

王道论坛计算机考研机试指南



王道论坛

2013.01.06

写在前面的话

各位王道的小崽子们，今天你们考完初试了，感觉解放了吧？轻松了吧？无论结果如何，总算坚持到了最后。但是，其实你的考研生活只刚刚走出了第一步，接下来会有初试成绩出来前的煎熬、分数线出来的煎熬、准备复试以及复试的煎熬以及录取结果出来前的煎熬，这些都远远比初试更折磨人，未来的两个月你会感觉到王道没有吓唬你们。

王道是个好姑娘，四年多的时光里陪伴了接近二十万计算机考研人，不离不弃。今年不小心又压中一道算法题，说实话，王道的书里有那么多的题，知识点又只有那么多，总能瞎猫碰见死耗子吧？**王道尊重的不是考研这个行业，而是你们这群执着的小崽子们的梦想！**看着你们圆梦，我们内心充满了成就感。

初试考完了，是不是应该好好放松放松？是不是初试考得好，录取就肯定没有问题了？对不起，这个不是计算机专业研究生考试的规则。目前已经有越来越多的高校采用上机考试的形式来考察考生的实际动手编程能力，并且机试在复试中所占的比例非常高，并且很多高校规定复试成绩不及格者，一律不得录取。目前国内高校开展 ACM 教学的高校非常少，而 ACM 是目前所有高校机试所采取的唯一形式，因此**提早开始准备和练习，对于一个完全没有接触过 ACM 的计算机考研人来说，是必须的！**

为了方便各位道友练习机试，我们编写了本书，搭建了九度Online Judge (<http://ac.jobdu.com>)，并收集了全国各大高校的复试上机真题，希望能给大家复试上机考试提供强有力的支持。你可以直接使用王道论坛的帐号进行登录。如果您在使用过程中遇到问题，欢迎你到复试机试讨论专区发帖提出。目前已经收录了我们能够收集到的各高校上机复试真题，欢迎大家继续向我们提供各高校上机真题，具体请站内信或者电子邮件联系浩帆 (Email:qihu#zju.edu.cn)。此外，华科的上机题我们经过了变型，将其中一些便于修改成OJ判题的题目收录进了我们的OJ。

考研其实没有什么诀窍，就是每天比别人早起一点，晚睡一点，比别人早准备一点，勤奋一点。考研离我已经很远了，同时我也坚信一个写不出合格代码的计算机专业的学生，即使考上了研究生，无非也只是给未来失业判个缓期执行而已。

小崽子们，要忠实于自己心底的梦想，勇敢地坚持下去，而当下，请开始准备复试吧，熬过这两个月，一切就都好了。

第1章 从零开始

一 机试的意义

众所周知，机试是计算机考研当中非常重要的一个环节。在越来越注重实践动手能力的今天，越来越多的知名高校在计算机研究生招生考试当中采用了机试的形式，通过这种考试手段来考察考生分析问题并利用计算机程序解决问题的能力。通过机试，可以考察一个考生从实际问题当中抽象得出数学模型的能力，利用所学的计算机专业知识对该模型进行分析求解的能力，以及利用计算机编程语言，结合数据结构和算法真正解决该实际问题的能力。

所以，我们在准备机试的过程中要特别注意以下几个方面：

1、如何将一个实际问题抽象成数学问题。例如将高速公路网抽象成带权图，这就是一种简单的、直接的抽象。

2、如何将我们所学的计算机专业知识运用到解决抽象出来的数学模型上去。这就要求我们在脑子里事先熟知一些常用的数据结构和算法，再结合模型求解的要求，很快地选择合适的编程思想来完成算法的设计。甚至可以利用一些经典算法特征，加入一些自己的优化，使得编写的程序更优雅、更高效（当然这是建立在充分理解经典算法的基础上）。

3、如何将我们为了解决该数学模型所设计的算法编写成一个能被计算机真正执行的计算机程序。我们认为，关于这个能力的定义有三个层次：1）会编写（默写）一些经典算法的程序代码。2）能够将自己的想法或设计的算法转换为程序代码。3）能够使得自己编写的程序在大量的、多种多样的、极限的测试数据面前依旧正常完成功能（程序的健壮性）。我们在准备机试的训练过程中，就要依次经历这三个层次，从而最后能够在实际考试当中取得理想的成绩。

本教程从分析经典机试真题出发，引入近几年频繁被考察的数据结构和算法，利用 C/C++ 语言讲解例题，并加以一些相关知识的扩展，希望在读者准备计算机考研机试的过程中充当指引者的角色。同时，由于笔者自身实力的限制以及编写时间的不足，教程中难免存在一些疏漏和错误，也欢迎读者提出、指正。

二 机试的形式

绝大部分机试所采用的形式，归结起来可以概括为：得到题目后，在计算机上完成作答，由计算机评判并实时告知结果的考试过程。

机试考试中的问题往往有五部分组成。首先是问题描述，问题描述描述该问题的题面，题面或直接告知考生所要解决的数学问题或给出一个生活中的实际案例，以待考生自己从中抽象出所要解决的数学模型。第二是输入格式，约定计算机将要给出的输入数据是以怎样的顺序和格式向程序输入的，更重要的是它将给出输入数据中各个数据的数据范围，我们通过这些给出的数据范围确定数据的规模，为我们设计算法提供重要依据。第三是输出格式，明确考生将要编写的程序将以怎样的顺序和格式向输出输出题面所要求的答案。第四第五部分即输入、输出数据举例 (Sample)。好的 Sample 不仅能为考生提供一组简单的测试用例，同时也能明确题意，为题面描述不清或有歧义的地方做适当的补充。

另外我们也要特别注意，题目中给定的两个重要参数：1、时间限制。2、空间限制。这两个重要的参数限定了考生提交的程序在输出答案之前所能耗费的时间和空间。

我们来看一个典型的题目描述，从而了解机试题的问题形式。

例 1.1 计算 A+B (九度 OJ 题号: 1000)

时间限制: 1 秒

内存限制: 32 兆

特殊判题: 否

题目描述:

求整数 a , b 的和。

输入:

测试案例有多行，每行为 a , b 的值， a , b 为 int 范围。

输出:

输出多行，对应 $a+b$ 的结果。

样例输入:

1 2

4 5

6 9

样例输出:

3

9

15

通过该例，我们基本明确了机试试题的问题形式，以及问题各部分所起到的作用。这里补充解释一下所谓特殊判题 (Special Judge) 的含义，特殊判题常被应用在可能存在多个符合条件的答案的情况下，若评判系统采用了特殊判题，那么系统只要求输出任何一组解即可；若系统对该题并没有采用特殊判题，那么你必须严格按照题目中对输出的限定输出对应的答案 (如输出字典序最小的解)。

得到题目后，考生在计算机上立即编写程序，确认无误后，将该程序源代码提交给评判系统。评判系统将考生提交的源代码编译后，将后台预先存储的输入测试数据输入考生程序，并将该程序输出的数据与预先存储在评判系统上的“答案”进行比对得出结果。

评判系统评判考生程序后，实时地将评判结果返回考生界面，考生可以根据该结果了解自己的程序是否被评判系统判为正确，从而根据不同的结果继续完成考试。

三 评判结果

本节将对评判系统评判考生提交程序后返回的结果做详细的说明，并且针对不同的返回结果，对可能出现错误的地方作出初步的界定。

Accepted (答案正确): 你的程序对所有的测试数据都输出了正确的答案，你已经得到了该题的所有分数，恭喜。

Wrong Answer (答案错误): 评判系统测试到你的程序对若干组（或者全部）测试数据没有输出正确的结果。

出现该种错误后，一般有两种解决方向：如果对设计的算法正确性有较大的把握，那么你可以重点考虑代码健壮性，即是否存在某些特殊数据使程序出现错误，比如边界数据，比如程序中变量出现溢出。另一种方向，即怀疑算法本身的正确性，那么你就需要重新考虑你的算法设计了。

Presentation Error (格式错误): 评判系统认为你的程序输出“好像”是正确的，只是没有严格按照题目当中输出所要求的输出格式来输出你的答案，例如你忽略了题目要求在每组输出后再输出一个空行。

出现这种错误，往往预示着你离完全正确已经不远了，出现错误似乎只是因为多输出了一些空格、换行之类的多余字符而已。但这不是绝对的，假如在排版题（后文会有介绍）中出现格式错误，那么有可能你离正确的答案仍然有一定的距离。

Time Limit Exceeded (超出时间限制): 你的程序在输出所有需要输出的答案之前已经超过了题目中所规定的时间。

若这种结果出现在你的评判结果里，依然有两种方向可供参考：1、假如你确定算法时间复杂度能够符合题目的要求，那么依旧可以检查是否程序可能在某种情况下出现死循环，是否有边界数据可能会让你的代码不按照预想的工作，从而使程序不能正常的结束。2、你设计的算法时间复杂度是否已经高于题目对复杂度的要求，如果是这样，那么你需要重新设计更加高效的算法或者对你现行的算法进行一定的优化。

Runtime Error (运行时错误): 你的程序在计算答案的过程中由于出现了某种致命的原因异常终止。

你可以考虑以下几个要点来排除该错误：1、程序是否访问了不该访问的内存地址，比如访问数组下标越界。2、程序是否出现了除以整数 0，从而使程序异常。3、程序是否调用了评判系统禁止调用的函数。4、程序是否会出现因为递归过深或其他原因造成的栈溢出。

Compile Error (编译错误): 你提交的程序并没有通过评判系统的编译，可根据更详细的编译信息修改你的程序。

Memory Limit Exceeded (使用内存超出限制): 你提交的程序在运行输出所有的答案之前所调用的内存已经超过了题目中所限定的内存限制。

造成这种错误的原因主要有两个方面：1、你的程序申请过多的内存来完成所要求的工作，即算法空间复杂度过高。2、因为程序本身的某种错误使得程序不断的申请内存，例如因为某种原因出现了死循环，使得队列中不断的被放入元素。当然也千万别忽略自己的低级错误，比如在声明数组大小时多打了一个 0。

Output Limit Exceeded (输出超出限制): 你的程序输出了过多的东西，甚至超出了评判系统为了自我保护而设定的被评判程序输出大小的最高上限。

一般来说该种错误并不常见，一旦出现了也很好找原因。要么就是你在提交时忘记关闭你在调试时输出的调试信息（我经常输出 DP 时的数组来动态的观察状态的转移）；要么就是程序的输出部分出现了死循环，使得程序不断地输出而超出系统的限制。

以上几种结果就是评判系统可能会返回的几个最基本的结果。若返回 **Accepted**，则你可以获得该题的所有分数。若返回其它错误，则根据不同的考试规则，你的得分将会有一定的差异。若你参加的考试采用按测试点给分规则，你依然能够获得你通过的测试点（即该程序返回正确结果的那部分测试数据）所对应的分数；但是，若你参加考试采用所有数据通过才能得分的评分规则，那么很可惜，到目前为止你在这道题上的得分依旧是 0 分。

假如评判结果显示你提交的程序错误的，你可以在修改程序后再次提交该题，直到获得满意的分数或者放弃作答该题。

四 复杂度的估计

本节将详细讨论题目中所给定的时间限定和空间限定对我们程序设计的指导作用。

如例 1.1 所示，该题给予我们的程序 1 秒的运行时限，这也是最常见的时间限制（或最常见的时间限制数量级）。对于该时限，通常，我们所设计的算法复

复杂度不能超过百万级别，即不能超过一千万。即若算法的时间复杂度是 $O(n^2)$ ，则该 n （往往在题目中会给出数据范围）不应大于 3000，否则将会达到我们所说的千万数量级复杂度，从而程序运行时间超出题目中给出的用时限定。举例来说，我们不能在 1 秒时限的题目当中对 10000 个整数进行冒泡排序，而必须使用快速排序等时间复杂度为 $O(n\log n)$ 的排序算法，否则程序很可能将会得到运行时间超出限制的评判结果。因此你可以对你的程序在最坏情况下的复杂度进行一个估算，假如确定其在百万数量级之内，那么你的程序一般是不会超出时间限制的。对于其它时间限制的情况，可以参考 1 秒时限对时间复杂度的要求，做出一定的估计，从而保证自己的程序运行所需的时间不会超过题目中对运行时间的限制。

我们同样可以知道，例 1.1 中限定的内存空间为 32 兆，即你的程序在评测系统中运行时，不得使用超过 32 兆大小的内存。空间限定则比较好处理，你可以简单的计算你所申请的内存空间的大小（例如我们可以轻易的计算 `int mat[300][300]` 所占用的内存大小）。只要该大小没有超过或过分接近空间限定（运行时需要一些额外的空间消耗），那么你的程序应当是符合空间限制条件的。现今的机试题，一般不会对空间做过多的限制，大多数情况只对时间做出明确的要求，所以考题在空间上将会尽量满足考生程序的需要。正因为如此，我们在很多情况下都应该有“空间换时间”的思想。

五 OJ的使用

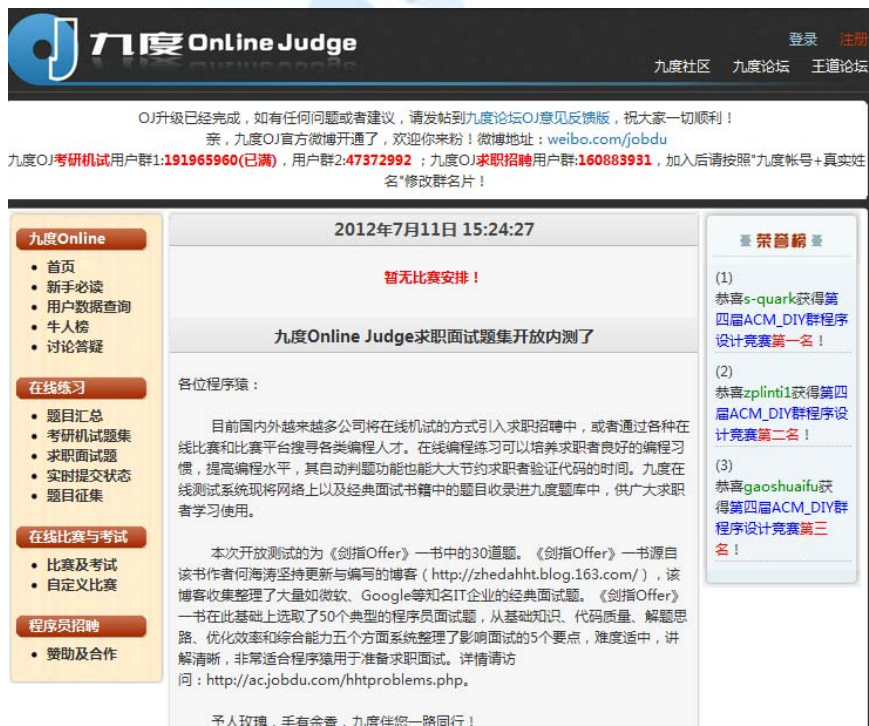
我们该去何处开始我们的机试训练和模拟考试呢？可能大部分人已经听说过在线评判系统（Online Judge），例如 HDOJ（<http://acm.hdu.edu.cn/>），POJ(<http://poj.org/>)。但是这些OJ都是为ACM/ICPC训练而设计的，虽然形式与机试大同小异但是难度却有较大差别，普通的考生在其中训练不仅打击了信心，也在一定程度上浪费了时间。因此，推荐专门为我们计算机考研机试所设计的九度OJ（<http://ac.jobdu.com/>），它收录了近三百道各大高校近年来的机试真题，真正为我们准备机试提供了极大的便利。

我们在它的首页上可以看到九度 OJ 所包含的各种功能。

首先,我们需要点击“注册”按钮,为自己注册一个账号,有了该账号你就能在九度 OJ 上进行日常练习,同样也有资格参加九度 OJ 举行的各类比赛和模拟考试。

现假设我们已经有有了一个账号,并且想要开始在线练习。那么,可以在页面左边的导航栏找到在线练习,我们可以选择题目汇总来查看九度 OJ 收录的所有在线练习题,或者点击考研机试题集来查看九度为各位考生收录的近几年各大高校机试真题。例如我们选择题目汇总中的题号为 1000 的问题,例 1.1 中的例子便出现在页面当中。当用户完成作答以后,可以在页面左侧点击“提交答案”按钮,将源代码提交给九度在线评测系统,提交完成后,系统会自动跳转到评测状态页面,通过查看你刚才提交的源代码的评测状态,你就可以知道你是否通过了该题,从而达到锻炼自我的目的。在本文中,凡是涉及九度 OJ 上收录的例题,本书会明确该题的题号。

同样的,九度 OJ 也经常为广大考生提供在线模拟考试训练。我们可以点击“比赛及考试”按钮,来查看过往的比赛以及近期的比赛安排,从而有选择的参加在线比赛,考察自己的训练质量,为日后进一步训练做出导向



九度Online Judge

登录 注册

九度社区 九度论坛 王道论坛

OJ升级已经完成,如有任何问题或者建议,请发帖到九度论坛OJ意见反馈版,祝大家一切顺利!

亲,九度OJ官方微博开通了,欢迎你来粉!微博地址:weibo.com/jobdu

九度OJ考研机试用户群1:191965960(已满),用户群2:47372992;九度OJ求职招聘用户群:160883931,加入后请按照“九度帐号+真实姓名”修改群名片!

2012年7月11日 15:24:27

暂无比赛安排!

九度Online Judge求职面试题集开放内测了

各位程序员:

目前国内外越来越多公司将在线机试的方式引入求职招聘中,或者通过各种在线比赛和比赛平台搜寻各类编程人才。在线编程练习可以培养求职者良好的编程习惯,提高编程水平,其自动判题功能也能大大节约求职者验证代码的时间。九度在线测试系统现将网络上以及经典面试题集中的题目收录进九度题库中,供广大求职者学习使用。

本次开放测试的为《剑指Offer》一书中的30道题。《剑指Offer》一书源自该书作者何海涛坚持更新与编写的博客(<http://zhedahht.blog.163.com/>),该博客收集整理了大量如微软、Google等知名IT企业的经典面试题。《剑指Offer》一书在此基础上选取了50个典型的程序员面试题,从基础知识和代码质量、解题思路、优化效率和综合能力五个方面系统整理了影响面试的5个要点,难度适中,讲解清晰,非常适合程序员用于准备求职面试。详情请访问: <http://ac.jobdu.com/hhtproblems.php>。

予人玫瑰,手有余香,九度伴您一路同行!

荣誉榜

(1)
恭喜s-quark获得第四届ACM_DIY群程序设计竞赛第一名!

(2)
恭喜zplint1获得第四届ACM_DIY群程序设计竞赛第二名!

(3)
恭喜gaoshuaifu获得第四届ACM_DIY群程序设计竞赛第三名!

总结

本节主要介绍机试的意义、形式,以及机试题的形式和考察的方法,并在最

后给出了我们可以完成机试训练的网站——九度 Online Judge。



王道论坛

第2章 经典入门

本章通过对频繁出现在考研机试真题中的一些经典问题进行讲解,并对解题方法做详细的说明,旨在使读者快速的了解机试的考察形式和答题方法,以便进一步的学习。

一 排序

排序考点在历年机试考点中分布广泛。排序既是我们必须要掌握的基本算法,同时也是学习其他大部分算法的前提和保证。

首先我们来学习对基本类型的排序。所谓对基本类型排序,即我们对诸如整数、浮点数等计算机编程语言内置的基本类型进行排序的过程。

例 2.1 排序 (九度教程第 1 题)

时间限制: 1 秒

内存限制: 32 兆

特殊判题: 否

题目描述:

对输入的 n 个数进行排序并输出。

输入:

输入的第一行包括一个整数 $n(1 \leq n \leq 100)$ 。接下来的一行包括 n 个整数。

输出:

可能有多组测试数据,对于每组数据,将排序后的 n 个整数输出,每个数后面都有一个空格。每组测试数据的结果占一行。

样例输入:

4

1 4 3 2

样例输出:

1 2 3 4

来源:

2006 年华中科技大学计算机保研机试真题

这便是一例曾经出现在机试中关于排序考点的真题。面对这样的考题,读者可能很快就会联想到《数据结构》教科书上各种形形色色、特点不同的排序算法。例如,我们可以自然的想到选择冒泡排序来完成此题。但是在确定使用冒泡排序之前,我们不应忽略对其复杂度的考量,我们应特别注意在选择将要采用的算法

时估计其复杂度。以此处为例，假如我们采用冒泡排序来完成此题，我们应该注意到冒泡排序的时间复杂度为 $O(\text{待排序个数的平方})$ ，在此例中即 $O(n^2)$ 。而 n 的取值范围也在题面中明确地给出 ($1 \leq n \leq 100$)，这样我们可以估算出 n^2 的数量级仅在万级别，其时间复杂度并没有超过我们在前一章中所讲的百万数量级复杂度，所以使用冒泡排序在该例限定的一秒运行时间里是完全可以接受的；同时冒泡排序的空间复杂度为 $O(n)$ ，即大致需要 $100 * 32\text{bit}$ （数组长度 * `int` 所占内存）的内存，这样，所需的内存也不会超过该例限定的内存大小（32 兆）。只有经过这样的复杂度分析，我们才能真正确定冒泡排序符合我们的要求。于是，我们可以开始着手编写该例的解题代码。

代码 2.1

```
#include <stdio.h>

int main () {
    int n;

    int buf[100]; //定义我们将要使用的变量n, 并用buf[100]来保存将要排序的数字
    while (scanf ("%d", &n) != EOF) { //输入n, 并实现多组数据的输入
        for (int i = 0; i < n; i++) {
            scanf ("%d", &buf[i]);
        } //输入待排序数字

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n - 1 - i; j++) {
                if (buf[j] > buf[j + 1]) {
                    int tmp = buf[j];
                    buf[j] = buf[j + 1];
                    buf[j + 1] = tmp;
                }
            }
        } //冒泡排序主体

        for (int i = 0; i < n; i++) {
            printf("%d ", buf[i]);
        } //输出完成排序后的数字, 注意, 题面输出要求在每个数字后都添加一个空格
        printf("\n"); //输出换行
    }

    return 0;
}
```

初见这段代码，读者可能对主循环体的循环条件大为不解。`while (scanf`

`(" %d", &n) != EOF)`，它在完成输入`n`的同时又完成了什么看起来令人迷惑的条件判断？要回答这个问题，我们首先要明确两点：第一，`scanf`函数是有返回值的（尽管大多时候都会被人忽略），它将返回被输入函数成功赋值的变量个数。在此例中，若成功完成输入并对`n`赋值，`scanf`函数的返回值即为1。我们即是通过该返回值来完成循环条件的判断的。第二，我们也要注意，该例题面中明确了输入数据会有多组，我们将要对每组输入都输出相应答案。并且，事先我们并不知道将会有多少组数据被输入到程序中。于是，我们可以使用该循环测试条件来完成对输入是否结束的判断。过程如下：如仍有测试数据未被测试完，那么该组测试的开头一行将如题面所说，为一个整数（`n`），其将会被`scanf`语句赋值给变量`n`，那么`scanf`返回值为1，不等于`EOF`（-1），循环条件成立，程序进入循环体，执行程序；如果输入已经到达结尾（输入文件到达末尾或在命令台输入中输入`Ctrl+z`），`scanf`函数无法再为变量`n`赋值，于是`scanf`函数返回`EOF`（end of file）。至此，循环条件不成立，程序跳过循环体，执行`return 0`，使程序正常的结束。该循环判断条件既保证了可以对多组测试数据进行计算，同时又使程序在输入结束后能够正常的退出。反之，如果我们不使用循环，程序在处理完一组数据后就会退出，后续测试数据无法被正常的处理。但若我们使用死循环而不加以任何退出条件，那么程序在处理完所有的测试数据后仍就不能正常的退出，虽然程序已经输出了所有测试数据的答案，但评判系统还是会认为程序在限制时间内无法运行完毕，于是返回超时的评判结果。正因为如此，初学者很容易因为这个原因，出现莫名其妙的程序超时。所以，我们必须使用该循环判断条件来完成以上两个功能。

值得一提的是，若输入为字符串而程序采用`gets()`的方法读入，则相同功能的循环判断语句为`while(gets(字符串变量))`。

解决了关于循环测试条件的疑惑，对于代码其它部分，相关注释已在代码中给出，读者应该非常容易理解它的工作流程。

另外需要指出的是，假如你使用`vc6.0`编译这段程序，很可能会出现编译错误。这是因为`vc6.0`对`for`循环测试条件中定义的指示变量`i`作用域的不同规定造成的。`C++`标准指出，在`for`循环循环条件中定义的变量，其作用域仅限于`for`循环循环体内部（就像在`for`循环循环体内定义的局部变量）。于是我们在多个`for`循环中都重新定义指示变量`i`来完成相应的功能。而`vc6.0`中则不同，在`for`循环退出以后，指示变量`i`依然可见（就像在`for`循环循环体外定义的局部变量）。所以我们在后续`for`循环循环条件中定义的新指示变量`i`会与该旧的指示变量`i`产生冲突，而无法被成功的定义，于是编译器于此给出了编译错误。因此，我个人并不推荐使用`vc6.0`编译器。假如你在练习和考试中，不得不使用该编译器，那么请在`for`循环

循环条件第一次定义指示变量*i*后，不必重新定义，而是直接使用这个我们已经定义的并且依旧可见的变量来完成新的for循环功能（莫忘初始化）。

在明确以上注意点，我们重新回到排序问题本身上来。该例程使用冒泡排序来完成题目要求。但假如我们修改题目中*n*的取值范围，使其最大能达到10000，我们又应该如何解决该问题呢。读者可能很快就能反应过来，冒泡排序因其较高的时间复杂度（ $O(10000 * 10000)$ 已经超过了百万数量级）而不能再被我们采用，于是我们不得不使用诸如快速排序、归并排序等具有更优复杂度的排序算法。那么，新问题出现了，恐怕大部分读者都对此感到恐惧：是否能正确的写出快速排序、归并排序等的程序代码？其实，我们并不用为此感到担心，C++已经为我们编写了快速排序库函数，我们只需调用该函数，便能轻易的完成快速排序。

代码 2.2

```
#include <stdio.h>
#include <algorithm>
using namespace std;
int main () {
    int n;
    int buf[10000];
    while (scanf ("%d", &n) != EOF) {
        for (int i = 0; i < n; i++) {
            scanf ("%d", &buf[i]);
        }
        sort (buf, buf + n); //使用该重载形式, 表明将要使用自己定义的排列规则
        for (int i = 0; i < n; i++) {
            printf("%d ", buf[i]);
        }
        printf("\n");
    }
    return 0;
}
```

因为我们将要在程序中使用了 `sort` 库函数，我们在头文件方面包含了 `algorithm` 头文件，并使用 `using namespace std` 语句声明了我们将会使用标准命名空间（`sort` 被定义在其中）。此例中，`sort` 函数的两个参数代表待排序内存的起始地址和结束地址（`sort` 函数还有另外的重载形式，在后文中会有所提及），在此例中起始地址为 `buf`，结束地址为 `buf + n`。该函数调用完成后，`buf` 数组中的数字就已经通过快速排序升序排列。我们只需对其输出即可。

可能习惯于用 C 来编写程序或对 C++ 不很熟悉的读者会对某些 C++ 特性感到有所困难。那么，关于 `sort`，我给出两种建议：要么记住 `sort` 并不复杂的基本用法；要么使用 C 中快速排序的函数 `qsort`（可自行学习）。

那么假如我们将要对给定的数组进行降序排列该如何操作呢？有一个简单的方法，我们只需简单的将升序排列后的数组倒序，便能得到降序排列的数组。但是这里，我们还要介绍另一种使用 `sort` 函数的方法来完成降序排列。

代码 2.3

```
#include <stdio.h>
#include <algorithm>
using namespace std;

bool cmp (int x, int y) { //定义排序规则
    return x > y;
}

int main () {
    int n;
    int buf[100];
    while (scanf ("%d", &n) != EOF) {
        for (int i = 0; i < n; i++) {
            scanf ("%d", &buf[i]);
        }
        sort (buf, buf + n, cmp); //使用该重载形式, 我们表明将要使用自己定义的排列规则
        for (int i = 0; i < n; i++) {
            printf ("%d ", buf[i]);
        }
        printf ("\n");
    }
    return 0;
}
```

如该代码所示，我们新定义了一个 `cmp` 函数，来实现对于新的排序规则的定义。关于 `cmp` 函数的定义规则我们只需简单的记得，当 `cmp` 的返回值为 `true` 时，即表示 `cmp` 函数的第一个参数将会排在第二个参数之前（即在我们定义的规则中，`cmp` 表示第一个参数是否比第二个参数大，若是，则排在前面）。为了实现降序排列，我们只要判断两个参数的大小，当第一个参数比第二个参数大时返回 `true`。然后，只需要调用 `sort` 函数的另一种重载方式：`sort`（排序起始地址，排序

结束地址，比较函数)，如代码 2.3 所示，我们便完成了对我们给定的内存区间的降序排列。

对于利用 `sort` 函数为其它基本类型（如 `double`，`char` 等）排序的方法也大同小异，读者可以自己动手进行尝试。

使用内置函数 `sort` 的方法，简单易懂，且不易出错，对各种类型的排序写法也大同小异。所以，强烈推荐大家使用 `sort` 来完成排序。

例 2.2 成绩排序（九度教程第 2 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

有 N 个学生的数据，将学生数据按成绩高低排序，如果成绩相同则按姓名字符的字母序排序，如果姓名的字母序也相同则按照学生的年龄排序，并输出 N 个学生排序后的信息。

输入：

测试数据有多组，每组输入第一行有一个整数 N ($N \leq 1000$)，接下来的 N 行包括 N 个学生的数据。每个学生的数据包括姓名（长度不超过 100 的字符串）、年龄（整数数）、成绩（小于等于 100 的正数）。

输出：

将学生信息按成绩进行排序，成绩相同的则按姓名的字母序进行排序。然后输出学生信息，按照如下格式：姓名 年龄 成绩

样例输入：

```
3
abc 20 99
bcd 19 97
bed 20 97
```

样例输出：

```
bcd 19 97
bed 20 97
abc 20 99
```

提示：

学生姓名的字母序区分字母的大小写，如 `A` 要比 `a` 的字母序靠前(因为 `A` 的 ASCII 码比 `a` 的 ASCII 码要小)。

来源：

2000 年清华大学计算机研究生机试真题

如读者所见，在该例中，我们不再对基本类型进行排序，而是对一些由基本

类型组成的结构体进行排序。排序的规则在题面中已经明确给出。虽然该例中待排序的个体的规模（N）表明我们可以使用冒泡排序，但这里我们依旧采用内置的 `sort` 函数来完成解题，从而体会其方便快捷的特点。

代码 2.4

```
#include <stdio.h>
#include <algorithm>
#include <string.h>
using namespace std;
struct E {
    char name[101];
    int age;
    int score;
}buf[1000];
bool cmp(E a,E b) { //实现比较规则
    if (a.score != b.score) return a.score < b.score; //若分数不相同则分数低
者在前面
    int tmp = strcmp(a.name,b.name);
    if (tmp != 0) return tmp < 0; //若分数相同则名字字典序小者在前面
    else return a.age < b.age; //若名字也相同则年龄小者在前面
}
int main () {
    int n;
    while (scanf ("%d",&n) != EOF) {
        for (int i = 0;i < n;i++) {
            scanf ("%s%d%d", buf[i].name, &buf[i].age, &buf[i].score);
        } // 输入
        sort(buf, buf + n, cmp); //利用自己定义的规则对数组进行排序
        for (int i = 0;i < n;i++) {
            printf ("%s %d %d\n", buf[i].name, buf[i].age, buf[i].score);
        } //输出排序后结果
    }
    return 0;
}
```

首先，我们定义了用来表示学生个体的结构体，其包含三个数据域：姓名、年龄、成绩，并由输入部分对其分别进行赋值。由于这是我们自己定义的结构体，

不属于 C++ 的内置基本类型，计算机并不知道两个结构体之间的如何比较大小，更不可能自动对其进行排序。于是，我们要做的即是向计算机说明该排序依据。与上一例相同，我们采用定义一个比较函数 `cmp` 的方法来实现，并在其中对规则的各要素进行详细的说明（如代码 2.4 所示）。一旦建立了 `cmp` 函数，该函数便可作为计算机对其进行排序的依据，所以我们依然可以使用 `sort` 函数对其进行快速排序，但是不要忘记，将 `sort` 的第三个参数设定为 `cmp` 函数，使 `sort` 函数知道该应用什么规则对其进行定序。假如，你忽略了第三个参数，那么系统会以不知道排序规则为由（找不到合适的“小于”运算符）而给出编译错误。

同样的，与编写 `cmp` 类似，我们也可以直接定义该结构体的小于运算符来说明排序规则。

代码 2.5

```
#include <stdio.h>
#include <algorithm>
#include <string.h>
using namespace std;
struct E {
    char name[101];
    int age;
    int score;
    bool operator < (const E &b) const { //利用C++算符重载直接定义小于运算符
        if (score != b.score) return score < b.score;
        int tmp = strcmp(name, b.name);
        if (tmp != 0) return tmp < 0;
        else return age < b.age;
    }
}buf[1000];

int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        for (int i = 0; i < n; i++) {
            scanf ("%s%d%d", buf[i].name, &buf[i].age, &buf[i].score);
        }
        sort(buf, buf + n);
        for (int i = 0; i < n; i++) {
```

```
        printf ("%s %d %d\n", buf[i]. name, buf[i]. age, buf[i]. score);
    }
}
return 0;
}
```

由于已经指明了结构体的小于运算符,计算机便知道了该结构体的定序规则 (sort 函数只利用小于运算符来定序,小者在前)。于是,我们在调用 sort 时,便不必特别指明排序规则 (即不使用第三个参数),只需说明排序起始位置和结束位置即可。虽然,该方法与定义 cmp 函数的方法类似,我个人还是建议使用第二种方法。这样,首先可以对重载算符的写法有一定的了解;其次,在今后使用标准模板库时,该写法也有一定的用处。后文中,将会广泛使用该方法编写例题。若读者坚持使用定义 cmp 函数的写法,只需自行将重载小于运算符中的语句转化为定义 cmp 函数函数体的相关语句即可,并在调用 sort 函数时加上第三个参数 cmp。

相信通过本节的学习与练习,读者不会再对排序问题感到恐惧。有了 sort 函数,你所需要做的只是按照题面要求为其定义不同的排序规则即可。

练习题: 特殊排序 (九度教程第 3 题); EXCEL 排序 (九度教程第 4 题); 字符串内排序 (九度教程第 5 题);

二 日期类问题

关于日期运算的各种问题,同样被频繁选入机试考题当中。但是这类问题都有规律可循。只要把握住这类问题的核心,解决这类问题就不会再有太大的难度。

例 2.3 日期差值 (九度教程第 6 题)

时间限制: 1 秒

内存限制: 32 兆

特殊判题: 否

题目描述:

有两个日期,求两个日期之间的天数,如果两个日期是连续的我们规定他们之间的天数为两天

输入:

有多组数据,每组数据有两行,分别表示两个日期,形式为 YYYYMMDD

输出:

每组数据输出一行,即日期差值

样例输入：

20110412

20110422

样例输出：

11

来源：

2009 年上海交通大学计算机研究生机试真题

该例题考察了日期类问题中最基本的问题——求两个日期之间的天数差，即求分别以两个特定日期为界的日期区间的长度。这里值得一提的是，解决这类区间问题有一个统一的思想——把原区间问题统一到起点确定的区间问题上去。在该例中，我们不妨把问题统一到特定日期与一个原点时间（如 0000 年 1 月 1 日）的天数差，当要求两个特定的日期之间的天数差时，我们只要将它们与原点日期的天数差相减，便能得到这两个特定日期之间的天数差（必要时加绝对值）。这样做有一个巨大的好处——预处理。我们可以在程序真正开始处理输入数据之前，预处理出所有日期与原点日期之间的天数差并保存起来。当数据真正开始输入时，我们只需要用 $O(1)$ 的时间复杂度将保存的数据读出，稍加处理便能得到答案。值得一提的是，预处理也是空间换时间的重要手段（保存预处理所得数据所需的内存来换取实时处理所需要的时间消耗）。

另外，日期类问题有一个特别需要注意的要点——闰年，每逢闰年 2 月将会有 29 天，这对我们计算天数势必会产生重大的影响。这里，我们必须明确闰年的判断规则——当年数不能被 100 整除时若其能被 4 整除则为闰年，或者其能被 400 整除时也是闰年。用逻辑语言表达出来即为 $\text{Year} \% 100 != 0 \ \&\& \ \text{Year} \% 4 == 0 \ || \ \text{Year} \% 400 == 0$ ，当逻辑表达式为 true 时，其为闰年；反之则不是闰年。从中我们也可以看出，闰年并不严格的按照四年一次的规律出现，在某种情况下也可能出现两个相邻闰年相隔八年的情况（如 1896 年与 1904 年）。所以，这里我们推荐严格按照上述表达式来判断某一年是否是闰年，而不采用某一个闰年后第四年又是闰年的规则。。

代码 2.6

```
#include <stdio.h>

#define ISYEAP(x) x % 100 != 0 && x % 4 == 0 || x % 400 == 0 ? 1 : 0

// 定义宏判断是否是闰年，方便计算每月天数

int dayOfMonth[13][2] = {
    0, 0,
    31, 31,
```

```

28, 29,
31, 31,
30, 30,
31, 31,
30, 30,
31, 31,
31, 31,
30, 30,
31, 31,
30, 30,
31, 31

}; //预存每月的天数, 注意二月配合宏定义作特殊处理

struct Date { //日期类, 方便日期的推移
    int Day;
    int Month;
    int Year;
    void nextDay() { //计算下一天的日期
        Day ++;
        if (Day > dayOfMonth[Month][ !ISYEAP(Year) ]) { //若日数超过了当月最大
日数
            Day = 1;
            Month ++; //进入下一月
            if (Month > 12) { //月数超过12
                Month = 1;
                Year ++; //进入下一年
            }
        }
    }
};

int buf[5001][13][32]; //保存预处理的天数

int Abs(int x) { //求绝对值
    return x < 0 ? -x : x;
}

```



```

int main () {
    Date tmp;
    int cnt = 0; //天数计数
    tmp.Day = 1;
    tmp.Month = 1;
    tmp.Year = 0; //初始化日期类对象为0年1月1日
    while(tmp.Year != 5001) { //日期不超过5000年
        buf[tmp.Year][tmp.Month][tmp.Day] = cnt; //将该日与0年1月1日的天数差保存起来
        tmp.nextDay(); //计算下一天日期
        cnt ++; //计数器累加，每经过一天计数器即+1，代表与原点日期的间隔又增加一天
    }
    int d1 , m1 , y1;
    int d2 , m2 , y2;
    while (scanf ("%4d%2d%2d", &y1, &m1, &d1) != EOF) {
        scanf ("%4d%2d%2d", &y2, &m2, &d2); //读入要计算的两个日期
        printf("%d\n", Abs(buf[y2][m2][d2] - buf[y1][m1][d1]) + 1); //用预处理的数据计算两日期差值，注意需对其求绝对值
    }
    return 0;
}

```

该代码展示了我们处理这类问题的基本方法。首先定义了一个类，不仅可以用来表示日期，还能够自动的计算出下一个日期。我们利用该类，一方面不断计算出当前日期的下一个日期，另一方面累加计数器，计算当前日期与原点日期的天数差，并将其保存在内存中，有了这些预处理出的数据，我们在计算两个特定日期差时，只要将两个日期与原点时间日期差相减，取绝对值，并根据题目需要加上1（“两个日期是连续的我们规定他们之间的天数为两天”），就能得出答案。

我们也可以考虑一下，假如问题需要我们输出某个特定的日期，是那年的第几天，我们该怎样利用我们已经得到的数据来计算呢？我们只需要用该日期与原点日期的天数减去那年元旦与原点日期便可得知该日期是当年的第几天。

在日期类中，为了判断该年是否是闰年，我们定义了一个宏。利用上文中提到的逻辑表达式来判断该年是否是闰年，并且根据该表达式的逻辑值使宏表达式为1或0，通过该宏表达式选择保存每月天数的数组的不同列来取得该月应该具有的天数，从而保证了当闰年时，2月29日存在。

另外，这段代码还有三个值得我们注意的地方。

其一，在保存某个特定日期与原点日期的天数差时，我们使用了三维数组，用年、月、日分别表示该数组下标，这便将日期本身与其存储地址联系了起来，这样我们在存取时不必再为查找其所在的存储地址而大费周章，而只需要直接利用它的年月日数字即可找到我们保存的值。将数据本身与数据存储地址联系起来，这是 Hash 的基本思想，也是 Hash 的一种基本方式。只不过这里的 Hash 不会产生冲突，我们并不需要为解决其冲突而另外下功夫。这种思想会在后文再着重讲解。

其二，该例程的输入采用了某种技巧。因为题面规定用一个连续的八位数来代替日期，我们使用 %4d 来读取该八位数的前四位并赋值给代表年的变量，同理使用 %2d%2d 来读取其它后四位并两两赋值给月日。这种利用在 %d 之间插入数字来读取特定位数的数字的技巧，也值得大家去利用，在本例中，它为处理输入带来了极大的便利。

其三，我们将 `buf[5001][13][32]` 这个相对比较耗费内存的数组定义成全局变量，这不是偶然的。由于需要耗费大量的内存，若在 `main` 函数（其它函数也一样）之中定义该数组，其函数所可以使用的栈空间将不足以提供如此庞大的内存，出现栈溢出，导致程序异常终止。所以，今后凡是涉及此类需要开辟大量内存空间的情况，我们都必须在函数体外定义，即定义为全局变量。或者在函数中使用 `malloc` 等函数动态申请变量空间。读者必须牢记这一点。

可能部分读者已经发现，此例本身也带有一些缺陷，它并没有明确的告知我们输入数据的数据范围。我们可以大致认为其年是从 0 年到 5000 年（事实上也在这个范围内），那么我们才可以估算该题的复杂度。关于这题的复杂度计算，我就不再赘述了，读者可以自己动手，计算其是否符合要求。

在接触了日期类最基本问题以后，其他问题也可以在其基础上得到答案。

例 2.4 Day of week（九度教程第 7 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

We now use the Gregorian style of dating in Russia. The leap years are years with number divisible by 4 but not divisible by 100, or divisible by 400. For example, years 2004, 2180 and 2400 are leap. Years 2004, 2181 and 2300 are not leap.

Your task is to write a program which will compute the day of week corresponding to a given date in the nearest past or in the future using today's

agreement about dating.

输入:

There is one single line contains the day number d , month name M and year number y ($1000 \leq y \leq 3000$). The month name is the corresponding English name starting from the capital letter.

输出:

Output a single line with the English name of the day of week corresponding to the date, starting from the capital letter. All other letters must be in lower case.

样例输入:

9 October 2001

14 October 2001

样例输出:

Tuesday

Sunday

来源:

2008 年上海交通大学计算机研究生机试真题

该题是一例以英文命题的机试真题。其大意为，输入一个日期，要求输出该日期为星期几。我们照样可以利用上例的思路来解答该题。星期几是以七为周期循环的，那么我们只需要知道：1.今天是星期几；2.今天和所给定的那天相隔几天。利用其对7求余数，我们便可以轻易的知道所给定的那天是星期几了。

代码 2.7

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

#define ISYEAP(x) x % 100 != 0 && x % 4 == 0 || x % 400 == 0 ? 1 : 0

int dayOfMonth[13][2] = {
    0, 0,
    31, 31,
    28, 29,
    31, 31,
    30, 30,
    31, 31,
    30, 30,
    31, 31,
    31, 31,
    30, 30,
```

```

31, 31,
30, 30,
31, 31
};

struct Date {
    int Day;
    int Month;
    int Year;
    void nextDay() {
        Day ++;
        if (Day > dayOfMonth[Month][ !SYEAP(Year) ]) {
            Day = 1;
            Month ++;
            if (Month > 12) {
                Month = 1;
                Year ++;
            }
        }
    }
};

int buf[3001][13][32];
char monthName[13][20] = {
    "",
    "January",
    "February",
    "March",
    "April",
    "May",
    "June",
    "July",
    "August",
    "September",
    "October",
    "November",
    "December"
};

```



```

    "December"
}; //月名 每个月名对应下标1到12
char weekName[7][20] = {
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
}; //周名 每个周名对应下标0到6

int main () {
    Date tmp;
    int cnt = 0;
    tmp.Day = 1;
    tmp.Month = 1;
    tmp.Year = 0;
    while(tmp.Year != 3001) {
        buf[tmp.Year][tmp.Month][tmp.Day] = cnt;
        tmp.nextDay();
        cnt ++;
    } //以上与上题一致, 预处理出每一天与原点日期的天数差
    int d, m, y;
    char s[20];
    while (scanf ("%d%s%d", &d, s, &y) != EOF) {
        for (m = 1; m <= 12; m++) {
            if (strcmp(s, monthName[m]) == 0) {
                break; //将输入字符串与月名比较得出月数
            }
        }
        int days = buf[y][m][d] - buf[2012][7][16]; //计算给定日期与今日日期的
        //天数间隔(注意可能为负)
        days += 1; //今天(2012. 7. 16)为星期一, 对应数组下标为1, 则计算1经过days天后
        //的下标
    }
}

```

```
puts(weekName[(days % 7 + 7) % 7]); //将计算后得出的下标用7对其取模,并且保证其为非负数,则该下标即为答案所对应的下标,输出即可  
}  
return 0;  
}
```

在该例中,预处理部分与上例保持一致。依旧是处理出每个日期与原点日期之间的天数间隔(虽然本例依然选用0年1月1日为原点日期,但实际上原点日期可以按照要求任意选取)。然后计算该日期与当前日期的天数间隔。若间隔为正,则表示目标日期在当前日期之后;若为负,则表示目标日期在当前日期之前。同时我们将周日到周一分别对应数组下标0到6(与保存周名数组保持一致),本例中,当前日期为星期一,固对应数组下标为1,我们在其上加上刚刚算得的日期间隔,如200天,那么我们取的目标日期的下标 $1+200=201$,又由于星期是以7为周期循环的,我们用7对其去模(即求余数),来计算其真正的下标(若为负数,则还要保证该下标为正,我们可以简单的处理成余数加7再求模), $201 \% 7 = 5$,则表示经过若干个循环后,目标天的周名下标为5。将该下标对应的星期名输出,即为我们需要的答案,weekName[5],“Friday”即为所求。

若读者对求模(%)符号,及其相关运算有困惑,可参考第三章中相关章节,那里将会对其有更加详细的讨论。

该例中的解法旨在展示一种处理日期类问题的基本思路和一般方法,即利用天数间隔来解决日期类问题,同时也提出了处理区间问题的一种重要方法——预处理。这里顺便可以提一句,计算星期几问题存在着公式解法:蔡勒(Zeller)公式。但是该公式比较复杂,完全记忆需要一定的时间,有兴趣读者可以查阅相关资料。

练习题: 今年的第几天? (九度教程第8题); 九度 OJ1186 打印日期 (九度教程第9题);

三 Hash的应用

相信各位读者对上一节中,将一个日期对应的预处理数据存储在一个以该日期的年月日为下标的三维数组中还有印象。这种将存储位置与数据本身对应起来的存储手段就是 Hash。本节将对其做更加详细的讨论。与以往不同的是,本节所讨论的 Hash 旨在讲述其在机试试题解答中的作用,而不像《数据结构》教科书上,对各种 Hash 方法、冲突处理做过多的阐述。

例 2.5 统计同成绩学生人数（九度教程第 10 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

读入 N 名学生的成绩，将获得某一给定分数的学生人数输出。

输入：

测试输入包含若干测试用例，每个测试用例的格式为

第 1 行：N

第 2 行：N 名学生的成绩，相邻两数字用一个空格间隔。

第 3 行：给定分数

当读到 N=0 时输入结束。其中 N 不超过 1000，成绩分数为（包含）0 到 100 之间的一个整数。

输出：

对每个测试用例，将获得给定分数的学生人数输出。

样例输入：

```
3
80 60 90
60
2
85 66
0
5
60 75 90 55 75
75
0
```

样例输出：

```
1
0
2
```

来源：

2006 年浙江大学计算机及软件工程研究生机试真题

初见此题，读者可能可以一下子想到非常多的方法来解决此题。比如将输入

的分数首先保存在数组里,当需要查询时可将要查询的目标分数依次与这些分数比对,从而得出答案。但是我们这里还是指出使用 Hash 解决此类问题不失为一种好的办法。

在开始具体解题之前,我们要注意这类问题存在的一个共同特点:输入的分数种类是有限的。在此例中,我们可以看到,输入的分数不仅严格在 0 到 100 的区间之内,同时它又一定是一个整数。这样,输入的数据一共只有 101 种可能,我们只要为这 101 种可能分别计数,便能在输入结束时便得知每一种分数的重复情况。

代码 2.8

```
#include <stdio.h>

int main () {
    int n;

    while (scanf ("%d",&n) != EOF && n != 0) { //输入判断增加对n是否等于零进行判断

        int Hash[101] = {0}; //建立一个初始为0的Hash数组用来记录各种分数出现的次数

        for (int i = 1;i <= n;i ++){

            int x;

            scanf ("%d",&x);

