清单 10-13。

程序清单 10-13

```

/*----- 10-13.c -----*/
#include "stdio.h"
long Sum(int a,int n)
{
    int i;
    long s = 0,aa = 0;
    for(i=1;i<=n;i++)
    {
        /*累加求和*/

```

```

        aa = aa*10+a;
        s = s + aa;
    }
    return s;                /*返回计算结果*/
}

main()
{
    printf("The result of 5+55+555+5555+55555 is %ld\n", Sum(5,5));
    getch();
}

```

本程序的运行结果如图 10-15 所示。

The result of 5+55+555+5555+55555 is 61725

图 10-15 程序 10-13 的运行结果

**【例 10-14】**设计一个递归算法将一个整数  $n$  转换成为字符串，例如将整数 123 转换为字符串“123”。整数  $n$  的位数不确定，可以是任意位数的整数。

**【分析】**

一般情况下要将一个整数转换为相应的字符串，可以将该数的每一位的数字提取出来，转换为相应的字符（ASCII 码），再拼凑在一起，打印出来即可。但是这里要使用递归的方法，因此可以这样考虑：先将高位的整数转换为字符串打印出来，再将个位数字转换为字符打印出来。“将高位的整数转换为字符串打印出来”的操作依然重复上述的过程。这样就构成了递归的结构。其代码描述如下：

```

void trans(int n)
{
    int i;
    i = n % 10;           /*得到整数 n 的个位数字*/
    if(n/10>0)
        trans(n/10);     /*递归调用 trans，将高位数转换为字符串*/
    printf("%c",i+48);
}

```

函数 trans() 为一个递归函数，它的功能是将整数  $n$  转换为字符串的形式输出。首先执行  $n\%10$  运算，并将结果赋值给  $i$ ，这样就得到了本次递归中参数  $n$  的个位数字。然后递归地调用函数 trans()，并将本次递归中参数  $n$  整除 10 后作为递归函数的参数进行传递。它的意思是将  $n$  的高位数转换为相应的字符串并输出。最后输出数字  $i$  对应的字符。在递归调用函数 trans() 的执行过程中，依然重复上述的操作。递归操作直到  $n/10 \leq 0$  为止，这样程序会自动控制结果的输出，并不需要提前知道整数  $n$  的位数。

下面给出完整的测试程序，程序清单 10-14。

程序清单 10-14

```

/*----- 10-14.c -----*/
#include "stdio.h"
void trans(int n)
{
    int i;

```

```

    i = n % 10;           /*得到整数n的个位数字*/
    if (n/10 > 0)
        trans(n/10);     /*递归调用trans, 将高位数转换为字符串*/
    printf("%c", i+48);
}

main()
{
    printf("The translation of 12345 from interger to string\n");
    trans(1235);
    getch();
}

```

本程序的运行结果如图 10-16 所示。

```

The translation of 12345 from interger to string
1235

```

图 10-16 程序 10-14 的运行结果

**【例 10-15】**在 C 语言的库函数中, 提供了将整型数字转换为相应的字符串的函数。即: `itoa()` 将整型数值转换为字符串。

现要求不使用上述提供的 C 语言库函数, 编写一个程序将输入的整数转换为相应的字符串形式。

#### 【分析】

可以想到, 只要将整数的每一位数字分离出来, 然后将其转换为对应的字符的 ASCII 码并存放到字符数组中就可以实现上述的功能。具体来说, 将输入的整数  $n$  反复进行整除 10 ( $/10$ )、模 10 ( $\%10$ ) 的运算即可得到该数的每一位的数字, 再将每一位的数字与 48 相加得到该数字对应的字符的 ASCII 码, 然后将它们存放在一个字符数组中。其算法描述如下:

```

void myitoa(int n, char *str)
{
    char a;
    int i=0;
    while(n!=0)
    {
        a = n % 10;           /*得到n的个位数字, 并存放到变量a中*/
        a = a + 48;          /*将变量a转换为字符ASCII码*/
        str[i] = a;          /*保存在字符数组str中*/
        i++;                 /*修改数组str的下标*/
        n = n / 10;
    }
    /*i 为字符串的长度*/
    reverse(str, i);         /*调用函数reverse()将字符串逆置*/
    str[i] = '\0';
}

```

该算法中函数 `myitoa()` 包含两个参数, 其中  $n$  为输入的整数 (应当小于等于 32767), `str` 为一个字符数组的头指针。函数 `myitoa()` 的作用是将整数  $n$  转换为相应的字符串的形式, 并将该字符串存放在字符数组 `str` 中, 因此这里要求 `str` 指向的字符数组的长度至少要比整数  $n$  的位数大 1 (字符串的末尾要添加字符串结束标志 `\0`)。

算法中通过一个循环获得整数  $n$  的每一位数字, 然后将其与 48 相加后存放到字符数组 `str` 中。每次循环中都执行  $n = n / 10$ ; 的操作, 这样可以通过  $a = n \% 10$ ; 的操作得到整数  $n$

的每一位的数字。当  $n$  等于 0 时循环结束。这样执行完 while 循环语句后, 数组 str 中存放的就是整数  $n$  的字符串表示的逆序形式。假设  $n$  的初值为 123, 表 10-1 中展示的是执行 while 循环语句的过程中各变量的情形。

表 10-1 执行while循环语句的过程中各变量的情形

循环次数	a	n	str[]
0	-	123	-
1	3	12	'3'
2	2	1	'3' '2'
3	1	0	'3' '2' '1'

从表 10-1 中可以看出, 循环结束后字符数组 str 中存放的字符串即为数  $n$  的字符串表示的逆序形式。因此还要再调用函数 reverse() 将字符串 str 的内容逆置。函数 reverse() 可描述如下:

```
void reverse(char *str,int i)
{
    int j=0;
    char tmp;
    while(j<i-j-1)
    {
        tmp = str[j];           /*交换数据*/
        str[j] = str[i-j-1];
        str[i-j-1] = tmp;
        j++;
    }
}
```

该算法中函数 reverse() 包含两个参数, 参数 str 为字符数组的头指针, 参数 i 为该字符串的长度。函数 reverse() 的功能是将字符串 str 中的内容逆置。这个工作是通过循环将数组 str 中的第  $j$  个元素同第  $i-j-1$  个元素不断地进行交换完成的。

值得注意的一点是, 最后还应将字符串 str 的末尾置上字符串结束标志“\0”, 这样才能构成一个标准的字符串结构。

下面给出完整的测试程序, 程序清单 10-15。

程序清单 10-15

```
/*----- 10-15.c -----*/
#include "stdio.h"

void reverse(char *str,int i)
{
    int j=0;
    char tmp;
    while(j<i-j-1)
    {
        tmp = str[j];
        str[j] = str[i-j-1];
        str[i-j-1] = tmp;
        j++;
    }
}
```

```

void myitoa(int n ,char *str)
{
    char a;
    int i=0;
    while(n!=0)
    {
        a = n % 10;
        a = a + 48;
        str[i] = a;
        i++;
        n = n / 10;
    }

    /*i 为字符串的长度*/
    reverse(str,i);
    /*调用函数 str 将字符串逆置*/
    str[i] = '\0';
}

main()
{
    int n = 1263;
    char str[5];
    myitoa(n,str);
    printf("The integer is %d\n",n);
    printf("The string of this integer is %s\n",str);
    /*打印出转换后的字符串 str*/
    getch();
}

```

本程序的运行结果如图 10-17 所示。

```

The integer is 1263
The string of this integer is 1263

```

图 10-17 程序 10-15 的运行结果

**【例 10-16】**在 C 语言的库函数中，提供了字符串复制函数 `strcpy()`，它可以实现字符串的复制。现要求不使用任何 C 语言库函数中提供的字符串相关的库函数，编程实现一个与 `strcpy` 功能相同的函数。

#### 【分析】

所谓字符串的复制，实际上就是将一个字符串中的内容复制到另一个字符数组空间中，再向该字符数组的末尾添加一个字符串结束标志`\0`。因此编写的函数应当至少包含两个参数，一个参数是源字符串（要被复制的字符串）的首地址，一个参数是目的字符数组（存放字符串副本的内存空间）的首地址。具体算法描述如下：

```

void mystrcpy(char *src,char *dst)
{
    int i=0;
    while(src[i] != '\0')
    {
        dst[i] = src[i];
        i++;
    }
    dst[i] = '\0';
    /*复制字符串*/
    /*指针后移*/
    /*目的字符串的末尾要添加字符串结束标志'\0'*/
}

```



该算法中，函数 `mystrcpy()` 包含两个参数。参数 `src` 为源字符串的首地址，参数 `dst` 为目的字符数组的首地址。函数 `mystrcpy()` 的功能是将字符串 `src` 复制到 `dst` 指向的内存空间中。因此在调用函数 `mystrcpy()` 进行字符串复制时，要求字符数组 `dst` 的长度至少等于字符串 `src` 的长度加 1（字符串的末尾要添加字符串结束标志）。

下面给出完整的测试程序，程序清单 10-16。

程序清单 10-16

```

/*----- 10-16.c -----*/
#include "stdio.h"

void mystrcpy(char *src, char *dst)
{
    int i=0;
    while(src[i] != '\0')
    {
        dst[i] = src[i];
        i++;
    }
    dst[i] = '\0';
}

main()
{
    char dst[14], *src="hello world!\n"; /*设置源字符串和目的字符串*/
    printf("src: %s", src);             /*打印出源字符串 src 中的内容*/
    mystrcpy(src, dst);                 /*调用函数 mystrcpy() 进行字符串复制*/
    printf("dst: %s", dst);             /*打印出目的字符串 dst 中的内容*/
    getche();
}

```

本程序的运行结果如图 10-18 所示。

**【例 10-17】**编写一个函数 `loopMove(char *str, int n)` 实现字符串的循环右移功能。例如字符串 `str="abcde"`，调用函数 `loopMove(str, 3)`，可将字符串 `str` 循环右移 3 位，变为 `cdeab`。

```

src: hello world!
dst: hello world!

```

图 10-18 程序 10-16 的运行结果

#### 【分析】

要实现字符串的循环右移  $n$  位，可以通过一个循环实现。每次从字符串的尾部取出一个字符存放到一个临时变量中，再将字符串中前面的所有字符（第一个字符至倒数第二个字符）右移一位，最后将临时变量中存放的字符复制到字符串的首部，如此循环  $n$  次。其算法描述如下：

```

void loopMove(char *str, int n)
{
    int i, j, strLength;
    char tmp;
    strLength = strlen(str);
    for(i=0; i<n; i++)
    {
        tmp = str[strLength-1];          /*取出字符串的最后一个元素*/
        for(j=0; j<strLength-1; j++)

```

```

    {
        str[strLength-j-1] = str[strLength-j-2];    /*前面的元素后移*/
    }
    str[0] = tmp;    /*将原字符串尾部的元素放在字符串首部*/
}
}

```

在本算法中，函数 loopMove() 包含两个参数，参数 str 为要进行移位的字符串的首地址，参数 n 为要移动的位数。

首先应用库函数 strlen() 计算出字符串 str 的长度，并将该值存放在变量 strLength 中。

然后通过一个二重循环实施循环右移操作。

外层循环控制循环右移的次数，循环的次数由参数 n 来指定。每次循环执行 3 步操作：

- (1) 取出字符串的最后一个元素，并将其存放到临时变量 tmp 中，字符串的最后一个元素通过 str[strLength-1] 来引用。
- (2) 将字符串前面的元素全部后移一位。
- (3) 将原字符串尾部的元素放在字符串首部，实现一次循环右移操作。

内层循环主要负责将字符串中前面的所有字符（第一个字符至倒数第二个字符）右移一位。内层循环需要移动 strLength-1 个字符元素，因此循环 strLength-1 次。

下面给出完整的测试程序，程序清单 10-17。

程序清单 10-17

```

/*----- 10-17.c -----*/
#include "stdio.h"

void loopMove(char *str, int n)
{
    int i,j,strLength;
    char tmp;
    strLength = strlen(str);
    for(i=0;i<n;i++)
    {
        tmp = str[strLength-1];    /*取出字符串的最后一个元素*/
        for(j=0;j<strLength-1;j++)
        {
            str[strLength-j-1] = str[strLength-j-2];    /*前面的元素后移*/
        }
        str[0] = tmp;    /*将原字符串尾部的元素放在字符串首部*/
    }
}

main()
{
    char *str="abcdefghijklmn";    /*初始化字符串 str*/
    printf("The original string is %s\n",str);    /*打印出该字符串*/
    loopMove(str,6);    /*循环右移 6 位*/
    printf("The string be loopmoved is %s\n",str);    /*打印出循环右移后的字符串 str*/
    getch();
}

```

本程序的运行结果如图 10-19 所示。

```

The original string is abcdefghijklmn
The string be loopmoved is ijklmnabcdefgh

```

图 10-19 程序 10-17 的运行结果

【例 10-18】找出 01 字符串中 0 和 1 连续出现的最大次数。例如 01 字符串“111001111110000”中 0 连续出现的最大次数为 4，1 出现的最大次数为 6。

