HZ BOOKS
华章教育

本书用于
ACM/ICPC、Google Code Jam、TopCoder
百度之星程序设计大赛
等程序设计竞赛的训练

数据结构编程实验

大学程序设计课程与竞赛训练教材

Data Structure Experiment

for Collegiate Programming Contest and Education

吴永辉 王建德 编著

附赠光盘

机械工业出版社
China Machine Press

数据结构编程实验

大学程序设计课程与竞赛训练教材

Data Structure Experiment

for Collegiate Programming Contest and Education

吴永辉 王建德 编著

机械工业出版社
China Machine Press

```
            cout << (valid ? "Yes" : "No") << endl;    //输出置换是否合法的信息

            cin >> x;                                  //读当前置换的下一个元素值
        }

        cout << endl;                                  //输出空行
        cin >> n;                                      //读下一个测试用例的元素数
    }

    return 0;
}
```

表达式求值是栈的典型应用。表达式一般包含操作数和运算符。

1）操作数：合法的变量名或常数。

2）运算符，包括：

- 用于数值计算的算术运算符，包括双目运算符（+、−、*、/、%）和单目运算符（−）；
- 用于比较大小的关系运算符，包括<、<=、==、!=、>、>=；
- 用于计算与（&&）、或（||）、非（！）关系的逻辑运算符；
- 用于改变运算顺序的括号。

为了正确执行表达式的计算，必须明确各个运算符的执行顺序，因此每个运算符都规定了一个优先级。C++中规定各运算符的优先级如下表：

优先级	运算符	优先级	运算符
1	−、！（单目）	5	==、!=
2	*、/、%	6	&&
3	+、−	7	||
4	<、<=、>、>=		

表达式中相邻两个运算符的计算次序是，优先级高的先计算；若优先级相同，则自左而右计算；使用括号时，从最内层的括号开始计算。例如表达式A+B*(C−D)−E/F，计算顺序如图5.2-3所示，R_1、R_2、R_3、R_4、R_5为中间计算结果。

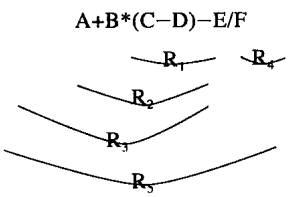


图 5.2-3

在计算表达式值的过程中，需要开设两个栈：

1）运算符栈op：存储运算符。

2）数值栈val：存储操作数和中间运算结果。若表达式中无优先级最高的单目算符，则操作数或中间运算结果直接压入val栈；否则操作数或中间运算结果入栈前需要分析op栈顶有多少个连续的单目运算符。这些单目运算符出op栈，并连续对操作数进行相应的单目运算，最后将运算结果压入val栈。

计算表达式的基本思想是顺序扫描表达式的每个字符，根据它的类型做如下相应操作：

- 若是操作数，则操作数压入val栈。
- 若是运算符<op>，则计算op栈顶中优先级比op高的双目算符$op_1..op_k$，弹出这些双目运算符。每弹出一个双目运算符$op_i(1\leqslant i\leqslant k)$，则从val栈中退出两个操作数a和b，形成运算指令a<op_i>b，并将结果重新压入val栈。进行完$op_1..op_k$的运算后，op压入运算符栈op。

扫描完表达式的所有字符后，若运算符栈op不空，则栈中的运算符相继出栈。每弹出一个运算符，从val栈中退出两个操作数，进行相应运算后的结果重新压入val栈。最后val栈的栈顶元素就是表达式的值。

前 言

我们长期从事数据结构教学和竞赛培训，教学实践使我们萌发了对数据结构课程的教学模式进行改革的想法：

1）在课程中需要增加思维方式和解题策略的引导，引导学生思考各类数据结构的本质特征是什么；面对当前问题为什么要采用这样的数据结构而不宜采用那样的数据结构；当有多个数据结构可用时，怎样权衡时间复杂度、空间复杂度、编程复杂度和思维复杂度四个因素，寻找最合时宜的数据结构，将“知识导向”与“智慧导向”真正结合起来。

