第1章 练习题参考答案

1.1 第1章 绪论

扫一扫

习题＋答案

1.2 第2章 线性表

扫一扫

习题＋答案

1.3 第3章 栈和队列

扫一扫

习题＋答案

1.4 第4章 串

扫一扫

习题＋答案

1.5 第5章 递归

扫一扫

习题＋答案

1.6 第6章 数组和稀疏矩阵

扫一扫

习题＋答案

1.7 第7章 树和二叉树

扫一扫

习题＋答案

1.8 第8章 图

扫一扫

习题＋答案

1.9 第9章 查找

扫一扫

习题＋答案

1.10 第10章 排序

扫一扫

习题＋答案

第2章 上机实验题参考答案

2.1 第1章 绪论

1. 从 https://docs.oracle.com/网站下载并安装 Java 1.8 或更高收本，访问该网站中的 javase/8/docs/阅读相关技术文档。

解：略。

2. 编写一个 Java 程序，求一元二次方程 $ax^2+bx+c=0$ 的根，并用相关数据测试。

解：设计 solve()方法求方程的根，用 Solution 类对象 s 返回结果，用 disp()方法输出结果。对应的程序如下：

```
class Solution {                                    //存放解
    int cnt;                                        //根的个数
    double x1;                                      //根 1
    double x2;                                      //根 2
}
public class Exp2 {
    static Solution solve(double a,double b,double c) {     //求方程的根
        Solution s=new Solution();
        double d=b*b-4*a*c;
        if(d<0)
            s.cnt=0;
        else if(Math.abs(d)<=0.0001) {                  //等于 0
            s.cnt=1;
            s.x1=(-b+Math.sqrt(d))/(2*a);
        }
        else {
            s.cnt=2;
            s.x1=(-b+Math.sqrt(d))/(2*a);
            s.x2=(-b-Math.sqrt(d))/(2*a);
        }
        return s;
    }
    static void disp(Solution s) {                  //输出结果
        if(s.cnt==0)
```

```
            System.out.println("无根");
        else if(s.cnt==1)
            System.out.printf("一个根为%.1f\n",s.x1);
        else
            System.out.printf("两个根为%.1f和%.1f\n",s.x1,s.x2);
    }
    public static void main(String[] args) {
        Solution s;
        double a=2,b=-3,c=4;
        System.out.printf("\n  测试 1\n");
        System.out.printf("     a=%.1f, b=%.1f, c=%.1f ",a,b,c);
        s=solve(a,b,c);
        disp(s);
        a=1; b=-2; c=1;
        System.out.printf("\n  测试 2\n");
        System.out.printf("     a=%.1f, b=%.1f, c=%.1f ",a,b,c);
        s=solve(a,b,c);
        disp(s);
        a=2; b=-1; c=-1;
        System.out.printf("\n  测试 3\n");
        System.out.printf("     a=%.1f, b=%.1f, c=%.1f ",a,b,c);
        s=solve(a,b,c);
        disp(s);
    }
}
```

上述程序的执行结果如图 2.1 所示。

3. 求 1+(1+2)+(1+2+3)+…+(1+2+3+…+n)有 3 种解法，解法 1 是采用两重迭代，依次求出(1+2+…+i)后累加；解法 2 是采用一重迭代求和，利用 $i(i+1)/2$ 求和后再累加；解法 3 是直接利用$n(n+1)(n+2)/6$ 求和。

编写一个 Java 程序，利用上述解法求 $n=1000$ 的结果，并且给出各种解法的运行时间。

解：3 种解法的方法分别是 solve1、solve2 和 solve3，对应的实验程序如下：

```
public class Exp3 {
    static long solve1(int n) {                          //解法 1
        long sum=0;
        for(int i=1;i<=n;i++) {
            for(int j=1;j<=i;j++) sum+=j;
        }
        return sum;
    }
    static long solve2(int n) {                          //解法 2
        long sum=0,sum1=0;
        for(int i=1;i<=n;i++) {
            sum1+=i;
            sum+=sum1;
        }
        return sum;
    }
    static long solve3(int n) {                          //解法 3
        long sum=n*(n+1)*(n+2)/6;
```

```
        return sum;
    }
    public static void main(String[] args) {
        int n=1000;
        System.out.printf("\n n=%d\n",n);
        long startTime=System.nanoTime();              //获取开始时间
        System.out.println(" 解法 1 sum1="+solve1(n));
        long endTime=System.nanoTime();                //获取结束时间
        System.out.println(" 运行时间: "+(endTime-startTime)+"ns");
        startTime=System.nanoTime();                   //获取开始时间
        System.out.println(" 解法 2 sum2="+solve2(n));
        endTime=System.nanoTime();                     //获取结束时间
        System.out.println(" 运行时间: "+(endTime-startTime)+"ns");
        startTime=System.nanoTime();                   //获取开始时间
        System.out.println(" 解法 3 sum3="+solve3(n));
        endTime=System.nanoTime();                     //获取结束时间
        System.out.println(" 运行时间: "+(endTime-startTime)+"ns");
    }
}
```