            Hash[x] ++; //统计分数出现次数

        }

        int x;

        scanf ("%d",&x);

        printf("%d\n",Hash[x]); //得到需要查询的目标分数后,只需简单的查询我们统计的数量即可

    }

    return 0;
}
```

该解法利用了输入只有 0 到 100 这 101 种可能的特点。利用其与数组下标对应的方法分别统计各分数出现的次数。初始时,我们将数组初始化为 0,代表着每一个分数出现的次数都是 0。当开始输入分数时,我们依据输入的分数 x ,累加代表其出现次数的数组元素 $\text{Hash}[x]$,从而统计其重复次数。这里,我们利用读入的分数直接作为数组下标来访问该元素,因此这个过程十分快捷。当输入完成后,Hash 数组中就已经保存了每一个分数出现的次数。当我需要查询分数 x 出现的次数时,只需访问统计其出现次数的数组元素 $\text{Hash}[x]$,便能得知答案。

从代码 2.8 不难看出，该解法写法简单、思路清晰。所以，在必要时使用 Hash 算法必能事半功倍。

经过上例的分析，读者对机试中 Hash 的原理及应用应有了初步的认识，那么还有哪些类型的题是可以利用该技巧的呢？

例 2.6 Sort （九度教程第 11 题）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

给你 n 个整数，请按从大到小的顺序输出其中前 m 大的数。

输入：

每组测试数据有两行，第一行有两个数 $n, m (0 < n, m < 1000000)$ ，第二行包含 n 个各不相同，且都处于区间 $[-500000, 500000]$ 的整数。

输出：

对每组测试数据按从大到小的顺序输出前 m 大的数。

样例输入：

5 3

3 -35 92 213 -644

样例输出：

213 92 3

我们容易联想到，要输出前 m 大的数，我们只需将其降序排序，然后输出前 m 个数即可。那么，读者可能不禁会产生疑问，这不就是排序么，而这个问题我们在本章第一节中不是已经讨论过了，为什么这里又再次提出来了。如果读者有这个疑问，你就要反问自己了：你是否忽略了什么？我们在前几例中并没有着重分析算法的复杂度，但这不代表复杂度分析在我们解题的过程中不那么重要，相反复杂度分析是一个算法可行的前提与保证。即使本文没有明确的分析复杂度，读者也要有自己估算复杂度的意识。在本例中，如果使用排序来解决该题，由于待排序数字的数量十分庞大（1000000），即使使用时间复杂度为 $O(n \log n)$ 的快速排序，其时间复杂度也会达到千万数量级，而这在一秒时限内是不能被我们所接受的，所以这里，我们并不能使用快速排序来解决本题。

有了上例的启发，读者应该很快就能注意到，本例与上例有一个共同的特点：输入数量的有限性。该例题面限定了输入的数字一定是 $[-500000, 500000]$ 区间里的整数，且各不相同。若利用一个数组分别统计每一种数字是否出现，其空间复杂度依旧在题目的限定范围内。且统计出现数字当中较大的 m 个数字，也仅需要从尾至头遍历这个数组，其时间复杂度仍在百万数量级，所以该解法是符合我

们要求的。

代码 2.9

```
#include <stdio.h>

#define OFFSET 500000 //偏移量, 用于补偿实际数字与数组下标之间偏移

int Hash[1000001]; //Hash数组, 记录每个数字是否出现, 不出现为0, 出现后被标记成1

int main () {
    int n , m;

    while (scanf ("%d%d", &n, &m) != EOF) {
        for (int i = -500000; i <= 500000; i ++ ) {
            Hash[i + OFFSET] = 0;
        } //初始化, 将每个数字都标记为未出现

        for (int i = 1; i <= n; i ++ ) {
            int x;

            scanf ("%d", &x);

            Hash[x + OFFSET] = 1; //凡是出现过的数字, 该数组元素均被设置成1
        }

        for (int i = 500000; i >= -500000; i -- ) { //输出前m个数
            if (Hash[i + OFFSET] == 1) { //若该数字在输入中出现
                printf("%d", i); //输出该数字

                m --; //输出一个数字后, m减一, 直至m变为0

                if (m != 0) printf(" "); //注意格式, 若m个数未被输出完毕, 在输出的
                数字后紧跟一个空格

            else {
                printf("\n"); //若m个数字已经被输出完毕, 则在输出的数字后面紧跟
                一个换行, 并跳出遍历循环

                break;
            }
        }

        }

    }

    return 0;
}
```

该代码介绍了一种在输入数据有如上所述的特点时, 一种在时间上更加高效

的排序方法。

这里，由于输入数据中出现了负数，于是我们不能直接把输入数据当做数组下标来访问数组元素，而是将每一个输入的数据都加上一个固定的偏移值，使输入数据的 $[-500000, 500000]$ 区间被映射到数组下标的 $[0, 1000000]$ 区间。所以无论我们在做统计还是读出统计值时，我们都需要注意这个偏移值，只有加上了这个偏移值，我们才能访问到正确的数组元素。

我们利用这个统计数组，将所有出现过的数字对应的数组元素都标记为 1，而没有出现过的数字对应的数组元素都保持为 0，当我们需要输出前 m 大个数字时，我们只需要从 500000 开始，降序遍历这个数组，查找前 m 个被标记成 1 的数组元素输出其对应的数字即可。

本例中还有一个值得我们关注的地方是它的输出，虽然题目中并没有明确指出，但通过观察输出样例我们一样可以得知（通过观察输出样例来明确输出格式也是一种好的习惯），在输出的每一个数字之间存在着一个空格，而在最后一个数字之后却不存在空格，所以我们在输出数字时应当特别注意控制空格的输出，忘记输出或者输出一个多余的空格都会造成评判系统对于该代码输出格式错误的评判。

本例对输入数字做了“各不相同”的限定，试问假如去除这一限定，使输入数字可能存在重复，该 Hash 方法是否依旧可用？若可用，请读者自己动手尝试编写相应程序。

练习题：谁是你的潜在朋友（九度教程第 12 题）；九度 OJ1088 剩下的树（九度教程第 13 题）；

四 排版题

本节将要讨论另外一种常见的题型，它不再把注意力放到处理输入的数据当中，而对输出作特别的关注。这类题型就是排版题，特别考察考生对于输出格式的把握。

例 2.7 输出梯形（九度教程第 14 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

输入一个高度 h ，输出一个高为 h ，上底边为 h 的梯形。

输入：

一个整数 $h(1 \leq h \leq 1000)$ 。

输出：

h 所对应的梯形。

样例输入：

4

样例输出：

```
****
*****
*****
*****
```

来源：

2001 年清华大学计算机研究生机试真题(第 II 套)

此例便是典型的考察考生对输出格式把握的机试真题。我们试着观察输出图形，容易发现其具有较强的规律性：1.首行“*”的个数和梯形高度都是 h；2.下一行总是比上一行多两个“*”；3.每行都是右对齐，左边空余的位置用空格代替。有了这些规律，我就能得知如下程序所需要的关键信息：1.需要 h 次循环来输出每一行。2.第一行包含 h 个“*”，第二行包含 h + 2 个“*”，依次类推，最后一行包含 h + (h - 1) * 2 个“*”；3.在每行输出“*”之前，都需要输出空格来使输出的“*”右对齐，每行输出的空格数是该行所要输出的“*”个数与最后一行所具有的“*”数目之差。这样，我们得出了该输出图形的所有有用的规律，并可以把规律应用到程序的输出过程当中。

代码 2.10

```
#include <stdio.h>

int main () {
    int h;
    while (scanf ("%d", &h) != EOF) {
        int maxLine = h + (h - 1) * 2; //计算最后一行包含的星号个数
        for (int i = 1; i <= h; i++) { //依次输出每行信息
            for (int j = 1; j <= maxLine; j++) { //依次输出每行当中的空格或星号
                if (j < maxLine - h - (i - 1) * 2 + 1) //输出空格
                    printf(" ");
                else //输出星号
                    printf("*");
            }
            printf("\n"); //输出换行
        }
    }
}
```

```
}  
return 0;  
}
```

该排版题有一个显著的特点：首先图形具有较强的规律性，且该规律顺序往往与输出顺序一致，即可以从上至下、从左至右应用规律。于是，我们只需仔细观察图形，把握其中所具有的规律，并将其量化后直接写入程序的输出部分，就可以输出题面所要求的图形。

但是有另一个类排版题，它所要求的图形不具有显著的规律性或者规律性较难直接应用到输出当中。为了解决此类问题，我们需要了解排版题常用的另一种方法：先完成排版，再进行输出。

例 2.8 叠筐（九度教程第 15 题）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

把一个个大小差一圈的筐叠上去，使得从上往下看时，边筐花色交错。这个工作现在要让计算机来完成，得看你的了。

输入：

输入是一个个的三元组，分别是，外筐尺寸 n (n 为满足 $0 < n < 80$ 的奇整数)，中心花色字符，外筐花色字符，后二者都为 ASCII 可见字符；

输出：

输出叠在一起的筐图案，中心花色与外筐花色字符从内层起交错相叠，多筐相叠时，最外筐的角总是被打磨掉。叠筐与叠筐之间应有一行间隔。

样例输入：

11 B A

5 @ W

样例输出：

```
AAAAAAAAAA  
ABBBBBBBBBBA  
ABAAAAAAABA  
ABABBBBBBABA  
ABABAAABABA  
ABABABABABA  
ABABAAABABA  
ABABBBBBBABA  
ABAAAAAAABA
```

```
ABBBBBBBBBBA
AAAAAAAAAA
```

```
   @@@
  @WWW@
 @W@W@
@WWW@
   @@@
```

如此例所示，其输出图形的规律性主要体现在由内而外的各个环上，而这与我们的输出顺序又不太契合（从上至下，从左至右），于是我们不容易将该图形存在的规律直接应用到输出当中，所以我们需要使用刚才所提到的办法——先排版后输出。并在排版（而不是输出）时利用我们观察到的“环形规律”完成排版。我们先来看如下代码：

代码 2.11

```
#include <stdio.h>

int main () {
    int outPutBuf[82][82]; //用于预排版的输出缓存
    char a , b; //输入的两个字符
    int n; //叠框大小
    bool firstCase = true; //是否为第一组数据标志，初始值为true
    while (scanf ("%d %c %c", &n, &a, &b) == 3) {
        if (firstCase == true) { //若是第一组数据
            firstCase = false; //将第一组数据标志标记成false
        }
        else printf("\n"); //否则输出换行
        for (int i = 1, j = 1; i <= n; i += 2, j++) { //从里至外输出每个圈
            int x = n / 2 + 1 , y = x;
            x -= j - 1; y -= j - 1; //计算每个圈右上角点的坐标
            char c = j % 2 == 1 ? a : b; //计算当前圈需要使用哪个字符
            for (int k = 1; k <= i; k++) { //对当前圈进行赋值
                outPutBuf[x + k - 1][y] = c; //左边赋值
                outPutBuf[x][y + k - 1] = c; //上边赋值
                outPutBuf[x + i - 1][y + k - 1] = c; //右边赋值
                outPutBuf[x + k - 1][y + i - 1] = c; //下边赋值
            }
        }
    }
}
```

```

    }
}

if (n != 1) { //注意当n为1时不需此步骤
    outPutBuf[1][1] = ' ';
    outPutBuf[n][1] = ' ';
    outPutBuf[1][n] = ' ';
    outPutBuf[n][n] = ' '; //将四角置为空格
}

for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        printf("%c", outPutBuf[i][j]);
    }
    printf("\n");
} //输出已经经过排版的在输出缓存中的数据
}

return 0;
}

```

如该代码所示，我们不再在输出时使用我们得到的规律，而是用另一种更容易的方法完成排版。我们利用一个缓存数组来表示将要输出的字符阵列，我们对该字符阵列的坐标作如下规定，规定阵列左上角字符坐标为 $(1, 1)$ ，阵列右下角字符坐标为 (n, n) ，其它坐标可由此推得。程序按照由最内圈至最外圈的顺序来完成图形的排列。在完成每圈排列时，我们都需要注意两个要点：首先需要确定该圈左上角的坐标。我们将以这个坐标为参照点来完成该圈的其它字符位置的确定（当然也可以选用其它点）。观察图形得知，最中间圈的左上角字符坐标为 $(n/2 + 1, n/2 + 1)$ ，次中间圈的左上角字符坐标为 $(n/2 + 1 - 1, n/2 + 1 - 1)$ ，依次类推即可得到图形中每一个圈的参照点。其次，我们需要计算该圈每边边长长度。这也较容易得出，中心圈长度为 1，次中心圈长度为 3，依次类推，外圈总比内圈长度增加 2。注意了这两点以后，我们按照以下顺序完成每圈的排版，首先明确该圈使用哪一个字符来填充，我们使用判断循环次数指示变量 j 的奇偶性来判断当前需要使用的字符，即奇数次循环时（ j 为奇数）时使用第一个字符，偶数次循环时使用第二个字符。然后，我们确定该圈左上角字符的坐标，我们使用中心坐标 $(n/2 + 1, n/2 + 1)$ 减去当前循环次数指示变量 j 来确定该圈左上角坐标，即 $(n/2 + 1 - j, n/2 + 1 - j)$ 。接着，我们计算该圈边长长度，我们利用初始值为 1 的循环指示变量 i 来表示边长长度，并在每次循环结束后加 2，代表

边长由 1 开始，每外移一个圈边长长度即加上 2。利用变量 `i` 所存的值我们即可对当前圈的四条边进行赋值，对应的坐标已在代码中给出，这里不再列举。在完成所有圈的编排后，我们只需按照题目的需要去除四个角的字符，最后将整个输出缓存中的字符阵列输出即可。

另外，此代码还有两个注意点值得我们指出。

1.输出格式。题面要求我们在输出的每个叠筐间输出一个空行，即除了最后一个叠筐后没有多余的空行，其它叠筐输出完成后都需要额外的输出一个空行。为了完成这个要求，我们将要求形式改变为除了在第一个输出的叠筐前不输出一个空行外，在其它每一个输出的叠筐前都需要输出一个额外的空行。为完成这一目的，我们在程序开头设立了 `firstCase` 变量来表示正在处理数据的是否为第一组数据，毫无疑问它的初始值为 `true`。在程序读取每组数据后，我们都测试 `firstCase` 的值，若其为 `true` 则表示当前处理的数据为第一组数据，我们不输出空行，并在此时将 `firstCase` 变量改变为 `false`。以后，每当程序读入数据，测试 `firstCase` 变量时，该变量均为 `false`，于是我们完成题目的要求，在输出的叠筐前额外的输出一个空行，来达到题面对于输出格式的要求。

2.边界数据处理。按上文所说，我们在输出缓存中完成字符阵列排版后，需要将该阵列四个角的字符修改为空格，但是这一修改不是一定需要的。当输入的 `n` 为 1 时，该修改会变得多余，它会使输出仅变为一个空格，这与题面要求不符。因此，在进行该修改之前，我们需要对 `n` 的数值作出判断，若其不为 1 则进行修改，否则跳过修改部分。由此不难看出，机试考题要求我们在作答时，不仅能够大致的把握算法，同时还要细致的考虑边界数据会给我们的程序造成什么样的影响。只有充分考虑了所有情况，并保证在所有题面明确将会出现的条件下，程序依旧能够正常工作，这样我们才能使自己的程序真正的万无一失、滴水不漏。本例介绍了另一种解决排版题的思路，当输出图形所具有的规律不能或者很难直接应用到输出上时，我们就要考虑采用该例所采用的方法，先用一个二维数组来保存将要输出的字符阵列，并在该数组上首先完成排版。因为没有了输出时从上至下、从左至右的顺序限制，我们能更加随意的按照自己的需要或者图形的规律来依次输出图形，从而完成题目要求。

练习题： Repeater（难度较大）（九度教程第 16 题）；

五 查找

查找是另外一类我们必须掌握的算法，它不仅会被机试直接考察，同时也可能是完成其他某些算法的重要环节。对于查找问题，有难有易。可能只是直接的

对某个数字的查找，也可能涉及搜索等相对难度更大的算法。本节所涉及的查找问题，都是查找的基础。只有先充分掌握查找的概念和方法，我们才能继续学习其他难度更大的算法。

例 2.9 找 x（九度教程第 17 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

输入一个数 n ，然后输入 n 个数值各不相同，再输入一个值 x ，输出这个值在这个数组中的下标（从 0 开始，若不在数组中则输出 -1）。

输入：

测试数据有多组，输入 $n(1 \leq n \leq 200)$ ，接着输入 n 个数，然后输入 x 。

输出：

对于每组输入,请输出结果。

样例输入：

2

1 3

0

样例输出：

-1

来源：

2010 年哈尔滨工业大学计算机研究生机试真题

由此例我们可以看出，即使是在数组中查找特定数字这样最最基本的问题，也会出现在考研机试真题当中，查找在整个算法体系中的重要性可见一斑。通过此例，我们可以了解一下查找所涉及的几个基本要素。

1. 查找空间。也常被称为解空间。所谓查找，就是在该查找空间中找寻符合我们要求的解的过程。在此例中，整个数组包含的整数集就是查找空间。

2. 查找目标。我们需要一个目标来判断查找空间中的各个元素是否符合我们的要求，以便判断查找活动是否已经成功。在此例中，即数组中的数字与目标数字是否相同。

3. 查找方法。即利用某种特定的策略在查找空间中查找各个元素。不同的策略对查找的效率和结果有不同的影响，所以对于某个特定的问题，我们要选择切实可行的策略来查找解空间，以期事半功倍。在此题中，查找方法即线性地遍历数组。

读者可以牢记这些概念，这对在后续章节中充分的理解搜索的原理是大有裨

益的。关于更高级的查找方式——搜索，将在后续章节详细讨论，这里只需对查找有初步的概念即可。

这里提出此例，即为了说明查找的相关概念。这里对解题不加过多的说明，仅提供解题代码供读者参考。

代码 2.12

```
#include <stdio.h>

int main () {
    int buf[200];
    int n;
    while (scanf ("%d", &n) != EOF) {
        for (int i = 0; i < n; i++) {
            scanf ("%d", &buf[i]);
        } //输入数据
        int x, ans = -1; //初始化答案为-1，以期在找不到答案时能正确的输出-1
        scanf ("%d", &x);
        for (int i = 0; i < n; i++) { //依次遍历数组元素
            if (x == buf[i]) { //目标数字与数组元素依次比较
                ans = i;
                break; //找到答案后跳出循环
            }
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

下面我们来接触一种新的查找策略，它不同于上例所采用的线性查找方法，而以一种带有策略性的、跳跃的方法来遍历查找空间，它就是二分查找。

二分查找建立在待查找元素排列有序的前提上，例如在一个升序有序的数组中查询某元素。我们以在有序表{1,3,4,5,6,8,10}查找3为例，了解它的查找过程：

- 1.将查找开始点设为第一个数组元素(1)，结束点设为最后一个数组元素(10)，即查找子集为整个搜索空间{1,3,4,5,6,8,10}。

- 2.然后将起始点和结束点正中间的数与查找目标进行比较，若该中间数字等于目标数字则查找成功，查找结束；若大于查找目标，则说明查找目标只可能存在于查找子集中以该中间数字为界的较小的一半中，则移动查找结束点为该中间

数字的前一个数字,即新的查找子集为旧的查找子集中以中间数字为界的较小的一半;若小于查找目标,则相应的得到新的查找子集为旧查找子集中以中间数字为界的较大的一半。在该例中,即目标数字 3 小于中间数字 5,移动查找结束点至中间点 (5) 的前一个元素 (4),新的查找子集为 {1,3,4},然后继续步骤 2。

3.若在查找过程中出现查找起始点大于查找结束点的情况,则说明查找子集已经为空集,查找失败。否则继续步骤 2 的查找过程。

该例的查找过程如下:

查找子集	查找起始点	查找终止点	中间点
{1,3,4,5,6,8,10}	1(下标 0)	10 (下标 6)	5 (下标 3)
↓ 目标数 3 小于 中间数 5			
{1,3,4,}	1(下标 0)	4 (下标 2)	3 (下标 3)
↓ 目标数 3 等于中间数 3			
查找成功。			

若我们查找一个待查找数组中不存在的数字 2,情况又会怎样呢?

查找子集	查找起始点	查找终止点	中间点
{1,3,4,5,6,8,10}	1(下标 0)	10 (下标 6)	5 (下标 3)
↓ 目标数 2 小于 中间数 5			
{1,3,4,}	1(下标 0)	4 (下标 2)	3 (下标 3)
↓ 目标数 2 小于中间数 3			
{1}	1(下标 0)	1 (下标 0)	1 (下标 0)
↓ 目标数 2 大于中间数 1			
{}	3(下标 1)	1 (下标 0)	无
↓ 查找子集变为空集			
查找失败。			

可见,若我们在数组中二分查找一个不存在的数字,其查找的最后结果为查找子集变为空集。用二分查找查找长度为 L 的有序数组,时间复杂度可由原本线性查找的 $O(L)$ 降低到 $O(\log L)$ 。

例 2.10 查找学生信息 (九度教程第 18 题)

时间限制: 1 秒 内存限制: 32 兆 特殊判题: 否

题目描述:

输入 N 个学生的信息,然后进行查询。
输入:

输入的第一行为 N ，即学生的个数($N \leq 1000$)

接下来的 N 行包括 N 个学生的信息，信息格式如下：

01 李江 男 21

02 刘唐 男 23

03 张军 男 19

04 王娜 女 19

然后输入一个 M ($M \leq 10000$), 接下来会有 M 行，代表 M 次查询，每行输入一个学号，格式如下：

02

03

01

04

输出：

输出 M 行，每行包括一个对应于查询的学生的信息。

如果没有对应的学生信息，则输出“No Answer!”

样例输入：

4

01 李江 男 21

02 刘唐 男 23

03 张军 男 19

04 王娜 女 19

5

02

03

01

04

03

样例输出：

02 刘唐 男 23

03 张军 男 19

01 李江 男 21

04 王娜 女 19

03 张军 男 19

来源：

2003 年清华大学计算机研究生机试真题

初见该例，读者可能会有该例与上一例没有特别的差别的感觉，我们要做的依然是在一个数组中查找特定字段相同的工作。那么，为什么在这里提出该题？要注意的依然是时间复杂度，读者必须对此非常的敏感。即无论提出什么算法，必须首先估计复杂度是否符合要求，若忽略了这一点往往会造成南辕北辙的悲剧。

若我们依旧采用每次询问时线性遍历数组来查找是否存在我们需要查找的元素，那么，该算法的时间复杂度达到了 $O(n * m)$ （查找次数 * 每次查找所需比较的个数），而这已经达到了千万数量级，是我们所不愿看到的。于是，我们只有另找方法来解决该题。

没错，就是二分查找。为了符合查找空间单调有序的要求，我们首先要对所有数组元素按照学号关键字升序排列。当数组内各元素已经升序有序时，我们就可以在每次询问某个特定学号的学生是否存在时，使用二分查找来查找该学生。

代码 2.13

```
#include <stdio.h>
#include <algorithm>
#include <string.h>
using namespace std;
struct Student{ //用于表示学生个体的结构体
    char no[100]; //学号
    char name[100]; //姓名
    int age; //年龄
    char sex[5]; //性别
    bool operator < (const Student & A) const { //重载小于运算符使其能使用sort
函数排序
        return strcmp(no, A.no) < 0;
    }
}buf[1000];

int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        for (int i = 0; i < n; i++) {
            scanf ("%s%s%d",
```

```

buf[i].no, buf[i].name, buf[i].sex, &buf[i].age);
} //输入

sort(buf, buf + n); //对数组排序使其按照学号升序排列

int t;

scanf ("%d", &t); //有t组询问

while (t -- != 0) { //while循环保证查询次数为t

    int ans = -1; //目标元素下标, 初始化为-1

    char x[30];

    scanf ("%s", x); //待查找学号

    int top = n - 1, base = 0; //初试时, 开始下标0, 结束下标n-1, 查找子集为
整个数组

    while(top >= base) { //当查找子集不为空集时重复二分查找

        int mid = (top + base) / 2; //计算中间点下标

        int tmp = strcmp(buf[mid].no, x); //比较中间点学号与目标学号

        if (tmp == 0) {

            ans = mid;

            break; //若相等, 则查找完成跳出二分查找

        }

        else if (tmp > 0) top = mid - 1; // 若大于, 则结束下标变为中间点
前一个点下标

        else base = mid + 1; //若小于, 则开始点下标变为中间点后一个点坐标

    }

    if (ans == -1) { //若查找失败

        printf("No Answer!\n");

    }

    else printf("%s %s %s %d\n",
buf[ans].no, buf[ans].name, buf[ans].sex, buf[ans].age); //若查找成功

    }

}

return 0;

}

```

利用二分查找, 原本 $O(n * m)$ 的时间复杂度被优化到 $O(n \log n (\text{排序}) + m * \log n)$, 而该复杂度是符合我们要求的。

结合代码和上文的讲解, 相信读者可以很快掌握二分查找的特点, 并可以独

立写出相关代码。

在查找某特定元素是否存在以外，二分查找还有另一类非常重要的运用，即定界。思考如下问题，在一个升序有序的数组中，确定一个下标点，使在这个下标点之前（包括该下标点）的数字均小于等于目标数字（该目标数字一定大于等于数组中最小的数字），而数组的其余部分均大于目标数字，我们该如何编写程序。要完成这个问题的解答，我们首先要对二分查找原理有一定的了解，读者可以自行考虑如何运用二分查找确定该边界点，这里仅给出代码以供参考。

```
//存在一个升序有序的数组buf, 其大小为size, 目标数字为target
int base = 0, top = size; //初始情况与二分查找一致
while (base <= top) { //二分循环条件与二分查找一致
    int mid = (base + top) / 2;
    if (buf[mid] <= target) base = mid + 1; //符合前一部分数字规定
    else top = mid - 1; //否则
}
int ans = top; //最后, top即为我们要求的数字数组下标, buf[top]为该数字本身
```

本节介绍了查找的基本概念，并提出了两种简单的查找方法。其中二分查找需要读者重点把握，其在机试中出现的概率较大。

练习题：打印极值点下标（九度教程第 19 题）；查找（九度教程第 20 题）；用二分查找重写九度教程第 20 题。

六 贪心算法

在本章的最后，我们来介绍一种思路相对简单的算法——贪心。说它是算法，倒不如说他是一种思想，一种总是选择“当前最好的选择”而不从整体上去把握的思想。但往往这种“贪心”的策略能得到接近最优的结果，甚至在某些情况下，这样就能得到最优解。虽然该算法在九度为我们准备的真题集里，几乎没有出现，但它确实是一个我们值得掌握的算法思想。另外，在我本人的机试过程中（浙大 2012）就碰到了一道利用贪心思想来解决的问题，此题也是当场考试难度最大的考题，可见贪心的重要性。

我们用一个简单的例子来引出该算法。

例 2.11 FatMouse' Trade（九度教程第 21 题）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述:

FatMouse prepared M pounds of cat food, ready to trade with the cats guarding the warehouse containing his favorite food, JavaBean.

The warehouse has N rooms. The i -th room contains $J[i]$ pounds of JavaBeans and requires $F[i]$ pounds of cat food. FatMouse does not have to trade for all the JavaBeans in the room, instead, he may get $J[i] * a\%$ pounds of JavaBeans if he pays $F[i] * a\%$ pounds of cat food. Here a is a real number. Now he is assigning this homework to you: tell him the maximum amount of JavaBeans he can obtain.

输入:

The input consists of multiple test cases. Each test case begins with a line containing two non-negative integers M and N . Then N lines follow, each contains two non-negative integers $J[i]$ and $F[i]$ respectively. The last test case is followed by two -1's. All integers are not greater than 1000.

输出:

For each test case, print in a single line a real number accurate up to 3 decimal places, which is the maximum amount of JavaBeans that FatMouse can obtain.

样例输入:

```
5 3
7 2
4 3
5 2
20 3
25 18
24 15
15 10
-1 -1
```

样例输出:

```
13.333
31.500
```

题目大意如下: 有 m 元钱, n 种物品; 每种物品有 j 磅, 总价值 f 元, 可以使用 0 到 f 的任意价格购买相应磅的物品, 例如使用 $0.3f$ 元, 可以购买 $0.3j$ 磅物品。要求输出用 m 元钱最多能买到多少磅物品。

可能精于计算的读者会马上反应过来, 每次都买剩余物品中性价比 (即重量价格比) 最高的物品, 直到该物品被买完或者钱耗尽。若该物品已经被买完, 则我们继续在剩余的物品中寻找性价比最高的物品, 重复该过程; 若金钱耗尽, 则

交易结束。

如上所说，这就是一种典型的贪心思想，我们每次都尽可能的多买性价比高的物品直到该物品被买完或者金钱耗尽。那么，这样朴素的贪心策略是否能真正获得最优解呢？我们需要自己动手证明一下：

按照我们的策略，最后所买的物品一定符合如下条件：在最优解中，如果存在性价比为 a 的物品，那么原物品中性价比高于 a 的物品一定全部被我们买下来。

要证明该命题，我们不妨假设该解并不是最优解。即存在，性价比为 b ($b > a$) 的物品，仍然剩余一部分没有被我们买入，但是我们却获得了最优解。那么在该解的基础上，我们退掉部分性价比为 a 物品（哪怕一点点），而将这些钱都用来买性价比为 b 的物品。读者可以想象一下这时会发生什么。对，因为性价比 $b > a$ ，我们用同样的金钱所买的性价比为 b 的物品的重量一定大于性价比为 a 的物品。那么，完成该一卖一买后，总重量有所增加。原解一定不是最优解，我们的假设不成立。命题“最优解中，如果存在性价比为 a 的物品，那么原物品中性价比高于 a 的物品一定全部被我们买下来”成立。这样我们就证明了，这种贪心的策略能够得到最优解。

只有证明了某种贪心的策略是正确的，是能够在该问题中得到最优解的，它才能被我们所采用。

代码 2.14

```
#include <stdio.h>
#include <algorithm>
using namespace std;
struct goods { //表示可买物品的结构体
    double j; //该物品总重
    double f; //该物品总价值
    double s; //该物品性价比

    bool operator <(const goods &A) const { //重载小于运算符，确保可用sort函数
        将数组按照性价比降序排列
        return s > A.s;
    }
}buf[1000];
int main () {
    double m;
    int n;
    while (scanf ("%lf %d", &m, &n) != EOF) {
        if (m == -1 && n == -1) break; //当n == -1且m == -1时跳出循环，程序运
```

行结束

```
for (int i = 0; i < n; i++) {
    scanf ("%lf%lf", &buf[i].j, &buf[i].f); //输入
    buf[i].s = buf[i].j / buf[i].f; //计算性价比
}

sort(buf, buf + n); //使各物品按照性价比降序排列

int idx = 0; //当前货物下标

double ans = 0; //累加所能得到的总重量

while (m > 0 && idx < n) { //循环条件为,既有物品剩余(idx < n)还有钱剩余
(m > 0)时继续循环
    if (m > buf[idx].f) {
        ans += buf[idx].j;
        m -= buf[idx].f;
    } //若能买下全部该物品
    else {
        ans += buf[idx].j * m / buf[idx].f;
        m = 0;
    } //若只能买下部分该物品
    idx++; //继续下一个物品
}

printf("%.3lf\n", ans); //输出
}

return 0;
}
```

除了我们已经提到的贪心思想外,本题还有一个关注点。在本题中,题面中规定了程序退出条件,即当 n 和 m 都等于 -1 时程序退出,并对该组数据不作输出(也没法输出, -1 件物品?)。那么我们在编写程序时就要特别为此加入判断语句, `if (m == -1 && n == -1) break`。读者应做到仔细读题,确定究竟题目是否约定了一种退出条件,从而保证程序不会出现莫名其妙的超时错误。

在了解该例以后,读者可能会想当然地认为贪心算法不过如此,这一切看起来就像理所当然的本能而已。是的,贪心是一种十分简单而不需要过多处理的算法,但是如何选择正确的贪心策略并不是每次都是显而易见的。

例 2.12 今年暑假不 AC (九度教程第 22 题)

时间限制: 1 秒

内存限制: 128 兆

特殊判题: 否

题目描述:

“今年暑假不 AC?” “是的。” “那你干什么呢?” “看世界杯呀，笨蛋!” “@#\$%^&*%...” 确实如此，世界杯来了，球迷的节日也来了，估计很多 ACMer 也会抛开电脑，奔向电视作为球迷，一定想看尽量多的完整的比赛，当然，作为新时代的好青年，你一定还会看一些其它的节目，比如新闻联播（永远不要忘记关心国家大事）、非常 6+7、超级女生，以及王小丫的《开心辞典》等等，假设你已经知道了所有你喜欢看的电视节目的转播时间表，你会合理安排吗？（目标是能看尽量多的完整节目）

输入:

输入数据包含多个测试实例，每个测试实例的第一行只有一个整数 n ($n \leq 100$)，表示你喜欢看的节目的总数，然后是 n 行数据，每行包括两个数据 Ti_s, Ti_e ($1 \leq i \leq n$)，分别表示第 i 个节目的开始和结束时间，为了简化问题，每个时间都用一个正整数表示。 $n=0$ 表示输入结束，不做处理。

输出:

对于每个测试实例，输出能完整看到的电视节目的个数，每个测试实例的输出占一行。

样例输入:

```
12
1 3
3 4
0 7
3 8
15 19
15 20
10 15
8 18
6 12
5 10
4 14
2 9
0
```

样例输出:

```
5
```

读者应该已经发现，此题的贪心策略就不再像上题一样那么显而易见了。读者可以在继续阅读之前，自己先考虑一下，何种贪心策略可以被应用到该题当中。

我们首先来思考这样一个问题：第一个节目我们应该选什么。

读者可能会有以下猜测过程。

选择开始时间最早的？

假如有电视节目 $A[0,5]$, $B[1,2]$, $C[3,4]$ 。显然，选择最先开始的节目并不一定能够得到最优解。

选择持续时间最短的？

假如电视节目是这样安排的 $A[0,10]$, $B[11,20]$, $C[9,12]$ 。显然，选择时间最短的节目也并不一定能够得到最优解。

那么选择结束时间最早的？

这在以上两组案例中优先选择结束时间最早的节目是可以得到最优解的。那么它是否就真的是我们所需要的贪心策略？我们可以试着先来证明该命题：最优解中，第一个观看的节目一定是所有节目里结束时间最早的节目。因为按照优先选择结束时间最早的节目，我们所观看的第一个节目一定是所有节目里结束时间最早的。

同样的我们用反证法来证明：假设命题：最优解中，第一个观看的节目 $A[s_1, e_1]$ 不是所有节目时间里结束时间最早的节目。即，存在节目 $B[s_2, e_2]$ ，其中 $e_2 < e_1$ 。那么 B 节目一定不在该解当中，因为若在，其顺序一定在 A 节目之前，但是 A 节目已经被假定为第一个节目，所以 B 节目一定没被我们收看。那么，我们可以将该解中的第一个节目 A 替换为 B 节目，该替换保证是合法的，即去除 B 节目以后，其它节目的播出时间一定不会与 A 节目冲突。做这样的替换以后，原解与当前解除了第一个节目不同（由节目 B 变为节目 A ），其它节目安排完全相同。那么这两组解所包含的节目数是一模一样的，该解也是最优解。

由以上证明可见，如果最优解的第一个节目并不是结束最早的节目，那么我们可以直接用结束时间最早的节目代替该解中的第一个节目，替换后的解也是最优解。这样，我们就可以得出当第一个节目选择所有节目中结束时间最早的节目，这样是一定不会得不到最优解的。于是，在我们所要求的最优解中，第一个被收看的节目可以安排所有节目中结束时间最早的节目（若有多个，则可任意选择一个）。

当第一个被收看的节目被决定了以后，那么第二个呢？

只要不断重复上述证明过程，我们就会知道：在选择第 x ($x \geq 1$) 个节目时，一定是选择在收看完前 $x-1$ 个节目后，其它所有可以收看节目中结束时间最早的节目，这就是我们要找的贪心策略。在每次选择节目时，都不断的利用这种贪心策略，我们就能完成最优解的求解。

代码 2.15


```

#include <stdio.h>
#include <algorithm>
using namespace std;
struct program { //电视节目结构体
    int startTime; //节目开始时间
    int endTime; //节目结束时间
    bool operator < (const program & A) const { //重载小于号, 保证sort函数能够
按照结束时间升序排列
        return endTime < A.endTime;
    }
}buf[100];
int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        if (n == 0) break;
        for (int i = 0; i < n; i++) {
            scanf ("%d%d", &buf[i].startTime, &buf[i].endTime);
        } //输入
        sort(buf, buf + n); //按照结束时间升序排列
        int currentTime = 0, ans = 0; //记录当前时间变量初始值为0, 答案计数初始
值为0
        for (int i = 0; i < n; i++) { //按照结束时间升序便利所有的节目
            if (currentTime <= buf[i].startTime) { //若当前时间小于等于该节目
开始时间, 那么收看该在剩余节目里结束时间最早的节目
                currentTime = buf[i].endTime; //当前时间变为该节目结束时间
                ans++; //又收看了一个节目
            }
        }
        printf("%d\n", ans); //输出
    }
    return 0;
}

```

由该例读者应该能够发现, 虽然贪心算法的思想相对简单, 但是有些时候选择一个合适的贪心策略确实需要一定的技巧, 读者可在自行练习时慢慢总结。

练习题：迷瘴（九度教程第 23 题）； Repair the wall（九度教程第 24 题）； To fill or not to fill（九度教程第 25 题）（此题即为作者参加浙大计算机研究生复试机试所遇到的贪心，贪心策略较复杂，是当次考试通过率最低的考题，全场完全正确的仅有三人）。

总结

本章介绍了排序、日期类问题、Hash、排版、查找、贪心等在机试中频繁出现的基本算法。读者学习完本章后，应不仅对以上基本算法有一个较好的掌握，同时还应对机试题的考察方式、注意要点，特别是复杂度的估计有一个基本的理解。这样，机试中的大部分简单题相信已经难不倒你了。



第3章 数据结构

本章介绍机试题中涉及数据结构的部分，主要包括堆栈的应用、求哈夫曼树和二叉树的相关问题。

一 栈的应用

堆栈是一种数据项按序排列的数据结构，只能在它的一端进行删除和插入。在本节中我们主要讨论其相关应用。

在使用堆栈之前，我们首先介绍标准模板库中的堆栈模板，通过对堆栈模板的使用可以使我们跳过对堆栈具体实现的编码，而专注于堆栈在程序中的应用。

我们用

```
stack<int> S;
```

定义一个保存元素类型为 `int` 的堆栈 `S`，这样所有有关堆栈实现的内部操作，标准模板库都已经帮我们实现了。

使用

```
S.push(i);
```

向堆栈中压进一个数值为 `i` 的元素。

使用

```
int x = S.top();
```

读取栈顶元素，并将其值赋予变量 `x`。

使用

```
S.pop();
```

弹出栈顶元素。

为了使用 `stack` 标准模板，我们还要在文件头部包括相应的预处理

```
#include <stack>
```

并声明使用标准命名空间。

了解了标准库中堆栈对象的使用之后，我们来看关于堆栈应用的一个例题，括号匹配。

例 3.1 括号匹配问题（九度教程第 26 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

在某个字符串（长度不超过 100）中有左括号、右括号和大小写字母；规定（与常见的算数式子一样）任何一个左括号都从内到外与在它右边且距离最近的右括号匹配。写一个程序，找到无法匹配的左括号和右括号，输出原来字符串，并在下一行标出不能匹配的括号。不能匹配的左括号用"\$"标注,不能匹配的右括号用"?"标注。

输入：

输入包括多组数据，每组数据一行，包含一个字符串，只包含左右括号和大小写字母，字符串长度不超过 100。

输出：

对每组输出数据，输出两行，第一行包含原始输入字符，第二行由"\$","?"和空格组成，"\$"和"?"表示与之对应的左括号和右括号不能匹配。

样例输入：

```
)(rttyy())sss)(
```

样例输出：

```
)(rttyy())sss)(
?          ?$
```

括号匹配问题是堆栈的一个典型应用。由于每一个右括号，必定是与在其之前的所有未被匹配的左括号中最靠右的一个匹配。若我们按照从左至右的顺序遍历字符串，并将遇到的所有左括号都放入堆栈中等待匹配；若在遍历过程中遇到一个右括号，由于按照从左向右的顺序遍历字符串，若此时堆栈非空，那么栈顶左括号即为与其匹配的左括号；相反，若堆栈为空，则表示在其之前不存在未被匹配的左括号，匹配失败。

代码 3.1

```
#include <stdio.h>
#include <stack>
using namespace std;
stack<int> S; //定义一个堆栈
char str[110]; //保存输入字符串
char ans[110]; //保存输出字符串
int main () {
    while (scanf ("%s", str) != EOF) { //输入字符串
        int i;
        for (i = 0; str[i] != 0; i++) { //从左到右遍历字符串
            if (str[i] == '(') { //若遇到左括号
```

```

        S.push(i); //将其数组下标放入堆栈中
        ans[i] = ' '; //暂且将对应的输出字符串位置改为空格
    }
    else if (str[i] == ')') { //若遇到右括号
        if (S.empty() == false) { //若此时堆栈非空
            S.pop(); //栈顶位置左括号与其匹配, 从栈中弹出该已经匹配的左括号
            ans[i] = ' '; //修改输出中该位置为空格
        }
        else ans[i] = '?'; //若堆栈为空, 则无法找到左括号与其匹配, 修改输出
        中该位置为?
    }
    else ans[i] = ' '; //若其为其它字符, 与括号匹配无关, 则修改输出为空格
}
while(!S.empty()) { //当字符串遍历完成后, 尚留在堆栈中的左括号无法匹配
    ans[ S.top() ] = '$'; //修改其在输出中的位置为$
    S.pop(); //弹出
}
ans[i] = 0; //为了使输出形成字符串, 在其最后一个字符后添加一个空字符
puts(str); //输出原字符串
puts(ans); //输出答案字符串
}
return 0;
}

```

括号匹配, 利用了从左往右遍历字符串时, 栈顶的左括号离当前位置最近的特性完成工作。它属于利用堆栈的性质完成的较为简单的应用。

堆栈还有一个著名的应用——表达式的求值。

例 3.2 简单计算器（九度教程第 27 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

读入一个只包含 +, -, *, / 的非负整数计算表达式, 计算该表达式的值。

输入：

测试输入包含若干测试用例, 每个测试用例占一行, 每行不超过 200 个字符, 整数和运算符之间用一个空格分隔。没有非法表达式。当一行中只有 0 时输入结

束，相应的结果不要输出。

输出：

对每个测试用例输出 1 行，即该表达式的值，精确到小数点后 2 位。

样例输入：

1 + 2

4 + 2 * 5 - 7 / 11

0

样例输出：

3.00

13.36

来源：

2006 年浙江大学计算机及软件工程研究生机试真题

利用堆栈对表达式求值的方法在任意一本数据结构教科书上都会做明确的阐述。这里简单的回顾：

1. 设立两个堆栈，一个用来保存运算符，另一个用来保存数字。

2. 在表达式首尾添加标记运算符，该运算符运算优先级最低。

3. 从左至右依次遍历字符串，若遍历到运算符，则将其与运算符栈栈顶元素进行比较，若运算符栈栈顶运算符优先级小于该运算符或者此时运算符栈为空，则将该运算符压入堆栈。遍历字符串中下一个元素。

4. 若运算符栈栈顶运算符优先级大于该运算符，则弹出该栈顶运算符，再从数字栈中依次弹出两个栈顶数字，完成弹出的运算符对应的运算得到结果后，再将该结果压入数字栈，重复比较此时栈顶运算符与当前遍历到的运算符优先级，视其优先级大小重复步骤 3 或步骤 4。

5. 若遍历到表达式中的数字，则直接压入数字栈。

6. 若运算符堆栈中仅存有两个运算符且栈顶元素为我们人为添加的标记运算符，那么表达式运算结束，此时数字堆栈中唯一的数字即为表达式的值。

下面给出详细的求解代码。

代码 3.2

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

using namespace std;

char str[220]; //保存表达式字符串

int mat[][5] = { //优先级矩阵, 若mat[i][j] == 1, 则表示i号运算符优先级大于j号运算符, 运算符编码规则为+为1号, -为2号, *为3号, /为4号, 我们人为添加在表达式首尾的标记
```

运算符为0号

```
1,0,0,0,0,  
1,0,0,0,0,  
1,0,0,0,0,  
1,1,1,0,0,  
1,1,1,0,0,  
};
```

```
stack<int> op; //运算符栈,保存运算符编号
```

```
stack<double> in; //数字栈,运算结果可能存在浮点数,所以保存元素为double
```

`void getOp(bool &reto, int &retn, int &i)` { //获得表达式中下一个元素函数,若函数运行结束时,引用变量reto为true,则表示该元素为一个运算符,其编号保存在引用变量retn中;否则,表示该元素为一个数字,其值保存在引用变量retn中.引用变量i表示遍历到的字符串下标

`if (i == 0 && op.empty() == true)` { //若此时遍历字符串第一个字符,且运算符栈为空,我们人为添加编号为0的标记字符

```
    reto = true; //为运算符
```

```
    retn = 0; //编号为0
```

```
    return; //返回
```

```
}
```

```
if (str[i] == 0) { //若此时遍历字符为空字符,则表示字符串已经被遍历完
```

```
    reto = true; //返回为运算符
```

```
    retn = 0; //编号为0的标记字符
```

```
    return; //返回
```

```
}
```

```
if (str[i] >= '0' && str[i] <= '9') { //若当前字符为数字
```

```
    reto = false; //返回为数字
```

```
}
```

```
else { //否则
```

```
    reto = true; //返回为运算符
```

```
    if (str[i] == '+') { //加号返回1
```

```
        retn = 1;
```

```
    }
```

```
    else if (str[i] == '-') { //减号返回2
```

```
        retn = 2;
```

```
    }
```



```

        else if (str[i] == '*') { //乘号返回3
            retn = 3;
        }
        else if (str[i] == '/') { //除号返回4
            retn = 4;
        }
        i += 2; //i 递增, 跳过该运算字符和该运算字符后的空格
        return; //返回
    }
    retn = 0; //返回结果为数字
    for (; str[i] != ' ' && str[i] != 0; i++) { //若字符串未被遍历完, 且下一个字
        符不是空格, 则依次遍历其后数字, 计算当前连续数字字符表示的数值
        retn *= 10;
        retn += str[i] - '0';
    } //计算该数字的数字值
    if (str[i] == ' ') //若其后字符为空格, 则表示字符串未被遍历完
        i++; //i 递增. 跳过该空格
    return; //返回
}

int main () {
    while(gets(str)) { //输入字符串, 当其位于文件尾时, gets返回0
        if (str[0] == '0' && str[1] == 0) break; //若输入只有一个0, 则退出
        bool retop; int retnum; //定义函数所需的引用变量
        int idx = 0; //定义遍历到的字符串下标, 初始值为0
        while(!op.empty()) op.pop();
        while(!in.empty()) in.pop(); //清空数字栈, 和运算符栈
        while(true) { //循环遍历表达式字符串
            getOp(retop, retnum, idx); //获取表达式中下一个元素
            if (retop == false) { //若该元素为数字
                in.push((double)retnum); //将其压入数字栈中
            }
            else { //否则
                double tmp;
                if (op.empty() == true || mat[retnum][op.top()] == 1) {

```

```

        op.push(retnum);
    } //若运算符堆栈为空或者当前遍历到的运算符优先级大于栈顶运算符, 将该运算符压入运算符堆栈

    else { //否则
        while(mat[retnum][op.top()] == 0) { //只要当前运算符优先级
            小于栈顶元素运算符, 则重复循环

            int ret = op.top(); //保存栈顶运算符
            op.pop(); //弹出
            double b = in.top();
            in.pop();
            double a = in.top();
            in.pop(); //从数字堆栈栈顶弹出两个数字, 依次保存在遍历a. b中
            if (ret == 1) tmp = a + b;
            else if (ret == 2) tmp = a - b;
            else if (ret == 3) tmp = a * b;
            else tmp = a / b; //按照运算符类型完成运算
            in.push(tmp); //将结果压回数字堆栈
        }
        op.push(retnum); //将当前运算符压入运算符堆栈
    }
}

if (op.size() == 2 && op.top() == 0) break; //若运算符堆栈只有两个元素, 且其栈顶元素为标记运算符, 则表示表达式求值结束

}

printf("%.2f\n", in.top()); //输出数字栈中唯一的数字, 即为答案
}

return 0;
}

```

本节主要介绍了括号匹配和表达式求值两个关于堆栈的经典应用。

对于堆栈的应用, 主要还是利用堆栈先进后出的访问规则解决一系列需要符合该访问特点的问题。在使用标准模板库中堆栈模板以后, 我们不再需要自行实现堆栈的相关操作, 而只需将注意力放在其它操作上即可。

练习题：堆栈的使用(九度教程第 28 题);表达式求值(九度教程第 29 题);

二 哈夫曼树

本节介绍哈夫曼树的求解。

在一棵树中，从任意一个结点到达另一个结点的通路被称为路径，该路径上所需经过的边的个数被称为该路径的长度。若树中结点带有表示某种意义的权值，那么从根结点到达该节点的路径长度再乘以该结点权值被称为该结点的带权路径长度。树所有的叶子结点的带权路径长度和为该树的带权路径长度和。给定 n 个结点和它们的权值，以它们为叶子结点构造一棵带权路径和最小的二叉树，该二叉树即为哈夫曼树，同时也被称为最优树。

给定结点的哈夫曼树可能不唯一，所以关于哈夫曼树的机试题往往需要求解的是其最小带权路径长度和。回顾一下我们所熟知的哈夫曼树求法。

1.将所有结点放入集合 K 。

2.若集合 K 中剩余结点大于 2 个，则取出其中权值最小的两个结点，构造他们同时为某个新节点的左右儿子，该新节点是他们共同的双亲结点，设定它的权值为其两个儿子结点的权值和。并将该父亲结点放入集合 K 。重复步骤 2 或 3。

3.若集合 K 中仅剩余一个结点，该结点即为构造出的哈夫曼树数的根结点，所有构造得到的中间结点(即哈夫曼树上非叶子结点)的权值和即为该哈夫曼树的带权路径和。

为了方便快捷高效率的求得集合 K 中权值最小的两个元素，我们需要使用堆数据结构。它可以以 $O(\log n)$ 的复杂度取得 n 个元素中的最小元素。为了绕过对堆的实现，我们使用标准模板库中的相应的标准模板——优先队列。

利用语句

```
priority_queue<int> Q;
```

建立一个保存元素为 `int` 的堆 Q ，但是请特别注意这样建立的堆其默认为大顶堆，即我们从堆顶取得的元素为整个堆中最大的元素。而在求哈夫曼树中，我们恰恰需要取得堆中最小的元素，于是我们使用如下语句定义一个小顶堆：

```
priority_queue<int, vector<int>, greater<int>> > Q;
```

关于堆的有关操作如下：

```
Q.push(x);
```

将元素 x 放入堆 Q 中。

```
int a = Q.top();
```

取出堆顶元素，即最小的元素保存在 a 中。

```
Q.pop();
```

弹出堆顶元素，取出后堆会自动调整为一个新的小顶堆。

它的定义与之前我们使用过的队列一样在标准模板库 `queue` 中，所以在使用它之前我们必须做相应预处理。

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

例 3.3 哈夫曼树（九度教程第 30 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

哈夫曼树，第一行输入一个数 n ，表示叶结点的个数。需要用这些叶结点生成哈夫曼树，根据哈夫曼树的概念，这些结点有权值，即 `weight`，题目需要输出所有结点的值与权值的乘积之和。

输入：

输入有多组数据。

每组第一行输入一个数 n ，接着输入 n 个叶节点（叶节点权值不超过 100， $2 \leq n \leq 1000$ ）。

输出：

输出权值。

样例输入：

5

1 2 2 5 9

样例输出：

37

来源：

2010 年北京邮电大学计算机研究生机试真题

我们以这个例题为例，求解哈夫曼树。

代码 3.3