2）在课程中需要引入案例教学，通过模拟或者重现现实生活中的一些场景，让学生置身问题情境之中，通过思考、讨论和上机编程来进行学习。传统教学将数据结构“束之高阁”，在理论教学和笔试上兜来兜去，可能会使学生浑然不知数据结构在现实生活中究竟派什么用处，懵懵懂懂，最终失去了学习的意义。“纸上得来终觉浅，绝知此事要躬行”。案例教学则是一种以问题和动手编程驱动学习的方式：将知识置于问题情境和实践过程之中，变枯燥乏味为生动活泼。学生拿到试题后，先进行审题，然后温习或查阅各种他认为必要的数据结构知识，这无形中激发了他们的求知欲望，加深了他们对知识真谛的理解。捕捉到相关的理论知识后，学生还要经过缜密思考和动手编程，使之变为解决问题的程序方案。这个“认识-实践-再认识-再实践”的过程，应视为知识理解上的一种提高，知识学习与应用能力间的一种转变和升华。

基于上述想法，我们近年来开设了“数据结构实验”课程，并将ACM国际大学生程序设计竞赛和其他程序设计竞赛中的典型试题分门别类，精选了其中204道试题，翻译后作为学习数据结构知识的实验案例。这些试题不仅为相关知识创设了丰富有趣的问题背景，而且在相关网站上都有试题的在线测试。学生不仅可以带着问题学习数据结构，而且所编程序的正确性和效率也可以通过相关网站上的测试系统得到实时检验，达到“学以致用”的目的。

本书试题的在线测试地址主要有：

在线评测系统	简称	网址
北京大学在线评测系统	POJ	http://poj.org/
浙江大学在线评测系统	ZOJ	http://acm.zju.edu.cn/onlinejudge/
UVA在线评测系统	UVA	http://uva.onlinejudge.org/ http://livearchive.onlinejudge.org/
Ural在线评测系统	Ural	http://acm.timus.ru/
SGU在线评测系统	SGU	http://acm.sgu.ru/

本书按照数据结构的知识结构和循序渐进的原则，共分四大篇（基本能力的编程实验、线性数据结构的编程实验、层次类非线性表的编程实验、群聚类非线性表的编程实验）14个章节。每个章节为相关数据结构知识提供了大量的实验范例，并且建立了相关的试题库，其中实验范例68道，题库试题136道。每个实验范例不仅有知识要点阐述和详尽的试题解析，还列出了写有详细注释的参考程序；题库中的所有试题无论难易，都有清晰的提示。教师既可以将实验范

例作为开设数据结构实验课程的教材，也可以从相关题库中挑选数据结构实验课程的作业、考题或指导ACM集训活动的培训资料。每道题都注明了试题来源和在线测试网址，考虑到网站更新给读者学习带来不便的可能情况，本书还附带了存储所有试题的英文原版描述和大部分试题的测试数据等资料的光盘。我们之所以这样做，就是力图构建一个尽可能丰富、实用和长效的数据结构实验课程资源库。

需要说明的是，本书是在复旦大学ACM集训队长期活动的基础上积累而成的。阿拉法特·居来提、姚哲云、张昊等同学精心编写了所有程序，每道程序都通过了严格的测试验证，其中一些程序经多次修改，精益求精。这些同学为本书的出版付出了辛勤的劳动，作出了不可或缺的贡献。在此，向他们表示由衷的感激。另外，衷心感谢复旦大学计算机学院2006、2007、2008级同学在使用本书讲义过程中提出的宝贵意见和建议。

由于时间和水平所限，书中肯定夹杂了不少缺点和错误，表述不当和笔误更是在所难免，热忱欢迎学术界同仁和读者赐正。如果你在阅读中发现了问题，请通过书信或电子邮件告诉我们，以便及时整理成勘误表放在本书的专门网站上，供广大读者查询更正。我们更期望读者对本书提出建设性意见，以便再版时改进。我们的联系方式是：

