第5章 栈和队列

5.1 知识要点

1. 栈是一种运算受限的线性表，它只允许在栈顶进行插入和删除等运算，各种运算的时间复杂度均为 $O(1)$ 。

2. 栈也适合采用顺序和链接这两种存储结构。通过栈顶指针能够访问到栈顶元素和进行插入、删除等运算。

3. 计算机执行递归算法时需要自动建立和使用一个数据栈，用来存储每次调用后的返回地址、形参和局部变量的值，每结束一次调用都要按栈中保存的返回地址返回到调用位置向下执行，并自动做一次退栈处理操作。

4. 队列是一种只允许在一端插入、而在另一端删除元素的线性表，进行插入的一端叫做队尾，而进行删除的一端叫做队首，插入和删除操作的时间复杂度均为 $O(1)$ 。

5. 队列也适合采用顺序和链接这两种存储结构。在顺序队列中，存储空间是首尾相接的一个环，故称为循环队列。若用于存储队列的数组空间大小为 n ，则在不设定队列长度变量的情况下，其所存队列的最大长度为 $n-1$ 。

6. 在顺序队列中，若队尾指针落后于队首指针一个元素位置，则表示队满；若队首指针与队尾指针相同，则表示队空。

7. 栈和队列是线性数据结构中运算最简单的数据结构，但又是最有用的数据结构，它普遍应用于各种算法的实现之中。特别是递归算法的执行过程是依赖栈这种数据结构来实现的。

8. 递归是分析问题和解决问题的一种非常重要的方法和手段，很多复杂的问题都可以通过递归迎刃而解。递归过程是采用解决自身的同样方法来逐渐解决规模较小的问题，最终递归到能够直接得到最小问题的处理结果，而后逐渐返回依次向上结束每次递归，最终结束所有递归而返回到非递归调用的位置继续向下执行。

9. 表达式的计算是栈的一种典型应用，中缀转后缀时需要使用运算符栈，后缀求值时需要使用操作数栈。

10. 栈的接口 Stack 的定义：

```
public interface Stack
{
    //栈的抽象接口的定义
    void push(Object obj);           //向栈顶插入一个新元素 obj
    Object pop();                   //从栈中删除栈顶元素并返回
    Object peek();                  //返回栈顶元素的值
    boolean isEmpty();              //判断栈是否为空，若是则返回 true，否则返回 false
    void clear();                   //清除栈中的所有元素使之成为一个空栈
}
```

11. 队列的接口 Queue 的定义:

```
public interface Queue
{
    //队列的抽象接口的定义
    void enter(Object obj);           //元素进队, 即向队列尾部添加一个新元素 obj
    Object leave();                   //元素出队, 即从队列首部删除队首元素并返回
    Object peek();                     //返回队首元素的值
    boolean isEmpty();                 //判断队列是否为空
    void clear();                       //清除队列中的所有元素使之变成一个空队
}

```

5.2 练习题

5.2.1 单选题

1. 当利用大小为 N 的数组顺序存储一个栈时, 假定用 $\text{top}==N$ 表示栈空, 则向这个栈插入一个元素时, 首先应执行 () 语句修改 top 指针。

A. $\text{top}++$; B. $\text{top}--$; C. $\text{top}=0$; D. $\text{top}=N$;

2. 假定利用数组 a 顺序存储一个栈, 用 top 表示栈顶指针, $\text{top}=-1$ 表示栈空, 并已知栈未满, 当元素 x 进栈时所执行的操作为 ()。

A. $a[--\text{top}]=x$; B. $a[\text{top}--]=x$; C. $a[++\text{top}]=x$; D. $a[\text{top}++]=x$;

3. 假定利用数组 a 顺序存储一个栈, 用 top 表示栈顶指针, $\text{top}=-1$ 表示栈空, 并已知栈不为空, 当退栈并返回栈顶元素时所执行的操作为 ()。