```
#include <queue>
#include <stdio.h>
using namespace std;
priority_queue<int, vector<int>, greater<int>> > Q; //建立一个小顶堆
int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        while(Q.empty() == false) Q.pop(); //清空堆中元素
```

```

    for (int i = 1; i <= n; i++) { //输入n个叶子结点权值
        int x;
        scanf ("%d", &x);
        Q.push(x); //将权值放入堆中
    }

    int ans = 0; //保存答案
    while(Q.size() > 1) { //当堆中元素大于1个
        int a = Q.top();
        Q.pop();
        int b = Q.top();
        Q.pop(); //取出堆中两个最小元素, 他们为同一个结点的左右儿子, 且该双亲结点的权值为它们的和

        ans += a + b; //该父亲结点必为非叶子结点, 固累加其权值
        Q.push(a + b); //将该双亲结点的权值放回堆中
    }

    printf("%d\n", ans); //输出答案
}

return 0;
}

```

在使用了优先队列以后, 求哈夫曼树的过程不仅时间复杂度降低许多 ($O(n\log n)$), 同时代码也轻便不少, 所以使用数据结构堆来辅助求解哈夫曼树, 是求哈夫曼树的最佳选择。

学会求解哈夫曼树, 还要学会在实际运用中确定哈夫曼树模型。最经典的问题模型为哈夫曼编码、多个数的两两合并等。关于哈夫曼树的相关应用, 我们在练习题中给出, 供读者参考。

练习题搬水果(九度教程第 31 题);

三 二叉树

本节讨论二叉树。读者对二叉树的定义、形态、特点应该不会陌生, 所以这里不再对它们进行重复。我们对二叉树的各种理论避而不谈, 而是将重点放在其操作上。

我们从二叉树的遍历谈起。

众所周知, 在对二叉树的遍历过程中, 根据遍历每一个结点的左子树、结点

本身、右子树的顺序不同可将对二叉树的遍历方法分为前序遍历、中序遍历、后序遍历。我们摒弃数据结构教科书上复杂的遍历方式，而是使用我们在上一章所重点讨论过的递归程序来简单的实现它。

假设二叉树结点由以下结构体表示：

```
struct Node {  
    Node *lchild; //指向其左儿子结点的指针，当其不存在左儿子时为NULL  
    Node *rchild; //指向其右儿子结点的指针，当其不存在右儿子时为NULL  
    /*  
    *  
    *  
    其它结点信息*/  
};
```

我们以中序遍历为例，给出其遍历方法。

```
void inOrder (Node *Tree) {  
    if (Tree -> lchild != NULL) //递归遍历左子树  
        inOrder(Tree -> lchild);  
    /*  
    *  
    *  
    *对当前结点Tree作遍历操作/  
    if (Tree -> rchild != NULL) //递归遍历右子树  
        inOrder(Tree -> rchild);  
    return;  
}
```

如读者所见,用递归方式编写的二叉树遍历代码较原始使用堆栈来编写的相同功能代码，在代码量上得到巨大的优化。为了完成对二叉树的中序遍历，在遍历任意一个结点 **Tree** 时，我们首先递归遍历其左儿子及其子树，再遍历该结点本身，最后遍历其右儿子及其子树，从而完成对二叉树的中序遍历。

相同的，若我们需要其他两种形式的遍历方式，只需简单的修改遍历自身结点和递归遍历左右儿子的相关语句顺序即可。

关于二叉树的三种遍历常有以下类型例题。

例 3.4 二叉树遍历（九度教程第 32 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

二叉树的前序、中序、后序遍历的定义：

前序遍历：对任一子树，先访问根，然后遍历其左子树，最后遍历其右子树；

中序遍历：对任一子树，先遍历其左子树，然后访问根，最后遍历其右子树；

后序遍历：对任一子树，先遍历其左子树，然后遍历其右子树，最后访问根。

给定一棵二叉树的前序遍历和中序遍历，求其后序遍历（提示：给定前序遍历与中序遍历能够唯一确定后序遍历）。

输入：

两个字符串，其长度 n 均小于等于 26。

第一行为前序遍历，第二行为中序遍历。二叉树中的结点名称以大写字母表示：A, B, C....最多 26 个结点。

输出：

输入样例可能有多组，对于每组测试样例，输出一行，为后序遍历的字符串。

样例输入：

ABC

BAC

FDXEAG

XDEFAG

样例输出：

BCA

XEDGAF

来源：

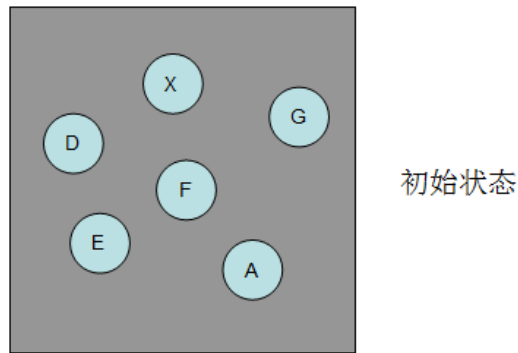
2006 年清华大学计算机研究生机试真题

该例题涉及二叉树的建立、由二叉树的两种遍历结果还原二叉树、二叉树的遍历等多种知识点。我们以分析该例题为例，介绍关于二叉树各知识点。

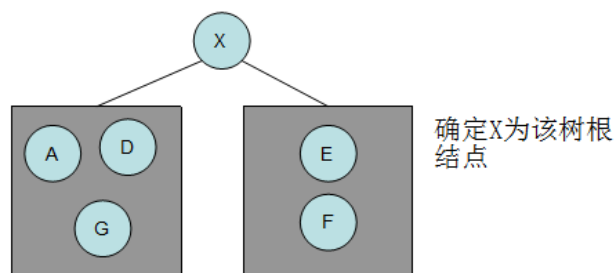
由该例要求，首先我们需要根据给定的二叉树前序和中序遍历结果还原该二叉树。其次，我们需要将还原的二叉树以二叉树的形式保存在内存中。最后，我们需要对建立的二叉树进行后序遍历。

后序遍历在前文中已经有所提及。下面我们对前两部分做重点的阐述。

由给定的前序和中序遍历还原得到该二叉树。以前序遍历结果 **XDAGFE**，和中序遍历结果 **ADGXFE** 为例详细讨论其还原方法。

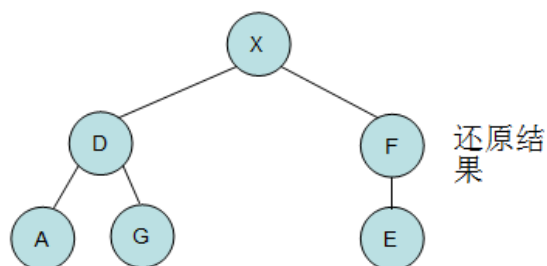


由前序遍历结果的首个元素为 **X** 可知，原树必是由 **X** 为根结点。在中序遍历中，遍历结果 **ADGXFE** 以 **X** 为界分为两个子串。其中第一个子串 **ADG** 为 **X** 的左子树的中序遍历结果，第二个子串 **FE** 为 **X** 的右子树的中序遍历结果。这样我们知道 **X** 的左子树具有 3 个元素，**X** 的右子树具有 2 个元素。根据元素的数量我们同样可以得知，在先序遍历中去除根结点 **X** 后剩余的串 **DAGFE** 中，前 3 个字符 **DAG** 为 **X** 的左子树的前序遍历结果，后 2 个字符 **FE** 为 **X** 的右子树的前序遍历结果。



同样的对于确定的左子树前序遍历结果 **DAG** 和中序遍历结果 **ADG** 重复以上确定过程，可知 **D** 为该子树根结点，其左儿子为 **A**，右儿子为 **G**。

X 的右子树前序遍历结果 **FE** 和中序遍历结果 **FE** 同样可以确定该子树以 **F** 为根节点，其左儿子不存在，右儿子为 **E**。



这样我们就还原了原始二叉树。

我们还需将还原出来的树保存在内存中。使用结构体：

```
struct Node {
```

```

Node *lchild;
Node *rchild;
char c;
}

```

表示树的一个结点，其字符信息保存在字符变量 `c`，若该结点存在左儿子或者右儿子，则指向他们的指针保存在 `lchild` 或 `rchild` 中，否则该指针为空。

下面给出本例代码，详细了解其实现。

代码 3.4

```

#include <stdio.h>
#include <string.h>
struct Node { //树结点结构体
    Node *lchild; //左儿子指针
    Node *rchild; //右儿子指针
    char c; //结点字符信息
}Tree[50]; //静态内存分配数组
int loc; //静态数组中已经分配的结点个数
Node *creat() { //申请一个结点空间, 返回指向其的指针
    Tree[loc].lchild = Tree[loc].rchild = NULL; //初始化左右儿子为空
    return &Tree[loc++]; //返回指针, 且loc累加
}
char str1[30], str2[30]; //保存前序和中序遍历结果字符串
void postOrder(Node *T) { //后序遍历
    if (T -> lchild != NULL) { //若左子树不为空
        postOrder(T -> lchild); //递归遍历其左子树
    }
    if (T -> rchild != NULL) { //若右子树不为空
        postOrder(T -> rchild); //递归遍历其右子树
    }
    printf("%c", T -> c); //遍历该结点, 输出其字符信息
}
Node *build(int s1, int e1, int s2, int e2) { //由字符串的前序遍历和中序遍历还原树, 并返回其根节点, 其中前序遍历结果为由str1[s1]到str1[e1], 中序遍历结果为由str2[s2]到str2[e2]
    Node* ret = creat(); //为该树根节点申请空间

```

```

ret -> c = str1[s1]; //该结点字符为前序遍历中第一个字符
int rootIdx;
for (int i = s2; i <= e2; i++) { //查找该根节点字符在中序遍历中的位置
    if (str2[i] == str1[s1]) {
        rootIdx = i;
        break;
    }
}
if (rootIdx != s2) { //若左子树不为空
    ret -> lchild = build(s1 + 1, s1 + (rootIdx - s2), s2, rootIdx - 1); //
递归还原其左子树
}
if (rootIdx != e2) { //若右子树不为空
    ret -> rchild = build(s1 + (rootIdx - s2) + 1, e1, rootIdx + 1, e2); //
递归还原其右子树
}
return ret; //返回根节点指针
}

int main () {
while (scanf ("%s", str1) != EOF) {
    scanf ("%s", str2); //输入
    loc = 0; //初始化静态内存空间中已经使用结点个数为0
    int L1 = strlen(str1);
    int L2 = strlen(str2); //计算两个字符串长度
    Node *T = build(0, L1 - 1, 0, L2 - 1); //还原整棵树, 其根结点指针保存在T中
    postOrder(T); //后序遍历
    printf("\n"); //输出换行
}
return 0;
}

```

在本例代码中我们并没有动态的申请内存空间,并在程序结束时释放这些空间。而是使用了静态数组,利用分配数组元素给相应的结点实现内存分配。这是对内存分配较为简单的实现方法,若读者对动态的申请和释放内存没有把握,或者对何时何地释放内存抱有疑惑,建议使用该较为保险的方法。

在本段代码中，包括了建树、遍历、还原等多个二叉树相关的操作，几乎涉及了机试中二叉树的所有考点，建议读者仔细研读并做适当记忆。

关于二叉树的其它考点，都与二叉树的性质有关，如每层的结点个数、确定结点个数的最小树高等。对于这些考点，考生只需对理论知识有一定的把握即可，在编码上可以说没有任何难度。

练习题：二叉树(九度教程第 33 题);树查找(九度教程第 34 题);

四 二叉排序树

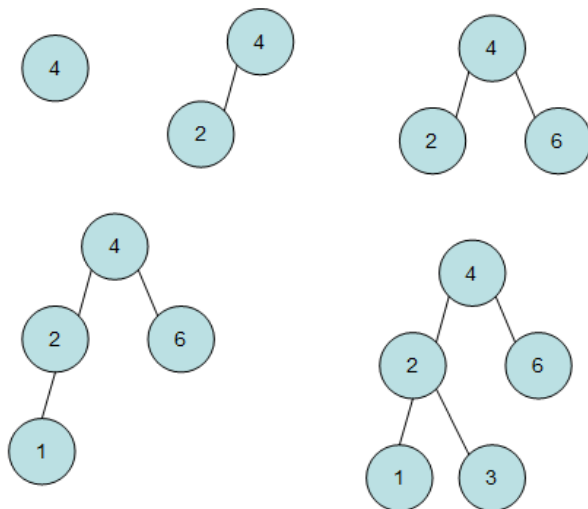
二叉排序树是一棵特殊的二叉树，它是一棵二叉树但同时满足如下条件：对于树上任意一个结点，其上的数值必大于等于其左子树上任意结点数值，必小于等于其右子树上任意结点的数值。

二叉排序树的存储方式与二叉树保持一致，我们更多的关注它独有的操作。

我们从二叉树的插入开始了解其建树方式，对二叉排序树插入数字 x ：

1.若当前树为空，则 x 为其根结点。

2.若当前结点大于 x ，则 x 插入其左子树；若当前结点小于 x ，则 x 插入其右子树；若当前结点等于 x ，则根据具体情况选择插入左右子树或者直接忽略。以插入 4、2、6、1、3 为例，其二叉排序树变化情况如下图。



由于各个数字插入的顺序不同，所得到的二叉排序树的形态也很可能不同，所以不同的插入顺序对二叉排序树的形态有重要的影响。但是，所有的二叉排序树都有一个共同的特点：若对二叉排序树进行中序遍历，那么其遍历结果必然是一个递增序列，这也是二叉排序树名字的来由，通过建立二叉排序树就能对原无序序列进行排序，并实现动态维护。

例 3.5 二叉排序树（九度教程第 35 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

输入一系列整数，建立二叉排序数，并进行前序，中序，后序遍历。

输入：

输入第一行包括一个整数 $n(1 \leq n \leq 100)$ 。接下来的一行包括 n 个整数。

输出：

可能有多组测试数据，对于每组数据，将题目所给数据建立一个二叉排序树，并对二叉排序树进行前序、中序和后序遍历。每种遍历结果输出一行。每行最后一个数据之后有一个空格。

样例输入：

```
5
1 6 5 9 8
```

样例输出：

```
1 6 5 9 8
1 5 6 8 9
5 8 9 6 1
```

提示：

输入中可能有重复元素，但是输出的二叉树遍历序列中重复元素不用输出。

来源：

2005 年华中科技大学计算机保研机试真题

该例考察要点为对于给定的输入顺序建立二叉排序树，并对二叉排序树进行前序、中序、后序遍历并输出遍历结果。

代码 3.5

```
#include <stdio.h>
#include <string.h>
struct Node { //二叉树结构体
    Node *lchild; //左儿子指针
    Node *rchild; //右儿子指针
    int c; //保存数字
}Tree[110]; //静态数组
int loc; //静态数组中被使用元素个数
Node *creat() { //申请未使用的结点
    Tree[loc].lchild = Tree[loc].rchild = NULL;
    return &Tree[loc++];
```

```

}

void postOrder(Node *T) { //后序遍历
    if (T -> lchild != NULL) {
        postOrder(T -> lchild);
    }
    if (T -> rchild != NULL) {
        postOrder(T -> rchild);
    }
    printf("%d ", T -> c);
}

void inOrder(Node *T) { //中序遍历
    if (T -> lchild != NULL) {
        inOrder(T -> lchild);
    }
    printf("%d ", T -> c);
    if (T -> rchild != NULL) {
        inOrder(T -> rchild);
    }
}

void preOrder(Node *T) { //前序遍历
    printf("%d ", T -> c);
    if (T -> lchild != NULL) {
        preOrder(T -> lchild);
    }
    if (T -> rchild != NULL) {
        preOrder(T -> rchild);
    }
}

Node *Insert(Node *T, int x) { //插入数字
    if (T == NULL) { //若当前树为空
        T = creat(); //建立结点
        T -> c = x; //数字直接插入其根结点
        return T; //返回根结点指针
    }
}

```

```

else if (x < T->c) //若x小于根结点数值
    T -> lchild = Insert(T -> lchild, x); //插入到左子树上
else if (x > T->c) //若x大于根结点数值
    T -> rchild = Insert(T -> rchild, x); //插入到右子树上. 若根结点数值与x
一样, 根据题目要求直接忽略

return T; //返回根节点指针
}

int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        loc = 0;
        Node *T = NULL; //二叉排序树树根结点为空
        for (int i = 0; i < n; i++) { //依次输入n个数字
            int x;
            scanf ("%d", &x);
            T = Insert(T, x); //插入到排序树中
        }
        preOrder(T); //前序遍历
        printf("\n"); //输出空行
        inOrder(T); //中序遍历
        printf("\n");
        postOrder(T); //后序遍历
        printf("\n");
    }
    return 0;
}

```

在学习了二叉排序树的建立和三种方式的遍历以后, 我们还要接触一种特殊的树操作——判断两棵二叉树是否相同。

判断两棵树是否相同, 我们不能简单的用某一种遍历方式去遍历两棵树, 并判断遍历的结果是否相同, 这种方法是错误的。由于一种遍历顺序并不能唯一的确定一棵二叉树, 所以两棵不同的树的某一种遍历顺序是可能相同的。如数字相同, 插入顺序不同而建立的两棵二叉排序树, 它们的中序遍历一定是一样的。但在之前例题中我们已经看到, 包括中序遍历在内的两种遍历结果可以唯一得确定一棵二叉树, 那么我们只需对两棵树进行包括中序遍历在内的两种遍历, 若两种

遍历的结果都相同，那么就可以判定两棵树是完全相同的。

例 3.6 二叉搜索树（九度教程第 36 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

判断两序列是否为同一二叉搜索树序列

输入：

开始一个数 n ，($1 \leq n \leq 20$) 表示有 n 个需要判断， $n=0$ 的时候输入结束。接下去一行是一个序列，序列长度小于 10，包含(0~9)的数字，没有重复数字，根据这个序列可以构造出一颗二叉搜索树。

接下去的 n 行有 n 个序列，每个序列格式跟第一个序列一样，请判断这两个序列是否能组成同一颗二叉搜索树。

输出：

如果序列相同则输出 YES，否则输出 NO

样例输入：

```
2
567432
543267
576342
0
```

样例输出：

```
YES
NO
```

来源：

2010 年浙江大学计算机及软件工程研究生机试真题

我们对输入的数字序列构建二叉排序树，并对它们进行前序和中序的遍历，依次比较两次遍历结果是否相同，若相同则说明两棵二叉排序树相同，否则不同。

代码 3.6

```
#include <stdio.h>
#include <string.h>
struct Node { //树节点结构体
    Node *lchild;
    Node *rchild;
    int c;
```

```

}Tree[110];

int loc;

Node *creat() { //申请结点空间
    Tree[loc].lchild = Tree[loc].rchild = NULL;
    return &Tree[loc++];
}

char str1[25] , str2[25]; //保存二叉排序树的遍历结果, 将每一棵树的前序遍历得到
的字符串与中序遍历得到的字符串连接, 得到遍历结果字符串

int size1 , size2; //保存在字符数组中的遍历得到字符个数

char *str; //当前正在保存字符串

int *size; //当前正在保存字符串中字符个数

void postOrder(Node *T) { //前序遍历
    if (T -> lchild != NULL) {
        postOrder(T -> lchild);
    }
    if (T -> rchild != NULL) {
        postOrder(T -> rchild);
    }
    str[ (*size) ++ ] = T -> c + '0'; //将结点中的字符放入正在保存的字符串中
}

void inOrder(Node *T) { //中序遍历
    if (T -> lchild != NULL) {
        inOrder(T -> lchild);
    }
    str[ (*size) ++ ] = T -> c + '0';
    if (T -> rchild != NULL) {
        inOrder(T -> rchild);
    }
}

Node *Insert(Node *T, int x) { //将数字插入二叉树
    if (T == NULL) {
        T = creat();
        T -> c = x;
        return T;
    }

```

```

}
else if (x < T->c)
    T -> lchild = Insert(T -> lchild, x);
else if (x > T->c)
    T -> rchild = Insert(T -> rchild, x);
return T;
}

int main () {
    int n;
    char tmp[12];
    while (scanf ("%d", &n) != EOF && n != 0) {
        loc = 0; //初始化静态空间为未使用
        Node *T = NULL;
        scanf ("%s", tmp); //输入字符串
        for (int i = 0; tmp[i] != 0; i++) {
            T = Insert(T, tmp[i] - '0'); //按顺序将数字插入二叉排序树
        }
        size1 = 0; //保存在第一个字符串中的字符初始化为0
        str = str1; //将正在保存字符串设定为第一个字符串
        size = &size1; //将正在保存字符串中的字符个数指针指向size1
        postOrder(T); //前序遍历
        inOrder(T); //中序遍历
        str1[ size1 ] = 0; //向第一个字符串的最后一个字符后添加空字符, 方便使用字符串函数

        while(n -- != 0) { //输入n个其它字符串
            scanf ("%s", tmp); //输入
            Node *T2 = NULL;
            for (int i = 0; tmp[i] != 0; i++) { //建立二叉排序树
                T2 = Insert(T2, tmp[i] - '0');
            }
            size2 = 0; //第二个字符串保存字符初始化为0
            str = str2; //将正在保存字符串设定为第二个字符串
            size = &size2; //正在保存字符串中字符数量指针指向size2
            postOrder(T2); //前序遍历
        }
    }
}

```

```

        inOrder(T2); //中序遍历

        str2[ size2 ] = 0; //字符串最后添加空字符

        puts (strcmp(str1,str2) == 0 ? "YES" : "NO"); //比较两个遍历字符串, 若相同则输出YES, 否则输出NO
    }
}
return 0;
}

```

同样的, 我们也可以选择中序和后序的排序结果共同对两棵树进行判定。但是请注意, 在选择两种遍历方式中必须要包括中序遍历。如在数据结构中所讲的, 只有包括中序的两种遍历顺序才能唯一的确定一棵二叉树。

最后, 我们对二叉排序树的删除作适当的补充。二叉排序树的删除在机试题中考察的概率非常小, 在之前我们已经得到的机试题中没有对其进行任何的考察。

要删除二叉排序树上的某一个结点, 我们按如下步骤进行:

1. 利用某种遍历找到该结点。
2. 若该结点为叶子结点, 则直接删除它, 即将其双亲结点中指向其的指针改为 NULL。释放该节点空间。
3. 若该结点仅不存在右子树, 则直接将其左子树的根结点代替其位置后, 删除该结点。即将其双亲结点指向其的指针改为指向其的左子树树根。
4. 若该节点存在右子树, 则找到右子树上最右下的结点 (即中序遍历中该子树上第一个被遍历到的结点), 将被删除结点的数值改为右子树上最右下结点的数值后, 删除最右下结点。

删除二叉树的原理非常简单, 即删除该结点后, 其中序遍历依然保持关键字递增的顺序, 只要符合这个条件, 不同于上述规则的删除也是可行的。

本节主要讨论二叉排序树的相关机试考题, 但是很遗憾, 由于二叉排序树在机试中出现的次数并不多, 我们并没有找到合适的机试题作为本节的练习题, 读者还请独立完成本节中所讨论过的例题作为巩固。

总结

本章主要讨论机试中有关数据结构的相关问题, 主要涉及栈的两个应用——括号匹配和表达式求值、哈夫曼树的建立、二叉树和二叉排序树的相关操作。数据结构在机试中考察的难度不大, 牢记几个经典数据结构的应用和实现即可。

第4章 数学问题

本章我们将着重讨论机试中将会涉及的一系列数学问题，包括数位拆解、分解素因数等考察频率较高的知识点，并提出求最小公倍数、最大公约数的基本方法，最后我们还将着重讨论高精度整数运算的实现。旨在使读者对计算机考研机试中所涉及的数学问题有一个很好的掌握。

一 %运算符

相信熟悉 C/C++ 的读者对 % 运算符一定不会陌生，我们将其称为求模运算符，通俗的讲即求一个数被另一个数除后剩余的余数。本节对该运算符的特点做一定的阐述。

% 运算符的用法非常简单，我们用形如 $a \% b$ 的语句来调用该运算符。其中变量 a , b 必须为整型变量，例如 `int`、`short` 等，而不能为浮点数。且 b 变量必须为非零值，若出现模零错误，程序会因为该异常意外终止。在评判系统中表现为评判系统给出了运行时错误，程序未运行完成就异常终止。所以若读者在练习、考试中出现了评判系统返回了运行时错误，可以试着检查是否可能出现模零错误。

以 $a \% b$ 语句为例，我们先不加说明的指出该运算的特点。其运算在行为上好像是按如下步骤进行的，首先计算出 a 的绝对值被 b 的绝对值除所得的余数，再使该余数的符号与 a 保持一致。即若 a 为正数，则该表达式结果必为非负数（可能为 0）；若 a 为负数，则表达式结果必为非正数（可能为 0）。而表达式结果与 b 的符号没有直接关系，即 $a \% -b$ 与 $a \% b$ 的结果相同。

我们注意到，通过求模运算符求得的余数存在着负数的可能。而这与数论中关于余数的定义是不相符的。数论指出，余数的取值范围为从 0 到除数减 1，即在 $a \% b$ 表达式中，其符合数论规定的结果取值范围应是 0 到 $b - 1$ 。% 运算符的运算特性仅保证余数的绝对值在如上所述的范围内，而不保证不会出现负数，出现负余数也为我们下一步操作带来诸多不便。如上一章中例 2.4 所示，我们利用求模运算来计算数组下标，而负数组下标是不能被我们所使用的。那么我们必须保证表达式求得的余数在数论定义的区间范围内。结合例 2.4 中的方法，相信很多读者心中都有答案，我们只需在该负的余数上再加上除数再对除数求一次余

$$r = a \% b;$$

$$a = k * b + r;$$

即可。那么它的原理又是如何的呢？我们来看下两式：

r 即为我们要求的余数，它是由第一式求得的。同时它应符合第二式中关于余数的原始定义，即 a 将等于某个整数与 b 的积再加上余数 r ，由于 C/C++ 的 % 算符特点，当 a 为负数时 r 很可能出现负数（或为 0），我们为了得到正确范围内的余数，我们可以对该式作如下变形（这里假设 b 大于 0，否则取绝对值）：

$$a = (k - 1) * b + r + b;$$

若 r 非零，该式同样符合关于余数的相关定义，但是它的余数部分 $(r + b)$ 将不再为负数，而是落在我们之前讨论的、数论规定的范围 $[0, b-1]$ 内，即由 0 至 $b-1$ 。而这个符合我们要求的新余数即为原负余数加上 b ，我们正是利用该方法来使余数落入所需的区间内。但是读者还需特别注意，即使被除数为负数，余数也是有可能为 0 的（刚好整除），那么假如我们与对待其他负余数一样为其加上余数以期其能够落入我们需要的区间内，这将会适得其反（将会使余数等于 b ）。所以我们可以统一的对取得的余数加上除数后再对该和求模，即：

$$r' = (r + b) \% b;$$

这样做，不仅能对可能出现的负余数做适当的修正，同时对出现的零和正余数也不会改变他们的值，在例 2.4 中我们正是利用该方法，对所有求得的余数都做了修正，保证其将会落在我们需要的区间内。

另外，我们也可以顺便来看一下为什么在 % 运算中余数的值看起来好像与除数的符号无关。我们假设 b 为正数，我们利用 $r = a \% b$ 求得的余数 r 将会满足下式：

$$a = k * b + r;$$

其中 r 绝对值的取值范围为从 0 到 $b-1$ ，其符号保持与 a 一致（除非 r 为 0）。若此时，我们利用 $r' = a \% -b$ 来求得的余数 r' 又会满足下式

$$a = k' * (-b) + r';$$

同样，其中 r 的取值范围为从 0 到 $b-1$ ，其符号保持与 a 一致。比较两式我们即能发现，当 k' 等于 $-k$ 时两式成立，且 r' 与 r 相同。这就是为什么我们用 % 运算符求得的余数看起来好像与除数的符号无关的原因。

最后我们总结一下 % 运算的数学解释，以式 $r = a \% b$ 为例，其所求得的 r

$$a = k * b + r$$

将满足下式：

其中 k 为某整数， r 的绝对值取值范围是 $[0, b-1]$ ，其符号将与 a 保持一致，除非其为 0。这就是我们需要了解的 % 运算符的工作特点。

$$(a * b) \% c = (a \% c * b \% c) \% c;$$

$$(a + b) \% c = (a \% c + b \% c) \% c;$$

另外 % 运算还具有如下运算规律，牢记这些规律将帮助我们有效的避免大数求模中的溢出问题，关于大数求模，在后文中会有所涉及。

读者请仔细的理解和掌握这些运算规律。

本节旨在说明 % 运算符的运算特点，使读者了解其需要注意的行为，同时介绍了我们需要为其所做的额外工作。本节并没有涉及特定的算法知识，固未安排例题。但给出练习题，使读者熟悉 % 运算符的使用。

练习题：还是 $A+B$ (九度教程第 37 题)；守形数(九度教程第 38 题)；

二 数位拆解

在了解完 % 运算符后，利用其求余数的运算，我们来介绍与其相关的第一个应用——数位拆解。

数位拆解即把一个给定的数字（如 3241）各个数位上的数字拆开，即拆成 3、2、4、1。数位拆解频繁出现在各类算法设计教程的课后习题中，相信读者一定不会对其感到陌生。

例 4.1 特殊乘法（九度教程第 39 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

写个算法，对 2 个小于 1000000000 的输入，求结果。

特殊乘法举例： $123 * 45 = 1*4 + 1*5 + 2*4 + 2*5 + 3*4 + 3*5$

输入：

两个小于 1000000000 的数

输出：

输入可能有多组数据，对于每一组数据，输出 Input 中的两个数按照题目要求的方法进行运算后得到的结果。

样例输入：

123 45

样例输出：

该例虽然要求比较奇特,但我们不难发现它的核心任务仍是将两个整数各个数位上的数字拆开,然后再将这些数字两两相乘后求和得到答案。如样例所示,我们将整数 123 拆解成 1、2、3,将整数 45 拆解成 4、5,然后将他们两两相乘后相加就得到了答案 54。那么我们该如何来完成数位拆解呢?我们不妨来看以下过程:

我们不妨以一个四位数 x 为例,该四位数千位为 a ,百位为 b ,十位为 c ,个位为 d 。那么这些数字满足方程

$$x = a * 1000 + b * 100 + c * 10 + d;$$

读者如果已经完成了上一节的练习题,那么应该得到启发。只需确定 x 被 10 除的余数,我们就能确定其个位上的数字 d 。即

$$x \% 10 = (a * 1000 + b * 100 + c * 10 + d) \% 10;$$

$$x \% 10 = a * 1000 \% 10 + b * 100 \% 10 + c * 10 \% 10 + d \% 10;$$

$$x \% 10 = d;$$

那么当我们得到个位上的数字后,该如何继续获得其它数位上的数字呢?我们只需依次地将其它数位上的数字移动到个位,然后重复以上过程即可。而为了达到这一目的,我们利用整数除法,如下式:

$$x / 10 = (a * 1000 + b * 100 + c * 10 + d) / 10;$$

$$x / 10 = a * 1000 / 10 + b * 100 / 10 + c * 10 / 10 + d / 10;$$

$$x / 10 = a * 100 + b * 10 + c;$$

我们只需对 x 做整数除,使 x 除以整数 10,即可将十位上的数字移动到个位,百位上的数字移动到了十位数字,其它位依次类推。重复求个位数字的方法,我们即得到已经移动到个位的十位数字。不断地重复对 x 除以 10,对 10 求模,即可得到数字 x 各个数位上的数字,我们来看本例的解题代码,更详细的了解数位拆解的操作。

代码 4.1

```
#include <stdio.h>

int main () {
    int a , b; //保存两个整数的变量
    while (scanf ("%d%d", &a, &b) != EOF) { //输入两个整数
```

```

    int buf1[20] , buf2[20] , size1 = 0, size2 = 0; //用buf1, buf2分别保存
    从两个整数中拆解出来的数位数字, 其数量由size1, size2表示

    while(a != 0) { //数位拆解, 只要当a依然大于零就不断重复拆解过程
        buf1[size1++] = a % 10; //取得当前个位上的数字, 将其保存
        a /= 10; //将所有数位上的数字移动到高一位上
    }

    while(b != 0) { //拆解第二个数字
        buf2[size2++] = b % 10;
        b /= 10;
    }

    int ans = 0; //计算答案
    for (int i = 0; i < size1; i++)
        for (int j = 0; j < size2; j++)
            ans += buf1[i] * buf2[j]; //两两相乘后相加

    printf("%d\n", ans);
}

return 0;
}

```

通过以上原理性阐述和代码示范, 相信读者已经能够独立的完成数位分解了。假如一时半会儿不能完全理解它的原理, 那么记住分解的过程(求余求模不断重复)也是一种折中的办法。请记住我们拆分出来的数字, 是按照从低位到高位排列的顺序得出的, 即从个位开始依次向高位排列, 若考题对顺序有明确的要求(如从高位向低位输出)那么读者可自行转换。另外, 本例中题面限定了输入数据不会出现 0 的情况。但我们假设输入数据为 0, 那么当程序运行到 **while** 循环时, 由于被分解整数已经为 0, 程序将直接跳过循环, 而使数位分解失败, 即分解不到任何数字。为了避免这一情况, 我们可以在对整数做数位分解之前, 对该整数是否为 0 进行特判, 若其为 0, 则分解结果仅为一个数字 0; 否则, 进入 **while** 循环按原步骤分解。后文中也将介绍处理该特殊情况的其它技巧。

以上完成数位拆解的方法是从数学原理出发的, 旨在使读者对利用数学原理解决数学问题有一个理性的认识, 为接下去的学习作铺垫。而接下去我们将介绍另一种数位拆解的过程, 它将绕过数学原理, 而采用较为投机的方法。我们依旧以解决上例的代码为例。

代码 4.2

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

```

int main () {
    char a[11], b[11];

    while (scanf ("%s%s", a, b) != EOF) { //利用字符串将两个数字读入, 作为字符串保存在内存中

        int ans = 0; //累加变量

        for (int i = 0; a[i] != 0; i++) //遍历a中每一个字符, 直到a字符串结尾
            for (int j = 0; b[j] != 0; j++) //遍历b中每一个字符, 直到b字符串结尾
                ans += (a[i] - '0') * (b[j] - '0'); //计算a, b中每一个字符所代表的数字两两乘积的和

        printf("%d\n", ans); //输出答案
    }

    return 0;
}

```

如读者所见, 我们不再用整数变量来保存输入的数字, 再利用求模、求商等数学方法得到拆解后的数字。而是转而采用将输入数据当做字符串的技巧, 直接将两个数字以字符串的形式保存起来, 再依次遍历这个字符串, 通过字符与字符 '0' 的 ASCII 值的差, 计算字符所表示的数字值, 从而完成数字的拆解。这种方法, 虽然没有使用任何的数学技巧, 但它思路简洁、写法清晰, 与代码 3.1 相比, 代码量大大减少。在时间宝贵的机试当中, 适当的采用该技巧, 也是一种不错的方法。

练习题：反序数(九度教程第 40 题); 对称平方数(九度教程第 41 题); Digital Root(九度教程第 42 题);

三 进制转换

我们继续上一节的话题, 考虑一种较为特殊的数位拆解——进制转换。为什么说进制转换也是数位拆解的一种呢? 那是因为, 进制转换与数位拆解一样, 最终目的也是要求各个数位上的数字。不同的是, 数位拆解要求的是十进制表示的数字各个数位上的数字, 而进制转换求的则是以某种进制表示的数字用另一种进制来表示时各个数位上的数字。

要谈进制的转换, 首先我们要明确什么是进制, 以十进制为例, 十进制数字就是用满足下式的整数 $d_0, d_1, d_2 \dots d_n$, 来表示一个特定的数字。

$$x = d_0 * 10^0 + d_1 * 10^1 + d_2 * 10^2 + \dots + d_n * 10^n$$

在十进制中我们就用这组整数的依次排列来表示这个数字，即 $d_n \dots d_2 d_1 d_0$ ，我们知道该表示一定是惟一的。同时与各个数位上的数字相乘的 10 的各次幂被称为各个数位的权重，其按照从低位到高位每一位都是前一位的 10 倍。同理，用二进制表示整数 x ，即求数字组 $b_n \dots b_2 b_1 b_0$ 使满足如下等式：

$$x = b_0 * 2^0 + b_1 * 2^1 + b_2 * 2^2 + \dots + b_n * 2^n$$

而进制转换要完成的工作就是在各个进制表示特定数字的数字组之间相互的转换。

那么我们该如何完成进制转换呢？我们不妨把从 m 进制转换到 n 进制转化为两个进制转换问题：1. 从 m 进制转换到十进制。2. 再从十进制转换到 n 进制。这样，我们只需要掌握十进制与其它进制的相互转换就可以完成任意进制间的转换了。

我们以十进制转换为二进制为例，首先讨论如何完成十进制转换为其它进制。

我们有十进制数 x ，完成转换其为二进制，即求 $b_0, b_1, b_2 \dots b_n$ 。

$$x = b_0 * 2^0 + b_1 * 2^1 + b_2 * 2^2 + \dots + b_n * 2^n$$

使其满足如下等式：

我们应从上节数位拆解的方法中得到启发。首先我们对 x 模 2，即：

$$x \% 2 = (b_0 * 2^0 + b_1 * 2^1 + b_2 * 2^2 + \dots + b_n * 2^n) \% 2$$

$$x \% 2 = b_0 * 2^0 \% 2 + b_1 * 2^1 \% 2 + b_2 * 2^2 \% 2 + \dots + b_n * 2^n \% 2$$

$$x \% 2 = b_0$$

这样我们就得到了该数字由二进制表示时最低位数字。接下去读者应该能够反应过来，我们只需对 x 做整数除法，对其除 2，即可同样地将高位数字向低位移动，即

$$x / 2 = b_0 * 2^0 / 2 + b_1 * 2^1 / 2 + b_2 * 2^2 / 2 + \dots + b_n * 2^n / 2$$

$$x / 2 = b_1 * 2^0 + b_2 * 2^1 + \dots + b_n * 2^{n-1}$$

再通过求模运算依次求得被移动到最低位上的数字，如此往复，直到得到所有数位上的数字。

反过来将二进制表示的数字转换为十进制则比较容易，我们只要依次计算各个数位上的数字与该位权重的乘积再将它们求和，即可得到十进制数字。例如有

二进制数字 1010，我们只需计算下式：

$$0 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3$$

即可求出该二进制数字表示的十进制数。

总结一下，当要求十进制数 x 的 k 进制表示时，我们只需不断的重复对 x 求余（对 k ），求商（除以 k ），即可由低到高依次得到各个数位上的数。反过来，要求得由 k 进制表示的数字的十进制值时，我们需要依次计算各个数位上的数字与该位权重的积（第 n 位则权重为 k^{n-1} ），然后将它们依次累加即可得到该十进制值。

例 4.2 又一版 A+B （九度教程第 43 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

输入两个不超过整型定义的非负 10 进制整数 A 和 B ($\leq 231-1$)，输出 $A+B$ 的 m ($1 < m < 10$) 进制数。

输入：

输入格式：测试输入包含若干测试用例。每个测试用例占一行，给出 m 和 A ， B 的值。

当 m 为 0 时输入结束。

输出：

输出格式：每个测试用例的输出占一行，输出 $A+B$ 的 m 进制数。

样例输入：

8 1300 48

2 1 7

0

样例输出：

2504

1000

来源：

2008 年浙江大学计算机及软件工程研究生机试真题

该例即为最普通的进制转换，我们只需求得两数的和并将其转换为 m 进制输出即可。我们通过对该例代码的学习，来进一步了解进制转换。

代码 4.3

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

```

int main () {
    long long a , b; //使用数据类型long long确保不会溢出
    int m;
    while (scanf ("%d",&m) != EOF) {
        if (m == 0) break; //当m等于0时退出
        scanf ("%lld%lld", &a, &b); //用%lld对long long变量赋值
        a = a + b; //计算a+b
        int ans[50], size = 0; //ans用来保存依次转换得到的各个数位数字的值, size表示
其个数
        do { //依次求的各个数位上的数字值
            ans[size ++] = a % m; //对m求模
            a /= m; //除以m
        } while (a != 0); //当a不为0时重复该过程
        for (int i = size - 1; i >= 0; i --) {
            printf("%d", ans[i]);
        } //输出, 注意顺序为从高位到低位
        printf("\n"); //输出换行
    }
    return 0;
}

```

该代码中使用了我们前文未曾使用的数据类型 `long long`（部分平台 `__int64`），这是一种用64位二进制来表示一个整数的数据类型，它的数字取值范围为 $-2^{63} \sim 2^{63}-1$ 。在本例中，虽然题面明确了输入数据将在 `int` 范围内（ $\leq 2^{31}-1$ ），但是两个 `int` 数字的和可能超过 `int` 所能表示的最大值，出现溢出。为了避免这种情况，我们采用 `long long` 来表示两个 `int` 数的和（代码中两个数字也用 `long long` 保存）。读者应该对溢出问题保持相当高的警觉，在题面明确的输入范围内，你所写的代码是否会产生溢出？若会，则应该采取相应措施。另外顺便一提的是，如果用 `scanf` 和 `printf` 来输出 `long long`，则使用转义字符 `%lld`（`__int64` 对应 `%I64d`）。

另一个与上例不同的地方是进制转换处我们用 `do while` 循环来代替上例数位拆解中的 `while` 循环，这样做的目的是保证该转换工作至少会被执行一次，那么即使被转换数字是0，程序也能正常工作。当然，我们也可以选择如上一节中所讲的方法，依然采用 `while` 循环，但在开始转换前，判断被转换数字是否为0，若是则做相应特殊处理工作。

我们再以一个例题来熟悉其它进制与十进制的相互转换：

例 4.3 数制转换 （九度教程第 44 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

求任意两个不同进制非负整数的转换（2 进制~16 进制），所给整数在 long 所能表达的范围之内。不同进制的表示符号为（0, 1, ..., 9, a, b, ..., f）或者（0, 1, ..., 9, A, B, ..., F）。

输入：

输入只有一行，包含三个整数 a, n, b。a 表示其后的 n 是 a 进制整数，b 表示欲将 a 进制整数 n 转换成 b 进制整数。a, b 是十进制整数， $2 \leq a, b \leq 16$ 。

输出：

可能有多组测试数据，对于每组数据，输出包含一行，该行有一个整数为转换后的 b 进制数。输出时字母符号全部用大写表示，即（0, 1, ..., 9, A, B, ..., F）。

样例输入：

15 Aab3 7

样例输出：

210306

提示：

可以用字符串表示不同进制的整数

来源：

2008 年北京大学图形实验室计算机研究生机试真题

为了完成两个进制之间的转换，我们如上文所说将这个转换分为两步：首先将 a 进制转换为十进制，再将得到的十进制数转换为 b 进制。

代码 4.4

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

int main () {
    int a , b;
    char str[40];
    while (scanf ("%d%s%d", &a, str, &b) != EOF) {
        int tmp = 0, len = strlen(str), c = 1; //tmp为我们将要计算的a进制对应的十进制数, len为字符串长度方便我们从低位到高位遍历每个数位上的数, c为各个数位的权重初始化为1, 表示最低位数权重为1, 之后每位权重都是前一位权重的a倍
```



```

for (int i = lenth - 1; i >= 0; i --) { //从低位到高位遍历每个数位上的数
    int x; //计算该位上数字
    if (str[i] >= '0' && str[i] <= '9') {
        x = str[i] - '0'; //当字符在0到9之间, 计算其代表的数字
    }
    else if (str[i] >= 'a' && str[i] <= 'z') {
        x = str[i] - 'a' + 10; //当字符为小写字母时, 计算其代表的数字
    }
    else {
        x = str[i] - 'A' + 10; //当字符为大写字母时, 计算其代表的数字
    }
    tmp += x * c; //累加该位数字与该数位权重的积
    c *= a; //计算下一位数位权重
}

char ans[40] , size = 0; //用ans保存转换到b进制的各个数位数字
do {
    int x = tmp % b; //计算该位数字
    ans[size ++] = (x < 10) ? x + '0' : x - 10 + 'A'; //将数字转换为
字符

    tmp /= b;
} while(tmp);
for (int i = size - 1; i >= 0; i --) {
    printf("%c", ans[i]);
}
printf("\n"); //输出
}
return 0;
}

```

我们按照如下过程完成其它进制向十进制的转换：利用一个变量c依次计算每个数位权重，它的初始值为1，每经过一位就累乘进制数a，使表示权重的变量c依次等于1、a、a²、a³....；从低位到高位依次遍历各个数位上的数字，同时将其与当前位的权重（即变量c）相乘；最后依次累加所得到的积，即可得到由十进制表示的数字。

通过本例的学习，我们不仅能回顾将十进制数转换到其它进制，同时也学习

了如何将其它进制转换到十进制。若读者能够独立完成本例的解题代码，相信你已经很好的掌握了本节的内容。

练习题：进制转换(九度教程第 45 题)；八进制(九度教程第 46 题)；

四 最大公约数（GCD）

前几节中我们讨论了数位拆解与进制转换，本节我们继而介绍最大公约数的求法。

所谓求整数 a 、 b 的最大公约数，就是求同时满足 $a\%c=0$ 、 $b\%c=0$ 的最大正整数 c ，即求能够同时整除 a 和 b 的最大正整数 c 。在介绍欧几里得算法之前，读者可能会有这样的思路：若 a 、 b 均不为 0，则依次遍历不大于 a （或 b ）的所有正整数，依次试验它是否同时满足两式，并在所有满足两式的正整数中挑选最大的那个即是所求；若 a 、 b 其中有一个为 0，那么最大公约数即为 a 、 b 中非零的那个；若 a 、 b 均为 0，则最大公约数不存在（任意数均可同时整除它们）。要说明的是这个朴素的思路是完全正确的，它的确能够正确的求得两个数的最大公约数。但是，该解法在大部分情况下要遍历不大于 a （或 b ）的所有正整数，并依次测试它们是否满足条件，当 a 和 b 数值较大时（如 10000000）该算法的时间复杂度较高，耗费的时间较多，往往不能在指定时间内得到结果。我们试着寻找一种更加高效的方法来求解最大公约数，首先我们来看如下的证明过程：

若整数 g 为 a 、 b （不同时为 0）的公约数，则 g 满足：

$$a = g * l;$$

$$b = g * m;$$

其中 l 、 m 为整数。同时 a 又可由 b 表示为下式：

$$a = b * k + r;$$

其中 k 为整数， r 为 a 除以 b 后的余数。那么对如上三式做如下变形：

$$g * l = g * m * k + r;$$

$$r = g * (l - m * k); (g \neq 0)$$

由上式可知， a 、 b 的公约数可以整除 a 除以 b 剩余的余数（记为 $a \bmod b$ ）。即， a 、 b 的公约数同时也必是 b 、 $a \bmod b$ 的公约数。

那么若 g 是 a 、 b 的最大公约数，它同样也是 b 、 $a \bmod b$ 的最大公约数吗？

我们假设 g 是 a 、 b 的最大公约数，但它并不是 b 、 $a \bmod b$ 的最大公约数，

即存在 $g' > g$ 且 g' 同时整除 b 与 $a \bmod b$ 。这样，必存在整数 l' 与 m' 使下式成立：

$$b = g' * m';$$

$$r = a \bmod b = g' * l';$$

同时 a 、 b 、 r 之间满足下式：

$$a = b * k + r;$$

$$a = g' * m' * k + g' * l';$$

$$a = g' * (m' * k + l'); (g' \neq 0)$$

如上式， g' 同时也整除 a ，那么 g' 同时也是 a 、 b 的公约数。但是假设中， a 、 b 的最大公约数为 g ，而 $g' > g$ 与假设不符，所以可证明 a 、 b 的最大公约数同时也是 b 、 $a \bmod b$ 的最大公约数。

这样，我们把求 a 、 b 的最大公约数转换成了求 b 、 $a \bmod b$ 的最大公约数，那么问题不变而数据规模则明显变小，我们可以不断重复该过程，直到问题缩小成求某个非零数与零的最大公约数（该情况一定会发生，证明略）。这样，该非零数即是所求。

我们来整理一下以上过程：

若 a 、 b 全为零则它们的最大公约数不存在；若 a 、 b 其中之一为零，则它们的最大公约数为 a 、 b 中非零的那个；若 a 、 b 都不为零，则使新 $a = b$ ；新 $b = a \% b$ 然后重复该过程。

这就是我们要介绍的欧几里得算法，它改变了上文所提到的朴素的枚举算法需要暴力遍历所有数字的情况，而改为利用数学原理巧妙的将问题转换为规模更小的问题，从而最后得出答案。

例 4.4 最大公约数（九度教程第 47 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

输入两个正整数，求其最大公约数。

输入：

测试数据有多组，每组输入两个正整数。

输出：

对于每组输入,请输出其最大公约数。

样例输入：

49 14

样例输出:

7

来源:

2011 年哈尔滨工业大学计算机研究生机试真题

该例就是曾经出现在研究生考试复试机试中的关于求最大公约数的真题,我们通过给出该题解题代码来详细地向读者介绍欧几里得算法。

代码 4.5

```
#include <stdio.h>

int gcd(int a,int b) {
    if (b == 0) return a; //若b为零则最大公约数为a
    else return gcd(b, a % b); //否则, 则改为求b与a%b的最大公约数
}

int main () {
    int a , b;
    while (scanf ("%d%d",&a, &b) != EOF) { //输入两个正整数
        printf("%d\n", gcd(a, b)); //输出所求的最大公约数
    }
    return 0;
}
```

该代码把求最大公约数的欧几里得算法写成了递归的形式,若读者对理解递归有困难,这里我们也给出其非递归形式,供读者参考。

代码 3.6

```
#include <stdio.h>

int gcd(int a,int b) {
    while(b != 0) { //只要b不为0则一直持续该过程
        int t = a % b;
        a = b; //使a变成b
        b = t; //使b变成a % b
    }
    return a; //当b为0时, a即是所求
}

int main () {
    int a , b;
    while (scanf ("%d%d",&a, &b) != EOF) {
```

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

读者可任意选择递归或者非递归形式的欧几里得算法理解并记忆，这样我们就已经掌握了求解最大公约数的方法。

五 最小公倍数（LCM）

我们在上一节结束时并没有按照惯例给出相应的例题，那是因为本节将要讨论的求最小公倍数与求最大公约数有着一定的联系，所以，我们把相关练习统一放到本节。

首先，我们确定什么叫作最小公倍数。求 a 、 b 的最小公倍数，即求最小正整数 c ，使满足 $c \% a = 0$ 且 $c \% b = 0$ 。那么它与求最大公约数又有什么关系呢？