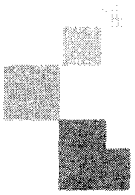
通信地址：上海市邯郸路220号复旦大学计算机科学技术学院　吴永辉

邮编：200433

E-mail：yhwu@fudan.edu.cn

目 录

第四篇 群聚类非线性表的编程实验



第一篇

基本能力的编程实验

数据结构实验课程不是听会的，也不是看会的，而是练会的，是让学生在充分上机动手编程的过程中逐步学会的。数据结构的先导课程是程序设计语言，其教学目的是让学生学会用计算机编程语言编写程序。因此，我们在数据结构实验课伊始，先引领学生“温故知新”，开展三个方面的编程实验：简单计算，简单模拟，简单递归。这三类实验既是程序设计语言课程的温习，也是数据结构实验课程的入门和前奏。

第1章

简单计算的编程实验

所谓简单计算题，指的是“输入-处理-输出”模式中，处理环节涉及的运算规则比较浅显，编程的重心应放在如何正确地输入、输出或优化计算上。学生可以通过编程解简单计算题的实验，掌握C/C++或Java程序设计语言的基本语法，熟悉在线测试系统和编程环境，初步学会怎样将一个自然语言描述的实际问题抽象成一个计算问题，给出计算过程，继而编程实现计算过程，并将计算结果还原成对原来问题的解答。

虽然简单计算题的运算相对简单，但还是应该取“取轻若重”的科学态度。因为试题的输入输出格式是多样的，而计算精度和时效一般有严格的定义。“细节决定成败”，一些编程细节若处理不好，则会导致整个程序“功亏一篑”。下面，我们将讨论几个相关问题：

1）改进程序书写风格；

2）正确处理多组测试数据；

3）提高实数的计算精度；

4）用二分法提高计算效率。

一般来讲，较复杂的问题大都由一些含简单计算的子问题组合而成。“万丈高楼平地起”，要提高编程能力，需要从娴熟简单计算题的解题方法做起。

1.1 改进程序书写风格的实验范例

如果一个程序具有良好的书写风格，不仅在视觉上给人以美感，而且也给程序的调试和检查带来诸多方便。初看程序，往往可先从第一印象中判断出编程者的思路清晰与否。但是，怎样的程序书写风格才算“好”呢？这很难讲。不过这并不意味着程序的书写风格就没有一定规律可循。我们不妨从一个程序范例展开讨论。

【1.1.1 Financial Management】

Larry今年毕业，找到了工作，并赚了很多钱。但不知为何，Larry总感觉钱不够用。因此，Larry想用财务报表来解决他的财务问题：他要计算自己能用多少钱。现在你可以通过Larry的银行账号看到他的财务状况。请你帮助Larry写一个程序，根据过去12个月他每个月的收入，计算要达到收支平衡，他每个月平均能用多少钱。

输入

输入12行，每一行是一个月的收入，收入的数字是正数，精确到分，不用美元的符号。

输出

输出一个数字，它是这12个月收入的平均值。精确到分，前面加美元符号，后面加行结束符。在输出中没有空格或其他字符。

样例输入	样例输出
100.00 489.12 12454.12 1234.10 823.05 109.20 5.27 1542.25 839.18 83.99 1295.01 1.75	$1581.42

试题来源：ACM Mid-Atlantic 2001

在线测试地址：POJ 1004，ZOJ 1048，UVA 2362

试题解析

本题采用了非常简单的“输入-处理-输出”模式：先通过结构为for(i=0;i<12;i++)的循环输入12个月的收入a[0..11]，累计总收入sum=$\sum_{i=0}^{11}$a[i]；然后计算月均收入avg=$\frac{sum}{12}$，最后输出avg。

