算法艺术与信息学竞赛

- 适合语言零基础的初学者
- 涵盖算法竞赛的主要知识点
- 大量经验教训与比赛技巧
- 简洁、清晰、高效的示例代码
- 丰富的辅助教学资源与配套习题

算法竞赛入门经典

◎ 刘汝佳 编著

清华大学出版社

丛书介绍

算法在计算机科学乃至于整个科学界的作用日益明显。它们不仅具有重要的理论意义，而且解决了生产生活中的很多实际问题。程序设计竞赛就是这样一类以算法为核心但是偏重实用性的比赛。随着各类比赛规模的逐渐扩大，程序设计竞赛在各高校、IT公司和其他社会各界中越来越受到认可和重视。很多研究工作者和从事IT行业的人尽管不参加这类竞赛，但也希望具有这方面的能力，受到这方面的专业训练。

本丛书的前身是5年前的同名图书《算法艺术与信息学竞赛》。5年来，更多的人加入到参赛、命题和组织的队伍中来，各类竞赛的参赛和命题水平也有了长足的进步。作者深知当年的经典之作开始显得题目陈旧，知识的广度和深度也无法达到当今高水平比赛的要求了。因此，将原书的内容扩充、完善后分成三本，以丛书的形式依次展现给读者。这三本书循序渐进，从零语言基础开始讲起，直到超越竞赛本身，真正把算法当成“艺术”。

- 《算法竞赛入门经典》
- 《算法实践手册》
- 《算法艺术与问题求解》

ISBN 978-7-302-20608-8
9 787302 206088 >
定价：24.00元

算法艺术与信息学竞赛

算法竞赛入门经典

刘汝佳　编著

清华大学出版社

北　京

内 容 简 介

本书是一本算法竞赛的入门教材，把C/C++语言、算法和解题有机地结合在了一起，淡化理论，注重学习方法和实践技巧。全书内容分为11章，包括程序设计入门、循环结构程序设计、数组和字符串、函数和递归、基础题目选解、数据结构基础、暴力求解法、高效算法设计、动态规划初步、数学概念与方法、图论模型与算法，覆盖了算法竞赛入门所需的主要知识点，并附有大量习题。书中的代码规范、简洁、易懂，不仅能帮助读者理解算法原理，还能教会读者很多实用的编程技巧。另外，书中包含的各种开发、测试和调试技巧也是在传统的语言、算法类书籍中难以见到的。

本书可作为全国青少年信息学奥林匹克联赛（NOIP）的复赛教材及ACM国际大学生程序设计竞赛（ACM/ICPC）的入门参考，还可作为IT工程师与科研人员的参考用书。

图书在版编目（CIP）数据

算法竞赛入门经典/刘汝佳编著. —北京：清华大学出版社，2009.11
（算法艺术与信息学竞赛）

ISBN 978-7-302-20608-8

I. 算… II. 刘… III. ①电子计算机-算法理论-教材 ②C语言-程序设计-教材 IV. TP301.6 TP312

中国版本图书馆CIP数据核字（2009）第118407号

责任编辑：朱英彪　郭　伟
封面设计：张　岩
版式设计：王世情
责任校对：王　云
责任印制：杨　艳
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：清华大学印刷厂
装 订 者：三河市李旗庄少明装订厂
经　　销：全国新华书店
开　　本：185×260　**印　张**：15　**字　数**：347千字
版　　次：2009年11月第1版　　**印　　次**：2009年11月第1次印刷
印　　数：1～5000
定　　价：24.00元

前　言

“听说你最近在写一本关于算法竞赛入门的书？”朋友问我。
“是的。”我微笑道。
“这是怎样的一本书呢？”朋友很好奇。
“C 语言、算法和题解。”我回答。
“什么？几样东西混着吗？”朋友很吃惊。
“对。”我笑了，“这是我思考许久后做出的决定。”

大学之前的我

12 年前，当我翻开 Sam A.Abolrous 所著《C 语言三日通》的第一页时，我不会想到自己会有机会编写一本讲解 C 语言的书籍。当时，我真的只花了 3 天就学完了这本书，并且自信满满：“我学会 C 语言啦！我要用它写出各种有趣、有用的程序！”但渐渐地，我认识到了：虽然浅显易懂，但书中的内容只是语言入门，离实际应用还有较大差距，就好比**小学生学会造句以后还要下很大功夫才能写出像样的作文。**

第二本对我影响很大的书是 Sun 公司的 Peter van der Linden（PvdL）所著的《C 程序设计奥秘》。作者称该书应该是每一个程序员“在 C 语言方面的第二本书”，因为“书中绝大部分内容、技巧和技术在其他任何书中都找不到”。原先我只是把自己当成是程序员，但在阅读的过程中，我开始渐渐了解到硬件设计者、编译程序开发者、操作系统编写者和标准制定者是怎么想的。继续的阅读增强了我的领悟：**要学好 C 语言，绝非熟悉语法和语义这么简单。**

后来，我自学了数据结构，懂得了编程处理数据的基本原则和方法，然后又学习了 8086 汇编语言，甚至曾没日没夜地用 SoftICE 调试《仙剑奇侠传》，并把学到的技巧运用到自己开发的游戏引擎中。再后来，我通过《电脑爱好者》杂志上的一则不起眼的广告了解到了全国信息学奥林匹克联赛（当时称为分区联赛，NOIP 是后来的称谓）。“学了这么久编程，要不参加个比赛试试？”想到这里，我拉着学校里另外一个自学编程的同学，找老师带我们参加了 1997 年的联赛——在这之前，学校并不知道有这个比赛。凭借自己的数学功底和对计算机的认识，我在初赛（笔试）中获得全市第二的成绩，进入了复赛（上机）。可我的上机编程比赛的结果是“惨烈”的：第一题有一个测试点超过了整数的表示范围；第二题看漏了一个条件，一分都没得；第三题使用了穷举法，全部超时。考完之后我原以为能得满分的，结果却只得了 100 分中的 20 多分，名落孙山。

痛定思痛，我开始反思这个比赛。一个偶然的机会，我拿到了一本联赛培训教材。书上说，比赛的核心是算法（Algorithm），并且推荐使用 Pascal 语言，因为它适合描述算法。我从别人那里复制来了 Turbo Pascal 7.0（那时网络并不发达），开始研究起来。由于先学

的是 C 语言，所以我刚开始学习 Pascal 时感到有些不习惯：赋值不是“=”而是“:=”，简洁的花括号变成了累赘的 begin 和 end，if 之后要加个 then，而且和 else 之间不允许写分号……但很快我就发现，这些都不是本质问题。在编写竞赛题的程序时，我并不会用到太多的高级语法。Pascal 的语法虽然稍微啰嗦一点，但总体来说是很清晰的。就这样，我只花了不到一天时间就把语法习惯从 C 转到了 Pascal，剩下的知识就是在不断编程中慢慢地学习和熟练——**学习 C 语言的过程是痛苦的，但收益也是巨大的，“轻松转到 Pascal”只是其中一个小小的例子。**

我学习计算机，从一开始就不是为了参加竞赛，因此，在编写算法程序之余，我几乎总是使用熟悉的 C 语言，有时还会用点汇编，并没有觉得有何不妥。随着编写应用程序的经验逐渐丰富，我开始庆幸自己先学的是 C 语言——在我购买的各类技术书籍中，几乎全部使用的是 C 语言而不是 Pascal 语言，尽管偶尔有用 Delphi 的文章，但这种语言似乎除了构建漂亮的界面比较方便之外，并没有太多的“技术含量”。**我始终保持着对 C 语言的熟悉，而事实证明这对我的职业生涯发挥了巨大的作用。**

中学竞赛和教学

在大学里参加完 ACM/ICPC 世界总决赛之后（当时 ACM/ICPC 还可以用 Pascal，现在已经不能用了），我再也没有用 Pascal 语言做过一件“正经事”（只是偶尔用它给一些只懂 Pascal 的孩子讲课）。后来我才知道，国际信息学奥林匹克系列竞赛是为数不多的几个允许使用 Pascal 语言的比赛之一。IT 公司举办的商业比赛往往只允许用 C/C++或 Java、C#、Python 等该公司使用较为频繁的语言（顺便说一句，C 语言学好以后，读者便有了坚实的基础去学习上述其他语言），而在做一些以算法为核心的项目时，一般来说也不能用 Pascal 语言——你的算法程序必须和已有的系统集成，而这个“现有系统”很少是用 Pascal 写成的。为什么还有那么多中学生非要用这个“以后几乎再也用不着”的语言呢？

于是，我开始在中学竞赛中推广 C 语言。这并不是说我希望废除 Pascal 语言（事实上，我希望保留它），而是希望学生多一个选择，毕竟并不是每个参加信息学竞赛的学生都将走入 IT 界。但如果简单地因为“C 语言难学难用，竞赛中还容易碰到诸多问题”就放弃学习 C 语言，我想是很遗憾的。

然而，推广的道路是曲折的。作为五大学科竞赛（数学、物理、化学、生物、信息学）中唯一一门高考中没有的“特殊竞赛”，学生、教师、家长所走的道路要比其他竞赛要艰辛得多。

第一，数理化竞赛中所学的知识，多是大学本科要学习的，只不过是提前灌输给高中生，但信息学竞赛中所涉及的很多东西甚至连本科都不会学到，即使学到了，也只是“简单了解即可”，和“满足竞赛的要求”有着天壤之别，这极大地增加了中学生学习算法和编程的难度。

第二，学科发展速度快。辅导信息学竞赛的教师常常有这样的感觉：必须不停地学习学习再学习，否则很容易跟不上“潮流”。事实上，学术上的研究成果常常在短短几年之内就体现在竞赛中。

第三，质量要求高。想法再伟大，如果无法在比赛时间之内把它变成实际可运行的程序，那么所有的心血都将是白费。数学竞赛中有可能在比赛结束前 15 分钟找到突破口并在交卷前一瞬间把解法写完——就算有漏洞，还有部分分数呢；但在信息学竞赛中，想到正确解法却 5 个小时都写不完程序的现象并不罕见。连程序都写不完当然就是 0 分，即使程序写完了，如果存在关键漏洞，往往还是 0 分。这不难理解——如果用这个程序控制人造卫星发射，难道当卫星爆炸之后你还可以向人炫耀说："除了有一个加号被我粗心写成减号从而引起爆炸之外，这个卫星的发射程序几乎是完美的。"

在这样的情况下，让学生和教师放弃自己熟悉的 Pascal 语言，转向既难学又容易出错的 C 语言确实难为他们了——尤其是在 C 语言资料如此缺乏的情况下。等一下！C 语言资料缺乏？难道市面上不是遍地都是 C 语言教材吗？对，C 语言教材多，但和算法竞赛相结合的却几乎没有。**不要以为语言入门以后就能轻易地写出算法程序**（这甚至是很多 IT 工程师的误区），多数初学者都需要详细的代码才能透彻地理解算法，只了解算法原理和步骤是远远不够的。

大家都知道，编程需要大量的练习，只看只听是不够的。反过来，如果只是盲目练习，不看不听也是不明智的。本书的目标很明确——提供算法竞赛入门所必需的一切"看"的蓝本。有效的"听"要靠教师的辛勤劳动，而有效的"练"则要靠学生自己。当然，就算是最简单的"看"，也是大有学问的。不同的读者，往往能看到不同的深度。请把本书理解为"蓝本"。没有一本教材能不加修改而适用于各种年龄层次、不同学习习惯和悟性的学生，本书也不例外。我喜欢以人为本，因材施教，不推荐按照本书的内容和顺序填鸭式地教给学生。

内容安排

前面花了大量篇幅讨论了语言，但语言毕竟只是算法竞赛的工具——尽管这个工具非常重要，却不是核心。正如前面所讲，算法竞赛的核心是算法。我曾考虑过把 C 语言和算法分开讲解，一本书讲语言，另一本书讲基础算法。但后来我发现，其实二者难以分开。

首先，语言部分的内容选择很难。如果把 C 语言的方方面面全部讲到，篇幅肯定不短，而且和市面上已有的 C 语言教材基本上不存在区别；如果只是提纲挈领地讲解核心语法，并只举一些最为初级的例子，看完后读者将会处于我当初 3 天看完《C 语言三日通》后的状态——以为自己都懂了，慢慢才发现自己学的都是"玩具"，真正关键、实用的东西全都不懂。

其次，算法的实现常常要求程序员对语言熟练掌握，而算法书往往对程序实现避而不谈。即使少数书籍给出了详细代码，但代码往往十分冗长，不适合用在算法竞赛中。更重要的是，这些书籍对算法实现中的小技巧和常见错误少有涉及，所有的经验教训都需要读者自己从头积累。换句话说，传统的语言书和算法之间存在着不小的鸿沟。

基于上述问题，本书采取一种语言和算法结合的方法，把内容分为如下 3 部分：

- 第 1 部分是语言篇（第 1～4 章），纯粹介绍语言，几乎不涉及算法，但逐步引入一些工程性的东西，如测试、断言、伪代码和迭代开发等。

- 第 2 部分是算法篇（第 5～8 章），在介绍算法的同时继续强化语言，补充了第 1 部分没有涉及的语言特性，如位运算、动态内存管理等，并延续第一部分的风格，在需要时引入更多的思想和技巧。学习完前两部分的读者应当可以完成相当数量的练习题。
- 第 3 部分是竞赛篇（第 9～11 章），涉及竞赛中常用的其他知识点和技巧。和前两部分相比，第 3 部分涉及的内容更加广泛，其中还包括一些难以理解的“学术内容”，但其实这些才是算法的精髓。

本书最后有一个附录，介绍开发环境和开发方法，虽然它们和语言、算法的关系都不大，却往往能极大地影响选手的成绩。另外，本书讲解过程中所涉及的程序源代码可登录网站 http://www.tup.tsinghua.edu.cn/下载。

致谢

在真正动笔之前，我邀请了一些对本书有兴趣的朋友一起探讨本书的框架和内容，并请他们撰写了一定数量的文字，他们是赖笠源（语言技巧、字符串）、曹正（数学）、邓凯宁（递归、状态空间搜索）、汪箜（数据结构基础）、王文一（算法设计）、胡昊（动态规划）。尽管这些文字本身并没有在最终的书稿中出现，但我从他们的努力中获得了很多启发。北京大学的杨斐瞳完成了本书中大部分插图的绘制，清华大学的杨锐和林芝恒对本书进行了文字校对、题目整理等工作，在此一并感谢。

在本书构思和初稿写作阶段，很多在一线教学的老师给我提出了有益的意见和建议，他们是绵阳南山中学的叶诗富老师、绵阳中学的曾贵胜老师、成都七中的张君亮老师、成都石室中学的文仲友老师、成都大弯中学的李植武老师、温州中学的舒春平老师，以及我的母校——重庆外国语学校的官兵老师等。

本书的习题主要来自 UVa 在线评测系统，感谢 Miguel Revilla 教授和 Carlos M. Casas Cuadrado 的大力支持。

最后，要特别感谢清华大学出版社的朱英彪编辑，与他的合作非常轻松、愉快。没有他的建议和鼓励，或许我无法鼓起勇气把“算法艺术与信息学竞赛系列”以丛书的全新面貌展现给读者。

刘汝佳

目　录

第1部分　语　言　篇

第1部分 语 言 篇

第1章 程序设计入门

学习目标

- ☑ 熟悉C语言程序的编译和运行
- ☑ 学会编程计算并输出常见的算术表达式的结果
- ☑ 掌握整数和浮点数的含义和输出方法
- ☑ 掌握数学函数的使用方法
- ☑ 初步了解变量的含义
- ☑ 掌握整数和浮点数变量的声明方法
- ☑ 掌握整数和浮点数变量的读入方法
- ☑ 掌握变量交换的三变量法
- ☑ 理解算法竞赛中的程序三步曲：输入、计算、输出
- ☑ 记住算法竞赛的目标及其对程序的要求

计算机速度快，很适合做计算和逻辑判断工作。本章首先介绍顺序结构程序设计，其基本思路是：把需要计算机完成的工作分成若干个步骤，然后依次让计算机执行。注意这里的“依次”二字——步骤之间是有先后顺序的。这部分的重点在于计算。

接下来介绍分支结构程序设计，用到了逻辑判断，根据不同情况执行不同语句。本章内容不复杂，但是不容忽视。

注意：编程不是看会的，也不是听会的，而是练会的，所以应尽量在计算机旁阅读本书，以便把书中的程序输入到计算机中进行调试，顺便再做做上机练习。千万不要图快——如果没有足够的时间用来实践，那么学得快，忘得也快。

1.1 算术表达式

计算机的“本职”工作是计算，因此下面先从算术运算入手，看看如何用计算机进行复杂的计算。

程序 1-1 计算并输出 1+2 的值

```
#include<stdio.h>
int main()
{
```

```
    printf("%d\n", 1+2);
    return 0;
}
```

这是一段简单的程序，用于计算 1+2 的值，并把结果输出到屏幕。如果你不知道如何编译并运行这段程序，可阅读附录或向指导教师求助。

即使你不明白上述程序除了“1+2”之外的其他内容，仍然可以进行以下探索：试着把“1+2”改成其他东西，而不要去修改那些并不明白的代码——它们看上去工作情况良好。

下面让我们做 4 个实验：

实验 1：修改程序 1-1，输出 3-4 的结果

实验 2：修改程序 1-1，输出 5×6 的结果

实验 3：修改程序 1-1，输出 8÷4 的结果

实验 4：修改程序 1-1，输出 8÷5 的结果

直接把“1+2”替换成“3+4”即可顺利解决实验 1，但读者很快就会发现：无法在键盘上找到乘号和除号。解决方法是：用星号“*”代替乘号，而用正斜线“/”代替除号。这样，4 个实验都顺利完成了。

等一下！实验 4 的输出结果居然是 1，而不是正确答案 1.6。这是怎么回事？计算机出问题了吗？计算机没有出问题，问题出在程序上：这段程序的实际含义并非和我们所想的一致。

在 C 语言中，8/5 的确切含义是 8 除以 5 所得商值的整数部分。同样地，（-8）/5 的值是-1，不信可以自己试试。那么如果非要得到 8÷5=1.6 的结果怎么办？下面是完整的程序。

程序 1-2　计算并输出 8/5 的值，保留小数点后 1 位

```
#include<stdio.h>
int main()
{
    printf("%.1lf\n", 8.0/5.0);
    return 0;
}
```

注意，百分号后面是个小数点，然后是数字 1，再然后是小写字母 l，最后是小写字母 f，千万不能打错，包括大小写——在 C 语言中，大写和小写字母代表的含义是不同的。

再来做 3 个实验：

实验 5：把%.1lf 中的数字 1 改成 2，结果如何？能猜想出“1”的确切意思吗？如果把小数点和 1 都删除，%lf 的含义是什么？

实验 6：字符串%.1lf 不变，把 8.0/5.0 改成原来的 8/5，结果如何？

实验 7：字符串%.1lf 改成原来的%d，8.0/5.0 不变，结果如何？

实验 5 并不难解决，但实验 6 和实验 7 的答案就很难简单解释了——真正原因涉及整数和浮点数编码，相信多数初学者对此都不感兴趣。原因并不重要，重要的是规范：根据规范做事情，则一切尽在掌握中。

提示 1-1：整数值用%d 输出，实数用%lf 输出[①]。

这里的“整数值”指的是 1+2、8/5 这样“整数之间的运算”。只要运算符的两边都是整数，则运算结果也会是整数。正因为这样，8/5 的值才是 1，而不是 1.6。

8.0 和 5.0 被看作是“实数”，或者说得更专业一点，叫“浮点数”。浮点数之间的运算结果是浮点数，因此 8.0/5.0=1.6 也是浮点数。注意，这里的运算符“/”其实是“多面手”，它既可以拿来做整数除法，又可以拿来做浮点数除法。

提示 1-2：整数/整数=整数，浮点数/浮点数=浮点数。

这条规则同样适用于加法、减法和乘法，不过没有除法这么容易出错——毕竟整数乘以整数的结果本来就是整数。

算术表达式可以和数学表达式一样复杂，例如：

程序 1-3　复杂的表达式计算

```
#include<stdio.h>
#include<math.h>
int main()
{
  printf("%.8lf\n", 1+2*sqrt(3)/(5-0.1));
  return 0;
}
```

相信读者不难把它翻译成数学表达式：$1+\frac{2\sqrt{3}}{5-0.1}$。尽管如此，读者可能还是有一些疑惑：5-0.1 的值是什么？“整数-浮点数”是整数还是浮点数？另外，多出来的#include<math.h>是做什么用的？

第 1 个问题相信读者能够“猜到”结果：整数-浮点数=浮点数。但其实这个说法并不准确。确切的说法是：整数先“变”成浮点数，然后浮点数-浮点数=浮点数。

第 2 个问题的答案是：因为程序 1-3 中用到了数学函数 sqrt。数学函数 sqrt(*x*)的作用是计算 *x* 的算术平方根（若不信，可输出 sqrt(9.0)的值试试）。一般来说，只要在程序中用到了数学函数，就需要在程序最开始的地方包含头文件 math.h，并在编译时连接数学库。如果你不知道如何编译并运行这段程序，可阅读附录或向指导教师求助。

1.2　变量及其输入

1.1 节的程序虽好，但有一个遗憾：计算的数据是事先确定的。为了计算 1+2 和 2+3，我们不得不编写两个程序。可不可以让程序读取键盘输入，并根据输入内容计算结果呢？答案是肯定的。程序如下：

[①] 从真正的语言规范来说，这个说法也有一点小问题，不过在算法竞赛中可以完全忽略。

程序 1-4　A+B 问题

```
#include<stdio.h>
int main()
{
  int a, b;
  scanf("%d%d", &a, &b);
  printf("%d\n", a+b);
  return 0;
}
```

该程序比 1.1 节的复杂了许多。简单地说，第一条语句“int a, b”声明了两个整型（即整数类型）变量 a 和 b，然后读取键盘输入，并放到 a 和 b 中。注意 a 和 b 前面的&符号——千万不要漏掉，不信可以试试[①]。

现在，你的程序已经读入了两个整数，可以在表达式中自由使用它们，就好比使用 12、597 这样的常数。这样，表达式 a+b 就不难理解了。

提示 1-3：scanf 中的占位符和变量的数据类型应一一对应，且每个变量前需要&符号。

可以暂时把变量理解成“存放值的场所”，或者形象地认为每个变量都是一个盒子、瓶子或箱子。在 C 语言中，变量有自己的数据类型，例如 int 型变量存放整数值，而 double 型变量存放浮点数值（专业的说法是“双精度”浮点数）。如果硬要把浮点数值塞给一个 int 型变量，将会丢失部分信息——我们不推荐这样做。

下面来看一个复杂一点的例子。

例题 1-1　圆柱体的表面积

输入底面半径 r 和高 h，输出圆柱体的表面积，保留 3 位小数，格式见样例。

样例输入：3.5　9

样例输出：Area = 274.889

【分析】

圆柱体的表面积由 3 部分组成：上底面积、下底面积和侧面积。由于上下底面积相等，完整的公式可以写成：表面积=底面积×2+侧面积。根据平面几何知识，底面积=πR^2，侧面积=$2\pi rh$。不难写出完整程序：

程序 1-5　圆柱体的表面积

```
#include<stdio.h>
#include<math.h>
int main()
{
  const double pi = 4.0 * atan(1.0);
  double r, h, s1, s2, s;
```

[①] 在学习编程时，“明知故犯”是有益的：起码你知道了错误时的现象。这样，当你真的不小心犯错时，可以通过现象猜测到可能的原因。

```
    scanf("%lf%lf", &r, &h);
    s1 = pi*r*r;
    s2 = 2*pi*r*h;
    s = s1*2.0 + s2;
    printf("Area = %.3lf\n", s)
    return 0;
}
```

这是本书中第一个完整的“竞赛题目”，因为和正规比赛一样，题目中包含着输入输出格式规定，还有样例数据。大多数的算法竞赛包含如下一些相同的“游戏规则”。

首先，选手程序的执行是自动完成的，没有人工干预。不要在用户输入之前打印提示信息（例如“Please input n:”），这不仅不会为程序赢得更高的“界面友好分”，反而会让程序丢掉大量的（甚至所有的）分数——这些提示信息会被当作输出数据的一部分。例如刚才的程序如果加上了“友好提示”，输出信息将变成：

```
Please input n:
Area = 274.889
```

比标准答案多了整整一行！

其次，不要让程序“按任意键退出”（例如调用 system(“pause”)，或者加一个多余的 getchar()），因为不会有人来“按任意键”的。不少早期的 C 语言教材会建议在程序的最后加这样一条语句来“观察输出结果”，但注意千万不要在算法竞赛中这样做。

提示 1-4：在算法竞赛中，输入前不要打印提示信息。输出完毕后应立即终止程序，不要等待用户按键，因为输入输出过程都是自动的，没有人工干预。

在一般情况下，你的程序不能直接读取键盘和控制屏幕：不要在算法竞赛中使用 getch()、getche()、gotoxy()、clrscr()（早期的教材中可能会介绍这些函数）。

提示 1-5：在算法竞赛中不要使用头文件 conio.h，包括 getch()、clrscr()等函数。

最后，最容易忽略的是输出的格式：在很多情况下，输出格式是非常严格的——多一个或者少一个字符都是不可以的！

提示 1-6：在算法竞赛中，每行输出均应以回车符结束，包括最后一行。除非特别说明，每行的行首不应有空格，但行末通常可以有多余空格。另外，输出的每两个数或者字符串之间应以单个空格隔开。

总结一下，算法竞赛的程序应当只做 3 件事情：读入数据、计算结果、打印输出。不要打印提示信息，不要在打印输出后“暂停程序”，更不要尝试画图、访问网络等与算法无关的任务。

回到刚才的程序，它多了几个新东西。首先是“const double pi = 4.0 * atan(1.0);”。这里也声明了一个叫 pi 的“符号”，但是 const 关键字表明它的值是不可以改变的——pi 是一个真正的数学常数[①]。

[①] 有的读者可能会用 math.h 中定义的常量 M_PI，但其实这个常数不是 ANSI C 标准的。不信的话用 gcc-ansi 编译试试。

提示 1-7：尽量用 const 关键字声明常数。

接下来是 s1 = pi * r * r。这条语句应该如何理解呢？“s1 等于 pi*r*r”吗？并不是这样的。不信，你把它换成“pi * r * r = s1”试试，编译器会给出错误信息：invalid lvalue in assignment。如果这条语句真的是“二者相等”的意思，为何不允许反着写呢？

事实上，这条语句的学术说法是赋值（assignment），它不是一个描述，而是一个动作。它的确切含义是：先把“等号”右边的值算出来，然后塞到左边的变量中。注意，变量是“喜新厌旧”的，即新的值将覆盖原来的值，一旦被赋了新的值，变量中原来的值就丢失了。

提示 1-8：赋值是个动作，先计算右边的值，再赋给左边的变量，覆盖它原来的值。

最后是那个“Area = %.3lf\n”，它的用法很容易被猜到：只有以%开头的部分才会被后面的值替换掉，其他部分原样输出。

提示 1-9：printf 的格式字符串中可以包含其他可打印符号，打印时原样输出。

1.3 顺序结构程序设计

例题 1-2 三位数反转

输入一个三位数，分离出它的百位、十位和个位，反转后输出。

样例输入：127

样例输出：721

【分析】

首先将三位数读入变量 n，然后进行分离。百位等于 n/100（注意这里取的是商的整数部分），十位等于 n/10%10（这里的%是取余数操作），个位等于 n%10。程序如下：

程序 1-6 三位数反转（1）

```
#include<stdio.h>
int main()
{
  int n;
  scanf("%d", &n);
  printf("%d%d%d\n", n%10, n/10%10, n/100);
  return 0;
}
```

此题有一个没有说清楚的细节，即：如果个位是 0，反转后应该输出吗？例如输入是 520，输出是 025 还是 25？如果在算法竞赛中遇到这样的问题，可向监考人员询问[①]。但是在这里，两种情况的处理方法都应学会。

[①] 如果是网络竞赛，还可以向组织者发信，在论坛中提问或者拨打热线电话。

提示 1-10：算法竞赛的题目应当是严密的，各种情况下的输出均应有严格规定。如果在比赛中发现题目有漏洞，应向相关人员询问，而尽量不要自己随意假定。

上面的程序输出 025，但要改成输出 25 似乎会比较麻烦——我们必须判断 n%10 是不是 0，但目前还没有学到“根据不同情况执行不同指令”（分支结构程序设计是 1.4 节的主题）。

一个解决方法是在输出前把结果储存在变量 m 中。这样，直接用%d 格式输出 m，将输出 25。要输出 025 也很容易，把输出格式变为%03d 即可。

程序 1-7　三位数反转（2）

```
#include<stdio.h>
int main()
{
  int n, m;
  scanf("%d", &n);
  m = (n%10)*100 + (n/10%10)*10 + (n/100);
  printf("%03d\n", m);
  return 0;
}
```

例题 1-3　交换变量

输入两个整数 *a* 和 *b*，交换二者的值，然后输出。

样例输入：824　16

样例输出：16　824

【分析】

按照题目所说，先把输入存入变量 a 和 b，然后交换。如何交换两个变量呢？最经典的方法是三变量法：

程序 1-8　变量交换（1）

```
#include<stdio.h>
int main()
{
  int a, b, t;
  scanf("%d%d", &a, &b);
  t = a;
  a = b;
  b = t;
  printf("%d %d\n", a, b);
  return 0;
}
```

可以将这种方法形象地比喻成将一瓶酱油和一瓶醋借助一个空瓶子进行交换：先把酱油倒入空瓶，然后将醋倒进原来的酱油瓶中，最后把酱油从辅助的瓶子中倒入原来的醋瓶子里。这样的比喻虽然形象，但是初学者应当注意它和真正的变量交换的区别。

借助一个空瓶子的目的是：避免把醋直接倒入酱油瓶子——直接倒进去，二者混合以后，将很难分开。在C语言中，如果直接进行赋值a=b，则原来a的值（酱油）将会被新值（醋）覆盖，而不是混合在一起。

当酱油被倒入空瓶以后，原来的酱油瓶就变空了，这样才能装醋。但在C语言中，进行赋值t=a后，a的值不变，它只是把值拷贝（即复制）给了变量t而已，自身并不会变化。尽管a的值马上就会被改写，但是从原理上看，t=a的过程和“倒酱油”的过程有着本质区别。

提示1-11：赋值a=b之后，变量a原来的值被覆盖，而b的值不变。

另一个方法没有借助任何变量，但是较难理解：

程序1-9 变量交换（2）

```
#include<stdio.h>
int main()
{
  int a, b;
  scanf("%d%d", &a, &b);
  a = a + b;
  b = a - b;
  a = a - b;
  printf("%d %d\n", a, b);
  return 0;
}
```

这次就不太方便用倒酱油做比喻了：硬着头皮把醋倒在酱油瓶子里，然后分离出酱油倒回醋瓶子？比较理性的方法是手工模拟这段程序，看看每条语句执行后的情况。

在顺序结构程序中，程序一条一条依次执行。为了避免值和变量名混淆，假定用户输入的是a_0和b_0，因此scanf语句执行完后$a=a_0, b=b_0$。

执行完a=a+b后：$a=a_0+b_0, b=b_0$。

执行完b=a-b后：$a=a_0+b_0, b=a_0$。

执行完a=a-b后：$a=b_0, b=a_0$。

这样就不难理解两个变量是如何交换的了。这个方法看起来很好（少用一个变量），但实际上很少使用，因为它的适用范围很窄：只有定义了加减法的数据类型才能这么做[①]。事实上，笔者并不推荐读者采用这样的技巧实现变量交换：三变量法已经足够好了，这个例子只是帮助读者提高程序阅读能力。

提示1-12：交换两个变量的三变量法适用范围广，推荐使用。

那么是不是说，三变量法是解决本题的最佳途径了呢？答案是否定的。多数算法竞赛采用黑盒测试，即只考查程序解决问题的能力，而不关心它采用的什么方法。对于本题而言，最合适的程序莫过于：

① 这个思想还有一个“变种”：用异或运算^代替加法和减法，它还可以进一步简写成a^=b^=a^=b。但不建议使用。

程序 1-10　变量交换（3）

```
#include<stdio.h>
int main()
{
  int a, b;
  scanf("%d%d", &a, &b);
  printf("%d %d\n", b, a);
  return 0;
}
```

换句话说，我们的目标是解决问题，而不是为了写程序而写程序，同时应保持简单（Keep It Simple and Stupid，KISS），而不是自己创造条件去展示编程技巧。

提示 1-13：算法竞赛是在比谁能更好地解决问题，而不是在比谁写的程序看上去更高级。

1.4　分支结构程序设计

例题 1-4　鸡兔同笼

已知鸡和兔的总数量为 n，总腿数为 m。输入 n 和 m，依次输出鸡的数目和兔的数目。如果无解，则输出“No answer”（不要引号）。

样例输入：14　32

样例输出：12　2

样例输入：10　16

样例输出：No answer

【分析】

设鸡有 a 只，兔有 b 只，则 $a+b=n$，$2a+4b=m$，联立解得 $a=(4n-m)/2$，$b=n-a$。在什么情况下此解“不算数”呢？首先，a 和 b 都是整数；其次，a 和 b 必须是非负的。可以通过下面的程序判断：

程序 1-11　鸡兔同笼

```
#include<stdio.h>
int main()
{
  int a, b, n, m;
  scanf("%d%d", &n, &m);
  a = (4*n-m)/2;
  b = n-a;
  if(m % 2 == 1 || a < 0 || b < 0)
    printf("No answer\n");
  else
```

```
    printf("%d %d\n", a, b);
  return 0;
}
```

上面的程序用到了 if 语句，它的一般格式是：

```
if(条件)
    语句1;
else
    语句2;
```

注意语句 1 和语句 2 后面的分号，以及 if 后面的括号。“条件”是一个表达式，当该表达式的值为“真”时执行语句 1，否则执行语句 2。另外，“else 语句 2”这个部分是可以省略的。语句 1 和语句 2 前面的空行是为了让程序更加美观，并不是必需的，但强烈推荐读者使用。

提示 1-14：if 语句的基本格式为：if(条件) 语句 1; else 语句 2。

换句话说，m%2==1 || a < 0 || b < 0 是一个表达式，它的字面意思是“m 是奇数，或者 a 小于 0，或者 b 小于 0”。这句话可能正确，也可能错误。因此这个表达式的值可能为真，也可能为假，取决于 m、a 和 b 的具体数值。

这样的表达式称为逻辑表达式。和算术表达式类似，逻辑表达式也由运算符和值构成，例如“||”运算符称为“逻辑或”，a || b 表示 a 为真，或者 b 为真。换句话说，a 和 b 只要有一个为真，a || b 就为真；如果 a 和 b 都为真，则 a || b 也为真。

提示 1-15：if 语句的条件是一个逻辑表达式，它的值可能为真，也可能为假。

细心的读者也许发现了，如果 a 为真，则无论 b 的值如何，a || b 均为真。换句话说，一旦发现 a 为真，就不必计算 b 的值。C 语言正是采取了这样的策略，称为短路(short-circuit)。也许你会觉得，用短路的方法计算逻辑表达式的唯一优点是速度更快，但其实并不是这样，稍后我们会通过几个例子予以证实。

提示 1-16：C 语言中的逻辑运算符都是短路运算符。一旦能够确定整个表达式的值，就不再继续计算。

例题 1-5　三整数排序

输入 3 个整数，从小到大排序后输出。

样例输入：20 7 33

样例输出：7 20 33

【分析】

a、b、c 3 个数一共只有 6 种可能的顺序：abc、acb、bac、bca、cab、cba，所以最简单的思路是使用 6 条 if 语句。

程序 1-12　三整数排序（1）（错误）

```
#include<stdio.h>
int main()
```

```
{
  int a, b, c;
  scanf("%d%d%d", &a, &b, &c);
  if(a < b && b < c)printf("%d %d %d\n", a, b, c);
  if(a < c && c < b)printf("%d %d %d\n", a, c, b);
  if(b < a && a < c)printf("%d %d %d\n", b, a, c);
  if(b < c && c < a)printf("%d %d %d\n", b, c, a);
  if(c < a && a < b)printf("%d %d %d\n", c, a, b);
  if(c < b && b < a)printf("%d %d %d\n", c, b, a);
  return 0;
}
```

上述程序看上去没有错误，而且能通过题目中给出的样例，但可惜有缺陷：输入 1 1 1 将得不到任何输出！这个例子告诉我们：即使通过了题目中给出的样例，程序仍然可能存在问题。

提示 1-17：算法竞赛的目标是编程对任意输入均得到正确的结果，而不仅是样例数据。

稍微修改一下：把所有的小于符号“<”改成小于等于符号“<=”（在一个小于号后添加一个等号）。这下总可以了吧？很遗憾，还是不行。对于“1 1 1”，6 种情况全部符合，程序一共输出了 6 次 1 1 1。

一种解决方案是人为地让 6 种情况没有交叉：把所有的 if 改成 else if。

程序 1-13　三整数排序（2）

```
#include<stdio.h>
int main()
{
  int a, b, c;
  scanf("%d%d%d", &a, &b, &c);
  if(a <= b && b <= c) printf("%d %d %d\n", a, b, c);
  else if(a <= c && c <= b) printf("%d %d %d\n", a, c, b);
  else if(b <= a && a <= c) printf("%d %d %d\n", b, a, c);
  else if(b <= c && c <= a) printf("%d %d %d\n", b, c, a);
  else if(c <= a && a <= b) printf("%d %d %d\n", c, a, b);
  else if(c <= b && b <= a) printf("%d %d %d\n", c, b, a);
  return 0;
}
```

最后一条语句还可以简化成单独的“else”（想一想，为什么），不过，幸好程序正确了。

提示 1-18：如果有多个并列、情况不交叉的条件需要一一处理，可以用 else if 语句。

另一种思路是把 a、b、c 这 3 个变量本身改成 a≤b≤c 的形式。首先检查 a 和 b 的值，如果 a>b，则交换 a 和 b（利用前面讲过的三变量交换法）；接下来检查 a 和 c，最后检查 b 和 c，程序如下：

程序 1-14　三整数排序（3）

```
#include<stdio.h>
int main()
{
  int a, b, c, t;
  scanf("%d%d%d", &a, &b, &c);
  if(a > b) { t = a; a = b; b = t; }
  if(a > c) { t = a; a = c; c = t; }
  if(b > c) { t = b; b = c; c = t; }
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

为什么这样做是对的呢？因为经过第一次检查以后，必然有 a≤b，而第二次检查以后 a≤c。由于第二次检查以后 a 的值不会变大，所以 a≤b 依然成立。换句话说，a 已经是 3 个数中的最小值。接下来只需检查 b 和 c 的顺序即可。

一个很自然的问题产生了：其他检查顺序是否也可以呢？例如先(a,b)，然后(b,c)，最后(a,c)？这个问题留给读者思考。提示：上机实验。

注意上面的程序中唯一的新东西：花括号。前面讲过，if 语句中有一个“语句 1”和可选的“语句 2”，且都要以分号结尾。有一种特殊的“语句”是由花括号括起来的多条语句。这多条语句可以作为一个整体，充当 if 语句中的“语句 1”或“语句 2”，且后面不需要加分号。当然，当 if 语句的条件满足时，这些语句依然会按顺序逐条执行，和普通的顺序结构一样。

提示 1-19：可以用花括号把若干条语句组合成一个整体。这些语句仍然按顺序执行。

最后一种思路再次利用了“问题求解”这一目标——它实际上并没有真的进行排序：求出了最小值和最大值，中间值是可以计算出来的。

程序 1-15　三整数排序（4）

```
#include<stdio.h>
int main()
{
  int a, b, c, x, y, z;
  scanf("%d%d%d", &a, &b, &c);
  x = a; if(b < x) x = b; if(c < x) x = c;
  z = a; if(b > z) z = b; if(c > z) z = c;
  y = a + b + c - x - z;
  printf("%d %d %d\n", x, y, z);
  return 0;
}
```

注意程序中包含的“当前最小值”x 和“当前最大值”z。它们都初始化为 a，但是随着

“比较”操作的进行而慢慢更新，最后变成真正的最小值和最大值。这个技巧极为实用。

提示 1-20：在难以一次性求出最后结果时，可以用变量储存“临时结果”，从而逐步更新。

1.5　小结与习题

经过前几个小节的学习，相信读者已经初步了解顺序结构程序设计和分支结构程序设计的核心概念和方法，然而对这些知识进行总结，并且完成适当的练习是很必要的。

首先，我们给出一些实验，旨在加深读者对本章内容的认识。

1.5.1　数据类型实验

实验 A1：表达式 11111*11111 的值是多少？把 5 个 1 改成 6 个 1 呢？9 个 1 呢？

实验 A2：把实验 A1 中的所有数换成浮点数，结果如何？

实验 A3：表达式 sqrt(-10)的值是多少？尝试用各种方式输出。在计算的过程中系统会报错吗？

实验 A4：表达式 1.0/0.0、0.0/0.0 的值是多少？尝试用各种方式输出。在计算的过程中系统会报错吗？

实验 A5：表达式 1/0 的值是多少？在计算的过程中系统会报错吗？

你不必解释背后的原因，但需注意现象。

1.5.2　scanf 输入格式实验

如果用语句 scanf(“%d%d”, &a, &b)来输入两个数，那么这两个数应以怎样的格式输入呢？（提示：可以在输入之后用 printf 打印出来，看看打印的结果是否和输入一致。）

实验 B1：在同一行中输入 12 和 2，并以空格分隔，是否得到了预期的结果？

实验 B2：在不同的两行中输入 12 和 2，是否得到了预期的结果？

实验 B3：在实验 B1 和 B2 中，在 12 和 2 的前面和后面加入大量的空格或水平制表符（TAB），甚至插入一些空行。

实验 B4：把 2 换成字符 s，重复实验 B1～B3。

同样，你不必解释背后的原因，但需注意现象。

1.5.3　printf 语句输出实验

和上面的实验不同，除了注意现象之外，你还需要找到问题的解决方案。

实验 C1：仅用一条 printf 语句，打印 1+2 和 3+4 的值，用两个空行隔开。

实验 C2：试着把%d 中的两个字符（百分号和小写字母 d）输出到屏幕。

实验 C3：试着把\n 中的两个字符（反斜线和小写字母 n）输出到屏幕。

实验 C4：像 C2、C3 那样也需要“特殊方法”才能输出的东西还有哪些？哪些是 printf 函数引起的问题，哪些不是？

1.5.4 测测你的实践能力

如何用实验方法确定以下问题的答案？注意，不要查书，也不要在网上搜索答案，必须亲手尝试——实践精神是极其重要的。

问题 1：int 型整数的最小值和最大值是多少？（需要精确值）

问题 2：double 型浮点数能精确到多少位小数？或者，这个问题本身值得商榷？

问题 3：double 型浮点数最大正数值和最小正数值分别是多少（不必特别精确）？

问题 4：逻辑运算符号&&、||和!（它表示逻辑非）的相对优先级是怎样的？也就是说，a&&b||c 应理解成(a&&b)||c 还是 a&&(b||c)，或者随便怎么理解都可以？

问题 5：if(a) if(b) x++; else y++的确切含义是什么？这个 else 应和哪个 if 配套？有没有办法明确表达出配套方法，以避免初学者为之困惑？

1.5.5 小结

对于不少读者来说，本章的内容都是直观、容易理解的，但这并不意味着所有人都能很快地掌握所有内容。相反，一些勤于思考的人反而更容易对一些常人没有注意到的细节问题产生疑惑。对此，笔者提出如下两条建议。

一是重视实验。哪怕不理解背后的道理，至少要清楚现象。例如，读者若亲自完成了本章的探索性实验和上机练习，一定会对整数范围、浮点数范围和精度、特殊的浮点值、scanf 对空格、TAB 和回车符的过滤、三角函数使用弧度而非角度等知识点有一定的了解。这些东西都没有必要死记硬背，但一定要学会实验的方法。这样即使编程时忘记了一些细节，手边又没有参考资料，也能轻松得出正确的结论。

二是学会模仿。本章始终没有介绍#include<stdio.h>语句的作用，但这丝毫不影响读者编写简单的程序。这看似是在鼓励读者“不求甚解”，但实为考虑到学习规律而作出的决策：初学者自学和理解能力不够，自信心也不够，不适合在动手之前被灌输大量的理论。如果初学者在一开始就被告知“stdio 是 standard I/O 的缩写，stdio.h 是一个头文件，它在 XXX 位置，包含了 XXX、XXX、XXX 等类型的函数，可以方便地完成 XXX、XXX、XXX 的任务；但其实这个头文件只是包含了这些函数的声明，还有一些宏定义，而真正的函数定义是在库中，编译时用不上，而在连接时……”，多数读者会茫然不知所云，甚至自信心会受到打击，对学习 C 语言失去兴趣。正确的处理方法是“抓住主要矛盾”——始终把学习、实验的焦点集中在最有趣的部分。如果直观的解决方案行得通，就不必追究其背后的机理。如果对一个东西不理解，就不要对其进行修改；如果非改不可，则应根据自己的直觉和猜测尝试各种改法，而不必过多地思考“为什么要这样”。

当然，这样的策略并不一定持续很久。当学生有一定的自学、研究能力之后，本书会在适当的时候解释一些重要的概念和原理，并引导学生寻找更多的资料进一步学习。要想把事情做好，必须学得透彻——但没有必要操之过急。

1.5.6　上机练习

程序设计是一门实践性很强的学科，读者应在继续学习之前确保下面的题目都能做出来。

习题 1-1　平均数（average）

输入 3 个整数，输出它们的平均值，保留 3 位小数。

习题 1-2　温度（temperature）

输入华式温度 f，输出对应的摄氏温度 c，保留 3 位小数。提示：$c=5(f-32)/9$。

习题 1-3　连续和（sum）

输入正整数 n，输出 $1+2+\cdots+n$ 的值。提示：目标是解决问题，而不是练习编程。

习题 1-4　正弦和余弦（sincos）

输入正整数 n（$n<360$），输出 n 度的正弦、余弦函数值。提示：使用数学函数。

习题 1-5　距离（distance）

输入 4 个浮点数 x_1, y_1, x_2, y_2，输出平面坐标系中点(x_1,y_1)到点(x_2,y_2)的距离。

习题 1-6　偶数（odd）

输入一个整数，判断它是否为偶数。如果是，则输出“yes”，否则输出“no”。提示：可以用多种方法判断。

习题 1-7　打折（discount）

一件衣服 95 元，若消费满 300 元，可打八五折。输入购买衣服件数，输出需要支付的金额（单位：元），保留两位小数。

习题 1-8　绝对值（abs）

输入一个浮点数，输出它的绝对值，保留两位小数。

习题 1-9　三角形（triangle）

输入三角形三边长度值（均为正整数），判断它是否能为直角三角形的三个边长。如果可以，则输出“yes”，如果不能，则输出“no”。如果根本无法构成三角形，则输出“not a triangle”。

习题 1-10　年份（year）

输入年份，判断是否为闰年。如果是，则输出“yes”，否则输出“no”。提示：简单地判断除以 4 的余数是不够的。

第 2 章　循环结构程序设计

学习目标

- ☑ 掌握 for 循环的使用方法
- ☑ 掌握 while 循环的使用方法
- ☑ 学会使用计数器和累加器
- ☑ 学会用输出中间结果的方法调试
- ☑ 学会用计时函数测试程序效率
- ☑ 学会用重定向的方式读写文件
- ☑ 学会用 fopen 的方式读写文件
- ☑ 了解算法竞赛对文件读写方式和命名的严格性
- ☑ 记住变量在赋值之前的值是不确定的
- ☑ 学会使用条件编译指示构建本地运行环境

第 1 章的程序虽然完善，但并没有发挥出计算机的优势。顺序结构程序自上到下只执行一遍，而分支结构中甚至有些语句可能一遍都执行不了。换句话说，为了让计算机执行大量操作，必须编写大量的语句。能不能只编写少量语句，就让计算机做大量的工作呢？这就是本章的主题。基本思想很简单：一条语句执行多次就可以了。但如何让这样的程序真正发挥作用，那并不是容易的事。

2.1　for 循环

考虑这样一个问题：打印 1, 2, 3,…, 10，每个占一行。本着“解决问题第一”的思想，很容易写出程序：10 条 printf 语句就可以了。或者也可以写一条，每个数后面加一个“\n”换行符。但如果把 10 改成 100 呢？1000 呢？甚至这个重复次数是可变的：“输入正整数 n，打印 1, 2, 3,…, n，每个占一行。”又怎么办呢？这时可以使用 for 循环。

程序 2-1　输出 1,2,3,…,*n* 的值

```
1  #include<stdio.h>
2  int main()
3  {
4    int i, n;
5    scanf("%d", &n);
6    for(i = 1; i <= n; i++)
7      printf("%d\n", i);
8    return 0;
9  }
```

暂时不用管细节，只要知道它是“让 i 依次等于 1, 2, 3,…, n，每次都执行 printf(“%d\n”, i);”就可以了。这个“依次”非常重要：程序运行结果一定是 1,2,3,…,n，而不是别的顺序。

提示 2-1：for 循环的格式为：for(初始化; 条件; 调整)　循环体;

在刚才的例子中，初始化是语句“i = 1”。前面已经讲过，它是一条赋值语句，含义是用 1 覆盖 i 原来的值。循环条件是“i <= n”，当循环条件满足时始终进行循环。调整方法是“i++”，它的含义和“i = i+1”相同——表示给 i 增加 1。循环体是语句“printf(“%d\n”, i);”，它就是计算机反复执行的内容。注意循环变量的妙用：尽管每次执行的语句相同，但是由于 i 的值不断变化，该语句的输出结果也是不断变化的。

提示 2-2：尽管 for 循环反复执行相同的语句，但这些语句每次的执行效果往往不同。

为了更深入地理解 for 循环，下面给出了程序 2-1 的执行过程：

当前行：5。scanf 请求键盘输入，假设输入 4。此时变量 n=4，i 的值随机，继续。
当前行：6。这是第一次执行到该语句，执行初始化语句，即 i=1。条件 i<=n 满足，继续。
当前行：7。由于 i=1，在屏幕输出 1 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=2，n=4，条件 i<=n 满足，继续。
当前行：7。由于 i=2，在屏幕输出 2 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=3，n=4，条件 i<=n 满足，继续。
当前行：7。由于 i=3，在屏幕输出 3 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=4，n=4，条件 i<=n 满足，继续。
当前行：7。由于 i=4，在屏幕输出 4 并换行。循环体结束，跳转回第 6 行。
当前行：6。先执行调整语句 i++，此时 i=5，n=4，条件 i<=n 不满足，跳出循环体。
当前行：8。程序结束。

这个执行过程对于理解 for 循环非常重要：语句是一条一条执行的。强烈建议教师在课堂上演示单步调试的方法，并打开 i 和 n 的 watch，以帮助学生掌握如何用实验验证上面所讲的执行过程。观察执行过程时应留意两个方面：“当前行”的跳转（在 IDE 中往往高亮显示），以及变量的变化。这二者也是编码、测试和调试的重点。根据实际情况，教师可以用 IDE（如 Code::Blocks）或者文本界面的 gdb 进行演示。gdb 的简明参考见附录。

提示 2-3：编写程序时，要特别留意“当前行”的跳转和变量的改变。

有了 for 循环，我们可以解决一些简单的问题。

例题 2-1　aabb

输出所有形如 aabb 的四位完全平方数（即前两位数字相等，后两位数字也相等）。

【分析】

分支和循环结合在一起时威力特别强大：我们枚举所有可能的 aabb，然后判断它们是否为完全平方数。注意，a 的范围是 1～9，但 b 可以是 0。主程序如下：

```
for(a = 1; a <= 9; a++)
  for(b = 0; b <= 9; b++)
    if(aabb 是完全平方数) printf("%d\n", aabb);
```

请注意，这里用到了循环的嵌套：for 循环的循环体自身又是一个循环。如果难以理解嵌套循环，可以用前面介绍的方法——在 IDE 或 gdb 中单步执行，观察“当前行”和循环变量 a 和 b 的变化过程。

上面的程序并不完整——“aabb 是完全平方数”是中文描述，而不是合法的 C 语言表达式，而 aabb 在 C 语言中也是另外一个变量，而不是把两个数字 a 和两个数字 b 拼在一起（C 语言中的变量名可以由多个字母组成）。但这个“程序”很容易理解，它甚至能让我们的思路更加清晰。

我们把这样“不是真正程序”的“代码”称为伪代码（pseudocode）。虽然有一些正规的伪代码定义，但在实际应用中，并不需要太拘泥于伪代码的格式。我们的主要目标是描述算法梗概，避开细节，启发思路。

提示 2-4： *不拘一格的使用伪代码来思考和描述算法是一种值得推荐的做法。*

写出伪代码之后，我们需要考虑如何把它变成真正的代码。上面的伪代码有两个“非法”的地方：完全平方数判定，以及 aabb 这个变量。后者相对比较容易一些：用另外一个变量 n = a*1100 + b*11 储存它即可。

提示 2-5： *把伪代码改写成代码时，一般先选择较为容易的任务来完成。*

接下来的问题就要困难一些了：如何判断 n 是否为完全平方数？第 1 章中，我们用过“开平方”函数，可以先求出它的平方根，然后看它是否为整数，即用一个 double 型变量 m 储存 sqrt(n)，然后判断 m 是否为整数。判断整数只需用它和它的整数部分比较即可。完整程序如下：

程序 2-2　7744 问题（1）

```
#include<stdio.h>
#include<math.h>
int main()
{
  int a, b, n;
  double m;
  for(a = 1; a <= 9; a++)
    for(b = 0; b <= 9; b++)
    {
      n = a*1100 + b*11;
      m = sqrt(n);
      if(floor(m+0.5) == m) printf("%d\n", n);
    }
  return 0;
}
```

函数 floor(x)返回 x 的整数部分，那么为什么不直接比较 floor(m)和 m 呢？原因在于：浮点数的运算（和函数）有可能存在误差——不是一定存在，但经常都会。

假设在经过大量计算后，由于误差的影响，整数 1 变成了 0.9999999999，floor 的结果会是 0 而不是 1！为了减小误差的影响，我们一般改成四舍五入，即 floor(x+0.5)。如果难

以理解，可以想象成在数轴上把一个单位区间往左移动 0.5 个单位的距离。floor(x)等于 1 的区间为[1,2)，而 floor(x+0.5)等于 1 的区间为[0.5, 1.5)。

提示 2-6：*浮点运算可能存在误差。在进行浮点数比较时，应考虑到浮点误差。*

另一个思路是枚举平方根 x，从而避免开平方操作。

程序 2-3　7744 问题（2）

```
#include<stdio.h>
int main()
{
  int x, n, hi, lo;
  for(x = 1; ; x++)
  {
    n = x * x;
    if(n < 1000) continue;
    if(n > 9999) break;
    hi = n / 100;
    lo = n % 100;
    if(hi/10 == hi%10 && lo/10 == lo%10) printf("%d\n", n);
  }
  return 0;
}
```

此程序中的新东西是 continue 和 break 语句。continue 是指跳回 for 循环的开始，执行调整语句并判断循环条件（用大白话描述就是“直接进行下一次循环”），而 break 是指直接跳出循环[①]。

这里的 continue 语句的作用是排除不足四位数的 n，直接检查后面的数。当然，也可以直接从 x=32 开始枚举，但是 continue 可以帮助我们偷懒：不必求出循环的起始点。有了 break，连循环终点也不必指定——当 n 超过 9999 后会自动退出循环。注意，这里是“退出循环”而不是“继续循环”（想一想，为什么），不信可以把 break 换成 continue 试试。

另外，注意到这里的 for 语句是“残缺”的：没有指定循环条件。事实上，3 个部分都是可以省略的。没错，for(;;) 就是一个死循环——如果不采取措施（如 break），它就永远不会结束。

2.2　循环结构程序设计

例题 2-2　3n+1 问题

猜想：对于任意大于 1 的自然数 n，若 n 为奇数，则将 n 变为 3n+1，否则变为 n 的一

① “逻辑与”&&似乎也没有出现过，但我们假设读者在学习||后已经翻阅了相关资料，或者教师已经给学生补充了这个运算符。如果确实没有学过，现在学也来得及。

半。经过若干次这样的变换，一定会使 n 变为 1。例如 3→10→5→16→8→4→2→1。

输入 n，输出变换的次数。$n \leqslant 10^9$。

样例输入：3

样例输出：7

【分析】

不难发现，程序完成的工作依然是重复性劳动：要么乘 3 加 1，要么除以 2，但和 2.1 节的程序又不太一样：循环的次数是不确定的，而且 n 也不是“递增”式的循环。这样的情况很适合用 while 循环来实现。

程序 2-4　3n+1 问题

```
#include<stdio.h>
int main()
{
  int n, count = 0;
  scanf("%d", &n);
  while(n > 1)
  {
    if(n % 2 == 1) n = n*3+1;
    else n /= 2;
    count++;
  }
  printfA("%d\n", count);
  return 0;
}
```

上面的程序有好几个值得注意的地方。首先是那个“=0”，意思是，定义整型变量 count 的同时初始化为 0。接下来是 while 语句。

提示 2-7：while 循环的格式为“while(条件) 循环体;”

它看上去比 for 循环更简单。的确如此。事实上，可以用 while 改写 for。“for(初始化; 条件; 调整) 循环体;”等价于：

```
初始化;
while(条件)
{
  循环体;
  调整;
}
```

建议读者再次利用 IDE 或者 gdb 跟踪调试，看看执行流程是怎样的。

n /=2 的含义是 n = n / 2，类似于前面介绍过的 i++。很多运算符都有类似的用法，例如 a *= 3 表示 a = a * 3。

count++在语法上并不是新鲜事物，但这里要强调的是它的作用：计数器。由于最终输

出的是变换的次数，需要一个变量来完成计数。

提示 2-8：当需要统计某种事物的个数时，可以用一个变量来充当计数器。

这个程序是正确的吗？先别急着下结论，让我们来测试一下。输入 987654321，看看结果是什么。很不幸，答案等于 1——这明显是错误的。题目中给出的范围是 $n \leqslant 10^9$，这个 987654321 是合法的输入数据。

提示 2-9：不要忘记测试。一个看上去正确的程序可能隐含错误。

问题出在哪里呢？反复阅读程序，我们仍然无法找到答案。既然观察无法解决问题，就动手实验吧！一种方法是利用 IDE 和 gdb 跟踪调试，但这并不是本书所推荐的调试方法。一个更通用的方法是：输出中间结果。

提示 2-10：在观察无法找出错误时，可以用“输出中间结果”的方法查错。

在给 n 做变换的语句后加一条输出语句 printf("%d\n", n)，我们将很快找到问题的所在：第一次输出为-1332004332，它不大于 1，所以循环终止。如果认真完成了前面的所有探索实验，你将立刻明白这其中的缘由：乘法溢出了[①]。

例题 2-3　阶乘之和

输入 n，计算 $S = 1! + 2! + 3! + \cdots + n!$的末 6 位（不含前导 0）。$n \leqslant 10^6$。这里，$n!$表示前 n 个正整数之积。

样例输入：10

样例输出：37913

【分析】

这个任务并不难，引入累加变量 S 之后，核心算法只有一句话：for(i = 1; i <= n; i++)S+= i!。不过慢着！C 语言并没有阶乘运算符，所以这句话只是伪代码，而不是真正的代码。事实上，我们还需要一次循环来计算 i!：for(j = 1; j <= i; j++)　factorial *= j;。代码如下：

程序 2-5　阶乘之和（1）

```
#include<stdio.h>
int main()
{
  int i, j, n, S = 0;
  scanf("%d", &n);
  for(i = 1; i <= n; i++)
  {
    int factorial = 1;
    for(j = 1; j <= i; j++)
      factorial *= j;
    S += factorial;
```

[①] 一个临时的解决方案是：因为 n 为奇数时 $3n+1$ 一定是偶数，下一步将其立刻除以 2。如果将两次操作一起做，可以在一定程度上缓解这个问题。有兴趣的读者可以试一试（提示：可以用 double）。

```
    }
    printf("%d\n", S % 1000000);
    return 0;
}
```

注意累乘器 factorial（英文“阶乘”的意思）定义在循环里面。换句话说，每执行一次循环体，都要重新声明一次 factorial，并初始化为 1（想一想，为什么不是 0）。因为只要末 6 位，所以输出时对 10^6 取模。

提示 2-11：*在循环体开始处定义的变量，每次执行循环体时会重新声明并初始化。*

有了刚才的经验，我们来测试一下这个程序：n=100 时，输出-961703。直觉告诉我们：乘法又溢出了。这个直觉很容易通过“输出中间变量”法得到验证，但若要解决这个问题，还需要一点数学知识。

提示 2-12：*要计算只包含加法、减法和乘法的整数表达式除以正整数 n 的余数，可以在每步计算之后对 n 取余，结果不变。*

在修正这个错误之前，我们还可以再多测试一点：试试看 n=10^6 时输出什么？更会溢出不是吗？话是没错，但是重点不在这里。事实上，它的速度太慢！让我们把程序改成“每步取模”的形式，然后加一个“计时器”，看看它到底有多慢。

程序 2-6　阶乘之和（2）

```
#include<stdio.h>
#include<time.h>
int main()
{
  const int MOD = 1000000;
  int i, j, n, S = 0;
  scanf("%d", &n);
  for(i = 1; i <= n; i++)
  {
    int factorial = 1;
    for(j = 1; j <= i; j++)
      factorial = (factorial * j % MOD);
    S = (S + factorial) % MOD;
  }
  printf("%d\n", S);
  printf("Time used = %.2lf\n", (double)clock() / CLOCKS_PER_SEC);
  return 0;
}
```

上面的程序再次使用到了常量定义，它的好处是可以在程序中使用代号 MOD 而不是常数 1000000，改善了程序的可读性，也方便修改（假设题目改成求末五位……）。

这个程序真正的特别之处在于计时函数 clock()的使用。该函数返回程序目前为止运行的时间。这样，在程序结束之前调用它，便可获得整个程序的运行时间。这个时间除以常数 CLOCKS_PER_SEC 之后得到的值以“秒”为单位。

提示 2-13：可以使用 time.h 和 clock()函数获得程序运行时间。常数 CLOCKS_PER_SEC 和操作系统相关，请不要直接使用 clock()的返回值，而应总是除以 CLOCKS_PER_SEC。

输入 20，按 Enter 键后，系统瞬间输出了答案：820313。但是请等一下，输出的 Time used 居然不是 0！明明是瞬间输出啊。原因在于，键盘输入的时间也被计算在内了——它的确是程序启动之后才进行的。为了避免输入数据的时间影响测试结果，我们使用一种称之为管道的小技巧：在 Windows 命令行下执行 echo 20 | abc，操作系统会自动帮你把 20 输入，其中 abc 是你的程序名[①]。如果不知道如何操作命令行，请参考附录。笔者建议每个读者都熟悉命令行操作，包括 Windows 和 Linux。

在尝试了多个 n 之后，我们得到了一张表（表 2-1）。

表 2-1　程序 2-6 的输出结果与运行时间表

n	20	40	80	160	1600	6400	12800	25600	51200
答案	820313	940313	940313	940313	940313	940313	940313	940313	940313
时间	<0.01	<0.01	<0.01	<0.01	0.05	0.70	2.70	11.08	43.72

表 2-1 告诉我们两件事。第一，程序的运行时间大致和 n 的平方成正比（因为 n 每扩大 1 倍，运行时间近似扩大 4 倍）。我们甚至可以估计 $n=10^6$ 时，程序大致需要近 5 个小时才能跑完。

提示 2-14：很多程序的运行时间与规模 n 存在着近似的简单关系。可以通过计时函数来发现或验证这一关系。

第二，从 40 开始，答案始终不变。这是真理还是巧合？聪明的读者也许已经知道了：25！末尾有 6 个 0，所以从第 5 项开始，后面的所有项都不会影响和的末 6 位数字——只需要在程序的最前面加一条语句“if(n>25)　n = 25;”，效率和溢出都不成问题了。

本节的两个例题展示了计数器、累加器的使用和循环结构程序设计中最常见的两个问题：算术运算溢出和程序效率低下。这两个问题都不是那么容易解决的，将在后面章节中继续讨论。另外，本节中介绍的两个工具——输出中间结果和计时函数，都是相当实用的。

2.3　文 件 操 作

例题 2-4　数据统计

输入一些整数，求出它们的最小值、最大值和平均值（保留 3 位小数）。输入保证这些数都是不超过 1000 的整数。

[①] Linux 下需要输入 echo | ./abc，因为在默认情况下，当前目录不在可执行文件的搜索路径中。

样例输入：2 8 3 5 1 7 3 6

样例输出：1 8 4.375

【分析】

如果是先输入整数 n，然后输入 n 个整数，相信读者能够写出程序来。关键在于：整数的个数是不确定的。下面直接给出程序：

程序 2-7　数据统计（错误）