【分析】

可以设置两个变量 max0 和 max1 分别存放字符 0 和 1 连续出现的最大次数，它们的初始值可设定为 0。在扫描整个 0/1 字符串时，每当扫描到一组连续的字符 0 或者字符 1 时，都记录下它们连续出现的次数，然后再分别与 max0 和 max1 进行比较，将较大的值替换给 max0 或者 max1。这样就保证了 max0 中始终存放的是当前已扫描过的 0/1 字符串中连续出现 0 的最大次数，保证了 max1 中始终存放的是当前已扫描过的 0/1 字符串中连续出现 1 的最大次数。这样通过扫描一次该 0/1 字符串就可以计算出字符串中 0 和 1 连续出现的最大次数。其算法描述如下：

```
void getMax(char *str,int *max0,int *max1)
{
    int i,len,tmp_max0 = 0, tmp_max1 = 0;
    len = strlen(str);
    for(i=0;i<len;i++)
    {
        if(str[i]=='0')
        {
            if(str[i-1] == '1')           /*如果是字符串的 1-0 转换点*/
            {
                if(tmp_max1>*max1)       /*判断是否需要修改 max1 的值*/
                    *max1 = tmp_max1;
                tmp_max1 = 0;           /*临时变量 tmp_max1 清零*/
            }
            tmp_max0++;                 /*变量 tmp_max0 自增 1，记录 0 的次数*/
        }
        if(str[i]=='1')                 /*如果是字符串的 0-1 转换点*/
        {
            if(str[i-1] == '0')         /*判断是否需要修改 max0 的值*/
            {
                if(tmp_max0>*max0)
                    *max0 = tmp_max0;
                tmp_max0 = 0;           /*临时变量 tmp_max0 清零*/
            }
            tmp_max1++;                 /*变量 tmp_max1 自增 1，记录 1 的次数*/
        }
    }
    if(tmp_max1>*max1)                  /*补充的比较*/
        *max1 = tmp_max1;
    if(tmp_max0>*max0)
        *max0 = tmp_max0;
}
```

在本算法中，函数 getMax() 包含 3 个参数，参数 str 为 0/1 字符串的首地址，参数 max0 和 max1 分别为记录字符 0 和字符 1 连续出现的最大次数的两个变量的指针。因为要在函数中修改变量 max0 和 max1 的值，所以这里用指针传递。

首先应用函数 strlen() 计算出 0/1 字符串 str 的长度，然后通过一个 for 循环遍历整个字符串，找出字符串中 0 和 1 连续出现的最大次数。这个过程如下：

当扫描到的当前字符为 0 时，需要做以下工作：

(1) 首先判断它的前一个字符是否为 1, 可以称它为字符串的“1-0 转换点”。如果是字符串的 1-0 转换点, 则要判断刚才记录下的连续的 1 的个数 `tmp_max1` 是否大于 `max1` 的值。如果大于 `max1` 的值, 则要将 `tmp_max1` 赋值给 `max1`。然后将变量 `tmp_max1` 清零。

(2) 将记录字符串中连续的 0 的个数变量的 `tmp_max0` 自加 1。

当扫描到的当前字符为 1 时, 需要做以下工作:

(1) 首先判断它的前一个字符是否为 0, 可以称它为字符串的“0-1 转换点”。如果是字符串的 0-1 转换点, 则要判断刚才记录下的连续的 0 的个数 `tmp_max0` 是否大于 `max0` 的值。如果大于 `max0` 的值, 则要将 `tmp_max0` 赋值给 `max0`。然后将变量 `tmp_max0` 清零。

(2) 将记录字符串中连续的 1 的个数变量的 `tmp_max1` 自加 1。

如此循环至扫描完整个 0/1 字符串。

由于只有在字符串的 1-0 转换点或者 0-1 转换点时, 程序才会判断临时记录下的连续 1 的次数 `tmp_max1` 或者连续 0 的次数 `tmp_max0` 是否大于 `max1` 或者 `max0`, 因此扫描完整个字符串时会少一次比较, 所以最后还要补充比较一次。这样 `max1` 的值就一定是字符串中连续出现字符 1 的最大次数, `max0` 的值一定是字符串中连续出现字符 0 的最大次数。

为了更加清楚地展示这个扫描字符串操作全过程, 请参看表 10-2, 假设字符串 `str` 的初值为“1010011”, `max0` 和 `max1` 的初值为 0。

表 10-2 扫描 0/1 字符串过程中各变量的状态变化

扫描的字符	Tmp_max0	max0	tmp_max1	max1
<u>1</u> 010011	0 (无操作)	0 (无操作)	1 (自增 1)	0 (无操作)
1 <u>0</u> 10011	1 (自增 1)	0 (无操作)	0 (清零)	1 (修改 max1)
10 <u>1</u> 0011	0 (清零)	1 (修改 max0)	1 (自增 1)	0 (无操作)
101 <u>0</u> 011	1 (自增 1)	1 (无操作)	0 (清零)	1 (不修改)
1010 <u>0</u> 11	2 (自增 1)	1 (无操作)	0 (无操作)	1 (无操作)
10100 <u>1</u> 1	0 (清零)	2 (修改 max0)	1 (自增 1)	1 (无操作)
101001 <u>1</u>	0 (无操作)	2 (无操作)	2 (自增 1)	1 (无操作)
最后补充比较	因为 <code>tmp_max1 &gt; max1</code> , 所以 <code>max1=2</code> (修改 max1)			

如表 10-2 所示, 通过一次扫描 0/1 字符串的操作, 可以得出字符串“1010011”中连续出现字符 0 的最大次数为 2, 连续出现字符 1 的最大次数为 2。

下面给出完整的测试程序, 程序清单 10-18。

程序清单 10-18

```

/*----- 10-18.c -----*/
#include "stdio.h"

void getMax(char *str, int *max0, int *max1)
{
    int i, len, tmp_max0 = 0, tmp_max1 = 0;
    len = strlen(str);
    for(i=0; i<len; i++)
    {
        if(str[i]=='0')
        {

```

```

        if(str[i-1] == '1')           /*如果是字符串的1-0转换点*/
        {
            if(tmp_max1>*max1)        /*判断是否需要修改max1的值*/
                *max1 = tmp_max1;
            tmp_max1 = 0;             /*临时变量 tmp_max1 清零*/
        }
        tmp_max0++;                  /*变量 tmp_max0 自增 1, 记录 0 的次数*/
    }
    if(str[i]=='1')                  /*如果是字符串的0-1转换点*/
    {
        if(str[i-1] == '0')          /*判断是否需要修改max0的值*/
        {
            if(tmp_max0>*max0)
                *max0 = tmp_max0;
            tmp_max0 = 0;             /*临时变量 tmp_max0 清零*/
        }
        tmp_max1++;                  /*变量 tmp_max1 自增 1, 记录 1 的次数*/
    }
}
if(tmp_max1>*max1)                  /*补充的比较*/
    *max1 = tmp_max1;
if(tmp_max0>*max0)
    *max0 = tmp_max0;
}

main()
{
    char *str="1010000000000111000000000000"; /*初始化字符串*/
    int max0 = 0, max1 = 0;
    getMax(str,&max0,&max1);          /*找出 0 和 1 连续出现的最大次数*/
    printf("\n%s\n",str);
    printf("The number of consecutive character '0'are %d\n",max0);
    printf("The number of consecutive character '1'are %d\n",max1);
    getch();
}

```

本程序的运行结果如图 10-20 所示。

```

10100000000001110000000000
The number of consecutive character '0'are 11
The number of consecutive character '1'are 3

```

图 10-20 程序 10-18 的运行结果

**【例 10-19】**有这样一个数列 1, 3, 6, 13, 26, 53……, 按照这个规律, 编程计算该数列第 20 项的值。

#### 【分析】

要解决这类问题, 首先要找到数列的排布规律。有些数列可以比较容易地总结出它的通项公式, 那样再求解数列的第  $n$  项的值就变得非常容易了。但是本题中的数列并不能很容易地找出它的通项公式, 因此要仔细分析数列每一项的值, 以找出其排布规律。通过仔细分析该数列的前几项的值不难发现, 该数列中奇数项的值(除第一项外)为它前一项的值的 2 倍, 偶数项的值为它前一项的值的 2 倍加 1, 第一项的值为 1。因此可以总结出该数

列每一项值的递归通项公式：

$$f(n) = \begin{cases} 1 & n=1 \\ 2f(n-1)+1 & n \text{ 为偶数} \\ 2f(n-1) & n \text{ 为奇数} \end{cases}$$

这样就不难设计出计算该数列任意项元素值的递归算法。

```
unsigned long func(int n)
{
    if(n==1)
        return 1;
    if(n % 2 == 0)
        return 2*func(n-1)+1;
    if(n % 2 != 0)
        return 2*func(n-1);
}
```

这个算法与上述的递归函数是一一对应的。值得注意的一点是，题目中要求计算该数列的第 20 项的值。从该数列各项元素的变化趋势来看，元素的递增速度超过几何级数增加，所以第 20 项的值应当很大（大约 100 万左右），因此函数 func() 的返回值类型应当尽量设置得大一些。

下面给出完整的测试程序，程序清单 10-19。

程序清单 10-19

```
/*----- 10-19.c -----*/
#include "stdio.h"

unsigned long func(int n)
{
    if(n==1)
        return 1;
    if(n % 2 == 0)
        return 2*func(n-1)+1;
    if(n % 2 != 0)
        return 2*func(n-1);
}

main()
{
    printf("The 10th item of the sequence is %ld",func(20));
    /*直接输出结果*/
    getche();
}
```

本程序的运行结果如图 10-21 所示。

The 10th item of the sequence is 873813

图 10-21 程序 10-19 的运行结果

**【例 10-20】**集邮爱好者把所有的邮票存放在 3 个集邮册中，在 A 册内存放全部的 2/10，在 B 册内存放全部的七分之几，在 C 册内存放 303 张邮票，问这位集邮爱好者集邮总数是多少？以及每册中各有多少邮票？

## 【分析】

假设该集邮爱好者拥有邮票总数为  $x$  张，那么通过上述的已知条件可得到下列方程：

$$\frac{2}{10}x + \frac{y}{7}x + 303 = x$$

很显然，该方程的解有无数多组，因为该方程包含两个未知量。但是作为本题， $x$  和  $y$  的取值都是有其范围规定的，因此它的解可能不会那么多。

可以用穷举法求解该方程中的未知量  $x$  和  $y$ ，但是关键的问题在于如何确定搜索的解空间。 $y$  的取值范围是明确的，它限定在  $[1, 6]$  之间的整数上。即  $y$  的取值只能是 1, 2, 3, 4, 5, 6 其中之一。而  $x$  的取值范围是多少呢？这个问题很难回答，因为题目中并没有限定邮票的总数。可以把上述方程式做如下转换：

$$\frac{1}{5}x + \frac{y}{7}x - x = -303$$

$$(\frac{1}{5} + \frac{y}{7} - 1)x = -303$$

$$(\frac{4}{5} - \frac{y}{7})x = 303$$

$$x = 303 \frac{35}{28 - 5y}$$

这样问题就简化了许多。因为  $x$  的取值依赖于  $y$  的取值，而  $y$  的取值只能是 1, 2, 3, 4, 5, 6 其中之一。这样  $y$  的取值一旦确定，得到的  $x$  也就是唯一确定的。而只有  $x$  和  $y$  的取值都为正整数时才是本题的答案。本题算法可描述如下：

```
func()
{
    float y,x;
    for(y=1;y<=6;y++)
    {
        x = 303*35/(28-5*y);
        if(isint(x))          /*如果 x 是整数*/
        {                      /*输出答案*/
            printf("The amount of stamps are %d\n", (int)x);
            printf("The stamp album A: %d\n", (int)x/5);
            printf("The stamp album B: %d\n", (int)x*(int)y/7);
            printf("The stamp album C: %d\n", 303);
        }
    }
}
```

本算法通过一个循环，枚举  $y$  的取值为 1, 2, 3, 4, 5, 6 的情况，得出对应的  $x$  的值。如果  $x$  的值为一个正整数（即调用函数 `isint(x)` 的返回值为 1），则输出  $x$  的值以及对应的答案。通过该算法可得出，当  $y$  的取值为 5 时， $x$  的值为 3535，即邮票的总数为 3535 张。其中，第 A 册中存放邮票 707 张，第 B 册中存放邮票 2525 张，第 C 册中存放邮票 303 张。

下面给出完整的测试程序，程序清单 10-20。

程序清单 10-20

```
/*----- 10-20.c -----*/
```

```

#include "stdio.h"

int isint(float x)          /*判断 x 是否是整数, 如果是返回 1, 否则返回 0*/
{
    if(x-(int) x == 0)
        return 1;
    else
        return 0;
}

func()
{
    float y,x;
    for(y=1;y<=6;y++)
    {
        x = 303*35/(28-5*y);
        if(isint(x))          /*如果 x 是整数*/
        {
            printf("The amount of stamps are %d\n", (int)x);
            printf("The stamp album 1: %d\n", (int)x/5);
            printf("The stamp album 2: %d\n", (int)x*(int)y/7);
            printf("The stamp album 3: %d\n", 303);
        }
    }
}

main()
{
    func();
    getch();
}