参考程序

```
      #include<iostream>                    //预编译命令
using namespace std;                        //使用C++标准程序库中的所有标识符
int main()                                  //主函数
{                                           //主函数开始
      double avg, sum=0.0, a[12]={0};       //定义双精度实数变量avg、sum和实数数组a的初始值
      int i;                                //声明整型变量i
      for(i=0;i<12;i++){                    //依次读入每个月的收入，并累计年收入
            cin>>a[i];
            sum+=a[i];
      }
      avg=sum/12;                           //计算月平均收入
      printf("$%.2f \n",avg);               //输出月平均收入
      return 0;
}
```

我们可从上述程序范例中得到4点启示：

1）严格按照题意要求的格式输入输出。例如，题目要求输入的月收入是精确到分的正数，因此程序中用提取操作符>>，将键盘输入的月收入存储到双精度实数类型的数组元素a[i]中；同样，程序中采用了printf("$%.2f \n",avg)语句，使得输出的月均收入精确到分，且前有美元符号，后有行结束符。程序运行于在线测试系统，关乎成败的首要因素是程序的输入输出格式是否符合题意。如果没有按照题目要求的格式输入输出，即便算法正确，结果也是零分。

2）同一顺序程序段内的所有语句（包括说明语句）与本顺序程序段的首行左对齐。

3）程序行按逻辑深度呈锯齿状排列。例如循环体缩进几个字符，用缩进表示选择结构，等等。这种锯齿型的编排格式能够清晰地反映程序结构，提高易读性。

4）在程序段前或开始位置最好加上程序段功能的注释；对于初学者，最好每条语句都加

上注释。这样做，不仅是为了调试程序和日后阅读的方便，更重要的是培养一种团队合作的精神。将来你可能参加一个研发团队，几十人甚至上百人一起合作编程，互相协助，就更需要将注释写得清清楚楚，让他人看得明明白白。

1.2 正确处理多组测试数据的实验范例

【1.1.1 Financial Management】仅给出了一组测试数据，该组测试数据的个数是已知的(12个月的收入)，且运算十分简单（累加月收入，计算月平均值）。在通常情况下，为了全面检验程序的正确性，大多试题都要求测试多组数据。如果数据组数和每组测试数据的个数都是预先确定的，则处理多组测试数据的循环结构比较简单；但问题是，若测试组数和每组测试数据的个数未知，仅知组内数据的结束标志和输入结束标志，怎么办？在数据量较大、所有测试数据都采用同一运算且运算结果的数据范围已知的情况下，有无提高计算时效的办法呢？ 我们不妨来看两个实例。

【1.2.1 Doubles】

给出2到15个不同的正整数，计算在这些数里面有多少数据对满足一个数是另一个数的两倍。比如给出

1　4　3　2　9　7　18　22

答案是3，因为2是1的两倍，4是2的两倍，18是9的两倍。

输入

输入包括多组测试数据。每组数据包括一行，给出2到15个两两不同且大于0小于100的正整数。每一行最后一个数是0，表示这一行的结束。输入的最后一行只包括一个整数−1，这行表示输入数据的结束，不用进行处理。

输出

对每组输入数据，输出一行，给出有多少个数对满足其中一个数是另一个数的两倍。

样例输入	样例输出
1 4 3 2 9 7 18 22 0 2 4 8 10 0 7 5 11 13 1 3 0 −1	3 2 0

试题来源：ACM Mid-Central USA 2003

在线测试地址：POJ 1552，ZOJ 1760，UVA 2787

试题解析

本题包含多组测试数据，因此需循环处理各组数据，循环的结束标志是当前数据组的首个数是−1。循环体内做两项工作：

1）通过一重循环读入当前数据组a，并累计数据元素个数n。循环的结束标志是读入数据0；

2）通过两重循环结构枚举组内的所有数据对a[i]和a[j]（0≤i<n−1，i+1≤j<n），判断a[i]和a[j]是否呈两倍关系（a[i]*2==a[j] || a[j]*2==a[i]）。