上述程序的一次执行结果如图 2.2 所示。解法 2 和解法 3 差别不大的原因是因为执行输出语句的时间总是相同的。

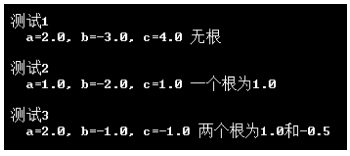

图 2.1　第 1 章实验题 2 的执行结果

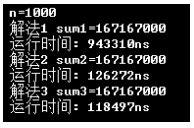

图 2.2　第 1 章实验题 3 的一次执行结果

2.2　第 2 章　线性表

1. 编写一个简单学生成绩管理程序，每个学生记录包含学号、姓名、课程和分数成员，采用顺序表存储，完成以下功能。

（1）屏幕显示所有学生记录。

（2）输入一个学生记录。

（3）按学号和课程删除一个学生记录。

（4）按学号排序并输出所有学生记录。

（5）按课程排序，一门课程学生按分数递减排序。

解：首先设计学生记录类 Stud，包含学生学号、姓名、课程和分数成员，以及构造方法、输出一个学生记录的方法和用于排序的 get 方法。

```
class Stud {                                    //学生记录类
    int no;                                     //学号
    String name;                                //姓名
    String course;                              //课程
```

```
    int fraction;                                                    //分数
    public Stud(int no1,String name1,String course1,int fraction1) {    //构造方法
        no=no1;
        name=name1;
        course=course1;
        fraction=fraction1;
    }
    public int getno() {
        return no;
    }
    public String getcourse() {
        return course;
    }
    public int getfraction() {
        return fraction;
    }
    public void disp() {
        System.out.println("  学号:"+no+"  姓名:"+name+"  课程:"+course+"  分数:"+fraction);
    }
}
```

然后设计包括相关基本运算算法的 StudList 类,其中采用 ArrayList 类对象 sl 作为学生顺序表,存放输入的所有学生记录。学生记录的插入、删除和排序等直接利用 ArrayList 类的方法实现。

```
class StudList {
    ArrayList<Stud> sl;                          //顺序表
    public StudList() {                          //构造方法
        sl=new ArrayList<Stud>();
    }
    public void Addstud() {                      //输入一个学生记录
        Scanner in=new Scanner(System.in);       //使用 Scanner 类定义对象
        System.out.println("  输入一个学生记录");
        System.out.print("  学号: ");
        int no1=in.nextInt();
        System.out.print("  姓名: ");
        String name1=in.next();
        System.out.print("  课程: ");
        String course1=in.next();
        System.out.print("  分数: ");
        int fraction1=in.nextInt();
        sl.add(new Stud(no1,name1,course1,fraction1));
    }
    public void Dispstud() {                     //输出所有学生记录
        if(sl.size()>0) {
            System.out.println("  ** 所有学生记录");
            for(int i=0;i<sl.size();i++)
                sl.get(i).disp();
        }
        else System.out.println("  ** 没有任何学生记录");
    }
    public void Delstud() {                      //删除指定学号指定课程的学生记录
```

```
        Scanner in=new Scanner(System.in);                    //使用 Scanner 类定义对象
        System.out.print("  删除的学号: ");
        int no1=in.nextInt();
        System.out.print("  课程: ");
        String course1=in.next();
        int i=0;
        boolean find=false;
        while(i<sl.size()) {
            Stud s=sl.get(i);
            if(s.no==no1 && s.course.equals(course1)) {
                find=true;
                break;
            }
            i++;
        }
        if(find) {
            sl.remove(i);
            System.out.println("  ** 成功删除该学号学生的成绩记录");
        }
        else System.out.println("  ** 没有找到该学生的成绩记录");
    }
    public void Sort1() {                                     //按学号递增排序
        sl.sort(Comparator.comparing(Stud::getno));
        Dispstud();
    }
    public void Sort2() {                                     //按课程、分数递减排序
        sl.sort(Comparator.comparing(Stud::getcourse).
            thenComparing(Stud::getfraction).reversed());
        Dispstud();
    }
}
```