```

本程序的运行结果如图 10-22 所示。

**【例 10-21】** 问 5 个人的年龄各是多少, 第五个人说比第四个人大 2 岁, 第四个人说比第三个人大 2 岁, 第三个人说比第二个人大 2 岁, 第二个人说比第一个人大 2 岁, 最后第一个人说自己 10 岁。编写一个递归算法, 计算出每个人的年龄。

#### 【分析】

很显然 5 个人对自己年龄的描述是一种递归的描述方法。要想知道第五个人的年龄, 必须知道第四个人的年龄, 要想知道第四个人的年龄, 必须知道第三个人的年龄, 要想知道第三个人的年龄, 必须知道第二个人的年龄, 而第二个人的年龄取决于第一个人的年龄。这样可以形式化地构成一个计算每个人年龄的递归函数:

$$\text{Age}(n) = \begin{cases} 10 & n=1 \\ \text{Age}(n-1)+2 & n>1 \end{cases}$$

其中  $\text{Age}(n)$  表示第  $n$  个人的年龄。这样就可以比较容易地编写出此递归算法。

```

int getAge(int n)
{
    if(n == 1) return 10;          /*第一个人 10 岁*/
    else return getAge(n-1)+2;      /*第 n 个人的年龄比第 n-1 人的年龄大 2 岁*/
}

```

上述算法与上面的递归函数一一对应。函数  $\text{getAge}()$  返回第  $n$  个人的年龄。

```

The amount of stamps are 3535
The stamp album 1: 707
The stamp album 2: 2525
The stamp album 3: 303

```

图 10-22 程序 10-20 的运行结果



下面给出完整的测试程序，程序清单 10-21。

程序清单 10-21

```

/*----- 10-21.c -----*/
#include "stdio.h"

int getAge(int n)
{
    if(n == 1) return 10;           /*第一个人 10 岁*/
    else return getAge(n-1)+2;      /*第 n 个人的年龄比第 n-1 人的年龄大 2 岁*/
}

main()
{
    printf("Age of the first person is %d\n",getAge(1));
    printf("Age of the second person is %d\n",getAge(2));
    printf("Age of the third person is %d\n",getAge(3));
    printf("Age of the forth person is %d\n",getAge(4));
    printf("Age of the fifth person is %d\n",getAge(5));
    getch();
}

```

本程序的运行结果如图 10-23 所示。

**【例 10-22】**输入两个数组（数组元素个数自定），输出在两个数组中都出现的元素，即两数组的子集。（例如输入  $a[5]=\{2,3,4,5,6\}$ ， $b[6]=\{3,5,7,9,10,-1\}$ ，则输出 3、5）。

```

Age of the first person is 10
Age of the second person is 12
Age of the third person is 14
Age of the forth person is 16
Age of the fifth person is 18

```

图 10-23 程序 10-21 的运行结果

#### 【分析】

一种最简单直观的办法就是将两个数组中的元素进行两两比较。如果第一个数组中包含  $m$  个元素，第二个数组中包含  $n$  个元素，则最多需要比较  $m*n$  次。一旦发现第一个数组中的第  $i$  个元素与第二个数组中的第  $j$  个元素相等，就将该数组元素输出，第二个数组中的后续元素可以不再继续比较下去。这个比较的过程可以通过一个二重循环实现。

```

void overlapNumber(int array1[],int n1,int array2[],int n2)
{
    int i,j;
    for(i=0;i<n1;i++)
        for(j=0;j<n2;j++)
        {
            if(array1[i] == array2[j])           /*找到相等的元素*/
            {
                printf("%d ",array1[i]);        /*输出该元素*/
                break;                          /*跳出内层循环*/
            }
        }
}

```

函数 `overlapNumber()` 包含 4 个参数，参数 `array1[]` 和 `array2[]` 为输入的两个数组的首地址，参数 `n1` 为数组 `array1` 的长度，参数 `n2` 为数组 `array2` 的长度。本算法通过一个二重循环将数组 `array1` 和数组 `array2` 中的每一个元素进行比较。外层循环控制第一个数组 `array1` 的元素遍历，内层循环控制第二个数组 `array2` 的元素遍历。当发现数组 `array1` 和数组 `array2`

中有相同的元素时就将其输出，并跳出内层循环，也就是说第二个数组中的后续元素可以不再继续比较。

下面给出完整的测试程序，程序清单 10-22。

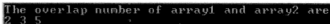
程序清单 10-22

```
/*----- 10-22.c -----*/
#include "stdio.h"

void overlapNumber(int array1[],int n1,int array2[],int n2)
{
    int i,j;
    for(i=0;i<n1;i++)
        for(j=0;j<n2;j++)
        {
            if(array1[i] == array2[j])
            {
                printf("%d ",array1[i]);
                break;
            }
        }
}

main()
{
    int array1[]={2,3,4,5,6};          /*初始化数组 array1*/
    int array2[]={3,5,7,2,10,-1};      /*初始化数组 array2*/
    printf("The overlap number of array1 and array2 are\n");
    overlapNumber(array1,5,array2,6);  /*找出 array1 和 array2 中的相同元素*/
    getch();
}
```

本程序的运行结果如图 10-24 所示。



```
The overlap number of array1 and array2 are
2 3 5
```

图 10-24 程序 10-22 的运行结果

**【例 10-23】**输入两个有序的数组（元素从小到大排列，数组元素个数自定），输出在两个数组中都出现的元素，即两数组的子集。（例如输入  $a[5]=\{2,3,4,5,6\}$ ， $b[6]=\{3,5,7,9,10\}$ ，则输出 3、5）。

#### 【分析】

本题与上一题很类似，不同之处在于本题中输入的两个数组是按值有序的。如果依然采用上一题中使用的元素值两两比较的方法寻找相同的元素固然可以，但是这样就没有充分利用题目中给出的“有序数组”的条件。我们可以充分利用两个数组是按值有序的条件，设计出更加高效的算法。

假设输入的两个数组为  $array1$  和  $array2$ ，用两个变量  $i$  和  $j$  记录数组的下标， $i$  和  $j$  的初值均为 0。

首先比较  $array1[i]$  和  $array2[j]$  是否相等，如果相等则输出该值，同时执行  $i++$  和  $j++$  操作；如果  $array1[i]>array2[j]$ ，则执行  $j++$  操作。如果  $array1[i]<array2[j]$ ，则执行  $i++$  操作。

重复上述的操作,直到其中一个数组的元素全部比较完为止(如果两个数组不等长)。利用这种方法可以只进行  $\min(m,n)$  次的比较就可以找出两数组中共同的元素,其中  $m$  为数组 `array1` 中元素的个数,  $n$  为数组 `array2` 中元素的个数,其时间复杂度为  $O(n)$ 。但是如果采用上一题中使用的元素值两两比较的方法寻找相同的元素,则大约需要比较  $m*n$  次,其时间复杂度为  $O(n^2)$ 。该算法描述如下:

```
void overlapNumber(int array1[],int n1,int array2[],int n2)
{
    int i = 0 , j = 0;
    while(i<n1 && j<n2)
    {
        if(array1[i]>array2[j]) j++; /*array1[i]>array2[j], 执行j++操作*/
        else if(array1[i]<array2[j]) i++;
        /*array1[i]<array2[j], 执行i++操作*/
        else if(array1[i] == array2[j])
        {
            printf("%d \n",array1[i]); /*找到相同元素, 将其输出*/
            i++; /*执行i++ 操作*/
            j++; /*执行j++ 操作*/
        }
    }
}
```

函数 `overlapNumber()` 包含 4 个参数,参数 `array1[]` 和 `array2[]` 为输入的两个数组的首地址,参数 `n1` 为数组 `array1` 的长度,参数 `n2` 为数组 `array2` 的长度。该算法通过一个 `while` 循环实现,当  $i < n1$  并且  $j < n2$  时,执行前面所述的各种比较操作,一旦  $i$  或  $j$  的值大于等于  $n1$  或  $n2$ ,则说明其中一个数组的元素全部比较完,因此循环结束。

下面给出完整的测试程序,程序清单 10-23。

程序清单 10-23

```
/*----- 10-23.c -----*/
#include "stdio.h"

void overlapNumber(int array1[],int n1,int array2[],int n2)
{
    int i = 0 , j = 0;
    while(i<n1 && j<n2)
    {
        if(array1[i]>array2[j]) j++;
        else if(array1[i]<array2[j]) i++;
        else if(array1[i] == array2[j])
        {
            printf("%d ",array1[i]);
            i++;
            j++;
        }
    }
}

main()
{
    int array1[]={1,2,3,4,5,6,8}; /*初始化数组 array1*/
```

```
int array2[]={3,5,6,7,8};          /*初始化数组 array2*/
printf("The overlap number of array1 and array2 are\n");
overlapNumber(array1,7,array2,5); /*找出 array1 和 array2 中的相同元素*/
}
```

本程序的运行结果如图 10-25 所示。

```
The overlap number of array1 and array2 are
3 5 6 8
```

图 10-25 程序 10-23 的运行结果

**【例 10-24】**编写一个算法，计算输入的一个字符串中所包含的单词的个数。

**【分析】**

要计算一个字符串中单词的个数，首先要弄清字符串中单词的特点。所谓一个单词，可以简单地理解为一组连续的字符，这些字符的 ASCII 码都处在字符 A~Z 和字符 a~z 之间（即字母字符）。同时，一个单词的前面一个字符和后面一个字符都不能是字母字符（它们可能是空格符、标点符号字符等）。因此要判断一组连续的字符是否是一个单词，要紧扣住这两点。不难理解，只要从第一个字母字符开始顺序向下遍历，当遍历到第一个非字母字符时就可以判定以上遍历的一组字符构成了一个单词，该过程如图 10-26 所示。

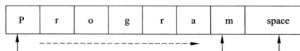


图 10-26 查找到一个单词

如图 10-26 所示，从第一个字母字符 P 开始顺序向下遍历，直到遍历到字母字符 m。当遍历到第一非字母字符 space 时表明以上遍历的一组字符恰好构成一个单词 Program。因此可以用这个办法高效地统计出一个字符串中的单词个数。其算法描述如下：

```
int latterNumber(char *str)
{
    int i,count=0,flag ;
    char *p = str;
    while(*p != '\0')
    {
        if((*p>='A'&&*p<='Z')||(*p>='a'&&*p<='z')) /*是字母*/
            flag = 1; /*将标记变量 flag 置 1*/
        else /*出现非字母字符*/
        {
            flag++; /*flag 自增 1*/
        }
        if(flag == 2) /*第一次出现非字母字符*/
            count++; /*单词统计变量 count 自增 1*/
        p++; /*指针后移*/
    }
    return count; /*返回字符串中单词的个数*/
}
```

函数 latterNumber()的作用是统计字符串 str 中的单词个数并将其返回。在该算法的设计中，巧妙地使用了一个标记变量 flag。一旦扫描到的字符为字母字符，就将 flag 置 1，

而出现非字母字符时, flag 就作自增 1 操作。这样当变量 flag 的值为 2 时, 就表明刚刚扫描到了一个单词, 此时第一次出现了非字母字符。于是将单词的统计变量 count 作自增 1 操作。最后将 count 的值返回即可。

细心的读者可能会提出这样的问题: 为什么当出现非字母字符时, 要将变量 flag 作自增 1 操作呢? 为什么不能直接赋一个常量值呢, 例如 2。这是因为无法确定输入的字符串中是否包含了多余的空格符或其他的非字母字符, 因此要判断出已经找到了一个单词, 必须以第一个非字母字符作为判断的依据。例如有这样一个字符串 This--is--a-test!, 其中-表示一个空格符, 如果将 flag 直接赋一个常量值, 则程序实际上统计的是字符串中非字母字符的个数, 而不是单词的个数。因此输出的结果为 7(字符串中包含 6 个空格符和一个!)。而如果按照上述的算法, 作 flag 的自增 1 操作, 则 flag 等于 2 实际上表示的是变量 flag 由 1 变为 2, 即第一次出现了非字母字符, 也就是扫描到一个单词。因此输出的结果为 4。

下面给出完整的测试程序, 程序清单 10-24。

程序清单 10-24

```

/*----- 10-24.c -----*/
#include "stdio.h"

int latterNumber(char *str)
{
    int i, count=0, flag ;
    char *p = str;
    while(*p != '\0')
    {
        if ((*p>='A' && *p<='Z') || (*p>='a' && *p<='z')) /*是字母*/
            flag = 1; /*将标记变量 flag 置 1*/
        else /*出现非字母字符*/
        {
            flag++; /*flag 自增 1*/
        }
        if (flag == 2) /*第一次出现非字母字符*/
            count++; /*单词统计变量 count 自增 1*/
        p++; /*指针后移*/
    }
    return count; /*返回字符串中单词的个数*/
}

main()
{
    char str[] = "This is a test!! "; /*初始化字符串*/
    printf("\n%s\n", str); /*显示原字符串*/
    printf("This string contains %d latter.\n", latterNumber(str)); /*输出其单词个数*/
    getch();
}

```

本程序的运行结果如图 10-27 所示。

```

This is a test!!
This string contains 4 latter.