我们指出 a 、 b 两数的最小公倍数为两数的乘积除以它们的最大公约数。其实这个结论并不难证明，首先明确 $k = a * b$ 一定是 a 、 b 的一个公倍数，那么最小公倍数一定不大于 k ，那么若有 a 、 b 的公约数 c ，则有：

$$k = a * b;$$

$$k / c = a * b / c;$$

$$k / c = a * (b / c);$$

$$k / c = b * (a / c);$$

其中 b/c 、 a/c 均为整数，即 k/c 同时为 a 、 b 的倍数；反之，若 c 不为 a 、 b 的公约数，则 b/c 与 a/c 至少有一个不为整数，则 k/c 不再是 a 、 b 的公倍数。显然，我们要取得最小的公倍数，即需找到最大的公约数 c 使 k/c 最小，该 k/c 就是我们要求的最小公倍数。

这样我们就把求最小公倍数问题统一到了求最大公约数上来。

例 4.5 最小公倍数（九度教程第 48 题）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

给定两个正整数，计算这两个数的最小公倍数。

输入：

输入包含多组测试数据，每组只有一行，包括两个不大于 1000 的正整数。

输出:

对于每个测试用例, 给出这两个数的最小公倍数, 每个实例输出一行。

样例输入:

10 14

样例输出:

70

我们按照如上所说的方法求两个数的最小公倍数。

代码 4.7

```
#include <stdio.h>

int gcd(int a, int b) { //求最大公约数
    return b != 0 ? gcd(b, a % b) : a;
}

int main () {
    int a , b;
    while (scanf ("%d%d", &a, &b) != EOF) {
        printf("%d\n", a * b / gcd(a, b)); //输出两数乘积与最大公约数的商
    }
    return 0;
}
```

如代码所示, 求最小公倍数非常简单, 只需求得最大公约数后稍作处理即可。

以上两节主要讲述了求最大公约数和最小公倍数的方法, 并给出了求最大公约数的欧几里得算法和相关数学证明。在本节的最后, 我们给出练习, 使读者能够自己动手求解最大公约数与最小公倍数问题。

练习题: Least Common Multiple(九度教程第 49 题);

六 素数筛法

本节将涉及数学问题中另一类十分常见的问题: 确定素数 (又叫质数)。素数即只能被自身和 1 整除的大于 1 的正整数。本节所要讲述的内容与其有关。

怎样确定一个数是素数? 我们可以用所有大于 1 小于其本身的整数去试着整除该数, 若在该区间内存在某个数能整除该数则该数不是素数; 若这些数都不能整除它, 则该数为素数。这一朴素的算法思想时间复杂度为 $O(n)$, n 为我们测试的数字。但其实, 我们并不用测试到 $n-1$ 为止, 我们只需测试到不比 \sqrt{n}

(n) (对 n 开根号) 大的整数即可, 若到这个整数为止, 所有正整数均不能整除 n , 则可以断定, n 为素数。若 n 不存在大于 $\text{sqrt}(n)$ 的因数时, 该做法显然正确。若我们假设 n 存在大于等于 $\text{sqrt}(n)$ 的因数 y , 则 $z = n/y$ 必同时为 n 的因数, 且其值小于等于 $\text{sqrt}(n)$ (否则 $z * y > n$)。所以, 若 n 存在相异于 1 与其本身的因数且该因数大于 $\text{sqrt}(n)$, 则必存在小于或等于 $\text{sqrt}(n)$ 的因数, 所以我们只需测试到 $\text{sqrt}(n)$ 为止。这样测试一个数是否是素数的复杂度就降低到了 $O(\text{sqrt}(n))$ 。

例 4.6 素数判定 (九度教程第 50 题)

时间限制: 1 秒

内存限制: 32 兆

特殊判题: 否

题目描述:

给定一个数 n , 要求判断其是否为素数 (0,1, 负数都是非素数)。

输入:

测试数据有多组, 每组输入一个数 n 。

输出:

对于每组输入, 若是素数则输出 yes, 否则输入 no。

样例输入:

13

样例输出:

yes

来源:

2009 年哈尔滨工业大学计算机研究生机试真题

我们只需按照上文所说的, 对每个输入的值依次测试大于 1 但不大于其平方根的数字能否整除它即可。

代码 4.8

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

bool judge(int x) { //判断一个数是否为素数
    if (x <= 1) return false; //若其小于等于1, 必不是
    int bound = (int)sqrt(x) + 1; //计算枚举上界, 为防止double值带来的精度损失,
    所以采用根号值取整后再加1, 即宁愿多枚举一个数也不能少枚举一个数
    for (int i = 2; i < bound; i++) {
        if (x % i == 0) return false; //依次枚举这些数能否整除x, 若能则必不为素数
    }
}
```



```

    return true; //若均不能则为素数
}

int main () {
    int x;
    while (scanf ("%d", &x) != EOF) {
        puts(judge(x) ? "yes" : "no"); //依据函数返回值输出答案
    }
    return 0;
}

```

该代码比较简单，但不知道读者是否注意到其中的一个小技巧。我们采用了先计算出枚举上界，将它赋值给 `bound`，再在 `for` 循环循环条件中与 `bound` 作比较的写法，而不采用在 `for` 循环循环条件中直接与 `sqrt(n) + 1` 进行比较。这是有原因的。我们的写法，将 `sqrt(n) + 1` 的值赋值给变量 `bound`，然后令 `i` 与 `bound` 作比较，这样做保证了 `sqrt` 运算只进行一次。而假如直接在 `for` 循环循环条件中与 `sqrt(n) + 1` 作比较，则比较多少次，`sqrt(n)` 也会运算多少次，而 `sqrt` 是众所周知的几个比较耗时的函数之一，我们采用这样的编码技巧为程序节省了不少时间（该策略同样适用于 `strlen` 函数）。

我们已经学习了，如何判断一个数是否是素数，那么我们该如何找出 0 到 1000000 中所有的素数呢？依次枚举每个数，然后按照上文中判断某个数是否是素数的方法确定其是否为素数，直到得出该区间中所有的素数？当然，这样做是可行的，但是该方法时间复杂度过高，且整个过程显得粗暴而不具有技巧性。这里，我们提出一种更优雅的方法解决该问题。

我们首先来考虑这样一个命题：若一个数不是素数，则必存在一个小于它的素数为其的因数。这个命题的正确性是显而易见的。那么，假如我们已经获得了小于一个数的所有素数，我们只需确定该数不能被这些素数整除，这个数即为素数。但是这样的做法似乎依然需要大量的枚举测试工作。正因为如此，我们可以换一个角度，在我们获得一个素数时，即将它的所有倍数均标记成非素数，这样当我们遍历到一个数时，它没有被任何小于它的素数标记为非素数，则我们确定其为素数。我们按照如下步骤完成工作：

从 2 开始遍历 2 到 1000000 的所有整数，若当前整数没有因为它是某个小于它的素数的倍数而被标记成非素数，则判定其为素数，并标记它所有的倍数为非素数。然后继续遍历下一个数，直到遍历完 2 到 1000000 区间内所有的整数。此时，所有没被标记成非素数的数字即为我们要求的素数。这种算法被我们称为素数筛法。

例 4.7 素数（九度教程第 51 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

输入一个整数 $n(2 \leq n \leq 10000)$ ，要求输出所有从 1 到这个整数之间(不包括 1 和这个整数)个位为 1 的素数，如果没有则输出-1。

输入：

输入有多组数据。

每组一行，输入 n 。

输出：

输出所有从 1 到这个整数之间(不包括 1 和这个整数)个位为 1 的素数(素数之间用空格隔开，最后一个素数后面没有空格)，如果没有则输出-1。

样例输入：

100

样例输出：

11 31 41 61 71

来源：

2008 年北京航空航天大学计算机研究生机试真题

我们以该题为例，对素数筛法做相关的示范。

代码 4.9

```
#include <stdio.h>

int prime[10000]; //保存筛得的素数
int primeSize; //保存的素数的个数
bool mark[10001]; //若mark[x]为true,则表示该数x已被标记成非素数
void init() { //素数筛法
    for (int i = 1; i <= 10000; i++) {
        mark[i] = false;
    } //初始化，所有数字均没被标记
    primeSize = 0; //得到的素数个数为0
    for (int i = 2; i <= 10000; i++) { //依次遍历2到10000所有数字
        if (mark[i] == true) continue; //若该数字已经被标记,则跳过
        prime[primeSize++] = i; //否则,又新得到一个素数
        for (int j = i * i; j <= 10000; j += i) { //并将该数的所有倍数均标记成非
```

素数

```
        mark[j] = true;
    }
}
}

int main () {
    init(); //在程序一开始首先取得2到10000中所有素数
    int n;
    while (scanf ("%d", &n) != EOF) {
        bool isOutput = false; //表示是否输出了符合条件的数字
        for (int i = 0; i < primeSize; i++) { //依次遍历得到的所有素数
            if (prime[i] < n && prime[i] % 10 == 1) { //测试当前素数是否符合条件
                if (isOutput == false) { //若当前输出为第一个输出的数字, 则标记已经输出了符合条件的数字, 且该数字前不输出空格
                    isOutput = true;
                    printf("%d", prime[i]);
                }
                else printf(" %d", prime[i]); //否则在输出这个数字前输出一个空格
            }
        }
        if (isOutput == false) { //若始终不存在符合条件的数字
            printf("-1\n"); //输出-1并换行
        }
        else printf("\n"); //换行
    }
    return 0;
}
```

我们利用素数筛法, 在处理输入的数字前, 先处理出 2 到 1000000 区间内所有素数。当输入 n 时, 则依次比较已经得到的素数是否符合条件, 若符合则输出, 否则继续比较下一个素数。

读者可能注意到, 筛法中我们使用了一个小技巧。当我们判定 i 为素数, 要标记其所有倍数为非素数时, 我们并没有从 $2 * i$ 开始标记, 而是直接从 $i * i$ 开始标记。其原因是显然的, $i * k$ ($k < i$) 必已经在求得 k 的某个素因数 (必小

于 i 时被标记过了，即 $i*k$ 同时也是 k 的素因数的倍数。所以这里，我们可以直接从 i 的平方开始标记起。尽可能的避免重复工作也是程序优化的一大思路。

本例中，我们依旧使用了，一个 `bool` 变量表示是否已经输出了符合条件的数字。这样的设置有两个目的，其一保证了除了第一个输出的数字外，其它数字输出时均在其前附加一个空格，以达到题目要求的输出数字之间存在空格而最后一个数字后没有空格的要求。其二，作为判断依据，使在不存在任何符合条件的数可以输出时，按题目要求输出 -1。

另外素数筛法常与在本例中一样，作为程序真正开始处理输入数据前的预处理使用，即预先处理出相关区间内的所有素数，以备后续工作的使用。

练习题：Prime Number(九度教程第 52 题); Goldbach's Conjecture(九度教程第 53 题);

七 分解素因数

本节继续讨论有关素数的问题，我们将要了解对一个数分解素因数。顾名思义，对一个数 x 分解素因数即确定素数 p_1, p_2, \dots, p_n ，使其满足下式：

$$x = p_1^{e_1} * p_2^{e_2} * \dots * p_n^{e_n}$$

必要时，我们还要确定 e_1, e_2 等幂指数。

我们先来看如下一例：

例 4.8 质因数的个数（九度教程第 54 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

求正整数 $N(N>1)$ 的质因数的个数。

相同的质因数需要重复计算。如 $120=2*2*2*3*5$ ，共有 5 个质因数。

输入：

可能有多组测试数据，每组测试数据的输入是一个正整数 N ， $(1<N<10^9)$ 。

输出：

对于每组数据，输出 N 的质因数的个数。

样例输入：

120

样例输出：

5

提示:

注意: 1 不是 N 的质因数; 若 N 为质数, N 是 N 的质因数。

来源:

2007 年清华大学计算机研究生机试真题

本例题意清晰, 即对输入的某个整数分解素因数, 并计算出每个素因数所对应的幂指数, 即对给定整数 x , 确定下式中

$$x = p_1^{e_1} * p_2^{e_2} * \dots * p_n^{e_n}$$

p_1, p_2, \dots, p_n 与 e_1, e_2, \dots, e_n 的取值。这样, 我们即能得到最后的结果为 e_1, e_2, \dots, e_n 的和。

首先我们按照如下思路来为整数分解素因数: 我们利用上节内容中所讲的素数筛法预先筛选出所有可能在题面所给定的数据范围内成为素因数的素数。并在程序输入待处理数字 n 时, 依次遍历所有小于 n 的素数, 判断其是否为 n 的因数。若确定某素数为 n 的因数, 则通过试除确定其对应的幂指数。最后求出各个幂指数的和即为所求。

代码 4.10

```
#include <stdio.h>
bool mark[100001];
int prime[100001];
int primeSize;
void init() {
    primeSize = 0;
    for (int i = 2; i <= 100000; i++) {
        if (mark[i] == true) continue;
        prime[primeSize++] = i;
        if (i >= 1000) continue;
        for (int j = i * i; j <= 100000; j += i) {
            mark[j] = true;
        }
    }
} //以上与上例一致, 用素数筛法筛选出2到100000内的所有素数
int main () {
    init();
    int n;
```

```

while (scanf ("%d", &n) != EOF) {
    int ansPrime[30]; //按顺序保存分解出的素因数
    int ansSize = 0; //分解出素因数的个数
    int ansNum[30]; //保存分解出的素因数对应的幂指数
    for (int i = 0; i < primeSize; i++) { //依次测试每一个素数
        if (n % prime[i] == 0) { //若该素数能整除被分解数
            ansPrime[ansSize] = prime[i]; //则该素数为其素因数
            ansNum[ansSize] = 0; //初始化幂指数为0
            while(n % prime[i] == 0) { //从被测试数中将该素数分解出来, 并统计其幂指数
                ansNum[ansSize]++;
                n /= prime[i];
            }
            ansSize++; //素因数个数增加
            if (n == 1) break; //若已被分解成1, 则分解提前终止
        }
    }
    if (n != 1) { //若测试完2到100000内所有素因数, n仍未被分解至1, 则剩余的因数一定是n一个大于100000的素因数
        ansPrime[ansSize] = n; //记录该大素因数
        ansNum[ansSize++] = 1; //其幂指数只能为1
    }
    int ans = 0;
    for (int i = 0; i < ansSize; i++) {
        ans += ansNum[i]; //统计各个素因数的幂指数
    }
    printf("%d\n", ans); //输出
}
return 0;
}

```

该代码按照如下步骤对输入的整数分解素因数：

- 1.利用素数筛法筛得 0 到 100000 区间内所有素数。
- 2.输入 n。
- 3.依次测试步骤 1 中得到的素数能否整除 n，若能则表明该素数为它的一个

素因数。

4.不断将 n 除以该素数，直到不能再被整除为止，同时统计其幂指数。

5.若在完成某个素数的幂指数统计后， n 变为 1，则表明 n 的所有素因数全部被分解出来，这样就不用再去遍历后续的素数，分解活动提前终止。

6.若遍历、测试、分解完所有预处理出来的素数， n 仍旧没被除成 1，则表明 n 存在一个大于 100000 的因子，且该因子必为其素因子，且其幂指数必然为 1。

我们首先说明为什么素数筛法只需筛到 100000 即可，而不是与输入数据同规模的 1000000000。这样处理的理论依据是： n 至多只存在一个大于 \sqrt{n} 的素因数（否则两个大于 \sqrt{n} 的数相乘即大于 n ）。这样，我们只需将 n 所有小于 \sqrt{n} 的素数从 n 中除去，剩余的部分必为该大素因数。正是由于这样的原因，我们不必依次测试 \sqrt{n} 到 n 的素数，而是在处理完小于 \sqrt{n} 的素因数时，就能确定是否存在该大素因数，若存在其幂指数也必为 1。

在完成查找素因数的工作以后，我们只需简单的把所有素因数对应的幂指数相加，即可得到该整数素因数的个数。顺便一提的是，当我们完成素因数分解后我们同样可以确定被分解整数因数的个数为 $(e_1+1)*(e_2+1)*\dots*(e_n+1)$ （由所有的素因数不同组合数得出）。

在了解了以上基础知识后，我们来看下面这非常有意思的一例。

例 4.9 整除问题（九度教程第 55 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

给定 n ， a 求最大的 k ，使 $n!$ 可以被 a^k 整除但不能被 $a^{(k+1)}$ 整除。

输入：

两个整数 $n(2 \leq n \leq 1000)$ ， $a(2 \leq a \leq 1000)$

输出：

一个整数。

样例输入：

6 10

样例输出：

1

来源：

2011 年上海交通大学计算机研究生机试真题

要解决该例我们首先应注意这样一个问题， $n!$ 和 a 的 k 次可能数值非常巨

大，而不能被 int（甚至 long long）保存，也就不能直接用求余数操作判断它们是否存在整除关系。那么，我们不得不从整除的特征入手，转而思考若整数 a 能整除整数 b 则它们之间有什么关系？我们不妨对 a 和 b 分解素因数：

$$a = p_1^{e_1} p_2^{e_2} \dots p_n^{e_n};$$

$$b = p_1^{e_1'} p_2^{e_2'} \dots p_n^{e_n'};$$

则，式 b 除以 a 能表示成：

$$\frac{b}{a} = \frac{p_1^{e_1'} * p_2^{e_2'} * \dots * p_n^{e_n'}}{p_1^{e_1} * p_2^{e_2} * \dots * p_n^{e_n}};$$

若 a 能整除 b，则该式为一个整数，但考虑到若素数 p1 能够整除素数 p2，则 p1 必等于 p2（两个素数必互质）。则我们可以得出如下规律：

若 a 存在素因数 p_x 则 b 也必存在该素因数，且该素因数在 b 中对应的幂指数必不小于在 a 中的幂指数。

现我们设 $x = n!$ ， $y = a^k$ ，我们对 n! 与 a 分解素因数，令：

$$x = p_1^{e_1} p_2^{e_2} \dots p_n^{e_n};$$

$$a = p_1^{e_1'} p_2^{e_2'} \dots p_n^{e_n'};$$

相应的我们也可以得到 a 的 k 次的素因数分解情况为：

$$a^k = (p_1^{e_1'} p_2^{e_2'} \dots p_n^{e_n'})^k;$$

$$a^k = p_1^{e_1' * k} p_2^{e_2' * k} \dots p_n^{e_n' * k};$$

即我们要确定最大的非负整数 k，使 a 中任一素因数的幂指数的 k 倍依旧小于或等于该素因数在 x 中对应的幂指数。要求得该 k，我们只需依次测试 a 中每一个素因数，确定 b 中该素因数对应的幂指数是 a 中幂指数的几倍（利用整数除法），这样所有倍数中最小的那个即为我们要求的 k。

分析到这里，剩余的工作似乎只剩下对 a 和 n! 分解素因数，对 a 分解素因数我们在前文中已经探讨过了。那么多 n! 呢？千万不要指望将 n! 计算出来后再类似 a 一样对其分解质因数，由于 n! 数值非常巨大（当 n>30 时），想要这样操作几乎是不可能的，那么我们该如何对其分解素因数呢？试着考虑 n! 中含有素因数 p 的个数，即确定素因数 p 对应的幂指数。我们易知，n! 中包含了 1 到

$n!$ 区间内所有整数的乘积，这些乘积中每一个 p 的倍数（包括其本身）都将对 $n!$ 贡献至少一个 p 因子，且我们知道在 1 到 n 中 p 的倍数共有 n/p (整数除法) 个，则 p 的因子数至少为 n/p 个，即有 n/p 个整数至少贡献了一个 p 因子。那么有多少个整数将贡献至少两个 p 因子呢，有了以上的分析读者应该知道所有 $p*p$ 的倍数将为 $n!$ 贡献至少 2 个 p 因子，且这样的整数有 $n/(p*p)$ ；同理 $p*p*p$ 的倍数将贡献至少 3 个，这样的数有 $n/(p*p*p)$ ； p 的四次方的倍数将贡献至少 4 个，这样的数由 $(n/(p*p*p*p))$ 。那么分析出这些结果对我们分解 $n!$ 的质因数有什么帮助呢，且看如下过程。

1. 计算器清零，该计数器表示 $n!$ 中将有几个 p 因子，即 $n!$ 分解质因数后素因子 p 对应的幂指数。

2. 计算 n/p ，有 n/p 个整数可以向 $n!$ 提供一个 p 因子，则计数器累加 n/p 。若 n/p 为 0，表示没有一个整数能向 $n!$ 提供一个或一个以上的 p 因子，分解结束。

3. 计算 $n/(p*p)$ ，有 $n/(p*p)$ 个整数可以向 $n!$ 提供两个 p 因子，但它们在之前步骤中 (p 的倍数必包括 $p*p$ 的倍数) 每个数都已经向计数器累加了 1 个 p 因子，所以此处他们还能够向计数器贡献 $n/(p*p)$ 个素因子 (即每个再贡献一个)，累加器累加 $n/(p*p)$ 。若 $n/(p*p)$ 为 0，表示没有一个整数能向 $n!$ 提供两个或两个以上的 p 因子，分解结束。

4. 计算 $n/(p*p*p)$ ，有 $n/(p*p*p)$ 个整数可以向 $n!$ 提供三个 p 因子，但它们在之前步骤中 (p 和 $p*p$ 的倍数必包括 $p*p*p$ 的倍数) 每个数都已经计算向计数器累加了 2 个 p 因子，所以此处他们还能够向计数器贡献 $n/(p*p*p)$ 个素因子，累加器累加 $n/(p*p*p)$ 。若 $n/(p*p*p)$ 为 0，表示没有一个整数能向 $n!$ 提供三个或三个以上的 p 因子，分解结束。

依次累加 p 的更高次的倍数能够再提供的素因子数，即每次向计数器累加 $n/(p^k)$ ，直到 $n/(p^k)$ 变为 0，表示没有整数能提供更多的 p 因子，关于 p 的分解结束。

完成这些步骤后，就能计算出 $n!$ 中所有 p 的因子数，即计数器中累加的结果即为素因数 p 的幂指数。

有了对 $n!$ 分解素因数的方法，我们只需依次遍历可能成为其素因子 (小于等于 n 的所有素数) 的素数，计算它们所对应的幂指数，即可完成对 $n!$ 的素因数分解。

在完成对 a 和 $n!$ 的素因子分解后，我们按照如上诉讨论的方法完成此题。

代码 4.9

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

```

bool mark[1010];
int prime[1010];
int primeSize;
void init() {
    primeSize = 0;
    for (int i = 2; i <= 1000; i++) {
        if (mark[i]) continue;
        mark[i] = true;
        prime[primeSize++] = i;
        for (int j = i * i; j <= 1000; j += i) {
            mark[j] = true;
        }
    }
} //筛选出0到1000范围内的所有素数

int cnt[1010]; //cnt[i]用来表示, prime[i]所保存的素数在n! 中的因子数, 即n! 分解素
因数后, 素因子prime[i]所对应的幂指数, 可能为0

int cnt2[1010]; //cnt2[i]用来表示, prime[i]所保存的素数在a中的因子数

int main () {
    int n , a;
    init();
    while (scanf ("%d%d", &n, &a) == 2) {
        for (int i = 0; i < primeSize; i++)
            cnt[i] = cnt2[i] = 0; //将两个计数器清零, 为新的分解做准备
        for (int i = 0; i < primeSize; i++) { //对n! 分解素因数, 遍历每一个0到1000
            的素数

            int t = n; //用临时变量t保存n的值
            while (t) { //确定素数prime[i]在n中的因子数
                cnt[i] += t / prime[i];
                t = t / prime[i];
            } //依次计算  $t / \text{prime}[i]^k$ , 累加其值, 直到  $t / \text{prime}[i]^k$  变为0
        }

        int ans = 123123123; //答案初始值为一个大整数, 为取最小值做准备
        for (int i = 0; i < primeSize; i++) { //对a分解素因数
            while (a % prime[i] == 0) {

```

```

        cnt2[i]++;
        a /= prime[i];
    } //计算a中素因数prime[i]对应的幂指数
    if(cnt2[i] == 0) continue; //若该素数不能从a中分解到, 即其对应幂指数
    为0, 则其不影响整除性, 跳过

    if (cnt[i] / cnt2[i] < ans) //计算素数prime[i]在两个数中因子数的商
        ans = cnt[i] / cnt2[i]; //统计这些商的最小值
    }

    printf("%d\n", ans); //该商即为所求
}

return 0;
}

```

该题难度在机试题中已属于较难题(上交的机试题一向难度不小), 若读者能够完全的理解本例的解题原理并自主写出代码, 相信你已经具有了一定的实力, 至少考研机试中的数学问题已经很难再难住你了。

练习题: 约数的个数(九度教程第 56 题);

八 二分求幂

我们再来讨论一个非常实用的小技巧, 二分求幂, 即怎样快速的求得 a 的 b 次方。在读者之前的实践中, 涉及这个问题的程序很可能使用了一个循环次数为 b 的 `for` 循环, 并在每次循环时都累乘 a , 这样在 b 次循环结束时我们就将获得 a 的 b 次。如

```

int ans = 1;
for (int i = 1; i <= b; i++) {
    ans *= a;
}

```

那么该方法是否是最优的呢? 我们来看下面这种情况。

假如, 我们将要计算 2 的 32 次, 即 2^{32} 。我们是否需要真的循环 32 次呢? 按照我们采用的原始策略, 当我们循环到第 i 次时, 此时的累乘的结果即为 2 的 i 次, 即 2^i 。那么, 当我们完成了前 16 次循环时, 我们就已经获得了 2 的 16 次的数值, 要获得 2 的 32 次我们还需要继续完成后续的 16 次循环么? 答案是否定的, 当我们已经获得了 2 的 16 次时, 我们只需将 2 的 16 次对应的数字求平方, 我们即可计算出 2 的 32 次方, 这样后 16 次乘法运算我们用一次平方 (同样也是

乘法)就完成了。既然2的32次可以由2的16次求平方取得,那么2的16次呢?你还确定我们需要使用16次循环来获得该值么?答案也是否定的,2的16次只需对2的8次求平方即可,同理要求2的8次我们只需对2的4次求平方……这样依次往复,最后我们可得到如下计算过程:

$$\begin{aligned}2^1 &= 2; \\2^2 &= 2 * 2 = 8; \\2^4 &= 2^2 * 2^2 = 64; \\2^8 &= 2^4 * 2^4 = 256; \\2^{16} &= 2^8 * 2^8 = 65536; \\2^{32} &= 2^{16} * 2^{16} = 4294967296;\end{aligned}$$

这样,原本需要使用32次乘法才能完成的工作,现在只需要6次乘法便能完成,效率提高了将近6倍,这种求某个数的指定次幂的方法即我们在本节要介绍给大家的二分求幂。

读者可能注意到,2的32次具有某种特殊性,即32次刚好为2的5次方,所以其可以一直被二分到2的1次方为止,那么假如我们要求的次数不再具有这种特殊性,二分求幂还能适用吗?答案是肯定的,我们以求2的31次为例,继续了解二分求幂的工作特点:

$$\begin{aligned}2^{31} &= 2^1 * 2^{30}; \\&= 2^1 * 2^2 * 2^{28}; \\&\dots \\&= 2^1 * 2^2 * 2^4 * 2^8 * 2^{16};\end{aligned}$$

其中2的各次幂可以由之前讨论过的方法求得,即求得2的1次幂后,利用对其平方,求得2的2次幂,再对2的2次幂求平方即可求得2的4次幂,依次类推。当求得这些2的各次幂后,只需按要求对其累乘,即可得到答案。那么我们该如何确定哪些2的次幂是我们需要的,是我们要累乘的呢?首先,我们应该注意到从a的1次出发,a的2次,a的4次,a的8次,即a的 2^k 次是可以由a的1次不断求平方取得的。我们的目标即分解a的b次变为若干个a的 2^k 次

的积，并尽可能减少分解结果的个数。在指数层面即分解 b 为若干个 2^k 的和，并尽可能减少分解结果的个数。若读者认真完成了本章之前的内容，就应该能联想到分解 b 为若干个 2^k 的和且分解个数最小，这便是求 b 的二进制数。在求得 b 的二进制数后，各个二进制位为 1 的数位所代表的权重即是分解的结果。以 2 的 31 次方为例，我们首先求得 31 的二进制数 11111，在二进制表达式中 31 即被表达成 $(11111) = 2^0 + 2^1 + 2^2 + 2^3 + 2^4$ ，这就是我们所需的分解结果。即拆 2 的 31 次为 2 的 0 次、1 次、2 次、3 次、4 次的乘积，即得到前文中所讲的结果。

所以，二分求幂对要求的次数并没有特殊的要求，而是对任何要求的次数都可以采用二分求幂来大大减少其乘法运算的次数。我们通过下面这个例题，来了解其编码方式。

例 4.10 人见人爱 A^B (九度教程第 57 题)

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

求 A^B 的最后三位数表示的整数。说明： A^B 的含义是“A 的 B 次方”

输入：

输入数据包含多个测试实例，每个实例占一行，由两个正整数 A 和 B 组成 ($1 \leq A, B \leq 10000$)，如果 $A=0, B=0$ ，则表示输入数据的结束，不做处理。

输出：

对于每个测试实例，请输出 A^B 的最后三位表示的整数，每个输出占一行。

样例输入：

```
2 3
12 6
6789 10000
0 0
```

样例输出：

```
8
984
1
```

相信经过本章前几节内容的学习以及相应的练习，读者对求一个数的后三位数已经没有什么问题了。那么在本例中，我们先求得 A^B 的具体数字再求其后三位数么？这毫无疑问是不可行的，按照题面给明的输入规模， A^B 的次至多可以达到 10000 的 10000 次，这么庞大的数字是非常不容易保存的，但是我们应

该注意到 A^B 的后三位数只与 A 的后三位数和 B 有关。这样，由于要求的仅是最后结果的后三位数，那么我们在保存为计算该最终值的中间值时也只需保存其后三位数即可。即，计算过程中的所有中间结果我们仅保存和使用其后三位数。那么，我们再利用二分求幂来求得 A^B 次，同时在计算中间结果时我们都仅保存后三位数。这样，我们就不用担心数字不能被保存的问题了。

代码 4.10

```
#include <stdio.h>

int main () {
    int a , b;
    while (scanf ("%d%d", &a, &b) != EOF) {
        if (a == 0 && b == 0) break;
        int ans = 1; //保存最终结果变量, 初始值为1
        while(b != 0) { //若b不为0，即对b转换二进制过程未结束
            if (b % 2 == 1) { //若当前二进制位为1，则需要累乘a的 $2^k$ 次至变量ans，
            其中 $2^k$ 次为当前二进制位的权重
                ans *= a; //最终结果累乘a
                ans %= 1000; //求其后三位数
            }
            b /= 2; //b除以2
            a *= a; //求下一位二进制位的权重，a求其平方，即从a的1次开始，依次求的a
            的2次，a的4次...
            a %= 1000; //求a的后三位
        } //一边计算b的二进制值，一边计算a的 $2^k$ 次，并将需要的部分累乘到变量ans上
        printf("%d\n", ans); //输出
    }
    return 0;
}
```

为了使读者更好的理解该代码的原理，我们模拟该代码计算 2 的 31 次的值，并给出相应原理。

循环次数	a 的值	b 的值	ans 的值
0	2^1	31	1
1	$2^2(a * a)$	15 ($b / 2$)	$2^1 (ans * a)$
2	$2^4(a * a)$	7 ($b / 2$)	$2^3 (ans * a)$
3	$2^8(a * a)$	3 ($b / 2$)	$2^7(ans * a)$

4	$2^{16(a * a)}$	$1(b / 2)$	$2^{15(ans * a)}$
5	$2^{32(a * a)}$	$0(b / 2)$	$2^{31(ans * a)}$

即我们从 b 的最低位开始依次求得 b 的各二进制位，在当前二进制位为 1 的条件下将 a 累乘到变量 ans 上，在完成本位的操作后对 a 求其平方计算下一位二进制位的权重，直到完成对 b 的二进制转换。。

通过对本节的学习，读者对二分求幂应该有了更充分的理解，现在我们可以快速地求得 a 的 b 次方幂了，其中 b 为一般整数。

练习题：A sequence of numbers(九度教程第 58 题); TrA(九度教程第 59 题)（矩阵快速幂，与二分求幂原理相同）

九 高精度整数

在前文中曾多次出现这样的问题：有一些整数可能数值非常巨大以致于我们不能使用任何内置整数类型来保存它的值，在前面的例子中我们总是利用各种技巧回避了直接对其保存。但在有些问题中，我们不得不保存并处理这些数值巨大的整数，那么我们该如何处理呢？这就是本节要讨论的内容——高精度整数。

还要提醒习惯于使用 Java 的读者，如果你的机试系统允许你使用 Java，那本节的内容对你来说毫无用处，因为 Java 类库中已经内置了 BigInteger 类，你只需查阅相关资料了解其用法并完成本节练习即可。

对于广大使用 C/C++ 的读者，我们首先明确高精度整数的保存形式，我们常用如下结构体来保存一个高精度整数：

```
struct BigInteger {
    int digit[1000];
    int size;
};
```

其中 digit 数组用来保存大整数中每若干位的数字，这里我们暂且使用每 4 位为一个单位保存，size 为 digit 数组中第一个我们还没使用过的数组单元（即下一个我们可以使用的数组单元）。以整数 123456789 为例，当我们使用该结构体来保存该值时，其结果是这样的： $digit[0] = 6789; digit[1] = 2345; digit[2] = 1; size = 3;$

现在我们已经能够利用该结构体来保存大整数了，接下来我们即将完成对其运算的实现。

例 4.11 a+b（九度教程第 60 题）

特殊判题：否

```
#include <stdio.h>

#include <string.h>

struct BigInteger { //高精度整数结构体
    int digit[1000]; //按四位数一个单位保存数值
    int size; //下一个我们未使用的数组单元
    void init() { //对结构体的初始化
        for (int i = 0; i < 1000; i++) digit[i] = 0; //所有数位清0
        size = 0; //下一个未使用数组单元为0, 即没有一个单元被使用
    }
    void set(char str[]) { //从字符串中提取整数
        init(); //对结构体初始化
        int L = strlen(str); //计算字符串长度
        for (int i = L - 1, j = 0, t = 0, c = 1; i >= 0; i--) { //从最后一个字符
            开始倒序遍历字符串, j 控制每4个字符转换为一个数字存入数组, t临时保存字符转换为数字的中间
            值, c表示当前位的权重, 按1, 10, 100, 1000顺序变化
```

```

        t += (str[i] - '0') * c; //计算这个四位数中当前字符代表的数字, 即数字
乘以当前位权重

        j++; //当前转换字符数增加

        c *= 10; //计算下一位权重

        if (j == 4 || i == 0) { //若已经连续转换四个字符, 或者已经到达最后一个
字符

            digit[size++] = t; //将这四个字符代表的四位数存入数组, size移动到
下一个数组单位

            j = 0; //重新开始计算下4个字符

            t = 0; //临时变量清0

            c = 1; //权重变为1

        }
    }
}

void output() { //将该高精度整数输出

    for (int i = size - 1; i >= 0; i--) {

        if (i != size - 1) printf("%04d", digit[i]); //若当前输出的数字不
是最高位数字, 用%04的输出前导0, 即当前数字不足4位时由0补充, 如输出110001的后四位数

        else printf("%d", digit[i]); //若是最高位, 则无需输出前导零

    }

    printf("\n"); //换行
}

BigInteger operator + (const BigInteger &A) const { //加法运算符

    BigInteger ret; //返回值, 即两数相加的结果

    ret.init(); //对其初始化

    int carry = 0; //进位, 初值为0

    for (int i = 0; i < A.size || i < size; i++) {

        int tmp = A.digit[i] + digit[i] + carry; //计算两个整数当前位以及
来自低位的进位和

        carry = tmp / 10000; //计算该位的进位

        tmp %= 10000; //去除进位部分, 取后四位

        ret.digit[ret.size++] = tmp; //保存该位结果

    }

    if (carry != 0) { //计算结束后若最高位有进位

```

```

        ret.digit[ret.size++] = carry; //保存该进位
    }
    return ret; //返回
}
}a , b , c;
char str1[1002] , str2[1002];
int main () {
    while (scanf ("%s%s",str1,str2) != EOF) { //输入
        a.set(str1);b.set(str2); //用两个字符串分别设置两个高精度整数
        c = a + b; //计算它们的和
        c.output(); //输出
    }
    return 0;
}

```

实现高精度加法，即用代码模拟加法的运算法则，按照从低位开始各对应位相加并加上来自低位的进位从而获得本位的数值以及进位的规则进行运算。思路相对简单。读者只需牢记加法运算规则便能写出实现的代码。本例中采用了重载+运算符的方法来实现该加法运算，若读者对其不熟悉也可以使用定义一个参数为两个加数返回值为它们和的求和函数，它们的运算原理完全相同。

高精度减法运算法则与加法类似，从最低位开始依次相减并减去来自低位的借位从而得出本位的结果和向高位的借位。原理和代码都与加法类似，这里不再赘述由读者自行完成。接下去，我们再来探讨高精度乘法的运算，这里指的乘法运算一般为高精度整数乘以一般小整数的运算。至于两个高精度整数相乘在机试中考察的可能不大，这里不展开探讨。有兴趣的读者可自行查阅相关资料。

例 4.12 N 的阶层（九度教程第 61 题）

时间限制：3 秒

内存限制：128 兆

特殊判题：否

题目描述：

输入一个正整数 N，输出 N 的阶乘。

输入：

正整数 N($0 \leq N \leq 1000$)

输出：

输入可能包括多组数据，对于每一组输入数据，输出 N 的阶乘

样例输入：

4

5

15

样例输出:

24

120

1307674368000

来源:

2006 年清华大学计算机研究生机试真题

本例中，虽然输入的数据并不大，但是计算的结果却可能非常的大(1000!)，所以我们依旧需要利用高精度整数来完成计算。本例中涉及的高精度运算即是高精度乘法。

代码 4.12

```
#include <stdio.h>
#include <string.h>
struct BigInteger {
    int digit[1000];
    int size;
    void init() { //初始化
        for (int i = 0; i < 1000; i++) digit[i] = 0;
        size = 0;
    }
    void set (int x) { //用一个小整数设置高精度整数
        init();
        do { //对小整数4位为一个单位分解依次存入digit当中
            digit[size++] = x % 10000;
            x /= 10000;
        } while (x != 0);
    }
    void output() { //输出
        for (int i = size - 1; i >= 0; i--) {
            if (i != size - 1) printf("%04d", digit[i]);
            else printf("%d", digit[i]);
        }
        printf("\n");
    }
};
```

```

    }

    BigInteger operator * (int x) const { //乘法运算符
        BigInteger ret; //将要返回的高精度整数
        ret.init(); //初始化
        int carry = 0; //进位初始值为0
        for (int i = 0; i < size; i++) {
            int tmp = x * digit[i] + carry; //用小整数x乘以当前位数字并加上来自
            //低位的进位
            carry = tmp / 10000; //计算进位
            tmp %= 10000; //去除进位部分
            ret.digit[ret.size++] = tmp; //保存该位数字
        }
        if (carry != 0) { //若最高位有进位
            ret.digit[ret.size++] = carry; //保存该进位
        }
        return ret; //返回结果
    }
}a;

int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        a.init(); //初始化a
        a.set(1); //a初始值为1
        for (int i = 1; i <= n; i++) {
            a = a * i; //依次乘上每一个整数
        }
        a.output(); //输出a
    }
    return 0;
}

```

高精度乘法的原理与高精度加法类似,用将要乘的小乘数来乘高精度整数的每一位数并加上来自低位的进位,从而得到该位的结果以及向高位的进位。本例依然采用了重载乘法运算符的方法定义乘法,读者也可以使用定义其它函数的方法完成高精度乘法。

下面我们用最后一个例题来结束本章的内容，该例题较为综合，涉及到本章所讲的多方面内容。

例 4.13 进制转换（九度教程第 62 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

将 M 进制的数 X 转换为 N 进制的数输出。

输入：

输入的第一行包括两个整数：M 和 N($2 \leq M, N \leq 36$)。

下面的一行输入一个数 X，X 是 M 进制的数，现在要求你将 M 进制的数 X 转换成 N 进制的数输出。

输出：

输出 X 的 N 进制表示的数。

样例输入：

16 10

F

样例输出：

15

提示：

输入时字母部分为大写，输出时为小写，并且有大数据。

来源：

2008 年清华大学计算机研究生机试真题

该题初看起来很像一般的进制转换，但提示中明确告知，输入会有较大的数据，即我们为了完成本例需要的进制转换，需要使用高精度整数。同时，考虑到进制转换的内容，我们的高精度整数需要进行以下运算：高精度整数与普通整数的求积，高精度整数之间求和，高精度整数除以普通整数，高精度整数对普通整数求模等。下面给出本例代码，作为本章最后的总结内容。

代码 4.13

```
#include <stdio.h>
#include <string.h>
#define maxDigits 100
struct BigInteger { //高精度整数结构体
    int digit[maxDigits];
    int size;
    void init() { //初始化
```



```

        for (int i = 0; i < maxDigits; i++) digit[i] = 0;
        size = 0;
    }

    void set (int x) { //用一个普通整数初始化高精度整数
        init();
        do {
            digit[size++] = x % 10000;
            x /= 10000;
        } while(x != 0);
    }

    void output() { //输出
        for (int i = size - 1; i >= 0; i--) {
            if (i != size - 1) printf("%04d", digit[i]);
            else printf("%d", digit[i]);
        }
        printf("\n");
    }

    BigInteger operator * (int x) const { //高精度整数与普通整数的乘积
        BigInteger ret;
        ret.init();
        int carry = 0;
        for (int i = 0; i < size; i++) {
            int tmp = x * digit[i] + carry;
            carry = tmp / 10000;
            tmp %= 10000;
            ret.digit[ret.size++] = tmp;
        }
        if (carry != 0) {
            ret.digit[ret.size++] = carry;
        }
        return ret;
    }

    BigInteger operator + (const BigInteger &A) const { //高精度整数之间的加
法运算

```

```

        bigInteger ret;
        ret.init();
        int carry = 0;
        for (int i = 0; i < A.size || i < size; i++) {
            int tmp = A.digit[i] + digit[i] + carry;
            carry = tmp / 10000;
            tmp %= 10000;
            ret.digit[ret.size++] = tmp;
        }
        if (carry != 0) {
            ret.digit[ret.size++] = carry;
        }
        return ret;
    }

    bigInteger operator / (int x) const { //高精度整数除以普通整数
        bigInteger ret; //返回的高精度整数
        ret.init(); //返回值初始化
        int remainder = 0; //余数
        for (int i = size - 1; i >= 0; i--) { //从最高位至最低位依次完成计算
            int t = (remainder * 10000 + digit[i]) / x; //计算当前位数值加上
            //高位剩余的余数的和对x求得的商
            int r = (remainder * 10000 + digit[i]) % x; //计算当前位数值加上
            //高位剩余的余数的和对x求模后得的余数
            ret.digit[i] = t; //保存本位的值
            remainder = r; //保存至本位为止的余数
        }
        ret.size = 0; //返回高精度整数的size初始值为0, 即当所有位数字都为0
        //时, digit[0]代表数字0, 作为最高有效位, 高精度整数即为数字0
        for (int i = 0; i < maxDigits; i++) {
            if (digit[i] != 0) ret.size = i;
        } //若存在非0位, 确定最高的非0位, 作为最高有效位
        ret.size++; //最高有效位的下一位即为下一个我们不曾使用的digit数组单元, 确定
        //为size的值
        return ret; //返回
    }

```

```

}

int operator % (int x) const { //高精度整数对普通整数求余数
    int remainder = 0; //余数
    for (int i = size - 1; i >= 0; i --) {
        int t = (remainder * 10000 + digit[i]) / x;
        int r = (remainder * 10000 + digit[i]) % x;
        remainder = r;
    } //过程同高精度整数对普通整数求商
    return remainder; //返回余数
}

}a , b , c;

char str[10000];
char ans[10000];

int main () {
    int n, m;
    while (scanf ("%d%d", &m, &n) != EOF) {
        scanf ("%s", str); //输入m进制数
        int L = strlen(str);
        a.set(0); //a初始值为0, 用来保存转换成10进制的m进制数
        b.set(1); //b初始值为1, 在m进制向10进制转换的过程中, 依次代表每一位的权重
        for (int i = L - 1; i >= 0; i --) { //由低位至高位转换m进制数至相应的10进
制数
            int t;
            if (str[i] >= '0' && str[i] <= '9') {
                t = str[i] - '0';
            }
            else t = str[i] - 'A' + 10; //确定当前位字符代表的数字
            a = a + b * t; //累加当前数字乘当前位权重的积
            b = b * m; //计算下一位权重
        }
        int size = 0; //代表转换为n进制后的字符个数
        do { //对转换后的10进制数求其n进制值
            int t = a % n; //求余数
            if (t >= 10) ans[size++] = t - 10 + 'a';

```

```

        else ans[size++] = t + '0'; //确定当前位字符
        a = a / n; //求商
    }while(a.digit[0] != 0 || a.size != 1); //当a不为0时重复该过程
    for (int i = size - 1; i >= 0; i--) printf("%c", ans[i]);
    printf("\n"); //输出
}
return 0;
}

```

该代码中大部分内容在前文中已有涉及，只有高精度整数与普通整数求商、求模是新增加的内容，读者稍加研习应能够了解其中原理。实现高精度整数的各种运算，归根到底还是利用加减乘除的运算法则，对高精度整数的各位模拟该运算规律，只要读者熟知这些运算法则，并能按照其运算步骤写成代码，即能写出相应的高精度运算程序。

另外若是只需求出一个高精度整数除以一个小数后余下的整数，可以使用如下简短的代码(具体原理读者可自行思考)：

```

int ans = 0; //其中高精度大整数由高位至低位保存在字符数组str中，小整数保存在mod
中
for (int i = 0; str[i]; i++) {
    ans *= 10;
    ans += str[i] - '0';
    ans %= mod;
}
printf("%d\n", ans); //ans即为计算后剩下的余数

```

最后，要提醒大家的是：使用了高精度整数后，其程序就不能忽略运算本身带来的复杂度。如有高精度整数 a 、 b ，则求 $a+b$ 的和就不再能再在常数时间里得到结果，由于其高精度运算需要计算每一位的和，所以该运算复杂度变为 $O(\text{size})$ ，其中 size 为两个整数中较多的位数，即运算本身可能带来巨大的耗时。所以，在计算复杂度时我们不能忽略对高精度运算的耗时估计。

练习题：浮点数加法(九度教程第 63 题);大整数排序(九度教程第 64 题) (高精度整数比较大小) ;10 进制 vs2 进制(九度教程第 65 题);

总结

本章介绍了机试中可能涉及的数学问题，从求模运算符出发，了解了数位拆解并进一步延伸到进制转换。还讨论了经典数学问题：如何求两个数的最大公约数，以及方法类似的求两个数的最小公倍数。又介绍了一类特殊的数——素数，并提出如何求素数和分解素因数及其应用。最后，我们讨论了高精度整数的保存及各种运算的实现。通过这些问题的讲解，使读者对机试中的数学类问题有一个很好的把握。



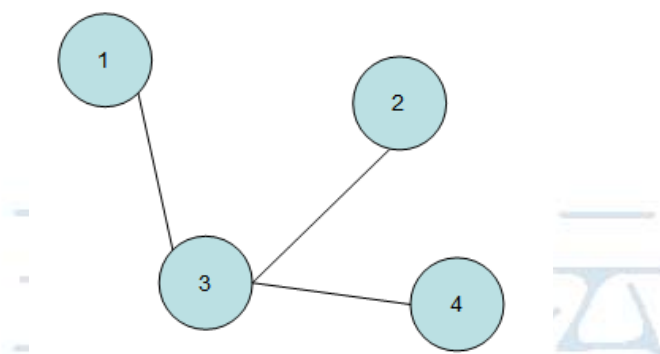
第5章 图论

在本章中，我们主要讨论计算机考研机试中有关图论的各类问题，主要涉及最小生成树、最短路径、拓扑排序等问题，为了解决这些问题，我们还需要掌握相关的数据结构并重点掌握并查集的相关操作和实现。

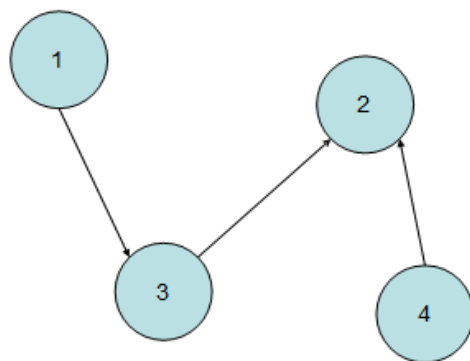
一 预备知识

本节主要介绍有关图论的一些预备知识，为接下来学习具体的图论问题作准备。

图是由结点（顶点）的有穷集合 V 和边的集合 E 组成的。分为有向图和无向图，在有向图中每条边都有方向，常将其称为弧，含箭头的一端称为弧头，另一端称为弧尾，记为 $\langle V_i, V_j \rangle$ ，表示存在一条从结点 V_i 指向结点 V_j 的有向边。无向图中每条边都没有方向，顶点之间存在边的关系称为相邻，记 (V_i, V_j) ，表示结点 V_i 和 V_j 相邻。

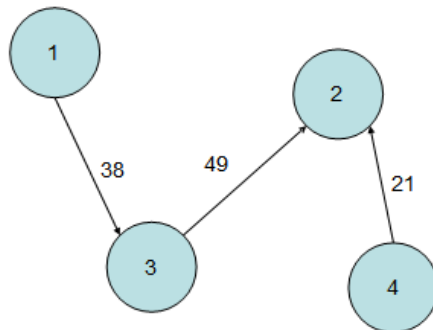


无向图



有向图

若为图上的边添加上代表某种实际意义的数值（长度、花费），则称这些数值为边的权，称包含这样带权边的图为带权图。



带权图

介绍完图的意义，我们就要考虑如何在计算机中表示图。表示图有两种常用的数据结构：邻接矩阵、邻接链表。

邻接矩阵用一个二维数组来表示图的相关信息，即用二维数组单元 $edga[i][j]$ 来表示结点 i 和结点 j 的关系。如上图所示无向图可由邻接矩阵表示为：

	1	2	3	4
1	0	0	1	0
2	0	0	1	0
3	1	1	0	1
4	0	0	1	0

我们用 $edge[i][j]$ 等于 1 来表示结点 i 和结点 j 之间存在边（有向图中表示存在从结点 i 指向结点 j 的一条弧），用 0 表示结点 i 和结点 j 之间不存在边（有向图中表示不存在从结点 i 指向结点 j 的一条弧）。除了用 0 和 1 来表示是否存在关系外，我们还能在邻接矩阵中保存边的权，从而用邻接矩阵表示带权图，如上带权图可被表示为：

	1	2	3	4
1	-1	-1	38	-1
2	-1	-1	-1	-1
3	-1	49	-1	-1
4	-1	21	-1	-1

若结点 i 和结点 j 之间存在边,我们用二维矩阵单元 $edge[i][j]$ 来保存其权值,若结点 i 和结点 j 之间不存在边,则 $edge[i][j]$ 为某约定的特殊字符(这里为-1)。

邻接矩阵原理易懂、用法简单,在确定某对结点之间是否存在关系时只需访问二维数组中相关单元即可,耗时较少。在有向图中也可以查找以任意结点为弧头或弧尾的所有弧。但其也存在一些缺陷,若需遍历与某结点相邻的所有结点,就需依次遍历二维数组中某行的所有元素,判断其值后决定是否相邻,也就是说即使只有一个点与其相邻,我们也需要耗费大量的时间来遍历该行中所有的数组单元,时间利用率较低。同时,用邻接矩阵来保存结点个数为 n 的图,其空间复杂度为 $O(n*n)$ 。当所要表示的图为稀疏图时该矩阵变为稀疏矩阵,大量的空间被浪费。所以,只有表示的图为稠密图,且频繁地判断某特定的结点对是否相邻时,使用邻接矩阵较为适宜。

邻接链表是一种链式存储结构,其为图的每个顶点建立一个单链表,第 i 个单链表中保存与结点 V_i 相邻的所有结点(无向图)或所有以结点 V_i 为弧尾的弧指向的结点(有向图)及其相关信息。

如上图所示有向图用邻接链表表示为:

1 →	3 →	NULL	
2 →	NULL		
3 →	2 →	NULL	
4 →	2 →	NULL	

当用其来表示带权图时,各链表单元在保存结点信息的同时也保存相应的边权信息,如上图所示的带权图用邻接链表表示为:

1 →	3 (38) →	NULL	
2 →	NULL		
3 →	2 (49) →	NULL	
4 →	2 (21) →	NULL	

邻接链表在遍历与某个特定结点相邻(无向图)或者以某个特定结点为弧尾

的弧的弧头指向结点（有向图）时效率较高，与邻接矩阵不同，它不用遍历不存在关系的其它结点，遍历的结点个数即为有效的结点个数，大大节省了时间。同时其空间复杂度为 $O(n + e)$ （ n 为点的数量， e 为边的数量），较邻接矩阵相比空间利用率较高。但是与邻接矩阵相比，当其需要判断结点 V_i 与 V_j 间是否存在关系时就显得比较繁琐，它需要遍历 V_i 和 V_j （无向图时任选一个）所有的邻接结点，才能判定它们之间是否存在关系。所以，若应用中存在大量遍历邻接结点的操作而较少判断两个特定结点的关系时，我们选用邻接链表较为适宜。

谈到邻接链表，读者可能会对链表产生恐惧。较好的使用链表而不出现错误需要对链表原理的深刻理解和扎实的编程功底，那么我该如何快速的学会使用邻接链表呢？我推荐使用标准模板库（STL）中的标准模板 `std::vector`。接下来，我们了解一些 `vector` 在实现邻接链表中的应用。

首先我们定义一个结构体，包括邻接结点和边权值，用来表示一条边。