参考程序

```
#include <iostream>                    //预编译命令
using namespace std;                   //使用C++标准程序库中的所有标识符
int main()                             //主函数
```

```
{                                                    //主函数开始
  int i, j, n, count, a[20];                         //声明整型变量i、j、n、count和整型数组a
  cin>>a[0];                                         //输入第1组的首个数据
  while(a[0]!=-1)                                    //若输入未结束，则输入新一组数据
  {  n=1;                                            //读入当前数据组
     for( ; ; n++)
     {
       cin>>a[n];
       if (a[n]==0) break;
     }
     count=0;                    //处理：计算当前数据组里有多少数据对满足一个数是另一个数的两倍
     for (i=0; i<n-1; i++)                           //枚举所有数据对
     {
       for (j=i+1; j<n; j++)
       {
          if (a[i]*2==a[j] || a[j]*2==a[i])        //若当前数据对满足2倍关系，则累计
          count++;
       }
     }
     cout<<count<<endl;                              //输出当前数据组里满足2倍关系的数据对数
     cin>>a[0];                                      //输入下一组的首个数据
  }
  return 0;
}
```

由题意可以看出，在测试数据组数和组内长度未知的情况下，输入一般是一个测试数据组为一行。求解程序大都采用双重循环结构：

外循环：枚举各组测试数据，结束标志为输入结束符（本题的输入结束符为−1）；

内循环：输入和处理当前组的测试数据，输入的结束标志为数据组的结束符（本题数据组的结束符为0）。

在处理多组测试数据的过程中，可能会遇到这样一种情况：数据量较大，且所有测试数据都采用同一运算，运算结果的数据范围已知。在这种情况下，为了提高计算时效，不妨采用离线计算方法：预先计算出指定范围内的所有解，存入某个常量数组。以后每测试一组数据，直接从常量数组中引用相关数据就可以了，避免重复运算。

【1.2.2 Sum of Consecutive Prime Numbers】

一些正整数能够表示为一个或多个连续素数的和。给出一个正整数，有多少个这样的表示？例如，整数53有两个表示5+7+11+13+17和53；整数41有三个表示 2+3+5+7+11+13，11+13+17，41；整数3只有一个表示3；整数20没有这样的表示。注意加法操作数必须是连续的素数，因此，对于整数20、7+13 和 3+5+5+7，都不是有效的表示。

请写一个程序，对于一个给出的正整数，给出连续素数的和的表示数。

输入

输入一个正整数序列，每个数一行，在2到10000之间取值。输入结束以0表示。

输出

输出的每一行对应输入的每一行，除了最后的0。输出的每一行，对于一个输入的正整数，给出连续素数的和的表示数。输出中没有其他的字符。

样例输入	样例输出
2	1
3	1
17	2
41	3

（续）

样例输入	样例输出
20 666 12 53 0	0 0 1 2

试题来源：ACM Japan 2005

在线测试地址：POJ 2739，UVA 3399

试题解析

由于每组测试数据都要计算素数，且素数上限为10000，因此我们采用离线计算方法，先预处理出所有10001以内的素数，按照递增顺序存入数组prime[1..total]。然后依次处理每个测试数据。设

当前输入为n；

连续素数的和为cnt；

cnt=n的表示数为ans。

采用双重循环计算n的表示数：

外循环i：通过for (int i=0; n>=prime[i]; i++)的循环结构枚举所有可能的最小素数prime[i]；

内循环j：通过for (int j=i; j < total && cnt<n; j++) cnt += prime[j]的循环结构计算连续素数的和cnt，内循环结束时cnt≥n。若cnt=n，则连续素数的和的表示数为ans+1，继续外循环。