```

图 10-27 程序 10-24 的运行结果

**【例 10-25】**编写一个程序，删除一个字符串中指定的字符。例如原字符串为 abcde，删除字符 b 后得到的字符串为 acde。

**【分析】**

一种比较简单的方法是从字符串中第 1 个字符开始向后扫描。当发现当前的字符与要删除的字符相同时，就将字符串中的后续字符全部前移一个字节，覆盖掉这个待删除的字符。然后继续扫描下去，直到遇到字符串结束标志“\0”为止。下面给出具体的算法描述。

```
void delChar(char *str, char c)
{
    char *q, *p = str;
    while(*p != '\0')
    {
        if(*p == c)                /*找到了要删除的字符*/
        {
            q = p;
            do{                    /*将后续字符串向前移动一个字节*/
                *q = *(q+1);
                q++;                /*指针后移*/
            } while(*q != '\0') ;
            p++;                    /*指针后移*/
        }
    }
}
```

在本算法中，函数 delChar() 包含两个参数，参数 str 为字符串的首地址，参数 c 为指定的要删除的字符。算法通过二重循环实现删除字符串中的指定字符的功能。外层循环负责扫描整个字符串，直到 \*p 的内容为“\0”为止。内层循环负责将后续字符串向前移动一个字节。当指针 p 指向的字符等于用户指定的要删除的字符 c 时，通过一个 do-while 语句将后续的字符串前移一个字节，从而覆盖掉要删除的字符。整个操作通过两个指针变量 p 和 q 来完成。

下面给出完整的测试程序，程序清单 10-25。

程序清单 10-25

```
/*----- 10-25.c -----*/
#include "stdio.h"
void delChar(char *str, char c)
{
    char *q, *p = str;
    while(*p != '\0')
    {
        if(*p == c)                /*找到了要删除的字符*/
        {
            q = p;
            do{                    /*将后续字符串向前移动一个字节*/
                *q = *(q+1);
                q++;                /*指针后移*/
            } while(*q != '\0') ;
        }
    }
}
```

```

        p++;                                /*指针后移*/
    }
}

main()
{
    char str[] = "abcdefdhidkldn";          /*初始化字符串*/
    char c;
    printf("Input the charactor for deleting\n");
    scanf("%c",&c);                          /*用户指定要删除的字符*/
    printf("The string before deleting: %s\n",str); /*输出原字符串的内容*/
    delChar(str,c);                          /*删除指定的字符*/
    printf("The string after deleting: %s\n",str); /*输出处理后的字符串*/
    getch();
}

```

本程序的运行结果如图 10-28 所示。

```

Input the charactor for deleting
d
The string before deleting: abcdefdhidkldn
The string after deleting: abcefhikln

```

图 10-28 程序 10-25 的运行结果

**【例 10-26】**计算一个字节中有多少位被置为 1。

**【分析】**

本题考查的内容是位运算的编程。要判断一串二进制位 (bits) 中的某一位是否是 1，只需将其与一个常数做按位与 (&) 操作。在该常数中，要判断的位设置为 1，其他位均设置为 0。若按位与的结果为 0，则表明该二进制串中的相应位为 0，否则表明该二进制串中的相应位为 1。例如要判断二进制串 “1101010001” 的第一位是否为 1，只需将它与常数 “100000000” 进行按位与运算，如果结果为 0，则表明二进制串 “1101010001” 第一位为 0，否则表明二进制串 “1101010001” 第一位为 1。本例的结果不为 0，表明该二进制串的第一位为 1。

根据这个思想，就不难设计出计算一个字节中二进制为 1 的个数的算法。

```

int bitNumber(unsigned char c)
{
    int count = 0;
    int i;
    unsigned char cmp = (0x1<<7);
    for(i=0;i<8;i++)
    {
        if((c & cmp)!=0)                /*结果不为 0*/
            count++;                    /*变量 count 记录字符串中'1'的个数*/
        cmp = cmp>>1;                  /*常数的'1'位向右移动 1 位*/
    }
    return count;
}

```

在本算法中，函数 bitNumber() 的功能是计算一个字节 (字符) c 中二进制位 1 的个数，并将计算的结果返回。变量 cmp 的初值设置为 0x1<<7，对应的二进制表示为 10000000。前面已经讲过，应用这个数可以判断一个字节的第 1 位是否为 1。然后通过一个循环将 cmp 的 1 位向后移动 8 次，从而可以判断出字符 c 的每一位是否为 1。如果某一位被置为 1，则

变量 count 自增 1。这样变量 count 最终记录下字节（字符）c 中 1 位的个数。

下面给出完整的测试程序，程序清单 10-26。

程序清单 10-26

```

/*----- 10-26.c -----*/
#include "stdio.h"

int bitNumber(unsigned char c)
{
    int count = 0;
    int i;
    unsigned char cmp = (0x1<<7);
    for(i=0;i<8;i++)
    {
        if((c & cmp)!=0)        /*向与结果不为 0*/
            count++;            /*变量 count 记录字符中 1 的个数*/
        cmp = cmp>>1;           /*常数的 1 位向右移动 1 位*/
    }
    return count;
}

main()
{
    unsigned char c;
    printf("Please input a character\n");    /*输入 1 个字符*/
    scanf("%c",&c);
    printf("The number of bit '1' in the character are %d\n",bitNumber(c));
    /*输出该字符中 1 位的个数*/
    getch();
}

```

本程序的运行结果如图 10-29 所示。

```

Please input a character
a
The number of bit '1' in the character are 3

```

图 10-29 程序 10-26 的运行结果

**说明：**本题中输入的字符为 a，其 ASCII 码为 97，对应的二进制表示为 1100001，共包含 3 个 1。

**【例 10-27】**阅读下列函数，说明函数实现的功能。

```

void conv(int b)
{
    if(b>=2) conv(b/2);
    printf("%d",b%2);
}

```

**【分析】**

本题考查的知识点是递归函数的应用。分析该函数可知，当输入的参数  $b \geq 2$  时，递归地调用函数 conv() 本身，并将  $b/2$  作为下一层递归的参数进行传递，直到  $b < 2$  为止递归结束。然后再逐层返回，并打印出该层函数调用中  $b\%2$  的值。例如最开始调用 conv 时，参数 b 等于 4，这个执行过程如图 10-30 所示。



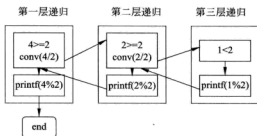


图 10-30 conv(4)的执行过程

因此，conv(4)的执行结果为 100。可以发现，这个过程其实就是一个“除 2 取余”的过程，并且先得到的结果（余数）最后输出，后得到的结果（余数）最先输出。它的功能是将十进制数转换为二进制表示，并输出。

函数 conv()其实是一个十进制数转换为二进制表示的递归算法，希望读者读懂记住。

**答案** 函数 conv 的功能是将十进制数转换为二进制表示，并输出。

**【例 10-28】**有如下程序：

```

long fib(int n)
{
    if(n>2) return (fib(n-1)+fib(n-2));
    else return 2;
}
main()
{
    printf("%d\n", fib(3));
}
  
```

该程序的输出结果是什么？

**【分析】**

本题考查的知识点是函数的递归调用。本题中，主函数调用函数 fib()，参数为 3，因此符合递归条件，再次调用函数 fib()；而第二次调用该函数时参数分别为 2 和 1，因此满足结束条件，都返回 2；最后第一层的 fib 返回  $2+2=4$ 。所以程序的输出结果为 4。

**【例 10-29】**编写一个程序，用位运算实现求整数 10 的相反数。

**【分析】**

在 Intel 80X86 系列的计算机中，整数是用 16 位补码表示的。其补码的表示范围是： $-2^{15} \sim 2^{15}-1$ 。补码之间存在着这样的关系：将一个数的补码按位取反，末位加 1 就得到这个补码数的相反数。因此可以通过这个方法求得在  $[-2^{15}, 2^{15}-1]$  范围内的数的相反数。具体的原因涉及到计算机中数制表示的相关知识，有兴趣可参看计算机组成原理或计算机组织结构等书籍。

下面给出参考程序清单 10-27。

程序清单 10-27

```

/*----- 10-27.c -----*/
#include <stdio.h>
  
```

```
int main(void)
{
    int a;
    a=-10+1;
    printf("%d",a);
    getchar();
    return 0;
}
```

本题的运行结果如图 10-31 所示。

-10

**【例 10-30】**有一个有序的字母序列：a, b, d, f, h, j, l, p, t；编写一个程序，要求从终端输入一个字母，将该字母插入这个序列中使得字母序列依然保持有序，然后输出新的字母序列。注意：如果输入的字母在原序列中存在，就将该字母插入到已存在字母的后面。

### 【分析】

首先要考虑的是这个字符串的存储方法。因为要在程序中添加一个新的字母，因此原有的字符数组的容量必须足够大。因为要将新的字母序列输出，为了方便操作，可将该字符数组按照字符串形式输出，所以数组最后应该加上字符串结束标志。

然后就要考虑程序的算法。可以通过比较字符的 ASCII 码判断输入的字母应该插入的位置。确定好字母应该插入的位置后，将该位置以后的所有字母顺序后移一个单元，再将该字母插入即可。

输出的时候可以应用格式符%s 按照字符串的形式输出。

下面给出本题的代码清单，程序清单 10-28。

程序清单 10-28

```
/*----- 10-28.c -----*/
#include "stdio.h"
main()
{
    char str[11]={'a','b','d','f','h','j','l','p','t'},c;
    int i,j;
    printf("Please input a alpha\n");
    scanf("%c",&c);
    for(i=0;i<9;i++)
        if(str[i]>=c)break;
    if(i<9)
        for(j=8;j>=i;j--)
            str[j+1]=str[j];
    str[i]=c;
    printf("The result is\n");
    printf("%s",str);
    getche();
}
```

本程序的执行结果如图 10-32 所示。

```
Please input a alpha
m
The result is
abdfhjlpm
```

图 10-32 程序 10-28 的运行结果

**【例 10-31】**编程实践，输入 5 个国家的名称按字母顺序排列输出。

**【分析】**

首先考虑数据的存储问题。可以用字符型的二维数组存储国家名，每一行存储一个国家名，共存储 5 行。因此需要定义一个 5 行  $n$  列的二维数组， $n$  要足够大，以便可以存放下国家的名字。C 语言规定可以把一个二维数组当成多个一维数组处理。因此本题可以按 5 个一维数组处理，而每一个一维数组就是一个国家名字符串。

然后考虑算法。可以用字符串比较函数 `strcmp()` 对字符串进行大小的比较，然后根据比较的结果对 5 个字符串排序输出。

下面给出本题的代码清单，程序清单 10-29。

程序清单 10-29

```

/*----- 10-29.c -----*/
#include "stdio.h"
main()
{
    char st[20],cs[5][20];
    int i,j,p;
    printf("input country's name:\n");
    for(i=0;i<5;i++)
        gets(cs[i]);
    printf("\n");
    for(i=0;i<5;i++)
    {
        p=i;
        strcpy(st,cs[i]);
        for(j=i+1;j<5;j++)
            if(strcmp(cs[j],st)<0)
                {p=j;strcpy(st,cs[j]);}
        if(p!=i)
        {
            strcpy(st,cs[i]);
            strcpy(cs[i],cs[p]);
            strcpy(cs[p],st);
        }
        puts(cs[i]);
    }
    printf("\n");
    getch();
}

```

本程序的执行结果如图 10-33 所示。

```

input country's name:
China
Japan
America
UK
Russia

America
China
Japan
Russia
UK

```

图 10-33 程序 10-29 的运行结果

## 10.2 常见的数据结构题

**【例 10-32】**一个顺序表所占的内存空间的大小与（ ）无关。

- (A) 表的长度                      (B) 元素的存放顺序  
(C) 元素的类型                    (D) 元素中各字段类型

**【分析】**

顺序表是一种最为简单的线性数据结构。如第1章所讲的，在创建一个顺序表时，系统会在内存中开辟一段连续的内存空间作为顺序表的存放容器。因此顺序表是逻辑上和物理上都连续的线性结构。对于本题，顺序表的表长直接决定着一个顺序表占用内存空间的大小，故选项A不正确。元素的类型也决定着顺序表占用内存空间的大小，因为一个顺序表的总容量等于表的长度和每个元素所占空间大小的乘积，因此选项C也不正确。如果顺序表的元素是结构体类型（或者其他构造类型，或者对象），那么顺序表所占空间的大小也与元素中各字段类型有关，因为元素中各字段类型决定着每个元素的大小，进而决定着整个顺序表的大小。因此选项D也不正确。只有选项B与顺序表所占的内存空间的大小无关。

**【答案】B**

**【例 10-33】**在包含1000个数据元素的线性表中，实现如下的4个操作，所需要的执行时间最长的是（ ）。

- (A) 线性表采用顺序存储结构，在第10个元素后面插入一个新的元素  
(B) 线性表采用链式存储结构，在第10个元素后面插入一个新的元素  
(C) 线性表采用顺序存储结构，删除第990个元素  
(D) 线性表采用链式存储结构，删除p所指向的链结点

**【分析】**

在顺序表中插入或删除一个元素，元素定位的时间是很短的（可以通过顺序表的数组下标直接找到要删除的元素或要插入元素的位置），其时间复杂度为 $O(1)$ 。但是要插入或删除该元素，则必须将该元素的后继元素都向后或向前移动一个存储单元，这个过程是很费时的，其时间复杂度为 $O(n)$ 。