```
#include<stdio.h>
int main()
{
  int x, n = 0, min, max, s = 0;
  while(scanf("%d", &x) == 1)
  {
    s += x;
    if(x < min) min = x;
    if(x > max) max = x;
    n++;
  }
  printf("%d %d %.3lf\n", min, max, (double)s/n);
  return 0;
}
```

看看这个程序多了些什么东西？什么，scanf 函数有返回值？对，它返回的是成功输入的变量个数，不信可以翻阅库函数参考手册。当输入结束时，scanf 无法再次读取 x，将返回 0。

下面该测试了。输入 2 8 3 5 1 7 3 6，按 Enter 键。怎么总不出结果？难道程序速度太慢？其实程序正在等待输入。还记得 scanf 的输入格式吗？空格、TAB 和回车符都是无关紧要的，所以按 Enter 键并不意味着输入的结束。那究竟如何才能告诉程序输入结束了呢？

提示 2-15：在 Windows 下，输入完毕后先按 Enter 键，再按 Ctrl+Z 键，最后再按 Enter 键，即可结束输入。在 Linux 下，输入完毕后按 Ctrl+D 键即可结束输入。

好了，输入终于结束了，可是输出让我们大失所望：1 2293624 4.375。这个 2293624 是从哪里来的？当我们用-O2 编译（按照惯例，不明白这是何物的读者请阅读附录）后答案变成了 1 10 4.375，和刚才不一样！换句话说：这个程序的运行结果是不确定的。在读者自己的机器上，答案甚至可能和上述两个都不同。

根据“输出中间结果”的方法，读者不难验证下面的结论：变量 max 在一开始就等于 2293624（或者 10），自然无法更新为比它小的 8。

提示 2-16：变量在未赋值之前的值是不确定的。特别地，它不一定等于 0。

解决的方法就很清楚了：在使用之前赋初值。由于 min 保存的是最小值，它的初值应该是一个很大的数；反过来，max 的初值应该是一个很小的数。一种方法是定义一个很大的常数，如 INF = 1000000000，然后让 max=INF，而 min=−INF，而另一种方法是先读取第

一个整数 x，然后令 max = min = x。这样的好处是避免了人为的“假想无穷大”值，程序更加优美；而 INF 这样的常数有时还会引起其他问题，如“无限大不够大”，或者“运算溢出”，后面还会继续讨论这个问题。

上面的程序并不是很方便：每次测试都要手工输入许多数。尽管可以用前面讲的管道的方法，但数据只是保存在命令行中，仍然不够方便。

一个好的方法是用文件——把输入数据保存在文件中，输出数据也保存在文件中。这样，只要事先把输入数据保存在文件中，就不必每次重新输入了；数据输出在文件中也避免了“输出太多，一卷屏前面的就看不见了”这样的尴尬——运行结束后，慢慢浏览输出文件即可。如果有标准答案文件，还可以进行文件比较[①]，而无须用眼睛检查输出是否正确。事实上，几乎所有算法竞赛的输入数据和标准答案都是保存在文件中的。

使用文件最简单的方法是使用输入输出重定向，只需在 main 函数的入口处加入以下两条语句：

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

它将使得 scanf 从文件 input.txt 读入，printf 写入文件 output.txt。事实上，不只是 scanf 和 printf，所有读键盘输入、写屏幕输出的函数都将改用文件。尽管这样做很方便，并不是所有算法竞赛都允许你用程序读写文件。甚至有的竞赛允许访问文件，但不允许你用 freopen 这样的重定向方式读写文件。请参赛之前仔细阅读文件读写的相关规定。

提示 2-17：请在比赛之前了解文件读写的相关规定：是标准输入输出（也称标准 I/O，即直接读键盘、写屏幕），还是文件输入输出？如果是文件输入输出，是否禁止用重定向方式访问文件？

多年来，无数选手因文件相关问题丢掉了大量的得分。一个普适的原则是：详细阅读比赛规定，并严格遵守。例如，输入输出文件名和程序名往往都有着严格规定，不要弄错大小写，不要拼错文件名，不要使用绝对路径或相对路径。

例如，如果题目规定程序名称为 test，输入文件名为 test.in，输出文件名为 test.out，就不要犯以下错误。

错误 1：程序存为 t1.c（应该改成 test.c）。

错误 2：从 input.txt 读取数据（应该从 test.in 读取）。

错误 3：从 tset.in 读取数据（拼写错误，应该从 test.in 读取）。

错误 4：数据写到 test.ans（扩展名错误，应该是 test.out）。

错误 5：数据写到 c:\\contest\\test.out（不能加路径，哪怕是相对路径。文件名应该只有 8 个字符：test.out）。

提示 2-18：在算法竞赛中，选手应严格遵守比赛的文件名规定，包括程序文件名和输入输出文件名。不要弄错大小写，不要拼错文件名，不要使用绝对路径或相对路径。

[①] 在 Windows 中可以使用 fc 命令，而在 Linux 中可以使用 diff 命令。

当然，这些错误都不是选手故意犯下的。前面说过，利用文件是一种很好的自我测试方法，但如果比赛要求采用标准输入输出，就必须在自我测试完毕之后删除重定向语句。选手比赛时一紧张，就容易忘记将其删除。

有一种方法可以在本机测试时用文件重定向，但一旦提交到比赛，就自动“删除”重定向语句。代码如下：

程序 2-8　数据统计（重定向版）

```
#define LOCAL
#include<stdio.h>
#define INF 1000000000
int main()
{
#ifdef LOCAL
  freopen("data.in", "r", stdin);
  freopen("data.out", "w", stdout);
#endif
  int x, n = 0, min = INF, max = -INF, s = 0;
  while(scanf("%d", &x) == 1)
  {
    s += x;
    if(x < min) min = x;
    if(x > max) max = x;
/*
    printf("x = %d, min = %d, max = %d\n", x, min, max);
*/
    n++;
  }
  printf("%d %d %.3lf\n", min, max, (double)s/n);
  return 0;
}
```

这是一份典型的比赛代码，包含了几个特别的地方：

- 重定向的部分被写在了#ifdef 和#endif 中。它的含义是：只有定义了符号 LOCAL，才编译两条 freopen 语句。
- 输出中间结果的 printf 语句写在了注释中——它在最后版本的程序中不应该出现，但是我们又舍不得删除它（万一发现了新的 bug，需要再次用它输出中间信息呢！）。把它注释化的好处是：一旦需要的时候，把注释符去掉即可。

上面的代码在程序首部就定义了符号 LOCAL，因此在本机测试时使用重定向方式读写文件。如果比赛要求读写标准输入输出，只需在提交之前删除#define LOCAL 即可。一个更好的方法是在编译选项而不是程序里定义这个 LOCAL 符号（不知道如何在编译选项里定义符号的读者请参考附录），这样，提交之前不需要修改程序，进一步降低了出错的可能。

如果比赛要求用文件输入输出，但禁止用重定向的方式，又当如何呢？程序如下：

程序 2-9　数据统计（fopen 版）

```
#include<stdio.h>
#define INF 1000000000
int main()
{
  FILE *fin, *fout;
  fin = fopen("data.in", "rb");
  fout = fopen("data.out", "wb");
  int x, n = 0, min = INF, max = -INF, s = 0;
  while(fscanf(fin, "%d", &x) == 1)
  {
    s += x;
    if(x < min) min = x;
    if(x > max) max = x;
    n++;
  }
  fprintf(fout, "%d %d %.3lf\n", min, max, (double)s/n);
  fclose(fin);
  fclose(fout);
  return 0;
}
```

虽然新东西不少，但也很直观：先声明变量 fin 和 fout（暂且不用管 FILE *为何物），把 scanf 改成 fscanf，第一个参数为 fin；把 printf 改成 fprintf，第一个参数为 fout，最后执行 fclose，关闭两个文件。

重定向和 fopen 两种方法各有优劣。重定向的方法写起来简单、自然，但是不能同时读写文件和标准输入输出；fopen 的写法稍显繁琐，但是灵活性比较大（例如可以反复打开并读写文件）。顺便说一句，如果想把 fopen 版的程序改成读写标准输入输出，只需赋值 fin = stdin; fout = stdout; 即可，不要调用 fopen 和 fclose[①]。

2.4　小结与习题

不知不觉，本章已经开始出现一些挑战了。尽管难度不算太高，本章的例题和习题已经出现了真正的竞赛题目——仅使用简单变量和基本的顺序、分支与循环结构就可以解决很多问题。还是老样子，在继续前进之前，请认真小结，并且完成习题。

[①] 有读者可能试过用 fopen("con", "r")的方法打开标准输入输出，但这个方法并不是可移植的——它在 Linux 下是无效的。

2.4.1 输出技巧

首先是输出技巧。假设你需要输出 2, 4, 6, 8,…, 2n，每个一行，能不能通过对程序 2-1 进行小小的改动来实现呢？为了方便，现把程序复制如下：

```
1  #include<stdio.h>
2  int main()
3  {
4    int i, n;
5    scanf("%d", &n);
6    for(i = 1; i <= n; i++)
7      printf("%d\n", i);
8    return 0;
9  }
```

任务 1：修改第 7 行，不修改第 6 行。

任务 2：修改第 6 行，不修改第 7 行。

2.4.2 浮点数陷阱

对接下来的问题，读者很容易犯错误。下面的程序运行结果是什么？“!=”运算符表示“不相等”。提示：请上机实验，不要凭主观感觉回答。

```
#include<stdio.h>
int main()
{
  double i;
  for(i = 0; i != 10; i += 0.1)
    printf("%.11f\n", i);
  return 0;
}
```

2.4.3 64 位整数

再接下来，让我们看一道题目：输入正整数 n，统计它的正因子个数。$n \leqslant 10^{12}$。例如 n=30 时，输出应该为 8。

不要想复杂了，直接枚举也不会超时。为什么呢？原因在于：如果 i 是 n 的约数，则 n/i 也是 n 的约数。除了 $i=n/i$ 这种特殊情况之外，i 和 n/i 恰好有一个不超过 n 的算术平方根。这样，从 1 枚举到 $\sqrt{n}$ 即可。另一个细节是 n 太大，超过了 int 类型的表示范围。其实，还有一种比 int 更大的类型，称为 long long，它的表示范围是 -2^{63}～$2^{63}-1$，比 -10^{19}～10^{19} 略窄——而我们一直使用的 int 的表示范围是 -2^{31}～$2^{31}-1$，只比 -2×10^{9}～2×10^{9} 略宽。

输入输出 long long 也可以借助于 printf 和 scanf 语句，但对应的占位符却是和平台与编

译器相关的：在 Linux 中，gcc 很统一的用%lld；在 Windows 中，MinGW 的 gcc 和 VC6 都需要用%I64d；但 VS2008 却是用%lld。另一个方法是使用 C++的输入输出流，我们马上就会看到。

2.4.4　C++中的输入输出

最后，我们来更仔细地研究一下输入输出。研究对象就是经典的“A+B”问题：输入若干对整数，输出每对之和。假设每个整数不超过 10^9，一共不超过 10^6 个数对。我们将用 C++而不是 C 语言来实验。别紧张，C++并没有想象中的复杂。

第 1 种方法是我们已经介绍过的：

```
#include<cstdio> // 功能和 C 中的 stdio.h 很接近，但有些许不同
using namespace std;
int main()
{
  int a, b;
  while(scanf("%d%d", &a, &b) == 2) printf("%d\n", a+b);
  return 0;
}
```

正如你所见，C++程序看上去很像是 C 程序。唯一的区别是原来的 stdio.h 变成了 cstdio，并且多了一条语句 using namespace std。这是一个普遍规律——要在 C++程序中使用 C 语言头文件，请去掉扩展名.h，并在最前面加上小写字母 c。例如，stdio.h 在 C++中的新名字[①]是 cstdio。另外，第一行中以//开头的是 C++特有的“单行注释”，它和 C 中的传统注释（/*和*/）可以混合使用。

需要说明的是：C++中保留着 C 语言的常用头文件。如果你愿意，可以继续用 stdio.h 并且省略 using namespace std 语句。事实上，很多（但不是所有）C 程序能不加修改地被 C++编译器所编译。

第 2 种方法也许更加常用（你再也不用记住%d、%lf 等恼人的占位符了）：

```
#include<iostream>
using namespace std;
int main()
{
  int a, b;
  while(cin >> a >> b) cout << a+b << "\n";
  return 0;
}
```

头文件 iostream 中包含着对输入输出流的定义，它的用法很直观，无须多说。我们已经介绍过用 fopen 和 freopen 把第 1 种写法改成文件输入输出，下面说说第 2 种写法如何修

[①] 严格地说，并不只是取了个新名字。cstdio 和 stdio.h 有些差别，只是我们平时遇不到而已。

改。首先，freopen 的方法仍然适用：在第一次使用 cin 和 cout 之前加上 freopen 函数调用，就可以方便地读写文件（再次强调，请确认竞赛规则允许你这么做）。

更加“正统”的方法是这样：

```
#include<fstream>
using namespace std;
ifstream fin("aplusb.in");
ofstream fout("aplusb.out");
int main()
{
  int a, b;
  while(fin >> a >> b) fout << a+b << "\n";
  return 0;
}
```

如果想再次使用 cin 和 cout，是否要逐个把程序中的所有 fin 和 fout 替换为 cin 和 cout？不用这么麻烦，只需要把 fin 和 fout 的声明语句去掉，并加上这样两行即可：

```
#define fin cin
#define fout cout
```

用前面介绍的条件编译，还可以让程序在本机上读写标准输入输出，比赛测试时读写文件（请读者自行实验）。

既然提出了两种不同的输入输出方法，自然应想到比较它们的效率。具体的方法前面已经讲过（提示：使用 clock()与 CLOCKS_PER_SEC），这个任务就留给读者完成吧。需要说明的是，要分别测试在重定向法与直接打开文件的方法。你会发现用输入输出流时，标准输入流 cin 比文件流 fin 慢很多很多！顺便说一句，这两种方法的效率和操作系统相关，有条件的读者可以在 Windows 和 Linux 下分别进行实验。在比赛时，请留意最终评测时所使用的操作系统。

2.4.5 小结

循环的出现让程序逻辑复杂了许多。在很多情况下，仔细研究程序的执行流程能够很好地帮助理解算法，特别是“当前行”和变量的改变。有些变量是特别值得关注的，如计数器、累加器，以及“当前最小/最大值”这样的中间变量。很多时候，用 printf 输出一些关键的中间变量能有效地帮助我们了解程序执行过程、发现错误，就像本章中多次使用的一样。

别人的算法理解得再好，遇到问题时还是需要自己分析和设计。本章介绍了“伪代码”这一工具，并建议“不拘一格”地使用。伪代码是为了让思路更清晰，突出主要矛盾，而不是写八股文。

在程序慢慢复杂起来时，测试就显得相当重要了。本章后面的几个例题几乎个个都有陷阱：运算结果溢出、运算时间过长等。程序的运行时间并不是无法估计的，我们有时能用实验的方法猜测时间和规模之间的近似关系（它的理论基础将在后面介绍），而海量数

据的输入输出问题也可以通过文件得到缓解。尽管不同竞赛在读写方式上的规定不同，熟练掌握了重定向、fopen 和条件编译后，各种情况都能轻松应付。

再次强调：编程不是看书看会的，也不是听课听会的，而是练会的。本章后面的上机编程习题中包含了不少正文中没有提到的内容，对能力的提高很有好处。如有可能，请在上机实践时运用输出中间结果、设计伪代码、计时测试等方法。

2.4.6　上机练习

注意，本章的题目需用文件输入输出。如果题目代号为 abc，那么输入文件为 abc.in，输出文件为 abc.out。如果对文件操作不熟练，请尽量把 fopen 和 freopen 两种方法都尝试一下。

习题 2-1　位数（digit）

输入一个不超过 10^9 的正整数，输出它的位数。例如 12735 的位数是 5。请不要使用任何数学函数，只用四则运算和循环语句实现。

习题 2-2　水仙花数（daffodil）

输出 100～999 中的所有水仙花数。若 3 位数 ABC 满足 $ABC=A^2+B^2+C^2$，则称其为水仙花数。例如 $153=1^3+5^3+3^3$，所以 153 是水仙花数。

习题 2-3　韩信点兵（hanxin）

相传韩信才智过人，从不直接清点自己军队的人数，只要让士兵先后以三人一排、五人一排、七人一排地变换队形，而他每次只掠一眼队伍的排尾就知道总人数了。输入 3 个非负整数 a,b,c，表示每种队形排尾的人数（$a<3, b<5, c<7$），输出总人数的最小值（或报告无解）。已知总人数不小于 10，不超过 100。

样例输入：2 1 6

样例输出：41

样例输入：2 1 3

样例输出：No answer

习题 2-4　倒三角形（triangle）

输入正整数 $n \leqslant 20$，输出一个 n 层的倒三角形。例如 $n=5$ 时输出如下：

```
#########
 #######
  #####
   ###
    #
```

习题 2-5　统计（stat）

输入一个正整数 n，然后读取 n 个正整数 $a_1, a_2, \cdots, a_n$，最后再读一个正整数 m。统计 $a_1, a_2, \cdots, a_n$ 中有多少个整数的值小于 m。提示：如果重定向和 fopen 都可以使用，哪个比较

方便？

习题 2-6 调和级数（harmony）

输入正整数 n，输出 $H(n)=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$ 的值，保留 3 位小数。例如 n=3 时答案为 1.833。

习题 2-7 近似计算（approximation）

计算 $\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots$，直到最后一项小于 10^{-6}。

习题 2-8 子序列的和（subsequence）

输入两个正整数 $n<m<10^6$，输出 $\frac{1}{n^2}+\frac{1}{(n+1)^2}+\cdots+\frac{1}{m^2}$，保留 5 位小数。例如 n=2, m=4 时答案是 0.42361；n=65536, m=655360 时答案为 0.00001。注意：本题有陷阱。

习题 2-9 分数化小数（decimal）

输入正整数 a, b, c，输出 a/b 的小数形式，精确到小数点后 c 位。$a,b\leqslant 10^6$, $c\leqslant 100$。例如 a=1, b=6, c=4 时应输出 0.1667。

习题 2-10 排列（permutation）

用 1,2,3,…,9 组成 3 个三位数 *abc*，*def* 和 *ghi*，每个数字恰好使用一次，要求 *abc*:*def*:*ghi*=1:2:3。输出所有解。提示：不必太动脑筋。

第 3 章　数组和字符串

学习目标

- ☑ 掌握一维数组的声明和使用方法
- ☑ 掌握二维数组的声明和使用方法
- ☑ 掌握字符串的声明、赋值、比较和连接方法
- ☑ 熟悉字符的 ASCII 码和 ctype.h 中的字符函数
- ☑ 正确认识++、+=等能修改变量的运算符
- ☑ 学会用编译选项-Wall 获得更多的警告信息
- ☑ 掌握 fgetc 和 getchar 的使用方法
- ☑ 了解不同操作系统中换行符的表示方法
- ☑ 掌握 fgets 的使用方法并了解 gets 的“缓冲区溢出”漏洞
- ☑ 理解预处理和迭代开发的技巧

第 2 章的程序很实用，也发挥出了计算机的计算优势，但没有发挥出计算机的存储优势——我们只用了屈指可数的变量。尽管有的程序也处理了大量的数据，但这些数据都只是“过客”，只参与了计算，并没有被保存下来。

本章介绍数组和字符串，二者都能保存大量的数据。字符串是一种数组（字符数组），但由于其应用的特殊性，适用一些特别的处理方式。

3.1　数　　组

考虑这样一个问题：读入一些整数，逆序输出到一行中。已知整数不超过 100 个。如何编写这个程序呢？首先当然是循环读取输入了。然后呢？读入每个整数以后，应该做些什么呢？思来想去，在所有整数全部读完之前，似乎没有其他事可做。换句话说，我们只能把每个数都存下来。存放在哪里呢？答案是：数组。

程序 3-1　逆序输出

```
#include<stdio.h>
#define MAXN 100 + 10
int a[MAXN];
int main()
{
  int i, x, n = 0;
  while(scanf("%d", &x) == 1)
    a[n++] = x;
```

```
    for(i = n-1; i >= 1; i--)
      printf("%d ", a[i]);
    printf("%d\n", a[0]);
    return 0;
}
```

语句 int a[100]声明了一个包含 100 个整型变量的数组，它们是：a[0],a[1],a[2],…,a[99]。注意，没有 a[100]。

提示 3-1：语句 int a[MAXN]声明了一个包含 MAXN 个整型变量的数组，即 a[0],a[1],…,a[MAXN-1]，但不包含 a[MAXN]。MAXN 必须是常数，不能是变量。

为什么这里声明 MAXN 为 100+10 而不是 100 呢？答案是：保险。

提示 3-2：在算法竞赛中，常常难以精确计算出需要的数组大小，数组一般会声明得稍大一些。在空间够用的前提下，浪费一点不要紧。

接下来是语句 a[n++] = x，它做了两件事：首先赋值 a[n]=x，然后执行 n++（即给 n 增加 1）。如果觉得难以理解，可以把它改写成{ a[n] = x; n++; }。注意这里的花括号是不能省略的，因为在默认情况下，for 语句的循环体只有一条语句。只有使用花括号时，花括号里的语句才会整体作为循环体。循环结束后，数据被存在了 a[0], a[1],…, a[n-1]，其中变量 n 是整数的个数（想一想，为什么）。

存好以后就可以输出了：依次输出 a[n-1],a[n-2],…,a[1]和 a[0]。这里有一个小问题：一般要求输出的行首行尾均无空格，相邻两个数据间用单个空格隔开。这样的话，一共要输出 n 个整数，但只有 n-1 个空格，所以只好分两条语句输出。

在上述程序中，数组 a 被声明在 main 函数的外面。请试着把 100 改成 1000000，比较一下把数组 a 放在 main 函数内外的运行结果是否相同。如果相同，试着把 1000000 改得再大一些。当你实验之后，你应该就能明白为什么要把 a 的定义放在 main 函数的外面了。简单地说，只有在放外面时，数组 a 才可以开得很大；放在 main 函数内时，数组稍大就会异常退出。它的道理将在后面讨论，现在只需要记住规则即可。

提示 3-3：比较大的数组应尽量声明在 main 函数外。

C 语言的数组并不是“一等公民”，而是“受歧视”的。例如，数组不能够进行赋值操作：在程序 3-1 中，如果声明的是 int a[MAXN], b[MAXN]，是不能赋值 b = a 的。如果要从数组 a 复制 k 个元素到数组 b，可以这样做：memcpy(b, a, sizeof(int) * k)。当然了，如果数组 a 和 b 都是浮点型的，复制时要写成 memcpy(b, a, sizeof(double) * k)。另外需要注意的是，使用 memcpy 函数要包含头文件 string.h。如果需要把数组 a 全部复制到数组 b 中，可以写得简单一些：memcpy(b, a, sizeof(a))。

例题 3-1　开灯问题

有 n 盏灯，编号为 1～n。第 1 个人把所有灯打开，第 2 个人按下所有编号为 2 的倍数的开关（这些灯将被关掉），第 3 个人按下所有编号为 3 的倍数的开关（其中关掉的灯将被打开，开着的灯将被关闭），依此类推。一共有 k 个人，问最后有哪些灯开着？输入：n

和 k，输出开着的灯编号。$k \leqslant n \leqslant 1000$。

样例输入：7 3

样例输出：1 5 6 7

【分析】

用 a[1], a[2],…, a[n]表示编号为 1, 2, 3,…,n 的灯是否开着。模拟这些操作即可：

程序 3-2　开灯问题

```
#include<stdio.h>
#include<string.h>
#define MAXN 1000 + 10
int a[MAXN];
int main()
{
  int i, j, n, k, first = 1;
  memset(a, 0, sizeof(a));
  scanf("%d%d", &n, &k);
  for(i = 1; i <= k; i++)
    for(j = 1; j <= n; j++)
      if(j % i == 0) a[j] = !a[j];
  for(i = 1; i <= n; i++)
    if(a[i]) { if(first) first = 0; else printf(" "); printf("%d", i); }
  printf("\n");
  return 0;
}
```

memset(a, 0, sizeof(a))的作用是把数组 a 清零，它也在 string.h 中定义。虽然也能用 for 循环完成相同的任务，但是用 memset 又方便又快捷。另一个技巧在输出：为了避免输出多余空格，设置了一个标志变量 first，可以表示当前要输出的变量是否为第一个。第一个变量前不应有空格，但其他都有。

例题 3-2　蛇形填数

在 $n*n$ 方阵里填入 1, 2,…, $n*n$，要求填成蛇形。例如 n=4 时方阵为：

```
10  11  12  1
 9  16  13  2
 8  15  14  3
 7   6   5  4
```

上面的方阵中，多余的空格只是为了便于观察规律，不必严格输出。$n \leqslant 8$。

【分析】

类比数学中的矩阵，我们可以用一个所谓的二维数组来储存题目中的方阵。只需声明一个 int a[MAXN][MAXN]，就可以获得一个大小为 MAXN×MAXN 的方阵。在声明时，两维的大小不必相同，因此也可以声明 int a[30][50]这样的数组，第一维下标范围是 0,1, 2,…,29，第二维下标范围是 0,1,2,…,49。

提示 3-4：可以用 int a[MAXN][MAXM]生成一个整型的二维数组，其中 MAXN 和 MAXM 不必相等。这个数组共有 MAXN*MAXM 个元素，分别为 a[0][0], a[0][1],…, a[0][MAXM-1], a[1][0],a[1][1],…,a[1][MAXM-1],…,a[MAXN-1][0],a[MAXN-1][1],…, a[MAXN-1] [MAXM-1]。

让我们从 1 开始依次填写。设“笔”的坐标为(x,y)，则一开始 $x=0$，$y=n-1$，即第 0 行，第 $n-1$ 列（别忘了行列的范围是 0 到 $n-1$，没有第 n 列）。“笔”的移动轨迹是：下，下，下，左，左，左，上，上，上，右，右，下，下，左，上。总之，先是下，到不能填了为止，然后是左，接着是上，最后是右。“不能填”是指再走就出界（例如 4→5），或者再走就要走到以前填过的格子（例如 12→13）。如果我们把所有格子初始化为 0，就能很方便地加以判断。

程序 3-3　蛇形填数

```
#include<stdio.h>
#include<string.h>
#define MAXN 10
int a[MAXN][MAXN];
int main()
{
  int n, x, y, tot = 0;
  scanf("%d", &n);
  memset(a, 0, sizeof(a));
  tot = a[x=0][y=n-1] = 1;
  while(tot < n*n)
  {
    while(x+1<n && !a[x+1][y]) a[++x][y] = ++tot;
    while(y-1>=0 && !a[x][y-1]) a[x][--y] = ++tot;
    while(x-1>=0 && !a[x-1][y]) a[--x][y] = ++tot;
    while(y+1<n && !a[x][y+1]) a[x][++y] = ++tot;
  }
  for(x = 0; x < n; x++)
  {
    for(y = 0; y < n; y++) printf("%3d", a[x][y]);
    printf("\n");
  }
  return 0;
}
```

这段程序充分利用了 C 语言简洁的优势。首先，赋值 $x=0$ 和 $y=n-1$ 后马上要把它们作为 a 数组的下标，因此可以合并完成；tot 和 a[0][n-1]都要赋值 1，也可以合并完成。这样，我们用一条语句完成了多件事情，而且并没有牺牲程序的可读性——这段代码的含义显而易见。

提示 3-5：可以利用 C 语言简洁的语法，但前提是保持代码的可读性。

那 4 条 while 语句有些难懂，不过十分相似，因此只需介绍其中的第一条：不断向下走，并且填数。我们的原则是：先判断，再移动，而不是走一步以后发现越界了再退回来。这样，我们需要进行“预判”，即是否越界，以及如果继续往下走会不会到达一个已经填过的格子。越界只需判断 $x+1<n$，因为 y 的值并没有修改；下一个格子是$(x+1,y)$，因此只需 a[x+1][y] == 0，简写成!a[x+1][y]（其中!是“逻辑非”运算符）。

提示 3-6：在很多情况下，最好是在做一件事之前检查是不是可以做，而不要做完再后悔。因为“悔棋”往往比较麻烦。

细心的读者也许会发现这里的一个“潜在 bug”：如果越界，$x+1$ 会等于 n，a[x+1][y] 将访问非法内存！幸运的是，这样的担心是不必要的。&&是短路运算符（还记得我们在哪里提到过吗？）。如果 $x+1<n$ 为假，将不会计算!a[x+1][y]，也就不会越界了。

至于为什么是++tot 而不是 tot++，留给读者思考。

3.2　字 符 数 组

文本处理在计算机应用中占有重要地位。本书到现在为止还没有正式讨论过字符串（尽管曾经使用过），因为在 C 语言中，字符串其实就是字符数组——你可以像处理普通数组一样处理字符串，只需要注意输入输出和字符串函数的使用。

例题 3-3　竖式问题

找出所有形如 *abc*de*（三位数乘以两位数）的算式，使得在完整的竖式中，所有数字都属于一个特定的数字集合。输入数字集合（相邻数字之间没有空格），输出所有竖式。每个竖式前应有编号，之后应有一个空行。最后输出解的总数。具体格式见样例输出（为了便于观察，竖式中的空格改用小数点显示，但你的程序应该输出空格，而非小数点）。

样例输入：2357

样例输出：

```
<1>
..775
X..33
-----
.2325
2325.
-----
25575

The number of solutions = 1
```

【分析】

本题的思路应该是很清晰的：尝试所有的 *abc* 和 *de*，判断是否满足条件。我们可以写出整个程序的伪代码：

```
char s[20];
int count = 0;
scanf("%s", s);
for(abc = 111; abc <= 999; abc++)
  for(de = 11; de <= 99; de++)
    if(“abc*de”是个合法的竖式)
    {
      printf("<%d>\n", count);
      打印abc*de的竖式和其后的空行
      count++;
    }
printf("The number of solutions = %d\n", count);
```

新东西是 char s[20]定义和输入语句 scanf(“%s”, s)。char 是“字符型”的意思，而字符是一种特殊的整数。为什么字符会是特殊的整数？请参见图 3-1 所示的 ASCII 编码表。

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	/	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	—	125	}	126	~	127	DEL

图 3-1　ASCII 编码表

看到了吗？每一个字符都有一个整数编码，称为 ASCII 码。为了方便书写，C 语言允许用直接的方法表示字符，例如‘a’代表的就是 a 的 ASCII 码。不过，有一些字符直接表示出来并不方便，例如回车符——它是‘\n’，而空字符是‘\0’，它也是 C 语言中字符串的结束标志。其他例子包括‘\\’（注意必须有两个反斜线）、‘\’’（这个是单引号），甚至还有的字符有两种写法：‘\”’和‘“’都表示双引号。像这种以反斜线开头的字符称为转义序列（Escape Sequence）。

提示 3-7：C 语言中的字符型用关键字 char 表示，它实际储存的是字符的 ASCII 码。字符常量可以用单引号法表示。在语法上可以把字符当作 int 型使用。

和 scanf(“%d”, &n)类似，它会读入一个不含空格、TAB 和回车符的字符串，存入字符数组 s。注意，不是 scanf(“%s”, &s)，s 前面没有&符号！

提示 3-8：在 scanf(“%s”, s)中，不要在 s 前面加上&符号。如果是字符串数组 char s[MAXN][MAXL]，可以用 scanf(“%s”, s[i])读取第 i 个字符串。

接下来有两个问题：判断和输出。根据我们的一贯作风，先考虑输出，因为它比较简单。每个竖式需要打印 7 行，但不一定要用 7 条 printf 语句，1 条足矣。我们首先计算第一行乘积 *x=abc*e*，然后是第二行 *y=abc*d*，最后是总乘积 *z=abc*de*，然后一次性打印出来：

```
printf("%5d\nX%4d\n-----\n%5d\n%4d\n-----\n%5d\n\n", abc, de, x, y, z);
```

注意这里的%5d，它表示按照 5 位数打印，不足 5 位在前面补空格（还记得%03d 吗）。完整程序如下：

程序 3-4　竖式问题

```
#include<stdio.h>
#include<string.h>
int main()
{
  int i, ok, abc, de, x, y, z, count = 0;
  char s[20], buf[99];
  scanf("%s", s);
  for(abc = 111; abc <= 999; abc++)
    for(de = 11; de <= 99; de++)
    {
      x = abc*(de%10); y = abc*(de/10); z = abc*de;
      sprintf(buf, "%d%d%d%d%d", abc, de, x, y, z);
      ok = 1;
      for(i = 0; i < strlen(buf); i++)
        if(strchr(s, buf[i]) == NULL) ok = 0;
      if(ok)
      {
       printf("<%d>\n", ++count);
       printf("%5d\nX%4d\n-----\n%5d\n%4d\n-----\n%5d\n\n", abc, de, x, y, z);
      }
    }
  printf("The number of solutions = %d\n", count);
  return 0;
}
```

还有两个函数是以前没有遇到的：sprintf 和 strchr。这个 sprintf 似曾相识：我们用过 printf 和 fprintf。没错！这 3 个函数是亲兄弟，printf 输出到屏幕，fprintf 输出到文件，而 sprintf 输出到字符串。多数情况下，屏幕总是可以输出的，文件一般也能写（除非磁盘满或者硬件损坏），但字符串就不一定了：你应该保证写入的字符串有足够的空间。

提示 3-9：可以用 sprintf 把信息输出到字符串，用法和 printf、fprintf 类似。但你应当保证字符串足够大，可以容纳输出信息。

多大才算足够大呢？答案是字符个数加 1，因为 C 语言的字符串是以空字符 '\0'结

尾的。我们还会在后面提到这个问题，但是基本原则仍然是以前说过的：如果算不清楚就把数组开大一点，空间够用的情况下浪费一点没关系。例如，在这里我们声明的缓冲字符串 buf 的长度为 99（可以包含 98 个字符），保存 *abc*, *de*, *x*, *y*, *z* 的所有数字绰绰有余。

函数 strlen(s)的作用是获取字符串 s 的实际长度。什么叫实际长度呢？字符数组 s 的大小是 20，但并不是所有空间都用上了。如果输入是“2357”，那么实际上 s 只保存了 5 个字符（不要忘记了还有一个“结束标记”‘\0’），后面 15 个字符是不确定的（还记得吗？变量在赋值之前是不确定的）。strlen(s)返回的就是结束标记之前的字符个数。因此这个字符串中的各个字符依次是 s[0], s[1],…, s[strlen(s)−1]，而 s[strlen(s)]正是结束标记‘\0’。

提示 3-10：C 语言中的字符串是以‘\0’结尾的字符数组，可以用 strlen(s)返回字符串 s 中结束标记之前的字符个数。字符串中的各个字符是 s[0], s[1],…,s[strlen(s)−1]。

提示 3-11：由于字符串的本质是数组，它也不是“一等公民”，只能用 strcpy(a, b), strcmp(a, b), strcat(a, b)来执行“赋值”、“比较”和“连接”操作，而不能用=、==、<=、+等运算符。上述函数都在 string.h 中声明。

除了字符串之外，我们还看到了++count 这种用法：它和 count++很相似——都表示给 count 加 1。它们的不同之处在于：++count 本身的值是加 1 以后的，但 count++的值是加 1 之前的（原来的值）。为了理解它的具体含义，你可以把上面的++count 换成 count++，看看效果有何不同。

注意：滥用++count 和 count++会带来很多隐蔽的错误！如果不信，猜猜看 count=0 时，printf(“%d %d %d\n”, count++, count++, count++)会输出什么（然后做个实验）。怎么样，是不是和你想的不同呢？

另一个例子是 count = count++。我们对 count++的解释是：count++本身的值是加 1 之前的值（即原来的值），但计算 count++之后 count 会增加 1。问题就出在这里了。这个“稍后再加 1”到底是何时进行的呢？如果是计算完赋值的右边（即 count++）之后就立刻执行，最后 count 的值不会变（别忘了最后执行的是赋值）；但如果是整个赋值完成之后才加 1，最后 count 的值会比原来多 1。如果你在理解刚才这段话时感到吃力，最好的方法是避开它。

提示 3-12：滥用++、--、+=等可以修改变量值的运算符很容易带来隐蔽的错误。建议每条语句最多只用一次这种运算符，并且它所修改的变量在整条语句中只出现一次。

事实上，就算充分理解了这条规则，在实际编程时也可能临时忘记。好在可以利用编译器减少这种错误。用-Wall 编译刚才的两个例子，编译器都会给出警告：对 count 的运算可能是没有定义的。

提示 3-13：用编译选项-Wall 编译程序时，会给出很多（但不是所有）警告信息，以帮助程序员查错。但这并不能解决所有的问题：有些“错误”程序是合法的，只是这些动作不是你所期望的。

3.3　最长回文子串

例题 3-4　回文串

输入一个字符串，求出其中最长的回文子串。子串的含义是：在原串中连续出现的字符串片段。回文的含义是：正着看和倒着看相同，如 abba 和 yyxyy。在判断时，应该忽略所有标点符号和空格，且忽略大小写，但输出应保持原样（在回文串的首部和尾部不要输出多余字符）。输入字符串长度不超过 5000，且占据单独的一行。应该输出最长的回文串，如果有多个，输出起始位置最靠左的。

样例输入：Confuciuss say: Madam, I'm Adam.

样例输出：Madam, I'm Adam

【分析】

如果输入全部是大写字母，问题就简单了：直接判断每个子串即可。可其他字符的出现把问题搞得复杂了起来。首先，我们不能用 scanf("%s")输入字符串，因为它碰到空格或者 TAB 就会停下来。可以用下述两种方法解决这个问题：

第 1 种方法是使用 fgetc(fin)，它读取一个打开的文件 fin，读取一个字符，然后返回一个 int 值。为什么返回的是 int 而不是 char 呢？因为如果文件结束，fgetc 将返回一个特殊标记 EOF，它并不是一个 char。如果把 fgetc(fin)的返回值强制转换为 char，将无法把特殊的 EOF 和普通字符区分开。如果要从标准输入读取一个字符，可以用 getchar()，它等价于 fgetc(stdin)。

提示 3-14：使用 fgetc(fin)可以从打开的文件 fin 中读取一个字符。一般情况下应当在检查它不是 EOF 后再将其转换成 char 值。从标准输入读取一个字符可以用 getchar()，它等价于 fgetc(stdin)。

fgetc 和 getchar()将读取“下一个字符”，因此你需要知道在各种情况下，“下一个字符”是哪个。如果用 scanf("%d", &n)读取整数 n，则要是在输入 123 后多加了一个空格，用 getchar()读取的将是这个空格；如果在“123”之后紧跟着换行，则读取到的将是回车符'\n'。

这里有个潜在的陷阱：不同操作系统的回车换行符是不一致的。Windows 是'\r'和'\n'两个字符，Linux 是'\n'，而 MacOS 是'\r'。如果在 Windows 下读取 Windows 文件，fgetc()和 getchar()会把'\r'“吃掉”，只剩下'\n'；但如果要在 Linux 下读取同样一个文件，它们会忠实地先读取'\r'，然后才是'\n'。如果编程不注意，你的程序可能会在某个操作系统上是完美的，但在另一个操作系统上就错得一塌糊涂。当然，比赛的组织方应该避免在 Linux 下使用 Windows 格式的文件，但选手也应该把自己的程序写得更鲁棒，即容错性更好。

提示 3-15：在使用 fgetc 和 getchar 时，应该避免写出和操作系统相关的程序。

第 2 种方法是使用 fgets(buf, MAXN, fin)读取完整的一行，其中 buf 的声明为 char buf[MAXN]。这个函数读取不超过 MAXN-1 个字符，然后在末尾添上结束符'\0'，因此不会出现越界的情况。之所以说可以用这个函数读取完整的一行，是因为一旦读到回车符'\n'，

读取工作将会停止，而这个'\n'也会是 buf 字符串中最后一个有效字符（再往后就是字符串结束符'\0'了）。只有在一种情况下，buf 不会以'\n'结尾：读到文件结束符，并且文件的最后一个不是以'\n'结尾。尽管比赛的组织方应避免这样的情况（和输出文件一样，保证输入文件的每行均以回车符结尾），但正如刚才所说，选手应该把自己的程序写得更鲁棒。

提示 3-16：fgets(buf, MAXN, fin)将读取完整的一行放在字符数组 buf 中。你应当保证 buf 足够存放下文件的一行内容。除了在文件结束前没有遇到'\n'这种特殊情况外，buf 总是以'\n'结尾。当一个字符都没有读到时，fgets 返回 NULL。

和 fgetc 一样，fgets 也有一个“标准输入版”gets。遗憾的是，gets 和它的兄弟 fgets 差别比较大：它的用法是 gets(s)，没有指明读取的最大字符数。这里就出现了一个潜在的问题：gets 将不停地往 s 里塞东西，而不管塞不塞得下！难道 gets 函数不去管 s 的可用空间有多少吗？你还真说对了。

提示 3-17：C 语言并不禁止程序读写“非法内存”。例如你声明的是 char s[100]，你完全可以赋值 s[10000] = 'a'（甚至-Wall 也不会警告），但后果自负。

正是因为如此，gets 已经被废除了，但为了向后兼容，你仍然可以使用它。从长远考虑，读者最好不要使用它。

提示 3-18：C 语言中的 gets(s)存在缓冲区溢出漏洞，不推荐使用。

说了那么多，我们终于解决了“输入中有空格”的问题。我们选择的是 fgets 函数，它可以一次性读取一整行，最为方便。

接下来，需要解决“判断时忽略标点，输出时却要按原样”的问题。首先，输入的标点符号不能直接删除（否则输出时就没办法了），但如果每次判断时都要跳过标点符号，又似乎不太直接，调试也麻烦。

这里介绍一个通用的方案：预处理。构造一个新的字符串，不包含原来的标点符号，而且所有字符变成大写（顺便解决了大小写的问题）：

```
n = strlen(buf);
m = 0;
for(i = 0; i < n; i++)
  if(isalpha(buf[i])) s[m++] = toupper(buf[i]);
```

上面的代码用到了一个新函数：ctype.h 中的 isalpha(c)，它用于判断字符 c 是否为大写字母或小写字母。用 toupper(c)返回 c 的大写形式。在这样的预处理之后，buf 保存的就是原串中的所有字母了。顺便说一句，c-'a'+'A'也可以把小写字母变成大写（想一想，为什么）。

提示 3-19：当任务比较复杂时，可以用预处理的方式简化输入，并提供更多的数据供使用。复杂的字符串处理题目往往可以通过合理的预处理简化任务，便于调试。

提示 3-20：头文件 ctype.h 中定义的 isalpha、isdigit、isprint 等工具可以用来判断字符的属性，而 toupper、tolower 等工具可以用来转换大小写。

接下来的问题就简单了：枚举回文串的起点和终点，然后判断它是否真的是回文串。

```
int max = 0;
for(i = 0; i < m; i++)
  for(j = i; j < m; j++)
if(s[i...j]是回文串 && j-i+1 > max) max = j-i+1;
```

这里用到了“当前最大值”变量 max，它保存的是目前为止发现的最长回文子串的长度。如果串 s 的第 i 个字符到第 j 个字符（记为 s[i…j]）是回文串，则检查长度 j-i+1 是否超过 max。

最后，判断 s[i…j]是否为回文串的方法也不难写出：

```
int ok = 1;
for(k = i; i <= j; i++)
  if(s[k] != s[i+j-k]) ok = 0;
```

注意这里的循环变量不能是 i 或者 j，因为它们已经在外层用过了。s[k]的“对称”位置是 s[i+j-k]（想一想，为什么），因此只要一次比较失败，就应把标记变量 ok 置为 0。

下面是到目前为止的程序：

程序 3-5　最长回文子串（1）

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAXN 5000 + 10
char buf[MAXN], s[MAXN];
int main()
{
  int n, m = 0, max = 0;
  int i, j, k;
  fgets(buf, sizeof(s), stdin);
  n = strlen(buf);
  for(i = 0; i < n; i++)
    if(isalpha(buf[i])) s[m++] = toupper(buf[i]);
  for(i = 0; i < m; i++)
    for(j = i; j < m; j++)
    {
      int ok = 1;
      for(k = i; k <= j; k++)
        if(s[k] != s[i+j-k]) ok = 0;
      if(ok && j-i+1 > max) max = j-i+1;
    }
  printf("max = %d\n", max);
  return 0;
}
```

这个程序还有一些功能没有完成，但已经离成功不远了——在实际编程时，我们经常先编写一个具备主要功能的程序，再加以完善。我们甚至可以先写一个只有输入输出功能的“骨架”，但是要确保它正确。这样，每次只添加一点点小功能，而且写一点就测试一点，和一次写完整个程序相比，更加不容易出错。这种方法称为迭代式开发。

提示 3-21：在程序比较复杂时，除了在设计阶段可以用伪代码理清思路外，编码阶段可以采用迭代时开发——每次只实现一点小功能，但要充分测试，确保它工作正常。

经测试，上面的代码已经可以顺利求出样例数据中最长回文串的长度——17 了。接下来的任务是输出这个回文串：要求原样输出，并且尽量靠左。输出靠左的条件已经满足了：我们是从左到右枚举的，且只在 j-i+1 严格大于 max 时才更新 max。这样，就只剩下唯一的问题了：原样输出。

经过一番思考以后，似乎小小的改动是不足够的——即使在更新 max 时把 i 和 j 保存下来，我们还是不知道 s[i]和 s[j]在原串 buf 中的位置。因此，我们必须增加一个数组 p，用 p[i]保存 s[i]在 buf 中的位置。它可以很容易地在预处理中得到；然后在更新 max 的同时把 p[i]和 p[j]保存到 x 和 y，最后输出 buf[x]到 buf[y]中的所有字符。

看上去很完美了，不是吗？等一下！题目说了字符可以多达 5000 个，程序效率会不会太低呢？只需要生成一个 5000 个‘a’的字符串就会发现：确实太慢了。

其实我们可以换一种方式：枚举回文串的“中间”位置 i，然后不断往外扩展，直到有字符不同。下面是完整程序，聪明的读者，你能看懂吗？提示：长度为奇数和偶数的处理方式是不一样的。

程序 3-6　最长回文子串（2）

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#define MAXN 5000 + 10
char buf[MAXN], s[MAXN];
int p[MAXN];
int main()
{
  int n, m = 0, max = 0, x, y;
  int i, j;
  fgets(buf, sizeof(s), stdin);
  n = strlen(buf);
  for(i = 0; i < n; i++)
    if(isalpha(buf[i]))
    {
      p[m] = i;
      s[m++] = toupper(buf[i]);
    }
  for(i = 0; i < m; i++)
```

```
  {
    for(j = 0; i-j >= 0 && i+j < m; j++)
    {
      if(s[i-j] != s[i+j]) break;
      if(j*2+1 > max) { max = j*2+1; x = p[i-j]; y = p[i+j]; }
    }
    for(j = 0; i-j >= 0 && i-j+1 < m; j++)
    {
      if(s[i-j] != s[i-j+1]) break;
      if(j*2+2 > max) { max = j*2+2; x = p[i-j]; y = p[i+j+1]; }
    }
  }
  for(i = x; i <= y; i++)
    printf("%c", buf[i]);
  printf("\n");
  return 0;
}
```

3.4 小结与习题

到目前为止，C 语言的核心内容已经全部讲完了。理论上，运用前 3 章的知识足以编写大部分算法竞赛程序了。

3.4.1 必要的存储量

数组可以用来保存很多数据，但在一些情况下，并不需要把数据保存下来。下面哪些题目可以不借助数组，哪些必须借助数组？请编程实现。假设输入只能读一遍。

- 输入一些数，统计个数。
- 输入一些数，求最大值、最小值和平均数。
- 输入一些数，哪两个数最接近。
- 输入一些数，求第二大的值。
- 输入一些数，求它们的方差。
- 输入一些数，统计不超过平均数的个数。

3.4.2 用 ASCII 编码表示字符

下面让我们探索一下字符在 C 语言中的表示。从正文中我们知道，有些特殊的字符需要转义才能表达，如\n 表示换行，\\表示反斜杠，\"表示引号，\0 表示空字符，那还有哪些转义符呢？如果在网上搜索一下，或者翻阅任何一本 C 语言参考书，你会发现转义字符表中有下面这样的说法。

提示 3-22：字符还可以直接用 ASCII 码表示。如果用八进制，应该写成\o, \oo 或\ooo（o 为一个八进制数字）；如果用十六进制，应该写成\xh（h 为十六进制数字串）。

到底什么是八进制和十六进制呢？我们平时使用的是“逢十进一”的进位制系统，称为十进制（Decimal System）。而在计算机内部，所有事物都是用“逢二进一”的二进制（Binary System）来表示。从表 3-1 很容易看出二者之间的关系。

表 3-1 十进制和二进制的转换关系

十进制	0	1	2	3	4	5	6	7
二进制	0	1	10	11	100	101	110	111

类似地，可以定义八进制和十六进制（注意，在十六进制中，用字符 A～F 表示十进制中的 10～15）。顺便说一句：如果你的操作系统是 Windows，打开“计算器”后，先切换成“科学型”，然后输入一个整数，例如 123，再单击“二进制”按钮，就可以看到它的二进制值 1111011、八进制值 173 和十六进制值 7B[①]。而语句 printf(“%d %o %x\n”, a)将把整数 a 分别按照十进制、八进制和十六进制输出。

3.4.3 补码表示法

计算机中的二进制是没有符号的。尽管 123 的二进制值是 1111011，-123 在计算机内并不表示为-1111011——这个“负号”也需要用二进制位来表示！

“正号和符号”只有两种情况，因此用一个二进制位就可以了。容易想到一个表示“带符号 32 位整数”的方法：用最高位表示符号（0：正数；1：负数），剩下 31 位表示数的绝对值。可惜，这并不是机器内部真正的实现方法。在笔者的机器上，语句 printf("%u\n",-1)的输出是 4294967295[②]。把-1 换成-2、-3、-4…后，很容易总结出一个规律：-n 的内部表示是 2^{32}-n。这就是著名的“补码表示法”（Complement Representation）。

提示 3-23：在多数计算机内部，整数采用的是补码表示法。

为什么计算机要用这样一个奇怪的表示方法呢？前面提到的“符号位+绝对值”的方法哪里不好了？答案是：运算不方便。试想，我们要计算 1 + (-1)的值（为了简单起见，假设两个数都是带符号 8 位整数）。如果用“符号位+绝对值”法，将要计算 00000001+10000001，而答案应该是 00000000。似乎想不到什么简单的方法进行这个“加法”。但如果采用补码表示，计算的是 00000001+11111111，只需要直接相加，并丢掉最高位的进位即可！顺便说一句，“符号位+绝对值”还有一个好玩的“bug”：存在两种不同的 0：一个是 00000000（正 0），一个是 10000000（负 0）。这个问题在补码表示法中不会出现（想一想，为什么）。

学到这里，你能解释“int 类型的最小、最大值”了吗？提示：在通常情况下，int 是 32 位的。

① 遗憾的是，Linux 下的 GUI 计算器 xcalc 无法进行进制转换。不过很多系统预装了 bc 程序，可以使用 echo 'obase=2; ibase=10; 123' | bc 把十进制 123 转换成二进制。

② 请记住这个整数，它等于 2^{32}-1。

3.4.4　重新实现库函数

在学习字符串时，重新实现一些库函数的功能是很有益的。

练习 1：只用 getchar 函数读入一个整数。假设它占据单独的一行，读到行末为止，包括换行符。输入保证读入的整数可以保存在 int 中。

练习 2：只用 fgets 函数读入一个整数。假设它占据单独的一行，读到行末为止，包括换行符。输入保证读入的整数可以保存在 int 中。

练习 3：只用 getchar 实现 fgets 的功能，即用每次一个字符的方式读取整行。

练习 4：实现 strchr 的功能，即在一个字符串中查找一个字符。

练习 5：实现 isalpha 和 isdigit 的功能，即判断字符是否为字母/数字。

3.4.5　字符串处理的常见问题

```
tot = 0;
for(i = 0; i < strlen(s); i++)
  if(s[i] == 1) tot++;
printf("There are %d character(s) '1' in the string.\n", tot);
```

不难看出，该程序的目的是统计字符串中字符 1 的个数。

实验 1：添加字符串 s 的声明语句，长度不小于 10^7。提示：放在 main 函数内还是外？

实验 2：添加读取语句，测试上述程序是否输出了期望的结果。如果不是，请改正。

实验 3：把输入语句注释掉，添加语句，用程序生成一个长度至少为 10^5 的字符串，并用程序验证字符串长度确实不小于 10^5。

实验 4：用计时函数测试这段程序的运行时间随着字符串长度的变化规律。如何改进？

3.4.6　关于输入输出

请用 getchar()读取一个字符。试试看，当程序请求键盘输入时，getchar()函数何时返回？是在输入第一个字符时吗？如果直接输入文件结束符（还记得吗？Windows 是 Ctrl+Z，Linux 是 Ctrl+D），getchar()读到的是什么？

如果你有一个格式为 HH:MM:SS 的字符串 s（例如“12:34:56”），如何仅用一条 sscanf 语句得到 HH、MM 和 SS 的值？你的语句看上去应该是这个样子：sscanf(s, “????”, &HH, &MM, &SS);。你只需把问号替换成某个特殊的格式串。

3.4.7　I/O 的效率

最后，像第 2 章末尾那样做一下性能测试。题目是“直接输出”：输入不超过 1000 行字符串，然后直接输出。每行都是长度不超过 10000 的字符串，包含大写字母和小写字母，不包含任意空白（如空格、TAB）。用 C++来测试——它又有新花样。

第 1 种方法是用 C++中的流和字符串对象。

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
  string s;
  while(cin >> s) cout << s << "\n";
  return 0;
}
```

注意，这里声明的是C++中的字符串，否则无法使用输入输出流。幸运的是，C++字符串和C语言的字符数组是可以相互转换的：如果s是一个字符数组，那么string(s)就是相应的字符串；如果s是一个字符串，则s.c_str()就是相应的字符数组。需要注意的是，c_str()返回的内容是只读的。例如，我们可以用printf("%s", s.c_str())来输出，但不能用scanf("%s", s.c_str())来输入。总之，不熟悉的人很容易错误地使用C++的字符串，还是先牢牢掌握好字符数组和string.h中的库函数吧。

第2种方法是用getchar：

```
#include<cstdio>
using namespace std;
int main()
{
  int ch;
  while((ch = getchar()) != EOF) putchar(ch);
  return 0;
}
```

注意赋值语句自身也是有值的，所以读取一个字符的同时可以立刻判断它是否为EOF。

第3种方法是用fgets：

```
#include<cstdio>
using namespace std;
#define MAXN 100010
char s[MAXN];
int main()
{
  int ch;
  while(fgets(s, MAXN, stdin) != NULL) puts(s);
  return 0;
}
```

下面请自行生成测试数据，并且开始计时测试吧。同样地，请同时使用重定向的方式和直接读写文件的方式（另外，在Windows和Linux下，测试结果会有不可忽视的差别）。

顺便说一句，C++中还有一种“字符串流”，可以实现类似sscanf和sprintf的功能：

```
#include<iostream>
#include<sstream>
using namespace std;
int main()
{
  char s[1000];
  cin.getline(s, 1000, '\n');
  stringstream ss(s);
  int a, b;
  ss >> a >> b;
  cout << a+b << "\n";
  return 0;
}
```

上面的函数先从 cin 读取一行。getline 函数的第 3 个参数是行分隔符。它的默认值就是'\n'，因此可以简化为 cin.getline(s, 1000)，其中 1000 的含义和 fgets 中的类似。

3.4.8　小结

本节介绍的语法和库函数都是很直观的，但是书中的程序理解起来比第 2 章复杂了很多，原因在于我们所关心的变量突然多了很多。每当用到 a[i]或者 s[i]这样的元素时，我们应该问自己："i 等于多少？它有什么实际含义吗？"作为数组下标，i 经常代表"当前考虑的位置"，或者与另一个下标 j 一起表示"当前考虑的子串的起点和终点"。

数组和字符串往往意味着大数据量，而处理大数据量时经常会遇到"访问非法内存"的错误。在语法上，C 语言并不禁止程序访问非法内存，但后果难料。这在理论上可以通过在访问数组前检查下标是否合法来缓解，但程序会比较累赘；另一种技巧是适当把数组开大，特别是不清楚数组应该开多大的时候。只要内存够用，开大一点没关系。顺便说一句，数组的大小可以用 sizeof 在编译时刻获得（它不是一个函数），它经常被用在 memset、memcpy 等函数中。有的函数并没有做大小检查，因而存在缓冲区溢出漏洞。本章中只讲了 gets，但其实 strcpy 也有类似问题——如果源字符串并不是以'\0'结尾的，复制工作将可能覆盖到缓冲区之外的内存！这也提醒我们：如果按照自己的方式处理字符串，千万要保证它以'\0'结尾。

在数组和字符串处理程序中，下标的计算是极为重要的。为了方便，很多人喜欢用++等可以修改变量（有副作用）的运算符，但千万注意保持程序的可读性。一个保守的做法是如果使用这种运算符，被影响的变量在整个表达式中最多出现一次（例如 i= i++就是不允许的）。

理解字符编码对于正确地使用字符串是至关重要的。算法竞赛中涉及的字符一般是 ASCII 表中的可打印字符。对于中文的 GBK 编码，简单的实验将得出这样的结论：如果 char 值为正，则是西文字符；如果为负，则是汉字的前一半（这时需要再读一个 char）。这个结论并不是普遍成立的（在某些环境下，char 类型是非负的），但在大多数情况下，这样做是可行的。关于字符另一个有意思的东西是转义序列——几乎所有编程语言都定义了自己的转义序列，但大都和 C 语言类似。

3.4.9 上机练习

习题 3-1 分数统计（stat）

输入一些学生的分数，哪个分数出现的次数最多？如果有多个并列，从小到大输出。

任务 1：分数均为不超过 100 的非负整数。

任务 2：分数均为不超过 100 的非负实数，但最多保留两位小数。

习题 3-2 单词的长度（word）

输入若干个单词，输出它们的平均长度。单词只包含大写字母和小写字母，用一个或多个空格隔开。

习题 3-3 乘积的末 3 位（product）

输入若干个整数（可以是正数、负数或者零），输出它们的乘积的末 3 位。这些整数中会混入一些由大写字母组成的字符串，你的程序应当忽略它们。提示：试试看，在执行 scanf("%d")时输入一个字符串会怎样？

习题 3-4 计算器（calculator）

编写程序，读入一行恰好包含一个加号、减号或乘号的表达式，输出它的值。这个运算符保证是二元运算符，且两个运算数均为不超过 100 的非负整数。运算数和运算符可以紧挨着，也可以用一个或多个空格、TAB 隔开。行首末尾均可以有空格。提示：选择合适的输入方法可以将问题简化。

样例输入：1+1
样例输出：2
样例输入：2-　　5
样例输出：-3
样例输入：0　*1982
样例输出：0

习题 3-5 旋转（rotate）

输入一个 $n*n$ 字符矩阵，把它左转 90° 后输出。

习题 3-6 进制转换 1（base1）

输入基数 b（$2 \leqslant b \leqslant 10$）和正整数 n（十进制），输出 n 的 b 进制表示。

习题 3-7 进制转换 2（base2）

输入基数 b（$2 \leqslant b \leqslant 10$）和正整数 n（b 进制），输出 n 的十进制表示。

习题 3-8 手机键盘（keyboard）

输入一个由小写字母组成的英文单词，输出用手机的默认英文输入法的敲键序列。例如要打出 pig 这个单词，需要按 1 次 p，3 次 i，（稍作停顿后）1 次 i，记为 p1i3i1。

第 4 章　函数和递归

学习目标

- ☑ 掌握多参数、单返回值的数学函数的定义和使用方法
- ☑ 学会用 typedef 定义结构体
- ☑ 学会用 assert 宏帮助调试
- ☑ 理解函数调用时用实参给形参赋值的过程
- ☑ 学会定义局部变量和全局变量
- ☑ 理解调用栈和栈帧，学会用 gdb 查看调用栈并选择栈帧
- ☑ 理解地址和指针
- ☑ 理解递归定义和递归函数
- ☑ 理解可执行文件中的正文段、数据段和 BSS 段
- ☑ 熟悉堆栈段，了解栈溢出的常见原因

运用前 3 章的知识尽管在理论上已经足以写出多数算法程序了，但实际上稍微复杂一点的程序往往由多个函数组成。函数是“过程式程序设计”的自然产物，但也产生了局部变量、参数传递方式、递归等诸多新的知识点。本章淡化例题，重点在于理解这纷繁复杂的、最后的语法。同时，通过请出 gdb 这一王牌，从根本上帮助读者理解，看清事物的本质。

4.1　数 学 函 数

4.1.1　简单函数的编写

我们已经用过了许多数学函数，如 cos、sqrt 等。能不能自己写一个呢？没问题。下面就编写一个计算两点欧几里德距离的函数：

```
double dist(double x1, double y1, double x2, double y2)
{
  return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
}
```

提示 4-1：C 语言中的数学函数可以定义成“返回类型 函数名(参数列表) { 函数体 }”，其中函数体的最后一条语句应该是“return 表达式;”。

这里，参数和返回值的类型一般是我们学过的“一等公民”，如 int 或者 double，也可以是 char。可不可以是数组呢？也不是不可以，但是比较麻烦，我们稍后再考虑。

提示 4-2：函数的参数和返回值最好是“一等公民”int 或者 double（注意 char 是一种特殊的 int）。其他“非一等公民”作为参数和返回值要复杂一些。

注意这里的 return 是一个动作，而不是描述。

提示 4-3：如果函数在执行的过程中碰到了 return 语句，将直接退出这个函数，不去执行后面的语句。相反，如果在执行过程中始终没有 return 语句，则会返回一个不确定的值。幸好，-Wall 可以捕捉到这一可疑情况并产生警告。

顺便说一句，main 函数也是有返回值的！到目前为止，我们总是让它返回 0，这个 0 是什么意思呢？尽管没有专门说明，读者应该已经发现了，main 函数是整个程序的入口。换句话说，有一个“其他的程序”来调用这个 main 函数——如操作系统、IDE、调试器，甚至自动评测系统。这个 0 代表“正常结束”，就是返回给这些调用者的。在算法竞赛中，除了有特殊规定之外，请总是让它返回 0，以免评测系统错误地认为你的程序异常退出了。

提示 4-4：在算法竞赛中，请总是让 main 函数返回 0。

函数不见得是要一步出结果的。下面是上述函数的另一种写法：

```
double dist(double x1, double y1, double x2, double y2)
{
  double dx = x1-x2;
  double dy = y1-y2;
  return hypot(dx, dy);
}
```

这里用到了一个新的数学函数——hypot，相信读者能猜到它的意思[①]。这个例子也告诉我们：一个函数也可以调用其他函数——在自定义函数中写代码和在 main 函数（没错！它也是个函数）中写代码并没有什么区别，以前讲过的东西都能用。

4.1.2 使用结构体的函数

下面来思考一个问题：这个函数好不好用？按理说，x1 和 y1 在语义上属于一个整体(x1,y1)，而 x2 和 y2 属于另一个整体(x2,y2)，它们代表两个点的坐标。我们能不能设计一个函数，它的参数是明显的两个点，而不是 4 个 double 型的坐标值呢？

```
struct Point{ double x, y; };

double dist(struct Point a, struct Point b)
{
  return hypot(a.x-b.x, a.y-b.y);
}
```

这里出现了一个新东西。我们定义了一个称为 Point 的结构体，包含两个域：double 型的 x 和 y。

提示 4-5：在 C 语言中，定义结构体的方法为：“struct 结构体名称{ 域定义 };”注意花括号的后面还有一个分号。

[①] 注意：这个函数不是 ANSI C 的。

这样用起来好象还是挺别扭的：所有用到 Point 的地方都得写一个“struct”。有一个方法可以避开这些 struct，让结构体用起来和 int、double 这样的“原生”类型更接近：

```
typedef struct{ double x, y; }Point;

double dist(Point a, Point b)
{
  return hypot(a.x-b.x, a.y-b.y);
}
```

虽然没少几个字符，但是看上去清爽多了！

提示 4-6：为了使用方便，往往用“typedef struct { 域定义; }类型名;”的方式定义一个新类型名。这样，就可以像原生数据类型一样使用这个自定义类型。

4.1.3　应用举例

例题 4-1　组合数

输入非负整数 n 和 m，输出组合数 $C_n^m = \dfrac{n!}{m!(n-m)!}$，其中 $m \leqslant n \leqslant 20$。

【分析】

既然题目中的公式出现了那么多次 $n!$，把它作为一个函数编写是比较合理的：

程序 4-1　组合数

```
#include<stdio.h>
int f(int n)
{
  int i, m = 1;
  for(i = 1; i <= n; i++)
    m *= i;
  return m;
}

int main()
{
  int m, n;
  scanf("%d%d", &m, &n);
  printf("%d\n", f(n)/(f(m)*f(n-m)));
  return 0;
}
```

你看，编写自己的函数并不是一件难事。写完之后可以像 cos、sqrt 等库函数一样调用它，很方便。

“别忘了测试！”如果你这样说，请为自己鼓掌。还记得第 2 章那个“阶乘”之和的

第一个程序吗？那个程序溢出了。那这个程序呢？很不幸：m=20，n=1 的输出竟然是-19。手算不难得到：m=20，n=1 的正确结果是 1。所以——

提示 4-7：即使最终答案在我们选择的数据类型范围之内，计算的中间结果仍然可能溢出。

这个题目还告诉我们：即使你认为题目在“暗示”你使用某种语言特性，也应该深入分析，不能贸然行事。如何避免中间结果溢出？这个问题留给读者思考。

例题 4-2　孪生素数

如果 n 和 n+2 都是素数，则称它们是孪生素数。输入 m，输出两个数均不超过 m 的最大孪生素数。5≤m≤10000。例如 m=20 时答案是 17、19，m=1000 时答案是 881、883。

【分析】

根据定义，被 1 和它自身整除的、大于 1 的整数称为素数，因为要找出最大的孪生素数，可以从大到小枚举 n，依次判断 n 和 n+2 是否均为素数。

尽管“判断这一素数”这一过程并不复杂，但由于在枚举中要判断两遍，也不太方便。最好的方式是把“判断素数”写成一个函数，只需两遍调用即可。这样的“判断一个事物是否具有某一性质”的函数还有一个学术名称——谓词（predicate）。

程序 4-2　孪生素数（1）

```
#include<stdio.h>
/* do NOT use this if x is very large or small */
int is_prime(int x)
{
  int i;
  for(i = 2; i*i <= x; i++)
    if(x % i == 0) return 0;
  return 1;
}

int main()
{
  int i, m;
  scanf("%d", &m);
  for(i = m-2; i >= 3; i--)
    if(is_prime(i) && is_prime(i+2))
    {
      printf("%d %d\n", i, i+2);
      break;
    }
  return 0;
}
```

注意在 is_prime 函数的编写中，我们用到了两个小技巧。一是只判断不超过 sqrt(x)的整数 i（想一想，为什么）。二是及时退出：一旦发现 x 有一个大于 1 的因子，立刻返回 0（假），