```
struct Edge {  
    int nextNode; //下一个结点编号  
    int cost; //该边的权重  
};
```

我们为每一个结点都建立一个单链表来保存与其相邻的边权值和结点的信息。我们使用 `vector` 来模拟这些单链表，利用如下语句为每一个结点都建立一个 `vector` 对象（结点数量为 N ）。

```
vector<Edge> edge[N];
```

该语句建立了一个大小为 N 的数组，而数组中保存的元素即为 `vector` 对象，我们用 `edge[i]` 的 `vector` 来表示为结点 i 建立的单链表。

为了使用 `vector` 我们还需在 C++ 源文件头部添加相应的头文件。

```
#include <vector>  
  
using namespace std; //声明使用标准命名空间
```

下面，我们学习如何为这些“单链表”添加和删除信息。

利用

```
for (int i = 0; i < N; i++) { //遍历所有结点  
    edge[i].clear(); //清空其单链表  
}
```

来实现对这些单链表的初始化，即利用 `vector::clear()` 操作清空这些单链表。

当我们要向其中添加信息时，调用 `vector::push_back(Edge)`。如下所示：

```
Edge tmp; //准备一个Edge结构体  
tmp.nextNode = 3; //下一结点编号为3
```

```
tmp.cost = 38; //该边权值为38
edge[1].push_back(tmp); //将该边加入结点1的单链表中
```

当我们需要查询某个结点的所有邻接信息时，则对 `vector` 进行遍历。

```
for (int i = 0; i < edge[2].size(); i++) { //对edge[2]进行遍历,即对所有与结
点2相邻的边进行遍历,edge[2].size()表示其大小
    int nextNode = edge[2][i].nextNode; //读出邻接结点
    int cost = edge[2][i].cost; //读出该边权值
}
```

可见，对使用 `vector` 实现的邻接链表的访问非常类似于对二维数组的访问，但是其每行的长度是根据边的数量动态变化的。

当我们需要删除某个单链表中的某些边信息时，我们调用 `vector::erase`。

若我们要删除结点 1 的单链表中 `edge[1][i]` 所对应的边信息时，我们使用如下语句：

```
edge[1].erase(edge[1].begin() + i, edge[1].begin() + i + 1); // 即
vector.erase(vector.begin() + 第一个要删除的元素编号, vector.begin() + 最后一个
要删除元素的编号 + 1
```

读者只要记住如上 `vector` 的基本用法，就能使用其来模拟单链表，并能对这些单链表进行清空、添加、删除、遍历等操作。

本节说明图的基本概念，和图的两种保存形式，固不安排例题与练习题，但读者必须牢记 `vector` 使用方法，以期能够用 `vector` 来模拟单链表（假设你对建立单链表没有其他更好的方法）。

二 并查集

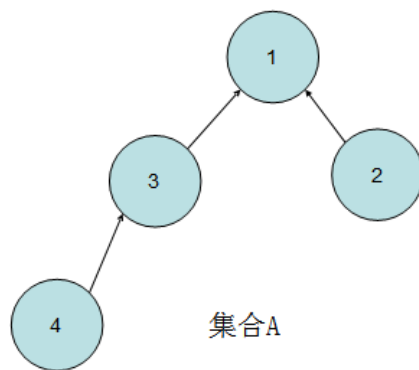
本节讨论在图论问题中常常要使用到的一种数据结构——集合，及其相关操作——并查集。

这种数据结构用来表示集合信息，用以实现如确定某个集合含有哪些元素、判断某两个元素是否存在同一个集合中、求集合中元素的数量等问题。

我们先来看如下的数字集合：

集合 $A\{1,2,3,4\}$ ，集合 $B\{5,6,7\}$ ，集合 $C\{8,0\}$

我们利用如下树结构来表示这些集合：

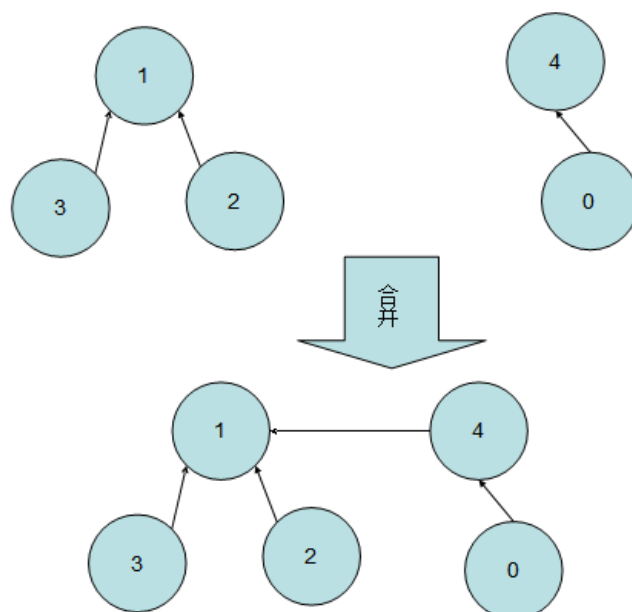


如图所示，我们用一棵树上的结点来表示在一个集合中的数字，要判断两个数字是否在一个集合中，我们只需判断它们是否在同一棵树中。那么我们使用双亲结点表示法来表示一棵树，即每个结点保存其双亲结点。若用数组来表示如上树，则得到如下结果：

1	2	3	4
-1	1	1	3

即我们在数组单元 i 中保存结点 i 的双亲结点编号，若该结点已经是根结点则其双亲结点信息保存为 -1。有了这样的存储结构，我们就能通过不断地求双亲结点来找到该结点所在树的根结点，若两个元素所在树的根结点相同，则可以判定它们在同一棵树上，它们同属一个集合。

对于合并两个集合的要求，我们该如何操作呢？我们只需要让分别代表两个集合的两棵树合并，合并方法为其中一棵树变为另一棵树根结点的子树，如下图所示：



如图，若我们对 2 所在的集合与 0 所在的集合合并，则先找到表示 2 所在集合的树的根结点 1 和表示 0 所在集合的树的根结点 4，并使其中之一（图中为 4）为另一个根结点的儿子结点，这样其中一棵树变为另一棵树根结点的一棵新子树，完成合并。在双亲结点表示法中，该合并过程为：

0	1	2	3	4
4	-1	1	1	-1

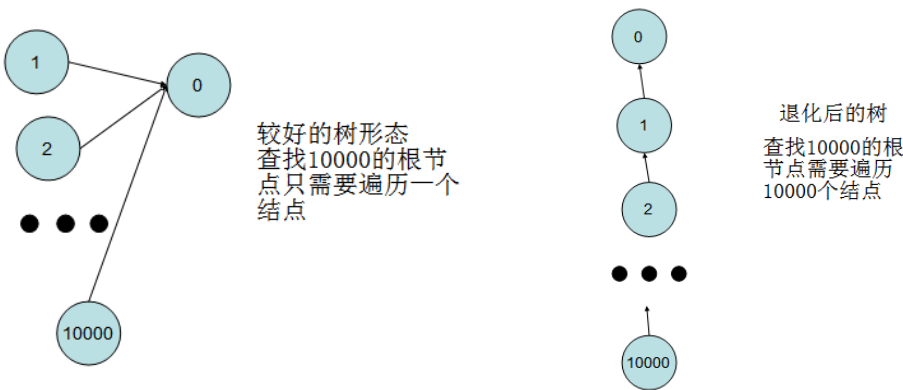
合并前

0	1	2	3	4
4	-1	1	1	1

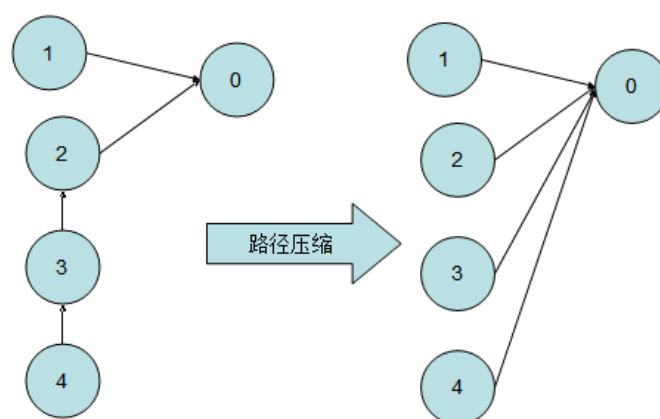
合并后

在树的双亲结点表示法中，两树的合并即表示为其中一棵树的根节的双亲结点变为另一棵树的根结点。

但是，采用这种策略而不加以任何约束也可能造成某些致命的问题。如前文所述，我们对集合的操作主要通过查找树的根结点来实现，那么并查集中最主要的操作即查找某个结点所在树的根结点，我们的方法是通过不断查找结点的双亲结点直到找到双亲结点不存在的结点为止，该结点即为根结点。那么，这个过程所需耗费的时间和该结点与树根的距离有关，即和树高有关。在我们合并两树的过程中，若只简单的将两树合并而不采取任何措施，那么树高可能会逐渐增加，查找根结点的耗时逐渐增大，极端情况下该树可能会退化成一个单链表。那么在其上进行查找根结点的操作将会变得非常得耗时，如下两图所示，树不同的形态对查找的效率将会有巨大的影响：



为了避免因为树的退化而产生额外的时间消耗，我们在合并两棵树时就不能任由其发展而应该加入一定的约束和优化，使其尽可能的保持较低的树高。为了达到这一目的，我们可以在查找某个特定结点的根结点时，同时将其与根结点之间所有的结点都直接指向根结点，这个过程被称为路径压缩，如下图所示：



如图所示，在完成路径压缩的工作后，树的形态发生巨大改变，树高大大降低，而该树所表示的集合信息却没有发生任何改变，所以其在保证集合信息不变的情况下大大优化了树结构，为后续的查找工作节约了大量的时间。

说了这么多，相信读者对并查集有关的原理性问题已经有了一定的了解。那么现在我们将注意力放到如何编写出相应的程序代码上来。

首先，我们定义一个数组，用双亲表示法来表示各棵树（所有的集合元素个数总和为 N）：

```
int Tree[N];
```

用 `Tree[i]` 来表示结点 `i` 的双亲结点，若 `Tree[i]` 为 -1 则表示该结点不存在双亲结点，即结点 `i` 为其所在树的根结点。

那么，为了查找结点 `x` 所在树的根结点，我们定义以下函数：

```
int findRoot(int x) {
    if (Tree[x] == -1) return x; //若当前结点为根结点则返回该结点号
    else return findRoot(Tree[x]); //否则递归查找其双亲结点的根结点
}
```

这里我们将查找函数写成了递归的形式，不熟悉递归的读者可以参考如下非递归形式的函数：

```
int findRoot(int x) {
    int ret;
    while (Tree[x] != -1)
        x = Tree[x]; //若当前结点为非根结点则一直查找其双亲结点
    return x;
}
```

```

ret = x; //返回根结点编号
return ret;
}

```

另外若需要在查找过程中添加路径压缩的优化，我们修改以上两个函数为：

```

int findRoot(int x) {
    if (Tree[x] == -1) return x;
    else {
        int tmp = findRoot(Tree[x]);
        Tree[x] = tmp; //将当前结点的双亲结点设置为查找返回的根结点编号
        return tmp;
    }
}

```

同样的，其非递归形式如下

```

int findRoot(int x) {
    int ret;
    int tmp = x;
    while (Tree[x] != -1)
        x = Tree[x];
    ret = x;
    x = tmp; //再做一次从结点x到根结点的遍历
    while (Tree[x] != -1) {
        int t = Tree[x];
        Tree[x] = ret;
        x = t; //遍历过程中将这些结点的双亲结点都设置为已经查找得到的根结点编号
    }
    return ret;
}

```

有了这些函数，结合并查集的工作原理和集合合并方法，我们就可以开始试着解决并查集问题了。

例 5.1 畅通工程（九度 OJ 1012）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

某省调查城镇交通状况，得到现有城镇道路统计表，表中列出了每条道路直

接连通的城镇。省政府“畅通工程”的目标是使全省任何两个城镇间都可以实现交通（但不一定有直接的道路相连，只要互相间接通过道路可达即可）。问最少还需要建设多少条道路？

输入：

测试输入包含若干测试用例。每个测试用例的第 1 行给出两个正整数，分别是城镇数目 N (< 1000) 和道路数目 M ；随后的 M 行对应 M 条道路，每行给出一对正整数，分别是该条道路直接连通的两个城镇的编号。为简单起见，城镇从 1 到 N 编号。当 N 为 0 时，输入结束，该用例不被处理。

输出：

对每个测试用例，在 1 行里输出最少还需要建设的道路数目。

样例输入：

```
4 2
1 3
4 3
3 3
1 2
1 3
2 3
5 2
1 2
3 5
999 0
0
```

样例输出：

```
1
0
2
998
```

来源：

2005 年浙江大学计算机及软件工程研究生机试真题

题面中描述的是一个实际的问题，但该问题可以被抽象成在一个图上查找连通分量（彼此连通的结点集合）的个数，我们只需求得连通分量的个数，就能得到答案（新建一些边将这些连通分量连通）。这个问题可以使用并查集完成，初始时，每个结点都是孤立的连通分量，当读入已经建成的边后，我们将边的两个

顶点所在集合合并，表示这两个集合中的所有结点已经连通。对所有的边重复该操作，最后计算所有的结点被保存在几个集合中，即存在多少棵树就能得知共有多少个连通分量（集合）。

代码 5.1

```
#include <stdio.h>

using namespace std;

#define N 1000

int Tree[N];

int findRoot(int x) { //查找某个结点所在树的根结点
    if (Tree[x] == -1) return x;
    else {
        int tmp = findRoot(Tree[x]);
        Tree[x] = tmp;
        return tmp;
    }
}

int main () {
    int n , m;
    while (scanf ("%d", &n) != EOF && n != 0) {
        scanf ("%d", &m);
        for (int i = 1; i <= n; i++) Tree[i] = -1; //初始时, 所有结点都是孤立的
        //集合, 即其所在集合只有一个结点, 其本身就是所在树根结点
        while(m -- != 0) { //读入边信息
            int a , b;
            scanf ("%d%d", &a, &b);
            a = findRoot(a);
            b = findRoot(b); //查找边的两个顶点所在集合信息
            if (a != b) Tree[a] = b; //若两个顶点不在同一个集合则合并这两个集合
        }
        int ans = 0;
        for (int i = 1; i <= n; i++) {
            if (Tree[i] == -1) ans++; //统计所有结点中根结点的个数
        }
    }
}
```

```

        printf("%d\n", ans - 1); //答案即为在ans个集合间再修建ans-1条道路即可使所有
        有结点连通
    }
    return 0;
}

```

该例很好的体现了机试题需要将实际问题抽象成数学问题的过程,为了解决该问题,我们将城市和道路抽象成结点和边,将相互可达的城市抽象成图上的连通分量,最后利用并查集这一数学手段来解决问题,这就是很典型的分析解决机试题的过程。

在上例中,我们了解了并查集可以被利用于求解图上连通分量个数。那么,并查集还有其他什么用处呢?

例 5.2 More is better (九度 OJ 1444)

时间限制: 1 秒

内存限制: 100 兆

特殊判题: 否

题目描述:

Mr Wang wants some boys to help him with a project. Because the project is rather complex, the more boys come, the better it will be. Of course there are certain requirements. Mr Wang selected a room big enough to hold the boys. The boy who are not been chosen has to leave the room immediately. There are 10000000 boys in the room numbered from 1 to 10000000 at the very beginning. After Mr Wang's selection any two of them who are still in this room should be friends (direct or indirect), or there is only one boy left. Given all the direct friend-pairs, you should decide the best way.

输入:

The first line of the input contains an integer n ($0 \leq n \leq 100\,000$) - the number of direct friend-pairs. The following n lines each contains a pair of numbers A and B separated by a single space that suggests A and B are direct friends. ($A \neq B$, $1 \leq A, B \leq 10000000$)

输出:

The output in one line contains exactly one integer equals to the maximum number of boys Mr Wang may keep.

样例输入:

```

4
1 2
3 4

```

5 6
1 6
4
1 2
3 4
5 6
7 8

样例输出：

4
2

题目大意：有 10000000 个小朋友，他们之中有 N 对好朋友，且朋友关系具有传递性：若 A 与 B 是朋友， B 与 C 是朋友，那么我们也认为 A 与 C 是朋友。在给出这 N 对朋友关系后，要求我们找出一个最大（人数最多）的集合，该集合中任意两人之间都是朋友或者该集合中只有一个人，输出该最大人数。

如前例所示，我们利用并查集相关操作已经可以求得有几个这样符合条件的集合，但是计算集合中的元素个数我们仍没有涉及。我们如果能够成功求得每个集合的元素个数，我们只需要选择包含元素最多的集合，并输出该集合中的元素个数即可。

为了计算每个集合的元素个数，我们不妨在表示每个集合的树的根结点记录该集合所包含的元素个数，在合并时累加被合并两个集合包含的元素个数。最后，找出所有集合中所包含元素最多的集合即是所求。

代码 5.2

```
#include <stdio.h>
using namespace std;
#define N 10000001
int Tree[N];
int findRoot(int x) { //查找结点x所在树的根结点
    if (Tree[x] == -1) return x;
    else {
        int tmp = findRoot(Tree[x]);
        Tree[x] = tmp;
        return tmp;
    }
}
```

```

    int sum[N]; //用sum[i]表示以结点i为根的树的结点个数,其中保存数据仅当Tree[i]为
-1即该结点为树的根结点时有效

    int main () {
        int n;
        while (scanf ("%d", &n) != EOF) {
            for (int i = 1; i < N; i++) { //初始化结点信息
                Tree[i] = -1; //所有结点为孤立集合
                sum[i] = 1; //所有集合的元素个数为1
            }
            while(n -- != 0) {
                int a , b;
                scanf ("%d%d", &a, &b);
                a = findRoot(a);
                b = findRoot(b);
                if (a != b) {
                    Tree[a] = b;
                    sum[b] += sum[a]; //合并两集时,将成为子树的树的根结点上保存的该集
合元素个数的数字累加到合并后新树的树根
                }
            }
            int ans = 1; //答案,答案至少为1。固这里先出初始化为1
            for (int i = 1; i <= N; i++) {
                if (Tree[i] == -1 && sum[i] > ans) ans = sum[i]; //统计最大值
            }
            printf("%d\n", ans); //输出
        }
        return 0;
    }
}

```

读者从本例中应该得到启发,我们在使用并查集的同时也可以在表示集合的树的根结点保存其它额外信息,并且在集合合并的过程中维护该值,以便于求解某些集合问题。

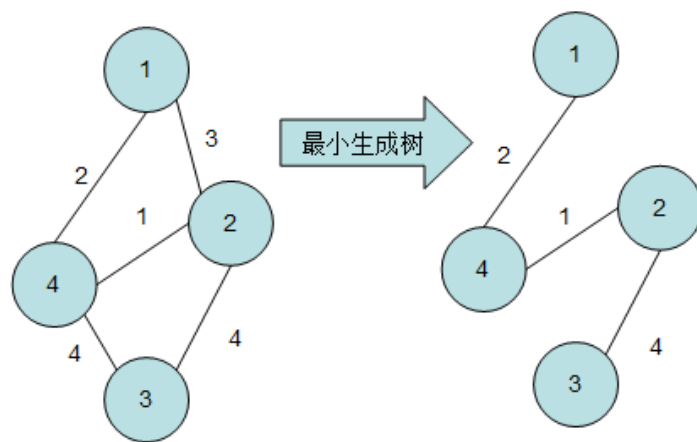
我们对并查集的讨论暂且到这里,在下一节最小生成树中并查集还会被我们所使用。

练习题: 九度 OJ1109 连通图; 九度 OJ1445 How Many Tables?; 九度 OJ1446 Head of a Gang;

三 最小生成树 (MST)

本节我们了解图论中的一类经典问题——最小生成树。

在一个无向连通图中, 如果存在一个连通子图包含原图中所有的结点和部分边, 且这个子图不存在回路, 那么我们称这个子图为原图的一棵生成树。在带权图中, 所有的生成树中边权的和最小的那棵 (或几棵) 被称为最小生成树。



最小生成树问题是图论中最经典的问题之一, 它在实际生活当中也有广泛的应用, 如在通信基站之间修建通信光缆使所有的基站间可以直接或间接通信, 最少需要多少长的光缆。要利用最小生成树来解决实际问题, 我们必须先学会怎样求解一个连通图的最小生成树。

我们先来看这样一个定理:

在要求解的连通图中, 任意选择一些点属于集合 A , 剩余的点属于集合 B , 必定存在一棵最小生成树包含两个顶点分别属于集合 A 和集合 B 的边 (即连通两个集合的边) 中权值最小的边。

我们可以用反证法来证明这个命题: 设连通结点集 A 和结点集 B 的边中权值最小的一条为 E , 在该图所有的最小生成树中都不包含该边。但在任意一棵最小生成树中必有一条边连通集合 A 和集合 B (若没有则两集合不连通, 若大于一条则出现回路), 设该边为 E' 。由命题假设可知, E 的权值不大于 E' 的权值, 若我们用边 E 替换边 E' , 替换后子图依然为原图的一棵生成树, 该生成树的权值为原最小生成树的权值减去 E' 的权值后加上 E 的权值, 该值将不会大于原最小生成树的权值, 那么新的生成树也是原图的一棵最小生成树, 我们就得到了一棵包含边 E 的最小生成树, 与假设矛盾, 故原命题得证。

这个结论就是我们将要介绍的求最小生成树 **Kruskal** 算法的算法原理，它按照如下步骤求解最小生成树：

1.初始时所有结点属于孤立的集合。

2.按照边权递增顺序遍历所有的边，若遍历到的边两个顶点仍分属不同的集合（该边即为连通这两个集合的边中权值最小的那条）则确定该边为最小生成树上的一条边，并将这两个顶点分属的集合合并。

3.遍历完所有边后，原图上所有结点属于同一个集合则被选取的边和原图中所有结点构成最小生成树；否则原图不连通，最小生成树不存在。

如步骤所示，在用 **Kruskal** 算法求解最小生成树的过程中涉及到大量的集合操作，我们恰好可以使用上一节中讨论的并查集来实现这些操作。

例 5.3 还是畅通工程 （九度 OJ 1017）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

某省调查乡村交通状况，得到的统计表中列出了任意两村庄间的距离。省政府“畅通工程的目标是使全省任何两个村庄间都可以实现公路交通（但不一定有直接的公路相连，只要间接通过公路可达即可），并要求铺设的公路总长度为最小。请计算最小的公路总长度。

输入：

测试输入包含若干测试用例。每个测试用例的第 1 行给出村庄数目 N (< 100)；随后的 $N(N-1)/2$ 行对应村庄间的距离，每行给出一对正整数，分别是两个村庄的编号，以及此两村庄间的距离。为简单起见，村庄从 1 到 N 编号。当 N 为 0 时，输入结束，该用例不被处理。

输出：

对每个测试用例，在 1 行里输出最小的公路总长度。

样例输入：

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


2 4 2

3 4 5

0

样例输出:

3

5

来源:

2006 年浙江大学计算机及软件工程研究生机试真题

在给定的道路中选取一些,使所有的城市直接或间接连通且使道路的总长度最小,该例即为典型的最小生成树问题。我们将城市抽象成图上的结点,将道路抽象成连接点的边,其长度即为边的权值。经过这样的抽象,我们求得该图的最小生成树,其上所有的边权和即为所求。

代码 5.3

```
#include <stdio.h>
#include <algorithm>
using namespace std;
#define N 101
int Tree[N];
int findRoot(int x) { //查找代表集合的树的根结点
    if (Tree[x] == -1) return x;
    else {
        int tmp = findRoot(Tree[x]);
        Tree[x] = tmp;
        return tmp;
    }
}

struct Edge { //边结构体
    int a, b; //边两个顶点的编号
    int cost; //该边的权值
    bool operator < (const Edge &A) const { //重载小于号使其可以按照边权从小到大排列
        return cost < A.cost;
    }
}edge[6000];
```

```

int main () {
    int n;
    while (scanf ("%d", &n) != EOF && n != 0) {
        for (int i = 1; i <= n * (n - 1) / 2; i++) {
            scanf ("%d%d%d", &edge[i].a, &edge[i].b, &edge[i].cost);
        } //输入
        sort(edge + 1, edge + 1 + n * (n - 1) / 2); //按照边权值递增排列所有的
边
        for (int i = 1; i <= n; i++)
            Tree[i] = -1; //初始时所有的结点都属于孤立的集合
        int ans = 0; //最小生成树上边权的和, 初始值为0
        for (int i = 1; i <= n * (n - 1) / 2; i++) { //按照边权值递增顺序遍历所
有的边
            int a = findRoot(edge[i].a);
            int b = findRoot(edge[i].b); //查找该边两个顶点的集合信息
            if (a != b) { //若它们属于不同集合, 则选用该边
                Tree[a] = b; //合并两个集合
                ans += edge[i].cost; //累加该边权值
            }
        }
        printf("%d\n", ans); //输出
    }
    return 0;
}

```

如代码所示, 我们使用并查集处理结点的集合属性, 初始时所有结点属于只包含其自身的孤立集合。我们按顺序遍历按照边权值递增排列的边时, 若该边的两个顶点属于两个不同的集合, 则合并这两个集合同时将该边的权值累加到答案中, 直到遍历完所有的边。该例不存在得不到最小生成树的情况, 所以最后我们并没有对所有结点是否属于同一个集合进行判断, 若可能出现不存在最小生成树的情况, 则该步不能省略。

在看了如此明显的最小生成树后, 我们再来看略微含蓄一点的例题。

例 5.4 Freckles (九度 OJ 1144)

时间限制: 1 秒

内存限制: 128 兆

特殊判题: 否

题目描述:

In an episode of the Dick Van Dyke show, little Richie connects the freckles on his Dad's back to form a picture of the Liberty Bell. Alas, one of the freckles turns out to be a scar, so his Ripley's engagement falls through. Consider Dick's back to be a plane with freckles at various (x,y) locations. Your job is to tell Richie how to connect the dots so as to minimize the amount of ink used. Richie connects the dots by drawing straight lines between pairs, possibly lifting the pen between lines. When Richie is done there must be a sequence of connected lines from any freckle to any other freckle.

输入:

The first line contains $0 < n \leq 100$, the number of freckles on Dick's back. For each freckle, a line follows; each following line contains two real numbers indicating the (x,y) coordinates of the freckle.

输出:

Your program prints a single real number to two decimal places: the minimum total length of ink lines that can connect all the freckles.

样例输入:

```
3
1.0 1.0
2.0 2.0
2.0 4.0
```

样例输出:

```
3.41
```

来源:

2009 年北京大学计算机研究生机试真题

题目大意为平面上有若干个点，我们需要用一些线段来将这些点连接起来使任意两个点能够通过一系列的线段相连，给出所有点的坐标，求一种连接方式使所有线段的长度和最小，求该长度和。

若我们将平面上的点抽象成图上的结点，将结点间直接相邻的线段抽象成连接结点的边，且权值为其长度，那么该类似于几何最优值的问题就被我们转化到了图论上的最小生成树问题。但在开始求最小生成树前，我们必须先建立该图，得出所有的边和相应的权值。

代码 5.4

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

```

#include <algorithm>
using namespace std;
#define N 101
int Tree[N];
int findRoot(int x) {
    if (Tree[x] == -1) return x;
    else {
        int tmp = findRoot(Tree[x]);
        Tree[x] = tmp;
        return tmp;
    }
}
struct Edge {
    int a , b;
    double cost; //权值变为长度, 固改用浮点数
    bool operator < (const Edge &A) const {
        return cost < A.cost;
    }
}edge[6000];
struct Point { //点结构体
    double x , y; //点的两个坐标值
    double getDistance(Point A) { //计算点之间的距离
        double tmp = (x - A.x) * (x - A.x) + (y - A.y) * (y - A.y);
        return sqrt(tmp);
    }
}list[101];
int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        for (int i = 1; i <= n; i++) {
            scanf ("%lf%lf", &list[i].x, &list[i].y);
        } //输入
        int size = 0; //抽象出的边的总数
        for (int i = 1; i <= n; i++) {

```

```

        for (int j = i + 1; j <= n; j++) { //连接两点的线段抽象成边
            edge[size].a = i;
            edge[size].b = j; //该边的两个顶点编号
            edge[size].cost = list[i].getDistance(list[j]); //边权值为两
点之间的长度

            size++; //边的总数增加
        } //遍历所有的点对
    }

    sort(edge, edge + size); //对边按权值递增排序
    for (int i = 1; i <= n; i++) {
        Tree[i] = -1;
    }

    double ans = 0;
    for (int i = 0; i < size; i++) {
        int a = findRoot(edge[i].a);
        int b = findRoot(edge[i].b);
        if (a != b) {
            Tree[a] = b;
            ans += edge[i].cost;
        }
    } //最小生成树

    printf("%.2lf\n", ans); //输出
}

return 0;
}

```

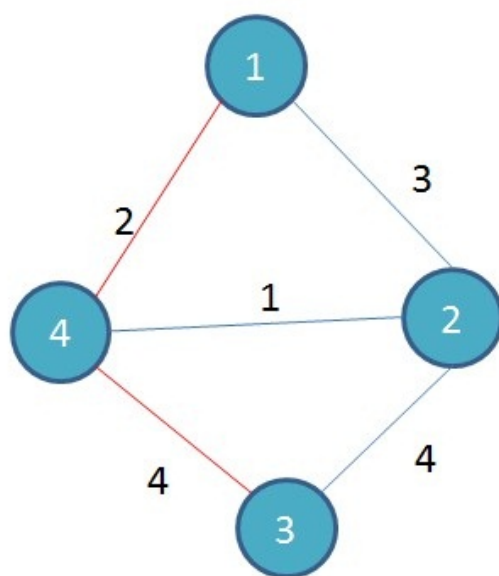
该例中的最小生成树就套上了马甲，不再像上例一样显而易见，只有我们仔细分析题目要求，并对实际问题作适当的抽象才能发现其真正数学模型，并用相应的方法解决它。

最小生成树算法除了本节所讨论的 **Kruskal** 算法，还有其他一些算法，比较有名的还有 **Prim** 算法。但两者在机试题上应用时差别不大，所以这里不再讨论 **Prim** 算法，有兴趣的读者可以自行查阅相关资料。读者只需牢记 **Kruskal** 算法的算法原理和相关编码技巧，就能掌握机试题中的最小生成树问题。

练习题：九度 OJ1154 Jungle Roads; 九度 OJ1024 畅通工程 (MST 可能不存在);

四 最短路径

在讨论完最小生成树后，我们再来了解图论中另一个经典问题：最短路径问题。即寻找图中某两个特定结点间最短的路径长度。所谓图上的路径，即从图中一个起始结点到一个终止结点途中经过的所有结点序列，路径的长度即所经过的边权和。



从结点1到结点3的
最短路径{1, 4, 3}
其长度为6

最短路径问题在实际中的应用也非常广泛，例如确定某两个城市间的最短行车路线长度。要解决这类问题，我们要根据图的特点和问题的特征选择不同的算法。

我们首先来介绍第一种计算最短路径长度的算法——Floyd 算法。

Floyd 算法又被称为佛洛依德算法，其算法思路如下：

用邻接矩阵保存原图，那么此时邻接矩阵中 $\text{edge}[i][j]$ 的值即表示从结点 i 到结点 j ，中间不经过任何结点时距离的最小值(若它们之间有多条边，取最小权值保存至邻接矩阵；也可能为无穷，即不可达)。假设结点编号为 1 到 N ，我们再考虑从结点 i 到结点 j 中间只能经过编号小于等于 1 的结点（也可以不经过）时最短路径长度。与原始状况相比，在中间路径上可以经过的结点增加了编号为 1 的结点。我们又知道，最短路径上的结点一定不会出现重复（不考虑存在负权值的情况）。那么，某两个结点间若由于允许经过结点 1 而出现了新的最短路径，则该路径被结点 1 分割成两部分：由 i 到结点 1，同时中间路径上不经过结点 1 的第一段路径；由结点 1 到 j ，中间路径上同样不经过结点 1 的第二段路径，其路径总长度为 $\text{edge}[i][1] + \text{edge}[1][j]$ 。要确定该路径是否比不允许经过结点 1 时

更短，我们比较 $\text{edge}[i][1] + \text{edge}[1][j]$ 与 $\text{edge}[i][j]$ 之间的大小关系。若前者较小，则说明中间路径经过结点 1 时比原来更短，则用该值代表由 i 到 j 中间路径结点编号小于等于 1 的最短路径长度；否则，该路径长度将依然保持原值 $\text{edge}[i][j]$ ，即虽然允许经过结点 1，但是不经过时路径长度最短。

考虑更一般的情况，若 $\text{edge}[i][j]$ 表示从结点 i 到结点 j ，中间只能经过编号小于 k 的点时的最短路径长度，我们可以由这些值确定当中间允许经过编号小于等于 k 的结点时，它们之间的最短路径长度。同样，与原情况相比，新情况中允许出现在中间路径的结点新增了编号为 k 的结点，同理我们确定 $\text{edge}[i][k] + \text{edge}[k][j]$ 的值与 $\text{edge}[i][j]$ 的值，若前者较小则该值代表了新情况中从结点 i 到结点 j 的最短路径长度；否则，新情况中该路径长度依旧保持不变。

如上文所说，在图的邻接矩阵表示法中， $\text{edge}[i][j]$ 表示由结点 i 到结点 j 中间不经过任何结点时的最短距离，那么我们依次为中间允许经过的结点添加结点 1、结点 2、……直到结点 N ，当添加完这些结点后，从结点 i 到结点 j 允许经过所有结点的最短路径长度就可以确定了，该长度即为原图上由结点 i 到结点 j 的最短路径长度。

我们设 $\text{ans}[k][i][j]$ 为从结点 i 到结点 j 允许经过编号小于等于 k 的结点时其最短路径长度。如上文， $\text{ans}[0][i][j]$ 即等于图的邻接矩阵表示中 $\text{edge}[i][j]$ 的值。我们通过如下循环，完成所有 k 对应的 $\text{ans}[k][i][j]$ 值的求解：

```
for (int k = 1; k <= n; k++) { //从1至n循环k
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) { //遍历所有的ij
            if (ans[k - 1][i][k] == 无穷 || ans[k - 1][k][j] == 无穷) { //
                //若当允许经过前k-1个结点时,i或j不能与k连通,则ij之间到目前为止不存在经过k的路径
                ans[k][i][j] = ans[k - 1][i][j]; //保持原值,即从i到j
                //允许经过前k个点和允许经过前k-1个结点时最短路径长度相同
            }
            continue; //继续循环
        }
        if (ans[k - 1][i][j] == 无穷 || ans[k - 1][i][k] + ans[k - 1][k][j] < ans[k - 1][i][j]) //若经过前k-1个结点,i和j不连通 或者 通过经过结点k可以得到比原来更短的路径
            ans[k][i][j] = ans[k - 1][i][k] + ans[k - 1][k][j];
        //更新该最短值
    }
    else ans[k][i][j] = ans[k - 1][i][j]; //否则保持原状
}
```



```

    }
}

```

经过这样的 n 次循环后，我们即可得到所有结点间允许经过所有结点条件下的最短路径长度，该路径长度即为我们要求的最短路径长度。即若要求得 ab 之间的最短路径长度，其答案为 $ans[n][a][b]$ 的值。

同时我们注意到，我们在通过 $ans[k-1][i][j]$ 的各值来递推求得 $ans[k][i][j]$ 的值时，所有的 $ans[k][i][j]$ 值将由 $ans[k-1][i][j]$ 和 $ans[k-1][i][k] + ans[k-1][k][j]$ 的大小关系确定，但同时 $ans[k][i][k]$ 和 $ans[k][k][j]$ 必定与 $ans[k-1][i][k]$ 和 $ans[k-1][k][j]$ 的值相同，即这些值不会因为本次更新而发生改变。所以我们将如上代码片段简化成如下形式：

```

for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            if (ans[i][k] == 无穷 || ans[k][j] == 无穷) continue;
            if (ans[i][j] == 无穷 || ans[i][k] + ans[k][j] < ans[i][j])
                ans[i][j] = ans[i][k] + ans[k][j];
        }
    }
}

```

如该代码片段所示，我们将原本的三维数组简化为二维数组，而每次更新时直接在该二维数组上进行更新。这是有原因的，当最外层循环由 $k-1$ 变为 k 时，各 $ans[i][k]$ 和 $ans[k][j]$ 的值不会因为本次更新发生改变（当前 i 到 k 的最短路径中途必不经过结点 k ），而本次更新又是由它们的值和各 $ans[i][j]$ 的值比较而进行的。所以我们直接在二维数组上进行本次更新，并不会影响到本次更新中其它各值的判定。节省了大量的内存空间，同时还省略了保持原值的操作。

我们来看一个例题，了解 Floyd 算法的实际运用：

例 5.5 最短路（九度 OJ 1447）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

在每年的校赛里，所有进入决赛的同学都会获得一件很漂亮的 t-shirt。但是每当我们的工作人员把上百件的衣服从商店运回到赛场的时候，却是非常累的！所以现在他们想要寻找最短的从商店到赛场的路线，你可以帮助他们吗？

输入：

输入包括多组数据。每组数据第一行是两个整数 N 、 M ($N \leq 100, M \leq 10000$)， N 表示成都的大街上有几个路口，标号为 1 的路口是商店所在地，标号为 N 的路口是赛场所在地， M 则表示在成都有几条路。 $N=M=0$ 表示输入结束。接下来 M 行，每行包括 3 个整数 A, B, C ($1 \leq A, B \leq N, 1 \leq C \leq 1000$)，表示在路口 A 与路口 B 之间有一条路，我们的工作人员需要 C 分钟的时间走过这条路。输入保证至少存在 1 条商店到赛场的路线。

当输入为两个 0 时，输入结束。

输出：

对于每组输入，输出一行，表示工作人员从商店走到赛场的最短时间。

样例输入：

```
2 1
1 2 3
3 3
1 2 5
2 3 5
3 1 2
0 0
```

样例输出：

```
3
2
```

我们首先分析复杂度，如我们在上文中给出的代码所示，floyd 算法主要包括一个三重循环，每重循环的循环次数均是 N ，这样 Floyd 算法的时间复杂度为 $O(N^3)$ ，空间复杂度为 $O(N^2)$ ，其中 N 均为图中结点的个数。

在本例中 N 最大值为 100， N^3 的时间复杂度尚在我们可以接受的范围内。

代码 5.5

```
#include <stdio.h>

int ans[101][101]; //二维数组, 其初始值即为该图的邻接矩阵

int main () {
    int n , m;
    while (scanf ("%d%d", &n, &m) != EOF) {
        if (n == 0 && m == 0) break;
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= n; j++) {
                ans[i][j] = -1; //对邻接矩阵初始化, 我们用-1代表无穷
```

```

    }

    ans[i][i] = 0; //自己到自己的路径长度设为0
}

while(m --) {
    int a , b , c;

    scanf ("%d%d%d", &a, &b, &c);

    ans[a][b] = ans[b][a] = c; //对邻接矩阵赋值, 由于是无向图, 该赋值操作
要进行两次
}

for (int k = 1; k <= n; k ++) { //k从1到N循环, 依次代表允许经过的中间结点编
号小于等于k

    for (int i = 1; i <= n; i ++) {
        for (int j = 1; j <= n; j ++) { //遍历所有ans[i][j], 判断其值保
持原值还是将要被更新

            if (ans[i][k] == -1 || ans[k][j] == -1) continue; //若
两值中有一个值为无穷, 则ans[i][j]不能由于经过结点k而被更新, 跳过循环, 保持原值

            if (ans[i][j] == -1 || ans[i][k] + ans[k][j] < ans[i][j])
                ans[i][j] = ans[i][k] + ans[k][j]; //当由于经过k可以获
得更短的最短路径时, 更新该值

        }
    }
}

printf("%d\n", ans[1][n]); //循环结束后输出答案
}

return 0;
}

```

我们总结 Floyd 算法的特点。首先，牢记其时间复杂度为 $O(N^3)$ ，所以在大部分机试题的时间允许范围内，它要求被求解图的大小不大于 200 个结点，若超过该数字该算法很可能因为效率不够高而被判超时。第二，Floyd 算法利用一个二维矩阵来进行相关运算，所以当图使用邻接矩阵表示时更为方便。若原图并非由邻接矩阵给出时我们设法将其转换，注意当两个结点间有多余一条边时，我们选择长度最小的边权值存入邻接矩阵。第三，当 Floyd 算法完成后，图中所有结点之间的最短路都将被确定。所以，其较适用于要求询问多个结点对之间的最短路径长度问题，即全源最短路问题。了解它的这些特点，将有利于我们在特定问

题中决定是否使用 Floyd 算法。

在讨论完 Floyd 算法后，我们来看另一种与其特点完全不同的最短路径算法——Dijkstra 算法(迪杰斯特拉算法)。它与之前讨论的 Floyd 算法有一个非常明显的区别，Floyd 算法可以计算出图上所有结点对之间的最短路径长度，Dijkstra 算法只能求得某特定结点到其它所有结点的最短路径长度，即单源最短路径问题。

那么该如何确定某特定结点到其它所有结点的最短距离呢？我们将按照最短路径长度递增的顺序确定每一个结点的最短路径长度，即先确定的结点的最短路径长度不大于后确定的结点的最短路径长度。这样有一个好处，当确定一个结点的最短路径长度时，该最短路径上所有中间结点的最短路径长度必然已经被确定了（中间路径的最短路径长度必小于这个结点的最短路径长度）。不妨设有结点 1 到 N，我们将要求从结点 1 出发到其它所有结点的最短路径长度。初始时，设结点 1 到其它所有点的最短路径长度均为无穷（或不确定的），我们立即确定由结点 1 到结点 1 的最短路径长度距离为 0。设已经确定最短路径长度的结点集合为集合 K，于是我们将结点 1 加入该集合。将问题一般化，假设集合 K 中已经保存了最短路径长度最短的前 m 个结点，它们是 P_1, P_2, \dots, P_m ，并已经得出它们的最短路径长度。那么第 m+1 近的结点与结点 1 的最短路径上的中间结点一定全部属于集合 K，这是因为若最短路径上中间有一个不属于集合 K 的结点，则它的最短路径距离一定小于第 m+1 近的结点的最短路径长度，与距离小于第 m+1 近的结点的最短路径已经全部确定、这样的结点全部属于集合 K 矛盾。那么第 m+1 近结点的最短路径必是由以下两部分组成，从结点 1 出发经由已经确定的最短路径到达集合 K 中的某结点 P_2 ，再由 P_2 经过一条边到达该结点。为此，我们遍历与集合 K 中结点直接相邻的边，设其为 (U, V, C) ，其中 U 属于集合 K，V 不属于集合 K，计算由结点 1 出发经过已经确定的最短路到达结点 U，再由结点 U 经过该边到达结点 V 的路径长度。该路径长度为已经确定的结点 U 的最短路径长度+C。所有与集合 K 直接相邻的非集合 K 结点中，该路径长度最短的那个结点即确定为第 m+1 近结点，并将该点假如集合 K。如此往复，直到结点 1 到所有结点的最短路径全部确定。

如上文所示，Dijkstra 算法流程如下：

- 1.初始化，集合 K 中加入结点 1，结点 1 到结点 1 最短距离为 0，到其它结点为无穷（或不确定）。

- 2.遍历与集合 K 中结点直接相邻的边 (U, V, C) ，其中 U 属于集合 K，V 不属于集合 K，计算由结点 1 出发按照已经得到的最短路到达 U，再由 U 经过该边到达 V 时的路径长度。比较所有与集合 K 中结点直接相邻的非集合 K 结点

该路径长度，其中路径长度最小的结点被确定为下一个最短路径确定的结点，其最短路径长度即为这个路径长度，最后将该结点加入集合 K。

3.若集合 K 中已经包含了所有的点，算法结束；否则重复步骤 2。

我们使用 Dijkstra 算法重写例 4.5。

代码 5.6

```
#include <stdio.h>
#include <vector>
using namespace std;

struct E{ //邻接链表中的链表元素结构体
    int next; //代表直接相邻的结点
    int c; //代表该边的权值(长度)
};

vector<E> edge[101]; //邻接链表
bool mark[101]; //标记, 当mark[j]为true时表示结点j的最短路径长度已经得到, 该结点已经加入集合K

int Dis[101]; //距离向量, 当mark[i]为true时, 表示已得的最短路径长度; 否则, 表示所有从结点1出发, 经过已知的最短路径达到集合K中的某结点, 再经过一条边到达结点i的路径中最短的距离

int main () {
    int n, m;
    while (scanf ("%d%d", &n, &m) != EOF) {
        if (n == 0 && m == 0) break;
        for (int i = 1; i <= n; i++) edge[i].clear(); //初始化邻接链表
        while(m --) {
            int a, b, c;
            scanf ("%d%d%d", &a, &b, &c);
            E tmp;
            tmp.c = c;
            tmp.next = b;
            edge[a].push_back(tmp);
            tmp.next = a;
            edge[b].push_back(tmp); //将邻接信息加入邻接链表, 由于原图为无向图, 固
            //每条边信息都要添加到其两个顶点的两条单链表中
        }
    }
}
```

```

for (int i = 1; i <= n; i++) { //初始化
    Dis[i] = -1; //所有距离为-1, 即不可达
    mark[i] = false; //所有结点不属于集合K
}