在链表中插入或删除一个元素，首先要确定要插入结点的位置或要删除的结点的位置，这个过程必须通过顺序遍历链表来实现，因此比较费时，时间复杂度为 $O(n)$ 。但是具体的插入结点或删除结点的操作却比较简单，其时间复杂度为 $O(1)$ 。

结合本题，在包含1000个元素的顺序表中的第10个元素后面插入一个新的元素，需要将该顺序表中第11~1000个元素全部向后移动1个存储单元，再将新的元素插入到顺序表中第11个元素的位置上。在上述过程中，需要做990次元素的移动操作。而如果采用链表的形式存储该线性表，只需从该链表的表头开始顺序查找到第10个元素，再将新的元素结点插入到第10个链表结点和第11个链表结点之间。如果从包含1000个元素的顺序表中删除第990个元素，只需将第990个元素后面的10个元素向前移动1个存储单元。如果从

链表中删除指针  $p$  所指向的链结点，其时间复杂度仅为  $O(1)$ 。综上所述，实现 A 所述的操作所需要的执行时间最长。

**【答案】A**

**【例 10-34】**如果存储分配是从低地址向高地址进行的，每个元素占用 4 个存储单元，则某元素的地址是指它所占用的单元的（ ）。

- (A) 第 1 个单元的地址                      (B) 第 2 个单元的地址  
(C) 第 3 个单元的地址                      (D) 第 4 个单元的地址

**【分析】**

因为内存的分配是从低地址向高地址进行的，因此当在内存中为一个元素开辟一段内存空间时，该元素的指针（地址）总是表示先分配到的内存地址，即低地址，如图 10-34 所示。



图 10-34 元素的地址

所以某元素的地址实际上指的是该元素所占用的内存单元的第 1 个单元的地址。

**【答案】A**

**【例 10-35】**若线性表采用顺序存储结构，每个元素占 4 个存储单元，第一个元素的存储地址为 100，则第 12 个元素的地址为（ ）。

- (A) 144                      (B) 148                      (C) 112                      (D) 412

**【分析】**

由于顺序表的存储空间是连续的，因此地址也是连续的。所以可以根据某一元素的存储地址推算出其他元素的存储地址。对于本题，已知第一个元素的存储地址为 100，又知每个元素占 4 个存储单元，同时我们又知道顺序表的地址分配是从低地址向高地址进行的（从答案的 4 个选项中也可以看出），因此第 2 个元素的存储地址为  $100+4=104$ ，第 3 个元素的存储地址为  $100+8=108$ ，第  $i$  个元素的存储地址为  $100+4 \times (i-1)$ 。所以第 12 个元素的存储地址为  $100+4 \times 11=144$ 。

**【答案】A**

**【例 10-36】**若长度为  $n$  的线性表采用顺序存储结构，在表的第  $i$  个位置上插入一个数据元素，需要移动表中（ ）个元素。

- (A)  $i$                       (B)  $n+i$                       (C)  $n-i+1$                       (D)  $n-i-1$

**【分析】**

首先需要明确一点，一般情况下，我们所讲的顺序表的元素位置是从 1 开始的，也就是说一个顺序表中，第 1 个元素的位置为 1，第 2 个元素的位置为 2……依次类推。这种说法是比较统一的，也是一种约定俗成的习惯，不要因为数组的下标是从 0 开始的就等价于

顺序表第一个元素的位置为0。

明确了这一点就很容易解答此题，因为要在表的第  $i$  个位置上插入一个数据元素，那么原先的第  $i$  个元素连同后面的  $(n-i)$  个元素都要向后移动，因此需要移动表中  $n-i+1$  个元素。

【答案】C

【例 10-37】一般情况下，链表中所占用的存储单元在地址上是（ ）。

(A) 无序的 (B) 连续的 (C) 不连续的 (D) 部分连续的

【分析】

链表也是一种线性表，它的数据的逻辑组织形式是一维的。但是与顺序表不同的是，链表的物理存储结构是用一组地址任意的存储单元存储数据的。也就是说，它不像顺序表那样占据一段连续的内存空间，而是将存储单元分散在内存的任意地址上。但是每个存储单元之间通过指针变量相关联在一起，因此链表实际是一种逻辑上连续而物理地址并不连续的线性结构。

【答案】C

【例 10-38】在非空链表中由  $p$  所指向的结点后面插入一个由  $q$  所指向的结点的过程是依次执行（ ）。

(A)  $q \rightarrow \text{next} = p$ ;  $p \rightarrow \text{next} = q$ ; (B)  $q \rightarrow \text{next} = p \rightarrow \text{next}$ ;  $p \rightarrow \text{next} = q$ ;  
(C)  $q \rightarrow \text{next} = p \rightarrow \text{next}$ ;  $p = q$ ; (D)  $p \rightarrow \text{next} = q$ ;  $q \rightarrow \text{next} = p$ ;

【分析】

本题考查的知识点是链表结点的插入操作。整个执行过程如图 10-35 所示。

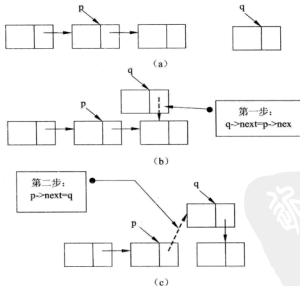


图 10-35 插入结点的执行过程

如图 10-35 所示，(a) 表示初始状态，要将  $q$  所指的结点插入到  $p$  所指结点的后面。

(b) 为第一步操作, 将  $q$  结点的指针域  $next$  指向  $p$  的后续结点, 执行  $q \rightarrow next = p \rightarrow next$ ; 操作。(c) 为第二步操作, 将  $p$  结点的指针域  $next$  指向  $q$  结点, 执行  $p \rightarrow next = q$ ; 操作。这样就可以将  $q$  所指的结点插入到  $p$  所指结点的后面。

注意这里的执行顺序不能够颠倒, 否则就不能达到预期的目标。链表结点的插入、删除等操作是链表操作中的基本操作, 大家应当掌握。

【答案】B

【例 10-39】删除非空链表中由  $p$  所指向的结点的直接后继结点的过程是依次执行 ( )。

- (A)  $r = p \rightarrow next$ ;  $p \rightarrow next = r$ ;  $free(r)$ ; (B)  $r = p \rightarrow next$ ;  $p \rightarrow next = r \rightarrow next$ ;  $free(r)$ ;  
(C)  $r = p \rightarrow next$ ;  $p \rightarrow next = r \rightarrow next$ ;  $free(p)$ ; (D)  $p \rightarrow next = p \rightarrow next \rightarrow next$ ;  $free(p)$ ;

【分析】

本题考查的知识点是链表结点的删除操作。删除  $p$  所指向的结点的直接后继结点的过程如图 10-36 所示。

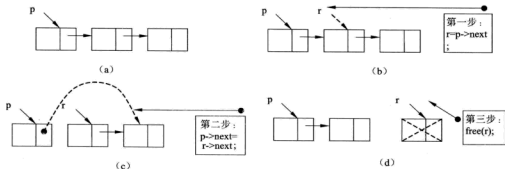


图 10-36 删除结点的执行过程

如图 10-36 所示, 要删除  $p$  所指向的结点的直接后继结点, 首先用  $r$  指向该结点, 通过执行语句  $r = p \rightarrow next$ ; 实现。然后将结点  $p$  的指针域赋值为  $r \rightarrow next$ , 这样结点  $p$  的后继结点就变为结点  $r$  的后继结点, 相当于从链表中将结点  $r$  删除。最后通过执行语句  $free(r)$ ; 释放结点  $r$ 。

【答案】B

【例 10-40】在非空的双向循环链表中由  $q$  所指向的结点后面插入一个由  $p$  所指向的结点的动作依次为:  $p \rightarrow lnext = q$ ;  $p \rightarrow rnext = q \rightarrow rnext$ ;  $q \rightarrow rnext = p$ ; ( )。

- (A)  $q \rightarrow lnext = p$ ; (B)  $q \rightarrow rnext \rightarrow lnext = p$ ;  
(C)  $p \rightarrow rnext \rightarrow lnext = p$ ; (D)  $p \rightarrow lnext \rightarrow lnext = p$ ;

【分析】

本题考查的知识点是循环双向链表的结点操作。由于循环双向链表中每个结点都包含 2 个指针域, 因此它的插入、删除等操作较单向链表要复杂一些, 因此解这类题目时要特别注意。对于本题, 要在  $q$  所指向的结点后面插入一个由  $p$  所指向的结点的操作如图 10-37 所示。

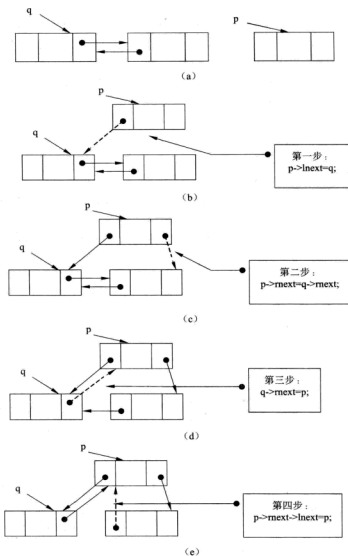


图 10-37 插入结点的执行过程

如图 10-37 所示为在  $q$  所指向的结点后面插入一个由  $p$  所指向的结点的完整过程。这个过程看起来比较复杂，其实总结起来，只需要修改 4 个指针域的值，如图 10-38 所示。

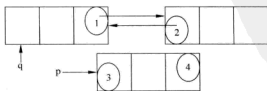


图 10-38 需要修改的指针



如图 10-38 所示, 要将  $p$  指向的结点插入到链表中两结点之间, 需要修改 (1) (2) (3) (4) 标记的指针域的值。具体说来, 将 (3) 的内容赋值为  $q$ , 这样  $p$  结点的左指针指向  $q$  结点; 将 (4) 的内容赋值为  $q \rightarrow \text{next}$ , 这样  $p$  结点的右指针指向  $q$  结点的后继结点; 将 (1) 的内容赋值为  $p$ , 这样  $q$  结点的右指针指向结点  $p$ ; 最后将 (2) 的内容赋值为  $p$ , 这样原先  $q$  结点的后继结点的左指针指向结点  $p$ 。通过上述 4 步的修改, 就将  $p$  指向的结点插入到  $q$  指向的结点的后面, 形成了一个新的双向链表。最终该双向链表变为如图 10-39 所示的样子。

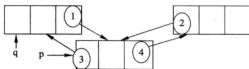


图 10-39 插入  $p$  指向的结点

图 10-39 所示的执行过程就是对 (1) (2) (3) (4) 这 4 处指针域值进行修改的具体实现。在题目中前 3 步的操作已经给出, 只需选择第四步的操作, 因此答案为 C 选项。

**【答案】C**

**【例 10-41】**编写一个函数, 测试单链表的长度。

**提示:** 函数原型可以是: `int LinkListLen(Lnode *head);`, 函数的返回值为该链表的长度。

**【分析】**

首先必须明确链表的结构和定义方式。在本书中, 链表的结点定义如下 (可参看第 1 章) :

```
typedef struct node{
    ElemType data;          /*数据域*/
    struct node *next;      /*指针域*/
}Lnode,*LinkList;
```

结点的类型为 `Lnode`。同时应用函数 `GreatLinkList()` 来创建 (初始化) 一个长度为  $n$  的链表。而且本书中所创建的链表都不包含所谓的头结点, 即每一个链表只有一个表头指针, 该指针指向链表的第一个结点元素。链表的结构如图 10-40 所示。

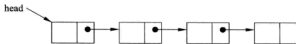


图 10-40 链表结构示意图

如图 10-40 所示, 链表并不包含头结点, 只有一个头指针指向链表的第一个结点。

明确了链表的结构, 就不难编写出测试单链表的长度的函数了。只需通过函数的参数 `head` 提供的链表头指针顺序访问链表结点, 直到访问到 `NULL` 指针为止。在这个过程中, 需设置一个变量 `count` 来记录链表的结点个数。函数 `int LinkListLen(Lnode *head)` 的描述如下:

```
int LinkListLen(Lnode *head)
{
    Lnode *p = head;
```

```

int count = 0;
while(p!=NULL)           /*循环顺序访问链表中的每一个结点*/
{
    count++;              /*记录结点的个数*/
    p = p->next;          /*指针指向下一个结点*/
}
return count;             /*返回链表中结点的个数*/
}

```

在函数 LinkListLen()中通过一个 while 循环语句顺序访问链表的每一个结点，每当访问到一个链表结点都进行 count++操作，从而使变量 count 记录下链表中结点的个数。while 循环操作直到指针 p 指向 NULL 为止。

下面给出本题的代码清单，程序清单 10-30。

程序清单 10-30

```

/*----- 10-30.c -----*/
#include "stdio.h"

typedef int ElemType;

typedef struct node{
    ElemType data;           /*数据域*/
    struct node *next;       /*指针域*/
}Lnode,*LinkList;

LinkList GreatLinkList(int n){
    LinkList p,r,list=NULL;
    ElemType e;
    int i;
    for(i=1;i<=n;i++){
        scanf("%d",&e);
        p=(LinkList)malloc(sizeof(Lnode));
        p->data=e;
        p->next=NULL;
        if(!list)
            list=p;
        else
            r->next=p;
        r=p;
    }
    return list;
}

int LinkListLen(Lnode *head)
{
    Lnode *p = head;
    int count = 0;
    while(p!=NULL)
    {
        count++;
        p = p->next;
    }
    return count;
}

main()