外循环结束后得出的ans即为问题解。

参考程序

```
#include <iostream>                          //预编译命令
using namespace std;                         //使用C++标准程序库中的所有标识符
const int maxp = 2000, n = 10000;            //设定素数表长和输入值的上限
int prime[maxp], total = 0;                  //素数表和表长初始化为0

bool isprime(int k)                          //判定k是否为素数
{
    for (int i = 0; i < total; i++)
        if (k % prime[i] == 0)
            return false;
    return true;
}

int main(void)                               //主函数
{                                            //主函数开始
    for (int i = 2; i <= n; i++)             //预先建立素数表
      if (isprime(i))
            prime[total++] = i;
    prime[total] = n + 1;

    int m;
    cin >> m;                                //输入第1个正整数
    while (m) {                              //循环，直至输入正整数0为止
        int ans = 0;                         //和初始化为0
        for (int i = 0; m >= prime[i]; i++) { //枚举最小素数
            int cnt = 0;                     //求连续素数的和
```

```
            for (int j = i; j < total && cnt < m; j++)
                cnt += prime[j];
            if (cnt == m)                    //若和恰等于m，则累计答案数
                ++ans;
        }
        cout << ans << endl;                 //输出答案数
        cin >> m;                            //输入下一个正整数
    }
    return 0;
}
```

1.3 提高实数精度的实验范例

在有些情况下，试题的数据对象是实数且蕴含实数运算。例如，如何判断两个实数相等，对实数取整究竟是取大于该实数的最小整数还是取小于该实数的最大整数，等等。任何程序设计语言的实数精度都是有限的。若试题要求输出实数值的误差小于语言规定的精度范围，怎么办？程序若不能处理好这些细节问题，即便计算步骤正确，也可能导致错误结果。我们不妨看一个实例。

【1.3.1 I Think I Need a Houseboat】

Fred Mapper考虑在Louisiana购买一些土地，并在土地上建他的家。在对土地的调查中他发现，由于Mississippi河的侵蚀，Louisiana州的土地每年减少50平方英里。因为Fred准备在他所建的家中度过他的后半生，所以他要知道是否他的土地会因为河流的侵蚀而丧失。

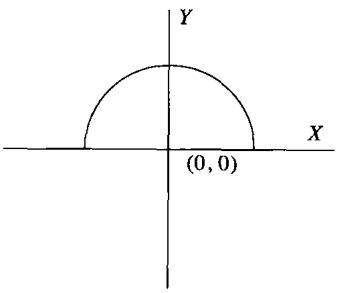


图 1.3-1

在做了很多的研究后， Fred发现正在失去的土地构成一个半圆形（半圆如图1.3-1所示）。这一半圆形是一个圆的一部分，圆心在(0，0)，二等分这个圆的线是X轴，X轴的下方是水。在第1年开始的时候，这一半圆的面积为0。

输入

输入的第一行是一个正整数，表示有多少个测试数据集（N）。后面有N行，每行给出笛卡儿坐标X和Y，表示Fred考虑购买的土地的位置。这些数是浮点数，以英里为单位。Y坐标非负。不会给出（0，0）。

输出

对每个输入的测试数据集，输出一行。这一行的形式为“Property N: This property will begin eroding in year Z.”，其中N是数据集（从1开始记数），Z是他的土地到第Z年结束的时候要落到半圆形中的那一年（从1开始记数），Z必须是一个整数。在最后一个数据集后，输出“END OF OUTPUT.”。

样例输入	样例输出
2 1.0 1.0 25.0 0.0	Property 1: This property will begin eroding in year 1. Property 2: This property will begin eroding in year 20. END OF OUTPUT.

说明：

1）购买的土地不会在半圆形边界上：或者落在半圆形内，或者落在半圆形外。

2）这一问题被自动裁判，所以输出要精确匹配，包括大小写、标点符号和空格以及到行末的完整的那一句句子。

3）所有的地点以英里为单位。

试题来源：ACM Mid-Atlantic 2001

在线测试地址：POJ 1005，ZOJ 1049，UVA 2363

试题解析

由于测试数据组数n预先确定，且每组测试数据仅为笛卡儿坐标，因此可直接采用for循环处理所有测试数据组。第i个测试组（X_i，Y_i）与圆心（0，0）构成的半圆面积即为土地被淹的范围。由于每年减少50平方英里土地，而年份是整数，因此淹没（X_i，Y_i）的年份应为大于$\frac{\text{半圆面积}}{50}$的最小整数，这个取整过程应使用向上取整函数ceil(x)。若使用向下取整函数floor(x)，则可能提前1年失去土地。