只有最后才返回 1（真）。函数名的选取是有讲究的：“is_prime”取自英文“is it a prime?”（它是素数吗？）。

提示 4-8：建议把谓词（用来判断某事物是否具有某种特性的函数）命名成“is_xxx”的形式。它返回 int 值，非 0 表示真，0 表示假。

注意程序 4-2 中 is_prime 函数上方的注释：不要用在太小或太大的 n 上！这是为什么呢？n 太小时不难解释：n=1 会被错误地判断为素数（因为确实没有其他因子）。n 太大时的理由不那么明显：i*i 可能会溢出！如果 n 是一个接近 int 的最大值的素数，则当循环到 i=46340 时 i*i=2147395600<n；但 i=46341 时 i*i=2147488281，超过了 int 的最大值，溢出变成负数，仍然满足 i*i<n！n 不是太大的话，运气好还能碰上 101128442 溢出后等于 2147483280，终止循环；但如果 n= 2147483647 的话，循环将一直进行下去。

提示 4-9：编写函数时，应尽量保证它能对任何合法参数都能得到正确的结果。如若不然，应在显著位置标明函数的缺陷，以避免误用。

下面是改进之后的版本：

程序 4-3　孪生素数（2）

```
#include<stdio.h>
#include<math.h>
#include<assert.h>
int is_prime(int x)
{
  int i, m;
  assert(x >= 0);
  if(x == 1) return 0;
  m = floor(sqrt(x) + 0.5);
  for(i = 2; i <= m; i++)
    if(x % i == 0) return 0;
  return 1;
}

int main()
{
  int i, m;
  scanf("%d", &m);
  for(i = m-2; i >= 3; i--)
    if(is_prime(i) && is_prime(i+2))
    {
      printf("%d %d\n", i, i+2);
      break;
    }
  return 0;
}
```

除了特判 n==1 的情况外，程序中还使用了变量 m，一方面避免了每次重复计算 sqrt(x)，另一方面也通过四舍五入避免了浮点误差——如果 sqrt“不小心”把某个本应是整数的值弄成了 xxx.99999，也将被修正，但直接写 m = sqrt(x)的话那个“.99999”会被无情地截掉。

最后，程序使用了 assert.h 中的 assert 宏来限制非法的函数调用：当 x>=0 不成立时，程序将异常终止，并给出提示信息。检查非法参数是很有用的：当算法很复杂时，一不小心就会用非法参数调用某些自定义函数，如果函数不对参数进行检查，则很可能让整个程序得到一个荒唐的结果。如果函数调用关系非常复杂，是难以查出错误的根源的。如果每个函数都检查参数，就能较快地找出“罪魁祸首”。

提示 4-10：编程时合理地利用 assert 宏，将给调试带来很大的方便。

总而言之，在实际的系统中，“一个地方的参数错误就引起整个程序异常退出”是不可取的，在编写和调试算法程序中，assert 会“迫使”我们编写出更高质量的程序。

4.2 地址和指针

4.1 节介绍的数学函数的特点是：做计算，然后返回一个值。有时候，我们要做的事情并不是“计算”——如交换两个变量；而有时候，我们需要返回两个甚至更多的值——如解一个二元一次方程组。

4.2.1 变量交换

程序 4-4 用函数交换变量（错误）

```
#include<stdio.h>
void swap(int a, int b)
{
  int t = a; a = b; b = t;
}

int main()
{
  int a = 3, b = 4;
  swap(3, 4);
  printf("%d %d\n", a, b);
  return 0;
}
```

你应当还记得，这就是三变量交换算法，其中 void 是“无值”的意思。返回“无值型”的函数没有返回值，可以用不带参数的 return 语句退出，也可以在函数体执行完毕后自然退

出。下面我们测试一下这个函数好不好用。很不幸，输出是 3 4，而不是 4 3。事实上，a 和 b 并没有被交换。为什么会这样呢？为了理解它，请回忆赋值。“诡异”的赋值语句 a = a+1 是这样解释的：分为两步，首先计算赋值符号右边的 a+1，然后把它装入变量 a，覆盖它原来的值。那函数调用的过程又是怎样的呢？

第一步，计算参数的值。在上面的例子中，因为 a=3，b=4，所以 swap(a,b)等价于 swap(3, 4)。这里的 3 和 4 被称为实际参数（简称实参）。

第二步，把实参赋值给函数声明中的 a 和 b。注意，这里的 a 和 b 与调用时的 a 和 b 是完全不同的。前面已经说过，实参最后将算出具体的值，swap 函数知道调用它的参数是 3 和 4，却不知道它们是怎么算出来的。函数声明中的 a 和 b 称为形式参数（简称形参）。

稍等一下，这里有个问题！这样一来，程序里有两个变量 a，一个在 main 函数里定义，一个是 swap 的形参，二者不会混淆吗？不会。函数（包括 main 函数）的形参和在该函数里定义的变量都被称为该函数的局部变量（local variable）。不同函数的局部变量相互独立——你无法访问其他函数的局部变量。需要注意的是，局部变量的存储空间是临时分配的，函数执行完毕时，局部变量的空间将被释放，其中的值无法保留到下次使用。与此对应的是全局变量（global variable）：它在函数外声明，可以在任何时候，由任何函数访问。需要注意的是，全局变量非常危险，应该谨慎使用。

提示 4-11：*函数的形参和在函数内声明的变量都是该函数的局部变量。无法访问其他函数的局部变量。局部变量的存储空间是临时分配的，函数执行完毕时，局部变量的空间将被释放，其中的值无法保留到下次使用。在函数外声明的变量是全局变量，它们可以被任何函数使用。操作全局变量有风险，应谨慎使用。*

这样一来，函数的调用过程就可以简单说成：计算实参的值，赋值给对应的形参，然后把“当前代码行”转移到函数的首部。换句话说，在 swap 函数刚开始执行时，局部变量 a=3，b=4，二者的值是在函数调用时，由实参复制而来。

那么执行完毕后，函数又干了些什么呢？把返回值扔给调用它的函数，然后再次修改“当前代码行”，恢复到调用它的地方继续执行。等一下！函数是如何知道该返回到哪里继续执行的呢？为了解释这一问题，我们需要暂时把讨论变得学术一些——不要紧张，很快就会结束。

4.2.2　调用栈

还记得在讲解 for 循环时，笔者是如何建议的吗？多演示程序执行的过程，把注意力集中在“当前代码行”的转移和变量值的变化。这个建议同样适用于对函数的学习，只是要加一样东西——调用栈（Call Stack）。

调用栈描述的是函数之间的调用关系。它由多个栈帧（Stack Frame）组成，每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量，因而不仅能在执行完毕后找到正确的返回地址，还很自然地保证了不同函数间的局部变量互不相干——因为不同函数对应着不同的栈帧。

提示 4-12：C 语言用调用栈（Call Stack）来描述函数之间的调用关系。调用栈由栈帧（Stack Frame）组成，每个栈帧对应着一个未运行完的函数。在 gdb[①]中可以用 backtrace（简称 bt）命令打印所有栈帧信息。若要用 p 命令打印一个非当前栈帧的局部变量，可以用 frame 命令选择另一个栈帧。

在继续学习之前，建议读者试着调试一下刚才几个程序，除了关心“当前代码行”和变量的变化之外，再看看调用栈的变化。强烈建议你在执行完 swap 函数的主体但还没有返回 main 函数之前，停下来看看 swap 和 main 函数所对应的栈帧中 a 和 b 的值。如果受条件所限制，在阅读到这里时没有办法完成这个实验，下面给出了用 gdb 完成上述操作的命令和结果。

第一步：编译程序

```
gcc 4-4.c -g
```

生成可执行程序 a.exe（在 Linux 下是 a.out）。编译选项-g 告诉编译器生成调试信息。

第二步：运行 gdb

```
gdb a.exe
```

这样，gdb 在运行时会自动装入刚才生成的可执行程序。

第三步：查看源码

```
(gdb) l
1       #include<stdio.h>
2       void swap(int a, int b){
3         int t = a; a = b; b = t;
4       }
5
6       int main(){
7         int a = 3, b = 4;
8         swap(3, 4);
9         printf("%d %d\n", a, b);
10        return 0;
```

这里(gdb)是 gdb 的提示符，字母 l 是输入的命令，它是 list（列出程序清单）的缩写。正如我们所看到的，swap 函数的最后一行是第 4 行，当执行到这一行时，swap 函数的主体已经结束，但函数还没有返回。

第四步：加断点并运行

```
(gdb) b 4
Breakpoint 1 at 0x401308: file 4-4.c, line 4.
(gdb) r
Starting program: D:\a.exe
```

[①] gdb 是一个功能强大的源码级调试器，虽然是基于命令的文本界面，但用熟了以后非常方便。关于 gdb 更多的介绍参见附录。

```
Breakpoint 1, swap (a=4, b=3) at 4-4.c:4
4       }
```

其中 b 命令把断点设在了第 4 行，r 命令运行程序，之后碰到了断点并停止。接下来，让我们来看看调用栈吧。

第五步：查看调用栈

```
(gdb) bt
#0  swap (a=4, b=3) at 4-4.c:4
#1  0x00401356 in main () at 4-4.c:8
(gdb) p a
$1 = 4
(gdb) p b
$2 = 3
(gdb) up
#1  0x00401356 in main () at 4-4.c:8
8         swap(3, 4);
(gdb) p a
$3 = 3
(gdb) p b
$4 = 4
```

这一步是关键。根据 bt 命令，调用栈中包含两个栈帧：#0 和#1，其中 0 号是当前栈帧——swap 函数，1 号是它的“上一个”栈帧——main 函数。在这里我们甚至能看到 swap 函数的返回地址 0x00401356，尽管我们看不懂它的具体含义。

使用 p 命令可以打印变量值。我们先查看了当前栈帧中 a 和 b 的值，分别等于 4 和 3——这正是用三变量法交换后的结果。接下来用 up 命令选择上一个栈帧，再次使用 p 命令查看 a 和 b 的值，这次却是 3 和 4 了——它们是 main 函数中的 a 和 b。前面说过，在函数调用时，它们只起到了“计算实参数”的作用。但实参被赋值到形参之后，main 函数中的 a 和 b 也完成了它们的使命。swap 函数甚至无法知道 main 函数中也有着和形参同名的 a 和 b 变量，当然也就无法修改它们。最后别忘了用 q 命令退出 gdb。

花了这么多篇幅解释调用栈和栈帧，你也许会觉得有点不值，但无数的经验告诉笔者：理解它们对于今后的学习和编程是至关重要的，特别是递归——初学者学习语言的最大障碍之一，调用栈将大大帮助理解。

4.2.3　用指针实现变量交换

是时候回到我们的 swap 函数中来了：好不容易讲清楚了为什么刚才的 swap 不能奏效，那应该如何编写 swap 函数呢？答案是用指针。

程序 4-5　用函数交换变量（正确）

```
#include<stdio.h>
void swap(int* a, int* b)
```

```
{
  int t = *a; *a = *b; *b = t;
}

int main()
{
  int a = 3, b = 4;
  swap(&a, &b);
  printf("%d %d\n", a, b);
  return 0;
}
```

怎么样，是不是觉得不太习惯，却又有点似曾相识呢？不太习惯的是 int 和 a 中间的“乘号”，而似曾相识的是 swap(&a, &b)这种“变量名前面加&”的用法——到目前为止，唯一采取这种用法的是 scanf 系列函数，而只有它改变了实参的值！

变量名前面加&得到的是该变量的地址。什么叫“地址”呢？

提示 4-13：C 语言的变量都是放在内存中的，而内存中的每个字节都有一个称为地址（address）的编号。每个变量都占有一定数目的字节（可用 sizeof 运算符获得），其中第一个字节的地址称为变量的地址。

下面用 gdb 来调试上面的程序，看看它和程序 4-4 到底有什么不同。前四步是一样的，直接看看调用栈。

```
(gdb) bt
#0  swap (a=0x22ff74, b=0x22ff70) at 4-5.c:4
#1  0x0040135c in main () at 4-5.c:8
(gdb) p a
$1 = (int *) 0x22ff74
(gdb) p b
$2 = (int *) 0x22ff70
(gdb) p *a
$3 = 4
(gdb) p *b
$4 = 3
(gdb) up
#1  0x0040135c in main () at 4-5.c:8
8         swap(&a, &b);
(gdb) p a
$5 = 4
(gdb) p b
$6 = 3
(gdb) p &a
$7 = (int *) 0x22ff74
```

```
(gdb) p &b
$8 = (int *) 0x22ff70
```

在打印 a 和 b 的值时，得到了“诡异”的东西——(int *) 0x22ff74 和(int *) 0x22ff70。数值 0x22ff74 和 0x22ff70 是两个地址（以 0x 开头的整数以十六进制表示，在这里暂时不需了解细节），而前面的(int *)告诉我们：a 和 b 是指向 int 类型的指针。

提示 4-14：*用 int* a 声明的变量 a 是指向 int 型变量的指针。赋值 a = &b 的含义是把变量 b 的地址存放在指针 a 中，表达式*a 代表 a 指向的变量，它既可以放在赋值符号的左边（左值），也可以放在右边（右值）。*

注意：*a 是指“a 指向的变量”，而不仅是“a 指向的变量所拥有的值”。理解这一点相当重要。例如，*a = *a + 1 就是让 a 指向的变量自增 1。你甚至可以把它写成(*a)++。注意不要写成*a++，因为++运算符的优先级高于“取内容”运算符*，实际上会被解释成*(a++)。

很复杂是吗？是的！有了指针，C 语言突然变得复杂了很多。一方面，你需要了解更多底层的东西才能彻底解释一些问题——包括运行时的地址空间布局，以及操作系统的内存管理方式等。另一方面，指针的存在，使得 C 语言中变量的说明变得异常复杂了——你能轻易地说出用 char * const *(*next)()声明的 next 是什么类型的吗[①]？毫不夸张地说，指针是程序员（不仅是初学者）杀手。

既然如此，我们应当如何使用指针呢？别忘了本书的背景——算法竞赛。算法竞赛的核心是算法，我们没有必要纠缠如此复杂的语言特性。了解底层的细节是有益的（事实上，我们已经介绍了一些底层细节！），但我们在编程时应尽量避开，只遵守一些注意事项即可。

提示 4-15：*千万不要滥用指针，这不仅会把自己搞糊涂，还会让程序产生各种奇怪的错误。事实上，本书的程序会很少使用指针。*

再次回到我们的主题：对正确 swap 程序的调试。在 swap 程序中，a 和 b 都是局部变量，在函数执行完毕以后就不复存在了，但是 a 和 b 里保存的地址却依然有效——它们是 main 函数中的局部变量 a 和 b 的地址。在 main 函数执行完毕之前，这两个地址将始终有效，并且分别指向 main 函数的局部变量 a 和 b。程序交换的是*a 和*b，也就是 main 函数中的局部变量 a 和 b。

4.2.4　初学者易犯的错误

这个 swap 函数看似简单，但初学者还是很容易写错。一种典型的错误写法是：

```
void swap(int* a, int* b)
{
  int *t = a; a = b; b = t;
}
```

[①] 它是一个指向函数的指针，该函数返回一个指针，该指针指向一个只读的指针，此指针指向一个字符变量。

它交换了 swap 函数的局部变量 a 和 b（辅助变量 t 必须是指针。int t = a 是错误的），但却始终没有修改它们指向的内容，因此 main 函数中的 a 和 b 不会改变。另一种错误写法是：

```
void swap(int* a, int* b)
{
  int *t;
  *t = *a; *a = *b; *b = *t;
}
```

这个程序错在哪里？t 是一个指向 int 型的指针，因此*t 是一个整数。用一个整数作为辅助变量去交换两个整数有何不妥？事实上，如果你用这个函数去替换程序 4-5，很可能会得到“4 3”的正确结果。为什么我要坚持说它是错误的呢？

问题在于，t 存储的地址是什么？也就是说，t 指向哪里？因为 t 是一个变量（指针也是一个变量，只不过类型是“指针”而已），所以根据规则，它在赋值之前是不确定的。如果这个“不确定的值”所代表的内存单元恰好是能写入的，那么这段程序将正常工作；但如果它是只读的，程序可能会崩溃！不信的话，赋初值 int *t = 0 试试，看看内存地址“0”到底能不能写。

花了这么多篇幅，我们终于初步理解了地址和指针——尽管只是初步理解，但是为将来的学习奠定了良好的基础。指针有很多巧妙但又令人困惑的用法。如果你曾听别人说过有一种语法，但在完整地学习本书的过程中始终没有看到它被用过一次，那么这通常意味着这个语法不必学（至少在算法竞赛中不必用到）。事实上，笔者在编写本书的例程时首先考虑通俗易懂，避开复杂的语言特性，其次才是简洁和效率。

4.3 递　　归

终于到了本书 C 语言部分的最后一站——递归。很多人都认为递归是语言中最难理解的东西之一，但也不要紧张：如果认真理解了 4.2 节中的指针、地址和调用栈，你会发现递归其实是一个很自然的东西。

4.3.1 递归定义

递归的定义如下：

递归：

参见“递归”。

什么？这个定义什么也没有说啊！好吧，改一下：

递归：

如果你还是没明白递归是什么意思的话，参见“递归”。

噢，也许这次你明白了：原来递归就是“自己用到自己”的意思。这个定义显然比上一个要好些，因为当你终于悟出其中的道理后，就不必继续“参见”下去了。事实上，递归的含义比这要广——

A 经理：“这事不归我管，去找 B 经理。”于是你去找 B 经理。

B 经理：“这事不归我管，去找 A 经理。”于是你回到了 A 经理这儿。

接下来发生的事情就不难想到了：只要两个经理的说辞不变，你又始终听话，你将会永远往返于两个经理之间。这叫无限递归（Infinite Recursion）。尽管在这里，A 经理并没有让你找他自己，但还是回到了他这里。换句话说，“间接地用到自己”也算递归。

回忆一下，正整数是如何定义的？正整数是 1，2，3，…这些数。这样的定义也许对于小学生来说是没有任何问题的，但当你开始觉得这个定义“不太严密”时，你或许会喜欢这样的定义：

（1）1 是正整数。

（2）如果 n 是正整数，n+1 也是正整数。

（3）只有通过（1）、（2）定义出来的才是正整数[①]。

这样的定义也是递归的：在“正整数”还没有定义完时，就用到了“正整数”的定义！这和前面的“参见递归”在本质上是相同的，只是没有它那么直接、那么明显。

同样地，可以递归定义“常量表达式”（以下简称表达式）：

（1）整数和浮点数都是表达式。

（2）如果 A 是表达式，则（A）是表达式。

（3）如果 A 和 B 都是表达式，则 A+B、A-B、A*B、A/B 都是表达式。

（4）只有通过 1、2、3 定义出来的才是表达式。

简洁而严密，这就是递归定义的优点。

4.3.2　递归函数

数学函数也可以递归定义：阶乘函数 $f(n)=n!$ 可以定义为：

$$\begin{cases} f(0)=1 \\ f(n)=f(n-1)\times n \quad (n\geqslant 1) \end{cases}$$

对应的程序为：

程序 4-6　用递归法计算阶乘

```
#include<stdio.h>
int f(int n)
{
  return n == 0 ? 1 : f(n-1)*n;
}
int main()
{
  printf("%d\n", f(3));
  return 0;
}
```

① 更严密的说法是：正整数集是满足（1）、（2）的最小集。这里牺牲一点严密性，换来的是更通俗易懂的表达方式。

这里出现了一个小东西——三元运算符“?:”。a?b:c 的含义是：如果 a 为真，则表达式的值是 b，否则是 c。所以 n == 0 ? 1 : f(n-1)*n 很好地表达了刚才的递归定义。

提示 4-16：C 语言支持递归——函数可以直接或间接地调用自己。但要注意为递归函数编写终止条件，否则将产生无限递归。

4.3.3 C 语言对递归的支持

尽管从概念上可以理解阶乘的递归定义，但在 C 语言中函数为什么真的可以“自己调用自己”呢？我们再次借助 gdb 来调试这段程序。

首先用 b f 命令设置断点——除了可以按行号设置外，也可以直接给出函数名，断点将设置在函数的开头。下面用 r 命令运行程序，并在断点处停下来。接下来用 s 命令单步执行：

```
(gdb) r
Starting program: C:\a.exe

Breakpoint 1, f (n=3) at 4-6.c:3
3       return n == 0 ? 1 : f(n-1)*n;
(gdb) s

Breakpoint 1, f (n=2) at 4-6.c:3
3       return n == 0 ? 1 : f(n-1)*n;
(gdb) s

Breakpoint 1, f (n=1) at 4-6.c:3
3       return n == 0 ? 1 : f(n-1)*n;
(gdb) s

Breakpoint 1, f (n=0) at 4-6.c:3
3       return n == 0 ? 1 : f(n-1)*n;
(gdb) s
4       }
```

看到了吗？在第一次断点处，n=3（3 是 main 函数中的调用参数），接下来将调用 f(3-1)即 f(2)，因此单步一次后显示 n=2。由于 n==0 仍然不成立，继续递归调用，直到 n=0。这时不再递归调用了，一次 s 以后会到达函数的结束位置。

接下来该做什么？没错！好好看看我们的调用栈吧！

```
(gdb) bt
#0  f (n=0) at 4-6.c:4
#1  0x00401308 in f (n=1) at 4-6.c:3
#2  0x00401308 in f (n=2) at 4-6.c:3
#3  0x00401308 in f (n=3) at 4-6.c:3
#4  0x00401359 in main () at 4-6.c:6
```

```
(gdb) s
4       }
(gdb) bt
#0  f (n=1) at 4-6.c:4
#1  0x00401308 in f (n=2) at 4-6.c:3
#2  0x00401308 in f (n=3) at 4-6.c:3
#3  0x00401359 in main () at 4-6.c:6
(gdb) s
4       }
(gdb) bt
#0  f (n=2) at 4-6.c:4
#1  0x00401308 in f (n=3) at 4-6.c:3
#2  0x00401359 in main () at 4-6.c:6
(gdb) s
4       }
(gdb) bt
#0  f (n=3) at 4-6.c:4
#1  0x00401359 in main () at 4-6.c:6
(gdb) s
6
main () at 4-6.c:7
7         return 0;
(gdb) bt
#0  main () at 4-6.c:7
```

每次执行完 s 指令，都会有一层递归调用终止，直到返回 main 函数。事实上，如果在递归调用初期查看调用栈，你会发现每次递归调用都会多一个栈帧——和普通的函数调用并没有什么不同。确实如此。由于使用了调用栈，C 语言自然支持了递归。在 C 语言的函数中，调用自己和调用其他函数并没有任何本质区别，都是建立新栈帧，传递参数并修改“当前代码行”。在函数体执行完毕后删除栈帧，处理返回值并修改“当前代码行”。

提示 4-17：由于使用了调用栈，C 语言支持递归。在 C 语言中，调用自己和调用其他函数并没有本质不同。

如果仍然无法理解上面的调用栈，可以作如下的比喻。

皇帝（拥有 main 函数的栈帧）：大臣，你给我算一下 f(3)。

大臣（拥有 f(3)的栈帧）：知府，你给我算一下 f(2)。

知府（拥有 f(2)的栈帧）：县令，你给我算一下 f(1)。

县令（拥有 f(1)的栈帧）：师爷，你给我算一下 f(0)。

师爷（拥有 f(0)的栈帧）：回老爷，f(0)=1。

县令：（心算 f(1)=f(0)*1=1）回知府大人，f(1)=1。

知府：（心算 f(2)=f(1)*2=2）回大人，f(2)=2。

大臣：（心算 f(3)=f(2)*3=6）回皇上，f(3)=6。

皇帝满意了。

虽然比喻不甚恰当，但也可以说明一些问题。递归调用时新建了一个栈帧并且跳转到函数开头处执行，就好比皇帝找大臣、大臣找知府这样的过程。尽管同一时刻可以有多个栈帧（皇帝、大臣、知府同时处于“等待下级回话”的状态），但“当前代码行”只有一个。

读者如果理解了这个比喻，但仍不理解调用栈，不必强求——知道递归为什么能正常工作就行。设计递归程序的重点在于给下级安排工作。

4.3.4 段错误与栈溢出

至此，对 C 语言的介绍已近尾声。别忘了，我们还没有测试 f 函数！也许你会说：不必了，我知道乘法会溢出——算阶乘时，乘法老是会溢出。可这次不一样了。把 main 函数的 f(3)换成 f(100000000)试试（别数了，有 8 个零）。什么？没有输出？不对呀，即使溢出，也应该是个负数或者其他“显然不对”的值，不应该没有输出啊！

gdb 再次帮了我们的忙。用-g 编译后用 gdb 载入，二话不说就用 r 执行。你猜怎么着？gdb 报错了！

```
(gdb) r
Starting program: C:\a.exe

Program received signal SIGSEGV, Segmentation fault.
0x00401303 in f (n=99869708) at 4-6.c:3
3       return n == 0 ? 1 : f(n-1)*n;
```

gdb 说：程序收到了 SIGSEGV 信号：段错误。这太让人沮丧了！眼看本章就要结束了，怎么又遇到个段错误？别急，让我们慢慢分析。我保证，这是本章最后的难点。

你有没有想过，编译后产生的可执行文件里都保存着什么内容？答案是：和操作系统相关。例如，UNIX/Linux 用的 ELF 格式，DOS 下用的是 COFF 格式，而 Windows 用的是 PE 文件格式（它由 COFF 扩充而来）。这些格式不尽相同，但都有一个共同的概念——段。

“段”（segmentation）是指二进制文件内的区域，所有某种特定类型信息被保存在里面。可以用 size 程序[①]得到可执行文件中各个段的大小。如刚才的 4-6.c，编译出 a.exe 以后执行 size 的结果是：

```
D:\>size a.exe
   text    data     bss     dec     hex filename
   2756     740     224    3720     e88 a.exe
```

此结果表示 a.exe 由正文段、数据段和 bss 段组成，总大小是 3720，用十六进制表示为 e88。这些段是什么意思呢？

[①] Linux 和 Windows 下的 MinGW 中都有这个程序。

提示 4-18：在可执行文件中，正文段（Text Segment）储存指令，数据段（Data Segment）储存已初始化的全局变量，BSS 段（BSS Segment）储存未赋值的全局变量所需的空间。

觉得少了点什么？调用栈在哪里？它并不储存在可执行文件中，而是在运行时创建。调用栈所在的段称为堆栈段（Stack Segment）。和其他段一样，它也有自己的大小，不能被越界访问，否则就会出现段错误（Segmentation Fault）。

这样，前面的错误就不难理解了：每次递归调用都需要往调用栈里增加一个栈帧，久而久之就越界了。用术语把它叫做栈溢出（Stack Overflow）。

提示 4-19：在运行时，程序会动态创建一个堆栈段，里面存放着调用栈，因此保存着函数的调用关系和局部变量。

那么栈空间到底有多大呢？这和操作系统相关。在 Linux 中，栈大小是由系统命令 ulimit 指定的，例如 ulimit -a 显示当前栈大小，而 ulimit -s 32768 将把栈大小指定为 32MB。但在 Windows 中，栈大小是储存在可执行文件的。使用 gcc 可以这样指定可执行文件的栈大小：gcc -Wl,--stack=16777216[①]，这样栈大小就变为了 16MB。

提示 4-20：在 Linux 中，栈大小并没有储存在可执行程序中，只能用 ulimit 命令修改；在 Windows 中，栈大小储存在可执行程序中，用 gcc 编译时可以通过-Wl,--stack=<byte count>指定。

聪明的读者，现在你能理解为什么在介绍数组时，建议“把较大的数组放在 main 函数外”了吗？别忘了，局部变量也是放在堆栈段的。栈溢出不见得是递归调用太多，也可能是局部变量太大。只要总大小超过了允许的范围，就会产生栈溢出。

4.4　本 章 小 结

到目前为止，本书要介绍的 C 语言已经全部讲完了。本章涉及了整个 C 语言中最难理解的两个东西：指针和递归。

4.4.1　小问题集锦

首先，来编写一个函数 solve，给定浮点数 a, b, c, d, e, f，求解方程组 ax + by = c，dx + ey = f。

任务 1：使用 assert 宏，让解不唯一时异常退出。

任务 2：解不唯一时仍然正常返回，但调用者有办法知道解的数量（无解、唯一解、无穷多组解）。

思考：函数的参数都有哪些？各是什么类型？

[①] 实际上，栈大小是由连接程序 ld 指定的。gcc 编译参数-Wl 的作用正是把其后的参数（--stack=<size>）传递给 ld。

然后，请编写一个程序，包含 3 个函数 f()、g()和 h()，3 个函数均无参数，返回值均为 int 类型。

任务 1：定义 int a, b，要求在依次执行 a = f()和 b = f()后，a 和 b 的值不同。

任务 2：定义 int a, b，要求在依次执行 a = (f()+g())+h()和 b=f()+(g()+h())后，a 和 b 的值不同。

思考：这个例子给你怎样的启示？

接下来做两个编程探索。注意，请不要上网搜索、求助或者查阅资料。自己通过实验得来的知识是最不容易忘记的。

问题 1：局部变量是否可以和全局变量重名？如果可以，实际上使用的是哪个？这可能会引起什么样的难以察觉到的错误？

问题 2：如果在函数中声明一个局部变量，然后返回它的地址，调用者获取该地址时，该地址是否是有效的？为什么？

下面来点抽象的。size 命令给出的是运行时的 BSS 段大小，但在可执行文件中，只保存每个变量所需的空间，并不储存它们的映象；相反，已初始化的全局变量必须把映象保存在可执行文件中。

实验 1：声明全局变量 int a[1000000]，不要声明其他变量，编译后查看可执行文件大小，并用 size 命令查看各个段的大小。

实验 2：声明全局变量 int a[1000000]={1}，不要声明其他变量，编译后查看可执行文件大小，并用 size 命令查看各个段的大小。如果把初值改成{0}呢？

实验 3：用-g（调试信息）和-O2（二级优化）编译任务 1 和任务 2 的代码。可执行文件的大小如何变化？各段的大小如何变化？

4.4.2 小结

指针还有很多相关内容我们没有介绍，例如指向 void 型的指针、指向函数的指针、指向常量的指针以及指针和数组之间的关系（注意，尽管在很多地方可以混用，但指针和数组不是一回事！《C 语言程序设计奥秘》花了一章的篇幅来叙述二者的区别）。正如书中所说，本书将尽量回避指针，但尽管如此，调试并理解 4.2 节中几个 swap 函数的工作方式对于理解计算机的工作原理大有好处。

递归需要从概念和语言两个方面理解。从概念上，递归就是“自己使用自己”的意思。递归调用就是自己调用自己，递归定义就是自己定义自己……当然，这里的“使用自己”可以是直接的，也可以是间接的。很多初学者在学习递归时专注于表象，从而未能透彻理解其“计算机”本质。由于我们的重点是设计算法和编写程序，理解递归函数的执行过程是非常重要的。因此，本章大量使用了 gdb 作为工具讲解内部机理，即使读者在平时编程时不用 gdb 调试，在学习初期用它帮助理解也是大有裨益的。关于 gdb 的更多介绍参见附录。

第2部分 算 法 篇

第5章 基础题目选解

学习目标

- ☑ 学会用常量表简化代码
- ☑ 学会用状态变量辅助字符串读入
- ☑ 学会用结构体定义高精度整数，并设计构造函数、复制构造函数和输入输出方法
- ☑ 学会为结构体定义“小于”运算符，并用它定义其他比较运算符
- ☑ 熟练掌握冒泡排序和顺序检索
- ☑ 熟练掌握用 qsort 库函数给整数和字符串排序的方法
- ☑ 熟练掌握小规模素数表的构造方法
- ☑ 熟练掌握素因子分解的方法
- ☑ 熟练掌握三角形有向面积的意义和计算方法
- ☑ 完成一定数量的编程练习

在算法竞赛中，编程能力是非常重要的。算法设计得再好，如果程序写不出来就是零分；即使程序写出来了，也可能会因为细小的错误导致丢失大量的得分。本章通过一定数量和类型的例题和习题让读者尽快熟悉常见的编程技巧，为接下来的算法学习打下坚实的基础。在本章中，程序的正确性是第一位的。

5.1 字 符 串

很多字符串题目都是直观的——很容易看出你的答案是否正确。

5.1.1 WERTYU

把手放在键盘上时，稍不注意就会往右错一位。这样的话，Q 会变成 W，J 会变成 K 等。电脑键盘如图 5-1 所示。

图 5-1 电脑键盘

输入一个错位后敲出的字符串，输出打字员本来想打出的句子。

样例输入：O S, GOMR YPFSU/

样例输出：I AM FINE TODAY.

【分析】

每输入一个字符，都可以直接输出一个字符。问题在于：如何进行这样的变换呢？一种方法是使用 if 语句或者 switch 语句，如 if(c == 'W') putchar('Q')。但很明显，这样做太麻烦。一个较好的方法是使用常量数组，下面是完整程序：

```
#include <stdio.h>
char *s = "`1234567890-=QWERTYUIOP[]\\ASDFGHJKL;'ZXCVBNM,./";
int main()
{
  int i,c;
  while ((c = getchar()) != EOF)
  {
     for (i=1; s[i] && s[i]!=c; i++);
     if (s[i]) putchar(s[i-1]);
     else putchar(c);
  }
  return 0;
}
```

5.1.2 TeX 括号

在 TeX 中，左双引号是``，右双引号是''。输入一篇包含双引号的文章，你的任务是把它转换成 TeX 的格式。

样例输入：

"To be or not to be," quoth the Bard, "that
is the question".

样例输出：

``To be or not to be,'' quoth the Bard, ``that
is the question''.

【分析】

本题的关键是，如何判断一个双引号是“左”双引号还是“右”双引号。方法很简单：使用一个标志变量即可。

```
#include <stdio.h>
int main()
{
  int c, q = 1;
  while((c= getchar()) != EOF)
  {
```

```
    if(c == '"') { printf("%s", q ? "``" : "''"); q = !q; }
    else printf("%c", c);
  }
  return 0;
}
```

5.1.3　周期串

如果一个字符串可以由某个长度为 k 的字符串重复多次得到，我们说该串以 k 为周期。例如，abcabcabcabc 以 3 为周期（注意，它也以 6 和 12 为周期）。输入一个长度不超过 80 的串，输出它的最小周期。

样例输入：HoHoHo

样例输出：2

【分析】

题目中说过，字符串可能会有多个周期。但因为只需求出最小的一个，可以从小到大枚举各个周期，一旦符合条件就立即输出。判断一个周期是否合法有很多方法，下面是一个相对简单易懂的，读者可以尝试其他方法。

下面的程序用到了一个新语法：临时定义变量。例如，变量 i 和 j 只定义在循环体内，因此在循环体后无法访问到它们。在使用前才定义变量能让程序更加清晰。但需要注意的是，这个语法并不是 ANSI C 的。建议把程序的扩展名保存成.cpp。

```
#include <stdio.h>
#include <string.h>
int main()
{
  char word[100];
  scanf("%s", word);
  int len = strlen(word);
  for (int i = 1; i <= len; j++) if (len % i == 0)
  {
    int ok = 1;
    for (int j = i; j < len; j++)
      if (word[j] != word[j % i]) { ok = 0; break; }
    if (ok) { printf("%d\n", i); break; }
  }
  return 0;
}
```

5.2　高精度运算

在介绍 C 语言时，大家已经看到了很多整数溢出的情形。如果运算结果真的很大，需

要用所谓的高精度算法，用数组来储存整数，模拟手算的方法进行四则运算。

5.2.1 小学生算术

很多学生在学习加法时，发现“进位”特别容易出错。你的任务是计算两个整数在相加时需要多少次进位。你编制的程序应当可以连续处理多组数据，直到读到两个0（这是输入结束标记）。假设输入的整数都不超过9个数字。

样例输入：

123 456
555 555
123 594
0 0

样例输出：

0
3
1

【分析】

注意int的上限约是2000000000，可以保存所有9位整数，因此可以用整数来保存输入。每次把a和b分别模10就能获取它们的个位数。程序如下：

```
#include <stdio.h>
int main()
{
  int a, b;
  while (scanf("%d%d",&a,&b) == 2)
  {
    if (!a && !b) return;
    int c = 0, ans = 0;
    for (int i = 9; i>=0; i--)
    {
      c = (a%10 + b%10 + c) > 9 ? 1 : 0;
      ans += c;
      a /= 10; b /= 10;
    }
    printf("%d\n", ans);
}
  return 0;
}
```

5.2.2 阶乘的精确值

输入不超过1000的正整数n，输出$n!=1\times2\times3\times\cdots\times n$的精确结果。

样例输入：30

样例输出：265252859812191058636308480000000

【分析】

为了保存结果，需要先分析 1000!有多大。用计算器算一算不难知道，1000!约等于 $4*10^{2567}$，因此可以用一个 3000 个元素的数组 f 保存。为了方便起见，我们让 f[0]保存结果的个位，f[1]是十位，f[2]是百位……（为什么要逆序表示呢？因为如果按照从高到低的顺序储存，一旦进位的话就……），则每次只需要模拟手算即可完成 n!。在输出时需要忽略前导 0。注意，如果结果本身就是 0，那么忽略所有前导 0 后将什么都不输出。所幸 n!肯定不等于 0，因此本题中可以忽略这个细节。程序如下：

```
#include<stdio.h>
#include<string.h>
const int maxn = 3000;
int f[maxn];
int main()
{
  int i, j, n;
  scanf("%d", &n);
  memset(f, 0, sizeof(f));
  f[0] = 1;
  for(i = 2; i <= n; i++)
  { // 乘以 i
    int c = 0;
    for(j = 0; j < maxn; j++)
    {
      int s = f[j] * i + c;
      f[j] = s % 10;
      c = s / 10;
    }
  }
  for(j = maxn-1; j >= 0; j--) if(f[j]) break; // 忽略前导 0
  for(i = j; i >= 0; i--) printf("%d", f[i]);
  printf("\n");
  return 0;
}
```

5.2.3　高精度运算类 bign

用上面的方法可以很方便地实现高精度运算，但代码有些难以复用。能否写一个“高精度函数库”以方便调用呢？在 ACM/ICPC 这样可以使用“代码模板”的比赛中，现成、好用的代码是很诱人的。即使是 IOI 这样不许带纸质材料的比赛中，把代码写成独立的模块也会让测试更加方便。

基于此，我们设计一个结构体 bign 来储存高精度非负整数：

```
const int maxn = 1000;
struct bign
{
  int len, s[maxn];
  bign() { memset(s, 0, sizeof(s)); len = 1; }
};
```

其中，len 表示位数，而 s 数组就是具体的各个数字。例如，若是要表示 1234，则 len=4，s[0]=4，s[1]=3，s[2]=2，s[1]=1。

上面的结构体中有一个函数，称为构造函数（Constructor）。构造函数是 C++中特有的，作用是进行初始化。事实上，当定义 bign x 时，就会执行这个函数，把 x.s 清零，并赋值 x.len = 1。需要说明的是，在 C++中，并不需要 typedef 就可以直接用结构体名来定义，而且还提供“自动初始化”的功能。从这个意义上说，C++比 C 语言方便。

好了，让我们重新定义赋值运算符（注意，下面的函数要写在 bign 结构体定义的内部，千万不要写在外面）：

```
bign operator = (const char* num)
{
  len = strlen(num);
  for(int i = 0; i < len; i++) s[i] = num[len-i-1] - '0';
  return *this;
}
```

有了它，就可以用 x = “123456789876543212345678 9”这样的方式来给 x 赋值了，它会把这个字符串转化为“逆序数组+长度”的内部表示方法。为了让它支持 x=1234 这样更常用的赋值方式，还需要再定义另外一种赋值运算（仍然定义在结构体内部）：

```
bign operator = (int num)
{
  char s[maxn];
  sprintf(s, "%d", num);
  *this = s;
  return *this;
}
```

如果做过实验，会发现虽然可以用“bign x; x = 100;”来声明一个 x 并给它赋值，却不能把它写成“bign x = 100;”。原因在于，bign x = 100 是初始化，而非普通的赋值操作。为了让代码支持“初始化”操作，需要增加两个函数（还是定义在结构体内部）：

```
bign(int num) { *this = num; }
bign(const char* num) { *this = num; }
```

如你所见，它们只是简单地调用刚才的赋值运算符。接下来需要提供一个函数把它转化为字符串，像这样（定义在结构体内部）：

```
string str() const
{
  string res = "";
  for(int i = 0; i < len; i++) res = (char)(s[i] + '0') + res;
  if(res == "") res = "0";
  return res;
}
```

注意函数定义后的 const 关键字，它表明“x.str()不会改变 x”。瞧，x.str()是变量 x 的“专属函数”，这也是 C++特有的语法。在这里，str()是结构体 bign 的成员函数（member function）。

定义和使用成员函数的方法和普通函数类似，但成员函数可以直接使用结构体中的域。例如，在 str()函数中，我们用到了 len 和 s。在调用 x.str()时，这里的 len 和 s 实际上是 x.len 和 x.s；如果调用的是 y.str()，那么这里的 len 和 s 就是 y.len 和 s.len。对于成员函数来说，this 是指向当前对象的指针。也就是说，调用 x.str()时，this 的值是&x。这样，就不难理解为什么初始化函数要写成*this = num 了。

接下来，重新定义>>和<<运算符，让输入输出流直接支持我们的 bign 结构体（这两个函数要定义在结构体 bign 的外边，不要写在里面）：

```
istream& operator >> (istream &in, bign& x)
{
  string s;
  in >> s;
  x = s.c_str();
  return in;
}

ostream& operator << (ostream &out, const bign& x)
{
  out << x.str();
  return out;
}
```

这样，就可以用 cout << x 的方式打印它。怎么样，很方便吧！

5.2.4　重载 bign 的常用运算符

加法的原理早在小学就学过，而且刚才再次演练过。下面只需把它写成程序（定义在结构体内部）：

```
bign operator + (const bign& b) const
{
  bign c;
  c.len = 0;
```

```
  for(int i = 0, g = 0; g || i < max(len, b.len); i++)
  {
    int x = g;
    if(i < len) x += s[i];
    if(i < b.len) x += b.s[i];
    c.s[c.len++] = x % 10;
    g = x / 10;
  }
  return c;
}
```

只要任何一个数还有数字，加法就要继续进行；即使两个数都加完了，也不要忘记处理进位。注意，上面的算法并没有假设 s 数组中“当前没有用到的数字都是 0”。如果事先已经清零，就可以把循环体的前 3 行简化为 int x = s[i]+b.s[i]+g。甚至可以直接把循环次数设置 max(len,b.len)+1，然后检查最终结果是否有前导零。如果有，则把 len 减 1。为了让使用更加简单，还可以重新定义+=运算符（定义在结构体内部）：

```
bign operator += (const bign& b)
{
  *this = *this + b;
  return *this;
}
```

减法、乘法和除法原理类似，这里不再赘述。接下来，让我们来实现“比较”操作（定义在结构体内部）：

```
bool operator < (const bign& b) const
{
  if(len != b.len) return len < b.len;
  for(int i = len-1; i >= 0; i--)
  if(s[i] != b.s[i]) return s[i] < b.s[i];
  return false;
}
```

一开始就比较两个 bign 的位数，如果不相等则直接返回，否则比较两个数组的逆序的字典序。注意，这样做的前提是两个数都没有前导零，如果不注意的话，很可能出现“运算结果都没问题，但一比较就错”的情况。

只需定义“小于”这一个符号，即可用它定义其他所有比较运算符：

```
bool operator > (const bign& b) const{ return b < *this; }
bool operator <= (const bign& b) const{ return !(b < *this); }
bool operator >= (const bign& b) const{ return !(*this < b); }
bool operator != (const bign& b) const{ return b < *this || *this < b; }
bool operator == (const bign& b) const{ return !(b < *this) && !(*this < b); }
```

当然，也可以同时用<和>把!=和==定义得更加简单，读者可以自行尝试。在本书后面

需要用到高精度运算的题目中，我们将直接使用 bign 类中的所有运算（包括这里略去的），读者可以参考本书配套资料上的完整源代码。

5.3　排序与检索

数据处理是计算机的强项，包括排序、检索和统计等。这里举一些经典的例子，向读者展示排序和检索的技巧。

5.3.1　6174 问题

假设你有一个各位数字互不相同的四位数，把所有数字从大到小排序后得到 a，从小到大排序后得到 b，然后用 a-b 替换原来这个数，并且继续操作。例如，从 1234 出发，依次可以得到 4321-1234=3087、8730-378=8352、8532-2358=6174。有趣的是，7641-1467=6174，回到了它自己。

输入一个 n 位数，输出操作序列，直到出现循环（即新得到的数曾经得到过）。输入保证在循环之前最多只会产生 1000 个整数。

样例输入：1234

样例输出：1234 -> 3087 -> 8352 -> 6174 -> 6174

【分析】

两个问题摆在我们面前：如何得到下一个数？如何检查这个数是否已经出现过？让我们一一解决。

第一个问题需要我们把各个数字排序，因此首先需要把各个数字提取出来。下面的函数使用一种称为“冒泡排序”的方法，可以方便地把一个数组按照从小到大或者从大到小的顺序排序：

```
int get_next(int x)
{
  int a, b, n;
  char s[10];
  // 转化成字符串
  sprintf(s, "%d", x);
  n = strlen(s);
  // 冒泡排序
  for(int i = 0; i < n; i++)
    for(int j = i+1; j < n; j++)
      if(s[i] > s[j])
      {
        char t = s[i]; s[i] = s[j]; s[j] = t;
      }
  sscanf(s, "%d", &b);
```

```
  // 字符串反转
  for(int i = 0; i < n/2; i++)
  {
    char t = s[i]; s[i] = s[n-1-i]; s[n-1-i] = t;
  }
  sscanf(s, "%d", &a);
  return a - b;
}
```

有的读者可能会尝试用函数 strrev 来完成字符串的反转操作，但请注意，strrev 函数不是 ANSI C 的（请用-ansi 编译试试看）。

第二步是逐个生成各个数，并判断是否曾经生成过。常用的方法是用数组：

```
int num[2000], count;
int main()
{
  scanf("%d", &num[0]);
  printf("%d", num[0]);
  count = 1;
  for(;;)
  {
    // 生成并输出下一个数
    num[count] = get_next(num[count-1]);
    printf(" -> %d", num[count]);
    // 在数组 num 中寻找新生成的数
    int found = 0;
    for(int i = 0; i < count; i++)
        if(num[i] == num[count]) { found = 1; break; }
    // 如果找到，则退出循环
    if(found) break;
    count++;
  }
  printf("\n");
  return 0;
}
```

5.3.2 字母重排

输入一个字典（用******结尾），然后再输入若干单词。每输入一个单词 *w*，你都需要在字典中找出所有可以用 *w* 的字母重排后得到的单词，并按照字典序从小到大的顺序在一行中输出（如果不存在，输出:()。输入单词之间用空格或空行隔开，且所有输入单词都由不超过 6 个小写字母组成。注意，字典中的单词不一定按字典序排列。

样例输入：

```
tarp given score refund only trap work earn course pepper part
******
resco nfudre aptr sett oresuc
```

样例输出：

```
score
refund
part tarp trap
:(
course
```

【分析】

首先需要把字典读入并保存下来。接下来需要怎么做呢？最容易想到的方法是这样的：

（1）每读入一个单词，就和字典中的所有单词比较，看看是否可以通过重排得到。

（2）把可以重排得到的单词放在一个数组中。

（3）把这个数组排序后输出。

这样做当然是正确的，但总显得有些“笨”。有没有更好的方法呢？下面依次看看这 3 个步骤能否简化。

如何判断两个单词是否可以通过重排得到呢？稍微思考就会发现：把各个字母排序，然后直接比较即可。既然如此，我们可以在读入时就把每个单词按照字母排好序，就不必每次重排了。

是不是必须把能重排的单词保存下来再排序呢？这个也没有必要——只要在读入字典之后把所有单词排序，就可以每遇到一个满足条件的单词就立刻输出。程序如下：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int n;
char word[2000][10], sorted[2000][10];
// 字符比较函数
int cmp_char(const void* _a, const void* _b)
{
  char* a = (char*)_a;
  char* b = (char*)_b;
  return *a - *b;
}

// 字符串比较函数
int cmp_string(const void* _a, const void* _b)
{
  char* a = (char*)_a;
```

```
  char* b = (char*)_b;
  return strcmp(a, b);
}

int main()
{
  n = 0;
  for(;;)
  {
    scanf("%s", word[n]);
    if(word[n][0] == '*') break;                    // 遇到结束标志就终止循环
    n++;
  }
  qsort(word, n, sizeof(word[0]), cmp_string);      // 给所有单词排序
  for(int i = 0; i < n; i++)
  {
    strcpy(sorted[i], word[i]);
    qsort(sorted[i], strlen(sorted[i]), sizeof(char), cmp_char);
                                                    // 给每个单词排序
  }

  char s[10];
  while(scanf("%s", s) == 1)                        // 持续读取到文件结尾
  {
    qsort(s, strlen(s), sizeof(char), cmp_char);    // 给输入单词排序
    int found = 0;
    for(int i = 0; i < n; i++)
      if(strcmp(sorted[i], s) == 0)
      {
        found = 1;
        printf("%s ", word[i]);                     // 输出原始单词，而不是排序
                                                       后的
      }
    if(!found) printf(":(");
    printf("\n");
  }
  return 0;
}
```

注意，不管是把字符串中的各个字符排序还是把所有字符串排序，上面的代码都用到了 stdlib.h 中的排序函数 qsort。使用库函数排序的代码量并不比用冒泡排序法小，但速度却快很多（我们将在后面的章节中阐述原因），请初学者熟悉它的用法。

5.4　数 学 基 础

数学是算法的基石。很多算法竞赛中都有一些数学味很浓的题。读者应从入门时就重视数学，逐步积累各种技巧。

5.4.1　Cantor 的数表

如下列数，第一项是 1/1，第二项是 1/2，第三项是 2/1，第四项是 3/1，第五项是 2/2，……。输入 n，输出第 n 项。

```
1/1  1/2  1/3  1/4  1/5
2/1  2/2  2/3  2/4
3/1  3/2  3/3
4/1  4/2
5/1
```

样例输入：

```
3
14
7
12345
```

样例输出：

```
2/1
2/4
1/4
59/99
```

【分析】

数表提示我们按照斜线分类。第 1 条斜线有 1 个数，第 2 条有 2 个数，第 3 条有 3 个数……第 i 条有 i 个数。这样，前 i 条斜线一共有 $S(k)=1+2+3+\cdots+k=\frac{1}{2}k(k+1)$ 个数。

n 在哪条斜线上呢？只要找到一个最小的正整数 k，使得 $n\leqslant S(k)$，那么 n 就是第 k 条斜线上的倒数第 $S(k)-n+1$ 个元素（最后一个元素是倒数第 1 个元素，而不是倒数第 0 个元素）。不难看出，第 k 条斜线的倒数第 i 个元素是 $i/(k+1-i)$。

```
#include<stdio.h>
int main()
{
  int n;
  while(scanf("%d", &n) == 1)
  {
```

```
    int k = 1, s = 0;
    for(;;)
    {
      s += k;
      if(s >= n) // 所求项是第 k 条对角线的倒数第 s-n+1 个元素
      {
        printf("%d/%d\n", s-n+1, k-s+n);
        break;
      }
      k++;
    }
  }
  return 0;
}
```

利用代数，我们还可以做得更好：

$$n \leqslant S(k)=\frac{1}{2}k(k+1) \Leftrightarrow k^2+k-2n \geqslant 0 \Leftrightarrow \left(k-\frac{-1+\sqrt{1+8n}}{2}\right)\cdot\left(k-\frac{-1-\sqrt{1+8n}}{2}\right) \geqslant 0$$

注意到 $k-\frac{-1-\sqrt{1+8n}}{2}$ 总是正数，因此 $n \leqslant S(k) \Leftrightarrow k \geqslant \frac{-1+\sqrt{1+8n}}{2}$，换句话说，我们可以直接求出 $k=\left\lceil \frac{-1+\sqrt{1+8n}}{2} \right\rceil$。为了避免浮点误差，下面的程序用到了一点小技巧：

```
#include<stdio.h>
#include<math.h>
int main()
{
  int n;
  while(scanf("%d", &n) == 1)
  {
    int k = (int)floor((sqrt(8.0*n+1)-1)/2-1e-9)+1;
    int s = k*(k+1)/2;
    printf("%d/%d\n", s-n+1, k-s+n);
  }
  return 0;
}
```

5.4.2 因子和阶乘

输入正整数 n（$2 \leqslant n \leqslant 100$），把阶乘 $n!=1\times2\times3\times\cdots\times n$ 分解成素因子相乘的形式，从小到大输出各个素数（2、3、5、…）的指数。例如 $825=3\times5^2\times11$ 应表示成(0, 1, 2, 0, 1)，表示分别有 0、1、2、0、1 个 2、3、5、7、11。你的程序应忽略比最大素因子更大的素数

（否则末尾会有无穷多个 0）。

样例输入：

5

53

样例输出：

5! = 3 1 1

53! = 49 23 12 8 4 4 3 2 2 1 1 1 1 1 1 1

【分析】

因为 $a^m \cdot a^n = a^{mn}$，我们只需把所有素因子对应的指数累加起来。注意到 $n \leqslant 100$，这些素因子不会超过 100，输出时忽略到最后的 0 即可。再次强调：is_prime 函数不适用于特别大的 n——如果你忘记了这是为什么，请翻回语言部分再看看。这里忽略了这个问题，因为 n 不超过 100。

```
#include<stdio.h>
#include<string.h>
// 素数判定。注意：n 不能太大
int is_prime(int n)
{
  for(int i = 2; i*i <= n; i++)
    if(n % i == 0) return 0;
  return 1;
}

// 素数表
int prime[100], count = 0;
int main()
{
  // n 和各个素数的指数
  int n, p[100];

  // 构造素数表
  for(int i = 2; i <= 100; i++)
    if(is_prime(i)) prime[count++] = i;

  while(scanf("%d", &n) == 1)
  {
    printf("%d! =", n);
    memset(p, 0, sizeof(p));
    int maxp = 0;
    for(int i = 1; i <= n; i++)
    {
      // 必须把 i 复制到变量 m 中，而不要在做除法时直接修改它
```

```
        int m = i;
        for(int j = 0; j < count; j++)
          while(m % prime[j] == 0) // 反复除以prime[j]，并累加p[j]
          {
            m /= prime[j];
            p[j]++;
            if(j > maxp) maxp = j; // 更新最大素因子下标
          }
      }
      // 只循环到最大下标
      for(int i = 0; i <= maxp; i++)
        printf(" %d", p[i]);
      printf("\n");
    }
    return 0;
}
```

5.4.3 果园里的树

果园里的树排列成矩阵。它们的 x 和 y 坐标均是 1～99 的整数。输入若干个三角形，依次统计每一个三角形内部和边界上共有多少棵树，如图 5-2 所示。

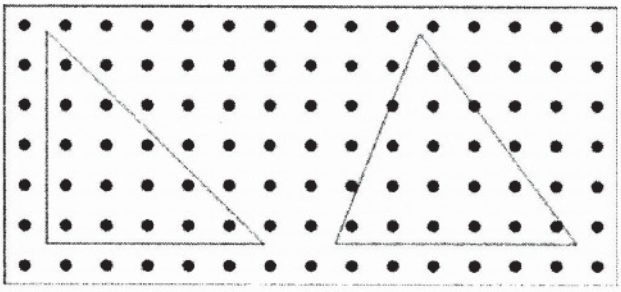


图 5-2 果园里的树

样例输入：

1.5 1.5　　1.5 6.8　　6.8 1.5

10.7 6.9　　8.5 1.5　　14.5 1.5

样例输出：

15

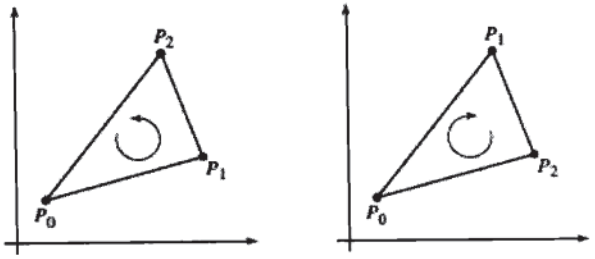
17

【分析】

最容易想到的方法是逐一判断：对于每个点(x,y)，看看它是否在三角形内。为此，我们来看这样一个函数：

```
double area2(double x0, double y0, double x1, double y1, double x2, double y2)
{
  return x0*y1+x2*y0+x1*y2-x2*y1-x0*y2-x1*y0;
}
```

它给出了三角形(x_0,y_0)-(x_1,y_1)-(x_2,y_2)的有向面积（signed area）的两倍。你暂时不必理解它为什么是对的，把它记住就可以了。什么叫有向面积呢？如图 5-3 所示。



（a）顺时针　　（b）逆时针

图 5-3　三角形的有向面积

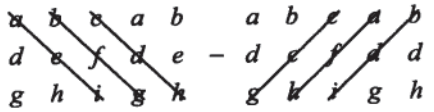
如果$\Delta P_0P_1P_2$的 3 个顶点呈逆时针排列，那么有向面积为正；如果是顺时针排列，则有向面积为负；3 点共线时，有向面积为零。读者如果觉得上面的公式比较难记，可以记它的行列式形式：

$$2A=\begin{vmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix}=x_0y_1+x_2y_0+x_1y_2-x_2y_1-x_0y_2-x_1y_0$$

如果你对行列式不熟悉也没关系，下面是3×3行列式的运算规律：

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}=aei+bfg+cdh-ceg-afh-bdi$$

如果把矩阵多复制两列，规律就很明显了：



有了它，判断就很简单了：假设输入三角形为 ABC，待判断的点为 O，则 O 在三角形 ABC 的内部或边界上当且仅当 $S_{\Delta ABC}=S_{\Delta OAB}+S_{\Delta OBC}+S_{\Delta OCA}$。代码不再赘述，但需要注意的是：在判断两个浮点数 a 和 b 是否相等时，请尽量判断 fabs(a-b)是否小于一个事先给定的 eps，如 1e-9。如有可能，请尽量避免浮点运算（例如，若本题的输入都是整数，则可以全部用整数运算；即使输入浮点数，如果规定只保留两位小数，也可以把所有坐标乘以 100，转化为整数）。

另外，有的读者可能会想到用海伦公式 $S=\sqrt{p(p-a)(p-b)(p-c)}$（其中半周长

$p=\frac{a+b+c}{2}$）来计算，但在使用时需要小心浮点误差（注意，前面的 area2 只有加减法和乘法，而海伦公式中却有除法和开平方）。事实上，笔者建议尽量避免使用海伦公式。

5.4.4 多少块土地

你有一块椭圆的地。你可以在边界上选 n 个点，并两两连接得到 $\frac{n(n-1)}{2}$ 条线段。它们最多能把土地分成多少个部分？

样例输入：4

样例输出：8

【分析】

不难发现，最优方案不会让任何 3 条线段交于一点。为了计算出答案，我们先要学习欧拉公式：$V-E+F=2$。其中，V 是顶点数（即所有线段的端点数加上交点数），E 是边数（即 n 段椭圆弧加上这些线段被切成的段数），F 是面数（即土地块数加上椭圆外那个无穷大的面）。换句话说，只需求出 V 和 E，答案就是 $E-V+1$。

不管是顶点还是边，计算时都要枚举一条从固定点出发（所以，最后要乘以 n）的所有对角线。假设该对角线的左边有 i 个点，右边有 $n-2-i$ 个点，则左右两边的点两两搭配后在这条对角线上形成了 $i(n-2-i)$个交点，得到了 $i(n-2-i)+1$ 条线段。注意，每个交点被重复计算了 4 次，而每条线段被重复计算了 2 次，因此在公式中需要有所体现。下面是完整的公式：

$$\begin{cases} V=n+\frac{n}{4}\cdot\sum_{i=0}^{n-2}i(n-2-i) \\ E=n+\frac{n}{2}\cdot\sum_{i=0}^{n-2}(i(n-2-i)+1) \end{cases}$$

如果对求和符号不熟，这里是它的定义：

$$\sum_{i=a}^{b}f(i)=f(a)+f(a+1)+\cdots+f(b)$$

有意思的是，如果把 n 从 1～5 的答案打印出来，将是：1、2、4、8、16。难道答案就是 2^{n-1}？很可惜，n=6 时，结果是 31，而非 32。这个故事告诉我们：找规律时要谨慎！

5.5 训 练 参 考

笔者曾多次强调：编程不是看会的，也不是听会的，而是练会的。从本章开始，本书的每章最后都会有一些正式的编程练习。在完成这些练习之前，读者需要知道应如何练习。

5.5.1 黑盒测试

算法竞赛一般采取黑盒测试：事先准备好一些测试用例，然后用它们测试选手程序，

根据运行结果评分。除了找不到程序（如程序名没有按照比赛规定取，或是放错位置）、编译错等连程序都没能跑起来的错误之外，一些典型的错误类型如下：

- 答案错（Wrong Answer，WA）。
- 输出格式错（Presentation Error，PE）。
- 超时（Time Limit Exceeded，TLE）。
- 运行错（Runtime Error，RE）。

在一些比较严格的比赛中，输出格式错被看成是答案错，而在另外一些比赛中，则会把二者区分开。在运行时，除了程序自身异常退出（如除 0、栈溢出、非法访问内存、断言为假、main 函数返回非 0 值）外，还可能是因为超过了评测系统的资源约束（如内存限制、最大输出限制）而被强制中止执行。有的评测系统会把这些情况和一般的运行错区分开，但在多数情况下会统一归到“运行错”中。

需要注意的是，超时不见得是因为程序效率太低，也可能是其他原因造成的。例如，比赛规定程序应从文件读入数据，但你的程序却正在等待键盘输入。其他原因包括：特殊数据导致程序进入死循环、程序实际上已经崩溃却没异常退出等。

如果上述错误都没有，那么恭喜你，你的程序通过了测试。在 ACM/ICPC 中，这意味着你的程序被裁判接受（accepted，AC），而在分测试点的比赛中，这意味着你拿到了该测试点的分数。

需要注意的是，一些比赛的测试点可以给出“部分分”——如答案正确但不够优，或者题目中有两个任务，你只成功完成了一个任务等。不管怎样，得分的前提是不超时、没有运行错。只有这样，你的程序输出才会参与评分。

5.5.2　在线评测系统

在线评测系统（Online Judge，OJ）为平时练习和网上竞赛提供了一个很好的平台。事实上，本书中的练习大都通过 OJ 给出。

首先，要向读者介绍的是历史最悠久、最著名的 OJ：西班牙 Valladolid 大学的 UVaOJ，网址为 http://uva.onlinejudge.org/[①]。除了收录了早期的 ACM/ICPC 区域比赛题目之外，这里还经常邀请世界顶尖的命题者共同组织网上竞赛，吸引了大量来自世界各地的高手同场竞技。

目前，UVaOJ 网站的题库已经包含了一个特殊的分卷（Volume）——“AOAPC I”，把本书的配套习题按照易于查找和提交的方式集中在了一起，并将逐步提供题目的中文翻译和算法提示。根据读者的反馈，网上题库可能在本书的基础上增加一些有价值的题目，并移除一些不太合适的题目，因此建议读者在做题时直接参考 UVaOJ 的 AOAPC 分卷。

经过了前 4 章的学习，相信你已经能够解决 UVaOJ 的下面这些题目了：勇士 Hashmat（10055）、重温高中物理（10071）、生态奖金（10300）、解码器（458）、幼儿园数数游戏（494）、机器加工的表面（414）、旋转句子（490）、非凡的迷宫（445）、三角波（488）、剑子手游戏（489）、Collatz 序列（694）、线性细胞自动机（457）。请至少在这 12 个题目中选择其中的 3 个进行练习，直到把它们顺利解决（得到 OJ 的“AC”）。

[①] 目前的 UVaOJ 网站与 IE 浏览器兼容性不好，推荐使用 FireFox 浏览器。

第二个要介绍的是浙江大学的 ZOJ：http://acm.zju.edu.cn/。它是国内第一个 OJ，也是目前最具影响力的 OJ 之一。和 UVa 一样，它也有很多简单题，如：A+B 问题（1001）、财务管理（1048）、反转文本（1295）。请至少选择一个进行练习。

第三个要介绍的是北京大学的 POJ：http://acm.pku.edu.cn/。它也是目前最具影响力的 OJ 之一。它和 ZOJ 都会收录很多最近的比赛题目，因此题库有一定数量的重叠（如“财务管理”也可以在 POJ 中提交）。尽管如此，ZOJ 与 POJ 和 UVa 一样，都举办过很多高质量的在线比赛，这些比赛中的很多题目都是在其他地方找不到的。

5.5.3 推荐题目

在算法学习尚未开始之前，训练的主要目的是练习编程。为了方便起见，我们只给出 UVaOJ 中的题目。

按照小节顺序，这里首先推荐 9 道字符串题目：回文词（401）、沃尔多夫在哪里（10010）、自动作诗机（10361）、人工智能（537）、“借口，借口！”（409）、磁带解码（10878）、安迪的第一个字典（10815）、立即可解码性（644）、自动编辑（10115）。

接下来是 5 道高精度运算的题目：整数查询（424）、乘积（10106）、溢出（465）、求幂（748）、如果我们重返童年（10494）。

然后是 12 道与排序、检索有关的题目：猜数字游戏的提示（340）、战利品列表（10420）、大理石在哪？（10474）、一堆树（152）、列车交换（299）、煎饼堆（120）、反片语（156）、Unix 的 ls 命令（400）、快速查找（123）、足球（10194）、487-3279（755）、疯狂的命理学家（10785）。

最后是一些数学类题目。先来 17 道杂题：密码学的力量（113）、棋盘上的蚂蚁（10161）、立方体着色（253）、秘密研究（621）、?1?2?…?n=k 问题（10025）、一盒砖（591）、帽子里的猫（107）、蜗牛（573）、步数（846）、正义的土地（10499）、多少个交点？（10790）、寻找 Nessy（11044）、多项式除法的商（10719）、2/3/4 维方形/方体（10177）、超级计算机（10916）、大块巧克力（10970）、简单计算（10014）。上面的题都不难，但有的需要思考和推导才能做出，甚至可能还需要一点灵感。建议读者阅读上面所有题目并尽量解决它们，但不必勉强。

下面是 9 道和数论有关的题目：斜二进制（575）、灯光（10110）、移位乘法（550）、阶乘（568）、均匀的生成器（408）、伪随机数（350）、多少个零和数字？（10061）、大数分解（10392）、代码重构（10879）。

最后是 5 道简单的几何计算题：另两棵树（10250）、时钟的指针（579）、内接圆和等腰三角形（375）、台球（10387）、Myacm 三角形（10112）。

第 6 章　数据结构基础

学习目标

- ☑ 熟练掌握栈和队列及其实现
- ☑ 了解双向链表及其实现
- ☑ 掌握对比测试的方法
- ☑ 掌握随机数据生成方法
- ☑ 掌握完全二叉树的数组实现
- ☑ 了解动态内存分配和释放方法及其注意事项
- ☑ 掌握二叉树的链式表示法
- ☑ 掌握二叉树的先序、后序、中序遍历和层次遍历
- ☑ 掌握图的 DFS 及连通块计数
- ☑ 掌握图的 BFS 及最短路的输出
- ☑ 掌握拓扑排序算法
- ☑ 掌握欧拉回路算法

本章介绍基础数据结构，包括线性表、二叉树和图。尽管这些内容本身并不能算“高级”，但却是很多高级内容的基础。如果数据结构基础没有打好，很难设计出正确、高效的算法。

6.1　栈 和 队 列

线性表是“所有元素排成一行”的数据结构。除了第一个元素之外，所有元素都有一个“前一个元素”；除了最后一个元素外，所有元素都有“后一个元素”。

线性结构看似简单，却是重要的算法和数据结构的基础。例如，环状结构也经常转化为线性结构——只需要从某个元素处把环切断，就变成了链。

本节介绍两种特殊的线性表：栈和队列。

6.1.1　卡片游戏

桌上有一叠牌，从第一张牌（即位于顶面的牌）开始从上往下依次编号为 1～n。当至少还剩两张牌时进行以下操作：把第一张牌扔掉，然后把新的第一张放到整叠牌的最后。输入 n，输出每次扔掉的牌，以及最后剩下的牌。

样例输入：7

样例输出：1 3 5 7 4 2 6

【分析】

这些牌像不像是在排队？每次从排头拿到两个，其中第二个再次排到尾部。我们把这种数据结构称为队列（queue）。或者说得更加学术一点：FIFO 表。其中 FIFO 表示先进先出（First In，First Out），符合我们在日常生活中的排队。

用一个数组 queue 来实现这个队列，再设两个指针 front 和 rear。从下面的代码中，能看出一些端倪吗？