最后设计包含主方法的 Exp1 类如下：

```
public class Exp1 {
    public static void main(String[] args) throws IOException,NotSerializableException {
        int sel;
        StudList L=new StudList();
        Scanner in=new Scanner(System.in);                    //使用 Scanner 类定义对象
        while(true) {
            System.out.print("1.显示全部记录 2.输入 3.删除 4.学号排序 5.课程排序 0:退出");
            System.out.print("  请选择: ");
            sel=in.nextInt();
            switch(sel) {
                case 0:break;
                case 1:L.Dispstud();break;
                case 2:L.Addstud();break;
                case 3:L.Delstud();break;
                case 4:L.Sort1();break;
                case 5:L.Sort2();break;
            }
            if(sel==0) break;
        }
    }
}
```

上述程序的执行结果如图 2.3 所示。

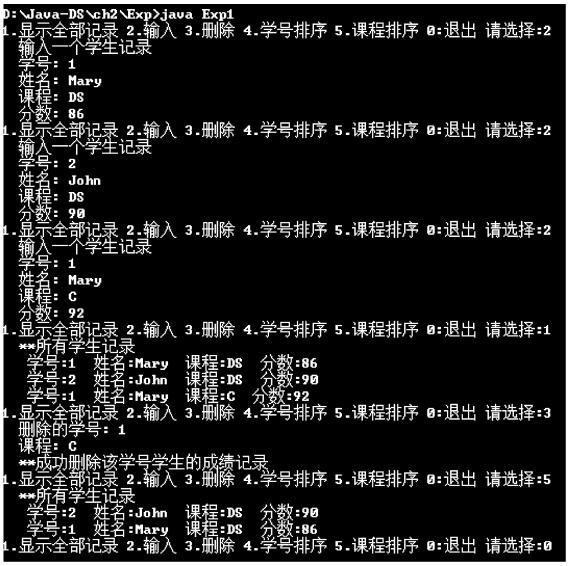

图 2.3 第 2 章实验题 1 的执行结果

2. 在《教程》中 2.5 节求解两个多项式相加运算的基础上编写一个实验程序，采用单链表存放多项式，实现两个多项式相乘运算，通过相关数据进行测试。

解：以两个多项式 $p(x)=2x^3+3x^2+5$ 和 $q(x)=2x+1$ 相乘为例说明，用多项式对象 $L1$ 存放 $p(x)$，多项式对象 $L2$ 存放 $q(x)$，多项式对象 $L3$ 存放相乘的结果，先置 $L3$ 为空。其过程如下：

(1) 由 $L1$ 的第 1 个多项式项 $2x^3$ 与 $L2$ 相乘得到多项式对象 tmpL，tmpL$=4x^4+2x^3$。将 $L3$ 与 tmpL 相加，得到 $L3=4x^4+2x^3$。注意，由于每个多项式中没有相同指数的项，所以生成的 tmpL 中也一定没有相同指数的项。

(2) 由 $L1$ 的第 2 个多项式项 $3x^2$ 与 $L2$ 相乘得到多项式对象 tmpL，tmpL$=6x^3+3x^2$。将 $L3$ 与 tmpL 相加，得到 $L3=4x^4+8x^3+3x^2$。

(3) 由 $L1$ 的第 3 个多项式项 5 与 $L2$ 相乘得到多项式对象 tmpL，tmpL$=10x+5$。将 $L3$ 与 tmpL 相加，得到 $L3=4x^4+8x^3+3x^2+10x+5$。

这样，得到的 $L3$ 就是最终结果。设计以下两个方法分别用于求 $L1$ 中一个多项式项 p 与 L 相乘的多项式和求两个多项式 $L1$、$L2$ 相乘运算的结果：

```
public static PolyClass aMulti(PolyNode p,PolyClass L):求 p 与 L 相乘的多项式
public static PolyClass Multi(PolyClass L1,PolyClass L2):两个多项式相乘的运算
```