```

```

{
    LinkList l;
    printf("Please input 10 integer to create a linklist\n");
    l = GreatLinkList(10);          /*创建 10 个链结点的单链表*/
    printf("The LinkList includes %d node\n", LinkListLen(l));
                                    /*输出链表的长度*/
    getch();
}

```

本程序的执行结果如图 10-41 所示。


```

Please input 10 integer to create a linklist
1 2 3 4 5 6 7 8 9 0
The LinkList includes 10 node

```

图 10-41 程序 10-30 的运行结果

**【例 10-42】**编写一个函数，返回一个单链表中指定位置结点的指针。

 **提示：**函数的原型可以是：Lnode \* getPtr(Lnode \*head, int pos);，函数的返回值类型为指向链结点的指针类型 Lnode \*，参数 pos 为指定的链结点处于链表中的位置。如果要得到链表中第 3 个结点的指针，参数 pos 为 3。

#### 【分析】

在使用链表这种数据结构进行程序设计时，常常需要得到链表中某个结点的指针，以便程序对该结点进行各种操作。可以把得到单链表中指定位置结点的指针这项操作编写成一个函数，供其他函数调用，这样更加符合程序设计的结构化要求。

得到一个链表中指定位置的结点的指针十分简单。只要给定该链表的头指针，就可以顺着该头指针顺序地访问链表中的结点。当访问到第 pos 个结点时就停止继续往下访问，然后返回该结点的指针。该算法可描述如下：

```

Lnode * getPtr(Lnode *head, int pos)
{
    Lnode *p = head;
    int i;
    for(i=1; i<pos; i++)          /*循环找到第 pos 个结点的位置*/
    {
        p = p->next;;            /*指针后移*/
    }
    return p;                    /*返回第 pos 个结点的指针*/
}

```

该算法在链表操作中使用的频率非常高，例如删除指定位置的结点、在指定位置的结点后面插入新的结点、修改指定结点中数据的内容等操作都可能用到该算法。

下面给出一段测试程序，程序清单 10-31，旨在说明函数 getPtr() 的用途。

程序清单 10-31

```

/*----- 10-31.c -----*/
#include "stdio.h"
typedef int ElemType;
typedef struct node{

```

```

ElemType data;          /*数据域*/
struct node *next;       /*指针域*/
}Lnode,*LinkList;

LinkList GreatLinkList(int n){
    LinkList p,r,list=NULL;
    ElemType e;
    int i;
    for(i=1;i<=n;i++){
        scanf("%d",&e);
        p=(LinkList)malloc(sizeof(Lnode));
        p->data=e;
        p->next=NULL;
        if(!list)
            list=p;
        else
            r->next=p;
        r=p;
    }
    return list;
}

Lnode * getPtr(Lnode *head, int pos)
{
    Lnode *p = head;
    int i;
    for(i=1;i<pos;i++){
        p = p->next;
    }
    return p;
}

main()
{
    LinkList l,p,q;
    int i;
    printf("Please input 10 integer to create a linklist\n");
    l = GreatLinkList(10);

    /*删除链表中第5个结点*/
    p = getPtr(l,4);
    q = p->next;
    p->next = p->next->next;
    free(q);

    /*输出链表中的元素*/
    printf("The linklist after deleting the fifth node\n");
    p = l;
    while(p)
    {
        printf("%d ",p->data);
        p = p->next;
    }
    getch();
}

```

本程序的执行结果如图 10-42 所示。

```
Please input 10 integer to create a linklist
1 2 3 4 5 6 7 8 9 0
The linklist after deleting the fifth node
1 2 3 4 6 7 8 9 0
```

图 10-42 程序 10-31 的运行结果

**说明：**在本测试中创建了一个包含 10 个结点的单链表，然后删除第 5 个链结点。在程序中使用函数 `getPtr()` 获得第 4 个链结点的指针，然后通过下面的语句：

```
q = p->next;
p->next = p->next->next;
free(q);
```

删除并释放掉第 5 个结点。

最后程序输出删除结点后的链表内容。

**【例 10-43】**堆栈和队列的共同之处在于它们具有相同的（ ）。

(A) 逻辑特性 (B) 物理特性 (C) 运算方法 (D) 元素类型

**【分析】**

堆栈和队列都是线性数据结构，它们在逻辑上都是地址连续的线性表，因此堆栈和队列具有相同的逻辑特性。但是它们的物理特性并不一定相同，一般情况下采用顺序结构作为堆栈的存储形式，采用链结构作为队列的存储形式，因此它们的物理特性不一定相同。堆栈是先进后出 (FILO) 的线性表，其基本操作包括入栈 (Push)、出栈 (Pop) 等，而队列是先进先出 (FIFO) 的线性表，其基本操作包括入队列 (EnQueue)、出队列 (DeQueue) 等，因此它们的运算方法是不同的。至于元素类型，要根据实际应用的情形而定，堆栈和队列的元素类型可以是任意类型，没有统一的要求。

**【答案】A**

**【例 10-44】**某堆栈初始为空，Push 和 Pop 分别表示对堆栈进行一次入栈操作和出栈操作。如果给定入栈队列为 a、b、c、d、e，经过操作 Push、Push、Pop、Push、Pop、Push、Push 以后，得到的出栈队列为（ ）。

(A) b,a (B) b,c (C) b,d (D) b,e

**【分析】**

本题考查的知识点是堆栈的入栈 (Push) 和出栈 (Pop) 操作。已知入栈序列 a、b、c、d、e，下面给出每一步操作 (Push、Pop) 后堆栈、入栈队列和出栈队列的状态，如图 10-40 所示。

按照入栈队列的顺序，每执行 Push 操作，都是将入栈队列中的第一个元素压入堆栈中。每执行 Pop 操作，都是将堆栈中的栈顶元素取出放入出栈队列中。最终入栈队列中的元素为空，出栈队列中的元素为 b、c。

**【答案】B**

**【例 10-45】**已知 5 个元素的出栈序列为 1, 2, 3, 4, 5，则入栈序列可能是（ ）。

(A) 2,4,3,1,5 (B) 2,3,1,5,4 (C) 3,1,4,2,5 (D) 3,1,2,5,4

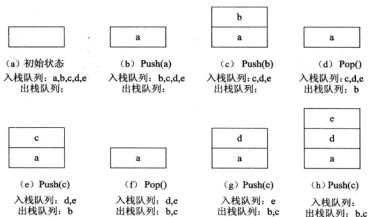


图 10-43 每一步操作 (Push、Pop) 后堆栈、入栈序列和出栈序列的状态

## 【分析】

根据堆栈的先进后出 (FILO) 的特性不难理解, 先出栈的元素一定比后出栈的元素晚进入到堆栈中, 除非该先出栈的元素在后续元素入栈之前就出栈了, 否则一旦后续元素入栈, 则后入栈元素一定比先入栈的元素先出栈。在这个原则的指导下, 可以对 ABCD4 个选项逐一判断得出答案。

如果像 A 选项那样入栈序列为 2, 4, 3, 1, 5, 它的出栈序列不可能为 1, 2, 3, 4, 5。因为只有当 2, 4, 3, 1 都入栈后才可能将元素 1 出栈, 但这种情况下, 元素 2 不可能为第 2 个出栈的元素。

如果像 B 选项那样入栈序列为 2, 3, 1, 5, 4, 它的出栈序列不可能为 1, 2, 3, 4, 5。因为只有当 2, 3, 1 都入栈后才可能将元素 1 出栈, 但这种情况下, 元素 2 不可能为第 2 个出栈的元素。

如果像 C 选项那样入栈序列为 3, 1, 4, 2, 5, 它的出栈序列不可能为 1, 2, 3, 4, 5。可以参看下列入栈出栈的步骤, 如表 10-3 所示。

表 10-3 入栈出栈的步骤

入 栈 序 列	栈中元素 (栈底—栈顶)	出 栈 序 列
3, 1, 4, 2, 5	-	-
1, 4, 2, 5	3	-
4, 2, 5	3, 1	-
4, 2, 5	3,	1
2, 5	3, 4	1
5	3, 4, 2	1
5	3, 4	1, 2

如表 10-3 所示, 当出栈序列为 1, 2 时, 栈顶元素为 4, 栈底元素为 3, 因此元素 3 不可能为第 3 个出栈元素。

如果像 D 选项那样入栈序列为 3, 1, 2, 5, 4, 按照如表 10-4 所示的入栈出栈步骤操作可得到出栈序列为 1, 2, 3, 4, 5。

表 10-4 入栈出栈的步骤

入栈序列	栈中元素(栈底—栈顶)	出栈序列
3, 1, 2, 4, 5	-	-
1, 2, 4, 5	3	-
2, 4, 5	3, 1	-
2, 4, 5	3,	1
4, 5	3, 2	1
4, 5	3	1, 2
4, 5	-	1, 2, 3
5	4	1, 2, 3
5	-	1, 2, 3, 4
-	5	1, 2, 3, 4
-	-	1, 2, 3, 4, 5

**【答案】D**

**【例 10-46】**若队列采用链存储结构，队头元素指针为  $front$ ，队尾元素指针为  $rear$ ，向该队列中插入一个由  $p$  指向的新结点的过程为执行 ( )， $rear=p$ 。

(A)  $rear = p$ ; (B)  $front = p$ ; (C)  $rear \rightarrow next = p$ ; (D)  $front \rightarrow next = p$ ;

**【分析】**

本题考查的知识点是向链队列中插入元素。首先必须明确向队列中插入元素一定是从队尾插入，而不能从队头插入。在链队列中，指针变量  $rear$  指向队尾元素，要向队尾插入由  $p$  指向的新结点，首先要将指针  $p$  赋值给  $rear \rightarrow next$  域，然后再将指针  $p$  赋值给  $rear$ ，这样  $rear$  依然指向队列的尾部。这个过程如图 10-44 所示。

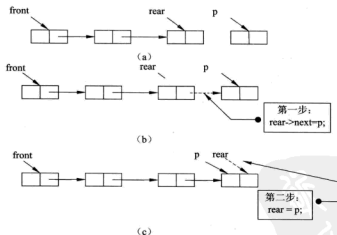


图 10-44 向链队列尾部插入元素的过程

**【答案】C**

**【例 10-47】**在循环队列中，如果  $front$  和  $rear$  分别表示队头元素和队尾元素的位置，则判断循环队列空的条件是 ( )。

(A) front = rear+1      (B) rear = front+1      (C) front = rear      (D) front=0

### 【分析】

本题考查的知识点是循环队列的结构及循环队列的判空条件。要解决本题，首先要知道循环队列的结构。通过第1章的介绍可知，循环队列一般由顺序表构成，它的容量是固定的，并且它的队头指针和队尾指针都可以随着元素入队列而发生改变，这样循环队列逻辑上就好像是一个环形的存储空间，只要队列中还有空单元未使用，就可以向队列中存放元素。循环队列的结构如图10-45所示。

如图10-45所示，循环队列的头指针 front 指向循环队列的第一个元素，用于出队列。每次出队列操作时，都将 front 指向的元素返回，然后执行 front 自增1操作。循环队列的尾指针 rear 指向队尾元素的下一个空间，每次入队列操作时，将队尾元素放置 rear 指向的空间，然后执行 rear 自增1操作。在实际的存储中，循环队列是用顺序表存储的。因此循环队列中只能存放其容量减1个元素。当  $(rear+1)\%n$  ( $n$  为存放循环队列的顺序表的容量) 等于 front 时，表明该循环队列已满，不能再进行元素的入队列操作。当 front 等于 rear 时，表明该循环队列为空。

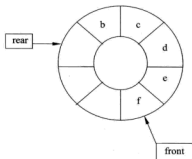


图 10-45 循环队列结构示意图

在这里希望读者牢记循环队列的入队列、出队列、队列判空、队列判满等基本操作，请参看第1章1.5节的内容。

### 【答案】C

【例 10-48】设  $n$  个元素的进栈序列为  $1, 2, 3, \dots, n$ ，出栈序列为  $p_1, p_2, \dots, p_n$ ，若  $p_1=n$ ，则  $p_i(1 \leq i < n)$  的值为 ( )。

(A)  $i$       (B)  $n-i$       (C)  $n-i+1$       (D) 有多种可能

### 【分析】

因为  $p_1=n$ ，即进栈序列中的最后一个元素最先出栈，因此说明元素是按照进栈序列的顺序  $1, 2, \dots, n$  依次进栈的。如果中途有元素出栈，那么出栈序列的第一个元素不可能为  $n$ ，即  $p_1$  不可能等于  $n$ 。因此元素出栈时也必须按照  $n, n-1, \dots, 2, 1$  的顺序出栈，这样才符合堆栈先进后出的原则。因此  $p_i(1 \leq i < n)$  的值为  $n-i+1$ 。