```
#include<stdio.h>
const int MAXN = 50;
int queue[MAXN];
int main()
{
  int n, front, rear;
  scanf("%d", &n);
  for(int i = 0; i < n; i++) queue[i] = i+1;    // 初始化队列
  front = 0;                                    // 队首元素的位置
  rear = n;                                     // 队尾元素的后一个位置
  while(front < rear)                           // 当队列非空
  {
    printf("%d ", queue[front++]);              // 输出并抛弃队首元素
    queue[rear++] = queue[front++];             // 队首元素转移到队尾
  }
  return 0;
}
```

尽管运行结果没错，但这个程序其实是有 bug 的：如果在最后把 rear 的值打印出来，你会发现 rear 比 n 大（事实上，恰好是 2n）。换句话说，在程序运行的后期，queue[rear++]=queue[front++]读写了非法内存！前面说过，读写非法内存不一定会引起程序崩溃，但我们不能因此就满足于这个程序。要么把数组空间开大些，要么采取一种称为循环队列的技术，重用已出队元素占用的空间。

C++提供了一种更加简单的处理方式——STL 队列。下面是代码：

```
#include<cstdio>
#include<queue>
using namespace std;

queue<int> q;
int main()
{
  int n;
  scanf("%d", &n);
  for(int i = 0; i < n; i++) q.push(i+1);       // 初始化队列
```

```
  while(!q.empty())                          // 当队列非空
  {
    printf("%d ", q.front());                // 打印队首元素
    q.pop();                                 // 抛弃队首元素
    q.push(q.front());                       // 把队首元素加入队尾
    q.pop();                                 // 抛弃队首元素
  }
  return 0;
}
```

程序并没有简洁很多，但可读性大大增强了。这里所说的可读性不仅体现在“queue”、“front”这些可望文知义的命名，还因为 STL 这个库的标准性——它是 C++不可分割的组成部分，每一个熟悉 C++的程序员都会对其有一定了解。如果你的代码需要更多人来阅读，应多使用这样标准化的东西。

除此之外，上面的代码还有两个附加的好处。首先，你不需要事先知道 n 的大小；其次，可以少用两个变量 front 和 rear。不要小看这些好处。减少魔术数（magic number）和变量个数都是提高代码可读性、减小错误可能性的重要手段。当编写过一定量的程序，并且犯过相当数量的错误之后，相信你会对此有所体会。

6.1.2　铁轨

某城市有一个火车站，铁轨铺设如图 6-1 所示。有 n 节车厢从 A 方向驶入车站，按进站顺序编号为 1～n。你的任务是让它们按照某种特定的顺序进入 B 方向的铁轨并驶出车站。为了重组车厢，你可以借助中转站 C。这是一个可以停放任意多节车厢的车站，但由于末端封顶，驶入 C 的车厢必须按照相反的顺序驶出 C。对于每个车厢，一旦从 A 移入 C，就不能再回到 A 了；一旦从 C 移入 B，就不能回到 C 了。换句话说，在任意时刻，只有两种选择：A→C 和 C→B。

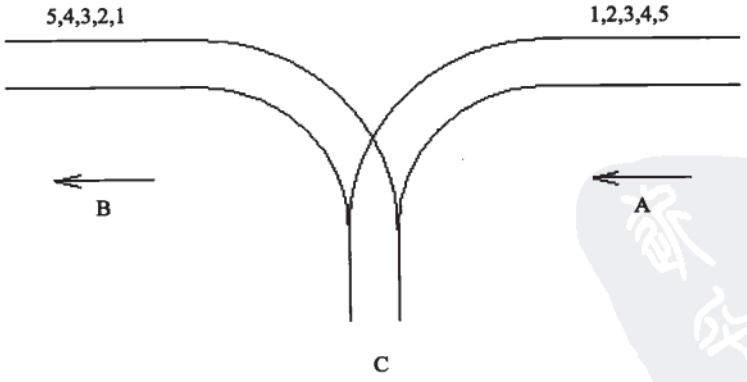


图 6-1　铁轨

样例输入：

5
1 2 3 4 5
5
5 4 1 2 3
6
6 5 4 3 2 1

样例输出：

Yes
No
Yes

【分析】

在中转站 C 中，车厢符合后进先出的原则，称为 LIFO 表。其中 LIFO 代表 Last In First Out。它还有一个更常用的名称——栈。本书在介绍递归时已经对它进行了简单的介绍。由于只有一端生长，实现栈时只需要一个数组 stack 和栈顶指针（始终指向栈顶元素）。代码如下：

```
#include<stdio.h>
const int MAXN = 1000 + 10;
int n, target[MAXN];

int main()
{
  while(scanf("%d", &n) == 1)
  {
    int stack[MAXN], top = 0;
    int A = 1, B = 1;
    for(int i = 1; i <= n; i++)
      scanf("%d", &target[i]);
    int ok = 1;
    while(B <= n)
    {
      if(A == target[B]){ A++; B++; }
      else if(top && stack[top] == target[B]){ top--; B++; }
      else if(A <= n) stack[++top] = A++;
      else { ok = 0; break; }
    }
    printf("%s\n", ok ? "Yes" : "No");
  }
  return 0;
}
```

为了方便起见，这里使用的数组下标均从 1 开始。例如，target[1]是指目标序列中第一个车厢的编号，而 stack[1]是栈底元素（这样，栈空当且仅当 top=0）。

和刚才一样，下面的程序也可以用 STL 栈来实现：

```
#include<cstdio>
#include<stack>
using namespace std;
const int MAXN = 1000 + 10;

int n, target[MAXN];

int main()
{
  while(scanf("%d", &n) == 1)
  {
    stack<int> s;
    int A = 1, B = 1;
    for(int i = 1; i <= n; i++)
      scanf("%d", &target[i]);
    int ok = 1;
    while(B <= n)
    {
      if(A == target[B]){ A++; B++; }
      else if(!s.empty() && s.top() == target[B]){ s.pop(); B++; }
      else if(A <= n) s.push(A++);
      else { ok = 0; break; }
    }
    printf("%s\n", ok ? "Yes" : "No");
  }
  return 0;
}
```

6.2 链　　表

在多数情况下，线性表都可以用数组轻松实现，但事情并不总是这么简单。本节介绍链表，并用它和数组进行比较。

6.2.1 初步分析

例题 6-1　移动小球

你有一些小球，从左到右依次编号为 1, 2, 3,…, n，如图 6-2 所示。

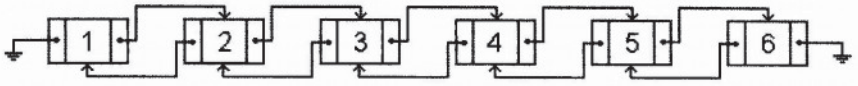


图 6-2　链表的初始状态

你可以执行两种指令。其中，A X Y 表示把小球 X 移动到小球 Y 左边，B X Y 表示把小球 X 移动到小球 Y 右边。指令保证合法，即 X 不等于 Y。

例如，在初始状态下执行 A 1 4 后，小球 1 被移到小球 4 的左边，如图 6-3 所示。

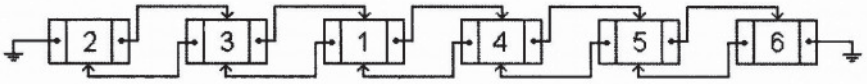


图 6-3　一次操作后的链表状态

如果再执行 B 3 5，结点 3 将会移到 5 的右边，如图 6-4 所示。

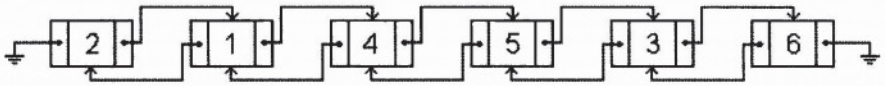


图 6-4　两次操作后的链表状态

输入小球个数 n，指令条数 m 和 m 条指令，从左到右输出最后的序列。注意，n 可能高达 500000，而 m 可能高达 100000。

样例输入：

```
6  2
A  1  4
B  3  5
```

样例输出：

```
2 1 4 5 3 6
```

【分析】

各个小球在逻辑上是相邻的，因此可以考虑把它们放在一个数组里，像这样：

```
for(int i = 1; i <= n; i++)                    // 初始化数组
  A[i] = i;
for(int i = 0; i < m; i++)
{
  scanf("%s%d%d", type, &X, &Y);
  p = find(X);                                 // 查找 X 和 Y 在数组中的位置
  q = find(Y);
  if(type[0] == 'A')
  {
    if(q > p) shift_circular_left(p, q-1);     // A[p]到A[q-1]往左循环移动
    else shift_circular_right(q, p);           // A[q]到A[p]往右循环移动
  } else
```

```
    {
      if(q > p) shift_circular_left(p, q);        // A[p]到A[q]往左循环移动
      else shift_circular_right(q+1, p);          // A[q+1]到A[p]往右循环移动
    }
  }
```

请读者花一点时间来验证这四种情况是否都对应于上面所说的循环移动。至于函数 find、shift_circular_left 和 shift_circular_right，相信读者可以独立完成，这里不再赘述。上面代码的 scanf 参数中，“指令类型”使用的是字符串占位符%s，而非字符占位符%c。这是有原因的：当用 scanf(“%d”)读取整数时，并没有读到它后面的回车换行符。这时，用%c 会读到这个回车换行符，而只有%s 才会跳过它，读取下一个非空白符组成的字符串。

上面的算法是正确的，但时间效率却不高——而题目已经强调：n 可能高达 500000，而 m 可能高达 100000。上面的代码到底会不会超时呢？一般来说，可以用两种方法判断：测试和分析。

计时测试的方法前面已经学过，它的优点是原理简单、可操作性强，缺点在于必须事先把程序写好——包括主程序和测试数据生成器。如果算法非常复杂，这是相当花时间的。

另一个方法是在写程序之前进行算法分析，估算时间效率。真的可以在写程序之前估算效率吗？这个问题我们留到第 8 章中详细讨论，不过这里先给出一些直观分析：如果反复执行 B 1 n 和 A 1 2，每次都将移动几乎所有元素。元素个数和指令条数都那么大，移动总次数将是相当可观的。

6.2.2　链式结构

第二种方法是强调小球之间的相对顺序，而非绝对顺序。我们用 left[i]和 right[i]分别表示编号为 i 的小球左边和右边的小球编号（如果是 0，表示不存在），则移动过程可以分成两个步骤：把 X 移出序列；把 X 重新插入序列。

第一步不难处理，让 left[X]和 right[X]相互连接即可，如图 6-5 所示。注意，其他所有的 left 和 right 都不会变化。

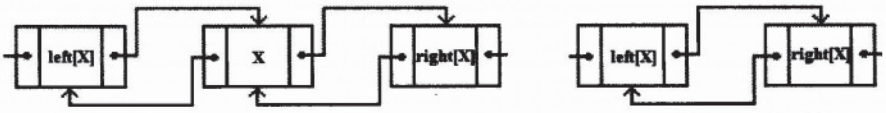


图 6-5　在链表中删除结点

第二步类似。对于 A 指令，需要修改 left[Y]的 right 值和 Y 的 left 值，如图 6-6 所示。

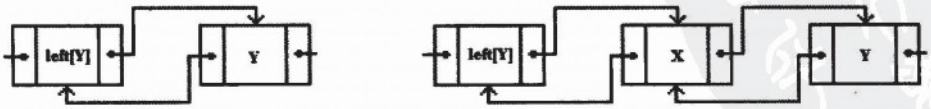


图 6-6　在链表中插入结点（情况 A）

而对于 B 指令，需要修改 Y 的 right 值和 right[Y]的 left 值，如图 6-7 所示。

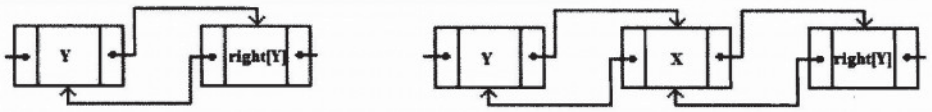


图 6-7　在链表中插入结点（情况 B）

不管在哪种情况下，最后都需修改 X 自己的 left 和 right。有一种特殊情况需要考虑：如果 X 是最左边的小球，那么 left[X]的值是 0，因为它没有“左边的小球”。可是在第一步里，需要修改 right[left[X]]的值，即 right[0]的值。这会引发问题吗？答案是：不会。其实我们想象最左边的小球左边还有一个编号为 0 的虚拟的小球，一切都解释得通。顺便说一句，最右边小球的右边的虚拟小球编号可以是 n+1，也可以是 0（如果是这样，实际上小球形成的是环而不是链，不过这又有什么关系呢？）。核心代码如下：

```
scanf("%s%d%d", type, &X, &Y);
link(left[X], right[X]);
if(type[0] == 'A')
{
    link(left[Y], X); // 这一行和下一行不能搞反
    link(X, Y);
} else
{
    link(X, right[Y]); // 这一行和下一行也不能搞反
    link(Y, X);
}
```

函数 link(X, Y)的作用是赋值 right[X] = Y，然后 left[Y] = X。注意到上述代码中的注释了吗？请读者思考：

（1）为什么那两行代码不能写反呢？（提示：left 和 right 数组是在动态改变的）。

（2）如何减小函数调用之间的顺序相关性，减小代码编写的出错可能？（提示：先用其他变量把重要的数据存起来，就不怕它变了）。

6.2.3　对比测试

如果读者曾独立编写过上面的程序，可能会花费较长的时间进行调试。又或者，自以为正确的程序却被判为错误。在链式结构中，这样的情况是时常发生的，我们需要具备一定的调试和测试能力。

简单地说，测试的任务就是检查一份代码是否正确。如果找到了错误，最好还能提供一个让它错误的数据。有了错误数据之后，接下来的任务便是调试：看看程序为什么是错的。如果找到了错，最好把它改对——至少对于刚才的错误数据能得到正确的结果。

改对一组数据之后，可能还有其他错误，因此需要进一步测试；即使以前曾经正确的数据，也可能因为多次改动之后反而变错了，需要再次调试。总之，在编码结束后，为了确保程序的正确性，测试和调试往往要交替进行。

如何确保上述代码的正确性呢？一个行之有效的方法是：再找一份完成同样功能的代

码与之对比。从哪里找呢？别担心，我们刚刚不就写过一个基于数组的版本吗？尽管那个程序运行很慢，但思路简单，不易出错。用它来和这个新程序“对答案”（俗称对拍）是再好不过的了。

如何进行对比测试呢？首先需要数据，而且是大量数据。为此，需要编写数据生成器，像这样：

```
#include<stdio.h>
#include<stdlib.h>  // rand()和srand()需要
#include<time.h>    // time()需要
int n = 100, m = 100000;

double random()     // 生成[0,1]之间的均匀随机数
{
  return (double)rand() / RAND_MAX;
}

int random(int m)   // 生成[0,m-1]之间的均匀随机数
{
  return (int)(random() * (m-1) + 0.5);
}

int main()
{
  srand(time(NULL)); // 初始化随机数种子
  printf("%d %d\n", n, m);
  for(int i = 0; i < m; i++)
  {
    if(rand() % 2 == 0) printf("A"); else printf("B"); // 随机指令种类
    int X, Y;
    for(;;)
    {
      X = random(n)+1;
      Y = random(n)+1;
      if(X != Y) break; // 只有X和Y不相等才是合法的
    }
    printf(" %d %d\n", X, Y);
  }
  return 0;
}
```

核心函数是 stdlib.h 中的 rand()，它生成一个闭区间[0,RAND_MAX]的均匀随机整数（均匀的含义是：该区间内每个整数被产生的概率相同），其中 RAND_MAX 至少为 32767（$2^{15}-1$），在不同环境下的值可能不同。严格地说，这里的随机数是“伪随机数”，因为

它也是由数学公式计算出来的，不过在算法领域，多数情况下可以把它当作真正的随机数。

6.2.4 随机数发生器

很多人喜欢用 rand()%n 产生区间[0,n)内的一个随机整数。姑且不论这样产生的整数是否仍然均匀分布，当 n 大于 RAND_MAX 时，此法并不能得到期望的结果。由于 RAND_MAX 很有可能只有 32767 这么小，在使用此法时应当小心。

上述代码采取的方法是先除以 RAND_MAX，得到[0,1]之间的随机实数，扩大 n-1 倍之后四舍五入，再加 1 得到[1,n]之间的均匀整数。这样做在 n 很大时“精度”不好（好比把小图放大后会看到“锯齿”），但这里的 n 很小，这样做已经可以满足要求了[①]。

提示 6-1：stdlib.h 中的 rand()生成闭区间[0,RAND_MAX]内均匀分布的随机整数，其中 RAND_MAX 至少为 32767。如果要生成更大的随机整数，在精度要求不太高的情况下可以用 rand()的结果“放大”得到。

程序最开始执行了一次 srand(time(NULL))，其中 srand 函数用来初始化“随机数种子”。简单地说，种子是伪随机数计算的依据。种子相同，计算出来的“随机数”序列总是相同。如果不调用 srand 而直接使用 rand()，相当于调用过一次 srand(1)，因此程序每次执行时，将得到同一套随机数。另外，不要在同一个程序每次生成随机数之前都重新调用一次 srand。有的初学者抱怨“rand()产生的随机数根本不随机，每次都相同”，就是因为误解了 srand 的作用——再次强调，请只在程序开头调用一次 srand，而不要在同一个程序中多次调用。

提示 6-2：可以用 stdlib.h 中的 srand 函数初始化随机数种子。如果需要程序每次执行时使用一个不同的种子，可以用 time.h 中的 time(NULL)为参数调用 srand。一般来说，只在程序执行的开头调用一次 srand。

“同一套随机数”可能是好事也可能是坏事。例如，若要反复测试程序对不同随机数据的响应，需要每次得到的随机数不同。一个简单的方法是用当前时间 time(NULL)（在 time.h 中）作为参数调用 srand。由于时间是不断变化的，每次运行时，一般会得到一套不同的随机数。之所以说“一般”，是因为事情并非总是如此。time 函数返回的是自 1970 年 1 月 1 日 0 点以来经过的“秒数”，因此每秒才变化一次。如果你的程序是由操作系统自动批量执行的，可能因为每次运行时间过短，导致在相邻若干次执行时 time 的返回值全部相同。一个解决办法是在测试程序的主函数中设置一个循环，做足够多次测试后再退出。

另一方面，如果发现某程序对于一组随机数据出错，就需要在调试时“重现”这组数据。这时，“每次相同的随机序列”就显得十分重要了。顺便说一句，不同的编译器计算随机数的方法可能不同。如果是不同编译器编译出来的程序，即使是用相同的参数调用 srand()，也可能得到不同的随机序列。

最后，可以反复执行下面的操作：生成随机数据，分别执行两个程序，比较它们的结果。合理地使用操作系统提供的脚本功能，可以自动完成对比测试，具体方法请读者参见

[①] 如果坚持需要更高的精度，可以采取多次随机的方法。

附录。

最后还要啰嗦一句的是：如果发现让两个程序答案不一致的数据，最好别急着对它进行调试。可以尝试着减小数据生成器中的 n 和 m，试图找到一组尽量简单的错误数据。一般来说，数据越简单，越容易调试。如果发现只有很大的数据才会出错，通常意味着程序在处理极限数据方面有问题——如 is_prime 中遇到了“过大的 n”，或者数组开得不够大等。这些都是很实用的技巧，建议读者多多积累。

6.3　二　叉　树

二叉树（Binary Tree）的递归定义如下：二叉树要么为空，要么由根结点（root）、左子树（left subtree）和右子树（right subtree）组成，而左子树和右子树分别是一棵二叉树。注意，在计算机中，树一般是“倒置”的——根在上，叶子在下。

6.3.1　小球下落

有一棵二叉树，最大深度为 D，且所有叶子的深度都相同。所有结点从上到下从左到右编号为 $1, 2, 3, \cdots, 2^D-1$。在结点 1 处放一个小球，它会往下落。每个内结点上都有一个开关，初始全部关闭，当每次有小球落到一个开关上时，它的状态都会改变。当小球到达一个内结点时，如果该结点上的开关关闭，则往左走，否则往右走，直到走到叶子结点，如图 6-8 所示。

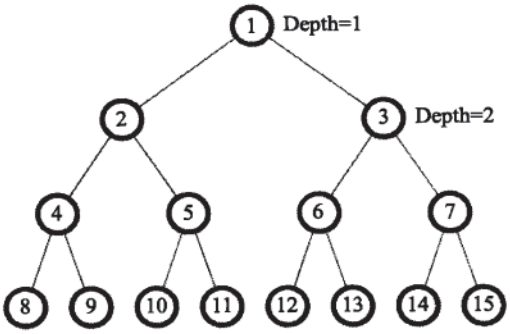


图 6-8　所有叶子深度相同的二叉树

一些小球从结点 1 处依次开始下落，最后一个小球将会落到哪里呢？输入叶子深度 D 和小球个数 I，输出第 I 个小球最后所在的叶子编号。假设 I 不超过整棵树的叶子个数。$D \leqslant 20$。输入最多包含 1000 组数据。

样例输入：

4 2

3 4

10 1
2 2
8 128
16 12345
样例输出：
12
7
512
3
255
36358

【分析】

不难发现：对于一个结点 k，它的左儿子、右儿子的编号分别是 $2k$ 和 $2k+1$。这样，不难写出如下的模拟程序：

```
#include<stdio.h>
#include<string.h>
const int MAXD = 20;
int s[1<<MAXD];              // 最大结点个数为 2^MAXD-1
int main()
{
  int D, I;
  while(scanf("%d%d", &D, &I) == 2)
  {
    memset(s, 0, sizeof(s));                  // 开关
    int k, n = (1<<D)-1;                      // n 是最大结点编号
    for(int i = 0; i < I; i++)
    {                                         // 连续让 I 个小球下落
      k = 1;
      for(;;)
      {
        s[k] = !s[k];
        k = s[k] ? k*2 : k*2+1;               // 根据开关状态选择下落方向
        if(k > n) break;                      // 已经落“出界”了
      }
    }
    printf("%d\n", k/2);                      // “出界”之前的叶子编号
  }
  return 0;
}
```

尽管在本题中，每次小球都是严格下落 $D-1$ 次，但用“if(k > n) break”的方法判断“出

界”更具一般性，所以上面的代码采用了这种方法。

这个程序和前面用数组模拟小球移动的程序有一个共同的缺点：运算量太大。由于 I 可以高达 2^{D-1}，每个测试数据下落总层数可能会高达 $2^{19}*19=9961472$，而一共可能有 10000 组数据……

每个小球都会落在根结点上，因此前两个小球必然是一个在左子树，一个在右子树。一般地，只需看小球编号的奇偶性，就能知道它是最终在哪棵子树中。对于那些落入根结点左子树的小球来说，只需知道该小球是第几个落在根的左子树里的，就可以知道它下一步往左还是往右了。依此类推，直到小球落到叶子上。

如果使用题目中给出的编号 I，则当 I 是奇数时，它是往左走的第$(I+1)/2$ 个小球；当 I 是偶数时，它是往右走的第 $I/2$ 个小球。这样，可以直接模拟最后一个小球的路线：

```
while(scanf("%d%d", &D, &I) == 2)
{
  int k = 1;
  for(int i = 0; i < D-1; i++)
    if(I%2) { k = k*2; I = (I+1)/2; }
    else { k = k*2+1; I /= 2; }
  printf("%d\n", k);
}
```

这样，程序的运算量就与小球编号无关了，而且节省了一个巨大的 s 数组。另外，使用一些程序上的小技巧，可以避开对 I 的奇偶性的讨论，读者不妨一试。

6.3.2　层次遍历

输入一棵二叉树，你的任务是按从上到下、从左到右的顺序输出各个结点的值。每个结点都按照从根结点到它的移动序列给出（L 表示左，R 表示右）。在输入中，每个结点的左括号和右括号之间没有空格，相邻结点之间用一个空格隔开。每棵树的输入用一对空括号()结束（这对括号本身不代表一个结点），如图 6-9 所示。

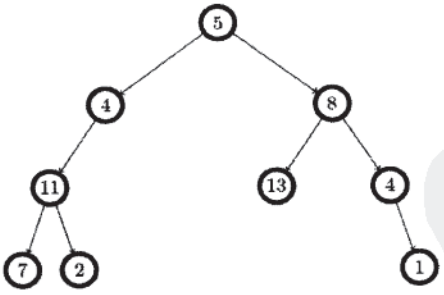


图 6-9　一棵二叉树

注意，如果从根到某个叶结点的路径上有的结点没有在输入中给出，或者给出了超过一次，应当输出-1。结点个数不超过 256。

样例输入：

(11,LL) (7,LLL) (8,R) (5,) (4,L) (13,RL) (2,LLR) (1,RRR) (4,RR) ()
(3,L) (4,R) ()

样例输出：

5 4 8 11 13 4 7 2 1
-1

【分析】

受 6.3.1 节的启发，我们是否可以把树上的结点编号，然后把二叉树储存在数组中呢？很遗憾，这样的方法在本题中是行不通的——题目中说了，结点最多有 256 个。如果各个结点形成一条链，最后一个结点的编号将是巨大的！就算你用高精度保存编号，数组也开不下。

看来，需要采用动态结构，根据需要建立新的结点，然后把它们组织成一棵树。首先，编写输入部分和主程序：

```
char s[MAXN + 10];                          // 保存读入结点
int read_input()
{
  failed = 0;
  root = newnode();                         // 创建根结点
  for(;;)
  {
    if(scanf("%s", s) != 1) return 0;       // 整个输入结束
    if(!strcmp(s, "()")) break;             // 读到结束标志，退出循环
    int v;
    sscanf(&s[1], "%d", &v);                // 读入结点值
    addnode(v, strchr(s, ',')+1);           // 查找逗号，然后插入结点
  }
  return 1;
}
```

程序不难理解：不停读入结点，如果在读到空括号之前文件结束，则返回 0（这样，在 main 函数里就能得知输入结束）。注意，这里两次用到了 C 语言中字符串的灵活性——可以把任意“指向字符的指针”看成是字符串，从该位置开始，直到字符‘\0’。例如，若读到的结点是“(11,LL)”，则&s[1]所对应的字符串是“11,LL)”。函数 strchr(s, ‘,’)返回字符串 s 中从左往右第一个字符‘,’的指针，因此 strchr(s, ‘,’)+1 所对应的字符串是“LL)”。这样，我们实际调用的是 addnode(11, “LL)”)。

接下来是重头戏了：二叉树的结点定义和操作。首先，需要定义一个称为 Node 的数据类型，并且对应整棵二叉树的树根 root：

```
// 结点类型
typedef struct Tnode
{
```

```
  int have_value;         // 是否被赋值过
  int v;                  // 结点值
  struct TNode *left, *right;
} Node;

Node* root;               // 二叉树的根结点
```

由于二叉树是递归定义的，它的左右儿子类型都是“指向结点类型的指针”。换句话说，如果结点的结构体名称为 TNode，则左右儿子的类型都是 struct TNode *。为什么不能写成 Node *呢？因为此时 Node 类型还没有定义完，不可用。

每次需要一个新的 Node 时，都要调用 stdlib.h 中的 malloc 函数申请内存，返回一个未初始化的空间。下面把申请新结点的操作封装到 newnode 函数中：

```
Node* newnode()
{
  Node* u = (Node*) malloc(sizeof(Node));   // 申请动态内存
  if(u != NULL)                             // 若申请成功
  {
    u->have_value = 0;  // 显式的初始化为 0，因为 malloc 申请内存时并不把它清零
    u->left = u->right = NULL;              // 初始时没有左右儿子
  }
  return u;
}
```

提示 6-3：可以用 stdlib.h 中的 malloc 函数申请空间。向该函数传入所需空间的大小，函数将返回一个指针。如果返回值为 NULL，说明空间不足，申请失败。

接下来是在 read_input 中调用的 addnode 函数。它按照移动序列行走，目标不存在时调用 newnode 来创建新结点。

```
void addnode(int v, char* s)
{
  int n = strlen(s);
  Node* u = root;                                   // 从根结点开始往下走
  for(int i = 0; i < n; i++)
    if(s[i] == 'L')
    {
      if(u->left == NULL) u->left = newnode();  // 结点不存在，建立新结点
      u = u->left;                              // 往左走
    } else if(s[i] == 'R')
    {
      if(u->right == NULL) u->right = newnode();
      u = u->right;
    }                                       // 忽略其他情况，即最后那个多余的右括号
  if(u->have_value) failed = 1;             // 已经赋过值，表明输入有误
```

```
  u->v = v;
  u->have_value = 1;                                    // 别忘了做标记
}
```

这样一来，输入和建树部分已经结束，接下来需要按照层次顺序遍历这棵树。我们使用一个队列来完成这个任务，初始时只有一个根结点，然后每次取出一个结点，就把它的左右儿子（如果有）放进队列。

```
int n = 0, ans[MAXN];                                   // 结点总数和输出序列
int bfs()
{
  int front = 0, rear = 1;
  Node* q[MAXN];
  q[0] = root;                                          // 初始时只有一个根结点
  while(front < rear)
  {
    Node* u = q[front++];
    if(!u->have_value) return 0;                        // 有结点没有被赋值过，表明输入
                                                           有误
    ans[n++] = u->v;                                    // 增加到输出序列尾部
    if(u->left != NULL) q[rear++] = u->left;            // 把左儿子（如果有）放进队列
    if(u->right != NULL) q[rear++] = u->right;          // 把右儿子（如果有）放进队列
  }
  return 1;                                             // 输入正确
}
```

这样遍历二叉树的方法称为宽度优先遍历（Breadth-First Search，BFS）。后面我们将看到，BFS 在显示图和隐式图算法中扮演着重要的角色。

上面的程序在功能上是正确的，但有一个小小的技术问题：在输入一组新数据时，我们没有释放上一棵二叉树所申请的内存空间。一旦执行了 root = newnode()，就再也没有人可以访问到那些内存了，尽管那些内存在物理上仍然存在！

当然，从技术上说，还是可以访问到那些内存的——如果你能“猜到”那些地址的话。之所以说“访问不到”，是因为我们丢失了指向这些内存的指针。如果你觉得这难以理解，想象一下把电话号码簿搞丢了以后的情形：理论上你仍然可以像以前一样给朋友们打电话，只是没有了电话簿，你猜不到他们的号码了。

有一个专业术语用来描述这样的情况：内存泄漏（memory leak）——它意味着有些内存被白白浪费了。在实际运行的过程中，一般很难看出这个问题：在很多情况下，内存空间都不会很紧张，浪费一些空间后，程序还是可以正常运行；况且在整个程序结束后，该程序占用的空间会被操作系统全部回收，包括泄漏的那些。

下面是释放一棵二叉树的代码，请在 root = newnode()之前加一行 remove_tree(root)：

```
void remove_tree(Node* u) {
  if(u == NULL) return;                                 // 提前判断比较稳妥
```

```
    remove_tree(u->left);          // 递归释放左子树的空间
    remove_tree(u->right);         // 递归释放右子树的空间
    free(u);                       // 释放 u 结点本身的内存
}
```

前面说过，本书尽量不使用指针。那为什么还在这里用了这么多指针呢？答案是：和不使用指针的版本进行对比，供读者参考。接下来，我们把指针完全去掉。

首先，还是给每个结点编号，但不是按照从上到下从左到右的顺序，而是按照结点的生成顺序。用计数器 cnt 表示已存在的结点编号的最大值，因此 newnode 函数需要改成这样：

```
const int root = 1;
void newtree() { left[root] = right[root] = 0; cnt = root; }
int newnode() { int u = ++cnt; left[u] = right[u] = 0; return u; }
```

上面的 newtree()是用来代替前面的 remove_tree(root)和 root = newnode()两条语句的：由于没有了动态内存的申请和释放，只需要重置结点计数器和根结点的左右子树了。

接下来，把所有的 Node*类型改成 int 类型，然后把结点结构中的成员变量改成全局数组（例如，u->left 和 u->right 分别改成 left[u]和 right[u]），除了 char*外，整个程序就没有任何指针了！

尽管笔者在编程时尽量避免指针和动态内存，但仍然需要具体问题具体分析。例如，用指针直接访问比“数组+下标”的方式略快，因此有的选手喜欢用“结构体+指针”的方式处理动态数据结构，但在申请结点时仍然用这里的“动态化静态”的思想，把 newnode 函数写成：

```
Node* newnode(){ Node* u = &node[++cnt]; u->left = u->right = NULL; return u;}
```

其中，node 是静态申请的结构体数组。

6.3.3　二叉树重建

对于二叉树 *T*，可以递归定义它的先序遍历、中序遍历和后序遍历如下：

```
PreOrder(T)=T 的根结点+PreOrder(T 的左子树)+PreOrder(T 的右子树)
InOrder(T)=InOrder(T 的左子树)+T 的根结点+InOrder(T 的右子树)
PostOrder(T)=PostOrder(T 的左子树)+PostOrder(T 的右子树)+T 的根结点
```

其中，加号表示字符串连接运算。例如，对于如图 6-10 所示的二叉树，先序遍历为 DBACEGF，中序遍历为 ABCDEFG。

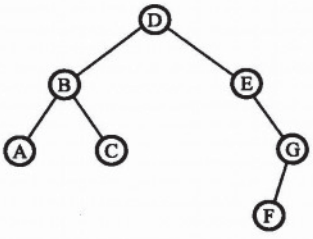


图 6-10　另一棵二叉树

输入一棵二叉树的先序遍历和中序遍历序列，输出它的后序遍历序列。

样例输入：

DBACEGF ABCDEFG

BCAD CBAD

样例输出：

ACBFGED

CDAB

【分析】

先序遍历的第一个字符就是根，因此只需在中序遍历中找到它，就知道左右子树的先序和后序遍历了。这样，可以编写一个递归程序：

```
void build(int n, char* s1, char* s2, char* s)
{
  if(n <= 0) return;
  int p = strchr(s2, s1[0]) - s2;        // 找到根结点在中序遍历中的位置
  build(p, s1+1, s2, s);                 // 递归构造左子树的后序遍历
  build(n-p-1, s1+p+1, s2+p+1, s+p);     // 递归构造右子树的后序遍历
  s[n-1] = s1[0];                        // 把根结点添加到最后
}
```

它的作用是根据一个长度为 n 的先序序列 s1 和中序序列 s2，构造一个长度为 n 的后序序列。注意，这里再次用到了 C 语言中字符串的储存方式，并灵活运用字符指针的加减法简化程序。主程序不难编写：

```
while(scanf("%s%s", s1, s2) == 2)
{
  int n = strlen(s1);
  build(n, s1, s2, ans);
  ans[n] = '\0';                         // 别忘了加上字符串结束标志
  printf("%s\n", ans);
}
```

由于本题是直接输出后序遍历，程序还可以写得更简单：省略 build 函数的最后一个参数 s，然后把 s[n-1]=s1[0]改成 printf("%c", s1[0])。当然，你还可以先递归构造出二叉树，然后再进行后序遍历。但这样的程序要复杂一些，留给读者练习。

6.4 图

图（Graph）描述的是一些个体之间的关系。与线性表和二叉树不同的是：这些个体之间既不是前驱后继的顺序关系，也不是祖先后代的层次关系，而是错综复杂的网状关系。

6.4.1　黑白图像

输入一个 *n*n* 的黑白图像（1 表示黑色，0 表示白色），任务是统计其中八连块的个数。如果两个黑格子有公共边或者公共顶点，就说它们属于同一个八连块。如图 6-11 所示的图形有 3 个八连块。

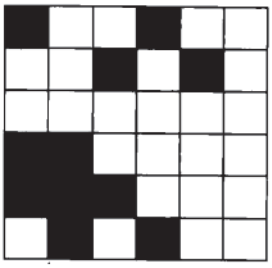


图 6-11　拥有 3 个八连块的黑白图形

【分析】

用递归求解：从每个黑格子出发，递归访问它所有的相邻黑格。

```
int mat[MAXN][MAXN], vis[MAXN][MAXN];
void dfs(int x, int y)
{
  if(!mat[x][y] || vis[x][y]) return;        // 曾经访问过这个格子，或者当前格子
                                                是白色
  vis[x][y] = 1; // 标记(x,y)已访问过
  dfs(x-1,y-1); dfs(x-1,y); dfs(x-1,y+1);
  dfs(x-1,y);           dfs(x,y+1);
  dfs(x+1,y-1); dfs(x+1,y); dfs(x+1,y+1);   // 递归访问周围的 8 个格子
}
```

这里，黑格(x,y)的mat[x][y]为1，白格为0。为了避免同一个格子访问多次，用标志vis[x][y]记录格子(x,y)是否被访问过。读者可能会奇怪：这个程序是在哪里判断“出界”的？答案是：在输入之前，在迷宫的外面加上一圈虚拟的白格子，见下面的程序。

```
    memset(mat, 0, sizeof(mat)); // 所有格子都初始化为白色，包括周围一圈的虚拟格子
    memset(vis, 0, sizeof(vis));// 所有格子都没有访问过
    for(int i = 0; i < n; i++)
    {
      scanf("%s", s);
      for(int j = 0; j < n; j++)
        mat[i+1][j+1] = s[j]-'0';// 把图像往中间移动一点，空出一圈白格子
}
```

接下来，只需不断找黑格，然后调用 dfs，从它出发寻找八连块：

```
int count = 0;
for(int i = 1; i <= n; i++)
  for(int j = 1; j <= n; j++)
    if(!vis[i][j] && mat[i][j]) { count++; dfs(i,j); }
                                                    // 找到没有访问过的黑格
printf("%d\n", count);
```

为什么函数名为 dfs 呢？可能读者已经猜到了：它是宽度优先遍历（BFS）的孪生兄弟——深度优先遍历（Depth-First Search，DFS）。DFS 和 BFS 一样，都是从一个结点出发，按照某种特定的次序访问图中的其他结点。不同的是，BFS 使用队列来存放待扩展结点，而 DFS 使用的是栈。等一等！好像上面的程序没有栈啊？尽管没有用 STL 栈，也没有 stack 数组和 top 指针，但递归调用时会把局部变量（当前结点编号）存入栈帧中。这也提醒我们：如果图像太大，递归方式的 DFS 有栈溢出的危险！为了保险起见，可以用显式栈来代替递归调用，程序留给读者编写（提示：可以把 BFS 程序中的队列直接改成栈）。

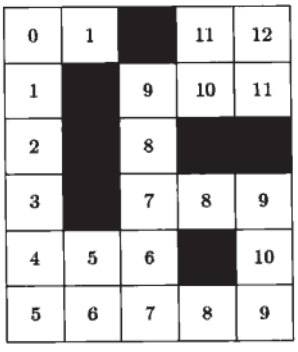
6.4.2 走迷宫

一个网格迷宫由 n 行 m 列的单元格组成，每个单元格要么是空地（用 1 来表示），要么是障碍物（用 0 来表示）。你的任务是找一条从起点到终点的最短移动序列，其中 UDLR 分别表示往上、下、左、右移动到相邻单元格。任何时候都不能在障碍格中，也不能走到迷宫之外。起点和终点保证是空地。$n,m \leqslant 100$。

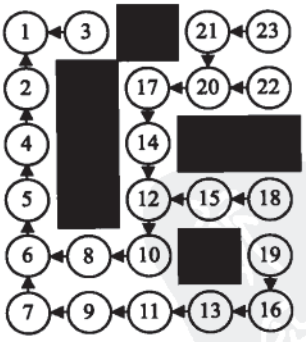
【分析】

还记得二叉树的 BFS 吗？结点的访问顺序恰好是它们到根结点距离从小到大的顺序。类似地，也可以用 BFS 来按照到起点的距离顺序遍历迷宫图。

举个例子：假定起点在左上角，我们就从左上角开始用 BFS 遍历迷宫图，逐步计算出它到每个结点的最短路距离（如图 6-12（a）所示），以及这些最短路径上每个结点的“前一个结点”（如图 6-12（b）所示）。



（a）从左上角出发到各个格子的最短距离



（b）扩展顺序和父亲指针

图 6-12　用 BFS 求迷宫中最短路

注意，如果把图 6-12（b）中的箭头理解成“指向父亲的指针”，那么迷宫中的格子就变成了一棵树——除了起点之外，每个结点恰好有一个父亲。如果看不出来，可以把这棵树画成如图 6-13 所示的样子。

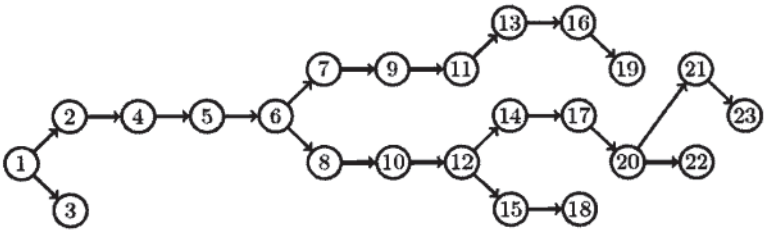


图 6-13　BFS 树的层次画法

图的 BFS 几乎与二叉树的 BFS 一样，但需要避免重复访问一个结点。下面的代码用标记 vis[x][y]记录格子(x,y)是否被走过，和 DFS 一样。

```
int q[MAXN*MAXN];
void bfs(int x, int y)
{
int front=0, rear=0, d, u;
  u = x*m+y;
  vis[x][y] = 1; fa[x][y] = u; dist[x][y] = 0;
  q[rear++] = u;
  while(front<rear)
  {
    u = q[front++];
    x = u/m; y = u%m;
    for(d = 0; d < 4; d++)
    {
      int nx = x+dx[d], ny = y+dy[d];
      if(nx>=0 && nx<n && ny>=0 && ny<m && maze[nx][ny] && !vis[nx][ny])
      {
        int v = nx*m+ny;
        q[rear++] = v;
        vis[nx][ny] = 1;
        fa[nx][ny] = u;
        dist[nx][ny] = dist[x][y]+1;
        last_dir[nx][ny] = d;
      }
    }
  }
}
```

为了方便起见，我们把格子从上到下编号为 0, 1, 2,…, n*m，因此第 i 行第 j 个格子的编

号为 i*m+j，而编号为 u 的行号为 u/m，列号为 u%m。当格子(x,y)扩展出格子(nx,ny)后，我们不仅需要更新 dist[nx][ny]=dist[x][y]+1，还要保存新格子(nx,ny)的父亲编号 fa[nx][ny]以及从父亲结点到它的移动方向 last_dir[nx][ny]。有了这两个值，就可以把路径打印出来了。

```
void print_path(int x, int y)
{
  int fx = fa[x][y]/m;
  int fy = fa[x][y]%m;
  if(fx != x || fy != y)
  {
    print_path(fx, fy);
    putchar(name[last_dir[x][y]]);
  }
}
```

这里用到了递归的技巧：如果格子(x,y)有父亲(fx,fy)，需要先打印从起点到(fx,fy)的最短路，然后再打印从(fx,fy)到(x,y)的移动方向，也就是 last_dir[x][y]所对应的方向名字。这个函数非常方便，但请当心一点：n 和 m 太大时可能会产生栈溢出（还记得吗？每次递归调用都会消耗栈空间），需要改写成如下的非递归形式。

```
int dir[MAXN*MAXN];
void print_path(int x, int y)
{
  int c = 0;
  for(;;)
  {
    int fx = fa[x][y]/m;
    int fy = fa[x][y]%m;
    if(fx == x && fy == y) break;
    dir[c++] = last_dir[x][y];
    x = fx;
    y = fy;
  }
  while(c--)
    putchar(name[dir[c]]);
}
```

思路很简单：不断沿着父亲指针走，保存方向序列 dir，最后反向输出。注意，我们把前面的递归函数改成上面的形式是为了避免栈溢出，因此需要把 dir 数组声明成全局变量（还记得吗？局部变量保存在栈中，所以即使没有递归调用，把 dir 声明成局部变量也是很危险的）。

6.4.3 拓扑排序

假设有 n 个变量，还有 m 个二元组(u, v)，分别表示变量 u 小于 v。那么，所有变量从

小到大排列起来应该是什么样子的呢？例如有 4 个变量 a, b, c, d，若已知 $a<b, c<b, d<c$，则这 4 个变量的排序可能是 $a<d<c<b$。尽管还有其他可能（如 $d<a<c<b$），你只需找出其中一个即可。

【分析】

把每个变量看成一个点，“小于”关系看成有向边，则我们得到了一个有向图。这样，我们的任务实际上是把一个图的所有结点排序，使得每一条有向边(u,v)对应的 u 都排在 v 的前面。在图论中，这个问题称为拓扑排序（topological sort）。

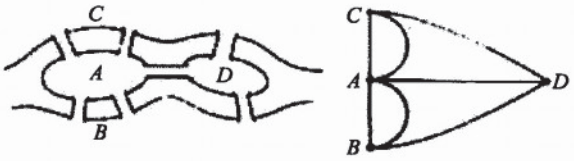
不难发现：如果图中存在有向环，则不存在拓扑排序，反之则存在。我们把不包含有向环的有向图称为有向无环图（Directed Acyclic Graph，DAG）。可以借助 dfs 函数完成拓扑排序：在访问完一个结点之后把它加到当前拓扑序的首部（想一想，为什么不是尾部）。

```
int c[MAXN];
int topo[MAXN], t;
bool dfs(int u)
{
  c[u] = -1;
  for(int v = 0; v < n; v++) if(G[u][v])
  {
    if(c[v]<0) return false; // 存在有向环，失败退出
    else if(!c[v] && !dfs(v)) return false;
  }
  c[u] = 1; topo[--t]=u;
  return true;
}
bool toposort()
{
  t = n;
  memset(c, 0, sizeof(c));
  for(int u = 0; u < n; u++) if(!c[u])
    if(!dfs(u)) return false;
  return true;
}
```

这里用到了一个 c 数组，c[u]=0 表示从来没有访问过（从来没有调用过 dfs(u)）；c[u]=1 表示已经访问过，并且还递归访问过它的所有子孙（即 dfs(u)曾被调用过，并已返回）；c[u]=-1 表示正在访问（即递归调用 dfs(u)正在栈帧中，尚未返回）。

6.4.4　欧拉回路

有一条名为 Pregel 的河流经过 Konigsberg 城。城中有 7 座桥，把河中的两个岛与河岸连接起来。当地居民热衷于一个难题：是否存在一条路线，可以不重复地走遍 7 座桥。这就是著名的七桥问题。它由大数学家欧拉首先提出，并给出了完美的解答，如图 6-14 所示。



(a) (b)

图 6-14 七桥问题

欧拉首先把图 6-14（a）中的七桥问题用图论的语言改写成图 6-14（b），则问题变成了：能否从无向图中的一个结点出发走出一条道路，每条边恰好经过一次。这样的路线称为欧拉道路（eulerian path），也可以形象地称为“一笔画”。

不难发现：在欧拉道路中，“进”和“出”是对应的——除了起点和终点外，其他点的“进出”次数应该相等。换句话说，除了起点和终点外，其他点的度数应该是偶数。很可惜，在七桥问题中，所有 4 个点的度数均是奇数（这样的点也称奇点），因此不可能存在欧拉道路。上述条件也是充分条件——如果一个无向图是连通的，且最多只有两个奇点，则一定存在欧拉道路。如果有两个奇点，则必须从其中一个奇点出发，另一个奇点终止；如果奇点不存在，则可以从任意点出发，最终一定会回到该点（称为欧拉回路）。

用类似的推理方式可以得到有向图的结论：最多只能有两个点的入度不等于出度，而且必须是其中一个点的出度恰好比入度大 1（把它作为起点），另一个的入度比出度大 1（把它作为终点）。当然了，还有一个前提条件：在忽略边的方向后，图必须是连通的。

下面是程序，它同时适用于欧拉道路和回路。但如果需要打印的是欧拉道路，在主程序中调用时，参数必须是道路的起点。另外，打印的顺序是逆序的，因此在真正使用这份代码时，你应当把 printf 语句替换成一条 push 语句，把边(u,v)压入一个栈内。

```
void euler(int u)
{
  for(int v = 0; v < n; v++) if(G[u][v] && !vis[u][v])
  {
    vis[u][v] = vis[v][u] = 1;
    euler(v);
    printf("%d %d\n", u, v);
  }
}
```

尽管上面的代码只适用于无向图，但不难改成有向图：把 vis[u][v] = vis[v][u] = 1 改成 vis[u][v]即可。

6.5 训 练 参 考

首先是一些涉及线性表的题目：纸牌游戏（127）、木块问题（101）、救济金发放（133）、

龟壳排序（10152）、括号平衡（673）、矩阵链乘（442）、一般 Matrioshka（11111）、表达式（11234）、小团体队列（540）、罢工（10050）。

然后是和二叉树有关的题目：树求和（112）、树（548）、四分树（297）、S 树（712）、落叶（699）、计算简单 C 表达式（327）、天平（839）、看图写树（10562）。

最后是和图与图遍历有关的题目：油田（572）、掷色子（657）、迷宫探索（784）、斜线迷宫（705）、骑士移动（439）、三维迷宫（532）、XYZZY（10557）、独轮车（10047）、二染色（10004）、单词（10129）、项链（10054）、清晨漫步（10596）、给任务排序（10305）、电子表格（196）。

第7章 暴力求解法

学习目标

- ☑ 掌握整数、子串等简单对象的枚举方法
- ☑ 熟练掌握排列生成的递归方法
- ☑ 熟练掌握用“下一个排列”枚举全排列的方法
- ☑ 理解解答树，并能估算典型解答树的结点数
- ☑ 熟练掌握子集生成的增量法、位向量法和二进制法
- ☑ 熟练掌握回溯法框架，并能理解为什么它往往比生成-测试法高效
- ☑ 熟练掌握解答树的宽度优先搜索和迭代加深搜索
- ☑ 理解倒水问题的状态图与八皇后问题的解答树的本质区别
- ☑ 熟练掌握八数码问题的 BFS 实现
- ☑ 熟练掌握集合的两种典型实现——hash 表和 STL 集合

很多问题都可以“暴力解决”——不用动太多脑筋，把所有可能性都列举出来，然后一一试验。尽管这样的方法显得很“笨”，但却常常是行之有效的。

7.1 简单枚举

在枚举复杂对象之前，让我们先尝试着枚举一些相对简单的东西，如整数、子串等。尽管暴力枚举不用太动脑筋，但对问题进行一定的分析往往会让算法更加简洁、高效。

7.1.1 除法

输入正整数 n，按从小到大的顺序输出所有形如 $abcde/fghij = n$ 的表达式，其中 a～j 恰好为数字 0～9 的一个排列，$2 \leqslant n \leqslant 79$。

样例输入：

62

样例输出：

79546 / 01283 = 62

94736 / 01528 = 62

【分析】

枚举 0～9 的所有排列？没这个必要。只需要枚举 *fghij* 就可以算出 *abcde*，然后判断是否所有数字都不相同即可。不仅程序简单，而且枚举量也从 10!=3628800 降低至不到 1 万。由此可见，即使采用暴力枚举，也是需要认真分析问题的。

7.1.2　最大乘积

输入 n 个元素组成的序列 S，你需要找出一个乘积最大的连续子序列。如果这个最大的乘积不是正数，应输出-1（表示无解）。$1 \leqslant n \leqslant 18, -10 \leqslant S_i \leqslant 10$。

样例输入：

```
3
2 4-3
5
2 5 -1 2 -1
```

样例输出：

```
8
20
```

【分析】

连续子序列有两个要素：起点和终点，因此只需枚举起点和终点即可。由于每个元素的绝对值不超过 10，一共又不超过 18 个元素，最大可能的乘积不会超过 10^{18}，可以用 long long 存下。

7.1.3　分数拆分

输入正整数 k，找到所有的正整数 $x \geqslant y$，使得 $\frac{1}{k}=\frac{1}{x}+\frac{1}{y}$。

样例输入：

```
2
12
```

样例输出：

```
2
1/2 = 1/6 + 1/3
1/2 = 1/4 + 1/4
8
1/12 = 1/156 + 1/13
1/12 = 1/84 + 1/14
1/12 = 1/60 + 1/15
1/12 = 1/48 + 1/16
1/12 = 1/36 + 1/18
1/12 = 1/30 + 1/20
1/12 = 1/28 + 1/21
1/12 = 1/24 + 1/24
```

【分析】

既然说的是找出所有的 x, y，枚举对象自然就是它们了。可问题在于：枚举的范围如何？从 1/12=1/156+1/13 可以看出，x 可以比 y 大很多。难道要无休止地枚举下去？当然不是。由于 $x \geqslant y$，有 $\frac{1}{x} \leqslant \frac{1}{y}$，因此 $\frac{1}{k} - \frac{1}{y} \leqslant \frac{1}{y}$，即 $y \leqslant 2k$。这样，只需要在 $2k$ 范围之内枚举 y，然后根据 y 尝试计算出 x 即可。

7.1.4 双基回文数

如果一个正整数 n 至少在两个不同的进位制 b_1 和 b_2 下都是回文数（$2 \leqslant b_1, b_2 \leqslant 10$），则称 n 是双基回文数（注意，回文数不能包含前导零）。输入正整数 $S < 10^6$，输出比 S 大的最小双基回文数。

样例输入：1600000

样例输出：1632995

【分析】

最自然的想法就是：从 n+1 开始依次判断每个数是否为双基回文数，而在判断时要枚举所有可能的基数（2～10），一切都是那么的“暴力”。令人有些意外的是：这样做对于 $S<10^6$ 这样的“小规模数据”来说是足够快的——双基回文数太多太密了。

7.2 枚举排列

有没有想过如何打印所有排列呢？输入整数 n，按字典序从小到大的顺序输出前 n 个数的所有排列。两个序列的字典序大小关系等价于从头开始第一个不相同位置处的大小关系。例如，(1,3,2) < (2,1,3)，字典序最小的排列是(1, 2, 3, 4,…, n)，最大的排列是(n, n−1, n−2,…, 1)。n=3 时，所有排列的排序结果是：(1, 2, 3)、(1, 3, 2)、(2, 1, 3)、(2, 3, 1)、(3, 1, 2)、(3, 2, 1)。

7.2.1 生成 1～n 的排列

我们尝试用递归的思想解决：先输出所有以 1 开头的排列（这一步是递归调用），然后输出以 2 开头的排列（又是递归调用），接着是以 3 开头的排列……最后才是以 n 开头的排列。

以 1 开头的排列的特点是：第一位是 1，后面是 2～9 的排列。根据字典序的定义，这些 2～9 的排列也必须按照字典序排列。换句话说，我们需要“按照字典序输出 2～9 的排列”，不过需注意的是，在输出时，每个排列的最前面要加上那个“1”。这样一来，我们设计的递归函数需要以下参数：

- 已经确定的“前缀”序列，以便输出。
- 需要进行全排列的元素集合，以便依次选做第一个元素。

这样，我们得到了一个伪代码：

```
void print_permutation(序列 A, 集合 S)
{
    if(S 为空) 输出序列 A;
    else 按照从小到大的顺序依次考虑 S 的每个元素 v
    {
        print_permutation(在 A 的末尾填加 v 后得到的新序列, S-{v});
    }
}
```

暂时不用考虑序列 A 和集合 S 如何表示，首先理解一下上面的伪代码。递归边界是 S 为空的情形，这很好理解：现在序列 A 就是一个完整的排列，直接输出即可。接下来按照从小到大的顺序考虑 S 中的每个元素，每次递归调用以 A 开头。

下面考虑程序实现。不难想到用数组表示序列 A，而集合 S 根本不用保存，因为它可以由序列 A 完全确定——A 中没有出现的元素都可以选。C 语言中的函数在接受数组参数时无法得知数组的元素个数，所以需要传一个已经填好的位置个数，或者当前需要确定的元素位置 cur，代码如下：

```
void print_permutation(int n, int* A, int cur)
{
  int i, j;
  if(cur == n)                    // 递归边界
  {
    for(i = 0; i < n; i++) printf("%d ", A[i]);
    printf("\n");
  }
  else for(i = 1; i <= n; i++)  // 尝试在 A[cur]中填各种整数 i
  {
    int ok = 1;
    for(j = 0; j < cur; j++)
      if(A[j] == i) ok = 0;       // 如果 i 已经在 A[0]～A[cur-1]出现过，则不能再选
    if(ok)
    {
      A[cur] = i;
      print_permutation(n, A, cur+1); // 递归调用
    }
  }
}
```

循环变量 i 是当前考察的 A[cur]。为了检查元素 i 是否已经用过，上面的程序用到了一个标志变量 ok，初始值为 1（真），如果发现有某个 A[j]==i 时，则改为 0（假）。如果最终 ok 仍为 1，则说明 i 没有在序列中出现过，把它添加到序列末尾（A[cur]=i）后递归调用。

声明一个足够大的数组 A，然后调用 print_permutation(n, A, 0)，即可按字典序输出 1～

n 的所有排列。

7.2.2 生成可重集的排列

如果把问题改成：输入数组 P，并按字典序输出数组 A 各元素的所有全排列，则需要对上述程序进行修改——把 P 加到 print_permutation 的参数列表中，然后把代码中的 if(A[j] == i)和 A[cur] = i 分别改成 if(A[j] == P[i])和 A[cur] = P[i]。这样，只要把 P 的所有元素按从小到大的顺序排序，然后调用 print_permutation(n, P, A, 0)即可。

这个方法看上去不错，可惜有一个小问题：输入 1 1 1 后，程序什么也不输出（正确答案应该是唯一的全排列 1 1 1），原因在于：我们禁止 A 数组中出现重复，而在 P 中本来就有重复元素时，这个“禁令”是错误的。

一个解决方法是统计 A[0]～A[cur-1]中 P[i]的出现次数 c1，以及 P 数组中 P[i]的出现次数 c2。只要 c1<c2，就能递归调用。

```
else for(i = 0; i < n; i++)
{
  int c1 = 0, c2 = 0;
  for(j = 0; j < cur; j++) if(A[j] == P[i]) c1++;
  for(j = 0; j < n; j++) if(P[i] == P[j]) c2++;
  if(c1 < c2)
  {
    A[cur] = P[i];
    print_permutation(n, P, A, cur+1);
  }
}
```

结果又如何呢？输入 1 1 1，输出了 27 个 1 1 1。遗漏倒是没有了，可是出现了重复：我们先试着把第 1 个 1 作为开头，递归调用结束后再尝试用第 2 个 1 作为开头，递归调用结束后再尝试用第 3 个 1 作为开头，再一次递归调用。可实际上这 3 个 1 是相同的，应只递归 1 次，而不是 3 次。

换句话说，我们枚举的下标 i 应不重复、不遗漏地取遍所有 P[i]值。由于 P 数组已经排过序，所以只需检查 P 的第一个元素和所有“与前一个元素不相同”的元素，即只需在 for(i = 0; i < n; i++)和其后的花括号之前加上 if(!i || P[i] != P[i-1])即可。

至此，结果终于正确了。

7.2.3 解答树

假设 n=4，序列为{1,2,3,4}，如图 7-1 所示的树显示出了递归函数的调用过程。其中，结点内部的序列表示 A，位置 cur 用高亮表示，另外，由于从该处开始的元素和算法无关，因此用星号表示。

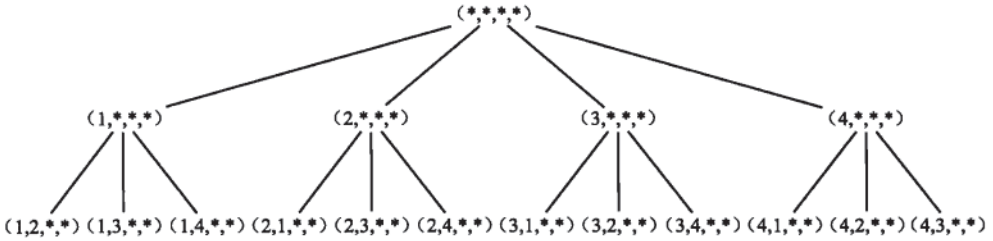


图 7-1　排列生成算法的解答树

这棵树和前面介绍过的二叉树不同。第 0 层（根）结点有 n 个儿子，第 1 层结点各有 $n-1$ 个儿子，第 2 层结点各有 $n-2$ 个儿子，第 3 层结点各有 $n-3$ 个儿子，……，第 n 层结点都没有儿子（即都是叶子），而每个叶子对应于一个排列，共有 $n!$个叶子。由于这棵树展示的是从“什么都没做”逐步生成完整解的过程，因此将其称为解答树。

提示 7-1：如果某问题的解可以由多个步骤得到，而每个步骤都有若干种选择（这些候选方案集可能会依赖于先前作出的选择），且可以用递归枚举法实现，则它的工作方式可以用解答树来描述。

这棵解答树一共有多少个结点呢？可以逐层查看：第 0 层有 1 个结点，第 1 层 n 个，第 2 层有 $n*(n-1)$个结点（因为第 1 层的每个结点都有 $n-1$ 个结点），第 3 层有 $n*(n-1)*(n-2)$个（因为第 2 层的每个结点都有 $n-2$ 个结点），……，第 n 层有 $n*(n-1)*(n-2)*\cdots*2*1=n!$个。

下面把它们加起来。为了推导方便，把 $n*(n-1)*(n-2)*\cdots*(n-k)$写成 $n!/(n-k-1)!$，则所有结点之和为：

$$T(n)=\sum_{k=0}^{n-1}\frac{n!}{(n-k-1)!}=n!\sum_{k=0}^{n-1}\frac{1}{(n-k-1)!}=n!\sum_{k=0}^{n-1}\frac{1}{k!}$$

根据高等数学中的泰勒展开公式，$\lim_{n\to\infty}\sum_{k=0}^{n-1}\frac{1}{k!}=e$，因此$T(n)<n!\cdot e=O(n!)$。由于叶子有 $n!$个，倒数第二层也有 $n!$个结点，因此上面的各层全部加起来也不到 $n!$。这是一个很重要的结论：在多数情况下，解答树上的结点几乎全部来源于最后一两层。和它们相比，上面的结点数可以忽略不计。

不熟悉泰勒展开公式也没有关系：可以写一个程序，输出$\sum_{k=0}^{n-1}\frac{1}{k!}$随着 n 增大时的变化，并发现它能很快收敛。这就是计算机的优点之一——可以通过模拟避开数学推导。即使无法严密而精确地求解，也可以找到令人信服的实验数据。

7.2.4　下一个排列

枚举所有排列的另一个方法是从字典序最小排列开始，不停调用“求下一个排列”的过程。

如何求下一个排列呢？C++的 STL 中提供了一个库函数 next_permutation。看看下面的代码片段，你就会明白如何使用它了。

```
#include<cstdio>
#include<algorithm>
using namespace std;
int main()
{
  int n, p[10];
  scanf("%d", &n);
  for(int i = 0; i < n; i++) scanf("%d", &p[i]);
  sort(p, p+n);                                          // 排序，得到p的最小排列
  do
  {
    for(int i = 0; i < n; i++) printf("%d ", p[i]); // 输出排列p
    printf("\n");
  } while(next_permutation(p, p+n));                     // 求下一个排列
  return 0;
}
```

需要注意的是，上述代码同样适用于可重集。

7.3 子集生成

第7.2节中介绍了排列生成算法。本节介绍子集生成算法：给定一个集合，枚举它所有可能的子集。为了简单起见，本节讨论的集合中没有重复元素。

7.3.1 增量构造法

第一种思路是一次选出一个元素放到集合中，程序如下：