完整的实验程序如下：

```
import java.util.*;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintStream;
class PolyNode {                                    //单链表结点类型
    public double coef;                             //系数
```

```
    public int exp;                                    //指数
    public PolyNode next;                              //指针成员
    PolyNode() {                                       //构造方法
        next=null;
    }
    PolyNode(double c,int e) {                         //重载构造方法
        coef=c;
        exp=e;
        next=null;
    }
}
class PolyClass {                                      //多项式单链表类
    PolyNode head;                                     //存放多项式单链表头结点
    public PolyClass() {                               //构造方法
        head=new PolyNode();                           //建立头结点 head
    }
    public void CreatePoly(double[] a,int[] b,int n) {  //采用尾插法建立多项式单链表
        PolyNode s,t;
        t=head;                                        //t 始终指向尾结点,开始时指向头结点
        for(int i=0;i<n;i++) {
            s=new PolyNode(a[i],b[i]);                 //创建新结点 s
            t.next=s;                                  //在 t 结点之后插入 s 结点
            t=s;
        }
        t.next=null;                                   //尾结点的 next 域置为 null
    }
    public void Sort() {                               //对多项式单链表按 exp 域递减排序
        PolyNode p,pre,q;
        q=head.next;                                   //q 指向开始结点
        if(q==null) return;                            //原单链表空时返回
        p=head.next.next;                              //p 指向 q 结点的后继结点
        if(p==null) return;                            //原单链表只有一个数据结点时返回
        q.next=null;                                   //构造只含一个数据结点的有序单链表
        while(p!=null) {
            q=p.next;                                  //q 用于临时保存 p 结点的后继结点
            pre=head;                                  //pre 指向插入结点 p 的前驱结点
            while(pre.next!=null && pre.next.exp>p.exp)
                pre=pre.next;                          //在有序表中找插入结点 p 的前驱结点 pre
            p.next=pre.next;                           //在 pre 结点之后插入 p 结点
            pre.next=p;
            p=q;                                       //扫描原单链表余下的结点
        }
    }
    public void DispPoly() {                           //输出多项式单链表
        boolean first=true;                            //first 为 true 表示是第一项
        PolyNode p=head.next;
        while(p!=null) {
            if(first) first=false;
            else if(p.coef>0) System.out.print("+");
            if(p.exp==0)                               //指数为 0 时不输出"x"
                System.out.print(p.coef);
            else if(p.exp==1)                          //指数为 1 时不输出指数
                System.out.print(p.coef+"x");
```

```
            else
                System.out.print(p.coef+"x^"+p.exp);
            p=p.next;
        }
        System.out.println();
    }
}
public class Exp2{
    public static PolyClass Add(PolyClass L1,PolyClass L2) {     //两个多项式相加的运算
        PolyNode p,q,s,t;                                        //t 始终指向 L3 的尾结点
        double c;
        PolyClass L3=new PolyClass();
        t=L3.head;
        p=L1.head.next;
        q=L2.head.next;
        while(p!=null && q!=null) {
            if(p.exp>q.exp) {                                    //L1 的结点的指数较大
                s=new PolyNode(p.coef,p.exp);
                t.next=s; t=s;
                p=p.next;
            }
            else if(p.exp<q.exp) {                               //L2 的结点的指数较大
                s=new PolyNode(q.coef,q.exp);
                t.next=s; t=s;
                q=q.next;
            }
            else {                                               //两个结点的指数相等
                c=p.coef+q.coef;                                 //求两指数相等结点的系数和 c
                if(c!=0) {                                       //系数和 c 不为 0 时
                    s=new PolyNode(c,p.exp);                     //新建结点 s
                    t.next=s; t=s;                               //将结点 s 链接到 L3 末尾
                }
                p=p.next; q=q.next;
            }
        }
        t.next=null;                                             //尾结点的 next 域置为 null
        if(p!=null) t.next=p;
        if(q!=null) t.next=q;
        return L3;
    }
    public static PolyClass aMulti(PolyNode p,PolyClass L) {     //求 p 与 L 相乘的多项式
        PolyClass L1=new PolyClass();
        PolyNode q=L.head.next,s,t;
        t=L1.head;                                               //t 始终指向 L1 的尾结点
        double c;
        int e;
        while(q!=null) {
            c=p.coef*q.coef;                                     //两项系数相乘
            e=p.exp+q.exp;                                       //两项指数相加
            s=new PolyNode(c,e);                                 //建立新结点 s
            t.next=s; t=s;                                       //s 结点链接到 L1 末尾
            q=q.next;
        }
```

```
        t.next=null;                                    //尾结点的 next 域置为 null
        return L1;                                      //返回 L1
    }
    public static PolyClass Multi(PolyClass L1,PolyClass L2) {      //两个多项式相乘的运算
        PolyNode p,q,s,t;
        double c;
        PolyClass L3=new PolyClass(),tmpL;
        t=L3.head;                                      //t 始终指向 L3 的尾结点
        p=L1.head.next;
        while(p!=null) {                                //遍历 L1 的每个多项式项
            tmpL=aMulti(p,L2);                          //p 结点多项式项与 L2 相乘得到 tmpL
            L3=Add(L3,tmpL);                            //L3 与 tmpL 相加得到 L3
            p=p.next;
        }
        return L3;                                      //返回 L3
    }
    public static void main(String[] args) throws FileNotFoundException {
        System.setIn(new FileInputStream("abc.in"));    //将标准输入流重定向至 abc.in
        Scanner fin = new Scanner(System.in);
        System.setOut(new PrintStream("abc.out"));      //将标准输出流重定向至 abc.out
        PolyClass L1=new PolyClass();
        PolyClass L2=new PolyClass();
        PolyClass L3;
        double[] a=new double[100];
        int[] b=new int[100];
        int n;
        n=fin.nextInt();                                //输入第 1 个多项式的 n
        for(int i=0;i<n;i++)                            //输入第 1 个多项式系数数组 a
            a[i]=fin.nextDouble();
        for(int i=0;i<n;i++)                            //输入第 1 个多项式指数数组 b
            b[i]=fin.nextInt();
        L1.CreatePoly(a,b,n);                           //创建第 1 个多项式单链表
        System.out.print("第 1 个多项式: "); L1.DispPoly();
        L1.Sort();                                      //第 1 个多项式按指数递减排序
        System.out.print("排序后结果: "); L1.DispPoly();
        n=fin.nextInt();                                //输入第 2 个多项式的 n
        for(int i=0;i<n;i++)                            //输入第 2 个多项式系数数组 a
            a[i]=fin.nextDouble();
        for(int i=0;i<n;i++)                            //输入第 2 个多项式指数数组 b
            b[i]=fin.nextInt();
        L2.CreatePoly(a,b,n);                           //创建第 2 个多项式单链表
        System.out.print("第 2 个多项式: "); L2.DispPoly();
        L2.Sort();                                      //第 2 个多项式按指数递减排序
        System.out.print("排序后结果: "); L2.DispPoly();
        L3=Multi(L1,L2);                                //两个多项式相乘
        System.out.print("相乘后多项式: "); L3.DispPoly();
    }
}
```