### 【答案】C

【例 10-49】如果堆栈采用顺序结构存储，向堆栈中插入一个元素，栈顶指针 top 的变化是 ( )。

(A) 不变      (B) top=0      (C) top--      (D) top++

### 【分析】

采用顺序结构存储的堆栈设置有两个指针，即 top 和 base，top 指向堆栈的栈顶，base 指向堆栈的栈底。初始化一个堆栈时，top 和 base 同时指向栈底单元。当向堆栈中插入一个元素时，将元素存放至指针 top 指向的存储单元中，再作 top 自增1操作。当从栈顶删除一个元素时，先作 top 自减1操作，再将 top 指向的存储单元中的数据取出。栈底指针 base 不发生任何变化，它是和 top 配合，用来判断堆栈中元素的个数的。因此可以得出这样的结论：栈顶指针 top 始终指向栈顶元素的上一个内存单元，top 指向的内存单元中不存



放数据。所以本题中要向堆栈中插入一个元素，栈顶指针  $top$  的变化是  $top++$ 。

【答案】D

【例 10-50】为了解决打印机与主机（PC）之间的速度不匹配的问题（主机的速度要比打印机快很多），一般会设置一个打印数据缓冲区存放要打印的数据。打印机则依次从该缓冲区中读取数据打印。请问该缓冲区应当是一个什么数据结构？

【分析】

为了缓和 I/O 设备与 CPU 之间速度不相匹配的矛盾，在数据的到达速率与其离去的速率不同的地方都可以设置缓冲区。例如在主机和打印机之间就可以设置缓冲区。当主机处理完数据，将数据暂存到打印缓冲区中后就可以继续处理其他数据，这样主机不必等待慢速的打印机打印完数据。而打印机设备每次从缓冲区中读取数据打印。打印缓冲区采用队列结构，因为打印缓冲的过程遵循先来先服务（FCFS）的原则，即先送到缓冲区中的数据先打印，后送到缓冲区中的数据后打印，否则打印的顺序就会发生混乱。

【答案】队列结构

【例 10-51】编写一个函数，测试一个队列的长度。

提示：函数的原型可以是：`int getQueueLen(LinkQueue *q)`；函数的返回值为队列的长度。

【分析】

首先必须明确队列的结构和定义方式。在本书中，队列采用链式存储结构，其定义如下（可参看第1章）：

```
typedef struct QNode{
    QElemType data;
    struct QNode *next;
} QNode , *QueuePtr;
typedef struct{
    QueuePtr front;    //队头指针
    QueuePtr rear;     //队尾指针
}LinkQueue;
```

队列类型 `LinkQueue` 中包含两个域。`front` 是队列的头指针，指向队列的队首。`Rear` 是队列的尾指针，指向队列的队尾。队列中每个结点的类型为 `QNode`，结点和结点之间通过指针连接到一起。队列的第一个结点为队头结点，不存放队列元素。队列的结构如图 10-46 所示。

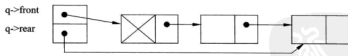


图 10-46 队列结构示意图

明确了队列的结构，就不难编写出测试队列的长度的函数了。可以从 `q->front` 开始顺序遍历整个队列，直到遍历到队列的尾部。在这个过程中使用一个变量 `count` 来记录下队列中结点的个数。其算法具体描述如下：

```
int getQueueLen(LinkQueue *q)
{
    QNode *p;
```

```

int count = 0;
p = q->front;          /*指针 p 指向队头结点*/
while(p!=q->rear)       /*循环遍历队列，计算队列的结点个数*/
{
    p = p->next;        /*指针后移*/
    count++;
}
return count;          /*返回队列的结点个数*/
}

```

在本算法中，首先将指针  $p$  指向队列的队头结点  $q \rightarrow \text{front}$ ，然后循环地顺序遍历整个队列，直到  $p$  等于队列的尾指针  $q \rightarrow \text{rear}$  为止。在这个过程中用变量  $\text{count}$  记录下队列中结点的个数。如果队列中的元素为空，那么该队列只存在一个队头结点，即开始时  $q \rightarrow \text{front}$  就等于  $q \rightarrow \text{rear}$ ，因此  $\text{count}$  的值为 0。

下面给出本题的代码清单，见程序清单 10-32。

程序清单 10-32

```

/*----- 10-32.c -----*/
#include "stdio.h"

typedef char ElemType;
typedef struct QNode{
    ElemType data;
    struct QNode *next;
} QNode , *QueuePtr;
typedef struct{
    QueuePtr front; //队头指针
    QueuePtr rear;  //队尾指针
}LinkQueue;

initQueue(LinkQueue *q){
    /*初始化一个空队列*/
    q->front = q->rear = (QueuePtr)malloc(sizeof(QNode));
    /*创建一个头结点，队头队尾指针指向该结点*/
    if( !q->front) exit(0); //创建头结点失败*/
    q->front->next = NULL; //头结点指针域置 NULL*/
}

EnQueue(LinkQueue *q, ElemType e){
    QueuePtr p;
    p = (QueuePtr)malloc(sizeof(QNode)); //创建一个队列元素结点*/
    if( !q->front) exit(0); //创建头结点失败*/
    p->data = e;
    p->next = NULL;
    q->rear->next = p;
    q->rear = p;
}

int getQueueLen(LinkQueue *q)
{
    QNode *p;
    int count = 0;
    p = q->front;
    while(p!=q->rear)

```

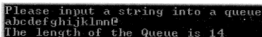
```

    {
        p = p->next;
        count++;
    }
    return count;
}

main()
{
    LinkQueue q;
    char e;
    initQueue(&q);                /*初始化一个队列 q*/
    printf("Please input a string into a queue\n");
    scanf("%c",&e);
    while(e!='@'){
        EnQueue(&q,e);            /*向队列中输入字符串, 以@表示结束*/
        scanf("%c",&e);
    }
    printf("The length of the Queue is %d\n",getQueueLen(&q));
    /*输出队列的长度*/
    getch();
}

```

本程序的执行结果如图 10-47 所示。



```

Please input a string into a queue
abcdefghijklmn@
The length of the Queue is 14

```

图 10-47 程序 10-32 的运行结果

**【例 10-52】**按照二叉树的定义，具有 3 个结点的二叉树具有多少种不同的形态。

**【分析】**

本题考查的知识点是二叉树的定义以及二叉树的形态。二叉树的定义为：它或者为空，或者由一个根结点加上两棵分别称为左子树和右子树的互不相交的二叉树组成。显然这个定义是递归形式的。因此不难理解，在左右子树的结点数确定的条件下，一棵二叉树的形态数等于它左子树的形态数乘以它右子树的形态数。例如本题，一个二叉树包含 3 个结点，根结点必然占据一个结点，因此左右子树的结点数有 3 种组合，即（左 1，右 1）；（左 2，右 0）；（左 0，右 2）。在每种组合下，二叉树的形态数都等于它左子树的形态数乘以它右子树的形态数。

第一种组合（左 1，右 1），显然左子树的形态数只能是 1 种，右子树的形态数也只能是 1 种。因此在左子树为 1，右子树为 1 的条件下，3 个结点的二叉树只有 1 种形态，如图 10-48 所示。

第二种组合（左 2，右 0），显然左子树的形态数只能是 2 种，右子树的形态数也只能是 1 种。因此在左子树为 2，右子树为 0 的条件下，3 个结点的二叉树只有  $1 \times 2 = 2$  种形态，如图 10-49 所示。

第三种组合（左 0，右 2），显然左子树的形态数只能是 1 种，右子树的形态数也只能是 2 种。因此在左子树为 0，右子树为 2 的条件下，3 个结点的二叉树只有  $1 \times 2 = 2$  种形

态,如图 10-50 所示。

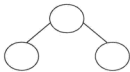


图 10-48 (左 1, 右 1) 条件下二叉树的形态

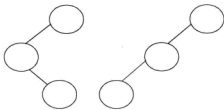


图 10-49 (左 2, 右 0) 条件下二叉树的两种形态

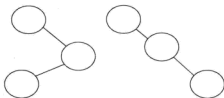


图 10-50 (左 0, 右 2) 条件下二叉树的两种形态

因此具有 3 个结点的二叉树共有  $1+2+2=5$  种形态。

本题很简单,只考虑 3 个结点的二叉树的形态情况,因此以上的分析显得有些繁琐。但是如果题目变得更复杂,例如问 10 个结点的二叉树的形态数是多少,就必须通过一种合理的思考路线去求解,而不能没有规律的猜想。因此可以用上述的这种递归的思想去求解。其实在计算结点数较多的二叉树的左右子树的形态数目时,应当将左右子树作为独立的二叉树去考虑。

**【答案】**5 种

**【例 10-53】**若一棵二叉树有 10 个度为 2 的结点,则该二叉树的叶结点的个数为( )。

(A) 9      (B) 11      (C) 12      (D) 不确定

**【分析】**

在树结构中,结点的度被定义为该结点拥有子树的数目,如图 10-51 所示。

结点 A 拥有 2 棵子树,因此结点 A 的度为 2。结点 B 和 C 为叶结点没有子树,因此度为 0。另外还需知道,二叉树中除了度为 2 的结点就是度为 1 的结点和叶结点。而且任何树中每个结点的度数之和等于总的结点数减 1。例如如图 10-51 所示的二叉树,每个结点的度数之和为  $\text{degree}(A)+\text{degree}(B)+\text{degree}(C)=2+0+0=2$ ,它等于总的结点数 3 减 1。这是因为在一棵树中,除了根结点之外,每个结点都对应一个指向其父结点的“度”。例如图 10-51 所示的二叉树,结点 B 和结点 C 都对应一个指向其父结点结点 A 的度,但是根结点没有父结点,因此根结点不对应这样的度。所以可以得出结论:一棵树的结点数为  $n$ ,总度数为  $d$ ,存在关系  $n-1=d$ 。



图 10-51 结点的度

对于本题,已知二叉树有 10 个度为 2 的结点,又知二叉树中除了度为 2 的结点就是度为 1 的结点和叶结点。因此可设二叉树中度为 2 的结点数为  $n_2$ ,度为 1 的结点数为  $n_1$ ,叶结点数为  $n_0$ 。由结点和度数的关系可知  $n_2+n_1+n_0-1=2n_2+n_1$ ,即  $n_0-1=n_2$ 。因为  $n_2=10$ ,因

此  $n_0=11$ ，因此该二叉树叶结点的个数为 11。

【答案】B

【例 10-54】若一棵满二叉树有 2047 个结点，则该二叉树中叶结点的个数为 ( )。

(A) 512 (B) 1024 (C) 2048 (D) 4096

【分析】

所谓满二叉树是指深度为  $k$  且有  $2^k-1$  个结点的二叉树。它的形态如图 10-52 所示。

满二叉树的叶结点都集中在二叉树的最下面一层。因为深度为  $k$  的满二叉树有  $2^k-1$  个结点，因此我们可以通过题目中给出的满二叉树的结点数 2047 求出该二叉树的深度。

$$2^k-1=2047$$

$$\Rightarrow k=11$$

因此该二叉树的深度为 11 层。又因为满二叉树中，每层的结点数为  $2^{k-1}$ ,  $k=1,2,\dots,n$ ，所以第 11 层的结点数，即叶结点数为  $2^{10}=1024$  个。

【答案】B

【例 10-55】已知一棵二叉树的根结点指针为 T，编写一个算法计算该二叉树叶结点的个数。

【分析】

由于二叉树结构为递归定义的，因此许多二叉树的操作（二叉树的遍历等）都是通过递归过程实现的。计算二叉树的结点个数就是这样一个典型的问题。可以通过遍历整棵二叉树访问到二叉树的叶结点，然后设置一个变量累加计算叶结点的个数。而二叉树的遍历并没有特殊的要求，采用先序遍历、中序遍历、后序遍历都可以。因此计算二叉树 T 的叶结点个数的算法描述如下：

```
getLeaves(BiTree T, int *count){
    if(T->lchild==NULL && T->rchild==NULL && T!=NULL){ /*访问到叶结点*/
        *count = *count + 1;
    }
    if(T){
        getLeaves(T->lchild, count); /*先序遍历 T 的左子树*/
        getLeaves(T->rchild, count); /*先序遍历 T 的右子树*/
    }
}
```

函数 getLeaves() 的作用是计算二叉树 T 的叶结点个数，叶结点的个数用参数 count 返回。条件语句

```
if(T->lchild==NULL && T->rchild==NULL && T!=NULL){ /*访问到叶结点*/
    *count = *count + 1;
}
```

的作用是判断当前访问到的结点是否是叶结点，如果是叶结点则 \*count 值加 1。这里注意，在函数 getLeaves() 中 count 为一个指针，\*count 为该指针指向的计数器变量。该计数器变量应在主调函数中定义。可参看答案中给出的参考程序。条件语句

```
if(T){
```

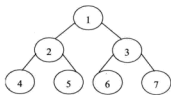


图 10-52 满二叉树示意

```

        getLeaves(T->lchild, count); /*先序遍历 T 的左子树*/
        getLeaves(T->rchild, count); /*先序遍历 T 的右子树*/
    }

```

的作用是实现在先序遍历二叉树 T。也就是说当 T 不为空时，先先序遍历 T 的左子树，再先序遍历 T 的右子树。参数 count 为一个指针（地址），它的形参为 int \*count，只有通过指针的传递才能实现 \*count 值的累加。