```
void print_subset(int n, int* A, int cur)
{
  for(int i = 0; i < cur; i++) printf("%d ", A[i]);     // 打印当前集合
  printf("\n");
  int s = cur ? A[cur-1]+1 : 0;          // 确定当前元素的最小可能值
  for(int i = s; i < n; i++)
  {
    A[cur] = i;
    print_subset(n, A, cur+1);           // 递归构造子集
  }
}
```

和前面不同：由于A中的元素个数不确定，每次递归调用都要输出当前集合。另外，递归边界也不需要显式确定——如果无法继续添加元素，自然就不会再递归了。

上面的代码用到了定序的技巧：规定集合 A 中所有元素的编号从小到大排列，就不会把集合{1, 2}按照{1, 2}和{2, 1}输出两次了。

顺便说一句，这棵解答树上有 1024 个结点。这不难理解：每个可能的 A 都对应一个结点，而 n 元素集合恰好有 2^n 个子集，2^{10}=1024。

7.3.2　位向量法

第二种思路是构造一个位向量 B[i]，而不是直接构造子集 A 本身，其中 B[i]=1 当且仅当 i 在子集 A 中。递归实现如下：

```
void print_subset(int n, int* B, int cur)
{
  if(cur == n)
  {
    for(int i = 0; i < cur; i++)
      if(B[i]) printf("%d ", i);          // 打印当前集合
    printf("\n");
    return;
  }
  B[cur] = 1;                             // 选第 cur 个元素
  print_subset(n, B, cur+1);
  B[cur] = 0;                             // 不选第 cur 个元素
  print_subset(n, B, cur+1);
}
```

必须当“所有元素是否选择”全部确定完毕后才是一个完整的子集，因此仍然像以前那样当 if(cur == n)成立时才输出。现在的解答树上有 2047 个结点，比刚才的方法略多。这个也不难理解：所有部分解（不完整的解）也对应着解答树上的结点。

这是一棵 n+1 层的二叉树（cur 从 0～n），第 0 层有 1 个结点，第 1 层有 2 个结点，第 2 层有 4 个结点，第 3 层有 8 个结点，……，第 i 层有 2^i 个结点，总数为 $1+2+4+8+\cdots+2^n=2^{n+1}-1$，和实验结果一致。图 7-2 所示便是这棵解答树。

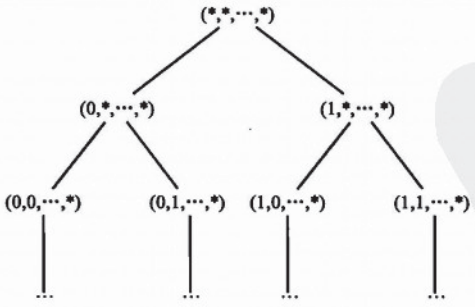


图 7-2　位向量法的解答树

这棵树依然符合前面的观察结果：最后几层结点数占整棵树的绝大多数。

7.3.3 二进制法

另外，还可以用二进制来表示{0, 1, 2,…,n−1}的子集 S：从右往左第 i 位（各位从 0 开始编号）表示元素 i 是否在集合 S 中。图 7-3 展示了二进制 0100011000110111 是如何表示集合{0, 1, 2, 4, 5, 9, 10, 14}的。

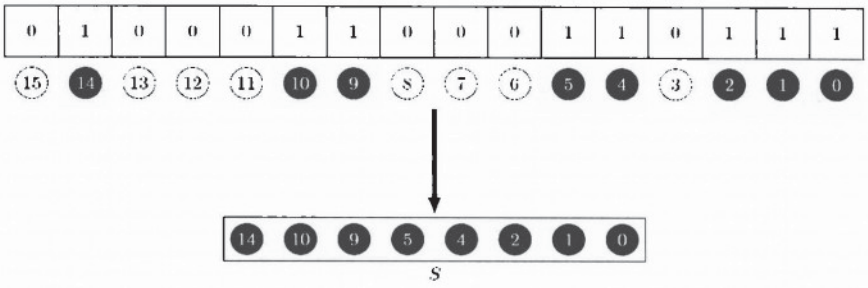


图 7-3 用二进制表示子集

注意：为了处理方便，最右边那个位总是对应元素 0，而不是元素 1。

提示 7-2：*可以用二进制表示子集，其中从右往左第 i 位（从 0 开始编号）表示元素 i 是否在集合中（1 表示“在”，0 表示“不在”）。*

此时仅仅表示出集合是不够的，还需要对集合进行操作。幸运的是，常见的集合运算都可以用位运算符简单实现。最常见的二元位运算是与（&）、或（|）、非（!），它们和对应的逻辑运算非常相似，如表 7-1 所示。

表 7-1 C 语言中的二元位运算

A	B	A&B	A|B	A^B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

表 7-1 中包括了“异或（XOR）”运算符^，它的规则是“如果 A 和 B 不相同，则 A^B 为 1，否则为 0”。异或运算最重要的性质就是“开关性”——异或两次以后相当于没有异或，即 A^B^B=A。另外，与、或和异或都满足交换率：A&B=B&A，A|B=B|A，A^B=B^A。

与逻辑运算符不同的是，位运算符（bitwise operator）是逐位进行的——两个 32 位整数的“按位与”相当于 32 对 0/1 值之间的运算。表 7-2 中表示了二进制数 10110（十进制为 22）和 01100（十进制为 12）之间的按位与、按位或、按位异或的值，以及对应的集合运算的含义。

表 7-2　位运算与集合运算

	A	B	A&B	A|B	A^B
二进制	10110	01100	00100	11110	11010
集合	{1,2,4}	{2,3}	{2}	{1,2,3,4}	{1,3,4}

不难看出，A&B，A|B 和 A^B 分别对应集合的交、并和对称差。另外，空集为 0，全集{0, 1, 2,…, n-1}的二进制为 n 个 1，即十进制的 2^n-1。为了方便，往往在程序中把全集定义为 ALL_BITS= (1<<n)-1，则 A 的补集就是 ALL_BITS^A。当然，直接用整数减法 ALL_BITS -A 也可以，但速度比位运算^慢。

提示 7-3：当用二进制表示子集时，位运算中的按位与、或、异或对应集合的交、并和对称差。

这样，不难用下面的程序输出子集 S 对应的各个元素：

```
void print_subset(int n, int s)     // 打印{0, 1, 2,…, n-1}的子集 S
{
  for(int i = 0; i < n; i++)
    if(s&(1<<i)) printf("%d ", i);  // 这里利用了 C 语言“非 0 值都为真”的规定
  printf("\n");
}
```

而枚举子集和枚举整数一样简单：

```
  for(int i = 0; i < (1<<n); i++)   // 枚举各子集所对应的编码 0, 1, 2,…, 2^n-1
    print_subset(n, i);
```

7.4　回　溯　法

无论是排列生成还是子集枚举，前面都给出了两种思路：递归构造和直接遍历。直接遍历法的优点是思路和程序都很简单，缺点在于无法简便地减小枚举量——必须生成（generate）所有可能的解，然后一一检查（test）。

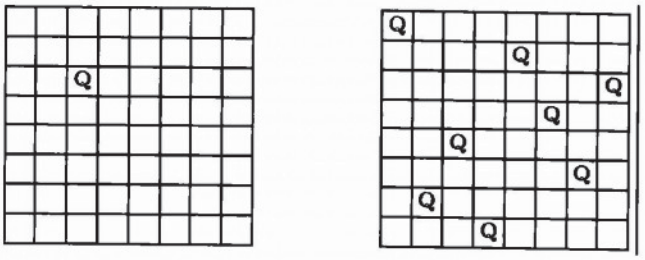
另一方面，在递归构造中，生成和检查过程就可以有机结合起来，从而减小不必要的枚举。这就是本节的主题——回溯法（backtracking）。

回溯法的应用范围很广，只要能把待求解的问题分成不太多的步骤，每个步骤又只有不太多的选择，都可以考虑应用回溯法。为什么说“不太多”呢？想象一棵包含 L 层，每层的分支因子均为 b 的解答树，它的结点数高达 $1+b+b^2+\cdots+b^{L-1}=\frac{b^L-1}{b-1}$。无论是 b 太大还是 L 太大，结点数都会是一个天文数字。

7.4.1　八皇后问题

在棋盘上放置 8 个皇后，使得她们互不攻击，此时每个皇后的攻击范围为同行同列和

同对角线，要求找出所有解，如图 7-4 所示。



（a）皇后的攻击范围　　（b）一个可行解

图 7-4　八皇后问题

【分析】

最简单的思路是把问题转化为“从 64 个格子中选一个子集”，使得“子集中恰好有 8 个格子，且任意两个选出的格子都不在同一行、同一列或同一个对角线上”。这正是子集枚举问题。然而，64 个格子的子集有 2^{64} 个，太大了，这并不是一个很好的模型。

第二个想法是把问题转化为“从 64 格格子中选 8 个格子”，这是组合生成问题。根据组合数学，有 $C_{64}^{8}=4.426\times10^{9}$ 种方案，比第一种方案优秀，但仍然不够好。

经过思考，不难发现以下事实：恰好每行每列各放置一个皇后。如果用 C[x]表示第 x 行皇后的列编号，则问题变成了全排列生成问题。而我们知道，0～7 的排列一共只有 8!=40320 个，枚举量不会超过它。

提示 7-4：*在编写递归枚举程序之前，需要深入分析问题，对模型精雕细琢。一般还应对解答树的结点数有一个粗略的估计，作为评价模型的重要依据，如图 7-5 所示。*

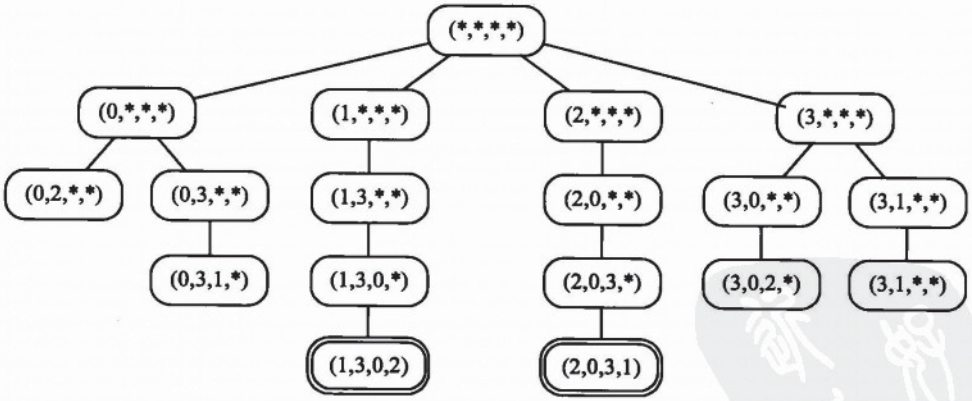


图 7-5　四皇后问题的解答树

图 7-5 中给出了四皇后问题的完整解答树。它只有 18 个结点，比 4!=24 小。为什么会这样呢？这是因为有些结点无法继续扩展。例如在(0,2,*,*)中，第 2 行无论将皇后放哪里，都会和第 0 行和第 1 行中已放好的皇后发生冲突，至于其他还未放置的皇后更是如此。

在这种情况下，递归函数将不再递归调用它自身，而是返回上一层调用，我们称这种现象为回溯（backtracking）。

提示 7-5：当把问题分成若干步骤并递归求解时，如果当前步骤没有合法选择，则函数将返回上一级递归调用，这种现象称为回溯。正是因为这个原因，递归枚举算法常被称为回溯法，它的应用十分普遍。

下面的程序简洁地求解了八皇后问题。在主程序中读入 n，并为 tot 清 0，然后调用 search(0)，即可得到解的个数 tot。

```
void search(int cur)
{
  int i, j;
  if(cur == n) tot++;                    // 递归边界。只要走到了这里，所有皇后必然不冲突
  else for(i = 0; i < n; i++)
  {
    int ok = 1;
    C[cur] = i;                          // 尝试把第 cur 行的皇后放在第 i 列
    for(j = 0; j < cur; j++)             // 检查是否和前面的皇后冲突
      if(C[cur] == C[j] || cur-C[cur] == j-C[j] || cur+C[cur] == j+C[j]) { ok
= 0; break; }
    if(ok) search(cur+1);                // 如果合法，则继续递归
  }
}
```

注意：既然是逐行放置的，则皇后肯定不会横向攻击，因此只需检查是否纵向和斜向攻击即可。条件 cur−C[cur] == j−C[j] || cur+C[cur] == j+C[j]用来判断皇后(cur,C[cur])和(j,C[j])是否在同一条对角线上。其原理可以用图 7-6 来说明。

0	1	2	3	4	5	6	7
-1	0	1	2	3	4	5	6
-2	-1	0	1	2	3	4	5
-3	-2	-1	0	1	2	3	4
-4	-3	-2	-1	0	1	2	3
-5	-4	-3	-2	-1	0	1	2
-6	-5	-4	-3	-2	-1	0	1
-7	-6	-5	-4	-3	-2	-1	0

（a）格子(x,y)的 y−x 值标识了主对角线

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14

（b）格子(x,y)的 x+y 值标识了副对角线

图 7-6　棋盘中的对角线标识

结点数似乎很难进一步减少了，但程序效率可以继续提高：利用二维数组 vis[2][]直接判断当前尝试的皇后所在的列和两个对角线是否已有其他皇后。注意到主对角线标识 y−x 可能为负，存取时要加上 n。

```
void search(int cur)
{
```

```
    int i, j;
    if(cur == n) tot++;
    else for(i = 0; i < n; i++)
    {
      if(!vis[0][i] && !vis[1][cur+i] && !vis[2][cur-i+n])
                                          // 利用二维数组直接判断
      {
        C[cur] = i;                       // 如果不用打印解，整个C数组都可以省略
        vis[0][i] = vis[1][cur+i] = vis[2][cur-i+n] = 1;  // 修改全局变量
        search(cur+1);
        vis[0][i] = vis[1][cur+i] = vis[2][cur-i+n] = 0;  // 切记！一定要改回来
      }
    }
  }
```

上面的程序有个极其关键的地方：vis 数组的使用。vis 数组的确切含义是什么？它表示已经放置的皇后占据了哪些列、主对角线和副对角线。将来放置的皇后不应该修改这些值——至少“看上去没有修改”。一般地，如果在回溯法中修改了辅助的全局变量，则一定要及时把它们恢复原状（除非你故意保留你的修改）。若不信，你可以把“vis[0][i]= vis[1][cur+i] = vis[2][cur−i+n] = 0”注释掉，看看还能否正确求解八皇后问题。另外，千万不要忘记在调用之前把 vis 数组清空。

提示 7-6：*如果在回溯法中使用了辅助的全局变量，则一定要及时把它们恢复原状。例如，若函数有多个出口，则需在每个出口处恢复被修改的值。*

7.4.2 素数环

输入正整数 n，把整数 1, 2, 3,…, n 组成一个环，使得相邻两个整数之和均为素数。输出时从整数 1 开始逆时针排列。同一个环应恰好输出一次。$n \leqslant 16$。

样例输入：

6

样例输出：

1 4 3 2 5 6

1 6 5 2 3 4

【分析】

模型不难得到：每个环对应于 1～n 的一个排列，但排列总数高达 $16! = 2 \times 10^{13}$，生成-测试法会超时吗？下面进行实验：

```
    for(int i = 2; i <= n*2; i++) isp[i] = is_prime(i);  // 生成素数表，加快后续判断
    for(int i = 0; i < n; i++) A[i] = i+1;               // 第一个排列
    do
    {
```

```
    int ok = 1;
    for(int i = 0; i < n; i++) if(!isp[A[i]+A[(i+1)%n]]) { ok = 0; break; }
                                                          // 判断合法性
    if(ok)
    {
      for(int i = 0; i < n; i++) printf("%d ", A[i]);    // 输出序列
      printf("\n");
    }
  }while(next_permutation(A+1, A+n));                     // 1 的位置不变
```

运行后发现：当 n=12 时就已经很慢，而当 n=16 时根本出不来。好了，试试回溯法吧：

```
void dfs(int cur)
{
  if(cur == n && isp[A[0]+A[n-1]]) // 递归边界。别忘了测试第一个数和最后一个数
  {
    for(int i = 0; i < n; i++) printf("%d ", A[i]);     // 打印方案
    printf("\n");
  }
  else for(int i = 2; i <= n; i++) // 尝试放置每个数 i
    if(!vis[i] && isp[i+A[cur-1]]) // 如果 i 没有用过，并且与前一个数之和为素数
    {
      A[cur] = i;
      vis[i] = 1;                  // 设置使用标志
      dfs(cur+1);
      vis[i] = 0;                  // 清除标志
    }
}
```

回溯法比生成-测试法快了很多，即使 n=18 速度也不错。顺便说一句，把上面的函数名取为 dfs 并不是巧合——从解答树的角度讲，回溯法正是按照深度优先的顺序在遍历解答树。在后面的内容中，我们还将学习更多遍历解答树的方法。

7.4.3 困难的串

如果一个字符串包含两个相邻的重复子串，则称它是“容易的串”，其他串称为“困难的串”。例如，BB、ABCDACABCAB、ABCDABCD 都是容易的，而 D、DC、ABDAB、CBABCBA 都是困难的。

输入正整数 n 和 L，输出由前 L 个字符组成的、字典序第 k 小的困难的串。例如，当 L=3 时，前 7 个困难的串分别为：A、AB、ABA、ABAC、ABACA、ABACAB、ABACABA。输入保证答案不超过 80 个字符。

样例输入：

7 3

30 3

样例输出：

ABACABA

ABACABCACBABCABACABCACBACABA

【分析】

基本框架不难确定：从左到右依次考虑每个位置上的字符。因此，问题的关键在于：如何判断当前字符串是否已经存在连续的重复子串。例如，如何判断 ABACABA 是否包含连续重复子串呢？一种方法是检查所有长度为偶数的子串，分别判断每个字串的前一半是否等于后一半。尽管是正确的，但这个方法做了很多无用功。还记得八皇后问题中我们是怎么判断合法性的吗？我们判断当前皇后是否和前面的皇后冲突，但并不判断以前的皇后是否相互冲突——那些皇后在以前已经判断过了。同样的道理，我们只需要判断当前串的后缀，而非所有子串。程序如下：

```
int dfs(int cur)                              // 返回 0 表示已经得到解，无须继续搜索
{
  if(cnt++ == n)
  {
    for(int i = 0; i < cur; i++) printf("%c", 'A'+S[i]); // 输出方案
    printf("\n");
    return 0;
  }
  for(int i = 0; i < L; i++)
  {
    S[cur] = i;
    int ok = 1;
    for(int j = 1; j*2 <= cur+1; j++)   // 尝试长度为 j*2 的后缀
    {
      int equal = 1;
      for(int k = 0; k < j; k++)        // 检查后一半是否等于前一半
        if(S[cur-k] != S[cur-k-j]) { equal = 0; break; }
      if(equal) { ok = 0; break; }      // 后一半等于前一半，方案不合法
    }
    if(ok) if(!dfs(cur+1)) return 0;    // 递归搜索。如果已经找到解，则直接退出
  }
  return 1;
}
```

有意思的是，$L=2$ 时一共只有 6 个串；当 $L\geqslant 3$ 时就很少回溯了。事实上，当 $L=3$ 时，可以构造出无限长的串，不存在相邻重复子串。

7.4.4 带宽

给出一个 n 个结点的图 G 和一个结点的排列，定义结点 i 的带宽 $b(i)$ 为 i 和相邻结点在

排列中的最远距离，而所有 $b(i)$的最大值就是整个图的带宽。给定图 G，求出让带宽最小的结点排列。

【分析】

如果不考虑效率，本题可以递归枚举全排列，分别计算带宽，然后选取最小的一种方案。能否优化呢？和八皇后问题不同的是：八皇后问题有很多可行性约束（feasibility constraint），可以在得到完整解之前避免扩展那些不可行的结点，但本题并没有可行性约束——任何排列都是合法的。难道我们只能老老实实地扩展所有结点了吗？当然不是。

可以记录下目前已经找到的最小带宽 k。如果发现已经有某两个结点的距离大于或等于 k，再怎么扩展也不可能比当前解更优，应当强制把它“剪”掉——就像园丁在花园里为树修剪枝叶一样，也可以为解答树“剪枝（prune）”。

除此之外，我们还可以剪掉更多的枝叶。如果在搜索到结点 u 时，u 结点还有 m 个相邻点没有确定位置，那么对于结点 u 来说，最理想的情况就是这 m 个结点紧跟在 u 后面，这样的结点带宽为 m，而其他任何“非理想情况”的带宽至少为 m+1。这样，如果 $m \geq k$，即“在最理想的情况下都不能得到比当前最优解更好的方案”，则显然应当剪枝。

7.5　隐式图搜索

在第 6 章中，我们曾经接触过图的遍历。很多问题都可以用图的遍历算法解决，但这些问题中的图却不是事先给定、从程序读入的，而是由程序动态生成的。

7.5.1　隐式树的遍历

到目前为止，本章中的解答树都是真真正正的树，有着严格的“层次”概念——从根开始往下走，你永远都无法回到根结点。例如，在八皇后问题中，皇后是越放越多的，因此当放了 5 个皇后以后，无论怎样继续放，也无法回到“只放了 3 个皇后”的状态。

回溯法是按照深度优先顺序遍历的，它的优点是空间很节省：只有递归栈中的结点需要保存。换句话说，回溯法的空间开销和访问到的最深结点的深度成正比。

前面介绍过，树不仅可以深度优先遍历，还可以宽度优先遍历。这样做的好处是：找到的第一个解一定是离根最近的解（想一想，为什么），但空间开销却大大增加（结点队列中的结点可能很多！）。

例题 7-1　最优程序

有一台只有一个数据栈的计算机，支持整数的 5 种运算：ADD、SUB、MUL、DIV、DUP。假设栈顶的 3 个元素分别为 a、b、c，则 5 种运算的效果依次如下。

ADD：a 和 b 依次出栈，计算 $a+b$，把结果压栈。

SUB：a 和 b 依次出栈，计算 $b-a$，把结果压栈。

MUL：a 和 b 依次出栈，计算 $a \times b$，把结果压栈。

DIV：a 和 b 依次出栈，计算 b/a 并向下取整，把结果压栈。

DUP：a 出栈，再连续把两个 a 压栈。

遇到以下情况之一，机器会产生错误：执行 DIV 操作时，栈顶元素为 0；执行 ADD、SUB、MUL、DIV 操作时，栈里只有一个元素；运算结果的绝对值大于 30000。

给出 n 个整数对(a_i,b_i)（a_i 互不相同），你的任务是设计一个最短的程序，使得对于 1 和 n 之间的所有 i，如果程序执行前栈中只有一个元素 a_i，则执行后栈顶元素是 b_i。$n \leqslant 5$。

【分析】

如果在这道题中用回溯法，情况会怎样？似乎没有办法回溯？是的，本题的解答树是无穷大的——多长的程序都是合法的。由于本题要求最短的程序，最简单可行的方法就是利用宽度优先遍历解答树。

例题 7-2　埃及分数

在古埃及，人们使用单位分数的和（如 $1/a$，a 是自然数）表示一切有理数。例如，$2/3=1/2+1/6$，但不允许 $2/3=1/3+1/3$，因为在加数中不允许有相同的。

对于一个分数 a/b，表示方法有很多种，其中加数少的比加数多的好，如果加数个数相同，则最小的分数越大越好。例如，$19/45=1/5+1/6+1/18$ 是最优方案。

输入整数 a,b $(0<a<b<1000)$，试编程计算最佳表达式。

【分析】

这次的解答树更“恐怖”了——不仅深度没有明显的上界，而且加数的选择在理论上也是无限的。换句话说，如果用宽度优先遍历，连一层都扩展不完（因为每一层都是无限大的）。

解决方案是采用迭代加深搜索（iterative deepening）：从小到大枚举深度上限 d，每次只考虑深度不超过 d 的结点。这样，只要解的深度有限，则一定可以在有限时间内枚举到。

深度上限还可以用来剪枝。我们按照分母递增的顺序来进行扩展，如果扩展到 i 层时，前 i 个分数之和为 c/d，而第 i 个分数为 $1/e$，则接下来至少还需要$(a/b-c/d) / (1/e)$个分数，总和才能达到 a/b。例如，当前搜索到 $19/45=1/5+1/100+\cdots$，则后面的分数每个最大为 $1/101$，至少需要$(19/45-1/5) / (1/101) =23$ 项总和才能达到 $19/45$，因此前 22 次迭代是根本不会考虑这棵子树的。使用了上述算法，题目规模中的任意数据都能很快解出。

7.5.2　一般隐式图的遍历

也有很多题目难以建立有严格层次的解答树，于是就只能把状态之间的关系理解成一般意义下的图。

例题 7-3　倒水问题

有装满水的 6 升的杯子、空的 3 升杯子和 1 升杯子，3 个杯子中都没有刻度。在不使用其他道具的情况下，是否可以量出 4 升的水呢？

方法如图 7-7 所示。

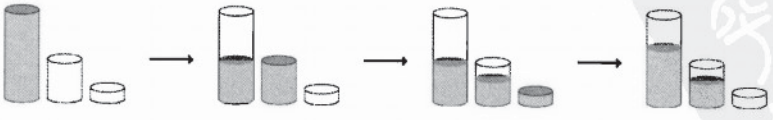


图 7-7　倒水问题：一种方法是(6,0,0)→(3,3,0)→(3,2,1)→(4,2,1)

注意：由于没有刻度，用杯子 x 给杯子 y 倒水时必须一直持续到把杯子 y 倒满或者把杯子 x 倒空，而不能中途停止。

你的任务是解决一般性的问题：设大、中、小 3 个杯子的容量分别为 a, b, c，最初只有大杯子装满水，其他两个杯子为空。最少需要多少步才能让某一个杯子中的水有 x 升呢？你需要打印出每步操作后各个杯子中的水量（$0 < c < b < a < 1000$）。

【分析】

假设在某一时刻，大杯子中有 v_0 升水，中杯子中有 v_1 升水，小杯子中有 v_2 升水，我们称当时的系统状态为(v_0,v_1,v_2)。这里提到了“状态”这个词，它是理解很多概念和算法的关键。简单地说，它就是“对系统当前状况的描述”。例如，在国际象棋中，当前游戏者和棋盘上的局面就是刻画游戏进程的状态。

把“状态”想象成图中的结点，可以得到如图 7-8 所示的状态图（state graph）。

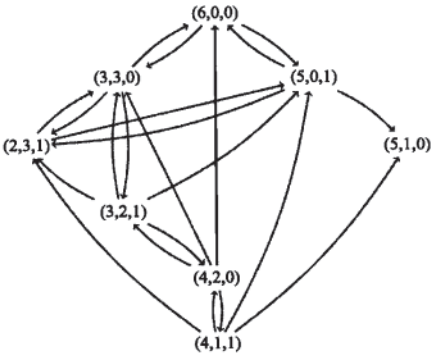


图 7-8　倒水问题的状态图

注意图 7-8 所示的状态图和迷宫是不同的。在迷宫中，只要可以成功往上走，走完后往下一定会回到刚才的地方，但此图并不相同：从状态(4,2,0)可以通过把水从中杯子倒入大杯子而一步到达(6,0,0)，但是从(6,0,0)却无法一步到达(4,2,0)。换句话说，迷宫图是无向图（undirected graph），而本题的“倒水状态图”是有向图（directed graph）。不过这并不要紧——BFS 同样适用于有向图。

由于无论如何倒，杯子中的水量都是整数（按照倒水次数归纳即可），因此小杯的水量最多只有 0, 1, 2,…, c 共 c+1 种可能；同理，中杯子的水量一共只有 b+1 种可能，大杯子一共只有 a+1 种可能，因此理论上状态最多有$(a+1)(b+1)(c+1)$种可能性。当 a、b、c 接近题目中的极限规模时，状态图中会有接近 10^9 个结点。

幸运的是，上面的估计是不精确的。由于水的总量 x 永远不变，如果有两个状态的小杯水量和中杯水量都相同，则大杯水量也相同。换句话说，最多可能的状态数不会超过 $(b+1)(c+1)$，因此结点数不超过 10^6，边数不超过 $6*10^6$，可以承受了。

7.5.3　八数码问题

编号为 1～8 的 8 个正方形滑块被摆成 3 行 3 列（有一个格子留空），如图 7-9 所示。

每次可以把与空格相邻的滑块（有公共边才算相邻）移到空格中，而它原来的位置就成为了新的空格。给定初始局面和目标局面（用 0 表示空格），你的任务是计算出最少的移动步数。如果无法到达目标局面，则输出-1。

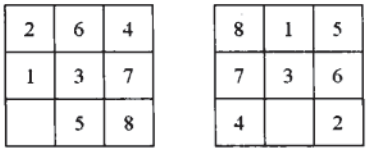


图 7-9　八数码问题举例

样例输入：

2 6 4 1 3 7 0 5 8

8 1 5 7 3 6 4 0 2

样例输出：

31

【分析】

不难把八数码问题归结为图上的最短路问题，其中每个状态就是 9 个格子中的滑块编号（从上到下、从左到右地把它们放到一个包含 9 个元素的数组中）。具体程序如下：

```
typedef int State[9];                          // 定义“状态”类型
const int MAXSTATE = 1000000;
State st[MAXSTATE], goal;                      // 状态数组。所有状态都保存在这里
int dist[MAXSTATE];                            // 距离数组
// 如果需要打印方案，可以在这里加一个“父亲编号”数组 int fa[MAXSTATE]

const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};
// BFS，返回目标状态在 st 数组下标
int bfs()
{
  init_lookup_table();                         // 初始化查找表
  int front = 1, rear = 2;                     // 不使用下标 0，因为 0 被看作“不存在”
  while(front < rear)
  {
    State& s = st[front];                                    // 用“引用”简化代码
    if(memcmp(goal, s, sizeof(s)) == 0) return front;// 找到目标状态，成功返回
    int z;
    for(z = 0; z < 9; z++) if(!s[z]) break;                  // 找“0”的位置
    int x = z/3, y = z%3;                                    // 获取行列编号（0～2）
    for(int d = 0; d < 4; d++)
    {
      int newx = x + dx[d];
```

```
      int newy = y + dy[d];
      int newz = newx * 3 + newy;
      if(newx >= 0 && newx < 3 && newy >= 0 && newy < 3)    // 如果移动合法
      {
        State& t = st[rear];
        memcpy(&t, &s, sizeof(s));                          // 扩展新结点
        t[newz] = s[z];
        t[z] = s[newz];
        dist[rear] = dist[front] + 1;                       // 更新新结点的距离值
        if(try_to_insert(rear)) rear++;                     // 如果成功插入查找表，修改队尾指针
      }
    }
    front++;                                                // 扩展完毕后再修改队首指针
  }
  return 0;                                                 // 失败
}
```

注意，此处用到了 string.h 中的 memcmp 和 memcpy 完成整块内存的比较和复制，比用循环比较和循环赋值要快。主程序很容易实现：

```
int main()
{
  for(int i = 0; i < 9; i++) scanf("%d", &st[1][i]);    // 起始状态
  for(int i = 0; i < 9; i++) scanf("%d", &goal[i]);     // 目标状态
  int ans = bfs();                                      // 返回目标状态的下标
  if(ans > 0) printf("%d\n", dist[ans]);
  else printf("-1\n");
  return 0;
}
```

注意，应在调用 bfs 函数之前设置好 st[1]和 goal。上面的代码几乎是完整的，唯一没有涉及的是 init_lookup_table()和 try_to_insert(rear)的实现。为什么会有这个东西呢？还记得 BFS 中的 vis 数组吗？我们用它进行 BFS 中的判重。这里的查找表和它的功能类似，也是避免我们将同一个结点访问多次。树的 BFS 不需要判重，因为根本不会重复；但对于图来说，如果不判重，时间和空间都将产生极大的浪费。

7.5.4 结点查找表

如何判重呢？难道要声明一个 9 维数组 vis，然后 if(vis[s[0]][s[1]][s[2]]…s[8])？无论程序好不好看，9 维数组的每维都要包含 9 个元素，一共有 9^9=387420489 项，太多了，数组开不下。实际的结点数并没有这么多（0～8 的排列总共只有 9!=362880 个），为什么 9 维数组开不下呢？原因在于，这样的用法存在大量的浪费——数组中有很多项都没有被用到，但却占据了空间。

下面通过讨论3种常见的方法来解决这个问题，同时将它们用到八数码问题中。

第1种方法是：把排列“变成”整数，然后只开一个一维数组。也就是说，我们设计一套排列的编码（encoding）和解码（decoding）函数，把0～8的全排列和0～362879的整数一一对应起来。我们将在第10章中详细讨论编码和解码问题，这里先给出代码以便为读者提供一个感性认识：

```
int vis[36288], fact[9];
void init_lookup_table()
{
  fact[0] = 1;
  for(int i = 1; i < 9; i++) fact[i] = fact[i-1] * i;
}
int try_to_insert(int s)
{
  int code = 0; // 把st[s]映射到整数code
  for(int i = 0; i < 9; i++)
  {
    int cnt = 0;
    for(int j = i+1; j < 9; j++) if(st[s][j] < st[s][i]) cnt++;
    code += fact[8-i] * cnt;
  }
  if(vis[code]) return 0;
  return vis[code] = 1;
}
```

尽管原理巧妙，时间效率也非常高，但编码解码法的适用范围并不大：如果隐式图的总结点数非常大，编码也将会很大，数组还是开不下。

第2种方法是使用哈希（hash）技术。简单地说，就是要把结点“变成”整数，但不必是一一对应。换句话说，只需要设计一个所谓的哈希函数$h(x)$，然后将任意结点x映射到某个给定范围$[0, M-1]$的整数即可，其中M是程序员根据可用内存大小**自选**的。在理想情况下，只需开一个大小为M的数组就能完成判重，但此时往往会有不同结点的哈希值相同，因此需要把哈希值相同的状态组织成链表，细节参见下面的代码：

```
const int MAXHASHSIZE = 1000003;
int head[MAXHASHSIZE], next[MAXSTATE];
void init_lookup_table() { memset(head, 0, sizeof(head)); }
int hash(State& s)
{
  int v = 0;
  for(int i = 0; i < 9; i++) v = v * 10 + s[i];// 随便算。例如，把9个数字组合成9位数
  return v % MAXHASHSIZE;                      // 确保hash函数值是不超过hash表的大小的非负整数
```

```
}
int try_to_insert(int s)
{
  int h = hash(st[s]);
  int u = head[h];                                          // 从表头开始查找链表
  while(u)
  {
    if(memcmp(st[u],st[s], sizeof(st[s]))==0)return 0;  // 找到了，插入失败
    u = next[u];                                            // 顺着链表继续找
  }
  next[s] = head[h];                                        // 插入到链表中
  head[h] = s;
  return 1;
}
```

哈希表的执行效率高，适用范围也很广。除了BFS中的结点判重外，你还可以把它用到其他需要快速查找的地方。不过需要注意的是：在哈希表中，对效率起到关键作用的是哈希函数。如果哈希函数选取得当，几乎不会有结点的哈希值相同，且此时链表查找的速度也较快；但如果冲突严重，整个哈希表会退化成少数几条长长的链表，查找速度将非常缓慢。有趣的是，前面的编码函数可以看作是一个完美的哈希函数，不需要解决冲突。不过，如果你事先并不知道它是完美的，也就不敢像前面一样只开一个vis数组。哈希技术还有很多值得探讨的地方，建议读者在网上查找相关资料。

第3种方法是用STL中的集合。读者如果在前面试过STL的栈和队列，就可以理解下面的定义：set<State> vis。它声明了一个类型为State的集合vis。这样，只需用if(vis.count(s))来判断s是否在集合vis中[①]，并用vis.insert(s)把s加入集合，用vis.remove(s)从集合中移除s。但问题在于，并不是所有类型的State都可以作为set中的元素类型。STL要求set的元素类型必须定义“<”运算符，如int、string或前面自定义的bign，但C语言原生的数组（包括字符数组）却不行。下面是一种使用int的方法：

```
set<int> vis;
void init_lookup_table() { vis.clear(); }
int try_to_insert(int s)
{
  int v = 0;
  for(int i = 0; i < 9; i++) v = v * 10 + st[s][i];
  if(vis.count(v)) return 0;
  vis.insert(v);
  return 1;
}
```

但在很多其他场合中，数组是没有办法简单转化成整数的，就只能像下面一样，自己

① 更加推荐的做法是判断vis.find(s)是否为vis.end()，但不如这里的方法容易被初学者接受。

声明一个结构体，并重载“括号运算”来比较两个状态。在下面的程序中，整数 a 和 b 分别是两个状态在状态数组 st 中的下标，在比较时直接使用 memcpy 来比较整个内存块。

```
struct cmp
{
  bool operator()(int a, int b) const   // 重新定义 a < b 的含义
  {
    return memcmp(&st[a], &st[b], sizeof(st[b])) < 0;
  }
};
set<int, cmp> vis;                      // 使用新的“a < b”来定义“整数集合”
void init_lookup_table() { vis.clear(); }
int try_to_insert(int s)
{
  if(vis.count(s)) return 0;
  vis.insert(s);
  return 1;
}
```

这个实现明显比刚才那个要慢很多，因为调用 memcmp 比直接比较两个整数要慢得多。事实上，在刚才的 3 种实现中，使用 STL 集合的代码最简单，但时间效率也最低（若此时不用-O2 优化则速度劣势更加明显）。建议读者在时间紧迫或对效率要求不太高的情况下使用，或者仅仅把它作为“跳板”——先写一个 STL 版的程序，确保主算法正确，然后把 set 替换成自己写的哈希表。

最后，笔者还想提一句：某些特定的 STL 实现中还有 hash_set，它正是基于前面的哈希表，但它并不是标准 C++的一部分，因此不是所有情况下都可用。

7.6 训 练 参 考

和第 6 章一样，这里也只给出 UVaOJ 题库中的题目。如果能完成这些题目的 80%，读者对暴力求解法的掌握就会比较扎实了。

首先是一些基础题目，如直接枚举法、下一个排列等：生日蛋糕（10167）、损坏的步数计（11205）、有超能力的纸牌玩家（131）、ID 码（146）、快速生成有序排列（10098）、Hamming 距离问题（729）、逻辑之岛（592）、没有循环的排序程序（110）。

接下来是需要用回溯法解决的问题：用栈重排字母（10474）、网络连线（216）、放车问题（639）、卡坦岛（539）、运输（301）、算 23 点（10344）、交换的方案数（331）、有多大（10012）、邮票（165）、苏丹的继承者（167）、伊甸园（10001）、带宽（140）、图着色（193）、救火车（208）。

下面的题目还是可以回溯，但难度有所提高：不要歪（10123）、服务站（10160）、拼立方体（197）、罗马数字（185）、木棍（307）、六边形（317）、谜题（387）、谜题

II（519）、数字链（529）、大众匹萨（565）、DEL 命令（502）、船（322）。

接下来是一些隐式图搜索问题：倒水（10603）、FEN 中的骑士（10422）、最远的状态（10085）、L 系统（310）、新别墅（321）、彩色转盘（704）。

下面是一些和哈希表有关或能用哈希表与 STL 集合发挥作用的题目：完美哈希（188）、字典（10282）、复合词（10391）、子集和数（10125）、语言的连接（10887）、斑点游戏（141）、快乐的数（10591）。

笔者从初学时就喜欢简单、有效的暴力法，因此所命题目中出现过很多可以用暴力法解决的题目，甚至还曾在 UVa 上举办了两场暴力题目比赛（题号 11195～11199 和 11208～11218）。其中，下列题目初学者可以一试：KTV（11218）、跳舞的数字（11198）、风扇和宝石（10274）、中国麻将（11210）、超级数（10624）。

第 8 章　高效算法设计

学习目标

- ☑ 理解“基本操作”、渐进时间复杂度的概念和大 O 记号的含义
- ☑ 掌握“最大连续和”问题的各种算法及其时间复杂度分析
- ☑ 正确认识算法分析的优点和局限性，能正确使用分析结果
- ☑ 掌握归并排序和逆序对统计的分治算法
- ☑ 理解快速排序和快速选择算法
- ☑ 熟练掌握二分查找算法，包括找上下界的算法
- ☑ 能用递归的方式思考和求解问题
- ☑ 熟练掌握用二分法求解非线性方程的方法
- ☑ 熟练掌握用二分法把优化问题转化为判定问题的方法
- ☑ 熟悉能用贪心法求解的各类经典问题

尽管直观、适用范围广，但枚举、回溯等暴力方法常常无法走出“低效”的阴影。这并不难理解：越是通用的算法，越不能深入挖掘问题的特殊性。本章介绍一些经典问题的高效算法。由于是“量身定制”的，这些算法从概念、思路到程序实现都是千差万别的。从某种意义上说，从本章开始，读者才刚刚开始接触“严肃”的算法设计理论。

8.1　算法分析初步

我们都希望自己的算法高效，但算法在写成程序之前是运行不了的。难道每设计出来一个算法都必须写出程序来才知道快不快吗？答案是否定的。本节介绍算法分析的基本概念和方法，力求在编程之前尽量准确地估计程序的时空开销，并作出决策——例如，如果算法又复杂速度又慢，就不要急着写出来了。

8.1.1　渐进时间复杂度

例题 8-1　最大连续和

给出一个长度为 n 的序列 $A_1, A_2, \cdots, A_n$，求最大连续和。换句话说，要求找到 $1 \leqslant i \leqslant j \leqslant n$，使得 $A_i+A_{i+1}+\cdots+A_j$ 尽量大。

【分析】

使用第 7 章的枚举思想，得出如下程序：

程序 8-1　最大连续和（1）

```
tot = 0;
best = A[1]; // 初始最大值
```

```
for(i = 1; i <= n; i++)
  for(j = i; j <= n; j++)          // 检查连续子序列 A[i],…, A[j]
  {
    int sum = 0;
    for(k = i; k <= j; k++) { sum += A[k]; tot++; } // 累加元素和
    if(sum > best) best = sum;     // 更新最大值
  }
```

注意 best 的初值是 A[1]，这是最保险的做法——不要写 best=0（想一想，为什么）。当 n=1000 时，输出 tot=167167000，这是加法运算的次数。当 n=50 时，输出 22100。

为什么要计算 tot 呢？因为它与机器的运行速度无关。不同机器的速度不一样，运行时间也会有所差异，但 tot 值一定相同。换句话说，它去掉了机器相关的因素，只衡量算法的“工作量”大小——具体来说，是“加法”操作的次数。

提示 8-1：统计程序中“基本操作”的数量，可以排除机器速度的影响，衡量算法本身的优劣程度。

在本题中，我们把“加法操作”作为基本操作，类似地也可以把其他四则运算、比较运算作为基本操作。一般并不会严格定义基本操作的类型，而是根据不同情况灵活处理。

刚才是实验得出 tot 值的，其实它也可以用数学方法直接推导出。设输入规模为 n 时加法操作的次数为 $T(n)$，则：

$$T(n)=\sum_{i=1}^{n}\sum_{j=i}^{n} j-i+1=\sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2}=\frac{n(n+1)(n+2)}{6}$$

上面的公式是关于 n 的三次多项式，它意味着当 n 很大时，平方项和一次项对整个多项式值的影响不大。我们可以用一个记号来表示：$T(n)=\Theta(n^3)$，或者说：$T(n)$和 n^3 同阶。

同阶是什么意思呢？简单地说，就是“增长情况相同”。前面说过，n 很大时，只有立方项起到决定因素，而立方项的系数对“增长”是不起作用的——n 扩大两倍时，n^3 和 $100n^3$ 都扩大 8 倍。这样一来，我们可以只保留“最大项”，并忽略它的系数，得到的简单式子称为算法的渐进时间复杂度（asympotic time complexity）。

提示 8-2：基本操作的数量往往可以写成关于“输入规模”的表达式，保留最大项并忽略系数后的简单表达式称为算法的渐进时间复杂度，它用于衡量算法中基本操作数随规模的增长情况。

读者可以做个实验，看看 n 扩大两倍时运行时间是否近似扩大 8 倍。注意这里的“8 倍”是近似的，因为在 $T(n)$的表达式中，二次项、一次项和常数项都被忽略掉了；程序中的其他运算，如 if(sum > best)中的比较运算，甚至改变循环变量所需的“自增”都没有考虑在内。尽管如此，算法分析的效果还是比较精确的，因为我们抓住了主要矛盾——执行得最多的运算是加法。

提示 8-3：渐进时间复杂度忽略了很多因素，因而分析结果只能作为参考，并不是精确的。尽管如此，如果成功抓住了最主要的运算量所在，算法分析的结果常常十分有用。

8.1.2 上界分析

对于上面的方法，读者可能会有疑问：难道每次都要作一番复杂的数学推导才能得到渐进时间复杂度吗？当然不必。

下面是另外一种推导方法：算法包含 3 重循环，内层最坏情况下需要循环 n 次，中层循环最坏情况下也需要 n 次，外层循环最坏情况下仍然需要 n 次，因此总运算次数不超过 n^3。这里我们采用了“上界分析”，假定所有最坏情况同时取到，尽管这是不可能的。不难预料，这样的分析和实际情况肯定会有一定偏差——在 $T(n)$的表达式中，n^3 的系数是 1/6，小于 n^3，但数量级是正确的——仍然可以得到“n 扩大两倍时，运行时间近似扩大 8 倍”的结论。上界也有记号：$T(n)=O(n^3)$。

提示 8-4：在算法设计中，常常不进行精确分析，而是假定各种最坏情况同时取到，得到上界。在很多情况下，这个上界和实际情况同阶（称为“紧”的上界），但也有可能会因为分析方法不够好，得到“松”的上界。

松的上界也是正确的上界，但可能让人过高估计程序运行的实际时间（从而不敢编写程序），而即使上界是紧的，过大（如 100）或过小（如 1/100）的最高项系数同样可能引起错误的估计。换句话说，算法分析不是万能，要谨慎对待分析结果。如果预感到上界不紧、系数过大或者过小，最好还是要编程实践。

下面试着优化一下这个算法。设 $S_i=A_1+A_2+\cdots+A_i$，则 $A_i+A_{i+1}+\cdots+A_j=S_j-S_{i-1}$。这个式子的用途相当广泛，其直观含义是“连续子序列之和等于两个前缀和之差”。有了这个结论，最内层的循环就可以省略了。

程序 8-2　最大连续和（2）

```
S[0] = 0;
for(i = 1; i <= n; i++) S[i] = S[i-1] + A[i];          // 递推前缀和 S
  for(i = 1; i <= n; i++)
    for(j = i; j <= n; j++) best >?= S[j] - S[i-1];    // 更新最大值
```

注意上面的程序用到了递推的思想：从小到大依次计算 S[1], S[2], S[3],…，每个只需要在前一个的基础上加上一个元素。换句话说，“计算 S”这个步骤的时间复杂度为 $O(n)$。接下来是一个二重循环，用类似的方法可以分析出：

$$T(n)=\sum_{i=1}^{n} n-i+1=\frac{n(n+1)}{2}$$

代入可得 $T(1000)=500500$，和运行结果一致。同样地，用上界分析可以更快地得到结论：内层循环最坏情况下要执行 n 次，外层也是，因此时间复杂度为 $O(n^2)$。

8.1.3 分治法

本节使用分治法来解决这个问题。分治算法一般分为如下 3 个步骤。

划分问题：把问题的实例划分成子问题。

递归求解：递归解决子问题。

合并问题：合并子问题的解得到原问题的解。

在本例中，“划分”就是把序列分成元素个数尽量相等的两半；“递归求解”就是分别求出完全位于左半或者完全位于右半的最佳序列；“合并”就是求出起点位于左半、终点位于右半的最大连续和序列，并和子问题的最优解比较。

前两部分没有什么特别之处，关键在于“合并”步骤。既然起点位于左半，终点位于右半，我们可以人为地把这样的序列分成两部分，然后独立求解：先寻找最佳起点，然后再寻找最佳终点。

程序 8-3　最大连续和（3）（如图 8-1 所示）

```
int maxsum(int* A, int x, int y) // 返回数组在左闭右开区间[x,y)中的最大连续和
{
  int i, m, v, L, R, max;
  if(y - x == 1) return A[x];                    // 只有一个元素，直接返回
  m = x + (y-x)/2;        // 分治第一步：划分成[x, m)和[m, y)
  max = maxsum(A, x, m) >? maxsum(A, m, y);// 分治第二步：递归求解
  v = 0; L = A[m-1];      // 分治第三步：合并(1)——从分界点开始往左的最大连续和 L
  for(i = m-1; i >= x; i--) L >?= v += A[i];
  v = 0; R = A[m];        // 分治第三步：合并(2)——从分界点开始往右的最大连续和 R
  for(i = m; i < y; i++) R >?= v += A[i];
  return max >? (L+R);  // 把子问题的解与 L 和 R 比较
}
```

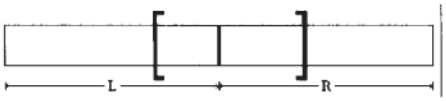


图 8-1　最大连续和的分治算法

除了使用了“取最大值”运算符外（注意，该运算符也可以和赋值运算“组合”），上面的代码还用到了“赋值运算本身具有返回值”的特点，这些都在一定程度上简化了代码，但不会牺牲可读性。

在上面的程序中，L 和 R 分别为从分界线往左、往右能达到的最大连续和。对于 n=1000，tot 值仅为 9976，在前面的 $O(n^2)$算法基础上又有大幅度改进。

是否可以像前面那样，得到 tot 的数学表达式呢？注意求和技巧已经不再适用，需要用递归的思路进行分析：设序列长度为 n 时的 tot 值为 $T(n)$，则 $T(n)=2T(n/2)+n$，$T(1)=1$。其中 $2T(n/2)$是两次长度为 $n/2$ 的递归调用，而最后的 n 是合并的时间（整个序列恰好扫描一遍）。注意这个方程是近似的，因为当 n 为奇数时两次递归的序列长度分别为$(n-1)/2$ 和$(n+1)/2$，而并不是 $n/2$。幸运的是，这样的近似对于最终结果影响很小，我们在分析算法时总是可以忽略它。

提示 8-5：*在算法分析中，往往可以忽略“除法结果是否为整数”，而直接按照实数除法分析。这样的近似对最终结果影响很小，一般不会改变渐进时间复杂度。*

解刚才的方程，可以得到$T(n)=\Theta(n\log n)$。由于 $n\log n$ 增长很慢，当 n 扩大两倍时，运行时间的扩大倍数只是略大于 2。现在不必懂得解方程的方法，可以把它作为一个重要结论记下来（建议有兴趣的读者试着借助于解答树来证明这个结论，它并不复杂）。

提示 8-6： 递归方程 $T(n)=2T(n/2)+\Theta(n)$，$T(1)=1$ 的解为 $T(n)=\Theta(n\log n)$。

在结束对分治算法的讨论之前，有必要再谈谈上述程序中的两个细节。首先是范围表示。上面的程序用左闭右开区间来表示一个范围，好处是在处理“数组分割”时比较自然：区间$[x, y)$被分成的是$[x,m)$和$[m,y)$，不需要在任何地方加减 1。另外，空区间表示为$[x,x)$，比$[x,x-1]$顺眼多了。

另一个细节是“分成元素个数尽量相等的两半”时分界点的计算。在数学上，分界点应当是 x 和 y 的平均数 $m=(x+y)/2$，我们却用的 $x+(y-x)/2$。在数学上二者相等，但在计算机中却有差别。不知读者是否注意到，运算符“/”的“取整”是朝零方向（towards zero）的取整，而不是向下取整。换句话说，5/2 的值是 2，而-5/2 的值是-2。为了方便分析，我们用 $x+(y-x)/2$ 来确保分界点总是靠近区间起点。这在本题中并不是必要的，但在后面要介绍的二分查找中，却是相当重要的技巧。

8.1.4 正确对待算法分析结果

对于“最大连续和”问题，本书先后介绍了时间复杂度为 $O(n^3)$、$O(n^2)$、$O(n\log n)$的算法，每个新算法较前一个来说，都是重大的改进。尽管分治法看上去很巧妙，它并不是最高效的。把 $O(n^2)$算法稍作修改，便可以得到一个 $O(n)$算法：当 j 确定时，“S[j]-S[i-1]最大”相当于“S[i-1]最小”，因此只需要扫描一次数组，维护“目前遇到过的最小 S”即可。

假设机器速度是每秒 10^8 次基本运算，运算量为 n^3、n^2、$n\log_2 n$、n、2^n（如子集枚举）和 $n!$（如排列枚举）的算法，在 1 秒之内能解决最大问题规模 n，如表 8-1 所示。

表 8-1 运算量随着规模的变化

运算量	$n!$	2^n	n^3	n^2	$n\log_2 n$	n
最大规模	11	26	464	10000	$4.5*10^6$	100000000
速度扩大两倍后	11	27	584	14142	$8.6*10^6$	200000000

表 8-1 还给出了机器速度扩大两倍后，算法所能解决规模的对比。可以看出，$n!$和 $2n$ 不仅能解决的问题规模非常小，而且增长缓慢；最快的 $n\log_2 n$ 和 n 算法不仅解决问题的规模大，而且增长快。我们把渐进时间复杂为多项式的算法称为多项式时间算法（polymonial-time algorithm），也称有效算法；而 $n!$或者 2^n 这样的低效的算法称为指数时间算法（exponential-time algorithm）。

不过需要注意的是，上界分析的结果在趋势上能反映算法的效率，但有两个不精确性：一是公式本身的不精确性。例如“非主流”基本操作的影响、隐藏在大 O 记号后的低次项和最高项系数；二是对程序实现细节与计算机硬件的依赖性，例如对复杂表达式的优化计算、把内存访问方式设计得更加“cache 友好”等。在不少情况下，算法实际能解决的问题规模与表 8-1 所示有着较大差异。

尽管如此，表 8-1 还是有一定借鉴意义的。考虑到目前主流机器的执行速度，多数算法竞赛题目所选取的数据规模基本符合此表。例如，一个指明 $n \leqslant 8$ 的题目，可能 $n!$ 的算法已经足够，$n \leqslant 20$ 的题目需要用到 2^n 的算法，而 $n \leqslant 300$ 的题目可能就必须用至少 n^3 的多项式时间算法了。

8.2　再谈排序与检索

假设你有 n 个整数，希望把它们按照从小到大的顺序排列，应该怎样做呢？也许你会说：调用 C 标准库中的 qsort 或者 C++ STL 中的 sort 或者 stable_sort 即可。可是你有没有想过：这些现成的排序函数是怎样工作的呢？

8.2.1　归并排序

第一种高效排序算法是归并排序。按照分治三步法，对归并排序算法介绍如下。

划分问题：把序列分成元素个数尽量相等的两半。

递归求解：把两半元素分别排序。

合并问题：把两个有序表合并成一个。

前两部分是很容易完成的，关键在于如何把两个有序表合成一个。图 8-2 演示了一个合并的过程。每次只需要把两个序列的最小元素加以比较，删除其中的较小元素并加入合并后的新表即可。由于需要一个新表来存放结果，所以附加空间为 n。

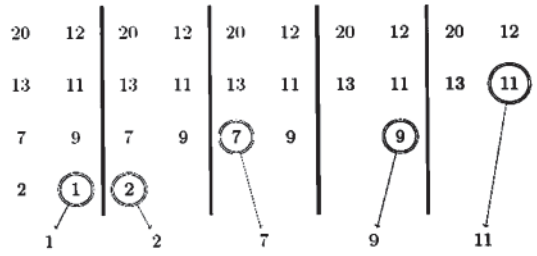


图 8-2　合并过程：时间是线性的，需要线性的辅助空间

这个过程极为重要，希望读者仔细体会。代码如下：

程序 8-4　归并排序（从小到大）

```
void merge_sort(int* A, int x, int y, int* T)
{
  if(y-x > 1)
  {
    int m = x + (y-x)/2;          // 划分
    int p = x, q = m, i = x;
    merge_sort(A, x, m, T);       // 递归求解
```

```
    merge_sort(A, m, y, T);                    // 递归求解
    while(p < m || q < y)
    {
      if(q >= y || (p < m && A[p] <= A[q])) T[i++] = A[p++];
                                               // 从左半数组复制到临时空间
      else T[i++] = A[q++];                    // 从右半数组复制到临时空间
    }
    for(i = x; i < y; i++) A[i] = T[i];        // 从辅助空间复制回 A 数组
  }
}
```

代码中的两个条件是关键。首先，只要有一个序列非空，就要继续合并（while(p<m||q<y)），因此在比较时不能直接比较 A[p] 和 A[q]，因为可能其中一个序列为空，从而 A[p]或者 A[q]代表的是一个实际不存在的元素。正确的方式是：

- 如果第二个序列为空（此时第一个序列一定非空），复制 A[p]。
- 否则（第二个序列非空），当且仅当第一个序列也非空，且 A[p]≤A[q]时，才复制 A[p]。

上面的代码巧妙地利用短路运算符“||”把两个条件连接在了一起：如果条件 1 满足，就不会计算条件 2；如果条件 1 不满足，就一定会计算条件 2。这样的技巧很实用，请读者细心体会。另外，读者如果仍然不太习惯 T[i++]=A[p++]这种“复制后移动下标”的方式，是时候把它们弄懂、弄熟了。

不难看出，归并排序的时间复杂度和最大连续和的分治算法一样，都是 $O(n\log n)$的。

例题 8-2　逆序对数

给一列数 $a_1, a_2, \cdots, a_n$，求它的逆序对数，即有多少个有序对(i,j)，使得 $i<j$ 但 $a_i>a_j$。n 可以高达 10^6。

【分析】

n 这么大，$O(n^2)$的枚举将超时，因此需要寻找更高效的方法。受到归并排序的启发，我们来试试“分治三步法”是否适用。“划分问题”过程是把序列分成元素个数尽量相等的两半；“递归求解”是统计 i 和 j 均在左边或者均在右边的逆序对个数；“合并问题”则是统计 i 在左边，但 j 在右边的逆序对个数。

和归并排序一样，划分和递归求解都好办，关键在于合并：如何求出 i 在左边，而 j 在右边的逆序对数目呢？统计的常见技巧是“分类”。我们按照 j 的不同把这些“跨越两边”的逆序对进行分类：只要对于右边的每个 j，统计左边比它大的元素个数 $f(j)$，则所有 $f(j)$之和便是答案。

幸运的是，归并排序可以帮我们“顺便”完成 $f(j)$的计算：由于合并操作是从小到大进行的，当右边的 A[j]复制到 T 中时，左边还没来得及复制到 T 的那些数就是左边**所有**比 A[j]大的数。此时在累加器中加上左边元素个数 $m-p$ 即可（左边所剩的元素在区间$[p,m)$中，因此元素个数为 $m-p$）。换句话说，在代码上的唯一修改就是把“else T[i++] = A[q++]; ”改成“else { T[i++] = A[q++]; cnt += m-p; }”。当然，别忘了在调用之前给 cnt 清零。

8.2.2　快速排序

既然敢叫“快速排序”，必然有其过人之处。事实上，它确实是最快的通用内部排序算法。它由 Hoare 于 1962 年提出，相对归并排序来说不仅速度更快，并且不需辅助空间（还记得那个 T 数组吗）。按照分治三步法，将快速排序算法作如下介绍。

划分问题：把数组的各个元素重排后分成左右两部分，使得左边的任意元素都小于或等于右边的任意元素。

递归求解：把左右两部分分别排序。

合并问题：不用合并，因为此时数组已经完全有序。

你也许会觉得这样的描述太过笼统，但事实上，快速排序本来就不是只有一种实现方法。“划分过程”有多个不同的版本，导致快速排序也有不同版本。读者很容易在互联网上找到各种快速排序的版本，这里不再给出代码。

例题 8-3　第 k 小的数

输入 n 个整数和一个正整数 k($1 \leqslant k \leqslant n$)，输出这些整数从小到大排序后的第 k 个（例如，k=1 就是最小值）。$n \leqslant 10^7$。

【分析】

选第 k 大的数，最显然的方法是先排序，然后直接输出下标为 k-1 的元素（别忘了 C 语言中数组下标从 0 开始），但 10^7 的规模即使对于 $O(n\log n)$的算法来说也稍微大了点。有没有更快的方法呢？

答案是肯定的。假设在快速排序的“划分”结束后，数组 A[p...r]被分成了 A[p...q]和 A[q+1...r]，则可以根据左边的元素个数 $q-p+1$ 和 k 的大小关系**只**在左边或者右边递归求解。可以证明，在期望意义下，程序的时间复杂度为 $O(n)$。

8.2.3　二分查找

排序的重要意义之一，就是为检索带来方便。试想有 10^6 个整数，你希望确认其中是否包含 12345。最容易想到的方法就是把它们放到数组 A 中，然后依次检查这些整数是否等于 12345（事实上，这正是第 5 章中介绍的方法）。这样的方式对于“单次询问”来说工作得很好，但如果需要找 10000 个数，就需要把整个数组 A 遍历 10000 次。而如果先将数组 A 排序，就可以查找得更快——就好比在字典中查找单词不必一页一页翻一样。

在有序表中查找元素常常使用二分查找（Binary Search），有时也译为“折半查找”，它的基本思路就像是“猜数字游戏”：你在心里想一个不超过 1000 的正整数，我可以保证在 10 次之内猜到它——只要你每次告诉我猜的数比你想的大一些、小一些，或者正好猜中。

猜的方法就是“二分”。首先我猜 500，除了运气特别好正好猜中之外[①]，不管你说“太大”还是“太小”，我都能把可行范围缩小一半：如果“太大”，那么答案在 1～499 之间；如果“太小”，那么答案在 501～1000 之间。只要每次选择可行区间的中点去猜，每次都

[①] 如果没有公证人，你可以不动声色地换一个数。

可以把范围缩小一半。由于 $\log_2 1000<10$，10 次一定能猜到。

这也是二分查找的基本思路。

提示 8-7：逐步缩小范围法是一种常见的思维方法。二分查找便是基于这种思路，它遵循分治三步法，把原序列划分成元素个数尽量接近的两个子序列，然后递归查找。二分查找只适用于有序序列。

尽管可以用递归实现，人们一般把二分查找写成非递归的：

程序 8-5　二分查找（迭代实现）

```
int bsearch(int* A, int x, int y, int v)
{
  int m;
  while(x < y)
  {
    m = x+(y-x)/2;
    if(A[m] == v) return m;
    else if(A[m] > v) y = m;
    else x = m+1;
  }
  return -1;
}
```

上述 while 循环常常直接写在程序中。二分查找常常用在一些抽象的场合，没有数组 A，也没有要查找的 v，但是二分的思想仍然适用。

提示 8-8：二分查找一般写成非递归形式。

下面提一个有趣的问题：如果数组中有多个元素都是 v，上面的函数返回的是哪一个的下标呢？第一个？最后一个？都不是。不难看出，如果所有元素都是我们要找的，它返回的是中间那一个。有时，这样的结果并不是很理想——能不能求出值等于 v 的完整区间呢（由于已经排好序，相等的值会排在一起）？

下面的程序当 v 存在时返回它出现的第一个位置。如果不存在，返回这样一个下标 i：在此处插入 v（原来的元素 A[i], A[i+1],…全部往后移动一个位置）后序列仍然有序。

程序 8-6　二分查找求下界

```
int lower_bound(int* A, int x, int y, int v)
{
  int m;
  while(x < y)
  {
    m = x+(y-x)/2;
    if(A[m]>=v) y=m;
    else x=m+1;
```

```
  }
  return x;
}
```

下面来分析一下这段程序。首先，最后的返回值不仅可能是 $x, x+1, x+2, \cdots\cdots, y-1$，还可能是 y——如果 v 大于 A[y−1]，就只能插入这里了。这样，尽管查找区间是左闭右开区间$[x,y)$，返回值的候选区间却是闭区间$[x,y]$。A[m]和 v 的各种关系所带来的影响如下。

A[m] = v：至少已经找到一个，而左边可能还有，因此区间变为$[x,m]$。

A[m] > v：所求位置不可能在后面，但有可能是 m，因此区间变为$[x,m]$。

A[m] < v：m 和前面都不可行，因此区间变为$[m+1,y]$。

合并一下：A[m]≥v 时新区间为$[x,m]$；A[m]<v 时新区间为$[m+1,y]$。这里有一个潜在的危险：如果$[x,m]$或者$[m+1,y]$和原区间$[x,y]$相同，将发生死循环！幸运的是，这样的情况并不会发生，原因留给读者思考。

类似地，可以写一个 upper_bound 程序，当 v 存在时返回它出现的最后一个位置的后面一个位置。如果不存在，返回这样一个下标 i：在此处插入 v（原来的元素 A[i], A[i+1],…全部往后移动一个位置）后序列仍然有序。不难得出，只需把“if(A[m]>=v) y=m; else x=m+1;”改成“if(A[m]<=v) x=m+1; else y=m;”即可。

这样，对二分查找的讨论就相对比较完整了：设 lower_bound 和 upper_bound 的返回值分别为 L 和 R，则 v 出现的子序列为$[L,R)$。这个结论当 v 不在时也成立：此时 $L=R$，区间为空。

例题 8-4　范围统计

给出 n 个整数 x_i 和 m 个询问，对于每个询问(a,b)，输出闭区间$[a,b]$内的整数 x_i 的个数。

【分析】

有了前面的经验，我们知道“把数据存在数组 A 里并排序”是一个很好的预处理方法。接下来思考两个问题。

问题 1：大于等于 a 的第一个元素的下标 L 是什么？它等于 a 的 lower_bound 值。如果所有元素都小于 a，则 L=n，相当于把不存在的元素看作无穷大。

问题 2：小于等于 b 的最后一个元素的“下一个下标”R 是什么？它等于 b 的 upper_bound 值。如果所有元素都大于 b，则相当 R=0，相当于想象 A[0]前面还有一个 A[−1]等于负无穷，则这个 A[−1]的“下一个位置”就是 0。

这样，问题的答案就是区间$[L,R]$的长度，即 $R-L$。顺便说一句，STL 中已经包含了 lower_bound 和 upper_bound，可以直接使用。