Dis[1] = 0; //得到最近的点为结点1, 长度为0
mark[1] = true; //将结点1加入集合K
int newP = 1; //集合K中新加入的点为结点1

for (int i = 1; i < n; i++) { //循环n-1次, 按照最短路径递增的顺序确定其他
n-1个点的最短路径长度
    for (int j = 0; j < edge[newP].size(); j++) { //遍历与该新加入集合
K中的结点直接相邻的边
        int t = edge[newP][j].next; //该边的另一个结点
        int c = edge[newP][j].c; //该边的长度
        if (mark[t] == true) continue; //若另一个结点也属于集合K, 则跳过
        if (Dis[t] == -1 || Dis[t] > Dis[newP] + c) //若该结点尚不可
达, 或者该结点从新加入的结点经过一条边到达时比以往距离更短
            Dis[t] = Dis[newP] + c; //更新其距离信息
    }

    int min = 123123123; //最小值初始化为一个大整数, 为找最小值做准备
    for (int j = 1; j <= n; j++) { //遍历所有结点
        if (mark[j] == true) continue; //若其属于集合K则跳过
        if (Dis[j] == -1) continue; //若该结点仍不可达则跳过
        if (Dis[j] < min) { //若该结点经由结点1至集合K中的某点在经过一条
边到达时距离小于当前最小值
            min = Dis[j]; //更新其为最小值
            newP = j; //新加入的点暂定为该点
        }
    }

    mark[newP] = true; //将新加入的点加入集合K, Dis[newP]虽然数值不变, 但
意义发生变化, 由 所有经过集合K中的结点再经过一条边到达时的距离中的最小值 变为 从结点1到
结点newP的最短距离
}

printf("%d\n", Dis[n]); //输出
}

```



```
return 0;  
}
```

如代码所示，我们用 `mark[i]` 的值来确定结点 `i` 的最短路径是否已经被确定，并根据结点 `i` 是否属于集合 `K`，`Dis[i]` 中的数字体现出不同的意义。当结点 `i` 属于集合 `K` 时，`Dis[i]` 代表结点 `i` 已经被确定的最短路径长度；相反，若结点 `i` 不属于集合 `K`，那么 `Dis[i]` 表示，所有从结点 1 出发先按照某条最短路径到达某已经在集合 `K` 中的结点，并由该结点经过一条边到达结点 `i` 路径中的最短距离。每当有新的结点 `newP` 加入集合 `K` 时，我们标记 `mark[newP]` 为 `true`，同时计算与结点 `newP` 直接相邻的非集合 `K` 中的结点，经过由结点 1 经过最短路径到达 `newP`，再由 `newP` 经过一条边到达该结点的距离，是否比由集合 `K` 中其它结点经过一条边到达时更短，若更短则修改 `Dis` 数组中相关的值，否则不改变 `Dis` 中的值。当所有与 `newP` 相邻的点修改完成后，我们遍历 `Dis` 数组中不属于集合 `K` 的结点，找到距离最短的那个，该结点即为下一个最短距离被确定的点，将该点加入集合 `K`，并使 `newP` 等于该结点的编号。当所有点的最短路径都确定后，`Dis[n]` 中的值即为由结点 1 到达其它所有结点的最短路径长度。

该代码中，使用了邻接链表保存图信息。由此可见，Dijkstra 算法很好的支持邻接链表，但同时它也可以被应用于邻接矩阵。这里使用邻接链表，是为了方便读者更详细的了解用 `vector` 模拟邻接链表的方法。

我们再来看一个例题：

例 5.6 最短路径问题（九度 OJ 1008）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

给你 `n` 个点，`m` 条无向边，每条边都有长度 `d` 和花费 `p`，给你起点 `s` 终点 `t`，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：

输入 `n,m`，点的编号是 `1~n`，然后是 `m` 行，每行 4 个数 `a,b,d,p`，表示 `a` 和 `b` 之间有一条边，且其长度为 `d`，花费为 `p`。最后一行是两个数 `s,t`；起点 `s`，终点 `t`。
`n` 和 `m` 为 0 时输入结束。（ $1 < n \leq 1000$, $0 < m < 100000$, $s \neq t$ ）

输出：

输出 一行有两个数， 最短距离及其花费。

样例输入：

3 2

1 2 5 6

2 3 4 5

1 3

0 0

样例输出:

9 11

来源:

2010 年浙江大学计算机及软件工程研究生机试真题

在该例中，我们不仅要求得起点到终点的最短距离，还需要在有多条最短路径的时，选取花费最少的那一条。要解决这个问题，我们只要更改 Dijkstra 算法中关于“更近”的评判标准即可：有两条路径，若它们距离不一样时，距离小的更近；若距离一样时花费少的更近。当定义这种新的评判标准后，Dijkstra 算法照样能为我们求得“最近”的路径长度。

代码 5.7

```
#include <stdio.h>
#include <vector>
using namespace std;
struct E{ //邻接链表元素结构体
    int next;
    int c;
    int cost;
};
vector<E> edge[1001]; //邻接链表
int Dis[1001]; //距离数组
int cost[1001]; //花费数组
bool mark[1001]; //是否属于集合K数组
int main () {
    int n, m;
    int S, T; //起点, 终点
    while (scanf ("%d%d", &n, &m) != EOF) {
        if (n == 0 && m == 0) break;
        for (int i = 1; i <= n; i++) edge[i].clear(); //初始化邻接链表
        while(m --) {
            int a, b, c, cost;
            scanf ("%d%d%d%d", &a, &b, &c, &cost);
```



```

    E tmp;

    tmp.c = c;

    tmp.cost = cost; //邻接链表中增加了该边的花费信息
    tmp.next = b;
    edge[a].push_back(tmp);
    tmp.next = a;
    edge[b].push_back(tmp);
}

scanf ("%d%d", &S, &T); //输入起点终点信息
for (int i = 1; i <= n; i++) { //初始化
    Dis[i] = -1;
    mark[i] = false;
}
Dis[S] = 0;
mark[S] = true;
int newP = S; //起点为S, 将其加入集合K, 且其最短距离确定为0
for (int i = 1; i < n; i++) {
    for (int j = 0; j < edge[newP].size(); j++) {
        int t = edge[newP][j].next;
        int c = edge[newP][j].c;
        int co = edge[newP][j].cost; // 花费
        if (mark[t] == true) continue;
        if (Dis[t] == -1 || Dis[t] > Dis[newP] + c || Dis[t] == Dis[newP]
+ c && cost[t] > cost[newP] + co) { //比较大小, 将距离相同但花费更短也作为更新的
条件之一
            Dis[t] = Dis[newP] + c;
            cost[t] = cost[newP] + co; //更新花费
        }
    }
}

int min = 123123123;
for (int j = 1; j <= n; j++) { //选择最小值, 选择时不用考虑花费的因素,
因为距离最近的点的花费已经不可能由于经过其它点而发生改变了
    if (mark[j] == true) continue;
    if (Dis[j] == -1) continue;

```

```

        if (Dis[j] < min) {
            min = Dis[j];
            newP = j;
        }
    }

    mark[newP] = true;
}

printf("%d %d\n", Dis[T], cost[T]); //输出答案
}

return 0;
}

```

值得一提的是，若由结点 U 到结点 V 的最短路径不存在，即他们不连通，那么当 Dijkstra 算法完成以后， V 结点仍然不属于集合 K 。即当完成 Dijkstra 算法后， $\text{mark}[V]$ 依然为 `false` 即说明，结点 U 到结点 V 的最短路不存在。注：该最短路不存在，指结点 U 和 V 不连通的情况，我们不考虑存在负环的情况，边的权值为负这种特殊的情况在机试中考察的可能性不大，但若真的出现边的权值为负，若不存在负环则最短路存在，但我们不能使用 Dijkstra 对其进行求解，因为 Dijkstra 算法原理在存在负权值的图上不成立；若存在负环则最短路不存在。要求解包含负权值边上的最短路问题，我们需要使用 SPFA 算法，有兴趣的读者可以自行查阅相关资料（该知识点在机试中考察的概率不大）。

最后，我们总结一下 Dijkstra 算法的特点：它的时间复杂度为 $O(N^2)$ （若在查找最小值处利用堆进行优化，则时间复杂度可以降到 $O(N \cdot \log N)$ ）， N 为结点的个数。空间复杂度为 $O(N)$ （不包括保存图所需的空间）。它同时适用于邻接矩阵和邻接链表形式保存的有向图和无向图。它求解从某一个特定的起点出发，到其它所有点的最短路径，即单源最短路径问题。

本节介绍了两种求最短路径算法，分别适用于求解两种不同的最短路径问题：全源最短路和单源最短路。他们的时空复杂度均有一些差异，读者需要自行权衡利弊后选择。

练习题：九度 OJ1100 最短路径（需要使用高精度整数）；九度 OJ1162 I wanna go home（提示：对跨越两个阵营的边只保存单向边即可）；

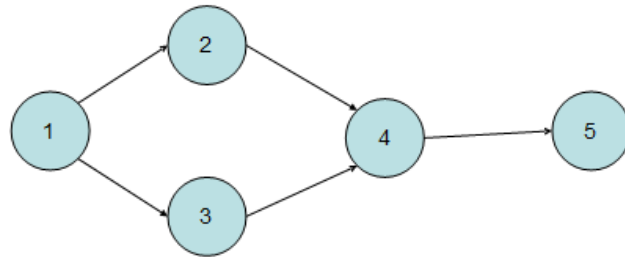
五 拓扑排序

本节我们来讨论图论中又一个经典的问题——拓扑排序，并以该问题作为本

章最后的内容。

设有一个有向无环图 (DAG 图), 对其进行拓扑排序即求其中结点的一个拓扑序列, 对于所有的有向边 (U, V) (由 U 指向 V), 在该序列中结点 U 都排列在结点 V 之前。满足该要求的结点序列, 被称为满足拓扑次序的序列。求这个序列的过程, 被称为拓扑排序。

由满足拓扑次序序列的特征我们也能得出其如下特点: 若结点 U 经过若干条



一个满足拓扑次序的序列为
 $\{1, 2, 3, 4, 5\}$

有向边后能够到达结点 V , 则在求得的序列中 U 必排在 V 之前。

在了解了拓扑次序的定义以后, 我们就知道了前文中为什么将拓扑排序限定在一个有向无环图上。若图无向, 则边的两个顶点等价, 不存在先后关系; 若图为有向图, 但存在一个环路, 则该环中所有结点无法判定先后关系 (任意结点间都能通过若干条有向边到达)。

在了解了拓扑排序的相关信息之后, 我们来讨论如何求一个有向无环图的拓扑序列, 即拓扑排序的方法。

首先, 所有有入度 (即以该结点为弧头的弧的个数) 的结点均不可能排在第一个。那么, 我们选择一个入度为 0 的结点, 作为序列的第一个结点。当该结点被选为序列的第一个顶点后, 我们将该点从图中删去, 同时删去以该结点为弧尾的所有边, 得到一个新图。那么这个新图的拓扑序列即为原图的拓扑序列中除去第一个结点后剩余的序列。同样的, 我们在新图上选择一个入度为 0 的结点, 将其作为原图的第二个结点, 并在新图中删去该点以及以该点为弧尾的边。这样我们又得到了一张新图, 重复同样的方法, 直到所有的结点和边都从原图中删去。若在所有结点尚未被删去时即出现了找不到入度为 0 的结点的情况, 则说明剩余的结点形成一个环路, 拓扑排序失败, 原图不存在拓扑序列。

以上即为拓扑排序的方法。

我们来看一个例题, 了解拓扑排序的应用:

例 5.7 Leagal or Not (九度 OJ 1448)

时间限制: 1 秒 内存限制: 128 兆 特殊判题: 否

题目描述:

ACM-DIY is a large QQ group where many excellent acmers get together. It is so harmonious that just like a big family. Every day, many "holy cows" like HH, hh, AC, ZT, lcc, BF, Qinz and so on chat on-line to exchange their ideas. When someone has questions, many warm-hearted cows like Lost will come to help. Then the one being helped will call Lost "master", and Lost will have a nice "prentice". By and by, there are many pairs of "master and prentice". But then problem occurs: there are too many masters and too many prentices, how can we know whether it is legal or not? We all know a master can have many prentices and a prentice may have a lot of masters too, it's legal. Nevertheless, some cows are not so honest, they hold illegal relationship. Take HH and 3xian for instant, HH is 3xian's master and, at the same time, 3xian is HH's master, which is quite illegal! To avoid this, please help us to judge whether their relationship is legal or not. Please note that the "master and prentice" relation is transitive. It means that if A is B's master and B is C's master, then A is C's master.

输入:

The input consists of several test cases. For each case, the first line contains two integers, N (members to be tested) and M (relationships to be tested) ($2 \leq N, M \leq 100$). Then M lines follow, each contains a pair of (x, y) which means x is y's master and y is x's prentice. The input is terminated by N = 0. TO MAKE IT SIMPLE, we give every one a number (0, 1, 2, ..., N-1). We use their numbers instead of their names.

输出:

For each test case, print in one line the judgement of the messy relationship. If it is legal, output "YES", otherwise "NO".

样例输入:

```
3 2
0 1
1 2
2 2
0 1
```

```
1 0
```

```
0 0
```

样例输出：

```
YES
```

```
NO
```

该例大意为，在一个 qq 群里有着许多师徒关系，如 A 是 B 的师父，同时 B 是 A 的徒弟，一个师父可能有许多徒弟，一个徒弟也可能会有许多不同的师父。输入给出该群里所有的师徒关系，问是否存在这样一种非法的情况：以三个人为例，即 A 是 B 的师父，B 是 C 的师父，C 又反过来是 A 的师父。若我们将该群里的所有人都抽象成图上的结点，将所有的师徒关系都抽象成有向边（由师父指向徒弟），该实际问题就转化为一个数学问题——该图上是否存在一个环，即判断该图是否为有向无环图。

无论何时，当需要判断某个图是否属于有向无环图时，我们都需要立刻联想到拓扑排序。若一个图，存在符合拓扑次序的结点序列，则该图为有向无环图；反之，该图为非有向无环图。也就是说，若在该图上拓扑排序成功，该图为有向无环图；反之，则存在环路。

在给出本例的代码之前，我们先来了解标准模板库中又一个标准模板：std::queue，顾名思义，它的用处为建立一个队列并完成相关的操作。使用代码：

```
queue<int> Q;
```

将建立一个保存对象为 int 的队列 Q，其相关操作如下：

Q.push(x);将元素 x 放入对尾；

x = Q.front();读取对头元素，将其值赋值给 x；

Q.pop();对头元素弹出；

Q.empty();判断队列是否为空，若返回值为 true 代表队列为空。

为了保存在拓扑排序中不断出现的和之前已经出现的入度为 0 的结点，我们使用一个队列。每当出现一个入度为 0 的结点，我们将其放入队列；若需要找到一个入度为 0 的结点，就从对头取出。值得一提的是，这里使用队列仅仅为了保存入度为 0 的结点，而与队列先进先出的性质无关，若读者愿意，也可以使用堆栈来保存，这与拓扑排序本身的原理无关。

代码 5.8

```
#include <stdio.h>
#include <vector>
#include <queue>
using namespace std;
```

```

vector<int> edge[501]; //邻接链表, 因为边不存在权值, 只需保存与其邻接的结点编号即可, 所以vector中的元素为int

queue<int> Q; //保存入度为0的结点的队列

int main () {
    int inDegree[501]; //统计每个结点的入度
    int n , m;
    while (scanf ("%d%d", &n, &m) != EOF) {
        if (n == 0 && m == 0) break;
        for (int i = 0; i < n; i++) { //初始化所有结点, 注意本题结点编号由0到n-1
            inDegree[i] = 0; //初始化入度信息, 所有结点入度均为0
            edge[i].clear(); //清空邻接链表
        }
        while(m --) {
            int a , b;
            scanf ("%d%d", &a, &b); //读入一条由a指向b的有向边
            inDegree[b] ++; //又出现了一条弧头指向b的边, 累加结点b的入度
            edge[a].push_back(b); //将b加入a的邻接链表
        }
        while (Q.empty() == false) Q.pop(); //若队列非空, 则一直弹出队头元素, 该操作的目的是清空队列中所有的元素(可能为上一组测试数据中遗留的数据)
        for (int i = 0; i < n; i++) { //统计所有结点的入度
            if (inDegree[i] == 0) Q.push(i); //若结点入度为0, 则将其放入队列
        }
        int cnt = 0; //计数器, 初始值为0, 用于累加已经确定拓扑序列的结点个数
        while (Q.empty() == false) { //当队列中入度为0的结点未被取完时, 重复
            int nowP = Q.front(); //读出队头结点编号, 本例不要求出确定的拓扑序列, 固不做处理; 若要求求出确定的拓扑次序, 则将该结点紧接着放在已经确定的拓扑序列之后
            Q.pop(); //弹出对头元素
            cnt ++; //被确定的结点个数加一
            for (int i = 0; i < edge[nowP].size(); i++) { //将该结点以及以其为弧尾的所有边去除
                inDegree[ edge[nowP][i] ] --; //去除某条边后, 该边所指后继结点入度减一
                if (inDegree[ edge[nowP][i] ] == 0) { //若该结点入度变为0

```

```

        Q.push(edge[nowP][i]); //将其放入队列当中
    }
}
}
if (cnt == n) puts("YES"); //若所有结点都能被确定拓扑序列, 则原图为有向无
环图
else puts("NO"); //否则, 原图为非有向无环图
}
return 0;
}

```

该代码所有结点至多进入队列一次, 但在每个结点被取出时我们都要遍历以其为弧尾的边, 故复杂度为 $O(N+E)$, 其中 N 为结点的个数, E 为边的个数。

如本例所示, 很多涉及拓扑排序的问题都没有直接给出图, 而是将图隐藏在一个实际问题当中, 需要考生自己将实际问题抽象成一个图论问题, 而这恰恰是所有图论题共同的难点, 这种把实际问题抽象出来的能力, 需要独读者在大量训练中慢慢的总结。

练习题: 九度 OJ1449 确定比赛名次; 九度 OJ1450 产生冠军;

总结

本章着重讨论了图论的相关问题, 算法原理及编码。从图的表示法出发, 我们得到了两种图的表示方法: 用二维数组保存的邻接矩阵和用若干条单链表表示的邻接链表, 其中邻接链表可以由 **vector** 进行模拟, 熟悉 **vector** 的各种功能对编码将大有好处。在了解图的表示法之后, 我们学习了解决集合合并与查询问题的并查集。随后讨论了最小生成树、最短路径、拓扑排序三个最基本也是最经典的图论问题。图论在机试中考察难度并不大, 但需要读者对图论基本方法和性质有都很好的掌握。

第6章 搜索

本章继续第二章中查找的相关内容，将查找的范围和方式做一定的扩展，并介绍搜索的相关方法和注意要点。

一 枚举

枚举是最简单也是最直白的搜索方式，它依次尝试搜索空间中所有的解，测试其是否符合条件，若符合则输出答案，否则继续测试下一组解。

但是在使用枚举这种相对较为暴力的算法来进行解题时，我们对其时间复杂度要做特别的关注。枚举问题的时间复杂度往往与需要枚举的情况个数有关，因为我们必须不遗不漏的枚举每一种可能成为答案的情况。所以搜索空间越大，枚举的时间复杂度就越高。所以，我们在对某一问题进行枚举时，必须保证其时间复杂度在题目时限可以接受的范围内。

例 6.1 百鸡问题（九度 OJ 1045）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

用小于等于 n 元去买 100 只鸡，大鸡 5 元/只，小鸡 3 元/只，还有 1/3 元每只的一种小鸡，分别记为 x 只, y 只, z 只。编程求解 x, y, z 所有可能解。

输入：

测试数据有多组，输入 n 。

输出：

对于每组输入,请输出 x, y, z 所有可行解，按照 x, y, z 依次增大的顺序输出。

样例输入：

40

样例输出：

$x=0, y=0, z=100$

$x=0, y=1, z=99$

$x=0, y=2, z=98$

$x=1, y=0, z=99$

来源：

2009 年哈尔滨工业大学计算机研究生机试真题

该例就是非常适合枚举的一例考研机试真题。首先，它需要枚举的情况十分简单，只需简单的枚举 x , y 的值即可，而 z 可由 $100-x-y$ 的值得到。其次，它的枚举量仅有 $100*100$ 数量级，在题目给定的时间范围内可以枚举完毕。所以，对于此类枚举情况不多，非常适合枚举的考题，无需再去考虑另外更具技巧的解法，毫不犹豫的暴力即可。

代码 6.1

```
#include <stdio.h>

int main () {
    int n;
    while (scanf ("%d", &n) != EOF) {
        for (int x = 0; x <= 100; x++) { //枚举x的值
            for (int y = 0; y <= 100 - x; y++) { //枚举y的值, 注意它们的和不可能超过100
                int z = 100 - x - y; //计算z的值
                if (x * 5 * 3 + y * 3 * 3 + z <= n * 3) { //考虑到一只小小鸡的价格为1/3, 为避免除法带来的精度损失, 这里采用了对不等式两端所有数字都乘3的操作, 这也是避免除法的常用技巧
                    printf("x=%d, y=%d, z=%d\n", x, y, z); //输出
                }
            }
        }
    }
    return 0;
}
```

如该例所示，在复杂度允许的范围内，直白的枚举思路简单，代码清晰，所以在一些看似无从下手的题目面前，我们要换个角度，试着从更暴力的角度去思考。

我们回顾第二章讨论过的查找的几个要素：

1. 查找空间。在枚举问题中，所有可能成为答案的解组成了其查找空间。枚举过程即枚举查找空间中的每一个解。在枚举过程中，要做到不遗漏，不重复。

2. 查找目标。即查找到一组符合问题要求的解。为此，我们必须对枚举出来的每一个解进行相关判定。

3. 查找方式。与之前所讨论的查找方式相比，枚举的查找方式依旧比较原始，它简单的依次遍历所有的解，直到得到符合要求的解。

本节讨论枚举的相关知识，为接下来学习搜索做准备。

练习题：九度 OJ1059 abc；九度 OJ1036 Old Bill；

二 广度优先搜索（BFS）

首先需要说明，这里所说的广度优先搜索，与利用广度优先搜索对图进行遍历有一定的差别。广度优先搜索确实可以被应用在图的遍历当中，但其应用远不仅如此。我们通过一个例题，引出广度优先搜索：

例 6.2 胜利大逃亡（九度 OJ 1456）

时间限制：2 秒

内存限制：32 兆

特殊判题：否

题目描述：

Ignatius 被魔王抓走了,有一天魔王出差去了,这可是 Ignatius 逃亡的好机会. 魔王住在一个城堡里,城堡是一个 $A*B*C$ 的立方体,可以被表示成 A 个 $B*C$ 的矩阵,刚开始 Ignatius 被关在 $(0,0,0)$ 的位置,离开城堡的门在 $(A-1,B-1,C-1)$ 的位置,现在知道魔王将在 T 分钟后回到城堡,Ignatius 每分钟能从一个坐标走到相邻的六个坐标中的其中一个.现在给你城堡的地图,请你计算出 Ignatius 能否在魔王回来前离开城堡(只要走到出口就算离开城堡,如果走到出口的时候魔王刚好回来也算逃亡成功),如果可以请输出需要多少分钟才能离开,如果不能则输出-1。

输入：

输入数据的第一行是一个正整数 K ,表明测试数据的数量.每组测试数据的第一行是四个正整数 A,B,C 和 $T(1 \leq A,B,C \leq 50, 1 \leq T \leq 1000)$,它们分别代表城堡的大小和魔王回来的时间.然后是 A 块输入数据(先是第 0 块,然后是第 1 块,第 2 块.....),每块输入数据有 B 行,每行有 C 个正整数,代表迷宫的布局,其中 0 代表路,1 代表墙。

输出：

对于每组测试数据,如果 Ignatius 能够在魔王回来前离开城堡,那么请输出他最少需要多少分钟,否则输出-1.

样例输入：

```
1
3 3 4 20
0 1 1 1
0 0 1 1
0 1 1 1
1 1 1 1
```

```
1 0 0 1
```

```
0 1 1 1
```

```
0 0 0 0
```

```
0 1 1 0
```

```
0 1 1 0
```

样例输出：

```
11
```

在本例中，有一个三维的迷宫，每个迷宫的点都用三维坐标 (x, y, z) 表示。主人公每次行走仅能走到上、下、左、右、前、后与主人公所在点相邻的位置，即从 (x, y, z) 点行走至 $(x-1, y, z)$ 、 $(x+1, y, z)$ 、 $(x, y+1, z)$ 、 $(x, y-1, z)$ 、 $(x, y, z+1)$ 、 $(x, y, z-1)$ 六个点的其中一个。在这其中还存在着一些墙，主人公在任何情况下都不能走到墙所在的位置上。求从点 $(0, 0, 0)$ 走到点 $(A-1, B-1, C-1)$ 最少需要几步。

首先来分析这个看似更加高级的查找问题。

查找空间：该题的查找空间中的元素不再是之前例题中的一个数或者几个数。它由所有从点 $(0, 0, 0)$ 到点 $(A-1, B-1, C-1)$ 合法的行走路径组成。

查找目标：在查找空间中的所有路径中寻找一条最短的路径，即行走步数最少的路径。

查找方法：与以往相比，在广度优先搜索中的查找方法将变的有些特殊，它不再机械地、暴力地遍历查找空间中所有路径，而是采用了某种策略。

在探讨这种特殊的查找方法之前，我们先指明该问题的状态。为了能够更一般的考虑搜索的问题，我们常在搜索问题中指明某种状态，从而使搜索问题变为对状态的搜索。在本问题中，由于要查找从起点到终点的最短耗时，设定状态 (x, y, z, t) 四元组，其中 (x, y, z) 为某个点的坐标， t 为从 $(0,0,0)$ 点走到这个点所耗费的时间。在指明该问题的状态以后，查找的几个相关要素也相应的发生改变。

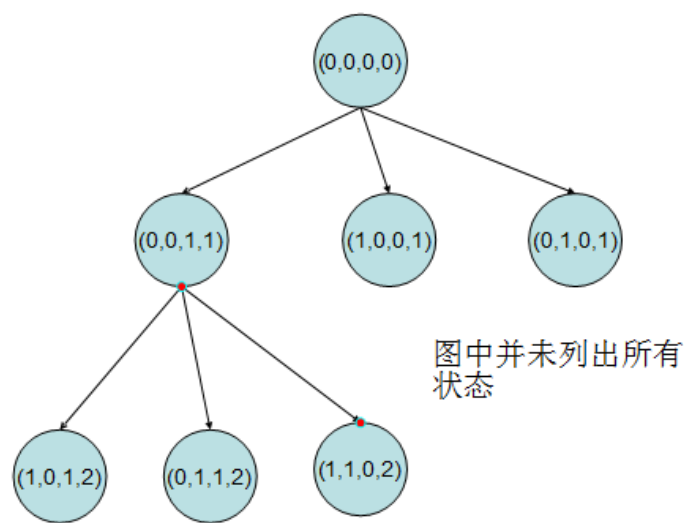
查找空间：也被称为搜索空间，从之前的所有路径变为对所有状态的搜索，即所有可能出现的四元组 (x, y, z, t) 。

查找目标：也被称为搜索目标，即在所有的状态中搜索这样一个四元组 (x, y, z, t) ，其中 x 、 y 、 z 分别等于 $A-1$ 、 $B-1$ 、 $C-1$ ， t 为达到这个状态所需要的最少的时间。

查找方法：在确定状态的定义之前我们并没有讲明其查找方法，这是因为我们实际上并不是在原始查找空间中进行查找的，而是在我们人为定义的状态上进行所需的查找，所以其查找方法是针对状态来定义的。我们将通过状态的扩展转

移来遍历查找所有的状态，下面给出其具体方法。

我们知道，由于可以由任意一个点经过一秒的行走进入下一个点，所以由任意一个状态 (x, y, z, t) 可以扩展得到下面六个状态 $((x-1, y, z, t+1)$ 、 $(x+1, y, z, t+1)$ 、 $(x, y+1, z, t+1)$ 、 $(x, y-1, z, t+1)$ 、 $(x, y, z+1, t+1)$ 、 $(x, y, z-1, t+1)$) 中合法的所有状态，所谓合法即该点不是墙的所在点且该点在立方体范围之内。为了找到到达点 (A-1、B-1、C-1) 的最短时间，我们从初始状态 $(0, 0, 0, 0)$ 开始，按照状态的不断扩展转移查找每一个状态。将初始状态视为根节点，并将每一个状态扩展得到的新状态视为该状态的儿子结点，那么状态的转移与生成就呈现出了树的形态，如下图：



我们将这棵包含搜索空间中所有状态的树称为解答树，我们采用的搜索方法就是在对这棵解答树进行遍历时所使用的遍历方法。

所谓广度优先搜索，即在遍历解答树时使每次状态转移时扩展出尽可能多的新状态，并且按照各个状态出现的先后顺序依次扩展它们。其在解答树上的表现为对解答树的层次遍历，先由根结点扩展出所有深度为 1 的结点，再由每一个深度为 1 的结点扩展出所有深度为 2 的结点，依次类推，且先被扩展的结点其深度不大于后被扩展的结点深度。在这里，深度与所需的行走时间等价。这样，当搜索过程第一次查找到状态中坐标为终点的结点，其记录的时间即为所需最短时间。

但是，即便这样所需查找的状态还是非常得多，最坏情况下，因为每个结点都能扩展出六个新结点，那么仅走了 10 步，其状态数就会达到 6 的十次方，需要大量的时间才能依次遍历完这些状态。那么，我们必须采取相应的措施来制约状态的无限扩展。这个措施被称为剪枝。

剪枝，顾名思义即剪去解答树上不可能存在我们所需答案的子树，从而大大减少所需查找的状态总数。在本例中，我们可以注意到这样一个细节。在起点走向终点的最短路径上，到达任意一个中间结点所用的时间都是起点到达这个结点的最短时间。那么，在搜索过程中，若有状态 (x, y, z, t) ，其中 t 不是从起点到达 (x, y, z) 的最短时间，那么我们所要查找的答案必不可能由该状态进行若干次扩展后得到。在解答树上，即我们所要查找的状态结点，不可能在该状态结点的子树上。有了这个结论，又考虑到广度优先搜索中，先查找到的状态深度必不大于后查找到的状态深度（深度与状态中耗时成正比），所以包含每个立方体中坐标的状态至多被扩展一次。例如，当我们第一次查找到包含点 (x, y, z) 的坐标状态后，其后查找到的任意包含该坐标的状态都不必被扩展，这是因为在后续被查找的状态中，所耗时间 t 必不小于先被查找到的状态。这样，我们限定了每个坐标仅有一个有效状态，所需遍历的状态总数大大降低，在本例中所需遍历的状态总数变为 $A*B*C$ ，完全在我们可以接受的范围内。

明确了查找的方法后，我们来讨论其实现。

首先，使用如下结构体保存每一个状态：

```
struct N {  
    int x , y , z; //位置坐标  
    int t; //所需时间  
};
```

其次，为了实现各个状态按照其被查找到的顺序依次转移扩展，我们需要使用一个队列。即将每次扩展得到的新状态放入队列中，待排在其之前的状态都被扩展完成后，该状态才能得到扩展。

最后，为了防止对无效状态的搜索，我们需要一个标记数组 $mark[x][y][z]$ ，当已经得到过包含坐标 (x, y, z) 的状态后，即把 $mark[x][y][z]$ 置为 $true$ ，当下次再由某状态扩展出包含该坐标的状态时，则直接丢弃，不对其进行任何处理。

在明确了以上三点后，我们给出该题代码。

代码 6.2

```
#include <stdio.h>  
#include <queue>  
using namespace std;  
bool mark[50][50][50]; //标记数组  
int maze[50][50][50]; //保存立方体信息  
struct N { //状态结构体  
    int x , y , z;
```

```

    int t;
};

queue<N> Q; //队列, 队列中的元素为状态

int go[][3] = { //坐标变换数组, 由坐标(x, y, z)扩展得到的新坐标均可通过
(x+go[i][0], y+go[i][1], z+go[i][2])得到
    1, 0, 0,
    -1, 0, 0,
    0, 1, 0,
    0, -1, 0,
    0, 0, 1,
    0, 0, -1
};

int BFS(int a, int b, int c) { //广度优先搜索, 返回其最少耗时
while(Q.empty() == false) { //当队列中仍有元素可以扩展时循环
    N now = Q.front(); //得到队头状态
    Q.pop(); //从队列中弹出队头状态
    for (int i = 0; i < 6; i++) { //依次扩展其六个相邻节点
        int nx = now.x + go[i][0];
        int ny = now.y + go[i][1];
        int nz = now.z + go[i][2]; //计算新坐标
        if (nx < 0 || nx >= a || ny < 0 || ny >= b || nz < 0 || nz >= c)
continue; //若新坐标在立方体外, 则丢弃该坐标
        if (maze[nx][ny][nz] == 1) continue; //若该位置为墙, 则丢弃该坐标
        if (mark[nx][ny][nz] == true) continue; //若包含该坐标的状态已经被
得到过, 则丢弃该状态
        N tmp; //新的状态
        tmp.x = nx;
        tmp.y = ny;
        tmp.z = nz; //新状态包含的坐标
        tmp.t = now.t + 1; //新状态的耗时
        Q.push(tmp); //将该状态放入队列
        mark[nx][ny][nz] = true; //标记该坐标
        if (nx == a - 1 && ny == b - 1 && nz == c - 1) return tmp.t; //
若该坐标即为终点, 可直接返回其耗时
    }
}
}

```

```

    }
}

return -1; //若所有的状态被查找完后, 仍得不到所需坐标, 则返回-1

}

int main () {
    int T;
    scanf ("%d", &T);
    while (T --) {
        int a , b, c , t;
        scanf ("%d%d%d%d", &a, &b, &c, &t); //输入
        for (int i = 0; i < a; i ++ ) {
            for (int j = 0; j < b; j ++ ) {
                for (int k = 0; k < c; k ++ ) {
                    scanf ("%d", &maze[i][j][k]); //输入立方体信息
                    mark[i][j][k] = false; //初始化标记数组
                }
            }
        }
        while(Q.empty() == false) Q.pop(); //清空队列
        mark[0][0][0] = true; //标记起点
        N tmp;
        tmp.t = tmp.y = tmp.x = tmp.z = 0; //初始状态
        Q.push(tmp); //将初始状态放入队列
        int rec = BFS(a, b, c); //广度优先搜索
        if (rec <= t) printf("%d\n", rec); //若所需时间符合条件, 则输出
        else printf("-1\n"); //否则输出-1
    }
    return 0;
}

```

我们通过状态之间的相互扩展完成在所有状态集合中的搜索, 并查找我们需要的状态。利用这种手段, 我们将原本对路径的搜索转化到了对状态的搜索上来。广度优先搜索即对由状态间的相互转移构成的解答树进行的按层次遍历。为了更好的了解状态的含义, 我们来看如下有趣的问题:

例 6.3 非常可乐（九度 OJ 1457）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

大家一定觉的运动以后喝可乐是一件很惬意的事情，但是 seeyou 却不这么认为。因为每次当 seeyou 买了可乐以后，阿牛就要求和 seeyou 一起分享这一瓶可乐，而且一定要喝的和 seeyou 一样多。但 seeyou 的手中只有两个杯子，它们的容量分别是 N 毫升和 M 毫升 可乐的体积为 S ($S < 101$) 毫升(正好装满一瓶)，它们三个之间可以相互倒可乐（都是没有刻度的，且 $S = N + M$, $101 > S > 0$, $N > 0$, $M > 0$ ）。聪明的 ACMER 你们说他们能平分吗？如果能请输出倒可乐的最少的次数，如果不能输出"NO"。

输入：

三个整数： S 可乐的体积， N 和 M 是两个杯子的容量，以"0 0 0"结束。

输出：

如果能平分的话请输出最少要倒的次数，否则输出"NO"。

样例输入：

7 4 3

4 1 3

0 0 0

样例输出：

NO

3

这是一个非常能够说明状态搜索含义的例题。在该例中，题面中丝毫没有涉及到图的概念，也没有给出任何地图模型。那么，它也能进行搜索么？答案是肯定的，搜索的途径即是对状态进行搜索。

与第一例一样，我们使用四元组 (x, y, z, t) 来表示一个状态，其中 x 、 y 、 z 分别表示三个瓶子中的可乐体积， t 表示从初始状态到该状态所需的杯子间互相倾倒的次数。状态间的相互扩展，就是任意四元组经过瓶子间的相互倾倒而得到若干组新的四元组的过程。这样，当平分的状态第一次被搜索出来以后，其状态中表示的杯子倾倒次数即是所求。

同样的，由于要搜索的是最少倒杯子次数，若四元组 (x, y, z, t) 中 t 并不是得到体积组 x 、 y 、 z 的最少倒杯子次数，那么该状态为无效状态，我们将其舍弃。其原因与第一例中一致，在程序中的实现方法也与第一例中一致。

代码 6.3

```
#include <stdio.h>
#include <queue>
using namespace std;
struct N { //状态结构体
    int a , b, c; //每个杯子中可乐的体积
    int t; //得到该体积组倾倒次数
};
queue<N> Q; //队列
bool mark[101][101][101]; //对体积组 (x, y, z) 进行标记, 即只有第一次得到包含
//体积组 (x, y, z) 的状态为有效状态, 其余的舍去
void AtoB (int &a, int sa, int &b, int sb) { //倾倒函数, 由容积为sa的杯子倒往容
//积为sb的杯子, 其中引用参数a和b, 初始时为原始杯子中可乐的体积, 当函数调用完毕后, 为各自杯
//中可乐的新体积
    if (sb - b >= a) { //若a可以全部倒到b中
        b += a;
        a = 0;
    }
    else { //否则
        a -= sb - b;
        b = sb;
    }
}
int BFS(int s, int n, int m) {
    while(Q.empty() == false) { //当队列非空时, 重复循环
        N now = Q.front(); //拿出队头状态
        Q.pop(); //弹出队头状态
        int a , b , c; //a. b. c临时保存三个杯子中可乐体积
        a = now.a;
        b = now.b;
        c = now.c; //读出该状态三个杯子中可乐体积
        AtoB(a, s, b, n); //由a倾倒向b
        if (mark[a][b][c] == false) { //若该体积组尚未出现
            mark[a][b][c] = true; //标记该体积组
```

```

    N tmp;

    tmp.a = a;
    tmp.b = b;
    tmp.c = c;
    tmp.t = now.t + 1; //生成新的状态

    if (a == s / 2 && b == s / 2) return tmp.t;
    if (c == s / 2 && b == s / 2) return tmp.t;
    if (a == s / 2 && c == s / 2) return tmp.t; //若该状态已经为平分
    状态, 则直接返回该状态的耗时

    Q.push(tmp); //否则放入队列
}

a = now.a;
b = now.b;
c = now.c; //重置a. b. c为未倾倒前的体积
AtoB(b, n, a, s); //b倒向a
if (mark[a][b][c] == false) {
    mark[a][b][c] = true;

    N tmp;

    tmp.a = a;
    tmp.b = b;
    tmp.c = c;
    tmp.t = now.t + 1;

    if (a == s / 2 && b == s / 2) return tmp.t;
    if (c == s / 2 && b == s / 2) return tmp.t;
    if (a == s / 2 && c == s / 2) return tmp.t;

    Q.push(tmp);
}

a = now.a;
b = now.b;
c = now.c;
AtoB(a, s, c, m); //a倒向c
if (mark[a][b][c] == false) {
    mark[a][b][c] = true;

    N tmp;

```

```

        tmp.a = a;
        tmp.b = b;
        tmp.c = c;
        tmp.t = now.t + 1;
        if (a == s / 2 && b == s / 2) return tmp.t;
        if (c == s / 2 && b == s / 2) return tmp.t;
        if (a == s / 2 && c == s / 2) return tmp.t;
        Q.push(tmp);
    }
    a = now.a;
    b = now.b;
    c = now.c;
    AtoB(c, m, a, s); //c倒向a
    if (mark[a][b][c] == false) {
        mark[a][b][c] = true;
        N tmp;
        tmp.a = a;
        tmp.b = b;
        tmp.c = c;
        tmp.t = now.t + 1;
        if (a == s / 2 && b == s / 2) return tmp.t;
        if (c == s / 2 && b == s / 2) return tmp.t;
        if (a == s / 2 && c == s / 2) return tmp.t;
        Q.push(tmp);
    }
    a = now.a;
    b = now.b;
    c = now.c;
    AtoB(b, n, c, m); //b倒向c
    if (mark[a][b][c] == false) {
        mark[a][b][c] = true;
        N tmp;
        tmp.a = a;
        tmp.b = b;

```

```

        tmp.c = c;
        tmp.t = now.t + 1;
        if (a == s / 2 && b == s / 2) return tmp.t;
        if (c == s / 2 && b == s / 2) return tmp.t;
        if (a == s / 2 && c == s / 2) return tmp.t;
        Q.push(tmp);
    }
    a = now.a;
    b = now.b;
    c = now.c;
    AtoB(c, m, b, n); //c倒向b
    if (mark[a][b][c] == false) {
        mark[a][b][c] = true;
        N tmp;
        tmp.a = a;
        tmp.b = b;
        tmp.c = c;
        tmp.t = now.t + 1;
        if (a == s / 2 && b == s / 2) return tmp.t;
        if (c == s / 2 && b == s / 2) return tmp.t;
        if (a == s / 2 && c == s / 2) return tmp.t;
        Q.push(tmp);
    }
}
return -1;
}

int main () {
    int s , n , m;
    while (scanf ("%d%d%d", &s, &n, &m) != EOF) {
        if (s == 0) break; //若s为0, 则n. m为0则退出
        if (s % 2 == 1) { //若s为奇数则不可能平分, 直接输出NO
            puts("NO");
            continue;
        }
    }
}

```

```

    for (int i = 0; i <= s; i++) {
        for (int j = 0; j <= n; j++) {
            for (int k = 0; k <= m; k++) {
                mark[i][j][k] = false;
            }
        }
    } //初始化状态
    N tmp;
    tmp.a = s;
    tmp.b = 0;
    tmp.c = 0;
    tmp.t = 0; //初始时状态
    while(Q.empty() == false) Q.pop(); //清空队列中状态
    Q.push(tmp); //将初始状态放入队列
    mark[s][0][0] = true; //标记初始状态
    int rec = BFS(s, n, m); //广度优先搜索
    if (rec == -1) //若为-1输出NO
        puts("NO");
    else printf("%d\n", rec); //否则输出答案
}
return 0;
}

```

可见，与动态规划问题一样，广度优先搜索的关键也是确定状态。只有确定了需要搜索的状态，我们才能更好的进行搜索活动。

同时，广度优先搜索的复杂度也与状态的数量有关。由于我们舍弃了很多无效的状态，那么其时间复杂度与有效状态正相关。如本例中，所有可能出现的状态为 $100*100*100$ 个，即每个体积组对应一个有效状态，所以其复杂度也大致为这个数量级，在进行广搜之前读者要判断其复杂度是否符合要求。

广搜的代码量较之前的例题相比大大增加，读者可能对如此长的代码感到惊慌，但只要掌握其要点，编写广搜代码还是比较容易掌握的。

最后，我们总结广度优先搜索的几个关键字：

- 1.状态。我们确定求解问题中的状态。通过状态的转移扩展，查找遍历所有的状态，从而从中寻找我们需要的答案。

- 2.状态扩展方式。在广度优先搜索中，我们总是尽可能扩展状态，并先扩展

得出的状态先进行下一次扩展。在解答树上的变现为,我们按层次遍历所有状态。

3.有效状态。对有些状态我们并不对其进行再一次扩展,而是直接舍弃它。因为根据问题分析可知,目标状态不会由这些状态经过若干次扩展得到。即目标状态,不可能存在其在解答树上的子树上,所以直接舍弃。

4.队列。为了实现先得出的状态先进行扩展,我们使用队列,将得到的状态依次放入队尾,每次取队头元素进行扩展。

5.标记。为了判断哪些状态是有效的,哪些是无效的我们往往使用标记。

6.有效状态数。问题中的有效状态数与算法的时间复杂度同数量级,所以在进行搜索之前必须估算其是否在我们所可以接受的范围内。

7.最优。广度优先搜索常被用来解决最优值问题,因为其搜索到的状态总是按照某个关键字递增(如前例中的时间和倒杯子次数),这个特性非常适合求解最优值问题。所以一旦问题中出现最少、最短、最优等关键字,我们就要考虑是否是广度优先搜索。

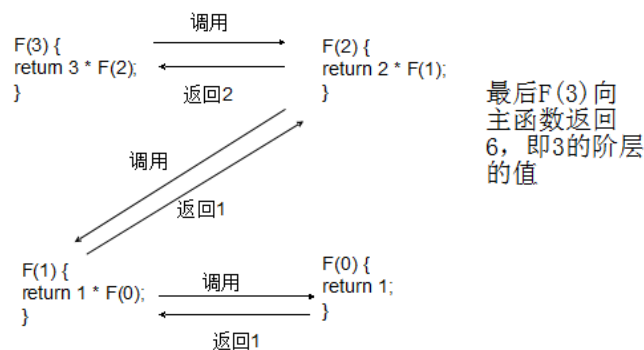
本节介绍了广度优先搜索相关内容,并给出几道练习题。

三 递归

本节讨论递归。递归是一种十分常用的编码技巧,所谓即递归即函数调用函数本身,调用的方式按照问题的不同人为定义,这种调用方式被称为递归方式。同时,为了不使这样的递归无限的发生,我们必须设定递归的出口,即当函数到达某种条件时停止递归。如下求 n 阶层的递归程序:

```
int F(int x) {  
    if (x == 0) return 1;  
    else return x * F(x - 1);  
}
```

其递归方式为,每次递归调用时函数的参数减一,每一个函数在等待递归函数返回时将其返回值与本函数的参数相乘后返回给上一层函数。为了防止这样的递归无限发生,我们设定了一个递归出口,即当参数变为 0 时停止递归调用,直接返回 1。其递归过程如下图:



递归方式和递归出口是递归函数两个重要的要素，只要明确了这两个要素，那么递归函数就比较容易编写了。

我们通过经典的例题，更好的了解递归问题。

例 6.4 汉诺塔 III（九度 OJ 1458）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

约 19 世纪末，在欧洲的商店中出售一种智力玩具，在一块铜板上有三根杆，最左边的杆上自上而下、由小到大顺序串着由 64 个圆盘构成的塔。目的是将最左边杆上的盘全部移到右边的杆上，条件是一次只能移动一个盘，且不允许大盘放在小盘的上面。现在我们改变游戏的玩法，不允许直接从最左(右)边移到最右(左)边(每次移动一定是移到中间杆或从中间移出)，也不允许大盘放到下盘的上面。Daisy 已经做过原来的汉诺塔问题和汉诺塔 II，但碰到这个问题时，她想了很久都不能解决，现在请你帮助她。现在有 N 个圆盘，她至少多少次移动才能把这些圆盘从最左边移到最右边？

输入：

包含多组数据，每次输入一个 N 值($1 \leq N \leq 35$)。

输出：

对于每组数据，输出移动最小的次数。

样例输入：

1
3
12

样例输出：

2
26
531440

汉诺塔问题历来被认为是理解递归最好的例题,凡是涉及递归的教科书其教学用例十有八九会采用汉诺塔,这里我们给出一种汉诺塔问题的变形,旨在使读者对其有更好的理解。

与原始的汉诺塔问题不同,这里对圆盘的移动做了更多的限制,即每次只允许将圆盘移动到中间柱子上或者从中间柱子上移出,而不允许由第一根柱子直接移动圆盘到第三根柱子。

在这种情况下,我们考虑 K 个圆盘的移动情况。为了首先将初始时最底下、最大的圆盘移动到第三根柱子上,我们首先需要将其上的 $K-1$ 个圆盘移动到第三根柱子上,而这恰好等价于移动 $K-1$ 个圆盘从第一根柱子到第三根柱子。当这一移动完成以后,第一根柱子仅剩余最大的圆盘,第二根柱子为空,第三根柱子按顺序摆放着 $K-1$ 个圆盘。我们将最大的圆盘移动到此时没有任何圆盘的中间柱子上,并再次将 $K-1$ 个圆盘从第三根柱子移动到第二根柱子,此时仍然需要移动 $K-1$ 个圆盘从第一根柱子到第三根柱子所需的移动次数(第一根柱子和第三根柱子等价),当这一移动完成以后将最大的圆盘移动到第三根柱子上,最后将 $K-1$ 个圆盘从第一根柱子移动到第三根柱子上。若移动 K 个圆盘从第一根柱子到第三根柱子需要 $F[K]$ 次移动,那么综上所述 $F[K]$ 的组成方式为,先移动 $K-1$ 个圆盘到第三根柱子需要 $F[K-1]$ 次移动,再将最大的圆盘移动到中间柱子需要 1 次移动,然后将 $K-1$ 个圆盘移动回第一根柱子同样需要 $F[K-1]$ 次移动,移动最大的盘子到第三根柱子需要 1 次移动,最后将 $K-1$ 个圆盘也移动到第三根圆盘需要 $F[K-1]$ 次移动,这样 $F[K] = 3 * F[K - 1] + 2$ 。即从第一根柱子移动 K 个圆盘到第三根柱子,需要三次从第一根柱子移动 $K-1$ 个圆盘到第三根柱子,外加三次对最大圆盘的移动。若函数 $F(x)$ 返回移动 x 根子所需要的移动次数,那么其递归方式为 $3 * F(x-1) + 2$ 。

同时我们要确定递归的出口。当 x 为 1 时,即移动一个盘子从第一根柱子移动到第三根柱子,其所需的移动次数是显而易见的,为 2。即当函数的参数为 1 时直接返回 2。

这样我们就确定了解决该问题的递归函数。

代码 6.4

```
#include <stdio.h>
#include <string.h>
Long long F (int num) { //递归函数,返回值较大使用long long类型
    if (num == 1) return 2; //当参数为1时直接返回2
    else
        return 3 * F(num - 1) + 2; //否则递归调用F(num-1)
```



```

}
int main () {
    int n;
    while (scanf ("%d", &n) != EOF) { //输入
        printf("%lld\n", F(n)); //输出答案
    }
    return 0;
}

```

读者可能对如此简单的代码感到惊奇，这正是递归函数的魅力所在。如本例所示，我们并没有关注具体的移动方法，而仅将当前问题与规模更小的问题联系起来，并利用此关系确定递归关系式。为了确定递归的出口，我们必须在问题较小时返回事先计算得到的答案。除此之外，我们不需要做任何额外的工作，递归程序按照我们设定的递归方式“自动”地计算出了答案。

本节主要讨论递归函数的特点和基本要素，下节将以例题的形式了解其更多的用途。

四 递归的应用

在介绍完递归函数的相关信息之后，本节中我们将讨论一些递归函数的应用。使读者对递归用途之广有一定的认识。

例 6.5 Prime ring problem （九度 OJ 1459）

时间限制：2 秒

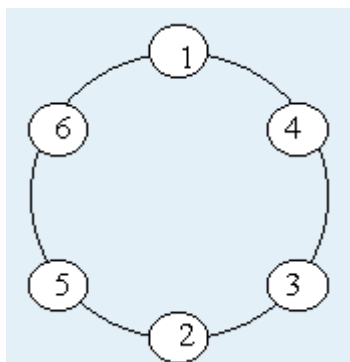
内存限制：32 兆

特殊判题：否

题目描述：

A ring is composed of n circles as shown in diagram. Put natural number 1, 2, ..., n into each circle separately, and the sum of numbers in two adjacent circles should be a prime.

Note: the number of first circle should always be 1.



输入:

$n (1 < n < 17)$.

输出:

The output format is shown as sample below. Each row represents a series of circle numbers in the ring beginning from 1 clockwise and anticlockwisely. The order of numbers must satisfy the above requirements. Print solutions in lexicographical order.

You are to write a program that completes above process.

Print a blank line after each case.

样例输入:

6

8

样例输出:

Case 1:

1 4 3 2 5 6

1 6 5 2 3 4

Case 2:

1 2 3 8 5 6 7 4

1 2 5 8 3 4 7 6

1 4 7 6 5 8 3 2

1 6 7 4 3 8 5 2

题目大意为由给定的 1 到 n 数字中, 将数字依次填入环中, 使得环中任意两个相邻的数字间的和为素数。对于给定的 n , 按字典序由小到大输出所有符合条件的解 (第一个数恒定为 1)。这就是有名的素数环问题。

为了解决该问题, 我们可以采用回溯法枚举每一个值。当第一个数位为 1 确定时, 我们尝试放入第二个数, 使其和 1 的和为素数, 放入后再尝试放入第三个数, 使其与第二个数的和为素数, 直到所有的数全部被放入环中, 且最后一个数与 1 的和也是素数, 那么这个方案即为答案, 输出; 若在尝试放数的过程中, 发现当前位置无论放置任何之前未被使用的数均不可能满足条件, 那么我们回溯改变其上一个数, 直到产生我们所需要的答案, 或者确实不再存在更多的解。

为了实现这一回溯枚举的过程, 我们采用递归的形式:

代码 6.5

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