函数 getLeaves() 在遍历二叉树 T 的同时对遍历到的结点进行判断，看它是不是叶结点，如果是叶结点，则通过由函数形参传递下来的指针 count 将主调函数中的变量值 \*count 作加 1 操作。最终可得到该二叉树的叶结点数。

下面给出完整的测试程序，见程序清单 10-33。

程序清单 10-33

```

/*----- 10-33.c -----*/
#include <string.h>
#include <stdio.h>

typedef struct BiTNode{
    char data; /*结点的数据域*/
    struct BiTNode *lchild , *rchild; /*指向左孩子和右孩子*/
} BiTNode , *BiTree;
/*创建一棵二叉树*/
CreatBiTree(BiTree *T){
    char c;
    scanf("%c",&c);
    if(c == ' ' ) *T = NULL;
    else{
        *T = (BiTNode * )malloc(sizeof(BiTNode)); /*创建根结点*/
        (*T)->data = c; /*向根结点中输入数据*/
        CreatBiTree(&((*T)->lchild)); /*递归地创建左子树*/
        CreatBiTree(&((*T)->rchild)); /*递归地创建右子树*/
    }
}


/*遍历二叉树，得到叶结点的个数*/
getLeaves(BiTree T,int *count){
    if(T->lchild==NULL && T->rchild==NULL && T!=NULL){
        *count = *count + 1; /*访问到叶结点*/
    }
    if(T){
        getLeaves(T->lchild, count); /*先序遍历 T 的左子树*/
        getLeaves(T->rchild, count); /*先序遍历 T 的右子树*/
    }
}

main()
{
    int count = 0;
    BiTree T = NULL; /*最开始 T 指向空*/
    CreatBiTree(&T); /*创建二叉树*/
    getLeaves(T, &count); /*遍历二叉树，得到叶结点的个数，用 count 返回叶结点的个数*/
}

```

```
printf("The number of leaves of BTree are %d\n",count);
getche();
}
```

本程序的运行结果如图 10-53 所示。

说明：程序首先用函数 CreatBiTree() 创建一棵二叉树，并将该二叉树的根结点指针赋值给变量 T。执行该程序创建的二叉树如图 10-54 所示。

```
abcd e f gh
The number of leaves of BTree are 4
```

图 10-53 程序 10-33 的运行结果

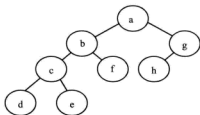


图 10-54 用户创建的二叉树

然后通过函数 getLeaves() 计算该二叉树的叶结点数目，并打印出来。用户创建的二叉树中叶结点数目为 4。

**【例 10-56】** 已知内存中存储有如图 10-55 所示的二叉树。

编写一个算法，求出字母 C 位于二叉树中的层数。

**【分析】**

与上一例题类似，可以通过遍历该二叉树找到字符 C 所在的结点，并返回该结点所处二叉树的层数。关键的问题在于如何求得该结点所处二叉树中的层数。解决这类问题的通用的办法是在递归函数的参数中设置一个变量 level，记录当前访问的结点所处二叉树中的层数。每调用一次该递归函数，就将该变量 level 加 1，并作为递归函数的参数进行传递。变量 level 是本次递归调用的局部变量，它会随着递归调用层数的变化而发生变化。一旦程序遍历到包含字符 C 的结点就停止遍历后续结点，并返回此时的层数 level。具体算法描述如下：

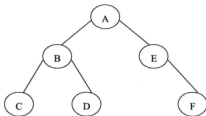


图 10-55 已有的二叉树

```
PreOrderTraverse(BiTree T,int level){
    if(T){ /*递归结束条件，T为空*/
        if(T->data=='C'){
            return level;
        }
        l = PreOrderTraverse(T->lchild,level+1); /*先序遍历T的左子树*/
        if(l != 0) return l;
        else
            return PreOrderTraverse(T->rchild,level+1); /*先序遍历T的右子树*/
    }
    return 0;
}
```

在函数 PreOrderTraverse() 中，参数 level 为一个局部变量。当递归地调用函数 PreOrderTraverse() 时，要将 level 加 1 后作为参数传递，这样变量 level 会随着递归深度的

变化而变化,因为每一层递归调用中变量 level 都不是同一个变量。当找到包含字符 C 的结点后,程序返回 level 的值。但是必须注意的是,程序一旦找到了包含字符 C 的结点后,遍历二叉树的行为也要随即结束,因为如果继续遍历下去,变量 level 仍旧会随着递归深度的变化而变化,这样记录的层数就没有意义了。本算法中采用 return 语句结束递归,即一旦找到包含字符 C 的结点,就将程序从递归中一层一层地用 return 语句返回,返回的结果为最终记录下的包含字符 C 的结点处在二叉树中的层数 level。如果没有找到包含字符 C 的结点,则程序返回 0。

下面给出完整的测试程序,见程序清单 10-34。

程序清单 10-34

```

/*----- 10-34.c -----*/
#include "stdio.h"

typedef struct BitNode{
    char data; /*结点的数据域*/
    struct BitNode *lchild, *rchild; /*指向左孩子和右孩子*/
} BitNode, *BiTree;

/*创建一棵二叉树*/
CreatBiTree(BiTree *T){
    char c;
    scanf("%c",&c);
    if(c == ' ' ) *T = NULL;
    else{
        *T = (BitNode *)malloc(sizeof(BitNode)); /*创建根结点*/
        (*T)->data = c; /*向根结点上输入数据*/
        CreatBiTree(&((*T)->lchild)); /*递归地创建左子树*/
        CreatBiTree(&((*T)->rchild)); /*递归地创建右子树*/
    }
}

int PreOrderTraverse(BiTree T,int level){
    int l;
    if(T){ /*递归结束条件, T 为空*/
        if(T->data=='C')
            return level;
        l = PreOrderTraverse(T->lchild,level+1); /*先序遍历 T 的左子树*/
        if(l != 0) return l;
        else
            return PreOrderTraverse(T->rchild,level+1); /*先序遍历 T 的右子树*/
        }
    return 0;
}

main()
{
    int level = 1;
    BiTree T = NULL; /*最开始 T 指向空*/
    CreatBiTree(&T); /*创建二叉树*/
    /*遍历二叉树, 找到包含 D 字符结点位于二叉树中的层数*/
    printf("C is at level %d in the BiTree\n",PreOrderTraverse(T,level) );
    getch();
}

```



本程序的运行结果如图 10-56 所示。

```
abcd e f gh
The number of leaves of BTree are 4
```

图 10-56 程序 10-34 的运行结果

**说明：**首先通过函数 `CreatBiTree()` 创建一棵形如图 10-34 所示的二叉树，然后调用函数 `PreOrderTraverse()` 计算出字母 C 位于二叉树中的层数，并将其打印在屏幕上。

**【例 10-57】** 在一个图中，所有顶点的度数之和等于所有边数的（ ）倍。

- (A)  $1/2$       (B) 1      (C) 2      (D) 4

**【分析】**

本题考查的知识点是图的顶点的度数与边数的关系。在图结构中，顶点的度是指依附于该顶点的边数，如图 10-57 所示。

在图中每个顶点 (A, B, C) 的度都为 2。因为图中的每一条边都连接着图中的两个顶点，例如边 1 连接着顶点 A 和顶点 B，因此每一条边都分担着它所连接的两个顶点的—一个度数。例如顶点 A 的度数为 2，边 1 占据顶点 A 的一个度数；顶点 B 的度数为 2，边 1 也占据顶点 B 的一个度数。这样就不难理解，图中每个顶点的度数之和等于该图所有边数的 2 倍。例如图 10-57 所示的图，所有顶点的度数之和为 6，而该图的边数为 3。

**【答案】** C

**【例 10-58】** 在一个具有  $n$  个顶点的无向图中，要连通全部顶点至少需要（ ）条边。

- (A)  $n$       (B)  $n+1$       (C)  $n-1$       (D)  $2n$

**【分析】**

本题考查的知识点是无向图中顶点数与边数的关系。首先要明确一些概念。所谓无向图就是指图中两个顶点之间的边不存在方向。例如顶点 A 和顶点 B 之间存在一条边，那么既可以从顶点 A 通过该边到达顶点 B，又可以从顶点 B 通过该边到达顶点 A。对于无向图，若从顶点  $v_i$  到顶点  $v_j (i \neq j)$  有路径，则称顶点  $v_i$  到顶点  $v_j$  之间是连通的。所谓连通全部顶点，就是指图中任意顶点之间都是连通的，也就是说在图中任选两个顶点，它们之间都是连通的，都存在路径。

在一个具有  $n$  个顶点的无向图中，要连通全部顶点，最简单的办法就是将全部的顶点用边单向无环“串连”在一起，如图 10-58 所示。

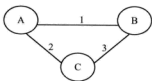


图 10-57 图的示意

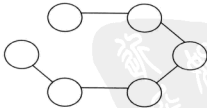


图 10-58 将顶点“串连”在一起

这样从任意顶点出发都可以到达图中的其他顶点。显然应用这种方法构成的连通的无向图边数是最少的。因此可以得出结论：在一个具有  $n$  个顶点的无向图中，要连通全部顶

点至少需要  $n-1$  条边。

【答案】C

【例 10-59】一个具有  $n$  个顶点的有向图最多有 ( ) 条边。

- (A)  $n(n-1)/2$  (B)  $n(n-1)$  (C)  $n(n+1)/2$  (D)  $n^2$

【分析】

本题考查的知识点是有向图中顶点数与边数的关系。所谓有向图是指图中的顶点之间的边存在方向,如图 10-59 所示。

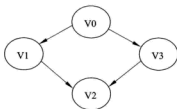


图 10-59 有向图示意

图 10-59 所示为一个有向图,该有向图由 4 个顶点和 4 条边构成。每个顶点之间都有一条边相连接,而且这些边都存在着方向。

在一个具有  $n$  个顶点的有向图中,如果从每个顶点都发射出  $n-1$  条边,指向其他  $n-1$  个顶点,那么这样的有向图边数是最多的。它的边数可达到  $n(n-1)$  条。

【答案】B

【例 10-60】一个具有  $n$  个顶点的无向图最多有 ( ) 条边。

- (A)  $n(n-1)/2$  (B)  $n(n-1)$  (C)  $n(n+1)/2$  (D)  $n^2$

【分析】

本题考查的知识点是无向图中顶点数与边数的关系。与上一题不同的是,这里要求计算无向图的边数。无向图与有向图的区别在于顶点之间的边是没有方向的,因此在无向图中,两顶点之间至多存在一条边。而在有向图中,两顶点之间可以存在两条方向不同的边。因此不难理解,一个具有  $n$  个顶点的有向图最多有  $n(n-1)$  条边,而一个具有  $n$  个顶点的无向图最多有  $n(n-1)/2$  条边。

【答案】A

【例 10-61】若图的邻接矩阵中主对角线上的元素为 0,其余元素全为 1,则该图一定是 ( )。

- (A) 无向图 (B) 有向图 (C) 完全图 (D) 不确定

【分析】

本题考查的知识点是图的邻接矩阵表示。图的一种表示方法或存储形式是邻接矩阵,它是这样定义的:定义一个二维数组  $A[0 \cdots n-1][0 \cdots n-1]$ ,  $A$  称为邻接矩阵,它存放顶点之间的关系信息。 $A[i][j]$  定义为:

$$A[i][j] = \begin{cases} 1 & \text{当顶点 } i \text{ 与顶点 } j \text{ 之间有边} \\ 0 & \text{当顶点 } i \text{ 与顶点 } j \text{ 之间无边} \end{cases}$$

因此,如果图的邻接矩阵中主对角线上的元素为 0,就表明图中的每个顶点自身不存在环,即不会出现如图 10-60 所示的情形。

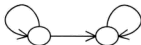


图 10-60 顶点存在环的有向图

如图 8-60 所示，图中两个顶点都存在环。

又知该图的邻接矩阵中其余元素全为 1，也就说明图中任意两个顶点之间都存在边，而且顶点  $v_i$  与顶点  $v_j$  之间存在边，同时顶点  $v_j$  与顶点  $v_i$  之间也存在边。这样就出现了一个问题，无法从该图的邻接矩阵中判断出该图是有向图还是无向图。因为如果顶点  $v_i$  与顶点  $v_j$  之间存在一条无向的边，那么其邻接矩阵中  $A[i][j]$  与  $A[j][i]$  的值都为 1。但是如果顶点  $v_i$  与顶点  $v_j$  之间存在两条有向的边，一条从顶点  $v_i$  指向顶点  $v_j$ ，另一条从顶点  $v_j$  指向  $v_i$ ，那么其邻接矩阵中  $A[i][j]$  与  $A[j][i]$  的值也都为 1。因此无法从邻接矩阵中判断出该图是有向图还是无向图。但是可以明确的是，该图一定是一个完全图。对于无向图来说，完全图就是包含  $n(n-1)/2$  条边的图，对于有向图来说，完全图就是包含  $n(n-1)$  条边的图。它们对应的邻接矩阵都是主对角线上的元素为 0，其余元素全为 1。

【答案】C