```
#include<cstdio>
#include<algorithm>// STL 算法的头文件，包含 sort, lower_bound 和 upper_bound 等
using namespace std;
int v[10000];
int main()
{
  int n, m, a, b;
```

```
    scanf("%d%d", &n, &m);
    for(int i = 0; i < n; i++) scanf("%d", &v[i]);
    sort(v, v+n);                    // 从小到大排序
    for(int i = 0; i < m; i++)
    {
      scanf("%d%d", &a, &b);         // 询问[a,b]内的整数个数
      printf("%d\n", upper_bound(v, v+n, b)-lower_bound(v, v+n, a));
    }
}
```

提示 8-9：用“上下界”函数求解范围统计问题的技巧非常有用，建议读者用心体会左闭右开区间的使用方法和上下界函数的实现细节。

8.3 递归与分治

除了排序与检索外，递归还有更广泛的应用。

8.3.1 棋盘覆盖问题

有一个 2^k*2^k 的方格棋盘，恰有一个方格是黑色的，其他为白色。你的任务是用包含 3 个方格的 L 型牌覆盖所有白色方格。黑色方格不能被覆盖，且任意一个白色方格不能同时被两个或更多牌覆盖。如图 8-3 所示为 L 型牌的 4 种旋转方式。

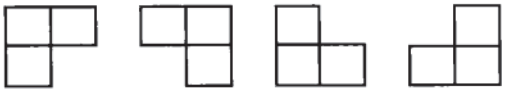


图 8-3 L 型牌

【分析】

本题的棋盘是 2^k*2^k 的，很容易想到分治：把棋盘切为 4 块，则每一块都是 $2^{k-1}*2^{k-1}$ 的。有黑格的那一块可以递归解决，但其他 3 块并没有黑格子，应该怎么办呢？可以构造出一个黑格子，如图 8-4 所示。递归边界也不难得出：k=1 时一块牌就够了。

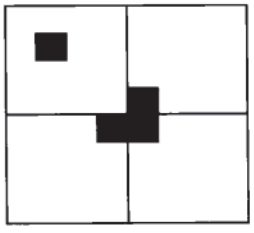


图 8-4 棋盘覆盖问题的递归解法

8.3.2　循环日程表问题

有 $n=2^k$ 个运动员进行网球循环赛，需要设计比赛日程表。每个选手必须与其他 $n-1$ 个选手各赛一次；每个选手一天只能赛一次；循环赛一共进行 $n-1$ 天。按此要求设计一张比赛日程表，它有 n 行和 $n-1$ 列，第 i 行 j 列为第 i 个选手第 j 天遇到的选手。

【分析】

本题的方法有很多，递归是其中一种比较容易理解的方法。图 8-5 所示是 $k=3$ 时的一个可行解，它是 4 块拼起来的。左上角是 $k=2$ 时的一组解，左下角是左上角每个数加 4 得到，而右上角、右下角分别由左下角、左上角复制得到。

1	2	3	4	5	6	7	8
2	1	4	3	6	5	8	7
3	4	1	2	7	8	5	6
4	3	2	1	8	7	6	5
5	6	7	8	1	2	3	4
6	5	8	7	2	1	4	3
7	8	5	6	3	4	1	2
8	7	6	5	4	3	2	1

图 8-5　循环日程表问题 $k=3$ 时的解

8.3.3　巨人与鬼

在平面上有 n 个巨人和 n 个鬼，没有三者在同一条直线上。每个巨人需要选择一个不同的鬼，向其发送质子流消灭它。质子流由巨人发射，沿直线行进，遇到鬼后消失。由于质子流交叉是很危险的，所有质子流经过的线段不能有交点。请设计一种给巨人和鬼配对的方法。

【分析】

由于只需要一种配对方法，从直观上来说本题一定是有解的。由于每一个巨人和鬼都需要找一个目标，我们不妨先给“最特殊”的巨人或鬼寻找“搭档”。

考虑 y 坐标最小的点（即最低点）。如果有多个这样的点，考虑最左边的点（即其中最左边的点），则所有点的极角在范围$[0,\pi)$内。不妨设它是一个巨人，然后把所有其他点按照极角从小到大的顺序排序后依次检查。

情况 1：第一个点是鬼，那么配对完成，剩下的巨人和鬼仍然是一样多，而且肯定不会和这一条线段交叉，如图 8-6（a）所示。

情况 2：第一个点是巨人，那么继续检查，直到已检查的点中鬼和巨人一样多为止。找到了这个“鬼和巨人”配对区间后，只需要把此区间内的点配对，再把区域外的点配对即可，如图 8-6（b）所示。这个配对过程是递归的，好比棋盘覆盖中一样。会不会找不到这样的配对区间呢？不会的。因为检查完第一个点后鬼少一个，而检查完最后一个点时鬼多一个，而巨人和鬼的数量差每次只能改变 1，因此“从少到多”的过程中一定会有“一样多”的时候。

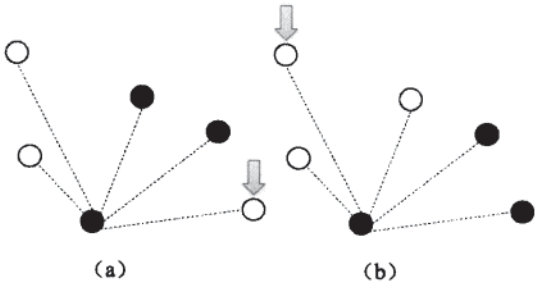


图 8-6 巨人与鬼问题

8.3.4 非线性方程求根

一次向银行借 a 元钱，分 b 月还清。如果需要每月还 c 元，月利率是多少（按复利计算）？例如借 2000 元，分 4 个月共还 510 元，则月利率为 0.797%。答案应不超过 100%。

【分析】

设月利率为 x，则第一个月还钱后还需要还 $a(1+x)-c$ 元。重复 b 个月后可以得到一个方程，解出 x 即可。例如 a=2000, b=4, c=510 时，方程为：

$$f(x)=(((2000(1+x)-c)(1+x)-c)(1+x)-c)(1+x)-c=0$$

这个方程不仅解着麻烦，甚至列出来也不简单。不妨换一个思路：猜数字！现在已知 x 在范围[0,100]内，如果每次猜一个值，又有人能告诉我大了还是小了，问题便迎刃而解。

如何知道“大了还是小了”呢？我们把 x 代入方程的左边，看结果是多少。如果等于 0，则猜中；如果小于 0，则 x 太小；如果大于 0，则 x 太大。

为什么呢？因为 $f(x)$在[0,100]内关于 x 单调递增，因此 $f(x)$和 0 的大小关系与 x 和方程的解 x_0 的大小关系等价。

需要注意的是，这里的 x 是实数，而实数区间是可以无限二分的。这样一来，算法就无法终止了？理论上的确是这样，但实际上，如果 x 的区间被缩小到了[1.234,1.234001]之间，而输出时又是保留三位小数，则不需要继续二分——输出一定是 1.234。

程序 8-7 贷款

```
#include<stdio.h>
int main()
{
  double a, c, x = 0, y = 100;
  int i, b;
  scanf("%lf%d%lf", &a, &b, &c);
  while(y-x > 1e-5)
  {
    double m = x+(y-x)/2;
    double f = a;
    for(i = 0; i < b; i++) f += f*m/100.0-c;
```

```
    if(f < 0) x=m;
    else y=m;
  }
  printf("%.3lf%%\n", x);
  return 0;
}
```

在上述程序中，引用部分就是在计算 $f(m)$的值——不需要把方程列出来，只需要代入即可。

8.3.5　最大值最小化

把一个包含 n 个正整数的序列划分成 m 个连续的子序列（每个正整数恰好属于一个序列）。设第 i 个序列的各数之和为 $S(i)$，你的任务是让所有 $S(i)$的最大值尽量小。例如序列 1 2 3 2 5 4 划分成 3 个序列的最优方案为 1 2 3 | 2 5 | 4，其中 $S(1)$、$S(2)$、$S(3)$分别为 6、7、4，最大值为 7；如果划分成 1 2 | 3 2 | 5 4，则最大值为 9，不如刚才的好。$n \leqslant 10^6$，所有数之和不超过 10^9。

【分析】

“最大值尽量小”是一种很常见的优化目标。我们考虑一个新的问题：**能否**把输入序列划分成 m 个连续的子序列，使得所有 $S(i)$均不超过 x？我们把这个问题的答案用谓词 $P(x)$表示，则让 $P(x)$为真的最小 x 就是原题的答案。$P(x)$并不难计算：每次尽量往右划分即可（想一想，为什么）。

接下来又可以猜数字了——随便猜一个 x_0，如果 $P(x_0)$为假，那么答案比 x_0 大；如果 $P(x_0)$为真，则答案小于或等于 x_0。至此，解法已经水落石出：二分最小值 x，把优化问题转化为判定问题 $P(x)$。设所有数之和为 M，则二分次数为 $O(\log M)$，计算 $P(x)$的时间复杂度为 $O(n)$（从左到右扫描一次即可），因此总时间复杂度为 $O(n\log M)$。

8.4　贪　心　法

贪心是一种解决问题的策略。如果策略正确，那么贪心法往往是易于描述、易于实现的。本节介绍可以用贪心法解决的若干经典问题。

8.4.1　最优装载问题

给出 n 个物体，第 i 个物体重量为 w_i。选择尽量多的物体，使得总重量不超过 C。

【分析】

由于只关心物体的数量，所以装重的没有装轻的划算。只需把所有物体按重量从小到大排序，依次选择每个物体，直到装不下为止。这是一种典型的贪心算法，它只顾眼前，但却能得到最优解。

8.4.2 部分背包问题

有 n 个物体，第 i 个物体的重量为 w_i，价值为 v_i。在总重量不超过 C 的情况下让总价值尽量高。每一个物体都可以只取走一部分，价值和重量按比例计算。

【分析】

本题在上一题的基础上增加了价值，所以不能简单地像上题那样先拿轻的（轻的可能价值也小），也不能先拿价值大的（可能它特别重），而应该综合考虑两个因素。一种直观的贪心策略是：优先拿“价值除以重量的值”最大的，直到重量和正好为 C。

注意：由于每个物体可以只拿一部分，因此一定可以让总重量恰好为 C（或者全部拿走重量也不足 C），而且除了最后一个以外，所有的物体要么不拿，要么拿走全部。

8.4.3 乘船问题

有 n 个人，第 i 个人重量为 w_i。每艘船的最大载重量均为 C，且最多只能乘两个人。用最少的船装载所有人。

【分析】

考虑最轻的人 i，他应该和谁一起坐呢？如果每个人都无法和他一起坐船，则唯一的方案就是每人坐一艘船（想一想，为什么）。否则，他应该选择能和他一起坐船的人中最重的一个 j。这样的方法是贪心的，因此它只是让“眼前”的浪费最少。幸运的是，这个贪心策略也是对的，可以用反证法说明。

假设这样做不是最好的，那么最好方案中 i 是什么样的呢？

情况 1：i 不和任何一个人坐同一艘船，那么可以把 j 拉过来和他一起坐，总船数不会增加（而且可能会减少！）。

情况 2：i 和另外一人 k 同船。由贪心策略，k 比 j 轻。把 j 和 k 交换后 k 原来所在的船仍然不会超重（因为 j 比 k 轻），而 i 和 k 所在的船也不会超重（由贪心法过程），因此所得到的新解不会更差。

由此可见，贪心法不会丢失最优解。最后说一下程序实现。在刚才的分析中，比 j 更重的人只能每人坐一艘船。这样，我们只需用两个下标 i 和 j 分别表示当前考虑的最轻的人和最重的人，每次先将 j 往左移动，直到 i 和 j 可以共坐一艘船，然后 i 加 1，j 减 1，并重复上述操作。不难看出，程序的时间复杂度仅为 $O(n)$，是最优算法（别忘了，读入数据也需要 $O(n)$时间，因此无法比这个更好了）。

8.4.4 选择不相交区间

数轴上有 n 个开区间(a_i, b_i)。选择尽量多个区间，使得这些区间两两没有公共点。

【分析】

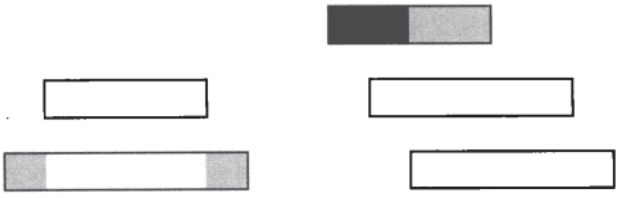
首先明确一个问题：假设有两个区间 x, y，区间 x 完全包含 y。那么，选 x 是不划算的，因为 x 和 y 最多只能选一个，选 x 还不如选 y，这样不仅区间数目不会减少，而且给其他区

间留出了更多的位置。接下来，按照 b_i 从小到大的顺序给区间排序。贪心策略是：一定要选第一个区间。为什么？

现在区间已经排序成 $b_1 \leqslant b_2 \leqslant b_3 \cdots$ 了，考虑 a_1 和 a_2 的大小关系。

情况 1：$a_1 > a_2$，如图 8-7（a）所示，区间 2 包含区间 1。前面已经讨论过，这种情况下一定不会选择区间 2。不仅区间 2 如此，以后所有区间中只要有一个 i 满足 $a_1 > a_i$，i 都不要选。在今后的讨论中，我们不考虑这些区间。

情况 2：排除了情况 1，一定有 $a_1 \leqslant a_2 \leqslant a_3 \leqslant \cdots$，如图 8-7（b）所示。如果区间 2 和区间 1 完全不相交，那么没有影响（因此一定要选区间 1），否则区间 1 和区间 2 最多只能选一个。注意到如果不选区间 2，黑色部分其实是没有任何影响的（它不会挡住任何一个区间），区间 1 的有效部分其实变成了灰色部分，它被区间 2 所包含！由刚才的结论，区间 2 是不能选的。依此类推，不能因为选任何区间而放弃区间 1，因此选择区间 1 是明智的。



（a）$a_1 > a_2$　　（b）$a_1 < a_2 < a_3$

图 8-7　贪心策略图示

选择了区间 1 以后，需要把所有和区间 1 相交的区间排除在外，需要记录上一个被选择的区间编号。这样，在排序后只需要扫描一次即可完成贪心过程，得到正确结果。

8.4.5　区间选点问题

数轴上有 n 个闭区间 $[a_i, b_i]$。取尽量少的点，使得每个区间内都至少有一个点（不同区间内含的点可以是同一个）。

【分析】

如果区间 i 内已经有一个点被取到，我们称此区间已经被满足。受上一题的启发，我们先讨论区间包含的情况。由于小区间被满足时大区间一定也被满足。所以在区间包含的情况下，大区间不需要考虑。

把所有区间按 b 从小到大排序（b 相同时 a 从大到小排序），则如果出现区间包含的情况，小区间一定排在前面。第一个区间应该取哪一个点呢？我们的贪心策略是：取最后一个点，如图 8-8 所示。

根据刚才的讨论，所有需要考虑的区间的 a 也是递增的，我们可以把它画成上图的形式。如果第一个区间不取最后一个，而是取中间的，如灰色点，那么把它移动到最后一个点后，被满足的区间增加了，而且原先被满足的区间现在一定被满足。不难看出，这样的贪心策略是正确的。

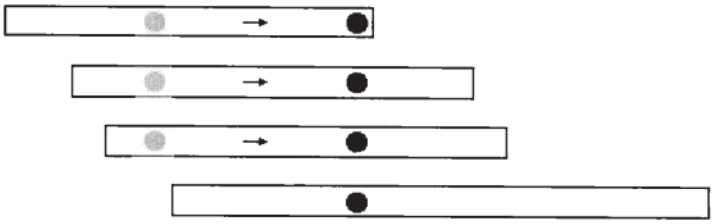


图 8-8　贪心策略

8.4.6　区间覆盖问题

数轴上有 n 个闭区间$[a_i, b_i]$，选择尽量少的区间覆盖一条指定线段 $[s, t]$。

【分析】

本题的突破口仍然是区间包含和排序扫描，不过先要进行一次预处理。每个区间在[s,t]外的部分都应该预先被切掉，因为它们的存在是毫无意义的。在预处理后，在相互包含的情况下，小区间显然不应该考虑。

把各区间按照 a 从小到大排序。如果区间 1 的起点不是 s，无解（因为其他区间的起点更大，不可能覆盖到 s 点），否则选择起点在 s 的最长区间。选择此区间$[a_i, b_i]$ 后，新的起点应该设置为 b_i，并且忽略所有区间在 b_i 之前的部分，就像预处理一样。虽然贪心策略比上题复杂，但是仍然只需要一次扫描，如图 8-9 所示。s 为当前有效起点（此前部分已被覆盖），则应该选择区间 2。

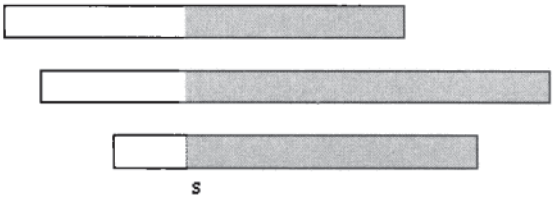


图 8-9　区间覆盖问题

8.4.7　Huffman 编码

假设某文件中只有 6 种字符：a, b, c, d, e, f，可以用 3 个二进制位来表示它们，如表 8-2 所示（以下 3 个表中，频率的单位均为“千次”）。

表 8-2　各种字符的编码

字符	a	b	c	d	e	f
频率	45	13	12	16	9	5
编码	000	001	010	011	100	101

这样，一共需要(45+13+12+16+9+5)*3=300 千比特（即二进制的位）。第二种方法是采用变长编码，如表 8-3 所示。

表 8-3　变长码举例

字符	a	b	c	d	e	f
频率	45	13	12	16	9	5
编码	0	101	100	111	1101	1100

总长度为 1*45+3*13+3*12+3*16+4*9+4*5=224 千比特，比定长码省。读者可能会说：还可以更省，如表 8-4 所示。

表 8-4　错误的变长码举例

字符	a	b	c	d	e	f
频率	45	13	12	16	9	5
编码	0	1	00	01	10	11

总长度只有 1*(45+13)+2*(12+16+9+5)=142 千比特，不是更省吗？可惜，这样的编码方案是有问题的。如果收到了 001，到底是 aab，还是 cb，还是 ad？换句话说，这样的编码有歧义，因为其中一个字符的编码是另一个码的前缀（prefix）。表 8-3 所示的码没有这样的情况，任一个编码都不是另一个的前缀。这里把满足这样性质的编码称为前缀码（Prefix Code）。下面正式叙述编码问题。

例题 8-5　编码问题

给出 n 个字符的频率 c_i，给每个字符赋予一个 01 编码串，使得任一个字符的编码不是另一个字符编码的前缀，而且编码后总长度（每个字符的频率与编码长度乘积的总和）尽量小。

【分析】

在解决这个问题之前，首先来看一个结论：任何一个前缀编码都可以表示成每个非叶结点恰好有两个儿子的二叉树。如图 8-10 所示，每个非叶结点与左儿子的边上写 1，与右儿子的边上写 0。

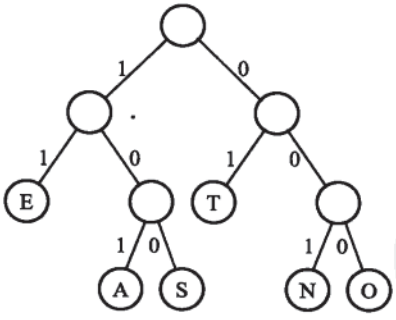


图 8-10　前缀码的二叉树表示

每个叶子对应一个字符，编码为从根到该叶子的路径上的 01 序列。在图 8-10 中，N 的编码为 001，而 E 的编码为 11。为了证明在一般情况下，都可以用这样的二叉树来表示最优前缀码，我们需要证明两个结论。

结论 1：n 个叶子的二叉树一定对应一个前缀码。如果编码 a 为编码 b 的前缀，则 a 所对应的结点一定为 b 所对应结点的祖先。而两个叶子不会有祖先后代的关系。

结论 2：最优前缀码一定可以写成二叉树。逐个字符构造即可。每拿到一个编码，都可以构造出从根到叶子的一条路径，沿着已有结点走，创建不存在的结点。这样得到的二叉树不可能有单儿子结点，因为如果存在，只要用这个儿子代替父亲，得到的仍然是前缀码，且总长度更短。

接下来的问题就变为：如何构造一棵最优的编码树。

Huffman 算法：把每个字符看作一个单结点子树放在一个树集合中，每棵子树的权值等于相应字符的频率。每次取权值最小的两棵子树合并成一棵新树，并重新放到集合中。新树的权值等于两棵子树权值之和。

下面分两步证明算法的正确性。

结论 1：设 x 和 y 是频率最小的两个字符，则存在前缀码使得 x 和 y 具有相同码长，且仅有最后一位编码不同。换句话说，第一步贪心选择保留最优解。

证明：假设深度最大的结点为 a，则 a 一定有一个兄弟 b。不妨设 $f(x) \leqslant f(y)$，$f(a) \leqslant f(b)$，则 $f(x) \leqslant f(a)$，$f(y) \leqslant f(b)$。如果 x 不是 a，把 x 和 a 交换；如果 y 不是 b，把 y 和 b 交换。这样得到的新编码树不会比原来的差。

结论 2：设 T 是加权字符集 C 的最优编码树，x 和 y 是树 T 中两个叶子，且互为兄弟，z 是它们的父亲。若把 z 看成具有频率 $f(z)=f(x)+f(z)$ 的字符，则树 $T'=T-\{x,y\}$ 是字符集 $C'=C-\{x,y\}\cup\{z\}$ 的一棵最优编码树。换句话说，原问题的最优解包含子问题的最优解。

证明：设 T' 的编码长度为 L，其中字符 $\{x,y\}$ 的深度为 h，则把字符 $\{x,y\}$ 拆成两个后，长度变为 $L-(f(x)+f(y))\cdot h+f(x)\cdot(h+1)=L+f(x)+f(y)$。因此 T' 必须是 C' 的最优编码树，T 才是 C 的最优编码树。

结论 1 通常称为贪心选择性质，结论 2 通常称为最优子结构性质。根据这两个结论，Huffman 算法正确。在程序实现上，可以先按照频率把所有字符排序成表 P，然后创建一个新结点队列 Q，在每次合并两个结点后把新结点放到队列 Q 中。由于后合并的频率和一定比先合并的频率和大，因此 Q 内的元素是有序的。类似有序表的合并过程，每次只需要检查 P 和 Q 的首元素即可找到频率最小的元素，时间复杂度为 $O(n)$。算上排序，总时间复杂度为 $O(n\log n)$。

8.5 训 练 参 考

和第 7 章一样，本章仍然给出 UVaOJ 中的题目。为了不给读者提供明显的提示，下面直接给出题目。

首先是 UVa 中的题目：孩子的游戏（10905）、出国交换（10763）、文件碎片（10132）、共线点（270）、解方程（10341）、仲夏夜之梦（10057）、数字序列（10706）、最接近的和（10487）、子序列判定（10340）、无括号的表达式（10700）、鞋匠的难题（10026）、打包（311）、最小覆盖（10020）、蚂蚁（10714）、全加一起（10954）、复制书稿（714）、

声控编辑器（10602）、货物装载 II（10400）、位掩码（10718）、葡萄酒交易（11054）、给草浇水（10382）、减轻工作（10670）、图重建（10720）、数字之积（993）、交换成回文数（10716）、旅行 2007（11100）、最近点对问题（10245）、反算术级数（11129）、Vito 的家庭（10041）、Jill 又骑车了（507）、最大和（108）、圆环上的最大和（10827）、钓鱼吧（757）、广告（10148）。

下面我们再介绍另外一个著名 OJ——从 UVa“分支”出来的“二十世纪实况题库”CII（http://acmicpc-live-archive.uva.es/nuevoportal），专门收录 2000 年以来 ACM/ICPC 的区域赛和总决赛题目，是一个半官方的 OJ。下面是其中一些题目：木棍（2322）、移动桌子（2326）、熵（2088）、安装雷达（2519）、组装电脑（3971）、派（3635）、实时磁盘调度（3349）、最大值（2911）、你在我身边……（4062）、电梯调度（2949）。

第3部分 竞 赛 篇

第9章 动态规划初步

学习目标

- ☑ 理解状态和状态转移方程
- ☑ 理解最优子结构和重叠子问题
- ☑ 熟练运用递推法和记忆化搜索求解数字三角形问题
- ☑ 熟悉 DAG 上动态规划的常见思路
- ☑ 掌握记忆化搜索在实现方面的注意事项
- ☑ 掌握记忆化搜索和递推中输出方案的方法
- ☑ 掌握递推中滚动数组的使用方法
- ☑ 实现常见动态规划问题
- ☑ 掌握集合上动态规划的基本思路和方法
- ☑ 理解并灵活运用增加维度的方法

动态规划的理论性和实践性都比较强，一方面需要理解“状态”、“状态转移”、“最优子结构”、“重叠子问题”等概念，另一方面又需要根据题目的条件灵活设计算法。可以这样说，对动态规划的掌握情况在很大程度上能直接影响一个选手的分析和建模能力。

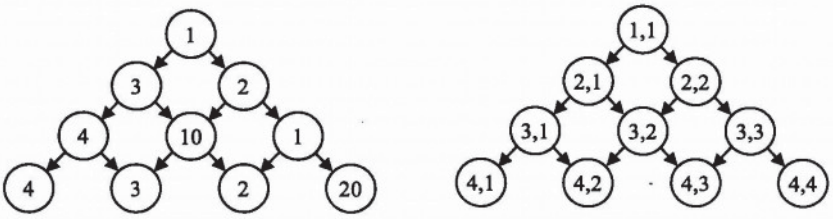
9.1 数字三角形

动态规划是一种用途很广的问题求解方法，它本身并不是一个特定的算法，而是一种思想，一种手段。下面通过一个例题阐述动态规划的基本思路和特点。

9.1.1 问题描述与状态定义

例题 9-1 数字三角形

有一个由非负整数组成的三角形，第一行只有一个数，除了最下行之外每个数的左下方和右下方各有一个数，如图 9-1 所示。



（a）数字三角形　　（b）格子编号

图 9-1　数字三角形问题

从第一行的数开始，每次可以往左下或右下走一格，直到走到最下行，把沿途经过的数全部加起来。如何走才能使得这个和尽量大？

【分析】

如果熟悉回溯法，你可能会立刻发现这是一个动态的决策问题：每次有两种选择——左下或右下。如果用回溯法求出所有可能的路线，就可以从中选出最优路线。但和往常一样，回溯法的效率太低：一个 n 层数字三角形的完整路线有 2^n 条，当 n 很大时回溯法的速度将让人无法忍受。

为了得到高效的算法，需要用抽象的方法思考问题：把当前的位置(i,j)看成一个状态（还记得吗？），然后定义状态(i,j)的指标函数 $d(i,j)$为从格子(i,j)出发时能得到的最大和（包括格子(i,j)本身的值）。在这个状态定义下，原问题的解是 $d(1,1)$。

下面看看不同状态之间是如何转移的。从格子(i,j)出发有两种决策。如果往左走，则走到$(i+1,j)$后需要求“从$(i+1,j)$出发后能得到的最大和”这一问题，即 $d(i+1,j)$。类似地，往右走之后需要求解 $d(i+1,j+1)$。由于可以在这两个决策中自由选择，所以应选择 $d(i+1,j)$和$d(i+1,j+1)$中较大的一个。换句话说，得到了所谓的状态转移方程：

$$d(i,j)=a(i,j)+\max\{d(i+1,j),d(i+1,j+1)\}$$

如果往左走，那么最好情况等于(i,j)格子里的值 $a(i,j)$与“从$(i+1,j)$出发的最大总和”之和，此时需注意这里的“最大”二字。如果连“从$(i+1,j)$出发走到底部”这部分的和都不是最大的，加上 $a(i,j)$之后肯定也不是最大的。这个性质称为最优子结构（optimal substructure），也可以描述成“全局最优解包含局部最优解”。不管怎样，状态和状态转移方程一起完整地描述了具体的算法。

提示 9-1：*动态规划的核心是状态和状态转移方程。*

9.1.2　记忆化搜索与递推

有了状态转移方程之后，应怎样计算呢？

方法 1：递归计算。程序如下（需注意边界处理）：

```
int d(int i, int j)
{
    return a[i][j] + (i == n ? 0 : d(i+1,j)>?d(i+1,j+1));
}
```

这样做是正确的，但时间效率太低，其原因在于重复计算。

图 9-2 所示为函数 $d(1, 1)$对应的调用关系树。看到了吗？$d(3, 2)$被计算了两次（一次是$d(2, 1)$需要的，一次是 $d(2, 2)$）。也许你会认为重复算一两个数没什么大不了的，但事实是：这样的重复不是单个结点，而是一棵子树。如果原来的三角形有 n 层，则调用关系树也会有 n 层，一共有 2^n-1 个结点。

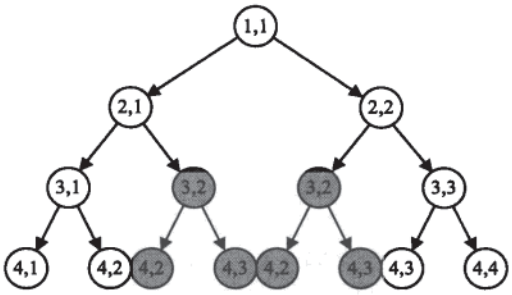


图 9-2　重叠子问题

提示 9-2：用直接递归的方法计算状态转移方程，效率往往十分低下。其原因是相同的子问题被重复计算了多次。

方法 2：递推计算。程序如下（需再次注意边界处理）：

```
int i, j;
for(j = 1; j <= n; j++) d[n][j] = a[n][j];
for(i = n-1; i >= 1; i--)
  for(j = 1; j <= i; j++)
    d[i][j] = a[i][j] + d[i+1][j]>?d[i+1][j+1];
```

程序的时间复杂度显然是 $O(n^2)$，但为什么可以这样计算呢？原因在于：i 是**逆序**枚举的，因此在计算 d[i][j]前，它所需要的 d[i+1][j]和 d[i+1][j+1]一定已经计算出来了。

提示 9-3：可以用递推法计算状态转移方程。递推的关键是边界和计算顺序。在多数情况下，递推法的时间复杂度是：状态总数×每个状态的决策个数×决策时间。如果不同状态的决策个数不同，需具体问题具体分析。

方法 3：记忆化搜索。程序分成两部分。首先用“memset(d,-1,sizeof(d));”把 d 全部初始化为-1，然后编写递归函数[①]：

```
int d(int i, int j)
{
    if(d[i][j] >= 0) return d[i][j];
    return d[i][j] = a[i][j] + (i == n ? 0 : d(i+1,j)>?d(i+1,j+1));
}
```

[①] 注意这个函数的工作方式并不像它表面显示的那样——如果你把-1 改成-2，并不是在把所有 d 值都初始化为-2！请只用 0 和-1 作为“批量赋值”的参数。

上述程序依然是递归的，但同时也把计算结果保存在数组 d 中。题目中说各个数都是非负的，因此如果已经计算过某个 d[i][j]，则它应是非负的。这样，只需把所有 d 初始化为 -1，即可通过判断是否 d[i][j]≥0 得知它是否已经被计算过。

最后，千万不要忘记在计算之后把它保存在 d[i][j]中。根据 C 语言“赋值语句本身有返回值”的规定，可以把保存 d[i][j]的工作合并到函数的返回语句中。

上述程序的方法称为记忆化（memoization），它虽然不像递推法那样显式地指明了计算顺序，但仍然可以保证每个结点只访问一次，如图 9-3 所示。

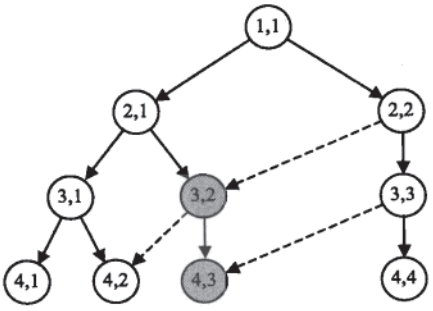


图 9-3　记忆化搜索

由于 i 和 j 都在 1～n 之间，所有不相同的结点一共只有 $O(n^2)$个。无论以怎样的顺序访问，时间复杂度均为 $O(n^2)$。从 2^n～n^2 是一个巨大的优化，它正是利用了数字三角形具有大量重叠子问题的特点。

提示 9-4：可以用记忆化搜索的方法计算状态转移方程。当采用记忆化搜索时，不必事先确定各状态的计算顺序，但需要记录每个状态“是否已经计算过”。

9.2　DAG 上的动态规划

有向无环图上的动态规划是学习动态规划的基础。很多问题都可以转化为 DAG 上的最长路、最短路或路径计数问题。

9.2.1　DAG 模型

例题 9-2　嵌套矩形

有 n 个矩形，每个矩形可以用两个整数 a, b 描述，表示它的长和宽。矩形 $X(a,b)$可以嵌套在矩形 $Y(c, d)$中当且仅当 $a<c, b<d$，或者 $b<c, a<d$（相当于把矩形 X 旋转 90°）。例如(1, 5)可以嵌套在(6, 2)内，但不能嵌套在(3, 4)内。你的任务是选出尽量多的矩形排成一行，使得除了最后一个之外，每一个矩形都可以嵌套在下一个矩形内。

【分析】

矩形之间的“可嵌套”关系是一个典型的二元关系，二元关系可以用图来建模。如果

矩形 X 可以嵌套在矩形 Y 里，我们就从 X 到 Y 连一条有向边。这个有向图是无环的，因为一个矩形无法直接或间接地嵌套在自己内部。换句话说，它是一个 DAG。这样，我们的任务便是求 DAG 上的最长路径。

例题 9-3　硬币问题

有 n 种硬币，面值分别为 $V_1, V_2, \cdots, V_n$，每种都有无限多。给定非负整数 S，可以选用多少个硬币，使得面值之和恰好为 S？输出硬币数目的最小值和最大值。$1 \leqslant n \leqslant 100$，$0 \leqslant S \leqslant 10000$，$1 \leqslant V_i \leqslant S$。

【分析】

尽管看上去和上一题很不一样，但本题的本质也是 DAG 上的路径问题。我们把每种面值看作一个点，表示“还需要凑足的面值”，则初始状态为 S，目标状态为 0。若当前在状态 i，每使用一个硬币 j，状态便转移到 $i-V_j$。

这个模型和上一题类似，但也有一些明显的不同之处：上题并没有确定路径的起点和终点（可以把任意矩形放在第一个和最后一个），而本题的起点必须为 S，终点必须为 0；把终点固定之后“最短路”才是有意义的。在上题中，最短序列显然是空（如果不允许空的话，就是单个矩形，不管怎样都是平凡的），而本题的最短路却不是那么容易确定的。

9.2.2　最长路及其字典序

首先思考“嵌套矩形”。如何求 DAG 中不固定起点的最长路径呢？仿照数字三角形的做法，设 $d(i)$表示从结点 i 出发的最长路长度，应该如何写状态转移方程呢？第一步只能走到它的相邻点，因此：

$$d(i) = \max\{d(j)+1 \mid (i,j) \in E\}$$

其中，E 为边集。最终答案是所有 $d(i)$中的最大值。根据前面的介绍，我们可以尝试按照递推或记忆化搜索的方式计算上式。不管怎样，都需要先把图建立出来，假设用邻接矩阵保存在矩阵 G 中（别忘了在编写主程序之前测试和调试这段程序，以确保建图过程正确无误）。接下来就来编写记忆化搜索程序（调用前需初始化 d 数组的所有值为 0）：

```
int dp(int i)
{
  int& ans = d[i];
  if(ans > 0) return ans;
  ans = 1;
  for(int j = 1; j <= n; j++)
     if(G[i][j]) ans >?= dp(j)+1;
  return ans;
}
```

这里用到了一个技巧：为表项 d[i]声明一个引用 ans。这样，任何对 ans 的读写实际上都是在对 d[i]进行。当 d[i]换成 d[i][j][k][l][m][n]这样很长的名字时，该技巧的优势就会很明显。

提示 9-5：在记忆化搜索中，可以为正在处理的表项声明一个引用，简化对它的读写操作。

还记得字典序吗？简单地说，就是先比第 1 个元素，再比第 2 个元素，……，在第一个能“分出胜负”的地方决定整体的大小关系。例如，序列 1, 2, 4, 3 比 1, 2, 3, 4 大，因为在从左到右第一个分出胜负的地方，4>3。

当然，字典序只是消除并列名次（tie breaking）的方法，长度才是首要要素。我们把所有 d 值计算出来以后，选择其中最大的 d[i]所对应的 i。如果有多个 i，则选择最小的 i，这样才能保证字典序最小。接下来可以选择 $d(i)=d(j)+1$ 且 $(i,j)\in E$ 的任何一个 j。为了让方案的字典序最小，应选择其中最小的 j。程序如下[①]：

```
void print_ans(int i)
{
  printf("%d ", i);
  for(int j = 1; j <= n; j++) if(G[i][j] && d[i] == d[j]+1)
  {
    print_ans(j);
    break;
  }
}
```

提示 9-6：根据各个状态的指标值可以依次确定各个最优决策，从而构造出完整方案。由于决策是依次确定的，所以很容易按照字典序打印出所有方案。

注意，当找到一个满足 d[i]==d[j]+1 的结点 j 后就应立刻递归打印从 j 开始的路径，并在递归返回后退出循环。如果要打印所有方案，只把 break 语句删除是不够的（想一想，为什么）。正确的方法是记录路径上的所有点，在递归结束时才一次性输出整条路径。程序留给读者编写。

有趣的是，如果把状态定义成“$d(i)$表示以结点 i 为终点的最长路径长度”，也能顺利求出最优值，却难以打印出字典序最小的方案。想一想，为什么？你能总结出一些规律吗？

9.2.3　固定终点的最长路和最短路

接下来考虑“硬币问题”。注意到最长路和最短路的求法是类似的，下面只考虑最长路。由于终点固定，$d(i)$的确切含义变为“从结点 i 出发到结点 0 的最长路径长度”。下面是求最长路的代码：

```
int dp(int S)
{
  int& ans = d[S];
  if(ans >= 0) return ans;
  ans = 0;
  for(int i = 1; i <= n; i++) if(S >= V[i]) ans >?= dp(S-V[i])+1;
```

[①] 输出的最后会有一个多余空格，并且没有回车符。在使用时，应在主程序调用 print_ans 后加一个回车符。如果比赛明确规定行末不允许有多余空格，则可以像前面介绍的那样加一个变量 first 来帮助判断。

```
  return ans;
}
```

注意到区别了吗？由于在本题中，路经长度是可以为 0 的（S 本身可以是 0），所以不能再用 d=0 来表示“这个 d 值还没有算过”。相应地，初始化时也不能再把 d 全设为 0，而要设置为一个负值——在正常情况下是取不到的。常见的方法是用-1 来表示“没有算过”，则初始化时只需用 memset(d,-1, sizeof(d))即可。至此，我们完整解释了上面的代码为什么把 if(ans>0)改成了 if(ans>=0)。

提示 9-7：当程序中需要用到特殊值时，应确保该值在正常情况下不会被取到。这不仅意味着特殊值不能有“正常的理解方式”，而且也不能在正常运算中“意外得到”。

不知读者有没有看出，上述代码有一个致命的错误，即由于结点 S 不一定真的能到达结点 0，所以需要用特殊的 d[S]值表示“无法到达”，但在上述代码中，如果 S 根本无法继续往前走，返回值是 0，将被误以为是“不用走，已经到达终点”的意思。如果把 ans 初始化为-1 呢？别忘了-1 代表“还没算过”，所以返回-1 相当于放弃了自己的劳动成果。如果把 ans 初始化为一个很大的整数，例如 2^{30} 呢？一开始就这么大，你觉得 ans >?= dp(i)+1 还能把 ans 变回“正常值”吗？如果改成很小的整数，例如-2^{30} 呢？从目前来看，它也会被认为是“还没算过”，但至少可以和所有 d 的初值分开了——只需把 if(ans≥0)改为 if(ans!=-1)即可，如下所示：

```
int dp(int S)
{
  int& ans = d[S];
  if(ans != -1) return ans;
  ans = -1<<30;
  for(int i = 1; i <= n; i++) if(S >= V[i]) ans >?= dp(S-V[i])+1;
  return ans;
}
```

提示 9-8：在记忆化搜索中，如果用特殊值表示“还没算过”，则必须将其和其他特殊值（如无解）区分开。

上述错误都是很常见的，甚至“顶尖高手”有时也会一时糊涂，掉入陷阱。意识到这些问题，寻求解决方案是不难的，但怕就怕调试很久以后仍然没有发现是哪里出了问题。另一个解决方法是不用特殊值表示“还没算过”，而用另外一个数组 vis[i]表示状态 i“是否被访问过”，如下所示：

```
int dp(int S)
{
  if(vis[S]) return d[S];
  vis[S] = 1;
  int& ans = d[S];
  ans = -1<<30;
```

```
  for(int i = 1; i <= n; i++) if(S >= V[i]) ans >?= dp(S-V[i])+1;
  return ans;
}
```

尽管多了一个数组，但可读性增强了许多：再也不用担心特殊值之间的冲突了，在任何情况下，记忆化搜索的初始化都可以用 memset(vis, 0, sizeof(vis))[①]执行。

提示 9-9：在记忆化搜索中，可以用 vis 数组记录每个状态是否计算过，以一些内存为代价增强程序的可读性，同时减少出错的可能。

本题要求最小、最大两个值，记忆化搜索就必须写两个。在这种情况下，还是递推来得更加方便（此时需注意递推的顺序）：

```
min[0] = max[0] = 0;
for(int i = 1; i <= S; i++)
{
  min[i] = INF; max[i] = -INF;
}
for(int i = 1; i <= S; i++)
  for(int j = 1; j <= n; j++)
    if(i >= V[j])
    {
      min[i] <?= min[i-V[j]] + 1;
      max[i] >?= max[i-V[j]] + 1;
    }
printf("%d %d\n", min[S], max[S]);
```

如何输出字典序最小的方案呢？刚刚介绍的方法仍然适用，如下所示：

```
void print_ans(int* d, int S)
{
  for(int i = 1; i <= n; i++)
    if(S>=V[i] && d[S]==d[S-V[i]]+1)
    {
      printf("%d ", i);
      print_ans(d, S-V[i]);
      break;
    }
}
```

然后分别调用 print_ans(min, S)（别忘了在后面加个回车符）和 print_ans(max, S)即可。注意到输出路径部分和上题的区别：上题打印的是路径上的点，而这里打印的是路径上的边。还记得数组可以作为指针传递吗？这里需要强调的一点是：数组作为指针传递时，不

[①] 如果状态比较复杂，推荐用 STL 中的 map 而不是普通数组保存状态值。这样，判断状态 S 是否算过只需用 if(d.count(S))即可。有兴趣的读者可以查阅相关资料。

会复制数组中的数据，因此不必担心这样会带来不必要的时间开销。

提示 9-10：当用递推法计算出各个状态的指标之后，可以用与记忆化搜索完全相同的方式打印方案。

不少人喜欢另外一种打印路径的方法：递推时直接用 min_coin[S]记录满足 min[S]==min[S-V[i]]+1 的最小的 i，则打印路径时可以省去 print_ans 函数中的循环，并可以方便地把递归改成迭代（原来的也可以改成迭代，但不那么自然）。具体来说，需要把递推过程改成以下形式：

```
for(int i = 1; i <= S; i++)
  for(int j = 1; j <= n; j++)
    if(i >= V[j])
    {
      if(min[i] > min[i-V[j]] + 1)
      {
        min[i] = min[i-V[j]] + 1;
        min_coin[i] = j;
      }
      if(max[i] < max[i-V[j]] + 1)
      {
        max[i] = max[i-V[j]] + 1;
        max_coin[i] = j;
      }
    }
```

注意，判断中用的是“>”和“<”，而不是“>=”和“<=”，原因在于“字典序最小解”要求当 min/max 值相同时取最小的 i 值。反过来，如果 j 是从大到小枚举的，就需要把“>”和“<”改成“>=”和“<=”才能求出字典序最小解。

在求出 min_coin 和 max_coin 之后，只需调用 print_ans(min_coin, S)和 print_ans(max_coin, S)即可。

```
void print_ans(int* d, int S)
{
  while(S)
  {
    printf("%d ", d[S]);
    S -= V[d[S]];
  }
}
```

有趣的是，这个方法正是前面曾经提到过的“用空间换时间”的经典例子——用 min_coin 和 max_coin 数组消除了原来 print_ans 中的循环。

提示 9-11：无论是用记忆化搜索还是递推，如果在计算最优值的同时“顺便”算出各个状态下的第一次最优决策，则往往能让打印方案的过程更加简单、高效。这是一个典型的“用空间换时间”的例子。

9.3　0-1 背包问题

0-1 背包问题是最广为人知的动态规划问题之一，拥有很多变形。尽管在理解之后并不难写出程序，但初学者往往需要较多的时间才能掌握它。

9.3.1　多阶段决策问题

在介绍 0-1 背包问题之前，先来看一个引例。

例题 9-4　物品无限的背包问题

有 n 种物品，每种均有无穷多个。第 i 种物品的体积为 V_i，重量为 W_i。选一些物品装到一个容量为 C 的背包中，使得背包内物品在总体积不超过 C 的前提下重量尽量大。$1 \leqslant n \leqslant 100, 1 \leqslant V_i \leqslant C \leqslant 10000, 1 \leqslant W_i \leqslant 10^6$。

【分析】

很眼熟是吗？没错，它很像 9.2 节中的硬币问题，只不过“面值之和恰好为 S”改成了“体积之和不超过 C”，另外增加了一个新的属性——重量，相当于把原来的无权图改成了带权图（weighted graph）。这样，问题就变为了求以 C 为起点（终点任意）的、边权之和最大的路径。

与前面相比，DAG 从“无权”变成了“带权”，但这并没有给我们带来任何困难，此时只需在代码上把某个地方从“+1”变成“+W[i]”即可。你能找到吗？

提示 9-12：动态规划的适用性很广。不少可以用动态规划解决的题目，在条件稍微变化后只需对状态转移方程做少量修改即可解决新问题。

例题 9-5　0-1 背包问题

有 n 种物品，每种只有一个。第 i 种物品的体积为 V_i，重量为 W_i。选一些物品装到一个容量为 C 的背包，使得背包内物品在总体积不超过 C 的前提下重量尽量大。$1 \leqslant n \leqslant 100, 1 \leqslant V_i \leqslant C \leqslant 10000, 1 \leqslant W_i \leqslant 10^6$。

【分析】

不知你有没有发现，刚才的方法已经不适用了：只凭“剩余体积”这个状态，我们无法得知每个物品是否已经用过。换句话说，原来的状态转移太乱了，任何时候都允许使用任何一种物品，难以控制。为了消除这种混乱，需要让状态转移（也就是决策）有序化。

还记得“多阶段决策问题”这个名称吗？我们曾经在回溯法中提到过它。简单地说，每做一次决策就可以得到解的一部分，当所有决策做完之后，完整的解就“浮出水面”了。在回溯法中，每次决策对应于给一个结点产生新的子树，而解的生成过程对应一棵解答树，

结点的层数就是“下一个待填充位置”cur。

提示 9-13：多阶段决策的最优化问题往往可以用动态规划解决，其中，状态及其转移类似于回溯法中的解答树。解答树中的“层数”，也就是递归函数中的“当前填充位置”cur，描述的是即将完成的决策序号，在动态规划中被称为“阶段”。

引入“阶段”之后，算法便不难设计了：用 $d(i,j)$表示当前在第 i 层，背包剩余容量为 j 时接下来的最大重量和，则 $d(i,j)=\max\{d(i+1,j),d(i+1,j-V[i])+W[i]\}$，边界是 $i>n$ 时 $d(i,j)=0$，$j<0$ 时为负无穷（一般不会初始化这个边界，而是只当 $j\geq V[i]$时才计算第二项）。

说得更“白话”一点，$d(i,j)$表示“把第 $i, i+1, i+2,\cdots, n$ 个物品装到容量为 j 的背包中的最大总重量”。事实上，这个说法更加常用——“阶段”只是帮助我们思考的，在动态规划的状态描述中最好避免“阶段”、“层”这样的术语。很多教材和资料直接给出了这样的状态描述，而我们则是花费了大量的篇幅叙述为什么会想到要划分阶段以及和回溯法的内在联系——如果对此理解不够深入，很容易出现“每次碰到新题自己都想不出来，但一看题解就懂”的尴尬情况。

提示 9-14：学习动态规划的题解，除了要理解状态表示及其转移方程外，最好思考一下为什么会想到这样的状态表示。

和往常一样，在得到状态转移方程之后，还需思考如何编写程序。尽管在很多情况下，记忆化搜索程序更直观、易懂，但在 0-1 背包问题中，递推法更加理想。为什么呢？因为当有了“阶段”定义后，计算顺序变得非常明显。

提示 9-15：在多阶段决策问题中，阶段定义了天然的计算顺序。

下面是代码，答案是 d[1][C]：

```
for(int i = n; i >= 1; i--)
    for(int j = 0; j <= C; j++)
    {
      d[i][j] = (i==n ? 0 : d[i+1][j]);
      if(j >= V[i]) d[i][j] >?= d[i+1][j-V[i]]+W[i];
}
```

前面说过，i 必须逆序枚举，但 j 的循环次序是无关紧要的。

9.3.2 规划方向

聪明的读者也许看出来了，还有另外一种“对称”的状态定义：用 $f(i,j)$表示“把前 i 个物品装到容量为 j 的背包中的最大总重量”，它的状态转移方程也不难得出：

$$f(i,j)=\max\{f(i-1,j),f(i-1,j-V[i])+W[i]\}$$

边界是类似的：$i=0$ 时为 0，$j<0$ 时为负无穷，最终答案为 $f(n,C)$。代码也是类似的：

```
for(int i = 1; i <= n; i++)
    for(int j = 0; j <= C; j++)
```

```
    {
      f[i][j] = (i==1 ? 0 : f[i-1][j]);
      if(j >= V[i]) f[i][j] >?= f[i-1][j-V[i]]+W[i];
    }
```

看上去这两种方式是完全对称的，但其实存在细微区别：新的状态定义 $f(i, j)$允许我们边读入边计算，而不必把 V 和 W 保存下来。

```
for(int i = 1; i <= n; i++){
    scanf("%d%d", &V, &W);
    for(int j = 0; j <= C; j++){
      f[i][j] = (i==1 ? 0 : f[i-1][j]);
      if(j >= V) f[i][j] >?= f[i-1][j-V]+W;
    }
}
```

9.3.3　滚动数组

更奇妙的是，还可以把数组 f 变成一维的：

```
memset(f, 0, sizeof(f));
for(int i = 1; i <= n; i++)
{
  scanf("%d%d", &V, &W);
  for(int j = C; j >= 0; j--)
  if(j >= V) f[j] >?= f[j-V]+W;
}
```

为什么这样做是正确的呢？下面来看一下 $f(i, j)$的计算过程，如图 9-4 所示。

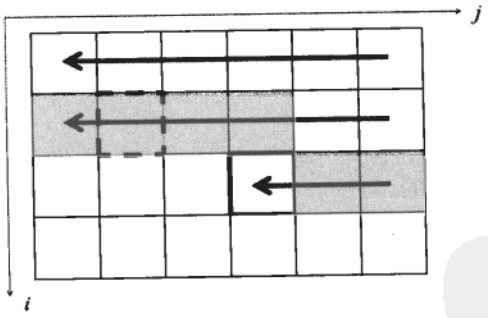


图 9-4　0-1 背包问题的计算顺序

f 数组是从上到下、从右往左计算的。在计算 $f(i,j)$之前，f[j]里保存的就是 $f(i-1,j)$的值，而 f[j-W]里保存的是 $f(i-1,j-W)$而不是 $f(i,j-W)$——别忘了 j 是逆序枚举的，此时 $f(i,j-W)$还没有算出来。这样，f[j]>?=f[j-V]+W 实际上是把 $\max\{f(i-1,j), f(i-1,j-V)\}$ 保存在 f[j]中，覆盖掉 f[j]原来的 $f(i-1,j)$。

提示 9-16：在递推法中，如果计算顺序很特殊，而且计算新状态所用到的原状态不多，可以尝试着用滚动数组减少内存开销。

滚动数组虽好，但也存在一些不尽如人意的地方，例如打印方案较困难。当动态规划结束之后，只有最后一个阶段的状态值，而没有前面的值。不过这也不能完全归咎于滚动数组，规划方向也有一定责任——即使用二维数组，打印方案也不是特别方便。事实上，对于“前 i 个物品”这样的规划方向，只能用逆向的打印方案，而且还不能保证它的字典序最小（字典序比较是从前往后的）。

提示 9-17：在使用滚动数组后，解的打印变得困难了，所以在需要打印方案甚至要求字典序最小方案的场合，应慎用滚动数组。

9.4 递归结构中的动态规划

本节介绍一些递归结构中的动态规划，包括表达式、凸多边形和树。尽管它们的形式和解法千差万别，但都用到了动态规划的思想：从复杂的题目背景中抽象出状态表示，然后设计它们之间的转移。

9.4.1 表达式上的动态规划

例题 9-6 最优矩阵链乘

一个 $n\times m$ 矩阵由 n 行 m 列共 $n\times m$ 个数排列而成。两个矩阵 A 和 B 可以相乘当且仅当 A 的列数等于 B 的行数。一个 $n\times m$ 的矩阵乘以一个 $m\times p$ 的矩阵等于一个 $n\times p$ 的矩阵，运算量为 mnp。

矩阵乘法不满足分配律，但满足结合律，因此 $A\times B\times C$ 既可以按顺序 $(A\times B)\times C$ 进行，也可以按 $A\times(B\times C)$ 来进行。假设 A、B、C 分别是 2×3，3×4 和 4×5 的，则 $(A\times B)\times C$ 的运算量为 $2\times3\times4+2\times4\times5=64$，$A\times(B\times C)$ 的运算量为 $3\times4\times5+2\times3\times5=90$。显然第一种顺序节省运算量。

给出 n 个矩阵组成的序列，设计一种方法把它们依次乘起来，使得总的运算量尽量小。假设第 i 个矩阵 A_i 是 $p_{i-1}\times p_i$ 的。

【分析】

本题任务是设计一个表达式。在整个表达式中，一定有一个“最后一次乘法”。假设它是第 k 个乘号，则在此之前已经算出了 $P=A_1\times A_2\times\cdots\times A_k$ 和 $Q=A_{k+1}\times A_{k+2}\times\cdots\times A_n$。由于 P 和 Q 的计算过程互不相干，而且无论按照怎样的顺序，P 和 Q 的值都不会发生改变，因此只需分别让 P 和 Q 按照最优方案计算（最优子结构！）即可。为了计算 P 的最优方案，还需要继续枚举 $P=A_1\times A_2\times\cdots\times A_k$ 的“最后一次乘法”，把它分成两部分。不难发现，无论怎么分，在任意时候，我们需要处理的子问题都形如“把 A_i、A_{i+1}、…、A_j 乘起来需要多少次乘法？”如果用状态 $f(i,j)$ 表示这个子问题的值，不难列出如下的状态转移方程：

$$f(i,j)=\max\{f(i,k)+f(k,j)+p_{i-1}p_kp_j\}$$

边界为 $f(i,i)=0$。上述方程有些特殊：记忆化搜索固然没问题，但如果要写成递推，无论按照 i 还是 j 的递增或递减顺序均不正确。正确的方法是按照 $j-i$ 递增的顺序递推，因为长区间的值依赖于短区间的值。

9.4.2　凸多边形上的动态规划

例题 9-7　最优三角剖分

对于一个 n 个顶点的凸多边形，有很多种方法可以对它进行三角剖分（triangulation），即用 $n-3$ 条互不相交的对角线把凸多边形分成 $n-2$ 个三角形。为每个三角形规定一个权函数 $f(i,j,k)$（如三角形的周长或 3 个顶点的权和），求让所有三角形权和最大的方案。

【分析】

本题和最优矩阵链乘问题十分相似，但存在一个显著不同：链乘表达式反映了决策过程，而剖分不反映决策过程。举例来说，在链乘问题中，方案$((A_1A_2)(A_3(A_4A_5)))$只能是先把序列分成 A_1A_2 和 $A_3A_4A_5$ 两部分，而对于一个三角剖分，“第一刀”可以是任何一条对角线，如图 9-5 所示。

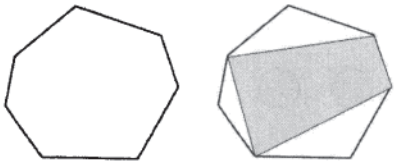


图 9-5　难以简洁表示的状态

如果允许随意切割，则“半成品”多边形的各个顶点是可以在原多边形中随意选取的，很难简洁定义成状态，而“矩阵链乘”就不存在这个问题——无论怎样决策，面临的子问题一定可以用区间表示。在这样的情况下，有必要把决策的顺序规范化，使得在规范的决策顺序下，任意状态都能用区间表示。

定义 $f(i,j)$为“从顶点 i 到顶点 j 所构成的子多边形的最大三角剖分权和”，则边 P_iP_j 在此多边形内一定恰好属于一个三角形 $P_iP_jP_k$。只要枚举 k 的位置，就能把多边形分割为两部分。注意到两个多边形的顶点序列仍是连续的，因此有状态转移方程：

$$f(i,j)=\max\{f(i,k)+f(k,j)+w(i,j,k)\}$$

时间复杂度为 $O(n^3)$。原问题的解为 $f(1,1)$，因为第一次枚举决策 k 后得到的退化三角形 $P_1P_kP_1$ 的权为 0。另外，这里的 i 可以大于 j，在这种情况下 k 需要枚举 $i+1,i+2,\cdots,n,1,2,\cdots,j-1$。

9.4.3　树上的动态规划

例题 9-8　树的最大独立集

对于一棵 n 个结点的无根树，选出尽量多的结点，使得任何两个结点均不相邻（称为

最大独立集），然后输入 $n-1$ 条无向边，输出一个最大独立集（如果有多解，则任意输出一组）。

【分析】

我们试着用 $d(i)$表示以 i 为根结点的子树的最大独立集大小。此时需要注意的是，本题的树是无根的：没有所谓的“父子”关系，而只有一些无向边。没关系，只要任选一个根 r，无根树就变成了有根树，上述状态定义也就有意义了。

结点 i 只有两种决策：选和不选。如果不选 i，则问题转化为了求出 i 的所有儿子的 d 值再相加；如果选 i，则它的儿子全部不能选，问题转化为了求出 i 的所有孙子的 d 值之和。换句话说，状态转移方程为：

$$d(i)=\max\{1+\sum_{j\in gs(i)}d(j),\sum_{j\in s(i)}d(j)\}$$

其中 $gs(i)$和 $s(i)$分别为 i 的孙子集合与儿子集合，如图 9-6 所示。

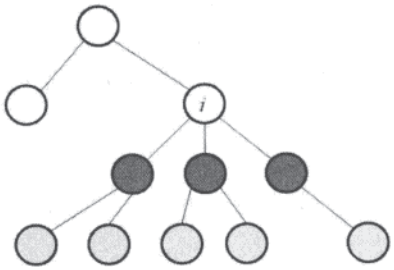


图 9-6　结点 i 的 $gs(i)$（浅灰色）和 $s(i)$（深灰色）

代码应如何编写呢？上面的方程涉及“枚举结点 i 的所有儿子和所有孙子”，颇为不便。其实可以换一个角度来看：不从 i 找 $s(i)$和 $gs(i)$的元素，而从 $s(i)$和 $gs(i)$的元素找 i。换句话说，当计算出一个 $d(i)$后，用它去更新 i 的父亲和祖父结点的累加值 $\sum_{j\in gs(i)}d(j)$ 和 $\sum_{j\in s(i)}d(j)$。这样一来，每个结点甚至不必记录它的儿子结点有哪些，只需记录父亲结点即可。

提示 9-18：传统的递推法可以表示成“对于每个状态 i，计算 $f(i)$”，或者称为“填表法”。这需要对于每个状态 i，找到 $f(i)$依赖的所有状态，在某些时候并不方便。另一种方法是“对于每个状态 i，更新 $f(i)$所影响到的状态”，或者称为“刷表法”，有时比填表法方便。但需要注意的是，只有当每个状态所依赖的状态对它的影响相互独立时才能用刷表法。例如，在最优矩阵链乘问题中就无法直接使用刷表法。

9.5　集合上的动态规划

例题 9-9　最优配对问题

空间里有 n 个点 $P_0,P_1,\cdots,P_{n-1}$，你的任务是把它们配成 $n/2$ 对（n 是偶数），使得每个点恰好在一个点对中。所有点对中两点的距离之和应尽量小。$n\leqslant 20$，$|x_i|,|y_i|,|z_i|\leqslant 10000$。

【分析】

既然每个点都要配对，很容易把问题看成如下的多阶段决策过程：先确定 P_0 和谁配对，然后是 P_1，接下来是 P_2，……，最后是 P_{n-1}。按照前面的思路，设 $d(i)$表示把前 i 个点两两配对的最小距离和，然后考虑第 i 个点的决策——它和谁配对呢？假设它和点 j 配对($j<i$)，那么接下来的问题应是“把前 i-1 个点中除了 j 之外的其他点两两配对”，它显然无法用任何一个 d 值来刻画——我们的状态定义无法体现出“除了一些点之外”这样的限制。

9.5.1　状态及其转移

当发现状态无法转移后，常见的方法是增加维度，即增加新的因素，更细致地描述状态。既然刚才提到了“除了某些元素之外”，不妨把它作为状态的一部分，设 $d(i,S)$表示把前 i 个点中，位于集合 S 中的元素两两配对的最小距离和，则状态转移方程为：

$$d(i,S)=\min\{|P_iP_j|+d(i-1,S-\{i\}-\{j\})\mid j\in S\}$$

其中$|P_iP_j|$表示点 P_i 和 P_j 之间的距离。边界是 $d(-1,S)=0$。方程看上去很不错，但实现起来有问题：如何表示集合 S 呢？由于它要作为数组 d 中的第二维下标，所以需要用整数来表示集合，确切地说，是{0, 1, 2,…,n−1}的任意子集（subset）。

在“子集枚举”部分，曾介绍过子集的二进制表示。现在，它再次派上用场：

```
for(int i = 0; i < n; i++)
  for(int S = 0; S < (1<<n); S++)
  {
    d[i][S] = INF;
    for(int j = 0; j < i; j++) if(S & (1<<j))
      d[i][S] <?= dist(i, j) + d[i-1][S^(1<<i)^(1<<j)];
}
```

上述程序故意用了很多括号，传达给读者的信息是：位运算的优先级低，初学者很容易弄错。例如，1<<n-1 的正确解释是 1<<(n-1)，因为减法的优先级比左移要高。为了保险起见，应多用括号。另一个技巧是利用 C 语言中“0 为假，非 0 为真”的规定简化表达式：if(S & (1<<j))的实际含义是 if((S & (1<<j)) != 0)。

提示 9-19：*位运算的优先级往往比较低。如果心里对表达式的计算顺序不确定，应多用括号。*

由于大量使用了形如 1<<n 的表达式，这里“费点笔墨”再说两句。左移运算符<<的含义是“把各个位往左移动，右边补 0”。根据二进制运算法则，每次左移一位就相当于乘以 2，因此 a<<b 相当于 $a*2^b$，而在集合表示法中，1<<i 代表单元素集合{i}。由于 0 表示空集，S & (1<<j)不等于 0 就意味着“S 和{j}的交集不为空”。

9.5.2　隐含的阶段

上面的方程可以进一步简化。事实上，阶段 i 根本不用保存，它已经隐含在 S 中了——S 中的最大元素就是 i。这样，可直接用 $d(S)$表示“把 S 中的元素两两配对的最小距离和”，

则状态转移方程为：

$$d(S)=\min\{|P_iP_j|+d(S-\{i\}-\{j\})\mid j\in S, i=\max\{S\}\}$$

状态有 2^n 个，每个状态有 $O(n)$ 种转移方式，总时间复杂度为 $O(n2^n)$。

提示 9-20：如果用二进制表示子集并进行动态规划，集合中的元素就隐含了阶段信息。例如，可以把集合中的最大元素想象成“阶段”。

值得一提的是，不少人一直在用这样的状态转移方程：

$$d(S)=\min\{|P_iP_j|+d(S-\{i\}-\{j\})\mid i,j\in S\}$$

它和刚才的方程很类似，唯一的不同是：i 和 j 都是需要枚举的。这样做虽然也没错，但每个状态的转移次数高达 $O(n^2)$，总时间复杂度为 $O(n^22^n)$，比刚才的方法慢。这个例子再次说明：即使用相同的状态描述，减少决策也是很重要的。

提示 9-21：即使状态定义相同，过多地考虑不必要的决策仍可能会导致时间复杂度上升。

接下来出现了一个新问题：如何求出 S 中的最大元素呢？别想复杂了，用一个循环判断即可。当 S 取遍 $\{0, 1, 2, \cdots, n-1\}$ 的所有子集时，平均判断次数仅为 2（想一想，为什么）。