实验输入文件 abc.in 如下：

```
3
2 3 5
```

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

程序执行后产生的输出文件 abc.out 如下：

```
第 1 个多项式: 2.0x^3+3.0x^2+5.0
排序后结果: 2.0x^3+3.0x^2+5.0
第 2 个多项式: 2.0x+1.0
排序后结果: 2.0x+1.0
相乘后多项式: 4.0x^4+8.0x^3+3.0x^2+10.0x+5.0
```

3. 重新排列一个单链表的结点顺序。给定一个单链表 L 为 $L_0 \rightarrow L_1 \rightarrow \cdots L_{n-1} \rightarrow L_n$，将 L 排列成 $L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \cdots$，不能够修改结点值。例如，给定 L 为(1,2,3,4)，重新排列后为(1,4,2,3)。

定义单链表的结点类型如下：

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
        val=x;
        next=null;
    }
}
```

实现的方法如下：

```
public class Solution{
    public void reorderList(ListNode head) {
        …
    }
}
```

解：这里改为本地编程，关键是 reorderList(head)方法的设计，其中的单链表是不带头结点的，与 head 标识整个单链表。reorderList(head)的过程如下。

(1) 断开：通过快慢指针 slow 和 fast 找到中间结点，从中间结点断开，即置 head1=solw.next 构成后半部分的不带头结点的单链表 head1，置 slow.next=null 构成前半部分的不带头结点的单链表 head。

(2) 逆置：采用头插法将 head1 逆置。这里注意逆置不带头结点的单链表与逆置带头结点的单链表稍有不同。

(3) 合并：采用尾插法，head 为首结点，p=head.next，q=head1，t 指向 head 结点，在 p 或和 q 不空时循环。先将 q 结点链接到 t 结点的后面，再将 p 结点链接到 t 结点的后面，最后置 t.next 为空。

对应的完整程序(本机运行)如下：

```
class ListNode {                                    //单链表结点类
    int val;
    ListNode next;
    ListNode(int x) {
        val=x;
        next=null;
    }
}
class LinkList {                                    //单链表类
    ListNode head;
    public LinkList() {                             //构造方法
        head=null;
    }
    public void CreateList(int[] a) {               //尾插法:由数组 a 整体建立不带头结点的单链表
        ListNode s,t;
        head=new ListNode(a[0]);
        t=head;                                     //t 始终指向尾结点,开始时指向头结点
        for(int i=1;i<a.length;i++) {               //循环建立数据结点 s
            s=new ListNode(a[i]);                   //新建存放 a[i]元素的结点 s
            t.next=s;                               //将 s 结点插入 t 结点之后
            t=s;
        }
        t.next=null;                                //将尾结点的 next 域置为 null
    }
    public String toString() {                      //将线性表转换为字符串
        String ans="";
        ListNode p=head;
        while(p!=null) {
            ans+=p.val+" ";
            p=p.next;
        }
        return ans;
    }
}
class Solution {                                    //求解类
    public void reorderList(ListNode head) {
        if(head==null || head.next==null || head.next.next==null)
            return;
        //(1)断开
        ListNode slow=head,fast=head;
        while(fast.next!=null && fast.next.next!=null) {
            slow=slow.next;                         //找到中间结点 slow
            fast=fast.next.next;
        }
        ListNode head1=slow.next;                   //head1 为后半部分单链表的首结点
        slow.next=null;                             //断开
        //将 head1 单链表逆置
        ListNode p=head1,q;
        head1=null;
        while(p!=null) {
            q=p.next;
            p.next=head1;
            head1=p;
```