```

int ans[22]; //保存环中每一个被放入的数
bool hash[22]; //标记之前已经被放入环中的数

int n;

int prime[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41}; //素数, 若需判断一个数
是否为素数则在其之中查找, 因为输入不大于16, 故两数和构成的素数必在该数组内

bool judge(int x){ //判断一个数是否为素数
    for(int i = 0; i < 13; i ++){
        if(prime[i] == x) return true; //在素数数组中查找, 若查找成功则该数为素数
    }
    return false; //否则不是素数
}

void check(){ //检查输出由回溯法枚举得到的解
    if(judge(ans[n] + ans[1]) == false) return; //判断最后一个数与第一个数的和
是否为素数, 若不是则直接返回

    for(int i = 1; i <= n; i ++){ //输出解, 注意最后一个数字后没有空格
        if(i != 1) printf(" ");
        printf("%d", ans[i]);
    }
    printf("\n");
}

void DFS(int num){ //递归枚举, num为当前已经放入环中的数字
    if(num > 1) //当放入的数字大于一个时
        if(judge(ans[num] + ans[num - 1]) == false) return; //判断最后两个数
字的和是否为素数, 若不是则返回继续枚举第num个数

    if(num == n){ //若已经放入了n个数
        check(); //检查输出
        return; //返回, 继续枚举下一组解
    }

    for(int i = 2; i <= n; i ++){ //放入一个数
        if(hash[i] == false){ //若i 还没有被放入环中
            hash[i] = true; //标记i 为已经使用
            ans[num + 1] = i; //将这个数字放入ans数组中
            DFS(num + 1); //继续尝试放入下一个数
            hash[i] = false; //当回溯回枚举该位数字时, 将i 重新标记为未使用
        }
    }
}

```

```

    }
}
}
int main(){
    int cas = 0; //记录Case数
    while(scanf("%d", &n) != EOF){
        cas ++; //Case数递增
        for (int i = 0; i < 22; i ++ ) hash[i] = false; //初始化标记所有数字为未被使用
        ans[1] = 1; //第一个数字恒定为1
        printf("Case %d: \n", cas); //输出Case数
        hash[1] = true; //标记1被使用
        DFS(1); //继续尝试放入下一个数字
        printf("\n"); //输出换行
    }
    return 0;
}
}

```

读者在理解以下代码片段时可能有所困难，下面我们对其做补充说明。

```

hash[i] = true;
ans[num + 1] = i;
DFS(num + 1);
hash[i] = false;

```

在尝试放入第 `num+1` 个数字时，我们依次尝试放入所有在之前位置上未被使用的数字，假设当前 `x` 未被使用，我们将 `x` 放入第 `num+1` 个位置，标记 `x` 为已用，此时环中前 `num+1` 个数字全部确定，依次保存在 `ans[1]` 到 `ans[num+1]` 中，再进行下一个位置的枚举，即递归调用 `DFS`。当调用返回时，意味着当前 `num+1` 个数字确定为 `ans[1]` 到 `ans[num+1]` 中的值时对应的所有可行的答案已经全部处理完毕（可能一个解也不存在），此时我们需要改变 `ans[num + 1]` 的值，从而进行新的答案的搜索。所以，此时 `ans[num + 1]` 的值将不再为已经被枚举过的 `x`，而是一个相异于 `x`，同时又未在之前被使用过的新数字。那么对于后序数字而言，`x` 是未被使用的，是可以被放入后序的任意一个位置的，所以我们重新标记 `x` 为未使用，供后序数位选择。这就是为什么，我们先是标记 `x` 为已经使用，而后又解除该标记的原因。

这是我们要讨论的递归函数第一个已用——回溯法枚举。

递归函数在另一个问题上也具有巨大的优势——图的遍历。

例 6.6 Oil Deposit (九度 OJ 1460)

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

The GeoSurvComp geologic survey company is responsible for detecting underground oil deposits. GeoSurvComp works with one large rectangular region of land at a time, and creates a grid that divides the land into numerous square plots. It then analyzes each plot separately, using sensing equipment to determine whether or not the plot contains oil. A plot containing oil is called a pocket. If two pockets are adjacent, then they are part of the same oil deposit. Oil deposits can be quite large and may contain numerous pockets. Your job is to determine how many different oil deposits are contained in a grid.

输入：

The input file contains one or more grids. Each grid begins with a line containing m and n , the number of rows and columns in the grid, separated by a single space. If $m = 0$ it signals the end of the input; otherwise $1 \leq m \leq 100$ and $1 \leq n \leq 100$. Following this are m lines of n characters each (not counting the end-of-line characters). Each character corresponds to one plot, and is either '*', representing the absence of oil, or '@', representing an oil pocket.

输出：

For each grid, output the number of distinct oil deposits. Two different pockets are part of the same oil deposit if they are adjacent horizontally, vertically, or diagonally. An oil deposit will not contain more than 100 pockets.

样例输入：

```
1 1
*
3 5
*@* @*
** @**
*@* @*
1 8
@ @***** @*
5 5
```

```

****@
*@@*@
*@**@
@@@*@
@@**@
00

```

样例输出：

```

0
1
2
2

```

题目大意：在给定的 $n*m$ 图中，确定有几个@的块。块符合以下条件，其中的任意对@均互相直接或间接连通，两个@直接相邻或者对角相邻即被视为连通。

我们可以这样解决这个问题，首先对图上所有位置均设置一个标记位，表示该位置是否已经被计算过，且该标记仅对地图上为@的点有效。这样我们按从左至右、从上往下的顺序依次遍历地图上所有位置，若遍历到@，且该点未被标记，则所有与其直接相邻、或者间接相邻的@点与其一起组成一个块，该块即为一个我们需要计算的块，将该块中所有的@位置标记为已经计算。这样，当所有的位置被遍历过后，我们即得到了所需的答案。

代码 6.6

```

#include <stdio.h>

char maze[101][101]; //保存地图信息
bool mark[101][101]; //为图上每一个点设立一个状态

int n , m; //地图大小为n*m

int go[][2] = {1, 0, -1, 0, 0, 1, 0, -1,
1, 1, 1, -1, -1, -1, -1, 1}; //八个相邻点与当前位置的坐标差

void DFS(int x,int y) { //递归遍历所有与x,y直接或间接相邻的@
    for (int i = 0;i < 8;i ++){ //遍历八个相邻点
        int nx = x + go[i][0];
        int ny = y + go[i][1]; //计算其坐标
        if (nx < 1 || nx > n || ny < 1 || ny > m) continue; //若该坐标在地图
        外
        if (maze[nx][ny] == '*') continue; //若该位置不是@
    }
}

```

```

        if (mark[nx][ny] == true) continue; //若该位置已经被计算过
        mark[nx][ny] = true; //标记该位置为已经计算
        DFS(nx, ny); //递归查询与该相邻位置直接相邻的点
    }
    return;
}

int main () {
    while (scanf ("%d%d", &n, &m) != EOF) {
        if (n == 0 && m == 0) break;
        for (int i = 1; i <= n; i++) {
            scanf ("%s", maze[i] + 1); //第i行地图信息保存在maze[i][1]到
            maze[i][m]中
        } //按行为单位输入地图信息
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) {
                mark[i][j] = false;
            }
        } //初始化所有位置为未被计算
        int ans = 0; //初始化块计数器
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= m; j++) { //按顺序遍历图中所有位置
                if (mark[i][j] == true) continue; //若该位置已被处理, 跳过
                if (maze[i][j] == '*') continue; //若该位置不为@, 跳过
                DFS(i, j); //递归遍历与其直接或间接相邻的@
                ans++; //答案递增
            }
        }
        printf("%d\n", ans); //输出
    }
    return 0;
}

```

这类将相邻的点联合成一个块的算法有一个专门的名词——Flood Fill, 它常作为某些算法的预处理方式使用。

读者可以联想到, 用之前所讲的广度优先搜索也可以完成此例题。但是递归

的优势就在于其精简的代码，与 **BFS** 动辄百行的代码量相比，递归精巧简练的代码值得我们在机试中使用。

在结束对递归的讨论之前，我们还需要特别的强调，使用递归函数务必注意递归的层数。一个程序可以使用的栈空间是有限的，当递归的过深或者每层递归所需的栈空间太大将会造成栈的溢出，使评判系统返回程序运行时异常终止的结果，一旦你的递归程序出现了这种错误，你就要考虑是否是由递归的太深而造成了爆栈。这是使用递归程序一个很重要的注意要点。具体可使用的栈大小，因个评判系统不同而有所差异，需要读者自行测试后确定。

练习题：九度 OJ1120 全排列；

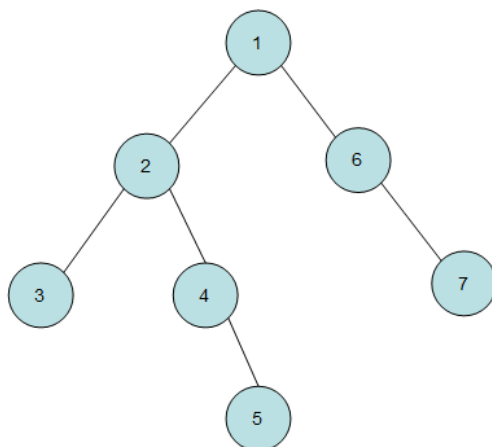
五 深度优先搜索（DFS）

在本章最后，我们讨论深度优先搜索。

首先回顾之前已经介绍过的广度优先搜索。在由状态的转移和扩展构成的解答树中，广度优先搜索按照层次遍历所有的状态，直到找到我们需要的状态。

与其相对的，假如我们改变对解答树的遍历方式，改为优先遍历层次更深的状态，直到遇到一个状态结点，其不再拥有子树，则返回上一层，访问其未被访问过的子树，直至解答树中所有的状态都被遍历完毕。这个过程，类似于树的前序遍历。其遍历顺序如下图所示：

在搜索过程中为了达到这一目的，我们立即扩展新得到的状态，而更早得到的状态则更迟得到扩展。在广度优先搜索中，我们使用队列来实现较早得到的状



态较先得到扩展，这是利用了队列先进先出的特性。那么这里，为了实现先得到的状态后得到扩展的效果，我们按理将使用堆栈保存和扩展搜索过程得到的状态，但是考虑到我们同样可以利用上节所提到的递归程序来实现这一功能，所以

这里给出的深度优先遍历不使用堆栈而是使用递归程序。

由于其缺少了广度搜索中按层次递增顺序遍历的特性。所以当深度优先搜索搜索到我们需要的状态时，其不再具有某种最优的特性。所以，在使用深度优先搜索时，我们更多的求解有或者没有的问题，即对解答树是否有我们需要的答案进行判定，而一般不使用深度优先搜索求解最优解问题。

例 6.7 Temple of the bone (九度 OJ 1461)

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

The doggie found a bone in an ancient maze, which fascinated him a lot. However, when he picked it up, the maze began to shake, and the doggie could feel the ground sinking. He realized that the bone was a trap, and he tried desperately to get out of this maze.

The maze was a rectangle with sizes N by M. There was a door in the maze. At the beginning, the door was closed and it would open at the T-th second for a short period of time (less than 1 second). Therefore the doggie had to arrive at the door on exactly the T-th second. In every second, he could move one block to one of the upper, lower, left and right neighboring blocks. Once he entered a block, the ground of this block would start to sink and disappear in the next second. He could not stay at one block for more than one second, nor could he move into a visited block. Can the poor doggie survive? Please help him.

输入：

The input consists of multiple test cases. The first line of each test case contains three integers N, M, and T ($1 < N, M < 7$; $0 < T < 50$), which denote the sizes of the maze and the time at which the door will open, respectively. The next N lines give the maze layout, with each line containing M characters. A character is one of the following:

'X': a block of wall, which the doggie cannot enter;

'S': the start point of the doggie;

'D': the Door; or

': an empty block.

The input is terminated with three 0's. This test case is not to be processed.

输出：

For each test case, print in one line "YES" if the doggie can survive, or "NO" otherwise.

样例输入：

```
4 4 5
```

```
S.X.
```

```
..X.
```

```
..XD
```

```
....
```

```
3 4 5
```

```
S.X.
```

```
..X.
```

```
...D
```

```
0 0 0
```

样例输出：

```
NO
```

```
YES
```

题目大意：有一个 $N \times M$ 的迷宫，包括起点 S ，终点 D ，墙 X ，和地面，0 秒时主人公从 S 出发，每秒能走到四个与其相邻的位置中的一个，且每个位置被行走之后都不能再次走入，问是否存在这样一条路径使主人公在 T 秒时刚好走到 D 。

在这个问题中，题面不再要求我们求解最优解，而是转而需要我们判定是否存在一条符合条件的路径，所以我们使用深度优先搜索来达到这个目的。

确定状态三元组 (x, y, t) ， (x, y) 为当前点坐标， t 为从起点走到该点所需的时间。我们需要的目标状态为 (dx, dy, T) ，其中 (dx, dy) 为 D 所在点的坐标， T 为所需的时间。初始状态为 $(sx, sy, 0)$ ，其中 (sx, sy) 为 S 所在点的坐标。

同样的，在深度优先搜索中也需要剪枝，我们同样不去理睬被我们判定不可能存在所需状态的子树，以期能够减少所需遍历的状态个数，避免不必要的超时。

这里我们注意到，主人公每走一步时，其所在位置坐标中，只有一个坐标分量发生增一或者减一的改变，那么两个坐标分量和的奇偶性将发生变化。这样，当主人公走过奇数步时，其所在位置坐标和的奇偶性必与原始位置不同；而走过偶数步时，其坐标和的奇偶性与起点保持不变。若起点的坐标和的奇偶性和终点的坐标和不同，但是需要经过偶数秒使其刚好达到，显然的这是不可能的，于是我们直接判定这种情况下，整棵解答树中都不可能存在我们所需的状态，跳过搜索部分，直接输出 **NO**。

代码 6.8

```

#include <stdio.h>

char maze[8][8]; //保存地图信息

int n , m , t; //地图大小为n*m, 从起点到终点能否恰好t秒

bool success; //是否找到所需状态标记

int go[][2] = {1,0,-1,0,0,1,0,-1}; //四方向行走坐标差

void DFS(int x,int y,int time) { //递归形式的深度优先搜索
    for (int i = 0;i < 4;i ++) { //枚举四个相邻位置
        int nx = x + go[i][0];
        int ny = y + go[i][1]; //计算其坐标
        if (nx < 1 || nx > n || ny < 1 || ny > m) continue; //若坐标在地图外
        则跳过
        if (maze[nx][ny] == 'X') continue; //若该位置为墙, 跳过
        if (maze[nx][ny] == 'D') { //若该位置为门
            if (time + 1 == t) { //若所用时间恰好为t
                success = true; //搜索成功
                return; //返回
            }
            else continue; //否则该状态的后续状态不可能为答案(经过的点不能再经过),
            跳过
        }
        maze[nx][ny] = 'X'; //该状态扩展而来的后续状态中, 该位置都不能被经过, 直接修
        改该位置为墙
        DFS(nx,ny,time + 1); //递归扩展该状态, 所用时间递增
        maze[nx][ny] = '.'; //若其后续状态全部遍历完毕, 则退回上层状态, 将因为要搜索
        其后续状态而改成墙的位置, 改回普通位置
        if (success) return; //假如已经成功, 则直接返回, 停止搜索
    }
}

int main () {
    while (scanf ("%d%d%d", &n, &m, &t) != EOF) {
        if (n == 0 && m == 0 && t == 0) break;
        for (int i = 1;i <= n;i ++) {
            scanf ("%s", maze[i] + 1);
        } //输入
    }
}

```

```

    success = false; //初始化成功标记

    int sx , sy;

    for (int i = 1; i <= n; i++) { //寻找D的位置坐标
        for (int j = 1; j <= m; j++) {
            if (maze[i][j] == 'D') {
                sx = i;
                sy = j;
            }
        }
    }

    for (int i = 1; i <= n; i++) { //寻找初始状态
        for (int j = 1; j <= m; j++) {
            if (maze[i][j] == 'S' && (i + j) % 2 == (sx + sy) % 2 + t %
2 ) % 2) { //找到S点后, 先判断S与D的奇偶性关系, 是否和t符合, 即符合上式, 若不符合直接跳
过搜索

                maze[i][j] = 'X'; //将起点标记为墙
                DFS(i, j, 0); //递归扩展初始状态
            }
        }
    }

    puts(success == true ? "YES" : "NO"); //若success为真, 则输出yes
}

return 0;

}

```

如该代码所示, 我们用递归函数完成深度优先搜索。深度优先搜索的各要素如下:

搜索空间: 广度优先搜索相同, 依旧是所有的状态。

搜索目的: 查找一个可以表示原问题解的状态。

搜索方法: 在解答树上进行先序遍历。

总结深度优先搜索的相关特点:

其查找空间和查找目的均与广度优先搜索保持一致, 与广度优先搜索有较大不同的是它的查找方式。深度优先搜索对状态的查找采用了立即扩展新得到的状态的方法, 我们常使用递归函数来完成这一功能。正是由于采用这样的扩展方法, 由它搜索而来的解不再拥有最优解的特性, 所以我们常用它来判断解是否存在的

存在性判定。

至此，我们已经学习了两种既有联系又有区别的搜索方式，在考研机试中，究竟要选择哪一种搜索方式进行搜索，还需考生联系实际考题，对其作出选择。

总结

本章继续查找的话题，首先讨论了关于递归的有关问题。此后，我们将查找空间扩展到状态中，并根据对状态查找的不同方式，介绍了深度优先搜索和广度优先搜索，他们各自具有各自的特点。为了更好的实现深度优先搜索，我们还介绍了递归函数及其相关应用，看到了递归函数的精巧应用。希望读者多加练习两种搜索的方法，一旦学会了搜索，你的机试水平就会明显的上了一个台阶。

第7章 动态规划

本章和下一章将讨论机试中的两个难点，同时也是所有算法中用途最广、变化最多的两大类算法——动态规划(DP)和搜索。

本章将从递推求解出发，了解几个经典的动态规划问题，最后总结动态规划的分析方法，并在实例中进行相关的练习。

一 递推求解

我们来看一个知名的数列——斐波那契数列。这个数列是这样定义的，它的第一个数是 1，第二个数也是 1，其后的每一个数都是前两个数的和，即这样一个数列：1、1、2、3、5、8、13……这样，只要我们确定了这个数列的前两个数，那么后面的每一个数都能经过一次次的累加得到。即，当我们知道这个数列的开头几个数字，并确定它的递推规则，只需通过重复的递推就能得到这个数列的每一个数。

利用递推解决问题，我们就要模仿求斐波那契数列的过程。首先，确定几个规模较小的问题答案。然后考虑如何由这几个规模较小的答案推得后面的答案。一旦有了递推规则和数列初始的几个值，计算机程序就能帮助我们求解数列后面的所有数字，我们的问题也得到了解决。

我们通过一个实例来了解这个过程。

例 7.1 N 阶楼梯上楼问题（九度教程第 93 题）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

N 阶楼梯上楼问题：一次可以走两阶或一阶，问有多少种上楼方式。（要求采用非递归）

输入：

输入包括一个整数 N, ($1 \leq N < 90$)。

输出：

可能有多组测试数据，对于每组数据，输出当楼梯阶数是 N 时的上楼方式个数。

样例输入：

4

样例输出：

来源:

2008 年华中科技大学计算机保研机试真题

这个问题就是一个典型的递推问题。若我们把 N 分别等于 1、2、3……的答案依次排列为一个数列，我们即需求这个数列每一个数的值。我们定义 $f[n]$ 为数列中第 n 个数，同时 $F[n]$ 为台阶总数为 n 时的上台阶方式总数。首先，当数据规模较小时我们可以直接得到答案，如 $F[1] = 1$ ， $F[2] = 2$ 。其次，我们必须确定数列间的递推关系。当 n 大于 2 时，我们考虑每种上台阶方式的最后一步，由于只有两种行走的方法，因此它只可能是从 $n-1$ 阶经过一步走到 n 阶，或者从 $n-2$ 阶经过二步走到 n 阶。我们分别考虑这两种走法，即我们将此时所有的上楼梯方式按照最后一步走法的不同分成两类，分别确定这两类的上楼梯方式数目。经 $n-1$ 阶到达 n 阶，因为其最后一步是确定的，所以其上楼梯方式数量与原问题中到达 $n-1$ 阶的方式数量相同，同为 $F[n-1]$ ；同理，经 $n-2$ 阶到达 n 阶，其上楼梯方式数量与原问题中到达 $n-2$ 阶的方式数量相同，同为 $F[n-2]$ 。这样，我们就确定了达到 n 阶楼梯总的上楼方式个数为 $F[n-1]$ 和 $F[n-2]$ 的和，即 $F[n] = F[n-1] + F[n-2]$ 。这就是这个数列的递推关系。由初始值 $F[1] = 1$ ， $F[2] = 2$ ，我们就能推得所有 $F[n]$ 的值。

代码 7.1

```
#include <stdio.h>

long long F[91]; //F数组保存数列的每一个值，由于数值过大，我们需要使用long long
类型

int main () {
    F[1] = 1;
    F[2] = 2; //数列初始值
    for (int i = 3; i <= 90; i++)
        F[i] = F[i - 1] + F[i - 2]; //递推求得数列的每一个数字
    int n;
    while (scanf ("%d", &n) != EOF) { //输入
        printf("%lld\n", F[n]); //输出相应的数列数字
    }
    return 0;
}
```

由上例的分析过程可以看出，递推求解的重点和难点即确定数列的递推关系，一旦确定了递推关系那么问题就迎刃而解了。我们再来看一道递推关系稍复

杂的例题：

例 7.2 不容易系列之一（九度教程第 94 题）

时间限制：1 秒

内存限制：128 兆

特殊判题：否

题目描述：

大家常常感慨，要做好一件事情真的不容易，确实，失败比成功容易多了！做好“一件”事情尚且不易，若想永远成功而总从不失败，那更是难上加难了，就像花钱总是比挣钱容易的道理一样。话虽这样说，我还是要告诉大家，要想失败到一定程度也是不容易的。比如，我高中的时候，就有一个神奇的女生，在英语考试的时候，竟然把 40 个单项选择题全部做错了！大家都学过概率论，应该知道出现这种情况的概率，所以至今我都觉得这是一件神奇的事情。如果套用一句经典的评语，我们可以这样总结：一个人做错一道选择题并不难，难的是全部做错，一个不对。不幸的是，这种小概率事件又发生了，而且就在我们身边：事情是这样的——HDU 有个网名叫做 8006 的男性同学，结交网友无数，最近该同学玩起了浪漫，同时给 n 个网友每人写了一封信，这都没什么，要命的是，他竟然把所有的信都装错了信封！注意了，是全部装错哟！现在的问题是：请大家帮可怜的 8006 同学计算一下，一共有多少种可能的错误方式呢？

输入：

输入数据包含多个测试实例，每个测试实例占用一行，每行包含一个正整数 n ($1 < n \leq 20$)， n 表示 8006 的网友的人数。

输出：

对于每行输入请输出可能的错误方式的数量，每个实例的输出占用一行。

样例输入：

2

3

样例输出：

1

2

同样的，在该例中我们也容易得到规模较小时的错装方式数量。如 n 为 1 时，数量为 0； n 为 2 时数量为 1。我们按照 n 的取值顺序将所有的错装方式数量排列为一个数列，同样用 $F[n]$ 表示数列里第 n 个数的取值， $F[n]$ 同时代表 n 个信封的错装方式总数，我们确定该数列的递推关系。当 n 大于 3 时，我们考虑 n 封信全部装错的情况。将信封按顺序由 1 到 n 编号。在任意一种错装方案中，假设 n 号信封里装的是 k 号信封的信，而 n 号信封里的信则装在 m 号信封里。我们按

照 k 和 m 的等值与否将总的错装方式分为两类。

若 k 不等于 m ，交换 n 号信封和 m 号信封的信后， n 号信封里装的恰好是对应的信，而 m 号信封中错装 k 号信封里的信，即除 n 号信封外其余 $n-1$ 个信封全部错装，其错装方式等于 $F[n-1]$ ，又由于 m 的 $n-1$ 个可能取值，这类错装方式总数为 $(n-1) * F[n-1]$ 。也可以理解为，在 $n-1$ 个信封错装的 $F[n-1]$ 种方式的基础上，将 n 号信封所装的信与 $n-1$ 个信封中任意一个信封（共有 $n-1$ 中选择）所装的信做交换后，得到所有信封全部错装的方式数。

另一种情况，若 k 等于 m ，交换 n 号信封和 m 号信封的信后， n 号信封和 m 号信封里装的恰好是对应的信，这样除它们之外剩余的 $n-2$ 个信封全部错装，其错装方式为 $F[n-2]$ ，又由于 m 的 $n-1$ 个取值，这类错装方式总数为 $(n-1) * F[n-2]$ 。也可以理解为，在 $n-2$ 个信封全部错装的基础上，交换最后两个信封中的信（ n 号信封和 1 到 $n-1$ 号信封中任意一个，共有 $n-1$ 种选择），使所有的信封全部错装的方式数。

综上所述， $F[n] = (n-1) * F[n-1] + (n-1) * F[n-2]$ 。这就是有名的错排公式。

代码 7.2

```
#include <stdio.h>

Long long F[21]; //数值较大选用long long

int main () {
    F[1] = 0;
    F[2] = 1; //初始值
    for (int i = 3; i <= 20; i++)
        F[i] = (i - 1) * F[i - 1] + (i - 1) * F[i - 2]; //递推求得数列每一个
数字
    int n;
    while (scanf ("%d", &n) != EOF) {
        printf ("%lld\n", F[n]); //输出
    }
    return 0;
}
```

在看过两例递推求解例题后，我们总结其特点。递推求解问题，根据输入的顺序，其答案往往排列成一个数列。为了求得数列中的每一个数字，我们首先得到输入规模较小时的答案，即数列开头的几个数字。分析问题，将每一个问题都分割成规模较小的几个问题，分割过程中要做到不遗漏不重复，并确定它们的关系从而得到递推关系式，利用它求出每一个输入所对应的答案。

练习题：吃糖果(九度教程第 95 题)；

二 最长递增子序列（LIS）

最长递增子序列是动态规划中最经典的问题之一，我们从讨论这个问题开始，循序渐进的了解动态规划的相关知识要点。

在一个已知的序列 $\{a_1, a_2, \dots, a_n\}$ 中，取出若干数组成新的序列 $\{a_{i1}, a_{i2}, \dots, a_{im}\}$ ，其中下标 i_1, i_2, \dots, i_m 保持递增，即新数列中的各个数之间依旧保持原数列中的先后顺序，那么我们称新的序列 $\{a_{i1}, a_{i2}, \dots, a_{im}\}$ 为原序列的一个子序列。若在子序列中，当下标 $i_x > i_y$ 时， $a_{i_x} > a_{i_y}$ ，那么我们称这个子序列为原序列的一个递增子序列。最长递增子序列问题，就是在一个给定的原序列中，求得其最长递增子序列长度。

有序列 $\{a_1, a_2, \dots, a_n\}$ ，我们求其最长递增子序列长度。按照递推求解的思想，我们用 $F[i]$ 代表若递增子序列以 a_i 结束时它的最长长度。当 i 较小，我们容易直接得出其值，如 $F[1] = 1$ 。那么，如何由已经求得的 $F[i]$ 值推得后面的值呢？假设， $F[1]$ 到 $F[x-1]$ 的值都已经确定，注意到，以 a_x 结尾的递增子序列，除了长度为 1 的情况，其它情况中， a_x 都是紧跟在一个由 $a_i (i < x)$ 组成递增子序列之后。要求以 a_x 结尾的最长递增子序列长度，我们依次比较 a_x 与其之前所有的 $a_i (i < x)$ ，若 a_i 小于 a_x ，则说明 a_x 可以跟在以 a_i 结尾的递增子序列之后，形成一个新的递增子序列。又因为以 a_i 结尾的递增子序列最长长度已经求得，那么在这种情况下，由以 a_i 结尾的最长递增子序列再加上 a_x 得到的新的序列，其长度也可以确定，取所有这些长度的最大值，我们即能得到 $F[x]$ 的值。特殊的，当没有 $a_i (i < x)$ 小于 a_x ，那么以 a_x 结尾的递增子序列最长长度为 1。即 $F[x] = \max\{1, F[i] + 1 \mid a_i < a_x \ \& \ i < x\}$ ；

我们给出求序列 $\{1, 4, 3, 2, 6, 5\}$ 的最长递增子序列长度的所有 $F[i]$ 供读者参考。

$F[1]$ (1)	$F[2]$ (4)	$F[3]$ (3)	$F[4]$ (2)	$F[5]$ (6)	$F[6]$ (5)
1	2	2	2	3	3

总结一下，求最长递增子序列的递推公式为：

$$F[1] = 1;$$

$$F[i] = \max\{1, F[j] + 1 \mid a_j < a_i \ \& \ j < i\};$$

例 7.3 拦截导弹（九度教程第 95 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

某国为了防御敌国的导弹袭击，开发出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭，并观测到导弹依次飞来的高度，请计算这套系统最多能拦截多少导弹。拦截来袭导弹时，必须按来袭导弹袭击的时间顺序，不允许先拦截后面的导弹，再拦截前面的导弹。

输入：

每组输入有两行，
第一行，输入雷达捕捉到的敌国导弹的数量 k ($k \leq 25$)，
第二行，输入 k 个正整数，表示 k 枚导弹的高度，按来袭导弹的袭击时间顺序给出，以空格分隔。

输出：

每组输出只有一行，包含一个整数，表示最多能拦截多少枚导弹。

样例输入：

8
300 207 155 300 299 170 158 65

样例输出：

6

来源：

2007 年北京大学计算机研究生机试真题

由题意不难看出，要求最多能够拦截多少枚导弹，即在按照袭击顺序排列的导弹高度中求其最长不增子序列。所谓不增子序列，即子序列中排在前面的数字不比排在后面的数字小。求最长不增子序列的原理与求最长递增子序列的原理完全一致，只是递推关系相应的发生一些变化，递推关系如下：

$$F[1] = 1;$$

$$F[i] = \max \{1, F[j] + 1 \mid j < i \text{ \& \& } a_j \geq a_i\};$$

代码 7.3

```
#include <stdio.h>

int max(int a, int b) {return a > b ? a : b;} //取最大值函数

int list[26]; //按袭击事件顺序保存各导弹高度

int dp[26]; //dp[i] 保存以第i 个导弹结尾的最长不增子序列长度

int main() {

    int n;

    while (scanf ("%d", &n) != EOF) {
```

```

    for (int i = 1; i <= n; i++) {
        scanf ("%d", &list[i]);
    } //输入

    for (int i = 1; i <= n; i++) { //按照袭击时间顺序确定每一个dp[i]
        int tmax = 1; //最大值的初始值为1, 即以其结尾的最长不增子序列长度至少为
1
        for (int j = 1; j < i; j++) { //遍历其前所有导弹高度
            if (list[j] >= list[i]) { //若j号导弹不比当前导弹低
                tmax = max(tmax, dp[j] + 1); //将当前导弹排列在以j号导弹结尾
的最长不增子序列之后, 计算其长度dp[j] + 1, 若大于当前最大值, 则更新最大值
            }
        }
        dp[i] = tmax; //将dp[i]保存为最大值
    }

    int ans = 1;
    for (int i = 1; i <= n; i++) {
        ans = max(ans, dp[i]);
    } //找到以每一个元素结尾的最长不增子序列中的最大值, 该最大值即为答案
    printf("%d\n", ans); //输出
}
return 0;
}

```

其时间复杂度为 $O(n \cdot n)$, 空间复杂度为 $O(n)$ 。

由上例的分析过程可知, 最长递增子序列问题的求解思想, 不仅可以用来解决单纯的最长递增子序列问题, 也可以经过类比来求解其它大小关系确定的子序列最长长度。

最长递增子序列问题, 是我们接触的第一个真正意义上的动态规划问题。我们来回顾它的特点。首先, 我们将这个问题分割成许多子问题, 每个子问题为确定以第 i 个数字结束的递增子序列最长长度。其次, 这些子问题之间存在某种联系, 以任意一个数字结束的递增子序列长度, 与以排在该数字之前所有比它小的元素结尾的最长递增子序列长度有关, 且仅与其数字量有关, 而与其具体排列无关。最后, 规模较小的子问题是容易被我们确定的。于是, 我们像递推求解一样, 一步步求得每个子问题的答案, 进而由这些答案推得后续子问题的答案。

本节介绍最长递增子序列的求法, 在向读者展示其算法的同时, 旨在使读者

对动态规划的特点有一个直观的了解。

练习题：合唱队形(九度教程第 97 题) (提示：正反两次运用 LIS)；

三 最长公共子序列 (LCS)

本章介绍另一个经典的动态规划问题，最长公共子序列。

与上节中数字序列的子序列定义相同，在字符串 S 中按照其先后顺序依次取出若干个字符，并将它们排列成一个新的字符串，这个字符串就被称为原字符串的子串。

有两个字符串 $S1$ 和 $S2$ ，求一个最长公共子串，即求字符串 $S3$ ，它同时为 $S1$ 和 $S2$ 的子串，且要求它的长度最长，并确定这个长度。这个问题被我们称为最长公共子序列问题。

与求最长递增子序列一样，我们首先将原问题分割成一些子问题，我们用 $dp[i][j]$ 表示 $S1$ 中前 i 个字符与 $S2$ 中前 j 个字符分别组成的两个前缀字符串的最长公共子串长度。显然的，当 i, j 较小时我们可以直接得出答案，如 $dp[0][j]$ 必等于 0。那么，假设我们已经求得 $dp[i][j](0 \leq i < x, 0 \leq j < y)$ 的所有值，考虑如何由这些值继而推得 $dp[x][y]$ ，求得 $S1$ 前 x 个字符组成的前缀子串和 $S2$ 前 y 个字符组成的前缀子串的最长公共子序列长度。若 $S1[x] = S2[y]$ ，即 $S1$ 中的第 x 个字符和 $S2$ 中的第 y 个字符相同，同时由于他们都是各自前缀子串的最后一个字符，那么必存在一个最长公共子串以 $S1[x]$ 或 $S2[y]$ 结尾，其它部分等价于 $S1$ 中前 $x-1$ 个字符和 $S2$ 中前 $y-1$ 个字符的最长公共子串。所以这个子串的长度比 $dp[x-1][y-1]$ 又增加 1，即 $dp[x][y] = dp[x-1][y-1] + 1$ 。相反的，若 $S1[x] \neq S2[y]$ ，此时其最长公共子串长度为 $S1$ 中前 $x-1$ 个字符和 $S2$ 中前 y 个字符的最长公共子串长度与 $S1$ 中前 x 个字符和 $S2$ 中前 $y-1$ 个字符的最长公共子串长度的较大者，即在两种情况下得到的最长公共子串都不会因为其中一个字符串又增加了一个字符长度发生改变。综上所述， $dp[x][y] = \max\{dp[x-1][y], dp[x][y-1]\}$ 。

总结一下，最长公共子序列问题的递推条件：

假设有两个字符串 $S1$ 和 $S2$ ，其中 $S1$ 长度为 n ， $S2$ 长度为 m ，用 $dp[i][j]$ 表示 $S1$ 前 i 个字符组成的前缀子串与 $S2$ 前 j 个字符组成的前缀子串的最长公共子串长度，那么：

$$dp[0][j](0 \leq j \leq m) = 0;$$

$$dp[i][0](0 \leq i \leq n) = 0;$$

$$dp[i][j] = dp[i-1][j-1] + 1; (S1[i] == S2[j])$$

$$dp[i][j] = \max\{dp[i-1][j], dp[i][j-1]\}; (S1[i] \neq S2[j])$$

由这样的递推公式和显而易见的初始值，我们即能依次求得各 $dp[i][j]$ 的值，最终 $dp[n][m]$ 中保存的值即为两个原始字符串的最长公共子序列长度。

例 7.4 Coincidence （九度教程第 98 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

Find a longest common subsequence of two strings.

输入：

First and second line of each input case contain two strings of lowercase character a...z. There are no spaces before, inside or after the strings. Lengths of strings do not exceed 100.

输出：

For each case, output k – the length of a longest common subsequence in one line.

样例输入：

abcd

cxbydz

样例输出：

2

来源：

2008 年上海交通大学计算机研究生机试真题

该例即为最朴素的求解两字符串的最长公共子串长度。

代码 7.4

```
#include <stdio.h>
#include <string.h>
int dp[101][101];
int max(int a, int b) {return a > b ? a : b;} //取最大值函数
int main () {
    char S1[101], S2[101];
    while (scanf ("%s%s", S1, S2) != EOF) { //输入
        int L1 = strlen(S1);
        int L2 = strlen(S2); //依次求得两个字符串的长度
        for (int i = 0; i <= L1; i++) dp[i][0] = 0;
        for (int j = 0; j <= L2; j++) dp[0][j] = 0; //初始值
```

```

        for (int i = 1; i <= L1; i++) {
            for (int j = 1; j <= L2; j++) { //二重循环依次求得每个dp[i][j]值
                if (S1[i - 1] != S2[j - 1]) //因为字符串数组下标从0开始，所以第i
                个字符位置为S1[i - 1]，若当前两个字符不相等
                    dp[i][j] = max(dp[i][j - 1], dp[i - 1][j]); //dp[i][j]为
                dp[i][j - 1] 和 dp[i - 1][j]中较大的一个
                else dp[i][j] = dp[i - 1][j - 1] + 1; //若它们相等，则dp[i][j]
                比dp[i - 1][j - 1]再加一
            }
        }

        printf("%d\n", dp[L1][L2]); //输出答案
    }
    return 0;
}

```

其时空复杂度都是 $O(L1 * L2)$ ，其中 $L1$ 和 $L2$ 分别为两个字符串的长度。

本节介绍了动态规划中又一个经典的问题——最长公共子序列问题，读者在牢记其算法原理的基础上还应着重考量算法分析的过程。在下一节中，我们将着重讨论解动态规划问题的一般思路和复杂度估计。

四 状态与状态转移方程

在之前的两节中，我们已经讨论了两类较为经典的动态规划问题的解法，本节将对两种算法进行总结，并探讨解动态规划问题的统一思路。

回顾两种经典问题的算法模式，我们都先定义了一个数字量，如最长递增子序列中用 $dp[i]$ 表示以序列中第 i 个数字结尾的最长递增子序列长度和最长公共子序列中用 $dp[i][j]$ 表示的两个字符串中前 i 、 j 个字符的最长公共子序列，我们就是通过对这两个数字量的不断求解最终得到答案的。这个数字量就被我们称为状态。状态是描述问题当前状况的一个数字量。首先，它是数字的，是可以被抽象出来保存在内存中的。其次，它可以完全的表示一个状态的特征，而不需要其他任何的辅助信息。最后，也是状态最重要的特点，状态间的转移完全依赖于各个状态本身，如最长递增子序列中， $dp[x]$ 的值由 $dp[i] (i < x)$ 的值确定。若我们在分析动态规划问题的时候能够找到这样一个符合以上所有条件的状态，那么多半这个问题是可以被正确解出的。这就是为什么人们常说，做 DP 题的关键，就是寻找一个好的状态。

我们将注意力放到状态的递推过程中来,由一个或多个老的状态得出一个新的状态的过程,被称为状态的转移。如最长公共子序列中,通过 $dp[i - 1][j - 1]$ 或 $dp[i][j - 1]$ 、 $dp[j - 1][i]$ 的值转移得出 $dp[i][j]$ 的值就是该问题中的状态转移。而之前我们所说的数字量间的递推关系就被我们称为状态的转移规则,也被称为状态转移方程,确定状态的转移规则即确定了怎样由前序状态递推求出后续状态。

如最长递增子序列问题中的状态转移方程为:

$$F[1] = 1;$$

$$F[i] = \max\{1, F[j] + 1 \mid j < i \ \& \ a_j \geq a_i\};$$

最后我们来讨论,动态规划问题的求解中,相关时间复杂度的估计。

以最长公共子序列为例,设两个字符串长度分别为 $L1$ 和 $L2$,则共有 $L1 * L2$ 个状态要求解,为了求解每个状态,我们按照相应字符是否相等选取 $dp[i - 1][j - 1] + 1$ 或者 $\max\{dp[i][j - 1], dp[i - 1][j]\}$ 为 $dp[i][j]$ 的值,即状态转移过程中每个状态的得出仅需要 $O(1)$ 的时间复杂度,所以总的时间复杂度为 $O(L1 * L2 * 1)$ 。

通过以上分析,同样的我们也可以判断出求解最长递增子序列问题的时间复杂度构成:假设原数列长度为 n ,则状态数量为 $dp[n]$,状态转移过程中每个状态的得出复杂度平均为 $O(n)$,所以其总的时间复杂度为 $O(n * n)$ 。

总结一下,动态规划问题的时间复杂度由两部分组成:状态数量和状态转移复杂度,往往程序总的复杂度为它们的乘积。

所以说,选择一个好的状态不仅关系到程序的正确性,同时对算法的复杂度也有较大的影响,所以它在动态规划问题中有着举足轻重的地位。

相对来说空间复杂度则没那么容易确定,我们可能需要额外的内存空间来辅助状态的确定和转移,也可能通过某种内存的复用提高内存的利用率。

本节主要讨论状态和状态转移的相关概念以及对复杂度的影响,固不安排练习题,下节将举例说明它们的重要性。

五 动态规划问题分析举例

本节通过对一般动态规划问题的求解举例,旨在使读者对状态和状态转移有一定的了解。

例 7.5 搬寝室 (九度教程第 99 题)

时间限制: 1 秒

内存限制: 32 兆

特殊判题: 否

题目描述:

搬寝室是很累的,xhd深有体会.时间追述2006年7月9号,那天xhd迫于无奈要从27号楼搬到3号楼,因为10号要封楼了.看着寝室里的n件物品,xhd开始发呆,因为n是一个小于2000的整数,实在是太多了,于是xhd决定随便搬2*k件过去就行了.但还是会很累,因为2*k也不小是一个不大于n的整数.幸运的是xhd根据多年的搬东西的经验发现每搬一次的疲劳度是和左右手的物品的重量差的平方成正比(这里补充一句,xhd每次搬两件东西,左手一件右手一件).例如xhd左手拿重量为3的物品,右手拿重量为6的物品,则他搬完这次的疲劳度为 $(6-3)^2 = 9$.现在可怜的xhd希望知道搬完这2*k件物品后的最佳状态是怎样的(也就是最低的疲劳度),请告诉他吧.

输入:

每组输入数据有两行,第一行有两个数n,k($2 \leq 2*k \leq n < 2000$).第二行有n个整数分别表示n件物品的重量(重量是一个小于 2^{15} 的正整数).

输出:

对应每组输入数据,输出数据只有一个表示他的最少的疲劳度,每个一行.

样例输入:

2 1

1 3

样例输出:

4

注意到任意选择一对物品,其累积的疲劳度为两个物品重量差的平方,在所有被选择的物品中任选两对,假设四个物品重量分别为a、b、c、d($a \leq b \leq c \leq d$),此时可能存在两种配对方案。若a、b为一组,则疲劳度为

$$\begin{aligned} & (a-b)*(a-b) + (c-d)*(c-d) \\ & = a^2 + b^2 + c^2 + d^2 - 2(ab + cd) \end{aligned}$$

在另一种配对方案中a、c为一组,则疲劳度为

$$\begin{aligned} & (a-c)*(a-c) + (b-d)*(b-d) \\ & = a^2 + b^2 + c^2 + d^2 - 2(ac + bd) \end{aligned}$$

两式相减得

$$\begin{aligned}
& (a-b)*(a-b)+(c-d)*(c-d) \\
& - (a-c)*(a-c)+(b-d)*(b-d) \\
& = 2(ac+bd)-2(ab+cd) \\
& = 2(a(c-b)+d(b-c)) \\
& = 2(c-b)(a-d)
\end{aligned}$$

因为 $c \geq b$ 而 $d \geq a$ ，所以，两式相减的差必为非正数。即，第一种方案中的疲劳度必不大于第二种配对方案中的疲劳度。由该分析可知，在选定的最优方案中，任选两对组合，其配对情况必为重量最大物品和重量次大物品为一对、重量最小物品和重量次小物品为一对，不存在交叉组合的情况。同时，当两个物品的重量差越小时，其差的平方也越小，所以每一对组合的两个物品重量，必为原物品中重量相邻的两个物品。

综上所述。在选定的最优方案中，每对物品都是重量相邻的一对物品。

在得出这个结论后，我们设计描述该问题的状态。首先将所有物品按照重量递增排序，并由 1 到 n 编号。设 $dp[i][j]$ 为在前 j 件物品中选择 i 对物品时最小的疲劳度，那么根据物品 j 和物品 $j-1$ 是否被配对选择，该状态有两个来源：若物品 j 和物品 $j-1$ 未被配对，则物品 j 一定没被选择，所以 $dp[i][j]$ 等价于 $dp[i][j-1]$ ；若物品 j 和物品 $j-1$ 配对，则 $dp[i][j]$ 为 $dp[i-1][j-2]$ 再加上这两件物品配对后产生的疲劳度，即前 $j-2$ 件物品配成的 $i-1$ 对再加上最后两件配成的一对物品，共得到 i 对物品。

初始时， $dp[0][i](1 \leq i \leq n)$ 为 0，即不选择任何一对物品时，疲劳度为 0。

综上所述，其状态转移方程为（设经过排序后第 i 件物品重量为 $list[i]$ ）：

$$dp[i][j] = \min\{dp[i][j-1], dp[i-1][j-2] + (list[j] - list[j-1])^2\}$$

递推求出 $dp[k][n]$ 即是所求。

最后，估计这种状态对应的复杂度。状态数量为 $k*n$ ，转移时间复杂度为 $O(1)$ ，综合时间复杂度为 $O(k*n)$ ，参考题面中给定的输入取值范围，该复杂度在我们可以接受的范围内。

代码 7.5

```

#include <stdio.h>
#include <algorithm>
using namespace std;

#define INF 0x7fffffff //预定义最大的int取值为无穷

```

```

int list[2001]; //保存每个物品重量
int dp[1001][2001]; //保存每个状态

int main () {
    int n , k;
    while (scanf ("%d%d", &n, &k) != EOF) {
        for (int i = 1; i <= n; i++) {
            scanf ("%d", &list[i]);
        } //输入

        sort(list + 1, list + 1 + n); //使所有物品按照重量递增排序
        for (int i = 1; i <= n; i++) //初始值
            dp[0][i] = 0;

        for (int i = 1; i <= k; i++) { //递推求得每个状态
            for (int j = 2 * i; j <= n; j++) {
                if (j > 2 * i) //若j > 2*i 则表明, 最后两个物品可以不配对, 即前j - 1件
                物品足够配成i 对, dp[i][j]可以由dp[i][j - 1]转移而来, 其值先被设置为dp[i][j - 1]
                    dp[i][j] = dp[i][j - 1];
                else
                    dp[i][j] = INF; //若j == 2 * i, 说明最后两件物品必须配对, 否则
                    前j 件物品配不成i 对, 所以其状态不能由dp[i][j - 1]转移而来, dp[i][j]先设置为正无穷
                    if (dp[i][j] > dp[i - 1][j - 2] + (list[j] - list[j - 1]) *
                    (list[j] - list[j - 1])) //若dp[i][j]从dp[i - 1][j - 2]转移而来时, 其值优于之前确定
                    的正无穷或者由dp[i][j - 1]转移而来的值时, 更新该状态
                        dp[i][j] = dp[i - 1][j - 2] + (list[j] - list[j - 1]) *
                        (list[j] - list[j - 1]); //更新
            }
        }

        printf("%d\n", dp[k][n]); //输出
    }
    return 0;
}

```

与该例一样, 某些动态规划问题看似无从下手, 但在得出某个结论后问题就逐渐明朗了, 这也是一类动态规划问题的特点, 只有先得到这个结论, 才能设计出合适的状态以及状态转移方程。

例 7.6 Greedy Tino (九度教程第 100 题)

时间限制: 1 秒

内存限制: 32 兆

特殊判题: 否

题目描述:

Tino wrote a long long story. BUT! in Chinese. So I have to tell you the problem directly and discard his long long story. That is tino want to carry some oranges with "Carrying pole", and he must make two side of the Carrying pole are the same weight. Each orange have its' weight. So greedy tino want to know the maximum weight he can carry.

输入:

The first line of input contains a number t , which means there are t cases of the test data. for each test case, the first line contain a number n , indicate the number of oranges. the second line contains n numbers, W_i , indicate the weight of each orange. n is between 1 and 100, inclusive. W_i is between 0 and 2000, inclusive. the sum of W_i is equal or less than 2000.

输出:

For each test case, output the maximum weight in one side of Carrying pole. If you can't carry any orange, output -1. Output format is shown in Sample Output.

样例输入:

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

样例输出:

```
Case 1: 7
```

本题大意: 有一堆柑橘, 重量为 0 到 2000, 总重量不大于 2000。要求我们从中取出两堆放在扁担的两头且两头的重量相等, 问符合条件的每堆重量最大为多少。没有符合条件的分堆方式则输出 -1。

在求解该问题之前, 我们先关注本题的输入特点。与以往我们讨论过的问题不同, 该例在输入中将预先告诉我们输入的测试数据个数, 即整数 T 。所以我们的程序只需要准确的处理 T 组数据即可, 以免造成不必要的错误。

首先, 我们只考虑柑橘重量为非 0 的情况。

因为本题要求解的是重量相等的两堆柑橘中每堆的最大重量, 并且在堆放过程中, 由于新的柑橘被加到第一堆或者第二堆, 两堆之间的重量差会动态发生改变, 所以我们设状态 $dp[i][j]$ 表示前 i 个柑橘被选择后 (每个柑橘可能放到第一堆

或者第二堆)后,第一堆比第二堆重 j 时(当 j 为负时表示第二堆比第一堆重),两堆的最大总重量和。

初始时, $dp[0][0]$ 为 0,即不往两堆中加任何柑橘时,两堆最大总重量为 0; $dp[0][j]$ (j 不等于 0) 为负无穷,即其它状态都不存在。

根据每一个新加入的柑橘被加入到第一堆或者第二堆或者不加入到任何一堆,设当前加入柑橘重量为 $list[i]$,这将造成第一堆与第二堆的重量差增大 $list[i]$ 或减小 $list[i]$ 或者不变,我们在它们之中取最大值,其状态转移为:

$$dp[i][j] = \max(dp[i-1][j-list[i]]+list[i], dp[i-1][j+list[i]]+list[i], dp[i-1][j]);$$

当根据该状态转移方程求出所有的状态后,状态 $dp[n][0]/2$ 即是所求。

我们再来考虑柑橘重量包含 0 的情况,当在不考虑柑橘重量为 0,推得 $dp[n][0]$ 为正数时,柑橘重量为 0 的柑橘将不对答案造成任何影响,固在这种情况下可直接排除重量为 0 的柑橘。当在不考虑柑橘重量为 0,推得 $dp[n][0]$ 为 0 时,即不存在任何非 0 的组合使两堆重量相等。此时,若存在重量为 0 的柑橘,则可组成两堆重量为 0 的柑橘(至少有一个柑橘重量为 0),它们重量相等;否则,将不存在任何分堆方式,输出 -1。

最后,分析其复杂度。由于柑橘总重量不大于 2000,所以总的状态数量为柑橘总数 $n*2*2000$,状态转移为 $O(1)$ 复杂度,所以综合时间复杂度为 $O(4000*m)$,在我们可以接收的范围内。

代码 7.6

```
#include <stdio.h>

#define OFFSET 2000 //因为柑橘重量差存在负数的情况,即第一堆比第二堆轻,所以在计算重量差对应的数组下标时加上该偏移值,使每个重量差对应合法的数组下标

int dp[101][4001]; //保存状态
int list[101]; //保存柑橘数量
#define INF 0x7fffffff //无穷

int main () {
    int T;
    int cas = 0; //处理的Case数,以便输出
    scanf ("%d", &T); //输入要处理的数据组数
    while (T -- != 0) { //T次循环
        int n;
        scanf ("%d", &n);
```

```

bool HaveZero = false; //统计是否存在重量为0的柑橘
int cnt = 0; //计数器, 记录共有多少个重量非零的柑橘
for (int i = 1; i <= n; i++) { //输入n个柑橘重量
    scanf ("%d", &list[++ cnt]);

    if (list[cnt] == 0) { //若当前输入柑橘重量为0
        cnt--; //去除这个柑橘

        HaveZero = true; //并记录存在重量为0的柑橘
    }
}

n = cnt;

for (int i = -2000; i <= 2000; i++) {
    dp[0][i + OFFSET] = -INF;
} //初始化, 所有dp[0][i]为负无穷。注意要对重量差加上OFFSET后读取或调用
dp[0][0 + OFFSET] = 0; //dp[0][0]为0

for (int i = 1; i <= n; i++) { //遍历每个柑橘
    for (int j = -2000; j <= 2000; j++) { //遍历每种可能的重量差
        int tmp1 = -INF, tmp2 = -INF; //分别记录当前柑橘放在第一堆或第二
堆时转移得来的新值, 若无法转移则为-INF

        if (j + list[i] <= 2000 && dp[i - 1][j + list[i] + OFFSET] !=
-INF) { //当状态可以由放在第一堆转移而来时
            tmp1 = dp[i - 1][j + list[i] + OFFSET] + list[i]; //记
录转移值
        }

        if (j - list[i] >= -2000 && dp[i - 1][j - list[i] + OFFSET] !=
-INF) { //当状态可以由放在第二堆转移而来时
            tmp2 = dp[i - 1][j - list[i] + OFFSET] + list[i]; //记
录该转移值
        }

        if (tmp1 < tmp2) {
            tmp1 = tmp2;
        } //取两者中较大的那个, 保存至tmp1

        if (tmp1 < dp[i - 1][j + OFFSET]) { //将tmp1与当前柑橘不放入任
何堆即状态差不发生改变的原状态值比较, 取较大的值保存至tmp1
            tmp1 = dp[i - 1][j + OFFSET];
        }
    }
}

```

```

        }
        dp[i][j + OFFSET] = tmp1; //当前值状态保存为三个转移来源转移得到
        的新值中最大的那个
    }
}

printf("Case %d: ", ++ cas); //按题目输出要求输出
if (dp[n][0 + OFFSET] == 0) { //dp[n][0]为0
    puts( HaveZero == true ? "0" : "-1"); //根据是否存在重量为0的柑橘输
    出0或-1
}
else printf("%d\n", dp[n][0 + OFFSET] / 2); //否则输出dp[n][0] / 2
}
return 0;
}

```

本节通过对两例一般动态规划问题的分析，旨在使读者对动态规划问题的求解以及确定状态和状态转移的重要性有更充分的认识。

若读者对本节中两个例题理解起来有困难，或者对自己能否独立求解这类问题有怀疑，你丝毫不必为此感到担心。以这两题的难度，在之前的机试真题中几乎可以被归为最难的一类。本节中，给出这两例只是为了使读者对动态规划有一个直观的了解，同时对复杂度的估计有初步的认识。而真正对动态规划有一个很好的掌握，还需要读者进行大量的练习。

六 背包

本节将要讨论动态规划问题中又一个十分常见，同时在机试中又是重点被考察的问题——背包问题。背包问题的变化之多让我们不容易一下子完全掌握它，根据考研机试的实际需要，我们在本节中主要讨论 0-1 背包、完全背包和多重背包三类背包问题，若读者对它的其它变化感兴趣，可自行查阅相关资料。

我们用一个例题，引出 0-1 背包的概念和问题。

例 7.7 采药（九度教程第 101 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

辰辰是个很有潜能、天资聪颖的孩子，他的梦想是称为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”如果你是辰辰，你能完成这个任务吗？

输入：

输入的第一行有两个整数 T ($1 \leq T \leq 1000$) 和 M ($1 \leq M \leq 100$)， T 代表总共能够用来采药的时间， M 代表山洞里的草药的数目。

接下来的 M 行每行包括两个在 1 到 100 之间（包括 1 和 100）的整数，分别表示采摘某株草药的时间和这株草药的价值。

输出：

可能有多组测试数据，对于每组数据，

输出只包括一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

样例输入：

```
70 3
71 100
69 1
1 2
```

样例输出：

```
3
```

来源：

2008 年北京大学图形实验室计算机研究生机试真题

首先我们将这个问题抽象：有一个容量为 V 的背包，和一些物品。这些物品分别有两个属性，体积 w 和价值 v ，每种物品只有一个。要求用这个背包装下价值尽可能多的物品，求该最大价值，背包可以不被装满。

因为最优解中，每个物品都有两种可能的情况，即在背包中或者不存在（背包中有 0 个该物品或者 1 个），所以我们把这个问题称为 0-1 背包问题。在该例中，背包的容积和物品的体积等效为总共可用的时间和采摘每个草药所需的时间。

在众多方案中求解最优解，是典型的动态规划问题。为了用动态规划来解决该问题，我们用 $dp[i][j]$ 表示在总体积不超过 j 的情况下，前 i 个物品所能达到的最大价值。初始时， $dp[0][j]$ ($0 \leq j \leq V$) 为 0。依据每种物品是否被放入背包，每

个状态有两个状态转移的来源。若物品 i 被放入背包，设其体积为 w ，价值为 v ，则 $dp[i][j] = dp[i-1][j-w] + v$ 。即在总体积不超过 $j-w$ 时前 $i-1$ 件物品可组成的最大价值的基础上再加上 i 物品的价值 v ；若物品不加入背包，则 $dp[i][j] = dp[i-1][j]$ ，即此时与总体积不超过 j 的前 $i-1$ 件物品组成的价值最大值等价。选择它们之中较大的值成为状态 $dp[i][j]$ 的值。综上所述，0-1 背包的状态转移方程为：

$$dp[i][j] = \max\{dp[i-1][j-w] + v, dp[i-1][j]\};$$

转移时要注意， $j-w$ 的值是否为非负值，若为负则该转移来源不能被转移。

代码 7.7

```
#include <stdio.h>

#define INF 0x7fffffff

int max (int a, int b) {return a > b ? a : b;} //取最大值函数

struct E { //保存物品信息结构体
    int w; //物品的体积
    int v; //物品的价值
} list[101];

int dp[101][1001]; //记录状态数组, dp[i][j] 表示前i个物品组成的总体积不大于j的最大价值和

int main () {
    int s, n;
    while (scanf ("%d%d", &s, &n) != EOF) {
        for (int i = 1; i <= n; i++) {
            scanf ("%d%d", &list[i].w, &list[i].v);
        } //输入
        for (int i = 0; i <= s; i++) {
            dp[0][i] = 0;
        } //初始化状态
        for (int i = 1; i <= n; i++) { //循环每一个物品
            for (int j = s; j >= list[i].w; j--) { //对s到list[i].w的每个j，
                //状态转移来源为dp[i-1][j]或dp[i-1][j-list[i].w]+list[i].v，选择其中较大的值
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-list[i].w] + list[i].v);
            }
            for (int j = list[i].w - 1; j >= 0; j--) //对list[i].w - 1到0的
```

每个j，状态仅能来源于dp[i - 1][j]，固直接赋值

```
        dp[i][j] = dp[i - 1][j];
    }
    printf("%d\n", dp[n][s]); //输出答案
}
return 0;
}
```

观察状态转移的特点，我们发现 dp[i][j]的转移仅与 dp[i-1][j-list[i].w]和 dp[i-1][j]有关，即仅与二维数组中本行的上一行有关。根据这个特点，我们可以将原本的二维数组优化为一维，并用如下方式完成状态转移：

$$dp[j] = \max\{dp[j - list[i].w] + v, dp[j]\};$$

其中在本次更新中未经修改的 dp[j-list[i].w]与 dp[j]与原始写法中的 dp[i-1][j-list[i].w]与 dp[i-1][j]等值。为了保证状态正确的转移，我们必须保证在每次更新中确定状态 dp[j]时，dp[j]和 dp[j-list[i].w]尚未被本次更新修改。考虑到 j - list[i].w < j，那么在每次更新中倒序遍历所有 j 的值，就能保证在确定 dp[j]的值时，dp[j - list[i].w]的值尚未被修改，从而完成正确的状态转移。

代码 7.8

```
#include <stdio.h>

#define INF 0x7fffffff

int max (int a,int b) {return a > b ? a : b;} //取最大值函数

struct E { //表示物品结构体
    int w;
    int v;
}list[101];

int dp[1001];

int main () {
    int s , n;
    while (scanf ("%d%d",&s,&n) != EOF) {
        for (int i = 1;i <= n;i ++){
            scanf ("%d%d",&list[i].w,&list[i].v);
        } //输入
        for (int i = 0;i <= s;i ++){
            dp[i] = 0;
```

```

    } //初始值
    for (int i = 1; i <= n; i++) {
        for (int j = s; j >= list[i].w; j--) { //必须倒序更新每个dp[j]的值,
j 小于list[i].w的各dp[j]不作更新, 保持原值, 即等价于dp[i][j] = dp[i-1][j]
            dp[j] = max(dp[j], dp[j - list[i].w] + list[i].v); //dp[j]
在原值和dp[j - list[i].w]+list[i].v中选取较大的那个
        }
    }

    printf("%d\n", dp[s]); //输出答案
}
return 0;
}

```

分析求解 0-1 背包问题的算法复杂度。其状态数量为 $n*s$ ，其中 n 为物品数量， s 为背包的总容积，状态转移复杂度为 $O(1)$ ，所以综合时间复杂度为 $O(n*s)$ 。经优化过后的空间复杂度仅为 $O(s)$ （不包括保存物品信息所用的空间）。

0-1 背包问题是最基本的背包问题，其它各类背包问题都是在其基础上演变而来。牢记 0-1 背包的特点：每一件物品至多只能选择一件，即在背包中该物品数量只有 0 和 1 两种情况。

0-1 背包存在一个简单的变化，即要求所选择的物品必须恰好装满背包。此时，我们设计新的状态 $dp[i][j]$ 为前 i 件物品恰好体积总和为 j 时的最大价值，其状态转移与前文中所讲的 0-1 背包完全一致，而初始状态发生变化。其初始状态变为， $dp[0][0]$ 为 0，而其它 $dp[0][j]$ （前 0 件物品体积总量为 j ）值均变为负无穷或不存在，经过状态转移后，得出 $dp[n][s]$ 即为答案。综上所述，该变化与原始 0-1 背包的差别仅体现在初始值方面，其它各步骤均保持不变。

接着，我们扩展 0-1 背包问题，使每种物品的数量无限增加，便得到完全背包问题：有一个容积为 V 的背包，同时有 n 个物品，每个物品均有各自的体积 w 和价值 v ，每个物品的数量均为无限个，求使用该背包最多能装的物品价值总和。

我们先按照 0-1 背包的思路试着求解该问题。设当前物品的体积为 w ，价值为 v ，考虑到背包中最多存放 V/w 件该物品，我们可以将该物品拆成 V/w 件，即将当前可选数量为无限的物品等价于 V/w 件体积为 w 、价值为 v 的不同物品。对所有的物品均做此拆分，最后对拆分后的所有物品做 0-1 背包即可得到答案。但是，这样的拆分将使物品数量大大增加，其时间复杂度为：

$$O(S * \sum_{i=1}^n S / w_i)$$

可见，当 S 较大同时每个物品的体积较小时其复杂度会显著增大，固将该问题转化为 0-1 背包的做法较不可靠。但是由该解法可窥见 0-1 背包的重要性，很多背包问题均可以推到 0-1 背包上来。

这里，我们要提出一种时间复杂度为 $O(n*s)$ 的解法，其使用如前文中经过空间优化过的 0-1 背包所使用的一维数组，按如下方法进行状态转移：

```
for (int i = 1; i <= n; i++) {
    for (int j = list[i].w; j <= s; j++) {
        dp[j] = max(dp[j], dp[j - list[i].w] + list[i].v);
    }
}
```

注意到该代码片段与上文中所讲的 0-1 背包相比，似乎只存在着对状态 j 的遍历顺序有所差异，这是有原因的。在 0-1 背包中，之所以逆序循环更新状态是为了保证更新 $dp[j]$ 时， $dp[j - list[i].w]$ 的状态尚未因为本次更新而发生改变，即等价于由 $dp[i - 1][j - list[i].w]$ 转移得到 $dp[i][j]$ 。逆序循环，保证了更新 $dp[j]$ 时， $dp[j - list[i].w]$ 是没有放入物品 i 时的数据($dp[i - 1][j - list[i].w]$)，这是因为 0-1 背包中每个物品至多只能被选择一次。而在完全背包中，每个物品可以被无限次选择，那么状态 $dp[i][j]$ 恰好可以由可能已经放入物品 i 的状态 $dp[i][j - list[i].w]$ 转移而来，固在这里将状态的遍历顺序改为顺序，使在更新状态 $dp[j]$ 时， $dp[j - list[i].w]$ 时可能因为放入物品 i 而发生改变，从而达到目的。

例 7.8 Piggy-Bank （九度教程第 102 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

Before ACM can do anything, a budget must be prepared and the necessary financial support obtained. The main income for this action comes from Irreversibly Bound Money (IBM). The idea behind is simple. Whenever some ACM member has any small money, he takes all the coins and throws them into a piggy-bank. You know that this process is irreversible, the coins cannot be removed without breaking the pig. After a sufficiently long time, there should be enough cash in the piggy-bank to pay everything that needs to be paid.

But there is a big problem with piggy-banks. It is not possible to determine how much money is inside. So we might break the pig into pieces only to find out that there is not enough money. Clearly, we want to avoid this unpleasant situation. The only possibility is to weigh the piggy-bank and try to guess how many coins are inside. Assume that we are able to determine the weight of the pig exactly and that we know the weights of all coins of a given currency. Then there is some minimum amount of money in the piggy-bank that we can guarantee. Your task is to find out this worst case and determine the minimum amount of cash inside the piggy-bank. We need your help. No more prematurely broken pigs!

输入:

The input consists of T test cases. The number of them (T) is given on the first line of the input file. Each test case begins with a line containing two integers E and F. They indicate the weight of an empty pig and of the pig filled with coins. Both weights are given in grams. No pig will weigh more than 10 kg, that means $1 \leq E \leq F \leq 10000$. On the second line of each test case, there is an integer number N ($1 \leq N \leq 500$) that gives the number of various coins used in the given currency. Following this are exactly N lines, each specifying one coin type. These lines contain two integers each, P and W ($1 \leq P \leq 50000$, $1 \leq W \leq 10000$). P is the value of the coin in monetary units, W is its weight in grams.

输出:

Print exactly one line of output for each test case. The line must contain the sentence "The minimum amount of money in the piggy-bank is X." where X is the minimum amount of money that can be achieved using coins with the given total weight. If the weight cannot be reached exactly, print a line "This is impossible.".

样例输入:

```
3
10 110
2
1 1
30 50
10 110
2
1 1
50 30
1 6
```

2

10 3

20 4

样例输出:

The minimum amount of money in the piggy-bank is 60.

The minimum amount of money in the piggy-bank is 100.

This is impossible.

题目大意：有一个储蓄罐，告知其空时的重量和当前重量，并给定一些钱币的价值和相应的重量，求储蓄罐中最少有多少现金。

由于每个钱币的数量都可以有任意多，所以该问题为完全背包问题。但是在该例中，完全背包有两处变化，首先，要求的不再是最大值，而变为了最小值，这就要求我们在状态转移时，在 $dp[j]$ 和 $dp[j-list[i].w]+list[i].v$ 中选择较小的转移值；其次，该问题要求钱币和空储蓄罐的重量恰好达到总重量，即在背包问题中表现为背包恰好装满，在前文中我们已经讨论了 0-1 背包的此类变化，我们只需变化 $dp[j]$ 的初始值即可。

代码 7.9

```
#include <stdio.h>
#define INF 0x7fffffff

int min (int a,int b) {return a < b ? a : b;} //取最小值函数
struct E { //代表钱币结构体
    int w; //重量
    int v; //价值
}list[501];
int dp[10001]; //状态
int main () {
    int T;
    scanf ("%d",&T); //输入测试数据组数
    while (T --) { //T次循环, 处理T组数据
        int s , tmp;
        scanf ("%d%d",&tmp, &s); //输入空储蓄罐数量, 和装满钱币的储蓄罐重量
        s -= tmp; //计算钱币所占重量
        int n;
        scanf ("%d",&n);
        for (int i = 1;i <= n;i ++) {
```

```

        scanf ("%d%d", &list[i].v, &list[i].w);
    } //输入数据

    for (int i = 0; i <= s; i++) {
        dp[i] = INF;
    }

    dp[0] = 0; //因为要求所有物品恰好装满, 所以初始时, 除dp[0]外, 其余dp[j]均为
    无穷 (或者不存在)

    for (int i = 1; i <= n; i++) { //遍历所有物品
        for (int j = list[i].w; j <= s; j++) { //完全背包, 顺序遍历所有可能
        转移的状态

            if (dp[j - list[i].w] != INF) //若dp[j - list[i].w]不为无穷, 就
            可以由此状态转移而来

                dp[j] = min(dp[j], dp[j - list[i].w] + list[i].v); //取
                转移值和原值的较小值
            }
        }

        if (dp[s] != INF) //若存在一种方案使背包恰好装满, 输出其最小值
            printf("The minimum amount of money in the piggy-bank
            is %d.\n", dp[s]);

        Else //若不存在方案
            puts("This is impossible.");
    }

    return 0;
}

```

总结一下完全背包问题, 其特点为每个物品可选的数量为无穷, 其解法与 0-1 背包整体保持一致, 与其不同的仅为状态更新时的遍历顺序。时间复杂度和空间复杂度均和 0-1 背包保持一致。

最后, 我们介绍多重背包问题, 其介于 0-1 背包和完全背包之间: 有容积为 V 的背包, 给定一些物品, 每种物品包含体积 w 、价值 v 、和数量 k , 求用该背包能装下的最大价值总量。

与之前的背包问题都不同, 每种物品可选的数量不再为无穷或者 1, 而是介于其中的一个确定的数 k 。与之前讨论的问题一样, 我们可以将多重背包问题直接转化到 0-1 背包上去, 即每种物品均被视为 k 种不同物品, 对所有的物品求 0-1 背包, 其时间复杂度为:

$$O(s * \sum_{i=1}^n k_i)$$

由此可见，降低每种物品的数量 k_i 将会大大的降低其复杂度，于是我们采用一种更为有技巧性的拆分。将原数量为 k 的物品拆分为若干组，每组物品看成一件物品，其价值和重量为该组中所有物品的价值重量总和，每组物品包含的原物品个数分别为：为：1、2、4... $k-2^{c+1}$ ，其中 c 为使 $k-2^{c+1}$ 大于 0 的最大整数。这种类似于二进制的拆分，不仅将物品数量大大降低，同时通过对这些若干个原物品组合得到新物品的不同组合，可以得到 0 到 k 之间的任意件物品的价值重量和，所以对所有这些新物品做 0-1 背包，即可得到多重背包的解。由于转化后的 0-1 背包物品数量大大降低，其时间复杂度也得到较大优化，为：

$$O(s * \sum_{i=1}^n \log_2(k_i));$$

例 7.8 珍惜现在，感恩生活（九度教程第 103 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

为了挽救灾区同胞的生命，心系灾区同胞的你准备自己采购一些粮食支援灾区，现在假设你一共有资金 n 元，而市场有 m 种大米，每种大米都是袋装产品，其价格不等，并且只能整袋购买。请问：你用有限的资金最多能采购多少公斤粮食呢？

输入：

输入数据首先包含一个正整数 C ，表示有 C 组测试用例，每组测试用例的第一行是两个整数 n 和 $m(1 \leq n \leq 100, 1 \leq m \leq 100)$ ，分别表示经费的金额和大米的种类，然后是 m 行数据，每行包含 3 个数 p ， h 和 $c(1 \leq p \leq 20, 1 \leq h \leq 200, 1 \leq c \leq 20)$ ，分别表示每袋的价格、每袋的重量以及对应种类大米的袋数。

输出：

对于每组测试数据，请输出能够购买大米的最多重量，你可以假设经费买不光所有的大米，并且经费你可以不用完。每个实例的输出占一行。

样例输入：

```
1
8 2
2 100 4
```


4 100 2

样例输出：

400

在该例中，对每个物品的总数量进行了限制，即多重背包问题。我们对每种物品进行拆分，使物品数量大大减少，同时通过拆分后的物品间的组合又可以组合出所有物品数量的情况。

代码 7.9

```
#include <stdio.h>

struct E { //大米
    int w; //价格
    int v; //重量
}list[2001];

int dp[101];

int max (int a,int b) {return a > b ? a : b;} //取最大值函数

int main () {
    int T;
    scanf ("%d",&T);
    while (T --) {
        int s , n;
        scanf ("%d%d",&s,&n);
        int cnt = 0; //拆分后物品总数
        for (int i = 1;i <= n;i ++) { //输入
            int v , w , k;
            scanf ("%d%d%d",&w,&v,&k);
            int c = 1;
            while (k - c > 0) { //对输入的数字k, 拆分成1, 2, 4... $k - 2^c + 1$ , 其中
c为使最后一项大于0的最大整数
                k -= c;
                list[++ cnt].w = c * w;
                list[cnt].v = c * v; //拆分后的大米重量和价格均为组成该物品的大米
的重量价格和
                c *= 2;
            }
            list[++ cnt].w = w * k;
```

```

        list[cnt].v = v * k;
    }
    for (int i = 1; i <= s; i++) dp[i] = 0; //初始值
    for (int i = 1; i <= cnt; i++) { //对拆分后的所有物品进行0-1背包
        for (int j = s; j >= list[i].w; j--) {
            dp[j] = max(dp[j], dp[j - list[i].w] + list[i].v);
        }
    }
    printf("%d\n", dp[s]); //输出答案
}
return 0;
}

```

总结多重背包问题：多重背包的特征是每个物品可取的数量为一个确定的整数，我们通过对这个整数进行拆分，使若干个物品组合成一个价值和体积均为这几个物品的和的大物品，同时通过这些大物品的间的组合又可以组合出选择任意件物品所包含的体积和重量情况，通过这种拆分使最后进行 0-1 背包的物品数量大大减少，从而降低复杂度，其时间复杂的为：

$$O(s * \sum_{i=1}^n \log_2(k_i));$$

空间复杂度与 0-1 背包保持一致。

本节主要讨论了背包问题，我们重点指出了三个类型的背包问题，0-1 背包、完全背包、多重背包，其中 0-1 背包是背包问题的基础，很多背包问题都可以转化到 0-1 背包上来。

总结

本章主要讲述动态规划相关问题。先后介绍了递推求解、最长递增子序列、最长公共子序列，并结合其中解题要点，提出了状态和状态转移，并试着通过它们估算动态规划的时间复杂度，再用两个实际问题对其进行举例说明。最后着重讨论了动态规划中最常见的背包问题，我们共涉及了 0-1 背包、完全背包、多重背包三类背包问题。动态规划问题，历来以难度高而著称，虽然考研机试对动态规划的考察不会太深入，但读者也要做好求解动态规划问题的准备。

第8章 其它技巧

作为全文的最后一部分，本章将会介绍许多在机试考试中能给考生带来极大方便的技巧，旨在使考生掌握机试知识的同时对一些简便的处理技巧也有一定的了解。

一 标准模板库（STL）

在前几个章节中我们已经使用了诸如队列、堆、堆栈、`vector` 等标准模板库中的模板，切身感受到了它给我们带来的极大便利。在本节中，我们还要介绍两种标准模板——`string` 和 `map`，了解他们又会给我们带来怎样的便利。

`string` 对象，顾名思义即用来保存和处理字符串的标准模板。我们介绍其相关的操作。

在使用它之前我们声明包括 `string` 模板

```
#include <string> //注意区别于string.h
```

并使用标准命名空间

```
using namespace std;
```

利用语句

```
string s;
```

定义 `string` 对象 `s`。我们可以使用 `cin` 对其进行输入

```
cin >> s;
```

也可以使用已经保存在字符数组里的字符串直接对其赋值

```
char str[] = "test";
```

```
s = str;
```

对已经存在的 `string` 对象 `s`，我们可以在其最后添加一个字符

```
s += 'c';
```

添加一个字符串

```
s += "string";
```

甚至添加一个 `string` 对象

```
string b = "class";
```

```
s += b;
```

可以这样操作的原因是标准模板库中已经帮我们重载了例如`+`、`+=`等运算符的行为，所以我们可以像使用基本类型一样直接调用它。

其它常用运算符有：

判断两个字符串是否相同

```
string b = "class";
string a = "Two";
if (a == b) {
    cout << a;
}
```

判断两个字符串间的大小关系

```
string b = "class";
string a = "Two";
if (a <= b) {
    cout << a;
}
```

同样的，与其对应的小于运算、大于运算、大于等于运算均可调用。

要输出一个 **string** 对象保存的字符串，我们可以使用 C++ 风格的输出

```
string c = "cout";
cout << c << endl;
```

也可以使用 C 风格的输出

```
string c = "cout";
printf("%s\\b", c.c_str());
```

若要对 **string** 对象 **s** 中的每一个字符中进行遍历，需要以下循环

```
for (int i = 0; i < s.size(); i++) { //注意循环终止条件
    char c = s[i];
}
```

除了以上基本操作以外，**string** 还包括以下常用的内置函数

```
s.erase(10, 8);
```

从 **string** 对象 **str** 中删除从 **s[10]** 到 **s[17]** 的字符，即从 **s[10]** 开始的 8 个字符。

```
string a = "asdfsdfadfafd";
string b = "fadf";
int startPos = 0;
int pos = a.find(b, startPos);
```

在 **string** 中下标 **startPos** 位置开始查找 **b** 字符串，若能够找到 **b** 字符串则返回其第一次出现的下标；否则，返回一个常数 **string::npos**。其中 **string** 对象 **b** 也可以为字符数组。

```
string a = "AAAA";  
string b = "BBB";  
a.insert(2, b);  
cout << a << endl;
```

在a中下标为2的字符前插入b字符串。其中string对象b也可以为字符数组。

以上为我们常用的几个函数，string还有其他函数在这里我们不在赘述，有兴趣的读者可自行查阅资料。

下面通过相关例题，体验string对象给我们带来的巨大便利。

例 8.1 字符串的查找删除（九度教程第 104 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

给定一个短字符串（不含空格），再给定若干字符串，在这些字符串中删除所含有的短字符串。

输入：

输入只有 1 组数据。

输入一个短字符串（不含空格），再输入若干字符串直到文件结束为止。

输出：

删除输入的短字符串(不区分大小写)并去掉空格,输出。

样例输入：

```
in  
#include  
int main()  
{  
  
printf(" Hi ");  
}
```

样例输出：

```
#clude  
tma()  
{  
  
prtf("Hi");  
}
```

提示：

注:将字符串中的 In、IN、iN、in 删除。

来源:

2009 年北京航空航天大学计算机研究生机试真题

若我们使用字符数组手动实现其所要求的查找删除操作,那么需要耗费大量的编码时间不说,即使是程序的正确性我们也很难保证。但是使用标准对象 string,情况就大不相同了。这里我们特别注意,题面中要求的“大小写不区分”,为了达到这一目的,我们将字符串全部改写成小写后进行匹配。

代码8.1

```
#include <stdio.h>
#include <string>
#include <iostream>
#include <ctype.h>
using namespace std;
int main () {
    char str[101];
    gets(str); //输入短字符串
    string a = str; //将其保存在a中
    for (int i = 0; i < a.size(); i++) {
        a[i] = tolower(a[i]);
    } //将a中的字符全部改成小写
    while (gets(str)) { //输入长字符串
        string b = str, c = b; //将字符串保存至b, c
        for (int i = 0; i < b.size(); i++) {
            b[i] = tolower(b[i]);
        } //将b中字符全部改成小写, 以便匹配
        int t = b.find(a, 0); //在b中查找a的位置
        while (t != string::npos) { //若查找成功, 则重复循环
            c.erase(t, a.size()); //删除c中相应位置字符, c为原串
            b.erase(t, a.size()); //删除b中相应位置字符, b为改为小写字母的串
            t = b.find(a, t); //继续查找b中下一个出现字符串a的位置
        }
        t = c.find(' ', 0); //查找c中空格
        while (t != string::npos) {
            c.erase(t, 1);
```

```

        t = c.find(' ', 0);
    } //删除c中所有空格

    cout << c << endl; //输出
}

return 0;
}

```

可见，在使用了string对象后，关于字符串处理的问题将得到大大简化。

这里，还要提醒大家注意另一个非常重要的地方——关于gets()函数的使用。

如代码中语句gets(str)，我们使用该语句读入输入中一整行的数据保存在str中。但其在与scanf()函数合用时，我们必须小心翼翼的处理一些情况。作为反例，若将上例的解题代码中输入不包含空格的短字符，改为scanf ("%s", str)的输入方法，其它地方不变，程序将会出现错误。

要回答其原因，先来了解gets的输入特点，当程序运行至gets语句后，它将依次读入遗留在输入缓冲中的数据直到出现换行符，并将除换行符外的所有已读字符保存在字符数组中，同时从输入缓冲中去除该换行符。

假设输入数据格式为，第一行为两个整数，第二行为一个字符串，如：

2 3 (换行)

Test (换行)

为了读入输入数据，我们使用语句

```

scanf ("%d%d", &a, &b);

gets(str);

```

我们来分析其运行过程，当程序运行至scanf时，程序读入输入缓冲中的数据2 3，并将数字2、3分别保存至变量a、b中，此时输入缓冲中遗留的数据为第一行一个换行符，第二行一个字符串，即

(换行)

Test (换行)

程序继续执行至gets语句，程序在输入缓冲中第一个读到的字符即为换行符，gets运行结束，它去除输入缓冲中换行的同时并没有读到任何字符，即str为空字符串，而原本将要输入的字符串却留在了输入缓冲中。这样，便造成了程序不能正确读入接下来的数据，这也就是为什么我们随意使用gets语句时会出现错误的原因。与其对应，scanf ("%s",str)函数读取输入缓冲的字符直到出现空格、换行字符，它将读到的字符保存至字符数组str中，但并不删除缓冲中紧接的空格与换行，上例中，若使用scanf ("%s") 读入短字符串后，其后换行符依然保留在缓冲中，从而导致后续gets函数不能正常使用。

所以，在使用gets时，我们对之前输入遗留在输入缓冲的换行符要特别的关注，确定其是否会对gets造成危险，如上的输入正确的处理方式为

```
scanf ("%d%d", &a, &b);  
getchar();  
gets(str);
```

即在scanf后使用一个getchar去消除输入缓冲的换行符，使程序正常运行。

基于如上原因，我们应尽可能的避免使用gets，除非输入要求输入“包括空格”的一整行，否则我们尽可能的使用scanf ("%s",str)去代替其完成功能。

上面我们主要讨论了string在机试中的用途，接下去我们还要介绍标准模板库中另一个十分实用的标准对象——map。

其功能为将一个类型的变量映射至另一类型。我们用一个例题，介绍和展示其用处和用法。

例 8.2 产生冠军（九度教程第 105 题）

时间限制：1 秒

内存限制：32 兆

特殊判题：否

题目描述：

有一群人，打乒乓球比赛，两两捉对厮杀，每两个人之间最多打一场比赛。

球赛的规则如下：如果 A 打败了 B，B 又打败了 C，而 A 与 C 之间没有进行过比赛，那么就认定，A 一定能打败 C。

如果 A 打败了 B，B 又打败了 C，而且，C 又打败了 A，那么 A、B、C 三者都不可能成为冠军。

根据这个规则，无需循环较量，或许就能确定冠军。你的任务就是面对一群比赛选手，在经过了若干场厮杀之后，确定是否已经实际上产生了冠军。

输入：

输入含有一些选手群，每群选手都以一个整数 n(n<1000)开头，后跟 n 对选手的比赛结果，比赛结果以一对选手名字（中间隔一空格）表示，前者战胜后者。如果 n 为 0，则表示输入结束。

输出：

对于每个选手群，若你判断出产生了冠军，则在一行中输出“Yes”，否则在一行中输出“No”。

样例输入：

```
3  
Alice Bob  
Smith John  
Alice Smith
```



```
5
a c
c d
d e
b e
a d
0
```

样例输出：

```
Yes
No
```

仔细阅读前面章节的读者应该对这样的例题并不感到陌生，这便是我们在图论中所讨论过的拓扑排序问题。将选手对应结点，胜负关系对应为结点之间的有向边，可以产生冠军的情况即为全图中入度为零的点唯一。

与普通的拓扑排序问题不同，这里我们需要将输入的选手姓名映射为结点编号，这就需要标准对象map。

下面给出该题解题代码，了解map的应用。

代码8.2

```
#include <stdio.h>
#include <vector>
#include <map> //要使用map, 必须包含此头文件.
#include <string>
#include <queue>
using namespace std; //声明使用标准命名空间
map<string, int> M; //定义一个完成从string到int映射的map
int in[2002];
int main () {
    int n;
    while (scanf ("%d", &n) != EOF && n != 0) {
        for (int i = 0; i < 2 * n; i++) { //n组胜负关系, 至多存在n个队伍
            in[i] = 0; //初始化入度
        }
        M.clear(); //对map中的映射关系清空
        int idx = 0; //下一个被映射的数字
        for (int i = 0; i < n; i++) {
```

```

char str1[50], str2[50];
scanf ("%s%s", str1, str2); //输入两个选手名称
string a = str1, b = str2; //将字符串保存至string中
int idxa , idxb;
if (M.find(a) == M.end()) { //若map中尚无对该a的映射
    idxa = idx;
    M[a] = idx ++; //设定其映射为idx , 并递增idx
}
else idxa = M[a]; //否则, 直接读出该映射
if (M.find(b) == M.end()) {
    idxb = idx;
    M[b] = idx ++;
}
else idxb = M[b]; //确定b的映射, 方法与a相同
in[idxb] ++; //b的入度递增
}
int cnt = 0;
for (int i = 0; i < idx; i ++) { //确定所有映射数字的入度, 统计入度为0的个数
    if (in[i] == 0)
        cnt ++;
}
puts(cnt == 1 ? "Yes" : "No"); //若入度为0输出Yes, 否则输出No.
}
return 0;
}

```

如例所示, map很好的完成了从string到int的映射, 即完成了选手姓名到结点编号的映射。

下面回顾它的用法:

```

map<string, int> M; //定义一个完成从string到int映射的map
M.clear(); //清空一个map
M.find(b); //确定map中是否保存string对象b的映射, 若没有函数返回M.end()
M[b] = idx; //若map中不存在string对象b的映射, 则定义其映射为b映射为idx
idxb = M[b]; //若map中存在string对象b的映射, 则读出该映射

```

在了解了以上map的用法和用处后，我们就能使用map完成特定类型变量之间的映射，而不需我们作过多的干预。

顺便一提的是，map的内部实现是一棵红黑树。

本节主要介绍了标准模板库中两种在机试中有应用价值的标准模板，虽然历来人们对STL褒贬不一，反对的人们认为长期使用STL会大大的减弱程序员的代码能力。但是在机试中我们还是要充分发挥其方便、快捷的优势，试想同样的考生一个写了红黑树，而另一个直接调用了map，那么，谁的程序容易出错，谁的程序编写起来更为迅速？答案是毋庸置疑的。所以，抛开人们对STL的争论，在机试中STL还是大有裨益的。

最后给出几个相关练习题供读者练习。

练习题：单词替换(九度教程第 106 题);字符串去特定字符(九度教程第 107 题);

二 滚动数组

本节将要介绍另一个十分实用的小技巧——滚动数组。它常被用来完成常数优化和减少代码量。

假设有如下状态转移方程：

$$dp[i][j] = \max(dp[i-1][j+1], dp[i-1][j-1]);$$

按照该状态转移方程，我们可以用二维数组保存其状态值，通过如下代码片段完成其状态的转移（这里仅作说明，不考虑边界情况）：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        dp[i][j] = max(dp[i-1][j+1], dp[i-1][j-1]);
    }
}

int ans = dp[n][m];
```

考虑到每次状态的转移仅与上一行有关，我们可以将二维数组优化到使用一维数组保存。如下：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        buf[j] = max(dp[j+1], dp[j-1]);
    }
    for (int j = 1; j <= m; j++) {
```

```

        dp[j] = buf[j];
    }
}

int ans = dp[m];

```

如该代码片段所示，我们将原本二维的状态空间优化到了一维，对应的我们需要在每次状态转移过后进行一次循环次数为 m 的赋值操作。该操作不仅增加了代码量，还增加了程序的耗时。于是我们使用滚动数组，对其再次进行优化：定义大小为 $2*m$ 的数组为其状态空间：

```
int dp[2][M];
```

初始状态保存在 `dp[0][i]` 中。

设定两个 `int` 类型指针

```

int *src; //源指针
int *des; //目的指针

```

由于初始状态保存在 `dp` 数组的第 0 行中，初始时

```

src = dp[1];
des = dp[0];

```

按照状态转移方程进行状态转移

```

for (int i = 1; i <= n; i++) {
    swap(src, des); //交换源和目的指针
    for (int j = 1; j <= m; j++) {
        des[j] = max(src[j + 1], src[j - 1]);
    }
}

int ans = des[m];

```

如代码所示，我们在每次循环进行状态转移之前交换源数组和目的数组的指针，使程序能够正确的从源数组中转移状态到目的数组中。当状态转移完成时，新得到状态保存于目的数组中，但它在下一次循环的状态转移中又将变为源数组，于是我们在下次状态转移开始前再次交换源数组和目的数组指针，这就是滚动数组的工作原理。

滚动数组这个技巧不仅优化了原始的状态空间，还减少了循环次数节约了程序运行时间，同时对代码量的缩减也有很好的效果，是一个我们值得学习的小技巧。

三 调试技巧

调试是我们在编写程序时不得不经历的过程，本节将介绍若干调试的技巧。

1.输出调试。

为了更好的看到例如动态规划问题中的状态转移，我们可以在每一次状态转移以后将每个状态输出，并与自己笔算出来的状态作对比，观察何时何处出现了错误，再相应的寻找 bug 的所在。

如

```
for (int i = 1; i <= n; i++) {
    for (int j = s; j >= list[i].w; j--) {
        dp[j] = max(dp[j], dp[j - list[i].w] + list[i].v);
    }
    for (int j = 0; j <= n; j++) {
        printf("%d ", dp[j]);
    }
    printf("\n");
}
```

我们输出背包问题中每一次状态转移后的状态，检查其正确性，并对出现错误的地方做适当的修改。

这是一种不依赖于任何开发工具功能的调试方法，其特点是方便快捷，可快速的确定程序出现错误的地方，同时也可以作为检察动态规划的转移是否按照预期进行的手段。

切记在真正提交程序前删掉调试输出部分，耗费时间是小，输出多余的信息而造成错误是大。

2.断点

断点是绝大部分的编译工具都会提供的调试工具。对某个特定语句下断点，即声明程序运行至该语句将进行一次暂停，方便我们对此时程序的运行状态作相应的判断。但是，某些情况下，下断点还需要我们特殊的处理。

例如有循环

```
for (int i = 0; i < 123321; i++) {
    /*处理程序*/
}
```

假设，在我们的预想中，程序应该在 $i=99999$ 时得出答案，而此时程序并没有输出答案。于是我们要检查当 i 为 99999 时的程序运行状况，并对其正确性

做出判断。

我们添加如下语句

```
for (int i = 0; i < 123321; i++) {  
    if(i == 99999)  
        i = i;  
    /*处理程序*/  
}
```

我们在新添加的 `i = i` 处添加断点，这样当程序停止运行时，程序便自动地停止在 `i` 为 99999 时，我们可以进一步对此时程序为什么没有得出答案进行检查。

这样做的好处是，除了每次循环增加了一次条件判断而产生的轻微耗时外添加后的调试语句对程序不会有任何影响，所以在提交时，即使我们不对该程序做任何处理，程序也能正常运行。当然，还是建议各位在提交时删除添加的调试信息。

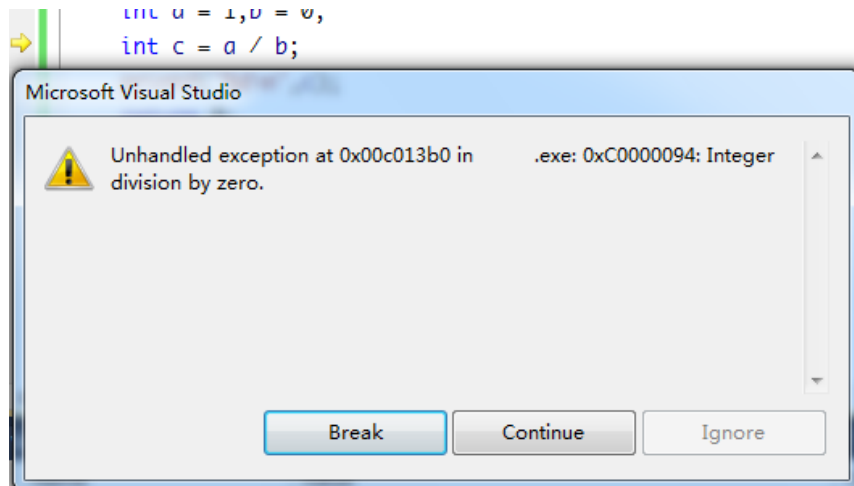
3.运行到断点的其它用处。

假如我们的程序在本地运行测试题目给出的样例就出现了程序异常终止的错误时，我们可以使用运行到断点命令判断何处出现了何种异常。

运行到断点命令，即使用 **debug** 模式运行程序，直到出现断点，程序将会暂停在断点语句，若程序不存在任何断点即运行至结束。

正因为此时程序运行至 **debug** 模式，编译器为程序添加了许多用于调试的断言，如除数必定不为 0，数组的访问下标必在数组范围内等。而在 **debug** 模式里，当我们的程序存在问题而造成异常终止时，将很可能触发这些断言而使程序抛出异常，同时程序停止在出现错误的语句上。我们可以根据抛出的异常信息和程序停留的位置判断程序出错的原因。

如下图：



当然，具体的调试方法还与读者具体使用的开发工具有关，还需读者在自己习惯的开发工具中慢慢的体会和总结。

本节给出一些我们总结出的调试技巧，供读者参考。

四 补充技巧

本节将按照顺序，给出一些在机试中有用的其它技巧和提示，作为前几节的补充。

1.位运算。

巧妙的使用位运算可以提高程序的效率。

如条件语句

```
if (a % 2 == 1)
```

可由位运算语句代替

```
if (a & 1 == 1)
```

众所周知，求模运算是运算符中较为耗时的一类，我们用位运算（变量与1求与）代替模2操作将会大大的提高该语句的执行效率。

再如语句

```
a /= 2;
```

可由位运算

```
a >>= 1;
```

用右移操作代替除法操作也可大幅度的提高效率。

用位运算去代替原始操作，若该操作百万次的出现，那么所节约的时间还是相当可观的。

2.输入外挂

我们知道，使用 C++风格的 `cin` 输入要比 C 风格的 `scanf` 函数完成输入耗时许多。正因为如此，在输入量非常巨大的机试题中，使用 `cin` 往往比使用 `scanf` 吃亏不少。所以，好心的出题者总是在这类题目最后加上这样一句话，“Huge input, scanf is recommended”，防止因为输入而出现超时的冤枉情形出现。这也是为什么本文给出的绝大部分解题代码都采用了 c 风格的输入。

输入外挂正是在 `scanf` 的基础上对输入再次进行优化，使那些刚超时一点的程序能够卡进时限范围内。输入外挂的工作原理即为在读入输入缓冲中字符串的前提下，手动分析字符串中输入的整数、浮点数等我们需要的输入类型，并将其保存在变量中。

下面给出两个我们常使用的输入外挂：

```

bool readint(int &ret){ //输入整数, 并将整数保存在引用变量ret中
    int sgn; //符号
    char c; //字符
    c=getchar(); //读入字符
    if(c==EOF)return false; //若达到文件尾返回true
    while(c!='-' &&c<'0' || c>'9')c=getchar(); //跳过不为整数的部分
    sgn=(c=='-')?-1:1; //若出现负号
    ret=(c=='-')?0:(c-'0'); //若未出现负号
    while((c=getchar())>='0' &&c<='9')ret=ret*10+(c-'0'); //计算连续几个字符
    组成数字的数值
    ret*=sgn; //乘上符号位
    return true; //读入成功返回true
}

bool readdouble(double &ret){ //读入浮点数, 结果保存在引用变量ret中
    int sgn; //符号位
    double bit=0.1; //小数点后数位的权重
    char c; //字符
    c=getchar(); //读入字符
    if(c==EOF)return false; //到达文件尾返回false
    while(c!='-' &&c!='.' &&(c<'0' || c>'9'))c=getchar(); //跳过之前不为数字的部分
    sgn=(c=='-')?-1:1; //若出现负号
    ret=(c=='-')?0:(c-'0'); //若不出现负号
    while((c=getchar())>='0' &&c<='9')ret=ret*10+(c-'0'); //计算整数位
    if(c=='.' || c=='\n'){ret*=sgn; return true;} //若不存在小数位
    while((c=getchar())>='0' &&c<='9')ret+=(c-'0')*bit, bit/=10; //计算小数位
    ret*=sgn; //乘符号位
    return true; //读入成功, 返回true
}

```

我们用输入外挂代替 `scanf` 函数, 在大量的输入中可节约一笔时间。但是请特别注意, 输入外挂不是万能的, 它的效率和输入的数据密切相关, 效率也可能发生很大的变化, 如输入整数数位过长时, 外挂所做的乘法运算过多, 其效率会急剧下降, 甚至会比 `scanf` 函数更耗时。所以输入外挂是在我们超时并且没有其他更靠谱的优化方法的情况下, 万不得已的方法。正如其名字“外挂”一样, 并

不是什么正常的优化手段。

3.流氓剪枝。

我们在学习搜索的时候已经介绍过了关于剪枝的相关概念，剪枝即减去解答树上被判定不可能存在我们所需状态的子树。

此处所讲的流氓剪枝与正常的剪枝不同，若程序在搜索所有必须搜索的状态后出现了超时，并且我们没有什么其他更靠谱的方法对其优化时可以采取流氓剪枝。即我们强行的剪去我们看来不太可能存在答案的状态，或者干脆随机剪去几棵子树从而达到节约运行时间的效果。由于其剪枝是不正确的，某些情况下是毫无道理的，所以它的名字被称为流氓剪枝。

例如深搜时，当递归层数超过一定限度时强行返回

```
void DFS(int deep) {  
    if (deep > MAXDEEP) return;  
    /*其他处理*/  
    DFS(deep + 1);  
}
```

又例如广搜时，随机对某些状态不进行扩展

```
if (rand() % 100 < 5) 不扩展当前状态;  
else 扩展当前状态;
```

流氓剪枝的原理在于，假如我们搜索了大部分可能存在答案的状态都搜索不到该状态，那么它存在于剩余少部分未被搜索的状态中的可能性也是很小的，所以我们直接判断答案不存在。

如上所述，流氓剪枝不一定是正确的，程序可能因为毫无道理的剪枝错过了我们所需的目标状态，所以它也是在程序超时时万不得已使用的一种优化。

本节介绍了一些在机试中可以适当使用的小技巧，在一定的条件下可能带来奇效。

五 最后的提醒

若读者能够看到这里，首先祝贺你看完了本文所有的内容。作为全文的最后一节，我们给出机试中的注意事项。

1.提前了解正式机试将会使用的评判系统、编译环境和可以使用的开发工具，尽早做适应练习，特别是 64 位整数的定义和使用（long long 或__int64）。

2.在正式考试前往往有试机、练习时间，检查开发工具各功能是否正常，如编译、调试等。

3.在编写每一题时都要特别注意其时间复杂度。

4.注意程序是否符合题目要求的输出格式。

5.在机试过程中注意“跟风”，即先挑选通过人数的较多的题作答，而别死磕一道题。

6.不要在程序中混用 `printf` 和 `cout`，由于它们的输出机理不同，混用将非常容易造成错误。

总结

本章主要介绍了在机试中可以适当使用的编码、考试技巧,包括标准模板库、滚动数组、调试技巧,同时也介绍了在走投无路时可想的一些投机的技巧如流氓剪枝、输入外挂。最后对机试也做出了几点提醒。作为本文的最后一章,最后祝愿大家在机试中均能考出理想的成绩。