```
for(int S = 0; S < (1<<n); S++)
{
  int i, j;
  d[S] = INF;
  for(i = 0; i < n; i++)
     if(S && (1<<i)) break;
  for(j = i+1; j < n; j++)
     if(S & (1<<j)) d[S] <?= dist(i, j) + d[S^(1<<i)^(1<<j)];
}
```

注意，在上述的程序中求出的 i 是 S 中的最小元素，而不是最大元素，但这并不影响答案。另外，j 的枚举只需从 $i+1$ 开始——既然 i 是 S 中的最小元素，则说明其他元素自然均比 i 大。最后需要说明的是 S 的枚举顺序。不难发现：如果 S' 是 S 的真子集，则一定有 $S'<S$，因此若以 S 递增的顺序计算，需要用到某个 d 值时，它一定已经计算出来了。

提示 9-22：如果 S' 是 S 的真子集，则一定有 $S'<S$。在用递推法实现子集的动态规划时，该规则往往可以确定计算顺序。

9.6 训 练 参 考

在此仍然先给出 UVaOJ 上的题目：历史考试（111）、堆砌盒子（103）、最长公共子序列（10405）、硬币找零（674）、切割木棍（10003）、单向 TSP（116）、越大越聪明？（10131）、双塔（10066）、假期（10192）、美元（147）、让我来数数方案（357）、划分硬币（562）、最优数组乘法序列（348）、CD（624）、超级天平（10130）、妥协（531）、

霍默辛普森（10465）、滑雪（10285）、巴比伦塔（437）、Bachet 的游戏（10404）、细胞结构（620）、走在安全的一边（825）、不同的子序列（10069）、波形序列（10534）、立方体塔（10051）、一维独立钻石（10651）、来去匆匆（590）、电子硬币（10306）、字符串变回文（10739）、最优排序二叉树（10304）、筷子（10271）、又是回文数（10617）、立方数拆分（11137）、重量和度量（10154）、移动大冒险第四部（10201）、制造回文串（10453）、编辑步数（10029）、付账（10313）、受伤的皇后（10401）、取数游戏（10891）、最长的回文串（11151）、智力比赛组队（10911）、王子和公主（10635）、沙漏中的路径（10564）、快餐（662）、买可乐（10626）、免费糖果（10118）、讲座安排（607）、化学反应（10604）、网格上行走（10913）、反物质射线（11008）、电子基因（10723）、字符串分割（11258）、机器人（10599）、校长的烦恼（10817）、守店人（10163）、文本格式化（709）、新瓶装旧酒（10280）、邪恶党的密谋（10558）、字符串穿插（11081）。

接下来是 CII 上的题目：最强装备（2422）、并行期望值（2344）、人类基因（2324）、回文分解（2560）、船（2511）、莫尔斯序列解码（2426）、括号序列（2451）、递增序列（2583）、单人猜价格游戏（2587）、免税店（2614）、折叠（2692）、珍珠（2675）、贪婪的 Steve（2669）、手机键盘（2161）、有趣的游戏（3136）、最大值最小的三角剖分（3132）、土地划分税（3151）、混乱的登录名（3189）、岛和桥（3267）、周游（3305）、指令（3492）、金字塔探险（3516）、装饰灯（3215）、乌龟的玩笑（3222）、马拉松（2810）、齐唱（2812）、机器卡车（3983）、速配（4404）、混乱（3978）、懒惰的工人（2532）、西洋双陆棋（3884）、赛车（3404）、ACM 拼图（4058）、周期（3608）、机器人迷宫（4040）。

第 10 章　数学概念与方法

学习目标

- ☑ 熟练掌握扩展欧几里德算法和它的时间复杂度
- ☑ 熟练掌握用筛法构造素数表，了解素数定理
- ☑ 学会求二元线性不定方程的整数解
- ☑ 熟练掌握模运算规则、快速幂取模算法和模线性方程的解法
- ☑ 熟悉杨辉三角、二项式定理和组合数的基本性质
- ☑ 学会推导约数个数公式和欧拉函数公式
- ☑ 熟练掌握可重集全排列的编码和解码算法
- ☑ 理解样本空间、事件和概率，学会用组合计数的方法计算离散概率
- ☑ 熟悉常见计数序列，如 Fibonacci 数列、Catalan 数列等
- ☑ 熟悉建立递推关系的基本方法、常见错误和实现技巧

没有数学就没有算法；没有好的数学基础，也很难在算法上有所成就。本章介绍算法竞赛中涉及的常见数学概念和方法，包括数论、排列组合、递推关系和离散概率等。

10.1　数 论 初 步

数论被“数学王子”高斯誉为整个数学王国的皇后。在算法竞赛中，数论常常以各种面貌出现，但万变不离其宗，大部分数论题目并不涉及多少特殊的知识，但对数学思维和能力要求较高。本节介绍几个最为常用的算法，并通过例题展示一些常用的思维方式。

10.1.1　除法表达式

给出一个这样的除法表达式：$X_1 / X_2 / X_3 / \cdots / X_k$，其中 X_i 是正整数。除法表达式应当按照从左到右的顺序求和，例如表达式 1/2/1/2 的值为 1/4。但可以在表达式中嵌入括号以改变计算顺序，例如表达式(1/2)/(1/2)的值为 1。

输入 $X_1, X_2, \cdots, X_k$，判断是否可以通过添加括号，使表达式的值为整数。$K \leqslant 10000$, $X_i \leqslant 10^9$。

【分析】

表达式的值一定可以写成 A/B 的形式：A 是其中一些 X_i 的乘积，而 B 是其他数的乘积。不难发现，X_2 必须放在分母位置，那其他数呢？

幸运的是，其他数都可以在分子位置：

$$E = X_1 / (X_2 / X_3 / \cdots X_k) = \frac{X_1 X_3 X_4 \cdots X_k}{X_2}$$

接下来的问题就变成了：判断 E 是否为整数。

第 1 种方法是利用前面介绍的高精度运算：k 次乘法加一次除法。显然，这个方法是正确的，但却比较麻烦。

第 2 种方法是利用唯一分解定理，把 X_2 写成若干素数相乘的形式：

$$X_2 = p_1^{a_1} p_2^{a_2} p_3^{a_3} \cdots$$

然后依次判断每个 $p_i^{a_i}$ 是否是 $X_1X_3X_4\cdots X_k$ 的约数。这次不用高精度乘法了：只需把所有 X_i 中 p_i 的指数加起来。如果结果比 a_i 小，说明还会有 p_i 约不掉，因此 E 不是整数。这种方法在第 5 章中已经用过，这里不再赘述。

第 3 种方法是直接约分：每次约掉 X_i 和 X_2 的最大公约数 $\gcd(X_i, X_2)$，则当且仅当约分结束后 X_2=1 时 E 为整数，程序如下。

```
int judge(int* X)
{
     X[2] /= gcd(X[2], X[1]);
     for(int i = 3; i <= k; i++) X[2] /= gcd(X[i], X[2]);
     return X[2] == 1;
}
```

整个算法的时间效率取决于这里的 gcd 算法。尽管依次试除也能得到正确的结果，但还有一个简单、高效，而且相当优美的算法——辗转相除法。它也许是最广为人知的数论算法了。

辗转相除法的关键在于如下恒等式：gcd(a,b) = gcd(b, a mod b)。它和边界条件 gcd(a, 0)=a 一起构成了下面的程序：

```
int gcd(int a, int b)
{
   return b == 0 ? a : gcd(b, a%b);
}
```

这个算法称为欧几里德算法（Euclid algorithm）。既然是递归，我们免不了问一句：会栈溢出吗？答案是：不会。可以证明，gcd 函数的递归层数不超过 $4.785\lg N + 1.6723$，其中 N=max{a,b}。值得一提的是，让 gcd 递归层数最多的是 gcd(F_n,F_{n-1})，其中 F_n 是后文要介绍的 Fibonacci 数。

顺便说一句，利用 gcd，还可以求出两个整数 a 和 b 的最小公倍数 $lcm(a,b)$。这个结论很容易由唯一分解定理得到。设

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$$
$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}$$

则

$$\gcd(a,b) = p_1^{\min\{e_1,f_1\}} p_2^{\min\{e_2,f_2\}} \cdots p_r^{\min\{e_r,f_r\}}$$
$$lcm(a,b) = p_1^{\max\{e_1,f_1\}} p_2^{\max\{e_2,f_2\}} \cdots p_r^{\max\{e_r,f_r\}}$$

有了它，就不难验证 $\gcd(a,b)\times lcm(a,b)=a\times b$ 了。不过即使有了公式也不要大意！如果你把 *lcm* 写成 a * b/gcd(a,b)，可能会因此丢掉不少分数——a*b 可能会溢出！正确的写法是先除后乘，即 a/gcd(a,b) * b。这样一来，只要题面上保证最终结果在 int 范围之内，这个函数就不会出错。但前一份代码却不是这样：即使最终答案在 int 范围之内，也有可能中间过程越界。注意这样的细节，毕竟算法竞赛不是数学竞赛。

10.1.2 无平方因子的数

给出正整数 n 和 m，区间[n, m]内的“无平方因子”的数有多少个？整数 p 无平方因子当且仅当不存在 $k>1$，使得 p 是 k^2 的倍数。$1\leqslant n\leqslant m\leqslant 10^{12}$, $m-n\leqslant 10^7$。

【分析】

对于这样的限制，直接枚举判断会超时：需要判断 10^7 个整数，而每个整数还需要花费一定的时间判断是否没有平方因子。怎么办呢？在介绍具体算法之前，需要学会用 Eratosthenes 筛法构造 $1\sim n$ 的素数表。

筛法的思想特别简单：对于不超过 n 的每个非负整数 p，删除 $2p$, $3p$, $4p$,…，当处理完所有数之后，还没有被删除的就是素数。如果用 vis[i]表示 i 已经被删除，筛法的代码可以写成这样：

```
memset(vis, 0, sizeof(vis));
for(int i = 2; i <= n; i++)
  for(int j = i*2; j <= n; j+=i) vis[j] = 1;
```

尽管可以继续改进，但这份代码已经相当高效了。为什么呢？给定外层循环变量 i，内层循环的次数是 $\left\lfloor\frac{n}{i}\right\rfloor-1<\frac{n}{i}$。这样，循环的总次数小于 $\frac{n}{2}+\frac{n}{3}+\cdots+\frac{n}{n}=O(n\log n)$。这个结论来源于欧拉在 1734 年得到的结果：$1+\frac{1}{2}+\frac{1}{3}+\cdots\frac{1}{n}=\ln(n+1)+\gamma$，其中欧拉常数 $\gamma\approx 0.577218$。这样低的时间复杂度允许我们在很短的时间内得到 10^6 以内的所有素数。

下面我们来改进这份代码。首先，在“对于不超过 n 的每个非负整数 p”中，p 可以限定为素数——只需在第二重循环前加一个判断 if(!vis[i])即可。另外，内层循环也不必从 i*2 开始——它已经在 i=2 时被筛掉了。改进后的代码如下：

```
int m = sqrt(n+0.5);
int c = 0;
memset(vis, 0, sizeof(vis));
for(int i = 2; i <= m; i++) if(!vis[i])
{
  prime[c++] = i;
  for(int j = i*i; j <= n; j+=i) vis[j] = 1;
}
```

注意，上面的代码加入了一项新功能：把素数保存在素数数组 prime 里，其中 c 为素数总数，prime[i]是编号为 i 的素数（从 0 开始编号）。这里有一个有意思的问题：对于给定

的 n，c 的值是多少呢？换句话说，不超过 n 的正整数中，有多少个是素数呢？

素数定理： $\pi(x) \sim \frac{x}{\ln x}$

其中 $\pi(x)$ 表示不超过 x 的素数的个数。上述定理的直观含义是：它和 $x/\ln x$ 比较接近——对于算法入门来说，这已足够。表 10-1 给出了一些值来加深读者的印象。

表 10-1　素数定理的直观验证

N	10^2	10^3	10^4	10^5	10^6	10^7	10^8
$\pi(n)$	25	168	1229	9592	78498	664579	5761455
$n/\ln n$	22	145	1086	8686	72382	620421	5428681

最后回到原题：如何求出区间内无平方因子的数？方法和筛素数是类似的：对于不超过 $\sqrt{m}$ 的所有素数 p，筛掉区间 $[n, m]$ 内 p^2 的所有倍数。

10.1.3　直线上的点

求直线 $ax+by+c=0$ 上有多少个整点 (x,y) 满足 $x \in [x_1, x_2], y \in [y_1, y_2]$。

【分析】

在解决这个问题之前，让我们首先学习扩展欧几里德算法——找出一对整数 (x,y)，使得 $ax+by=\gcd(a,b)$。注意，这里的 x 和 y 不一定是正数，也可能是负数或者 0。例如 gcd(6,15)=3，6*3−15*1=3，其中 $x=3, y=-1$。这个方程还有其他解，如 $x=-2, y=1$。

下面是扩展欧几里德算法的程序：

```
void gcd(int a, int b, int& d, int& x, int& y)
{
  if(!b){ d = a; x = 1; y = 0; }
  else{ gcd(b, a%b, d, y, x); y -= x*(a/b); }
}
```

用数学归纳法并不难证明算法的正确性，这里略去。注意在递归调用时，x 和 y 的顺序变了，而边界也是不难得出的：“gcd(a,0)=1*a−0*0=a”。这样，唯一需要记忆的是“y−=x*(a/b)”，哪怕暂时不懂得其中的原因也不要紧。

上面求出了 $ax+by=\gcd(a,b)$ 的一组解 (x_1,y_1)，那么其他解呢？任取另外一组解 (x_2,y_2)，则 $ax_1+by_1=ax_2+by_2$（它们都等于 gcd(a,b)），变形得 $a(x_1-x_2)=b(y_2-y_1)$。假设 $\gcd(a,b)=g$，方程左右两边同时除以 g[①]，得 $a'(x_1-x_2)=b'(y_2-y_1)$，其中 $a'=a/g, b'=b/g$。注意，此时 a' 和 b' 互素，因此 x_1-x_2 一定是 b' 的整数倍。设它为 kb'，计算得 $y_2-y_1=ka'$。注意，上面的推导过程并没有用到“$ax+by$ 的右边是什么”，因此得出如下结论。

提示 10-1： *设 a, b, c 为任意整数。若方程 $ax+by=c$ 的一组整数解为 (x_0,y_0)，则它的任意整数解都可以写成 (x_0+kb', y_0-ka')，其中 $a'=a/\gcd(a,b), b'=b/\gcd(a,b)$，$k$ 取任意整数。*

① 如果 $g=0$，意味着 a 或 b 等于 0，可以特殊判断。

有了这个结论，我们离原题已经很近了：移项得 $ax+by=-c$，然后求出一组解即可。如何求一组解呢？举两个例子就知道了。

例 1：$6x+15y=9$。根据欧几里德算法，我们已经得到了 $6\times(-2)+15\times1=3$，两边同时乘以 3 得 $6\times(-6)+15\times3=9$，即 $x=-6, y=3$ 是 $6x+15y=9$。

例 2：$6x+15=8$，两边除以 3 得 $2x+5=8/3$。左边是整数，右边不是整数，显然无解。综合起来，我们有下面的结论。

提示 10-2：*设 a, b, c 为任意整数，$g=\gcd(a,b)$，方程 $ax+by=g$ 的一组解是 (x_0,y_0)，则当 c 是 g 的倍数时 $ax+by=c$ 的一组解是 $(x_0c/g, y_0c/g)$；当 c 不是 g 的倍数时无整数解。*

这样，我们完整地解决了本问题。顺便说一句，本题的名称为什么叫“直线上的点”呢？这是因为在平面坐标系下，$ax+by+c=0$ 是一条直线的方程。

10.1.4 同余与模算术

你需要花多少时间做下面这道题目呢？

123456789*987654321=（　　）

A. 121932631112635266　　B. 121932631112635267

C. 121932631112635268　　D. 121932631112635269

既然是选择题，不必费力把答案完整地计算出来——4 个选项的个位数都不相同，因此只需要计算出答案的最后一位即可。不难得出，它等于 $1\times9=9$。把刚才的解题过程抽象出来就是下面的式子：

$$123456789\times987654321 \bmod 10 = ((123456789 \bmod 10)\times(987654321 \bmod 10)) \bmod 10$$

其中 $a \bmod b$ 表示 a 除以 b 的余数，C 语言表达式是 a % b。在本章中，b 一定是正整数，尽管 $b<0$ 时表达式 a % b 也是合法的（但 $b=0$ 时会出现除零错）。

不难得到下面的公式：

$$(a+b) \bmod n = ((a \bmod n)+(b \bmod n)) \bmod n$$
$$(a-b) \bmod n = ((a \bmod n)-(b \bmod n)+n) \bmod n$$
$$ab \bmod n = (a \bmod n)(b \bmod n) \bmod n$$

注意在减法中，由于 $a \bmod n$ 可能小于 $b \bmod n$，需要在结果加上 n，而在乘法中，需要注意 $a \bmod n$ 和 $b \bmod n$ 相乘是否会溢出。例如当 $n=10^9$ 时，$ab \bmod n$ 一定在 int 范围内，但 $a \bmod n$ 和 $b \bmod n$ 的乘积可能会超过 int。需要用 long long 保存中间结果，像这样：

```
int mul_mod(int a, int b, int n)
{
  a %= n; b %= n;
  return (int)((long long)a * b % n);
}
```

当然，如果 n 本身超过 int 但又在 long long 范围内，上述方法就不适用了。在这种情况下，建议初学者使用高精度乘法——尽管有办法可以避免，但技巧性很强，不推荐初学者学习。

例题 10-1　大整数取模

输入正整数 n 和 m，输出 $n \bmod m$ 的值。$n \leqslant 10^{100}$, $m \leqslant 10^9$。

【分析】

首先，把大整数写成“自左向右”的形式：1234= ((1*10+2)*10+3)*10+4，然后用前面的公式，每步取模，像这样：

```
#include<stdio.h>
int main()
{
    scanf("%s%d",n, &m);
    int len = strlen(n);
    int ans = 0;
    for(int i = 0; i < len; i++)
        ans = (int)(((long long)ans*10 + n[i]) % m);
    printf("%d\n",ans);
}
```

当然了，也可以把 ans 声明成 long long 类型的，然后在输出时临时转换为 int，但千万要注意乘法溢出的问题。

例题 10-2　幂取模

输入正整数 a、 n 和 m，输出 $a^n \bmod m$ 的值。$a, n, m \leqslant 10^9$。

【分析】

很容易写出下面的代码：

```
int pow_mod(int a, int n, int m)
{
  int ans = 1;
  for(int i = 0; i < n; i++) ans = (int)((long long)ans * n % m);
}
```

这个函数的时间复杂度为 $O(n)$，当 n 很大时速度很不理想。有没有办法算得更快呢？可以利用分治法：

```
int pow_mod(int a, int n, int m)
{
  int x = pow_mod(a, n/2, m);
  long long ans = (long long)x * x % m;
  if (n%2 == 1) ans = ans * a % m;
  return (int)ans;
}
```

例如，$a^{29}=(a^{14})^2 \times a$，而 $a^{14}=(a^7)^2$, $a^7=(a^3)^2 \times a$, $a^3=a^2 \times a$，一共只做了 7 次乘法。不知你有没有发现，上述递归方式和二分查找很类似——每次规模近似减小一半。因此，时间复杂度为 $O(\log n)$，比 $O(n)$好了很多。

例题 10-3　模线性方程

输入正整数 a, b, n，解方程 $ax \equiv b (\bmod n)$。$a, b, n \leqslant 10^9$。

【分析】

本题中出现了一个新记号：同余。$a \equiv b(\bmod n)$ 的含义是“a 和 b 关于模 n 同余”，即 $a \bmod n = b \bmod n$。不难得出，$a \equiv b(\bmod n)$ 的充要条件是：$a-b$ 是 n 的整数倍。

提示 10-3： *$a \equiv b(\bmod n)$ 的含义是“a 和 b 除以 n 的余数相同”，它的充要条件是“$a-b$ 是 n 的整数倍”。*

这样，原来的方程就可以理解成：$ax-b$ 是 n 的正整数倍。设这个“倍数”为 y，则 $ax-b=ny$，移项得 $ax-ny=b$，这恰好就是 10.1.3 节介绍的不定方程（a, n, b 是已知量，x 和 y 是未知数）！接下来的步骤就不必多说了吧。唯一需要说明的是，如果 x 是方程的解，满足 $x \equiv y(\bmod n)$ 的其他整数 y 也是方程的解。因此，当谈到同余方程的一个解时，其实指的是一个同余等价类。

尽管算法已无须继续讨论，有一个特殊情况需要引起读者重视。$b=1$ 时，$ax \equiv 1(\bmod n)$ 的解称为 a 关于模 n 的逆（inverse），它类似于实数运算中“倒数”的概念。什么时候 a 的逆存在呢？根据上面的讨论，方程 $ax-ny=1$ 要有解。这样，1 必须是 $\gcd(a,n)$ 的倍数，因此 a 和 n 必须互素（即 $\gcd(a,n)=1$）。在满足这个条件的前提下，$ax \equiv 1(\bmod n)$ 只有唯一解（别忘了，同余方程的解是指一个等价类）。

提示 10-4： *方程 $ax \equiv 1(\bmod n)$ 的解称为 a 关于模 n 的逆。当 $\gcd(a,n)=1$ 时，该方程有唯一解；否则，该方程无解。*

10.2 排列与组合

排列与组合是最基本的计数技巧。本节介绍一些基本的相关知识和方法，供读者参考。

10.2.1 杨辉三角与二项式定理

组合数 C_n^m 在组合数学中占有重要地位。与组合数相关的最重要的两个东西是：杨辉三角和二项式定理。下面就是一个杨辉三角：

```
                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
```

另一方面，把 $(a+b)^n$ 展开，将得到一个关于 x 的多项式：

$$(a+b)^0=1$$
$$(a+b)^1=a+b$$
$$(a+b)^2=a^2+2ab+b^2$$
$$(a+b)^3=a^3+3a^2b+3ab^3+b^3$$
$$(a+b)^4=a^4+4a^3b+6a^2b^2+4ab^3+b^4$$

系数正好和杨辉三角一致。一般地，我们有二项式定理：

$$(a+b)^n=\sum_{k=0}^{n}C_n^k a^{n-k}b^k$$

这不难理解：$(a+b)^n$ 是 n 个括号连乘，每个括号里任选一项乘起来都会对最后的结果有一个贡献。如果选了 k 个 a，就一定会选 $n-k$ 个 b，最后的项自然就是 $a^{n-k}b^k$。而从 n 个 a 里选 k 个（同时也相当于 n 个 b 里选 $n-k$ 个）有 C_n^k 种方法，这也是组合数的定义。

给定 n，如何求出 $(a+b)^n$ 中所有项的系数呢？一个方法是用递推，根据杨辉三角中不难发现的规律，可以写出如下程序：

```
memset(C, 0, sizeof(C));
for(int i = 0; i <= n; i++)
{
  C[i][0] = 1;
  for(int j = 1; j <= n; j++) C[i][j] = C[i-1][j-1] + C[i-1][j];
}
```

但遗憾的是，这个算法的时间复杂度是 $O(n^2)$——尽管我们只用得上杨辉三角的第 n 行的 $n+1$ 个元素，却把全部 n 行的 $O(n^2)$ 个元素都计算了一遍。

另一个方法是利用等式 $C_n^k=\frac{n-k+1}{k}C_n^{k-1}$，从 $C_n^0=1$ 开始从左到右递推，像这样：

```
C[0] = 1;
for(int i = 1; i <= n; i++) C[i] = C[i-1]*(n-i+1)/i;
```

注意，应该先乘后除，因为 C[i-1]/i 可能不是整数。但这样一来增加了溢出的可能性——即使最后结果在 int 或 long long 范围之内，乘法也可能溢出。如果担心这样的情况出现，可以先约分，不过一般来说是不必要的。

顺便说一句，尽管等式 $C_n^k=\frac{n-k+1}{k}C_n^{k-1}$ 的“实际意义”不是很明显，却很容易用组合数公式 $C_n^k=\frac{n!}{k!(n-k)!}$ 证明它，读者不妨一试。

例题 10-4　无关的元素

对于给定的 n 个数 $a_1, a_2, \cdots, a_n$，依次求出相邻两数之和，将得到一个新数列。重复上述操作，最后结果将变成一个数。问这个数除以 m 的余数与哪些数无关？例如 $n=3, m=2$ 时，第一次求和得到 a_1+a_2, a_2+a_3，再求和得到 $a_1+2a_2+a_3$，它除以 2 的余数和 a_2 无关。$1\leqslant n\leqslant 10^5$，$2\leqslant m\leqslant 10^9$。

【分析】

显然最后的求和式是 $a_1,a_2,\cdots,a_n$ 的线性组合。设 a_i 的系数为 $f(i)$，则和式除以 m 的余数与 a_i 无关当且仅当 $f(i)$ 是 i 的倍数。我们不妨看一个简单的例子：

$$\begin{array}{ccccc} a_1 & a_2 & a_3 & a_4 & a_5 \\ a_1+a_2 & a_2+a_3 & a_3+a_4 & a_4+a_5 & \\ a_1+2a_2+a_3 & a_2+2a_3+a_4 & a_3+2a_4+a_5 & & \\ a_1+3a_2+3a_3+a_4 & a_2+3a_3+3a_4+a_5 & & & \\ a_1+4a_2+6a_3+4a_4+a_5 & & & & \end{array}$$

看到最后的结果，你想到了什么？没错，“1 4 6 4 1”正是杨辉三角的第 5 行！不难证明，在一般情况下，最后 a_i 的系数是 C_{n-1}^{i-1}。这样，问题就变成了：C_{n-1}^0、C_{n-1}^1、…、C_{n-1}^{n-1} 中有哪些是 m 的倍数。

还记得二项式展开的方法吗？理论上，利用此方法可以递推出所有 C_{n-1}^{i-1}，但它们太大了，必须用高精度才能存得下。我们所关心的只是“哪些是 m 的倍数”，受到数论部分中的启发，只需要依次计算 m 的唯一分解式中各个素因子在 C_{n-1}^{i-1} 中的指数即可完成判断。这些指数仍然可以用 $C_n^k=\frac{n-k+1}{k}C_n^{k-1}$ 递推，并且不会涉及高精度。有的读者可能会尝试直接递推每个系数除以 m 的余数，但遗憾的是，递推式中有除法，而模 m 意义下的逆并不一定存在。

10.2.2 数论中的计数问题

例题 10-5 约数的个数

给出正整数 n 的唯一分解式 $n=p_1^{a_1}p_2^{a_2}p_3^{a_3}\cdots p_k^{a_k}$，求 n 的正约数的个数。

【分析】

不难看出，n 的任意正约数也只能包含 p_1,p_2,p_3 等素因子，而不能有新的素因子出现。对于 n 的某个素因子 p_i，它在所求约数中的指数可以是 0, 1, 2,…, a_i 共 a_i+1 种情况，而且不同的素因子之间相互独立。根据乘法原理，n 的正约数个数为：

$$\prod_{i=1}^{k}(a_i+1)=(a_1+1)(a_2+1)\cdots(a_k+1)$$

例题 10-6 小于 n 且与 n 互素的个数

给出正整数 n 的唯一分解式 $n=p_1^{a_1}p_2^{a_2}p_3^{a_3}\cdots p_k^{a_k}$，求 1, 2, 3,…, n 中与 n 互素的数的个数。

【分析】

用容斥原理。首先从总数 n 中分别减去是 $p_1,p_2,\cdots,p_k$ 的倍数的个数（对于素数 p 来说，“与 p 互素”和“不是 p 的倍数”等价），即 $n-\frac{n}{p_1}-\frac{n}{p_2}-\cdots-\frac{n}{p_k}$，然后加上“同时是两个素因子的倍数”的个数 $\frac{n}{p_1p_2}+\frac{n}{p_1p_3}+\cdots+\frac{n}{p_{k-1}p_k}$，再减去“同时是 3 个素因子的倍数”——写成一个“学术味比较浓”的公式就是：

$$\varphi(n)=\sum_{S\subseteq\{p_1,p_2,\cdots,p_k\}}(-1)^{|S|}\frac{n}{\prod_{p_i\in S}p_i}$$

这里引入的新记号 $\varphi(n)$ 就是题目中所求的结果，称为欧拉函数。强烈建议初学者花一些时间理解这个公式。对于 $\{p_1, p_2, \cdots, p_k\}$ 的任意子集 S，“不与其中任何一个互素”的元素个数是 $\frac{n}{\prod_{p_i\in S}p_i}$。不过这一项的前面是加号还是减号呢？这取决于 S 中的元素个数——奇数个就是“减号”，偶数个就是“加号”。

公式倒是出来了，可计算起来很不方便。如果直接根据公式，需要计算多达 2^k 项的代数和，甚至可能比“暴力枚举（依次判断 1～n 中每个数是否与 n 互素）”还要慢。

下一步并不显然。上述公式可以变形成如下的形式：

$$\varphi(n)=n(1-\frac{1}{p_1})(1-\frac{1}{p_2})\cdots(1-\frac{1}{p_k})$$

从而只需要 $O(k)$ 的计算时间，在刚才的基础上大大提高了效率。为什么这个式子和上一个等价呢？直接考虑新公式的“展开方式”即可。展开式的每一项是从每个括号各选一个（选 1 或者 $-\frac{1}{p_i}$），全部乘起来以后再乘以 n 得到。这不正是最初的推导过程吗？

如果没有给出唯一分解式，需要用试除法依次判断 $\sqrt{n}$ 内的所有素数是否是 n 的因子。这样的话，需要先生成 $\sqrt{n}$ 内的素数表。但其实并不用这么麻烦：只需要每次找到一个素因子之后把它“除干净”，即可保证找到的因子都是素数（想一想，为什么）。

```
int euler_phi(int n)
{
  int m = (int)sqrt(n+0.5);
  int ans = n;
  for(int i = 2; i <= m; i++) if(n % i == 0)
  {
     ans = ans / i * (i-1);
     while(n % i == 0) n /= I;
  }
  if(n > 1) ans = ans / n * (n-1);
}
```

另外，如果需要求出 1～n 中所有数的欧拉函数值，并不需要依次计算。可以用与筛法求素数非常类似的方法，在 $O(n\log\log n)$ 时间内计算完毕，像这样：

```
void phi_table(int n, int* phi)
{
  for(int i = 2; i <= n; i++) phi[i] = 0;
  phi[1] = 1;
  for(int i = 2; i <= n; i++) if(!phi[i])
    for(int j = i; i <= n; j += i)
```

```
    {
      if(!phi[j]) phi[j] = j;
      phi[j] = phi[j] / i * (i-1);
    }
  }
```

10.2.3 编码与解码

两个 a、一个 b 和一个 c 组成的所有串可以按照字典序编号为：

$$aabc(1)、aacb(2)、abac(3)、\cdots、cbaa(12)$$

任给一个字符串，能否方便地求出它的编号呢？例如，输入 *acab*，则应输出 5。

我们直接求解一般情况的问题（并不限定字母的种类和个数）。设输入串为 S，记 $d(S)$ 为 S 的各个排列中，字典序比 S 小的串的个数，则可以用递推法求解 $d(S)$，如图 10-1 所示。

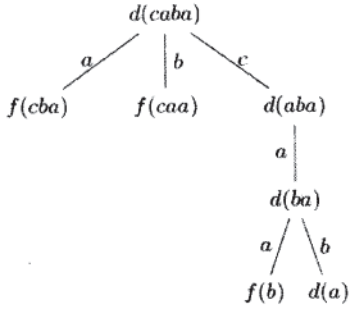


图 10-1　字符串编码的递推过程

其中边上的字母表示“下一个字母”，$f(x)$表示多重集 x 的全排列个数。例如，根据第一个字母，可以把字典序小于 *caba* 的字符串分为 3 种：以 a 开头的，以 b 开头的，以 c 开头的，分别对应 $d(caba)$的 3 棵子树。以 a 开头的所有串的字典序都小于 *caba*，所以剩下的字符可以任意排列，个数为 $f(cba)$；同理，以 b 开头的所有串的字典序也都小于 *caba*，个数为 $f(caa)$；以 c 开头的串字典序不一定小于 *caba*，关键要看后 3 个字符，因此这部分的个数为 $d(aba)$，还需要继续往下分。

至于 f 函数的求解，大部分组合数学书籍中均有介绍：设字符一共有 k 类，个数分别为 $n_1, n_2, \cdots, n_k$，则这个多重集的全排列个数为 $\dfrac{(n_1+n_2+\cdots+n_k)!}{n_1!n_2!\cdots n_k!}$。

不难算出，$f(caa)=\dfrac{(1+2)!}{1!2!}=\dfrac{6}{2}=3$，其他 f 值分别为 $f(cba)=6$，$f(b)=1$，故 $d(caba)=f(cba)+f(caa)+f(b)=3+6+1=10$。既然“比它小”的个数是 10，序号自然就是 11 了。

“给物体一个编号”称为编码，同理也有“解码”，即根据序号构造出这个物体。这个过程和刚才的很接近：依次确定各个位置上的字母即可。例如，要求出序号为 8（因此有 7 个比它小）的字符串，推理过程如图 10-2 所示。

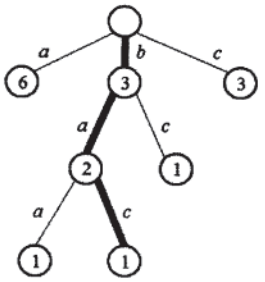


图 10-2　字符串解码的递推过程

10.2.4　离散概率初步

概率有一套很深的理论，不过很多和概率相关的问题并不需要特别的知识，熟悉排列组合就够了。

第 1 个例子是：连续抛 3 次硬币，恰好有两次正面的概率是多少？我们用 H 和 T 来表示正面和背面（取自英文单词 head 和 tail），则一共有 8 种可能的情况：HHH、HHT、HTH、HTT、THH、THT、TTH、TTT。根据我们对硬币的认识，这 8 种情况出现的可能性相同，概率各为 1/8。用概率论的专业术语说，这里的{HHH、HHT、HTH、HTT、THH、THT、TTH、TTT}称为样本空间（Sample Space）。我们所求的是“恰好有两次正面”这个事件（Event）的概率。借助于集合的记号，这个事件可以表示为{HHT, HTH, THH}，它的概率为 3/8。

提示 10-5：如果样本空间由有限个等概率的简单事件组成，事件 E 的概率可以用组合计数的方法得到——$P(E)=\dfrac{|E|}{|S|}$。

第 2 个例子是：如果一间屋子里有 23 个人，那么“至少有两个人的生日相同”的概率超过 50%。为了简单起见，假定已知每个人的生日都不是 2 月 29 日。

尽管看上去复杂了许多，其实这个例子和抛硬币是类似的。每个人的生日是 365 天中等概率随机选择的，因此样本空间大小 $|S|=365^{23}$。接下来需要计算“至少有两个人生日相同”的情况有多少种。这个数目不太好直接统计，所以统计“任何两个人的生日都不相同”的数目，然后用总数减去它即可。公式不难得到：

$$P(E)=\frac{|E|}{|S|}=\frac{|S|-|\bar{E}|}{|S|}=1-\frac{P_{365}^{23}}{365^{23}}$$

不管是 C_{365}^{23} 还是 365^{23} 都无法储存在 int 或者 long long 中，但好在概率是实数，并且我们并不需要太高的精度，所以可以直接计算，像这样：

```
double P(int n, int m)
{
  double ans = 1.0;
  for(int i = 0; i < m; i++) ans *= (n-i);
  return ans;
```

```
}

double birthday(int n, int m)
{
  double ans = P(n, m);
  for(int i = 0; i < m; i++) ans /= n;
  return 1 - ans;
}
```

函数 birthday(365,23)的返回值为 0.5073，即 50.73%。别高兴得太早，我们来算一算 birthday(365,365)。直观上，365 个人中几乎肯定会有两个人的生日相同，因此 birthday(365,365)应该返回一个很接近 1 的值。可结果呢？很不幸，返回值为-1.#INF0000——连 double 都溢出了。

解决方案是边乘边除，而不是连着乘 m 次，然后再连着除 m 次。像这样：

```
double birthday(int n, int m)
{
  double ans = 1.0;
  for(int i = 0; i < m; i++) ans *= (n-i) / n;
  return 1 - ans;
}
```

这个例子告诉我们：正如数论和组合计数中要注意 int 和 long long 溢出一样，在概率计算中要注意 double 溢出。顺便说一句，这个“改进版”程序其实有个直接的概率意义：

$$P(E)=1-P(\bar{E})=1-P(E_1)P(E_2)P(E_3)\cdots P(E_m)=1-\frac{n}{n}\times\frac{n-1}{n}\times\frac{n-2}{n}\cdots\frac{n-(m-1)}{n}$$

其中，E_i 表示“第 i 个人的生日不和前面的人重复”这个事件。

10.3 递 推 关 系

尽管排列组合能解决很多计数问题，仍然有很多情况需要用到其他方法。在这些方法中，递推关系是最重要的一种。

10.3.1 汉诺塔

假设有 A、B、C 3 个轴，有 n 个直径各不相同、从小到大依次编号为 1, 2, 3,…,n 的圆盘按照上小下大的顺序叠放在 A 轴上。现要求将这 n 个圆盘移至 B 轴上并仍按同样顺序叠放，但圆盘移动时必须遵循下列规则：

- 每次只能移动一个圆盘，它必须位于某个轴的顶部。
- 圆盘可以插在 A、B、C 中的任一轴上。
- 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

【分析】

这个问题看上去很容易，但当 n 稍大一点时，手工移动就开始变得困难起来。下面直接给出递归解法：首先，把前 $n-1$ 个圆盘放到 C 轴；接下来把 n 号圆盘放到 B 轴；最后，再把前 $n-1$ 个盘子放到 B 轴，如图 10-3 所示。

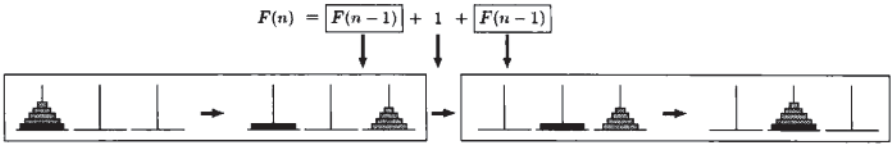


图 10-3　根据递归解法建立汉诺塔的递推关系

图 10-3 还给出了 n 个圆盘所需步数 $f(n)$的递推式：$f(n)=2f(n-1)+1$。如果把 $f(n)$的值从小到大列出来——1、3、7、15、31、63、127、255…，你会发现其实有一个简单的表达式：$f(n)=2^n-1$。

用数学归纳法不难证明：$f(1)=1$ 满足等式。假设 $n=k$ 满足等式，即 $f(k)=2^k-1$，则 $n=k+1$ 时，$f(k+1)=2f(k)+1=2(2^k-1)+1=2^{k+1}-2+1=2^{k+1}-1$。因此 $n=k+1$ 也满足等式。由数学归纳法可知，n 取任意正整数均成立。

如果还不熟悉数学归纳法，其实从上面的证明过程已经能看出来其基本原理——其实它正是一种递归证明。只要边界处理好（$f(1)=1$ 满足），递归时缩小规模（用 k 来证明 $k+1$），然后在“相信递归”（假设 $n=k$ 成立）的前提下证明即可。

提示 10-6：*数学归纳法是一种利用递归的思想证明的方法。如果要讨论的对象具有某种递归性质（如正整数），可以考虑用数学归纳法。*

10.3.2　Fibonacci 数列

先来考虑一个简单的问题：楼梯有 n 个台阶，上楼可以一步上 1 阶，也可以一步上 2 阶。一共有多少种上楼的方法？

这是一道计数问题。在没有思路时，不妨试着找规律。$n=5$ 时，一共有 8 种方法：

```
5=1+1+1+1+1
5=2+1+1+1
5=1+2+1+1
5=1+1+2+1
5=1+1+1+2
5=2+2+1
5=2+1+2
5=1+2+2
```

其中有 5 种方法第 1 步走了 1 阶（灰色），3 种方法第 1 步走了 2 阶。没有其他可能了。假设 $f(n)$为 n 个台阶的走法总数，把 n 个台阶的走法分成两类。

第 1 类：第 1 步走 1 阶。剩下还有 $n-1$ 阶要走，有 $f(n-1)$种方法。

第 2 类：第 1 步走 2 阶。剩下还有 $n-2$ 阶要走，有 $f(n-2)$种方法。

这样，就得到了递推式：$f(n)=f(n-1)+f(n-2)$。不要忘记边界情况：$f(1)=1$，$f(2)=2$。当然，也可以认为边界是 $f(0)=f(1)=1$。把 $f(n)$的前几项列出：1, 1, 2, 3, 5, 8,…。

再来一个：把雌雄各一的一对新兔子放入养殖场中。每只雌兔从第 2 个月开始每月产雌雄各一的一对新兔子。试问第 n 个月后养殖场中共有多少对兔子？

我们还是先找找规律。

第一个月：一对新兔子 r_1。用小写字母表示新兔子。

第二个月：还是一对新兔子，不过已经长大，具备生育能力了，用大写字母 R_1 表示。

第三个月：R_1 生了一对新兔子 r_2，一共两对。

第四个月：R_1 又生一对 r_3，一共 3 对。另外，r_2 长大了，变成 R_2。

第五个月：R_1 和 R_2 各生一对，记为 r_4 和 r_5，共 5 对。此外，r_3 长成 R_3。

第六个月：R_1、R_2 和 R_3 各生一对，记为 r_6～r_8，共 8 对，同时 r_4 到 r_5 长大。

……

把这些数排列起来：1, 1, 2, 3, 5, 8, …，和刚才的一模一样！事实上，可以直接推导出递推关系 $f(n)=f(n-1)+f(n-2)$：第 n 个兔子由两部分组成，一部分是上个月就有的老兔子，一部分是上个月出生的新兔子。前一部分等于 $f(n-1)$，后一部分等于 $f(n-2)$（第 $n-1$ 个月时具有生育能力的兔子数就等于第 $n-2$ 个月的兔子总数）。根据加法原理，$f(n)=f(n-1)+f(n-2)$。

提示 10-7：*满足 $F_1=F_2=1$，$F_n=F_{n-1}+F_{n-2}$ 的数列称为 Fibonacci 数列，它的前若干项是* 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…。

再来一个新的：有 2 行 n 列的长方形方格，要求用 n 个 1×2 的骨牌铺满。有多少种铺法？

考虑最左边一列的铺法。如果用一个骨牌直接覆盖，则剩下的 $2\times(n-1)$方格有 $f(n-1)$种铺法；如果是用两个横向骨牌覆盖，则剩下的 $2\times(n-2)$方格有 $f(n-2)$种方法，如图 10-4 所示。不难发现：第一列没有其他铺法，因此 $f(n)=f(n-1)+f(n-2)$。边界 $f(0)=1$，$f(1)=1$，正好是 Fibonacci 数列。

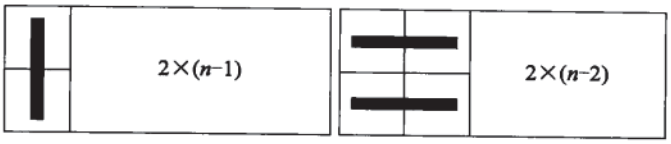


图 10-4　骨牌覆盖问题

这就是多数课本上讲解这道题目的方法，无须多说，因为重点并不在此。笔者曾想到过另一个解法，与各位读者分享：设第 i 列是纵向骨牌，则左边 $i-1$ 列和右边 $n-i$ 列各有 $f(i-1)$和 $f(n-i)$种铺法。根据乘法原理，一共有 $f(i-1)f(n-i)$种铺法。然后把 $i=1,2,3,\cdots,n$ 的情形全部加起来，根据加法原理，有：

$$f(n)=f(0)f(n-1)+f(1)f(n-2)+\cdots+f(n-1)f(0)$$

这个递推式对不对呢？聪明的读者也许已经看出，这个解法存在两个问题：

（1）有遗漏。只考虑了第 $1,2,3,\cdots,n$ 列是纵向骨牌的情形，但实际上可能所有的骨牌

都是横向的！当且仅当 n 为偶数时，恰好有一种这样的方案。

（2）有重复。根据“第 i 列有骨牌”对所有方案进行了分类，但其实这些方案是有重叠的！例如，第 1 列和第 2 列完全可以同时有骨牌。这些方案在递推式中被重复计算了。

既然如此，这个思路是不是走入死胡同了呢？不是的！只要把刚才的推理变得严密起来，我们同样可以得到一个正确的递推式：根据从左到右第一条纵向骨牌的列编号分类。如果不存在，当且仅当 n 为偶数时有一种方案；当第一条纵向骨牌的列编号为 i 时，意味着左边 $i-1$ 列必须全部是横向骨牌——当 i 为奇数时恰好有一个方案。而右边 $n-i$ 列则可以用任意铺法，共 $f(n-i)$种。换句话说：

n 为偶数时，$f(n)=f(n-1)+f(n-3)+f(n-5)+\cdots+f(1)+1$（最后加上的就是“没有纵向骨牌”的情形）；

n 为奇数时，$f(n)=f(n-1)+f(n-3)+f(n-5)+\cdots+f(2)+f(0)$。

边界是 $f(0)=f(1)=1$。我们已经知道，问题的答案应该是 Fibonacci 数列，自然会对这个复杂的递推式产生怀疑：它真的是正确的吗？

带着这个疑问，笔者写了一个程序。结果出乎我的意料：居然和 Fibonacci 数列一样！事实上，它确实是 Fibonacci 数列。Fibonacci 数列拥有很多有趣的性质，有兴趣的读者可以在网上搜索有关它的更多资料。不管怎样，这个“旧题新解”至少说明了两件事：

（1）一个数列可能有多个看上去完全不同的递推式。

（2）即使是漏洞百出的解法也有可能通过“打补丁”的方式修改正确。

10.3.3　Catalan 数

给一个凸 n 边形，用 $n-3$ 条不相交的对角线把它分成 $n-2$ 个三角形，求不同的方法数目。例如 $n=5$ 时，有 5 种剖分方法，如图 10-5 所示。

图 10-5　凸五边形的 5 种三角剖分

【分析】

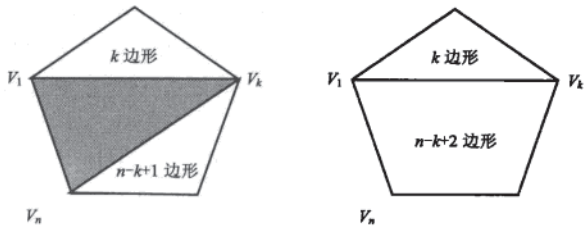
设答案为 $f(n)$。按照某种顺序给凸多边形的各个顶点编号为 $V_1,V_2,\cdots,V_n$。既然分成的是三角形，边 V_1V_n 在最终的剖分中一定恰好属于某个三角形 $V_1V_nV_k$，所以可以根据 k 进行分类。不难看出，三角形 $V_1V_nV_k$ 的左边是一个 k 边形，右边是一个 $n-k+1$ 边形（如图 10-6（a）所示）。根据乘法原理，包含三角形 $V_1V_nV_k$ 的方案数为 $f(k)f(n-k+1)$；根据加法原理有：

$$f(n)=f(2)f(n-1)+f(3)f(n-2)+\cdots+f(n-1)f(2)$$

边界是 $f(2)=f(3)=1$。不难算出从 $f(3)$开始的前几项 f 值依次为：1、2、5、14、42、132、429、1430、4862、16796。

提示 10-8：在建立递推式时，经常会用到乘法原理，它的核心是分步计数。如果可以把计数分成独立的两个步骤，则总数量等于两步计数之乘积。

另一种思路是考虑 V_1 连出的对角线。对角线 V_1V_k 把凸 n 边形分成两部分，一部分是 k 边形，另一部分是 $n-k+2$ 边形（如图 10-6（b）所示）。根据乘法原理，包含对角线 V_1V_k 的凸多边形有 $f(k)f(n-k+2)$个。根据对称性，考虑从 V_2、V_3、…、V_n 出发的对角线也会有同样的结果，因此一共有 $n(f(3)f(n-1)+f(4)f(n-2)+\cdots+f(n-1)f(3))$个剖分。



（a）　　（b）

图 10-6　凸多边形三角剖分数目的两种递推方法

但这并不是正确答案，因为同一个剖分被重复计算了多次！不过这次不必去消除重复了，因为这些重复很有规律：每个方案恰好被计算了 $2n-6$ 次——有 $n-3$ 条对角线，而考虑每条对角线的每个端点时均计算了一次。这样，得到了 $f(n)$的第 2 个递推式：

$$f(n)=(f(3)(n-1)+f(4)f(n-2)+\cdots+f(n-1)f(3))\times n/(2n-6)$$

它和第一个递推式有几分相似，但又不同。把 $n+1$ 代入第 1 个递推式后得到：

$$f(n+1)=f(2)f(n)+f(3)f(n-1)+f(4)f(n-2)+\cdots+f(n-1)f(3)+f(n)f(2)$$

灰色部分是相同的！根据第 2 个递推式，它等于 $f(n)\cdot(2n-6)/n$，把它和 $f(2)=1$ 一起代入上式得：$f(n+1)=f(n)+f(n)\cdot(2n-6)/n+f(n)=\dfrac{4n-6}{n}f(n)$。

这个递推式和前两个相比就简单多了。顺便说一句，这个数列称为 Catalan 数，也是常见的计数数列。

10.3.4　危险的组合

有一些装有铀（用 U 表示）和铅（用 L 表示）的盒子，数量均足够多。要求把 n 个盒子放成一行，但至少有 3 个 U 放在一起，有多少种放法？$n\leqslant 30$。

样例输入：4
样例输出：3
样例输入：5
样例输出：8
样例输入：30
样例输出：974791728

【分析】

设答案为 $f(n)$。既然有 3 个 U 放在一起，可以根据这 3 个 U 的位置分类——哦不对，根据前面的经验，要根据“最左边的 3 个 U”的位置分类。假定是 i、$i+1$ 和 $i+2$ 这 3 个盒

子，则前 i-1 个盒子不能有 3 个 U 放在一起的情况。设 n 个盒子“没有 3 个 U 放在一起”的方案数为 $g(n)=2^n-f(n)$，则前 i-1 个盒子的方案有 $g(i-1)$种。后面的 $n-i-2$ 个盒子可以随便选择，有 2^{n-i-2} 种。根据乘法原理和加法原理，$f(n)=\sum_{i=1}^{n-2} g(i-1)2^{n-i-2}$。

遗憾的是，这个推理是有瑕疵的。即使前 i-1 个盒子内部不出现 3 个 U，仍然可能和 i、i+1 和 i+2 组成 3 个 U！正确的方法是强制让第 i-1 个盒子（如果有）放 L，则前 i-2 个盒子内部不能出现连续的 3 个 U，因此 $f(n)=2^{n-3}+\sum_{i=2}^{n-2} g(i-2)2^{n-i-2}$，边界是 $f(0)=f(1)=f(2)=0$，$g(0)=1, g(1)=2, g(2)=4$。注意上式中的 2^{n-3} 对应于 i=1 的情况。

10.3.5　统计 n-k 特殊集的数目

如果由正整数构成的集合 X 满足以下条件，我们称它为 n-k 特殊集：

（1）集合 X 中的每个元素 x 均不超过 n，即 $1\leqslant x\leqslant n$。

（2）集合 X 中所有元素之和大于 k。

（3）集合 X 中不包含任意一对相邻的自然数。

给出 n, k，求 n-k 特殊集合有多少个。$1\leqslant n\leqslant 100$，$0\leqslant k\leqslant 400$。

样例输入：6 3

样例输出：17

样例输入：14 55

样例输出：1

样例输入：40 100

样例输出：267185615

【分析】

设 n-k 特殊集的个数为 $f(n,k)$，我们来想办法建立它的递推式。由于集合中的元素不能重复，元素 n 要么在集合中恰好出现一次，要么不出现，二者不重复、不遗漏地把 n-k 特殊集分成了两部分：

（1）n 不出现。除了规则 1 中的 n 需要修改为 n-1 外，其他均不变，因此有 $f(n-1,k)$个。

（2）n 出现一次。规则 1 中的 n 需要修改为 n-2（如果出现了 n-1，则相邻自然数 n-1、n 不满足条件），而规则 2 中的 k 应变为 k-n。

换句话说，递推关系是：$f(n,k)=f(n-1,k)+f(n-2,k-n)$。那么边界应该是什么呢？$n\leqslant 0$ 时只有空集一个集合满足规则 1，而此时所有元素和为 0，因此：

- 当 $n\leqslant 0, k\geqslant 0$ 时，$f(0,k)=0$。
- 当 $n\leqslant 0, k<0$ 时，$f(0,k)=1$。

在其他情况下，f 均能按此递推式计算。但这样一来，问题就来了：如果用 f[n][k]保存 $f(n,k)$的值，n 和 k 需要允许负数！

第 1 种处理方法是动态判断边界，而不是在初始化时一口气给全部边界赋上值。这样的话，上面的边界就不够了：尽管 $f(5,-3)$也可以按照递推式计算，但为了方便程序编写，

我们直接把$f(n,k)(k<0)$作为边界。可是，$k<0$时，$f(n,k)$应该等于多少呢？此时k已经不重要了，设$g(n)=f(n,-1)$，则可以仿照刚才的推理写出如下递推式：$g(n)=g(n-1)+g(n-2)$，边界$g(-1)=g(0)=1$。这正是Fibonacci数列！

这种方法很通用，但琐碎、不直观，而且还不得不借助一个辅助函数g。没关系，我们还有另外一种处理方法。既然$k<0$时$f(n,k)$与k无关，用-1来“代表”所有的负数。这样，边界就只剩两个了：$f(n,k)=0$ $(n=0,-1,k\geq 0)$，$f(n,-1)=1(n=0,-1)$。

可接下来又有了新问题：程序如何编写呢？别忘了，C语言是不支持负数下标的。这个问题有不少解决方案，但最简单的莫过于把所有下标全部加上1。如果觉得这样做破坏了程序的可读性，可以使用宏，像这样：

```
#include<iostream>
#define F(i,j) (f[(i)+1][(j)+1]) // 用宏处理支持负数下标
using namespace std;

bign f[200][500];                // 结果可能很大，需要用高精度
int main()
{
  int i, j, n, k;
  cin >> n >> k;
  for(j = -1; j <= k; j++)       // 边界
    F(-1,j) = F(0,j) = 0;
  F(-1,-1) = F(0,-1) = 1;
  for(i = 1; i <= n; i++)
    for(j = -1; j <= k; j++)     // 递推
    {
      F(i,j) = F(i-1,j);
      if(j-i < 0) F(i,j) += F(i-2,-1);
      else F(i,j) += F(i-2,j-i);
    }
  cout << F(n,k) << "\n";
  return 0;
}
```

注意，上面用到了第5章中介绍的高精度类bign，它为调试带来了很大的方便：可以先用int调试，正确之后直接把类型改成bign即可。

10.4 训练参考

首先是UVaOJ中的题目：街道数（138）、Carmichael数（10006）、软件CRC（128）、Fermat与Pythagoras（106）、Floor和Ceil（10673）、-2进制（11121）、总和最小的LCM（10791）、硬币做桌腿（10717）、交表（10820）、水壶（571）、头和尾（11029）、平

方根（10023）、淘气的孩子（10308）、多项式系数（10105）、选择与除法（10375）、回文排列（11027）、概率是多少？（10056）、牛和车（10491）、掷色子（10759）、法国世界杯'98（542）、炫耀红白袜子（10277）、缸和球的概率问题（10169）、条件概率（11181）、汉堡（557）、相当 2" 元富翁吗？（10900）、连续获胜（11176）、交换礼物（10417）、多少个 Fib（10183）、多少棵树（10303）、多少次调用（10518）、连接电缆（10862）、玻璃中的光线反射（10334）、模 Fibonacci（10229）、Fibonacci 进制（763）、Fibonacci 素数（10236）、安全的问候（991）、切 Pizza（10079）、把卡片扔掉 II（10940）、堆砌（10359）、真奇怪（10519）、三行铺砖（10918）、一个图论问题（11069）、分数分布（10910）、牧师数学家（10254）、抛硬币（10328）、表达式（10157）、完全树标号（10247）、又一个计数问题（10516）、队列（10128）、紧密的单词（10081）、如何做加法（10943）、条形码（10721）、简单的哈希（10912）、小组之和的整除性（10616）、大理石重排（11125）、把它们叠起来（10205）、地质学家的困境（701）、多少个骑士（696）、汉诺塔（254）、简单加法（10994）、加密（306）、会见外星人（10570）。

接下来是 CII 的：不同的数字（3262）、不可约分数（4406）、计算机变换（3308）、变形 Joseph 问题（3882）、磁列车轨道（4064）、灯塔（4004）、污水处理车间（2685）、押韵（2871）、装饰（2825）、格斗者（2933）、翻转和移位（2325）、二进制 Stirling 数（2431）、Vivian 的问题（2955）、可见的格点（3571）、树的排序（2357）。

第 11 章　图论模型与算法

学习目标

- ☑ 掌握无根树的常用存储法和转化为有根树的方法
- ☑ 掌握由表达式构造表达式树的算法
- ☑ 掌握 Kruskal 算法及其正确性证明，并用并查集实现
- ☑ 掌握基于优先队列的 Dijkstra 算法实现
- ☑ 掌握基于 FIFO 队列的 Bellman-Ford 算法实现
- ☑ 掌握 Floyd 算法和传递闭包的求法
- ☑ 理解最大流问题的概念、流量的 3 个条件、残量网络的概念和求法
- ☑ 理解增广路定理与最小割最大流定理的证明方法，会实现 Edmonds-Karp 算法
- ☑ 理解最小费用最大流问题的概念，以及平行边和反向弧可能造成的问题
- ☑ 会实现基于 Bellman-Ford 的最小费用路算法

本章介绍一些常见的图论模型和算法，包括最小生成树、单源最短路、每对结点的最短路、最大流、最小费用最大流等。限于篇幅，很多算法都没有给出完整的正确性证明（很容易在其他参考资料中找到相关内容），但给出了简单、易懂的完整代码，方便读者参考。

11.1　再　谈　树

在第 6 章中，我们第一次接触到二叉树；后来，又接触到了其他树状结构，如解答树、BFS 树。本节将继续讨论“树”这一话题。

有 n 个顶点的树具有以下 3 个特点：连通、不含圈、恰好包含 $n-1$ 条边。有意思的是，具备上述 3 个特点中的任意两个，就可以推导出第 3 个，有兴趣的读者不妨试着证明一下。

11.1.1　无根树转有根树

输入一个 n 个结点的无根树的各条边，并指定一个根结点，要求把该树转化为有根树，输出各个结点的父亲编号。$n \leqslant 10^6$，如图 11-1 所示。

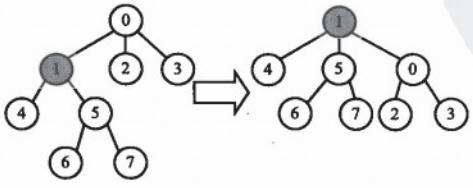


图 11-1　无根树转有根树

【分析】

树是一种特殊的图，因此很容易想到用邻接矩阵表示。可惜，n 个结点的图对应的邻接矩阵要占用 n^2 个元素的空间，开不下。

实际上，n 个结点的树只有 $n-1$ 条边，是否存在一个像邻接矩阵一样好用，但空间开销很小的数据结构呢？答案是肯定的。

```
vector<int> G[maxn];
void read_tree()
{
  int u, v;
  scanf("%d", &n);
  for(int i = 0; i < n-1; i++)
  {
    scanf("%d%d", &u, &v);
    G[u].push_back(v);
    G[v].push_back(u);
  }
}
```

vector 是 STL 中的可变长数组，因此 G 是一个“包含 maxn 行，但每行长度可以不同”的“二维数组”。结点 u 的所有相邻点都放在 G[u]中，用 G[u].size()获取 u 的相邻点个数，G[u][i]表示其中第 i 个相邻点。由于 vector 是变长的，这个“二维数组”占用的空间是 $O(n)$，而非 $O(n^2)$。

可以把 vector 理解成 STL 中的“超级数组”。它不仅可以用下标访问，还可以用 resize()改变大小，或者用 push_back()在数组尾部添加新元素。尽管 vector 还有一些普通数组不能使用的其他方法（如 push_front()、insert()、erase()），但它们都涉及大量元素移动，须慎用。

下面是转化过程：

```
void dfs(int u, int fa)           // 递归转化以 u 为根的子树，u 的父亲为 fa
{
  int d = G[u].size();            // 结点 u 的相邻点个数
  for(int i = 0; i < d; i++)
  {
    int v = G[u][i];              // 结点 u 的第 i 个相邻点 v
    if(v != fa) dfs(v, p[v] = u); // 把 v 的父亲设为 u，然后递归转化以 v 为根的子树
  }
}
```

主程序中设置 p[root]=−1（表示根结点的父亲不存在），然后调用 dfs(root, −1)即可。初学者最容易犯的错误之一就是忘记判断 v 是否和其父亲相等。如果忽略，将引起无限递归。

11.1.2　表达式树

二叉树是表达式处理的常用工具。例如，a+b*(c−d)−e/f 可以表示成如图 11-2 所示的二

叉树。

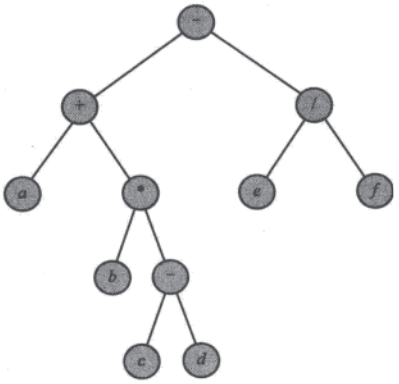


图 11-2　表达式树

其中，每个非叶结点表示一个运算符，左子树是第 1 个运算数对应的表达式，而右子树则是第 2 个运算数对应的表达式。如何给一个表达式建立表达式树呢？方法有很多，这里只介绍一种：找到“最后计算”的运算符（它是整棵表达式树的根），然后递归处理。下面是程序：

```
const int maxn = 1000;
int lch[maxn], rch[maxn]; char op[maxn];   // 每个结点的左右儿子编号和字符
int nc = 0;          // 结点数

int build_tree(char* s, int x, int y)
{
  int i, c1=-1, c2=-1, p=0;
  int u;
  if(y-x == 1)       // 仅一个字符，建立单独结点
  {
    u = ++nc;
    lch[u] = rch[u] = 0; op[u] = s[x];
    return u;
  }
  for(i = x; i < y; i++)
  {
    switch(s[i])
    {
      case '(': p++; break;
      case ')': p--; break;
      case '+': case '-': if(!p) c1=i; break;
      case '*': case '/': if(!p) c2=i; break;
    }
```

```
    }
    if(c1 < 0) c1 = c2;    // 找不到括号外的加减号，就用乘除号
    if(c1 < 0) return build_tree(s, x+1, y-1);    // 整个表达式被一对括号括起来
    u = ++nc;
    lch[u] = build_tree(s, x, c1);
    rch[u] = build_tree(s, c1+1, y);
    op[u] = s[c1];
    return u;
}
```

注意上述代码是如何寻找“最后一个运算符”的。代码里用了一个变量 p，只有当 p=0 时才考虑这个运算符。为什么呢？因为括号里的运算符一定不是最后计算的，应当忽略。例如(a+b)*c 中虽然有一个加号，但却是在括号里的，实际上比它优先级高的乘号才是最后计算的。由于加减和乘除号都是左结合的，最后一个运算符才是最后计算的，所以用两个变量 c1 和 c2 分别记录“最右”出现的加减号和乘除号。

再接下来的代码就不难理解了：如果括号外有加减号，它们肯定最后计算；但如果没有加减号，就需要考虑乘除号（if(c1<0) c1 = c2）；如果全都没有，说明整个表达式外面被一对括号括起来，把它去掉后递归调用。这样，就找到了最后计算的运算符 s[c1]，它的左子树是区间[x, c1]，右区间是[c1+1, y]。

提示 11-1：建立表达式树的一种方法是每次找到最后计算的运算符，然后递归建树。“最后计算”的运算符是在括号外的、优先级最低的运算符。如果有多个，根据结合性来选择：左结合的（如加、减、乘、除）选最右边；右结合的（如乘方）选最左边。根据规定，优先级相同的运算符的结合性总是相同。

11.1.3　最小生成树

航海家们在太平洋上发现了几座新岛屿，其中最大的一个岛（称为主岛）已经连接到 Internet，但是其他岛和主岛之间没有电缆连接，所以无法入网。我们的目的是让所有岛上的居民都能上网，即每个岛和主岛之间都有直接或间接的电缆连接。

如果要直接连接两个岛屿，所需的电缆长度等于两个岛屿中心位置的几何距离。为了节省成本，希望所有电缆的总长度尽量小。应该怎样连接呢？

【分析】

本题的目的是要选出一些边，要求所有点连通（所有居民都能上网），并且权和尽量小。由于“长度”一定是正数，因此要使总长度尽量小，所选择的边不能含有圈——否则，在圈上任意去掉一条边，所有点仍然连通，而且总长度变小。在无向图中，连通且不含圈的图称为树（Tree）。给定无向图 $G=(V,E)$，连接 G 中所有点，且边集是 E 的子集的树称为 G 的生成树（Spanning Tree），而权值最小的生成树称为最小生成树（Minimal Spanning Tree，MST）。本题的目标就是求出一棵最小生成树。

构造 MST 的算法有很多，最常见的有两个：Kruskal 算法和 Prim 算法。限于篇幅，这里只介绍 Kruskal 算法，它易于编写，而且效率很高。算法的第一步是给所有边按照从小到大的

顺序排列。这一步可以直接使用库函数 qsort 或者 sort。接下来从小到大依次考查每条边(*u*,*v*)。

情况 1：*u* 和 *v* 在同一个连通分量中，那么加入(*u*, *v*)后会形成环，因此不能选择。

情况 2：如果 *u* 和 *v* 在不同的连通分量，那么加入(*u*, *v*)一定是最优的。为什么呢？我们用反证法——如果不加这条边能得到一个最优解 *T*，则 *T'*+(*u*, *v*)一定有且只有一个环，而且环中至少有一条边(*u'*, *v'*)的权值大于或等于(*u*,*v*)的权值。删除该边后，得到的新树 *T'*=*T*+(*u*, *v*)-(*u'*, *v'*)不会比 *T* 更差。因此，加入(*u*, *v*)不会比不加入差！

下面是伪代码：