```
            p=q;
        }
        //合并操作
        p=head.next;                                    //head 含 L0,p 遍历 head 的其他结点
        q=head1;                                        //q 遍历 head1
        ListNode t=head;
        while(p!=null || q!=null) {
            if(q!=null) {
                t.next=q;
                t=q;
                q=q.next;
            }
            if(p!=null) {
                t.next=p;
                t=p;
                p=p.next;
            }
        }
        t.next=null;
    }
}
public class Exp3 {
    public static void main(String args[]) {
        System.out.println("测试 1");
        int[] a={1,2,3,4,5};
        LinkList L=new LinkList();
        L.CreateList(a);
        System.out.println("L: "+L.toString());      //输出:1 2 3 4 5
        Solution obj=new Solution();
        obj.reorderList(L.head);
        System.out.println("重新排列结点");
        System.out.println("L: "+L.toString());      //输出:1 5 2 4 3
        System.out.println("测试 2");
        int[] b={1,2,3,4,5,6};
        L.CreateList(b);
        System.out.println("L: "+L.toString());      //输出:1 2 3 4 5 6
        obj.reorderList(L.head);
        System.out.println("重新排列结点");
        System.out.println("L: "+L.toString());      //输出:1 6 2 5 3 4
    }
}
```

2.3 第3章 栈和队列

1. 编写一个程序,求 n 个不同元素通过一个栈的出栈序列的个数,输出 n 为 1～7 的结果。

解:设 n 个不同的元素通过一个栈的出栈序列的个数为 $f(n)$。不妨设 n 个不同元素为 a、b、c、……,对于 $n\leqslant 3$ 很容易得出:

$f(1)=1$，即出栈序列为 a，共一种。

$f(2)=2$，即出栈序列为 ab、ba，共两种。

$f(3)=5$，即出栈序列为 abc、acb、bac、cba、bca，共 5 种。

再考虑 $f(4)$，4 个元素为 a、b、c、d。在出栈序列中元素 a 可能出现在 1 号位置、2 号位置、3 号位置和 4 号位置：

(1) 若元素 a 在 1 号位置，那么只可能 a 进栈马上出栈，此时还剩元素 b、c、d 等待操作，即 $f(3)$。

(2) 若元素 a 在 2 号位置，那么一定有一个元素比 a 先出栈，即有 $f(1)$种可能顺序(只能是 b)，还剩 c、d，即 $f(2)$。根据乘法原理，出栈序列个数为 $f(1)\times f(2)$。

(3) 若元素 a 在 3 号位置，那么一定有两个元素比 a 先出栈，即有 $f(2)$种可能顺序(只能是 b、c)，还剩 d，即为 $f(1)$。根据乘法原理，出栈序列个数为 $f(2)\times f(1)$。

(4) 若元素 a 在 4 号位置，那么一定是 a 先进栈，最后出栈，那么元素 b、c、d 的出栈顺序即是此小问题的解，即 $f(3)$。

结合所有情况，即 $f(4)=f(3)+f(2)\times f(1)+f(1)\times f(2)+f(3)$。

为了规整化，定义 $f(0)=1$，于是 $f(4)$可以重新写为

$$f(4)=f(0)\times f(3)+f(1)\times f(2)+f(2)\times f(1)+f(3)\times f(0)$$

按照该思路可以推广到 n，可以得到 $f(n)=f(0)\times f(n-1)+f(1)\times f(n-2)+\cdots+f(n-1)\times f(0)$，即

$$f(n)=\sum_{i=0}^{n-1} f(i)\times f(n-i-1)$$

设置一个数组 f，$f[i]$元素表示 $f(i)$，则

$$f[0]=1$$

$$f[1]=1$$

$$f(i)=\sum_{j=0}^{i-1} f(j)\times f(i-j-1) \quad i>1$$

$f[n]$就是 $f(n)$的结果。

实际上，可以通过数学计算出 $f(n)=\frac{1}{n+1}\times C_{2n}^{n}$。利用上述两种方式得到如下程序：