参考程序

```
#include <stdio.h>                              //预编译命令
#include <math.h>

int num_props;                                  //定义测试数据组数为整数
float x, y;                                     //定义笛卡儿坐标为单精度实数
int i;
double calc;                                    //定义“半圆面积/50”为双精度实数
int years;                                      //定义失去土地的年份为整数

void main(void)                                 //主函数
{                                               //主函数开始
     scanf("%d", &num_props);                   //输入测试数据组数
     for (i = 1; i <= num_props; i++)
     {
       scanf("%f %f", &x, &y);                  //输入第i个考虑购买的土地位置
       calc = (x*x + y*y)* M_PI / 2 / 50; //计算“半圆面积/50”，上取整后即为土地失去的年份
       years = ceil(calc);
       printf("Property %d: This property will begin eroding in year %d.\n", i, years);//输出
     }
     printf("END OF OUTPUT.\n");
}
```

在实数运算中，经常需要判断实数x和实数y是否相等，而编程者往往把判断条件简单设成y−x是否等于0。实际上，这种写法是有失偏颇的，可能会产生精度误差。避免精度误差的办法是设一个精度常量delta。若y−x的实数值与0之间的区间长度小于delta，则认定x和y相等，这样就可将误差控制在delta范围内（图1.3-2）。

显然，判断实数x和实数y是否相等的条件应设成|y−x|≤delta。程序范例可看【1.4.1 Hangover】。

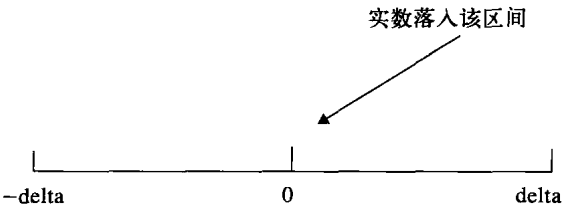


图 1.3-2

1.4 使用二分法提高计算时效的实验范例

在有些情况下，问题的所有数据对象为一个有序区间。二分法将这个区间等分成两个子区

间，根据计算要求决定下一步计算是在左子区间还是在右子区间进行；然后再根据计算要求等分所在区间，直至找到解为止。显然，对一个规模为$O(n)$的问题，如果采用盲目枚举的办法，则效率为$O(n)$；若采用二分法，则计算效率可提高至$O(\log_2 n)$。

许多算法都采用了二分法。例如二分法查找、减半递推技术、快速排序、合并排序、最优二叉树、线段树等。其中相对比较浅显的算法是二分法查找和减半递推技术，使用这两种方法解简单计算题，可以显著提高计算时效。

二分法查找的基本思想是：假设数据是按升序排序的，对于待查找值x，从序列的中间位置开始比较，若当前中间位置值等于x，则查找成功；若x小于当前中间位置值，则在数列的左子区间（数列的前半段）中查找；若x大于当前中间位置值，则在右子区间（数列的后半段）中继续查找。依此类推，直到找到x在序列中的位置为止。

【1.4.1 Hangover】

你能使一叠在桌子上的卡片向桌子外伸出多远？如果是一张卡片，这张卡片最多可以向桌子外伸出卡片的一半长度（卡片以直角伸出桌子）。如果有两张卡片，就让上面一张卡片向外伸出下面那张卡片的一半长度，而下面那张卡片伸出桌子卡片的三分之一长度，所以两张卡片总的向外延伸1/2 + 1/3 = 5/6卡片长度。依此类推，n张卡片向外延伸1/2 + 1/3 + 1/4 + … + 1/(n + 1)卡片长度，最上面的向外延伸1/2，第二张向外延伸1/3，第三张向外延伸1/4，…，最下面一张向外延伸1/(n + 1)，如图1.4-1所示。