```
把所有边排序，记第 i 小的边为 e[i] (1<=i<m)
初始化 MST 为空
初始化连通分量，让每个点自成一个独立的连通分量
for(int i = 0; i < m; i++)
    if(e[i].u 和 e[i].v 不在同一个连通分量)
    {
        把边 e[i]加入 MST
        合并 e[i].u 和 e[i].v 所在的连通分量
    }
```

在上面的伪代码中，最关键的地方在于“连通分量的查询与合并”：我们需要知道任何两个点是否在同一个连通分量中，还需要合并两个连通分量。

最容易想到的方法是“暴力”——每次“合并”时只在 MST 中加入一条边（使用邻接矩阵的话，只需 G[e[i].u][e[i].v]=1），而“查询”时直接在 MST 中进行图遍历（还记得吗？DFS 和 BFS 都可以判断连通性）。遗憾的是，这个方法不仅复杂（需要写 DFS 或者 BFS），而且效率不高。

11.1.4 并查集

有一种简洁高效的方法可用来处理这个问题：使用并查集（Union-Find Set）。我们可以把每个连通分量看成一个集合，该集合包含了连通分量中的所有点。这些点两两连通，而具体的连通方式无关紧要，就好比集合中的元素没有先后顺序之分，只有“属于”和“不属于”的区别。在图中，每个点恰好属于一个连通分量，对应到集合表示中，每个元素恰好属于一个集合。换句话说，图的所有连通分量可以用若干个不相交集合来表示。

并查集的精妙之处在于用树来表示集合。例如，若包含点 1，2，3，4，5，6 的图有 3 个连通分量{1,3}、{2,5,6}、{4}，则需要用 3 棵树来表示。这 3 棵树的具体形态无关紧要，只要有一棵树包含 1、3 两个点，一棵树包含 2、5、6 这 3 个点，还有一棵树只包含 4 这一个点即可。我们规定每棵树的根结点是这棵树所对应的集合的代表元（representative）。

如果把 x 的父结点保存在 p[x]中（如果 x 没有父亲，则 p[x]等于 x），则不难写出“查找结点 x 所在树的根结点”的递归程序：int find(int x) { p[x] == x ? x : find(p[x]); }，翻译成大白话就是：如果 p[x]等于 x，说明 x 本身就是树根，因此返回 x；否则返回 x 的父亲 p[x]所在树的树根。

问题来了：在特殊情况下，这棵树可能是一条长长的链。设链的最后一个结点为 x，则

每次执行 find(x)都会遍历整条链，效率十分低下。看上去是个很棘手的问题，其实改进方法很简单。既然每棵树表示的只是一个集合，因此树的形态是无关紧要的，并不需要在“查找”操作之后保持树的形态不变，只要顺便把遍历过的结点都改成树根的儿子，下次查找就会快很多了，如图 11-3 所示。

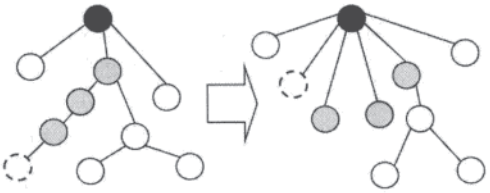


图 11-3　并查集中的路经压缩

这样，Kruskal 算法的完整代码便不难给出了。假设第 i 条边的两个端点序号和权值分别保存在 u[i], v[i]和 w[i]中，而排序后第 i 小的边的序号保存在 r[i]中（顺便说一句，这叫做间接排序——排序的关键字是对象的“代号”，而不是对象本身）。

```
int cmp(const int i, const int j) { return w[i]<w[j]; }    // 间接排序函数
int find(int x) { return p[x] == x ? x : p[x] = find(p[x]);}// 并查集的 find
int Kruskal()
{
  int ans = 0;
  for(int i = 0; i < n; i++) p[i] = i;       // 初始化并查集
  for(int i = 0; i < m; i++) r[i] = i;       // 初始化边序号
  sort(r, r+m, cmp); // 给边排序
  for(int i = 0; i < m; i++)
  {
     int e = r[i]; int x = find(u[e]); int y = find(v[e]);
                                             // 找出当前边两个端点所在集合编号
     if(x != y) { ans += w[e]; p[x] = y; } // 如果在不同集合，合并
  }
  return ans;
}
```

注意 x 和 y 分别是第 e 条边的两个端点所在连通分量的代表元。合并 x 和 y 所在集合可以简单地写成 p[x]=y，即直接把 x 作为 y 的儿子，则两个树就合并成一棵树了。注意不能写成 p[u[e]]=p[v[e]]，因为 u[e]和 v[e]不一定是树根。顺便说一句，并查集的效率非常高——在平摊意义下，find 函数的时间复杂度几乎可以看成是常数（而 union 显然是常数时间）。

11.2　最短路问题

最短路问题对我们来说并不陌生：在第 9 章中，我们曾学过无权和带权 DAG 上的最短

路和最长路——二者的算法几乎是一样的（只是初始化不同，并且状态转移时把 min 和 max 互换）。但如果图中可以有环，情况就不太妙了。

11.2.1 Dijkstra 算法

首先考虑所有边权均为正的情况。在这种情况下，最短路是一定存在的，但最长路却不一定存在——如果图中有圈（环），每走一圈路就会变长（别忘了，所有边权均为正）。为了简单起见，先考虑最短路问题。

下面直接给出 Dijkstra 算法的伪代码，它可用于计算正权图上的单源最短路（Single-Source Shortest Paths，SSSP），即从单个源点出发，到所有结点的最短路。该算法同时适用于有向图和无向图。

```
清除所有点的标号
设 d[0]=0，其他 d[i]=INF
循环 n 次
{
    在所有未标号结点中，选出 d 值最小的结点 x
    给结点 x 标记
    对于从 x 出发的所有边(x,y)，更新 d[y] = min{d[y], d[x]+w(x,y)}
}
```

下面是伪代码对应的程序。假设起点是结点 0，它到结点 i 的路径长度为 d[i]。未标号结点的 v[i]=0，已标号结点的 v[i]=1。为了简单起见，用 w[x][y]==INF 表示边(x,y)不存在。

```
memset(v, 0, sizeof(v));
for(int i = 0; i < n; i++) d[i] = (i==0 ? 0 : INF);
for(int i = 0; i < n; i++)
{
  int x, m = INF;
  for(int y = 0; y < n; y++) if(!v[y] && d[y]<=m) m = d[x=y];
  v[x] = 1;
  for(int y = 0; y < n; y++) d[y] <?= d[x] + w[x][y];
}
```

除了求出最短路的长度外，使用 Dijkstra 算法也能很方便地打印出结点 0 到所有结点的最短路本身，原理和动态规划中的方案打印一样——从终点出发，不断顺着 d[i]+w[i][j]==d[j] 的边(i,j)从结点 j “退回”到结点 i，直到回到起点。另外，仍然可以用空间换时间，在更新 d 数组时维护“父亲指针”。具体来说，需要把 d[y] <?= d[x]+w[x][y]改成：

```
if(d[y] > d[x] + w[x][y])
{
  d[y] = d[x] + w[x][y];
  fa[y] = x;
}
```

把它称为边(x,y)上的松弛操作（relaxation）。

11.2.2　稀疏图的邻接表

不难看出，上面程序的时间复杂度为$O(n^2)$——循环体一共执行了n次，而在每次循环中，“求最小d值”和“更新其他d值”均是$O(n)$的。由于最短路算法实在太重要了，我们花一些篇幅把它优化到$O(m\log n)$，并给出一份简单高效的完整代码。

等一等，为什么说是“优化到”呢？在最坏情况下，m和n^2是同阶的，$m\log n$岂不是比n^2要大？这话没错，但在很多情况下，图中的边并没有那么多，$m\log n$比n^2小得多。我们把m远小于n^2的图称为稀疏图（Sparse Graph），而m相对较大的图称为稠密图（Dense Graph）。

在学习稀疏图时，首先要掌握图的新表示法。既然m远小于n^2，那么邻接矩阵中会有大量的表示“此边不存在”的元素，不仅浪费了空间，而且也降低了时间效率——例如，为了遍历所有边，必须检查邻接矩阵中的所有元素。尽管只有那些实际存在的边参与了核心运算，但我们浪费了大量的时间去判断到底哪些边存在。

当然可以使用前面介绍的vector<int> G[maxn]存储稀疏图，但这里想介绍的是另外一种流行的表示法——邻接表（Adjacency List）。在这种表示法中，每个结点i都有一个链表，里面保存着从i出发的所有边。对于无向图来说，每条边会在邻接表中出现两次。和前面一样，这里继续用数组实现链表：首先给每条边编号，然后用first[u]保存结点u的第一条边的编号，next[e]表示编号为e的边的“下一条边”的编号。下面的函数读入有向图的边列表，并建立邻接表：

```
int n, m;
int first[MAXN];
int u[MAXM], v[MAXM], w[MAXM], next[MAXM];
void read_graph()
{
  scanf("%d%d", &n, &m);
  for(int i = 0; i < n; i++) first[i] = -1; // 初始化表头
  for(int e = 0; e < m; e++)
  {
    scanf("%d%d%d", &u[e], &v[e], &w[e]);
    next[e] = first[u[e]];                    // 插入链表
    first[u[e]] = e;
  }
}
```

上述代码的巧妙之处是插入到链表的首部而非尾部，这样就避免了对链表的遍历。不过需要注意的是，同一个起点的各条边在邻接表中的顺序和读入顺序正好相反。读者如果还记得哈希表，应该会发现这里的链表和哈希表中的链表实现很相似。

11.2.3 使用优先队列的 Dijkstra 算法

有了邻接表，“遍历从 x 出发的所有边(x,y)，更新 d[y]”就可以写成“for(int e = first[x]; e !=-1; e = next[e]) d[v[e]] <?= d[x]+w[e]”。尽管在最坏情况下，这个循环仍然会循环 n-1 次，但从整体上来看，每条边恰好被检查过一次（想一想，为什么），因此“d[v[e]]<?= d[x]+w[e]”这条语句执行的次数恰好是 m。这样，只需集中精力优化“找出未标号结点中的最小 d 值”即可。

幸运的是，C++的 STL 中提供了“优先队列”这一容器，可以帮助我们快速完成这一操作。优先队列和普通的 FIFO 队列都定义在<queue>中，有 push()和 pop()过程，分别表示“往队列里加入新元素”和“从队列里删除队首元素”。唯一的区别是，在优先队列中，元素并不是按照进入队列的先后顺序排列，而是按照优先级的高低顺序排列。换句话说，pop()删除的是优先级最高的元素，而不一定是最先进入队列的元素。正因为如此，获取队首元素的方法不再是 front()，而是 top()。

定义优先队列最简单的方法是 priority_queue<类型名> q，它利用元素自身的“小于”操作符来定义优先级。例如，在 priority_queue<int> q 这样的优先队列中，先出队的总是最大的整数。使用自定义比较的方法和 set 类似：

```
struct cmp
{
  bool operator() (const int a, const int b)    // a 的优先级比 b 小时返回 true
  {
    return a % 10 > b % 10;
  }
};

priority_queue<int, vector<int>, cmp> q;    // “个位数大的优先级反而小”的整数
                                               优先队列
```

在 Dijkstra 算法中，d[i]小的值应该先出队，因此需要使用自定义比较器。在 STL 中，可以用 greater<int>表示“大于”运算符，因此可以用 priority_queue<int, vector<int, vector<int>, greater<int> >q 来声明一个小整数先出队的优先队列。注意，最后两个大于号之间一定要有空格，不然会被误认为移位运算符“>>”。

然而，这个方法是有问题的：除了需要最小的 d 值之外，还要找到这个最小值对应的结点编号。解决方法是：把 d 值和编号“捆绑”成一个整体放到优先队列中，使得取出最小 d 值的同时也会取出对应的结点编号。

STL 中的 pair 便是专门把两个类型捆绑到一起的。为了方便起见，我们用 typedef pair<int,int> pii 自定义一个 pii 类型，则 priority_queue<pii, vector<pii>, greater<pii> > q 就定义了一个由二元组构成的优先队列。pair 定义了它自己的排序规则——先比较第一维，相等时才比较第二维，因此需要按(d[i],i)而不是(i,d[i])的方式组合。代码如下：

```
bool done[MAXN];
for(int i = 0; i < n; i++) d[i] = (i==0 ? 0 : INF);
```

```
memset(done, 0, sizeof(done));          // 初始化计算标志：所有结点都没有算出
q.push(make_pair(d[0], 0));             // 起点进入优先队列
while(!q.empty())
{
  pii u = q.top(); q.pop();
  int x = u.second;
  if(done[x]) continue;                 // 已经算过，忽略
  done[x] = true;
  for(int e = first[x]; e != -1; e = next[e]) if(d[v[e]] > d[x]+w[e])
  {
    d[v[e]] = d[x] + w[e];              // 松弛成功，更新 d[v[e]]
    q.push(make_pair(d[v[e]], v[e]));   // 加入优先队列
  }
}
```

在松弛成功后，需要修改结点 v[e]的优先级，但 STL 中的优先队列不提供“修改优先级”的操作。因此，只能将新的 d[i]重新插入优先队列。这样做并不会影响结果的正确性，因为 d 值小的结点自然会先出队。为了防止结点的重复扩展，如果发现新取出来的结点曾经被取出来过（done[x]），应该直接把它扔掉。顺便说一句：避免重复的另一个方法是把 if(done[x])改成 if(u.first != d[x])，可以省掉一个 done 数组。

再补充一点：即使是稠密图，使用 priority_queue 实现的 Dijkstra 算法也常常比基于邻接矩阵的 Dijkstra 算法的运算速度快。理由很简单：执行 push 操作的前提是 d[v[e]] > d[x]+w[e]，如果这个式子常常不成立，则 push 操作会很少。

11.2.4　Bellman-Ford 算法

当负权存在时，连最短路都不一定存在了。尽管如此，还是有办法在最短路存在的情况下把它求出来。在介绍算法之前，请读者确认这样一个事实：如果最短路存在，一定存在一个不含环的最短路。

理由如下：在边权可正可负的图中，环有零环、正环和负环 3 种。如果包含零环或正环，去掉以后路径不会变长；如果包含负环，则意味着最短路不存在（想一想，为什么）。

既然不含环，最短路最多只经过（起点不算）n-1 个结点，可以通过 n-1 “轮”松弛操作得到，像这样（起点仍然是 0）：

```
for(int i = 0; i < n; i++) d[i] = INF;
d[0] = 0;
for(int k = 0; k < n-1; k++) // 迭代 k 次
  for(int i = 0; i < m; i++) // 检查每条边
  {
    int x = u[i], y = v[i];
    if(d[x] < INF) d[y] <?= d[x]+w[i]; // 松弛
}
```

上述算法称为 Bellman-Ford 算法，不难看出它的时间复杂度为 $O(nm)$。在实践中，我们常常用 FIFO 队列来代替上面的循环检查，像这样：

```
queue<int> q;
bool inq[MAXN];
for(int i = 0; i < n; i++) d[i] = (i==0 ? 0 : INF);
memset(inq, 0, sizeof(inq)); // “在队列中”的标志
q.push(0);
while(!q.empty())
{
  int x = q.front(); q.pop();
  inq[x] = false;                    // 清除“在队列中”标志
  for(int e = first[x]; e != -1; e = next[e]) if(d[v[e]] > d[x]+w[e])
  {
    d[v[e]] = d[x] + w[e];
    if(!inq[v[e]])                   // 如果已经在队列中，就不要重复加了
    {
      inq[v[e]] = true;
      q.push(v[e]);
    }
  }
}
```

有没有注意到上面的代码和前面的 Dijkstra 算法很像？一方面，优先队列替换为了普通的 FIFO 队列，而另一方面，一个结点可以多次进入队列。可以证明，采取 FIFO 队列的 Bellman-Ford 算法在最坏情况下需要 $O(nm)$时间，不过在实践中，往往只需要很短的时间就能求出最短路。

11.2.5 Floyd 算法

如果你需要求出每两点之间的最短路，不必调用 n 次 Dijkstra（边权均为正）或者 Bellman-ford（有负权）。有一个更简单的方法可以满足你的需要——Floyd-Warshall 算法（请记住下面的代码！）：

```
for(int k = 0; k < n; k++)
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      d[i][j] <?= d[i][k] + d[k][j];
```

在调用它之前只需做一些简单的初始化：d[i][i]=0，其他 d 值为“正无穷”INF。注意这里有一个潜在的问题：如果 INF 定义太大（如 2000000000），加法 d[i][k] + d[k][j]可能会溢出！但如果 INF 太小，可能会使得长度为 INF 的边真的变成最短路的一部分。为了谨慎起见，最好估计一下实际最短路长度的上限，并把 INF 设置成“只比它大一点点”的值。例如，最多有 1000 条边，每条边长度不超过 1000 的话，可以把 INF 设成 1000001。

如果坚持认为不应该允许 INF 和其他值相加，更不应该得到一个大于 INF 的数，请把上述代码改成：

```
for(int k = 0; k < n; k++)
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      if(d[i][j] < INF && d[k][j] < INF) d[i][j] <?= d[i][k] + d[k][j];
```

在有向图中，有时不必关心路径的长度，而只关心每两点间是否有通路，则可以用 1 和 0 分别表示“连通”和“不连通”。这样，除了预处理需做少许调整外，主算法中只需把“d[i][j] <?= d[i][k] + d[k][j]”改成“d[i][j] = d[i][j] || (d[i][k] && d[k][j])”。这样的结果称为有向图的传递闭包（Transitive Closure）。

11.3　网络流初步

网络流是一个适用范围相当广的模型，相关的算法也非常多。尽管如此，网络流中的概念、思想和基本算法并不难理解。

11.3.1　最大流问题

如图 11-4 所示，假设你需要把一些物品从结点 s（称为源点）运送到结点 t（称为汇点），可以从其他结点中转。图 11-4（a）中各条有向边的权表示最多能有多少个物品从这条边的起点直接运送到终点。例如最多可以用 9 个物品从节点 v_3 运送到 v_2。

图 11-4（b）展示了一种可能的最优方案，其中每条边中的第 1 个数字表示实际运送的物品数目，而第 2 个数字就是题目中的上限。

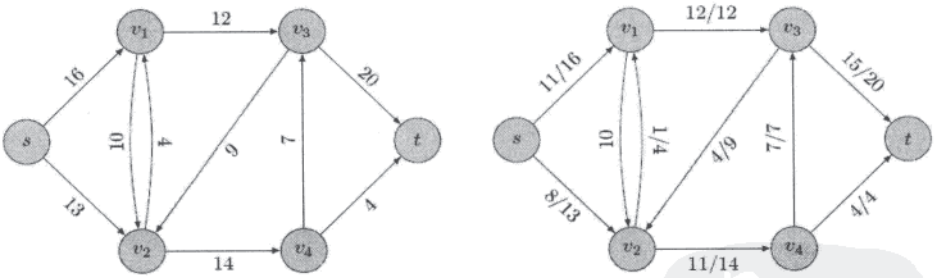


图 11-4　物资运送问题

我们把这样的问题称为最大流问题（Maximum-Flow Problem）。对于一条边(u,v)，它的物品上限称为容量（capacity），记为 $c(u,v)$（对于不存在的边(u,v)，$c(u,v)=0$）；实际运送的物品称为流量（flow），记为 $f(u,v)$。注意，“把 3 个物品从 u 运送到 v，又把 5 个物品从 v 运送到 u”没什么意义，因为它等价于把两个物品从 v 运送到 u。这样，我们就可以规

定 $f(u,v)$ 和 $f(v,u)$ 最多只有一个正数（可以均为 0），并且 $f(u,v)=-f(v,u)$。这样规定就好比“把 3 个物品从 u 运送到 v”等价于“把 -3 个物品从 v 运送到 u”一样。

最大流问题的目标是把最多的物品从 s 运送到 t，而其他结点都只是中转，因此对于除了结点 s 和 t 外的任意结点 u，$\sum_{(u,v)\in E} f(u,v)=0$（别忘了这些 f 中有些是负数）。从 s 运送出来的物品数目等于到达 t 的物品数目，而这正是我们最大化的目标。

提示 11-2：在最大流问题中，容量 c 和流量 f 满足 3 个性质：容量限制（$f(u,v)<=c(u,v)$）、斜对称性（$f(u,v)=-f(v,u)$）、流量平衡（对于除了结点 s 和 t 外的任意结点 u，$\sum_{(u,v)\in E} f(u,v)=0$）。问题的目标是最大化 $|f|=\sum_{(s,v)\in E} f(s,v)=\sum_{(u,t)\in E} f(u,t)$，即从 s 点流出的净流量（它也等于流入 t 点的净流量）。

11.3.2 增广路算法

介绍完最大流问题后，下面介绍求解最大流问题的算法。算法思想很简单，从零流（所有边的流量均为 0）开始不断增加流量，保持每次增加流量后都满足容量限制、斜对称性和流量平衡 3 个条件。

把图 11-5（a）中的每条边上容量与流量之差（称为残余容量，简称残量）计算出，得到图 11-5（b）中的残量网络（residual network）。同理，由图 11-5（c）可以得到图 11-5（d）。注意残量网络中的边数可能达到原图中边数的两倍，如原图中 c=16, f=11 的边在残量网络中对应正反两条边，残量分别为 16−11=5 和 0−(−11)=11。

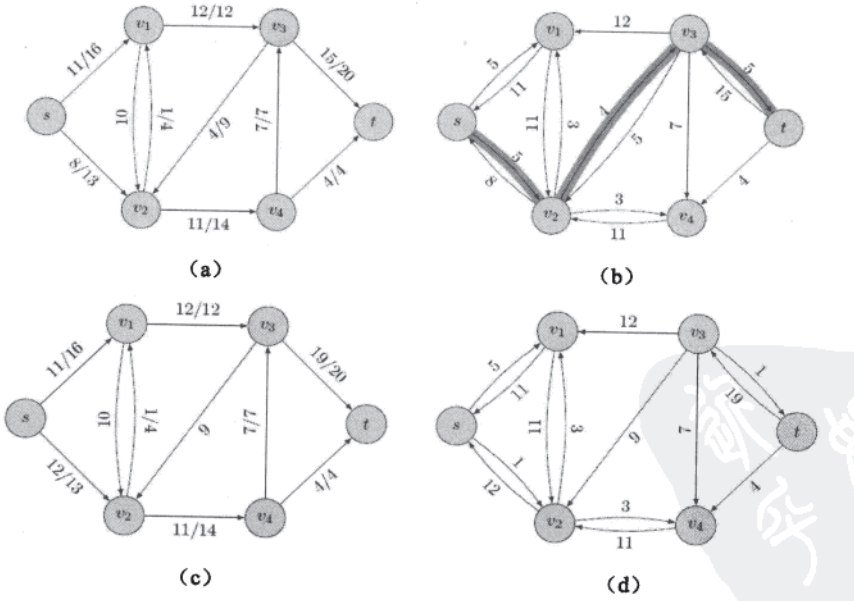


图 11-5 残量网络和增广路算法

我们的算法基于这样一个事实：残量网络中任何一条从 s 到 t 的有向道路都对应一条原图中的增广路（augmenting path）——只要求出该道路中所有残量的最小值 d，把对应的所有边上的流量增加 d 即可，这个过程称为增广（augmenting）。不难验证，如果增广前的流量满足 3 个条件，增广后仍然满足。显然，只要残量网络中存在增广路，流量就可以增大。可以证明它的逆命题也成立：如果残量网络中不存在增广路，则当前流就是最大流。这就是著名的增广路定理。

提示 11-3：当且仅当残量网络中不存在 s-t 有向道路（增广路）时，此时的流是从 s 到 t 的最大流。

“找任意路径”最简单的办法无疑是用 DFS，但很容易找出让它很慢的例子。一个稍微好一些的方法是使用 BFS，它足以应对数据不刁钻的网络流题目。这就是 Edmonds-Karp 算法。在下面的代码中，源点和汇点保存在变量 s 和 t 中，运行结束后，s-t 的净流量保存在变量 f 中。

```
queue<int> q;
memset(flow, 0, sizeof(flow));
f = 0;
for(;;)
{
  memset(a, 0, sizeof(a));
  a[s] = INF;
  q.push(s);
  while(!q.empty())                    // BFS 找增广路
  {
    int u = q.front(); q.pop();
    for(int v = 1; v <= n; v++) if(!a[v] && cap[u][v]>flow[u][v])
                                       // 找到新结点 v
   {
      p[v] = u; q.push(v);             // 记录 v 的父亲，并加入 FIFO 队列
      a[v] = a[u] <? cap[u][v]-flow[u][v];     // s-v 路径上的最小残量
    }
  }
  if(a[t] == 0) break;                 // 找不到，则当前流已经是最大流
  for(int u = t; u != s; u = p[u])     // 从汇点往回走
  {
    flow[p[u]][u] += a[t];             // 更新正向流量
    flow[u][p[u]] -= a[t];             // 更新反向流量
  }
  f += a[t];                           // 更新从 s 流出的总流量
}
```

正如所见，上面的代码和普通的 BFS 并没有太大的不同。唯一需要注意的是，在扩展

结点的同时还需递推出从 s 到每个结点 i 的路径上的最小残量 a[i]，则 a[t]就是整条 s-t 道路上的最小残量。另外，由于 a[i]总是正数，我们用它代替了原来的 vis 标志数组。上面的代码把流初始化为零流，但这并不是必需的。只要初始流是可行的（满足 3 个限制条件），就可以用增广路算法进行增广。

为了简单起见，假设图保存在邻接矩阵中，读者不难把它改成使用邻接表的版本。唯一需要注意的是从 t 走回 s 过程：如果还是只保存 u 的“上一个结点”p[u]，是无法找到对应的边(p[u],u)及其反向边(u,p(u))的。

11.3.3 最小割最大流定理

有一个与最大流关系密切的问题：最小割。如图 11-6 所示，把所有顶点分成两个集合 S 和 $T=V-S$，其中源点 s 在集合 S 中，汇点 t 在集合 T 中。

如果把“起点在 S 中，终点在 T 中”的边全部删除，就无法从 s 到达 t 了。我们把这样的集合划分(S,T)称为一个 s-t 割，它的容量定义为：$c(S,T)=\sum_{u\in S,v\in T}c(u,v)$，即起点在 S 中，终点在 T 中的所有边的容量和。

还可从另外一个角度看待割。如图 11-7 所示，从 s 运送到 t 的物品必然通过跨越 S 和 T 的边，所以从 s 到 t 的净流量等于$|f|=f(S,T)=\sum_{u\in S,v\in T}f(u,v)\leqslant\sum_{u\in S,v\in T}c(u,v)=c(S,T)$。

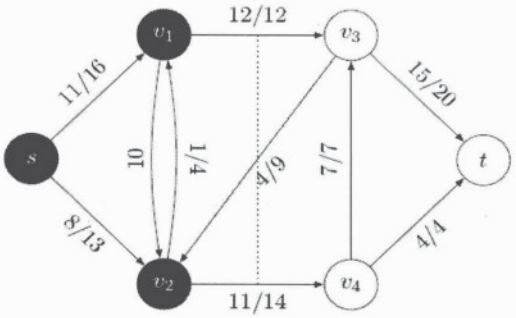


图 11-6 网络中的割

图 11-7 流和割的关系

注意这里的割(S,T)是任取的，因此得到了一个重要结论：对于任意 s-t 流 f 和任意 s-t 割(S,T)，有$|f|\leqslant c(S,T)$。

下面来看残量网络中没有增广路的情形。既然不存在增广路，在残量网络中 s 和 t 并不连通。当 BFS 没有找到任何 s-t 道路时，把已标号结点（a[u]>0 的结点 u）集合看成 S，令 $T=V-S$，则在残量网络中 S 和 T 分离，因此在原图中跨越 S 和 T 的所有弧均满载（这样的边才不会存在于残量网络中），且没有从 T 回到 S 的流量，因此$|f|\leqslant c(S,T)$成立。

前面说过，对于任意的 f 和(S,T)，都有$|f|\leqslant c(S,T)$，而我们又找到了一组让等号成立的 f 和(S,T)。这样，我们同时证明了增广路定理和最小割最大流定理：在增广路算法结束时，f 是 s-t 最大流，(S,T)是 s-t 最小割。

提示 11-4：增广路算法结束时，令已标号结点（a[u]>0 的结点）集合为 S，其他结点集合为 $T=V-S$，则(S,T)是图的 s-t 最小割。

11.3.4　最小费用最大流问题

下面给网络流增加一个因素：费用。假设每条边除了有一个容量限制外，还有一个单位流量所需的费用（cost）。图 11-8（a）中分别用 c 和 a 来表示每条边的容量和费用，而图 11-8（b）给出了一个在总流量最大的前提下，总费用最小的流（费用为 10），即最小费用最大流。另一个最大流是从 s 分别运送一个单位到 x 和 y，但总费用为 11，不是最优。

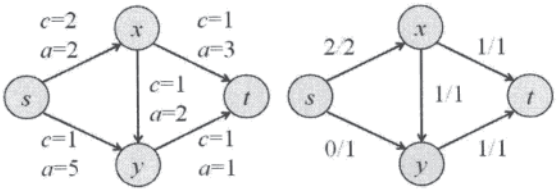


图 11-8　最小费用最大流

在最小费用流问题中，平行边变得有意义了：可能会有两条从 u 到 v 的弧，费用分别为 1 和 2。在没有费用的情况下，我们可以把二者合并，但由于费用的出现，我们无法合并这两条弧。再如，若边(u,v)和(v,u)均存在，且费用都是负数，则“同时从 u 流向 v 和从 v 流向 u”是个不错的主意。为了更方便地叙述算法，我们先假定图中不存在平行边和反向边。这样就可以用两个邻接矩阵 cap 和 cost 保存各边的容量和费用。为了便于增广，对于一条边(u,v)，我们规定 cap[v][u]=0 并且 cost[v][u]=-cost[u][v]，表示沿着(u,v)的相反方向增广时，费用减小 cost[u][v]。

限于篇幅，这里直接给出算法：和 Edmonds-Karp 算法类似，但每次用 Bellman-Ford 算法而非 BFS 找增广路。只要初始流是该流量下的最小费用可行流，每次增广后的新流都是新流量下的最小费用流。另外，费用值是可正可负的。

```
queue<int> q;
int d[MAXN];
memset(flow, 0, sizeof(flow));
c = f = 0;
for(;;)
{
  // bellman-ford 算法开始（在残量网络中找 s-t 最短路）
  bool inq[MAXN];
  for(int i = 0; i < n; i++) d[i] = (i==s ? 0 : INF);
  memset(inq, 0, sizeof(inq));
```

```
    q.push(s);
    while(!q.empty())
    {
      int u = q.front(); q.pop();
      inq[u] = false;
      for(int v = 0; v < n; v++) if(cap[u][v]>flow[u][v] && d[v]>d[u]+cost[u][v])
      {
        d[v] = d[u] + cost[u][v];
        p[v] = u;
        if(!inq[v])
        {
          inq[v] = true;
          q.push(v);
        }
      }
    }
    // bellman-ford 算法到此结束

    if(d[t] == INF) break;              // 汇点不可达，表明当前流已经是最小费用最大流
    int a = INF;
    for(int u = t; u != s; u = p[u]) a <?= cap[p[u]][u] - flow[p[u]][u];
                                        // 计算可改进量
    for(int u = t; u != s; u = p[u])// 增广
    {
      flow[p[u]][u] += a;
      flow[u][p[u]] -= a;
    }
    c += d[t]*a;                        // 更新总费用和流量
    f += a;
  }
```

运行结束后，f和c分别保存最大流流量和该流量下的最小费用。读者不难把上述代码改成基于邻接表的，在此情况下，可以去掉“不存在平行边和反向边”的限制——邻接表中可以有起点和终点相同的边，而算法不需要经过任何修改。

11.4 进一步学习的参考

尽管本书的第 1～11 章已经介绍了很多知识，但难以涵盖千变万化的题目。另外，限于篇幅，本书的很多定理和算法都没有给出严格证明。你如果是一个严密的选手，也许需要更加仔细地钻研本书中涉及的东西。

11.4.1　编程语言

尽管本书使用 C 和 C++，但它们并不是笔者最喜欢的编程语言。很多算法竞赛支持另外一些流行的编程语言，如 Java、C#和 Python。这里特别值得一提的是 Python，它能异常简洁地实现你的算法思想，并且非常易于学习。更妙的是，当你对开发商业软件和网站产生兴趣之后，Python 将继续成为你最好的伙伴。

编程语言有很多不错的在线参考，包括语法参考、库函数参考和常见错误集锦等。初学时，读者往往需要大量参考这些资料，但在完成了一定的练习之后，即可独立完全编写程序。

11.4.2　数据结构

数据结构是高等院校计算机专业的经典必修课程。在前面的章节中，读者不仅体会到了数组、链表、二叉树和树等基础数据结构的重要性，也学到了集合、优先队列、并查集等专门数据结构在提高程序效率方面的作用。尽管如此，仍有很多细节没有被涉及，如更多排序算法及其比较、字符串匹配算法、排序二叉树、Fenwick 树和线段树等。

关于数据结构方面的教材和著作，很多都包含伪代码或者完整代码实现。但需要注意的是，这些代码不见得适用于算法程序。一方面，很多常见数据结构已经包含于 C++的 STL 或者 Java 的 JCF 中，不必自行编写；另一方面，需要自己编写的复杂数据结构在传统教材中一般是找不到的。换句话说，对于准备算法竞赛来说，传统教材的最大价值在于概念和思想，而非代码。如果对本书的数据结构内容已经相当熟悉，只需找一本网上评价较好的书籍查漏补缺，并完成一定的非上机练习即可，而不必花大量的时间重新学习每一个细节。

11.4.3　算法设计

“算法课应该有哪些内容”存在着广泛的争议。不仅如此，算法课的讲授时间在不同的高等院校中也是不同的，讲授方法也是千差万别。国内外已经有算法课程以 ACM/ICPC 等算法竞赛为背景，用在线评测系统（Online Judge，OJ）作为作业布置和考试系统，填补了理论和实际之间的空白。这方面最权威的读物是 Skiena 和 Miguel 合著的《Programming Challenges》[①]，推荐读者阅读。

但也应看到，并不是所有的算法都适合在算法竞赛中体现。例如，为了避免选手对题目可解性的异议，ACM/ICPC 命题委员会在选取 NP-完全问题作为比赛题目时往往十分谨慎，导致人工智能搜索算法、近似算法等领域并不为选手所熟悉。

计算理论也是算法课程中很重要的一个方面。选手至少应当知道归约的含义和常见的 NP 完全问题，最好还能具备一些图灵机的初步知识，夯实自己的理论基础。

关于算法，最经典、最通俗易懂的入门读物是 Cormen、Leiserson、Rivest 和 Stein 合著的《Introduction to Algorithms》（算法导论，有时按作者首字母简称为 CLRS），近年来又

[①] 中译版《编程挑战》，已由清华大学出版社出版，译者：刘汝佳。

涌现出另一些相当优秀的教材，包括 Kleinberg 和 Tardos 合著的《Algorithm Design》，以及 Dasgupta、Papadimitriou 和 Vazirani 合著的《Algorithms》。

另外，笔者还要推荐两本手册（而不是教材）：Skenia 的《the Algorithm Design Manual》，Heineman、Pollice 和 Selkow 合著的《Algorithms in a Nutshell》。

11.4.4 数学

算法离不开数学。如果想在算法竞赛中脱颖而出，除了本书提到的数学外，算法竞赛中其他经常涉及的数学包括线性代数知识（如矩阵、线性方程组）、数值计算（如求导、数值积分）、概率论等，而组合数学、数论和几何也比本书介绍的内容更深。

推荐读者仔细阅读 Graham、Knuth 和 Patashnik 合著的《具体数学》，并抽一些时间翻阅上述领域的经典教材。

11.4.5 参赛指南

即使上面所说的知识读者都掌握得很好了，也并不意味着可以轻松地在各种算法竞赛中获胜。如果你真的很有兴趣和激情，推荐阅读《算法艺术与信息学竞赛》中另外两本即将出版的书籍：《算法实践手册》和《算法艺术与问题求解》。

当然，最大的提高来源于实践。如果你能亲自参加算法竞赛，收获将是最多的。

你如果是中学生，可以考虑参加 IOI 系列比赛。在中国，这意味着你需要首先参加全国信息学奥林匹克联赛（NOIP），它是国内参加人数最多、普及性最高的比赛。接下来，你需要参加全国信息学奥林匹克竞赛（National Olympiad in Informatics，NOI）。但这个比赛并不是任何人都能参加的，因为它有着严格的名额限制（例如，目前每个省只能有 3～5 名选手参加，奖励名额除外）。不同的省有着不同的省队选拔方式，但如果想参加 NOI，请先努力让自己的水平位于全省前列。NOI 的全国前 20 名将进入国家集训队，最终选拔出 4 名选手参加国际信息学奥林匹克竞赛（International Olympiad in Informatics，IOI）。

你如果是一个大学生，可以考虑参加 ACM/ICPC 系列比赛。和 IOI 相比，你的征途看上去要简单一些——获得国际金牌只需两个步骤：参加区域赛（Regional Contest）并获得出线资格，参加总决赛（World Finals）并获得金牌。

除此，你还有很多比赛可以参加。例如，TopCoder 就是一个商业的比赛平台，一年四季都在组织算法、组件开发等比赛。如果你够厉害，可以获得出国比赛的机会，并获取丰厚的奖金（详情请见 www.topcoder.com/tc）；各大 IT 公司也常常举办自己的程序设计大赛，如百度之星、Google Code Jam、网易的“有道难题”等，读者很容易在网上搜索到相关信息。另外，各大在线评测系统也常常举办各种比赛，如国内的 POJ（acm.pku.edu.cn）、ZOJ（acm.zju.edu.cn），国外的 UVa（uva.onlinejudge.org）、Ural（acm.timus.ru）、SGU（acm.sgu.ru）和 SPOJ（www.spoj. pl）等。另外，还有一些不属于上述类型，但影响甚广的比赛，如 IPSC（ipsc.ksp.sk），它最大的特点是提交答案而非程序，而且题目非常新颖、有趣。总之，在目前这样一个算法爱好者众多的时代，一定不愁没有比赛可以参加。

11.5　训 练 参 考

和往常一样，先是一些 UVaOJ 中的题目：Risk 游戏（567）、斑点（10034）、噪音恐惧症（10048）、连接校园（10397）、北极网络（10369）、“这不是 bug，而是特性”（658）、导游（10099）、电梯（10801）、发送 Email（10986）、虫洞（558）、国王（515）、套汇（104）、路径统计（125）、雷山（10803）、电力传输（10330）、“Dijkstra，Dijkstra”（10806）、合身的衣服（11045）、Unix 插头（753）、逃脱问题（563）、逃脱问题——大结局（10746）、数据流（10594）、环和绳子（10985）、朋友（10608）、战争（10158）、马丽奥大探险（10269）、吃还是不吃（10273）。

下面是 CII 中的题目：最短路对（2927）、电视电缆网络（3031）、去柏林（3645）、危险物（3644）、方程（4047）、企鹅进行曲（3972）、尊严捍卫者（3415）、星际门（2686）、驾车探险（3307）、锦标赛（2531）、几何地图（3886）、苗条的生成树（3887）、Argus（3135）、细胞（3486）、任务序列（2954）、动物大逃亡（3661）、摘水果（3939）、去括号（3513）、冰激凌加派（3562）、长城游戏（3276）。

附录 A　开发环境与方法

开发环境和开发方法能大大提高编程的速度和正确性，但却常常被人忽视。本附录介绍命令行、脚本编程和编译器以及调试器的基本使用方法，希望能给读者带来帮助。

A.1　命　令　行

在图形用户界面（Graphical User Interface，GUI）日益发达的今天，已经有越来越多的人不知命令行为何物了。但笔者仍然认为命令行操作是每一位编程竞赛的选手必须掌握的技能。它不仅可以让你看起来很“酷”很专业，而且确实能帮你很大的忙。

首先，进入命令行。在 Windows XP 中，可以选择“开始”菜单中的“运行”命令，在弹出的“运行”对话框中输入 cmd，然后按 Enter 键，将出现类似下面的提示信息：

```
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>
```

其中，C:\Documents and Settings\Administrator 是当前路径，而后的“>”符号是命令提示符，紧跟其后的是闪烁的光标（cursor）。在文本界面中，所输入的任何信息都将出现在光标的所在位置。输入命令之后不要忘记按 Enter 键。

在 Linux 中，打开终端（terminal）即可进行命令行操作。Linux 终端并不一定会显示当前路径，可以用 pwd 命令将其显示。无论是 Windows 还是 Linux，都可以用上下箭头来翻阅并使用历史记录。Windows 和 Linux 下都可以用 Tab 键补全命令，但在细节上存在一些差异，读者可以自己实践或查阅相关资料。

A.1.1　文件系统

学习命令行的第一步是理解文件系统。相信读者对“文件”这一概念已经有所认识，但除此之外还需要清楚文件所在的位置。“位置”的表达方式有两种，一种是相对路径，一种是绝对路径。

相对路径（relative path）是相对当前路径（current path）而言的，它在命令行中已有所体现。例如在上面的例子中，当前路径是 C:\Documents and Settings\Administrator。在这种情况下，命令 type abc.txt 即为试图显示 C:\Documents and Settings\Administrator\abc.txt。

除了直接给出文件名外，还可以借助当前目录“.”和父目录“..”进行更为灵活的相对路径引用。例如，在上面的命令行提示符下输入 type..\..\Windows\123.txt 实际上是在试图显

示 c:\Windows\123.txt。

在命令行中可以用“cd <目录名>”的方式改变当前路径。例如，“cd..”会进入父目录，而“cd aaa”会进入当前目录的 aaa 子目录。

绝对路径和相对路径的区别是，前者给出了“起点”，它的实际指向不随当前路径变化。在算法竞赛中，不要在提交的源代码中引用绝对路径，但在操作和调试程序的过程中你可以随意使用绝对路径。另外，Linux 中的路径分隔符是正斜线/，而非反斜线\。

如果在程序中读写文件，则“当前路径”一般是和该程序位于同一个目录，但也可以更改。如果在执行程序时出现“找不到文件”的错误，而你又认为文件确实存在，则极有可能是程序的“当前路径”和你想的不一样。一个笨（但有效）的方法是用 freopen(“test.txt”, “w”, stdout)的方法创建文件 test.txt。找到了这个文件，你就知道当前路径是什么了。如果要在 freopen 或者 fopen 中使用“..\..\Windows\123.txt”这样的相对路径，应注意反斜线字符在 C 语言的正确表示方法是‘\\’。不过，即使在调试中也尽量不要使用路径名。如果在提交程序前忘记把路径名删除，将导致程序得 0 分。事实上，这样的例子并不少见。当然，如果只在条件编译中使用路径名，则是没有问题的。

最后一个小问题是：你不一定有存取文件的权限。如果出现类似于“Permission Denied”的错误信息，需确认你的当前用户拥有你想访问的目录或者文件的访问/修改权。在现场比赛中，这可能是因为你没有使用比赛指定账户，而是改用 guest 登录了。

A.1.2 进程

简单地说，进程是一个程序正在执行时的实体。它消耗 CPU 资源且占用内存。进程一般都有名字，同时还有一个编号（称为 PID）。

在 Windows 和 Linux 中都能方便地列出进程。在 Windows 下可以使用 Ctrl+Alt+Del 键打开任务管理器，或者在命令行下用 tasklist 命令。在 Linux 下可以用 top 命令查看当前占用 CPU 资源最多的一些进程，而 ps 命令类似于 Windows 下的 tasklist 命令，它是使用列表的方式给出当前进程。在默认情况下，ps 命令并不会列出系统进程，用 ps ax 命令可以列出更多的进程。

强行终止进程有很多方法。在 Windows 下，可以用任务管理器直接终止，也可以在命令行下用 taskkill /pid <PID>或 taskkill /im <映像名>终止进程，可以通过执行 taskkill /?查看更多选项。

在 Linux 下可以用 kill 命令终止命令，还可以用 killall <进程名>把某个进程名对应的所有进程杀掉。一个典型情况是，如果 pascal 选手的 Lazarus IDE 不响应，就可以用 killall lazarus 把它们杀掉。

作为一个好习惯，当程序非正常终止，或者系统表现异常时，应检查进程。例如，若系统反应特别慢，可能是有一些看似运行结束，但其实残留在系统中继续占用系统资源的进程。

A.1.3 程序的执行

在命令行下执行一个程序比在 IDE 中执行要方便和灵活得多。基本的方法很简单：只

需直接输入程序名即可。

例如，在 Windows 下执行 abc.exe，可以进入它所在目录后直接输入 abc 并按 Enter 键。系统为什么能找到 abc.exe 呢，因为在 Windows 下，当前目录是最先搜寻可执行文件的位置，并且扩展名“.exe”在搜索之前会被自动加上。如果当前目录没有 abc.exe，是否会报错呢？不一定。运行 path 命令，会看到一连串目录。如果当前目录没有 abc.exe，系统会继续在这些目录中寻找，全部查找完毕仍没找到时才会报错。顺便说一句，在搜索文件时并不会检查上述目录下的子目录。

Linux 有一些不同。首先，它的可执行文件名并不是以“.exe”为扩展名的，因此 g++ abc.cpp -o abc 编译出的文件是 abc，而非 abc.exe（当然了，如果你执意要将其取名为 abc.exe，也无不可）。另外，当前目录并不在搜索路径中，因此，即使 abc 已经在当前目录中，仍需要用./abc 这样的方式告诉 Linux“可执行文件 abc 就在当前目录”。

A.1.4 重定向和管道

很多比赛要求选手直接读写标准输入输出（即用 printf/scanf 或 cin/cout 读写，且不用 freopen），难道在评分时裁判要将输入数据一一用键盘输入，等程序运行结束之后用眼睛看着屏幕，对照手中的标准答案吗？当然不是。可以使用重定向的技巧将输入文件塞到程序的标准输入中，然后再将程序输出存在文件中。

在 Windows 下可以这样：abc < abc.in > abc.out。而在 Linux 下则可以这样：./abc < abc.in > abc.out。当然，如果可执行文件和输入输出文件不在同一个目录，则需要进行相应调整。但基本方法是不变的：在输入文件名前面加一个“<”符号，而在输出文件名前面加一个“>”符号。注意，此时的输出文件将被覆盖。如果希望只是把输出附加在文件末尾，则可用“>>”代替“>”。此外，如果有大量的文本输出到标准错误输出，还可以用“2>”将它们重定向，但需注意，尽量不要在正式提交的程序中输出到标准错误输出，这样不仅可能会违反比赛规定，还可能会因为大量文本的输出而占用宝贵的 CPU 资源，甚至导致超时。

Windows 和 Linux 均提供“管道”机制，用于把不同的程序串起来。例如，如果你有一个程序 aplusb 从标准输入读取两个整数 a 和 b，计算并输出 $a+b$，还有一个程序 sqr 从标准输入读取一个整数 a，计算并输出 a^2，则可以这样计算$(10+20)^2$： echo 10 20 | aplusb | sqr。尽管也可以用重定向来完成这个任务，但管道明显要简单得多。

另一个常见用法是分页显示一个文本文件的内容。在 Windows 下可以用 type abc.txt | more，在 Linux 下则是用 cat abc.txt | more。

A.1.5 常见命令

在 Linux 中，可以用 time 命令计时。例如运行 time ./abc 会执行 abc 并输出它的运行时间。但 Windows 中并没有这样的命令，幸好在大多数情况下你只是在对自己编写的程序计时，因此只需在程序的最后打印出 clock() / (double)CLOCKS_PER_SEC 即可（需要包含 time.h）。

表 A-1 中给出了一些常见命令的 Linux 版本和 Windows 版本，供读者查阅。

表 A-1　常见的 Linux 命令和 Windows 命令

分　类	Linux 命令	Windows 命令
文件列表	ls	dir
改变/创建/删除目录	cd/mkdir/rmdir	cd/md/rd
显示文件内容	cat/more	type/more
比较文件内容	diff	fc
修改文件属性	chmod	attrib
复制文件	cp	copy/xcopy
删除文件	rm	del
文件改名	mv	ren
回显	echo	echo
关闭命令行	exit	exit
在文件中查找字符串	grep	find
查看/修改环境变量	set	set
帮助	man <命令>	help <命令>

A.2　操作系统脚本编程入门

读者如果不学习脚本的编写，就无法让命令行发挥最大威力。编写脚本和编写 C 语言程序有几分相似，但有一些不同。下面先来看一个常见任务：不停地随机生成测试数据，分别运行两个程序并对比其结果。我们形象地把这个任务称为“对拍”。

A.2.1　Windows 下的批处理

下面是 Windows 下的批处理程序：

```
@echo off
:again
r > input                          ;生成随机输入
a < input > output.a
b < input > output.b
fc output.a output.b > nul         ;比较文件
if not errorlevel 1 goto again ;相同时继续循环
```

第 1 行表明接下来的各个命令本身并不会回显。如果你不明白我在说什么，试着把这一行去掉就知道了。第 2 行是一个标号，后面的 goto 语句用得上。接下来我们调用数据生成器 *r*，把输入数据写到文件 input 中，然后分别执行 *a* 和 *b*，得到相应的输出，然后用命令 fc 比较它们。注意，fc 命令有输出，但我们对此不感兴趣，因此重定向到一个名为“nul”的设备中，它就好比一个黑洞。另一个有意思的设备是“con”，它代表标准输入输出。例如，命令 copy con con 的含义是直接把标准输入复制到标准输出（尽管有些傻）。试一试，

建立一个只包含一条语句的“C 程序”：#include<con>，用命令行编译一下试试——很不幸，看上去编译器“死掉”了，尽管它其实是在读键盘。如果你在设计一个基于 Windows 的在线评测系统，小心好事者用它来愚弄你的系统！另一方面，千万不要在正式比赛中使用这个伎俩——它很可能让你失去比赛资格。

最后一行是整个批处理程序的关键——只有当比较文件相同时才 goto，否则立刻终止程序。这样，你就有机会好好研究一下这个 input 文件，看看两个程序的输出到底为什么不同。你也许会问，这个 if not errorlevel 1 到底是什么意思呢？它是在测试上一个程序（在本例中，就是 fc 程序）的返回码。if errorlevel num 的意思是“如果返回码大于或者等于 num”，因此“if not errorlevel 1”的意思是，“如果返回码小于 1”。事实上，fc 程序当且仅当文件相同时返回 0。如果你不确定程序的返回码是多少，可以在程序执行完毕后用 echo %errorlevel%命令来输出它的返回码。

你自己编写的程序的返回码是多少呢？要看你在 main 函数的最后 return 的是几了。返回码 0 往往代表“正常结束”，因此本书的正文部分才建议你用 return 0。典型的评分程序将在执行选手程序之后判断它的返回码，如果非 0，则直接认为程序非正常退出，根本不去理会输出是否正确。说到这里，你也许已经想到一种故意让返回码非 0 的情况了——输出检查器。对于答案不唯一的情况（例如，走迷宫时要求输出最短路径，但不必是字典序最小的），对拍时不能简单地用 fc 命令比较文本内容，而应该单独编写一个程序，这个程序应当在答案不一致时返回 1，以便上面的批处理程序及时终止。

上面的程序应以“.bat”为扩展名保存，并且在执行时也可以省略扩展名。如果同时存在 abc.bat 和 abc.exe，将执行 abc.exe。但如果主文件名和系统命令重名，则连 exe 文件也无法执行，如 path.exe。

A.2.2 Linux 下的 Bash 脚本

下面是上述程序的 Linux 版：

```
#!/bin/bash
while true; do
  ./r > input                          # 生成随机数据
  ./a < input > output.a
  ./b < input > output.b
  diff output.a output.b               # 文件比较
  if [ $? -ne 0 ] ; then break; fi  # 判断返回值
done
```

和 Windows 版没有太大的不同，但需要注意的是，Linux 中的设备名和 Windows 有所不同，而且也没有必要执行类似@echo off 的命令——命令本来就不会回显。需要注意的是，如果在 Windows 下编写 Linux 脚本，复制到 Linux 后需要去掉所有的\r 字符，否则解释器会报错。

把上述程序保存成 test.sh 后，再执行 chmod +x test.sh，即可用./test.sh 来执行它。当然，扩展名也不是必需的，完全可以以不带扩展名的“test”命名。

上面的程序不是最简洁的（例如，可以直接把 diff 命令放在 if 语句中），但展示了 bash 脚本的一些其他用法。例如，while 循环是“while <命令集>; do <命令集>; done”，而 if 语句的基本是“if <命令集>; do <命令集>;”。不管是 while 还是 if，判断的都是命令集中最后一条语句的返回码（exit code）是否为 0。例如，若把上面的脚本改成 if diff output.a output.b; then break; fi，则当两个文件相同（diff 返回码为 0）时退出循环（这个不是我们所期望的）。如果忘记了命令格式，可以用 help if 和 help while 获取帮助。

顺便说一句，上面的“true”和“[”都是程序！前者的用途是什么都不做，直接返回 0；而后者的作用是计算表达式（该程序要求最后一个参数必须是“]”），其中“$?”是 bash 内部变量，表示“上一个程序的返回码”。

A.2.3　再谈随机数

如果做过测试，可能会发现上面的方法有一个问题：如果程序执行太快，随机数生成器在相邻两次执行时，time(NULL)函数返回值相同，因而产生出完全相同的输入文件。换句话说，我们每隔一秒才能产生出一个不同的随机数据。一个解决方案是利用系统自带的随机数发生器：在 Windows 下是环境变量%random%，而在 bash 中是$RANDOM。它们都是 0～32767 之间的随机整数。你可以直接用脚本编写随机数生成器，也可以把它们传递到你的程序中。

A.3　编译器和调试器

既然编译器和调试器都是程序，执行方法和普通程序没什么两样。在安装时，系统会自动把编译器和调试器程序所在路径加到搜索路径中，因此在执行时不必像./gcc 这样加上路径名。

A.3.1　gcc 的安装和测试

尽管在现场比赛中，编译器都已安装好，但如果平时在家练习，一般需要自己安装。如果使用 Linux，在安装操作系统时即可选择安装 gcc、g++、binutils 等包，但若要在 Windows 中使用 C/C++语言，需要手工安装编译器。

本书推荐使用 MinGW 环境下的 gcc，它的好处是和 Linux 下的 gcc 一致性较好，而且是免费的。可以到 www.mingw.com 中下载最新的安装包，然后在安装时选择 g++编译器。

安装完毕后，在命令行中执行 gcc 命令。如果显示“gcc: no input files”，则安装成功；如果提示不存在这个命令，可能是因为没有把 gcc 所在目录加到搜索路径中。可以双击控制面板的“系统”图标，并在“高级”选项卡中设置环境变量。在“系统变量”中找到“PATH”（大小写无所谓），它就是可执行程序的搜索路径。请在它的最后加入 MinGW 安装路径的 bin 子目录，如 C:\MinGW\bin（在安装时记住 MinGW 的安装路径），保存后重新启动命令行，gcc 就应该可以正常工作了。

A.3.2 常见编译选项

先建立一个 test.c，试试常见的编译选项。

```
#include<stdio.h>
main()
{
  int a, b;
  scanf("%d%d", &a, &b);
  int c = a+b;
  printf("%d%d\n", c);
}
```

编译一下试试：gcc test.c。程序没有输出，代表一切均好。检查目录（Windows 下用 dir，Linux 下用 ls），会发现多了一个 a.exe（Windows）或 a.out（Linux），这就是你的程序的编译结果。

下面的命令会让编译出的可执行程序名为 test.exe（Windows）或 test（Linux）：gcc test.c -o test。这样，你就能用 test（Windows）或./test（Linux）方式运行程序。

也许你已经看出了上述代码中的一些问题，不过当程序更加复杂时，人眼就不一定能快速找到错误了。在这样的情况下，编译选项能帮你的忙：gcc -test.c -o test -Wall。这次，编译器坐不住了，它指出了 3 个警告：main 函数没有返回类型、没有返回值、printf 的格式字符串可能有问题。你还可以进一步用-ansi-pedantic，它会检查代码是否符合 ANSI 标准（-ansi 只是判断是否和 ANSI 冲突，而-pedantic 更加严格）。它进一步指出了上述代码中的另外一个“问题”：ANSI C 中不允许临时声明变量，而必须在语句块的首部声明变量。

在 C 语言中，另一个常用的编译选项是-lm，它让编译器连接数学库，从而允许程序使用 math.h 中的数学函数。C++编译器会自动连接数学库，但如果你的程序扩展名是.c，且不连接数学库，有时会出现意想不到的结果。

另一个有用的选项是-DDEBUG，它在编译时定义符号 DEBUG（可以换成其他，如-DLOCAL 将定义符号 LOCAL），这样，位于#ifdef DEBUG 和#endif 中间的语句会被编译。而在通常情况下，这些语句将被编译器忽略（注意，不仅是不会执行，连编译都没有进行）。

可以用-O1、-O2 和-O3 对代码进行速度优化。一般情况下，直接编译出的程序比用-O1 编译出的程序慢，而后者比-O2 慢。尽管理论上-O3 编译出的程序更快，但由于某些优化可能会误解程序员的意思，一般比赛中不推荐使用。另外，如果你的程序中有一些不确定因素（如使用了未初始化的变量），运行结果可能会和编译选项有关——用-O1 和-O2 编译出的程序也许不仅是速度有差异，答案甚至都有可能不同！当然，这种情况出现的前提是你的程序有瑕疵。如果是一个规范的程序，运行结果不会和优化方式有关。

既然编译选项可以影响程序的行为，在正规比赛中，组织方应提前公布编译选项。如果没有公布，选手最好尽早询问。

A.3.3　gdb 简介

gdb 尽管只是一个文本界面的调试器，但功能十分强大。不管是 Linux 和 Windows 下的 MinGW，gcc 和 gdb 都是最佳拍档。

gdb 的使用方法很简单——用 gcc 编译成 test.exe 之后，执行 gdb test.exe 即可。不过请等一下！如果要用 gdb 调试，编译时应加上-g 选项，生成调试用的符号表。

接下来使用 l 命令，你将看到部分源程序清单。如果用 l 15，将会显示第 15 行（以及它前后的若干行）。除此之外，你还可以用函数名来定义，如 l main 将显示 main 函数开头的附近 10 行。如果不加参数执行 l，将显示下 10 行；list -将显示上 10 行。所有这些操作都可以用 help list 命令来查看。顺便说一句，gdb 中的命令可以简写（例如 list 简写成 l），大家可以多尝试（提示：试一下命令的前若干个字母）。

运行程序的命令是 r（run），但会一直执行到程序结束。如何让它停下来呢？方法是用 b（break）命令设置断点。例如，b main 命令将在 main 函数的开始处设置一个断点，则用 r 命令执行时会在这里停下来。如果想继续运行，请用 c（continue）命令，而不是继续用 r 命令。和 list 命令类似，b 既可以指定行号，也可以在指定函数的首部停下来。顺便说一句，笔者在调试很多程序时都是以命令 b main 和 r 开头的。

如果希望逐条语句地执行程序，不停地用 b 和 c 太麻烦。gdb 提供了一些更加方便的指令，其中最常用的有两个：next（简写为 n）和 step（简写为 s）。它们的作用都是执行当前行，区别在于如果当前行涉及函数调用，则 next 是把它作为一个整体执行完毕，而 step 是进入函数内部。尽管 n 和 s 都只有一个字母，但有时还是稍显繁琐。在 gdb 中，如果在提示符下直接按 Enter 键，等价于再次执行上一条指令，因此如果需要连续执行 s 或者 n，只需要第一次输入该命令，然后直接狂按 Enter 键即可。另外，和命令行一样，可以按上下箭头来使用历史记录。

另一个常用命令是 until（简写为 u），让程序执行到指定位置。例如 u 9 就是执行到第 9 行，u doit 就是执行到 doit 函数的开头位置。

停下来以后干什么呢？当然是打印一些函数值，看看是否和你想象的一致。用 p（print）命令可以打印出一些变量的值，而 info locals（可以简写为 i lo）可以显示所有局部变量。如果希望每次程序停下来，则可以用 display（简写为 disp）命令。例如，display i+1 就可以让你方便地读取 i+1 的值。它往往和 n、s 和 u 等单步执行指令配合使用。如果需要列出所有 display，可以用 info display（简写为 i disp）；还可以删除或者临时禁止/恢复一些 display，相应的命令为 delete display（d disp）、disable display（dis disp）和 enable display（en disp）。类似地，也可以根据断点编号删除、禁止和恢复断点，还可以用 clear（cl）命令，像 b 命令一样根据行号或者函数名直接删除断点。

在多数情况下，灵活运用上述功能已经能高效地调试程序了。下面把涉及的命令列表供读者参考，如表 A-2 所示。

表 A-2 gdb 常见命令

简　写	全　称	备　注
l	list	显示指定行号或者指定函数附近的源代码
b	break	在指定行号或者指定函数开头处设置断点。如 b main
r	run	运行程序，直到程序结束或者遇到断点而停下
c	continue	在程序中断后继续执行程序，直到程序结束或者遇到断点而停下。注意在程序开始执行前只能用 r，不能用 c
n	next	执行一条语句。如果有函数调用，则把它作为一个整体
s	step	执行一条语句。如果有函数调用，则进入函数内部
u	until	执行到指定行号或者指定函数的开头
p	print	显示变量或表达式的值
disp	display	把一个表达式设置为 display，当程序每次停下来时都会显示其值
cl	clear	取消断点，和 b 的格式相同。如果该位置有多个断点，将同时取消
i	info	显示各种信息。如 i b 显示所有断点，i disp 显示 display，而 i lo 显示所有局部变量

别忘了，如果对上述解释有疑问，可输入“help”以获得详尽的帮助信息。

A.3.4 gdb 的高级功能

gdb 的功能远不止刚才所讲述的那些。尽管很多功能是专为系统级调试所设，但还有很多功能也能为算法程序的调试带来很大方便。

首先是栈帧的相关命令，其中最常用的是 bt，其他命令可以通过 help stack 来学习。接下来是断点控制命令。commands（comm）命令可以指定在某个断点处停下来后所执行的 gdb 命令，ignore（ig）命令可以让断点在前 count 次到达时都不停下来，而 condition 则可以给断点加一个条件。例如，在下面的循环中：

```
10  for(i = 0; i < n; i++)
11  printf("%d\n", i);
```

首先用 b 11 设置断点（假设编号为 2），然后用 cond 2 i==5 让该断点仅当 i=5 时有效。这样的条件断点在进行细致的调试时往往很有用。

另外，gdb 还支持一种特殊的断点——watchpoint。例如 watch a（简写为 wa a）可以在变量 a 修改时停下，并显示出修改前后的变量值，而 awatch a（简写为 aw a）则是在变量被读写时都会停下来。类似地，rwatch a（rw a）则是在变量被读时停下。

最后需要说明的是，gdb 中可以自由调用函数（不管是源程序中新定义的函数还是库函数）。第一种方法是用 call 命令。例如，如果你想给包含 10 个元素的数组 a 排序，可以像这样直接调用 STL 中的排序函数 call sort(a, a+10)。

遗憾的是，你如果真的做过实验，会发现刚才所说完全是骗人的。gdb 会冷冷地告诉你：不存在函数 sort。怎么会这样呢？如果学过宏和内联函数你就会知道，很多看起来是函数的东西不一定真的是函数，或者说，不一定是调试器心目中的函数。为了在 gdb 中调用 sort，

可以将它打包：

```
void mysort(int*p, int*q)
{
    sort(p, q);
}
```

这样，就可以用 call mysort(a, a+10)来给数组 a 排序了。顺便说一句，print 命令、condition 命令和 display 命令都可以像这样使用 C/C++函数。例如，可以用 p rand()来输出一个随机数，或是专门编写一个打印二叉树的函数，然后在 print 或者 display 命令中使用它，还可以编写一个返回 bool 值的函数，并作为断点的条件。

怎么样，是不是觉得 gdb 很强大呢？注意，过分地依赖于 gdb 的调试功能并不是一件好事——它会让你敏锐的直觉变得迟钝起来。事实上，笔者建议读者尽量只使用 A.3.3 节提到的基本功能，甚至尽量不要使用 gdb——用输出中间变量的方法，加上直觉和经验来调试算法程序。如果是这样的话，不仅编程速度和准确性将大大提高，而且选手也就不会成天抱怨：这次比赛提供的 IDE 太难用啦，发挥得不好全赖它！如果你还不太明白我在说什么，想想对小学生的忠告：不要只会按计算器，连九九表都不会背了。

A.4　浅谈 IDE

所谓 IDE，是指集成开发环境（Integrated Development Environment）。顾名思义，开发程序所用到的各种功能都应该被集成到 IDE 中，包括编辑（edit）、编译（compile）、运行（run）、调试（debug）等。但工具始终总是工具，读者必须懂得如何使用它，才能发挥出它的最大威力。

可以用来编写 C/C++程序的 IDE 有很多，如 Linux 下的 Anjuta，Windows 下的 Dev-Cpp，以及跨平台的 Eclipse 和 Code::Blocks，还有一些强大的通用编辑器也可以用来编写 C/C++程序，如 vi、emacs、EditPlus 等。

也许和很多读者所期望的不同：笔者在这里不打算介绍任何一个 IDE。事实上，如果读者对本章所介绍的命令行、脚本、编译选项和 gdb 都能很好地掌握，IDE 是非常容易学习的——只需要熟悉它的编辑特色（语法高亮、代码折叠、查找与替换和代码补全等）和常用快捷键即可。

多数 IDE 会引入“工程”的概念，所以读者需要花一点时间来掌握工程的基本知识。例如，在编写算法程序时，工程类别需要的是命令行程序（console application），而不是图形界面程序（GUI application）或其他。如果熟练掌握了 gcc 编译参数和 gdb 的常见命令，在 IDE 下编译和调试更是小事一桩。

[General Information]
书名=算法竞赛入门经典
作者=刘汝佳编著
页数=225
出版社=北京市：清华大学出版社
出版日期=2009.10
SS号=12414711
DX号=000006813950
URL=http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000006813950&d=3624C49D2F6D657FDC47043BBB2F0BDB