```
public class Exp1 {
    public static int comp1(int n) {                    //方式 1
        int[] f=new int[100];
        f[0]=f[1]=1;
        for(int i=2;i<=n;i++) {
            for(int j=0;j<=i-1;j++)
                f[i]+=f[j] * f[i-j-1];
        }
        return f[n];
    }
    public static int factor(int n) {                   //求 n!
        int ans=1;
        for(int i=2;i<=n;i++)
            ans * =i;
```

```
        return ans;
    }
    public static int comp2(int n) {                    //方式 2
        int ans=1;
        for(int i=2*n;i>=(n+2);i--)
            ans*=i;
        ans/=factor(n);
        return ans;
    }
    public static void main(String[] args) {
        System.out.println();
        for(int i=1;i<=7;i++) {
            System.out.println(" 算法 1 n="+i+" :" + comp1(i));
            System.out.println(" 算法 2 n="+i+" :" + comp2(i));
        }
    }
}
```

上述程序的执行结果如图 2.4 所示。

图 2.4　第 3 章实验题 1 的执行结果

2. 用一个一维数组 S(设固定容量 MaxSize 为 5,元素类型为 int)作为两个栈的共享空间。编写一个程序,采用《教程》中例 3.7 的共享栈方法设计其判栈空运算 empty(i)、判栈满运算 full()、进栈运算 push(i,x)和出栈运算 pop(i),其中 i 为 1 或 2,用于表示栈号,x 为进栈元素。采用相关数据进行测试。

解:直接利用《教程》中例 3.7 的原理设计程序如下:

```
class BSTACK {                                          //共享栈类
    final int MaxSize=5;
    int[] S;                                            //存放共享栈元素
    int top1,top2;                                      //两个栈顶指针
    public BSTACK() {                                   //构造方法,栈初始化
        S=new int[MaxSize];
        top1=-1;
        top2=MaxSize;
    }
    //-----判栈空算法-------
    public boolean empty(int i) {                       //i=1:栈 1,i=2:栈 2
        if(i==1)
            return(top1==-1);
        else if(i==2)                                   //i=2
            return(top2==MaxSize);
```

```
        else
            throw new IllegalArgumentException("i 错误");
    }
    //－－－－－判栈满算法－－－－－－－
    public boolean full() {                                    //判断栈满
        return top1==top2－1;
    }
    //－－－－－进栈算法－－－－－－－
    public void push(int i,int x) {                            //i=1:栈 1,i=2:栈 2
        if(full())                                             //栈满
            throw new IllegalArgumentException("栈满");
        if(i==1) {                                             //x 进栈 S1
            top1＋＋;
            S[top1]=x;
        }
        else if(i==2) {                                        //x 进栈 S2
            top2－－;
            S[top2]=x;
        }
        else throw new IllegalArgumentException("i 错误");
    }
    //－－－－－出栈算法－－－－－－－
    public int pop(int i) {                                    //i=1:栈 1,i=2:栈 2
        int x;
        if(i==1) {                                             //S1 出栈
            if(empty(1))                                       //S1 栈空
                throw new IllegalArgumentException("栈 1 空");
            else {                                             //出栈 S1 的元素
                x=S[top1];
                top1－－;
            }
        }
        else if(i==2) {                                        //S2 出栈
            if(empty(2))                                       //S2 栈空
                throw new IllegalArgumentException("栈 2 空");
            else {                                             //出栈 S2 的元素
                x=S[top2];
                top2＋＋;
            }
        }
        else throw new IllegalArgumentException("i 错误");
        return x;                                              //操作成功返回 x
    }
}
public class Exp2 {
    public static void main(String[] args) {
        BSTACK st=new BSTACK();
        System.out.println("");
        System.out.println("  (1)新建立栈 st");
        System.out.println("     栈 1 空？"＋st.empty(1));
        System.out.println("     栈 2 空？"＋st.empty(2));
        int[] a={1,2,3};
        int[] b={4,5,6,7};
        System.out.println("  (2)栈 1 的进栈操作");
```