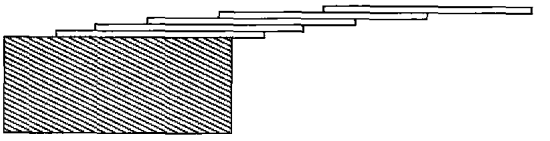


图 1.4-1

输入

输入由一个或多个测试数据组成，最后一行用0.00表示输入结束，每个测试数据占据一行，是一个3位的正浮点数c，最小值0.01，最大值5.20。

输出

对每个测试数据c，输出要至少伸出超过c的卡片长度所最少要用的卡片的数目，输出形式见样例输出。

样例输入	样例输出
1.00	3 card(s)
3.71	61 card(s)
0.04	1 card(s)
5.19	273 card(s)
0.00	

试题来源：ACM Mid-Central USA 2001

在线测试地址：POJ 1003，ZOJ 1045，UVA 2294

试题解析

由于数据范围很小，因此可先离线计算出截止长度不超过5.20所需的最少卡片数。设total为卡片数，len[i]为前i张卡片向外延伸的长度，len[i] = len[i − 1] + $\frac{1}{i+1}$。显然len为递增序列。

注意：由于len的表元素和被查找的x为实数，因此要严格控制精度误差。设精度delta=1e−

8。zero(x)为实数x为正负数和0的标志：

$$\text{zero}(x)=\begin{cases}1 & x > \text{delta}，\text{标志x是正实数}\\ -1 & x < -\text{delta}，\text{标志x是负实数}\\ 0 & \text{是否标志x是0}\end{cases}$$

初始时len[0]=0，通过结构为

```
for(total=1;zero(len[total-1]-5.20)<0;total++)
    len[total]=len[total-1]+1.0/double(total+1)
```

的循环递推len序列。

在计算出len数组后，先输入第1个测试数据x，并进入结构为while (zero(x))的循环，每一次循环，使用二分法在len表中查找至少伸出卡片长度x所最少要用的卡片数，并输入下一个测试数据x。这个循环过程直至输入测试数据x=0 .00为止。

二分查找的过程如下：

初始时区间[l，r]=[1，total]，区间的中间指针min=$\left\lfloor\frac{l+r}{2}\right\rfloor$。若zero(len[mid] − x) < 0，则所需的卡片数在右区间（l=mid）；否则所需的卡片数在左区间（r=mid）。继续二分区间[l，r]，直至l+1≥r为止。此时得出的r即为最少要用的卡片数。

参考程序

```
#include <iostream>                          //预编译命令
using namespace std;                         //使用C++标准程序库中的所有标识符
const int maxn = 300;                        //len数组容量
const double delta = 1e-8;                   //设定精度

int zero(double x)                           //在精度delta的范围内，若x是小于0的负实数，则返
                                             //回-1；若x是大于0的正实数，则返回1；若x为0，则返回0
{
    if (x < -delta)
        return -1;
    return x > delta;
}


int main(void)                               //主函数
{                                            //主函数开始
    double len[maxn];                        //定义len数组和数组长度
    int total;

    len[0] = 0.0;                            //直接计算出截止长度不超过5.20所需的最少卡片数
    for (total = 1; zero(len[total - 1] - 5.20) < 0; total++)
        len[total] = len[total - 1] + 1.0 / double(total + 1);

    double x;
    cin >> x;                                //输入第1个测试数据x
    while (zero(x)) {                        //用二分法在len表中查找不小于x的最少卡片数
        int l, r;
        l = 0;                               //设定查找区间的左右指针
        r = total;
        while (l + 1 < r) {
            int mid = (l + r) / 2;           //计算查找区间的中间指针
            if (zero(len[mid] - x) < 0)      //若中间元素值小于x，则在右区间查找；否则在左区间查找
                l = mid;
            else
```