```
        for(int i=0;i<a.length;i++) {
            if(!st.full()) {
                System.out.println("     "+a[i]+"进栈 1");
                st.push(1,a[i]);
            }
            else System.out.println("     "+a[i]+"进栈:栈满不能进栈");
        }
        System.out.println("     栈 1 空? "+st.empty(1));
        System.out.println("  (3)栈 2 的进栈操作");
        for(int i=0;i<b.length;i++) {
            if(!st.full()) {
                System.out.println("    "+b[i]+"进栈 2");
                st.push(2,b[i]);
            }
            else System.out.println("    "+b[i]+"进栈:栈满不能进栈");
        }
        System.out.println("  栈 2 空? "+st.empty(2));
        System.out.println("  (4)栈 1 的出栈操作");
        while(!st.empty(1))
            System.out.println("     出栈 1 元素"+st.pop(1));
        System.out.println("  (5)栈 2 的出栈操作");
        while(!st.empty(2))
            System.out.println("     出栈 2 元素"+st.pop(2));
    }
}
```

上述程序的执行结果如图 2.5 所示。

3. 改进《教程》中 3.1.7 节的用栈求解迷宫问题的算法，累计图 2.6 所示的迷宫的路径条数，并输出所有迷宫路径。

```
(1)新建立栈st
  栈1空? true
  栈2空? true
(2)栈1的进栈操作
  1进栈1
  2进栈1
  3进栈1
  栈1空? false
(3)栈2的进栈操作
  4进栈2
  5进栈2
  6进栈:栈满不能进栈
  7进栈:栈满不能进栈
  栈2空? false
(4)栈1的出栈操作
  出栈1元素3
  出栈1元素2
  出栈1元素1
(5)栈2的出栈操作
  出栈2元素5
  出栈2元素4
```

图 2.5　第 3 章实验题 2 的执行结果

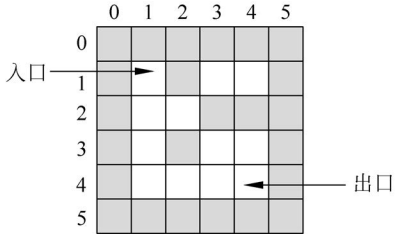

图 2.6　迷宫示意图

解：修改《教程》中 3.1.7 节用栈求解迷宫问题的 mgpath()算法，当找到一条路径后输出栈 st 中所有方块构成一条迷宫路径(通过临时栈 tmpst 恢复栈 st，保证输出迷宫路径后栈 st 元素不变)，此时并不立即返回，而是出栈 st 的栈顶方块并恢复其 mg 元素值，再从新栈顶方块继续搜索到达出口的其他迷宫路径，直到栈 st 变空为止，这样就输出了所有的迷宫路径。

对应的算法和程序如下：

```
import java.util.*;
class Box {                                          //方块结构体类型
    int i;                                           //方块的行号
    int j;                                           //方块的列号
    int di;                                          //di是下一个可走相邻方位的方位号
    public Box(int i1,int j1,int di1) {              //构造方法
        i=i1; j=j1; di=di1;
    }
}
class MazeClass {                                    //用栈求解所有迷宫路径类
    final int MaxSize=20;
    int[][] mg;                                      //迷宫数组
    int m,n;                                         //迷宫行/列数
    int cnt=0;                                       //迷宫路径条数
    public MazeClass(int m1,int n1) {                //构造方法
        m=m1;
        n=n1;
        mg=new int[MaxSize][MaxSize];
    }
    public void Setmg(int[][] a) {                   //设置迷宫数组
        for(int i=0;i<m;i++) {
            for(int j=0;j<n;j++)
                mg[i][j]=a[i][j];
        }
    }
    public int mgpath(int xi,int yi,int xe,int ye) {  //求一条从(xi,yi)到(xe,ye)的迷宫路径
        int i,j,di,i1=0,j1=0;
        boolean find;
        Box box,e;
        Stack<Box> st=new Stack<Box>();              //建立一个空栈
        st.push(new Box(xi,yi,-1));                  //入口方块进栈
        mg[xi][yi]=-1;                               //进栈方块的mg置为-1
        while(!st.empty()) {                         //栈不空时循环
            box=st.peek();                           //取栈顶方块,称为当前方块
            i=box.i; j=box.j; di=box.di;
            if(i==xe && j==ye) {                     //找到了出口,输出一条迷宫路径
                Stack<Box> tmpst=new Stack<Box>();   //建立临时栈
                cnt++;                               //路径条数增1
                System.out.print("  迷宫路径"+cnt+": ");     //输出一条迷宫路径
                while(!st.empty()) {
                    e=st.pop();
                    System.out.print("["+e.i+","+e.j+"]  ");
                    tmpst.push(e);                   //出栈元素进tmpst栈
                }
                while(!tmpst.empty())                //tmpst栈中的元素出栈并进st栈
                    st.push(tmpst.pop());            //恢复st栈
                System.out.println();
                box=st.pop();                        //退栈st
                mg[box.i][box.j]=0;                  //让该位置变为其他路径可走方块
                i=box.i; j=box.j; di=box.di;
            }
            find=false;                              //否则继续找路径
            while(di<4 && !find) {                   //找下一个相邻可走方块
```