A. $\text{return } a[--\text{top}]$; B. $\text{return } a[\text{top}--]$;

C. $\text{return } a[++\text{top}]$; D. $\text{return } a[\text{top}++]$;

4. 假定一个链栈的栈顶指针用 top 表示, 当 p 所指向的结点进栈时, 执行的操作为 ()。

A. $p.\text{next}=\text{top}; \text{top}=\text{top}.\text{next}$; B. $\text{top}=p; p.\text{next}=\text{top}$;

C. $p.\text{next}=\text{top}.\text{next}; \text{top}.\text{next}=p$; D. $p.\text{next}=\text{top}; \text{top}=p$;

5. 假定一个链栈的栈顶指针用 top 表示, 当进行退栈时所进行的操作为 ()。

A. $\text{top}.\text{next}=\text{top}$;

B. $\text{top}=\text{top}.\text{element}$;

C. $\text{top}=\text{top}.\text{next}$;

D. $\text{top}.\text{next}=\text{top}.\text{next}.\text{next}$;

6. 若让元素 1、2、3 依次进栈, 则出栈次序不可能出现 () 种情况。

A. 3、2、1

B. 2、1、3

C. 3、1、2

D. 1、3、2

7. 在一个顺序队列中, 队首指针指向队首元素的 () 位置。

A. 前一个

B. 后一个

C. 当前

D. 前两个

8. 从一个顺序队列删除元素时, 首先需要使 ()。

A. 队首指针循环加 1

B. 队首指针循环减 1

C. 队尾指针循环加 1

D. 队尾指针循环减 1

9. 假定一个顺序队列的队首和队尾指针分别为 f 和 r , 则判断队空的条件为 ()。
- A. $f+1==r$ B. $r+1==f$ C. $f==0$ D. $f==r$
10. 假定利用长度为 N 的数组 a 顺序存储一个队列, 用 f 和 r 分别表示队首和队尾指针, 并已知队列未空, 当元素 x 进队时所执行的操作为 ()。
- A. $a[++r\%N]=x$; B. $a[r++\%N]=x$; C. $a[--r\%N]=x$; D. $a[r--\%N]=x$;
11. 假定利用长度为 N 的数组 a 顺序存储一个队列, 用 f 和 r 分别表示队首和队尾指针, 并已知队列未空, 当进行出队运算时所执行的操作为 ()。
- A. `return a[++r%N];` B. `return a[--r%N];`
 C. `return a[+f%N];` D. `return a[f++%N];`
12. 栈又称为 ()。
- A. 后进先出表 B. 先进先出表 C. 有序表 D. 一般线性表
13. 设元素 a 、 b 、 c 、 d 依次进栈, 若要求得到的出栈次序为 c 、 b 、 d 、 a , 则共需要进行的入和出栈操作的总次数为 ()。
- A. 7 B. 8 C. 6 D. 9
14. 假定采用 r 进制数的 n 位进行数据编码, 则总编码量 (个数) 为 ()。
- A. $r \times n$ B. r^n C. r^n D. n^r

5.2.2 运算题

1. 有 3 个元素 a 、 b 、 c 依次进栈, 任何时候都可以出栈, 请写出所有可能的出栈序列和所有不存在的出栈序列。
2. 在进行后缀表达式计算时, 需要使用一个具有 `Double` 元素类型的操作数栈 `sck`, 假定需要计算的一个后缀表达式字符串为 “3 20 5 3*- 2+ 4/ 6*+”, 试采用线性表的书写方法写出在计算过程中栈 `sck` 的每次变化的情况。例如, 前 5 次的变化情况为 (3); (3,20); (3,20,5); (3,20,5,3); (3,20,15)。
3. 在把中缀表达式转换为后缀表达式的过程中, 需要使用一个暂存运算符的栈 `sck`, 假定一个中缀表达式为 “3+(20-5*3+2)/4*6@”, 试采用线性表的书写方法写出在计算过程中栈 `sck` 的每次变化的情况。例如, 前 4 次的变化情况为 (@); (@,+); (@,+,(); (@,+,(,-)。
4. 假定用于顺序存储一个队列的数组的长度为 N , 队首和队尾指针分别为 `front` 和 `rear`, 写出求此队列长度 (即所含元素个数) 的公式。

5.2.3 算法分析题

分析下列每个算法并写出对应的功能。

1.

```
public static int fun1(int a[],int n)
{
    //外部调用时使用的是一个整型数组参数和它的长度值
    if(n==1) return a[0];
```

```
    else return a[n-1]+fun1(a,n-1);  
}
```

2.

```
public static int fun2(int k, int s)  
{    //外部调用时使用的参数是 0 和 0  
    if(s>=100) return k-1;  
    else {  
        k++;  
        s+=k*k;  
        return fun2(k,s);  
    }  
}
```

3.

```
public static void fun3(int num, int r)  
{    //外部调用时对应的 r 参数是 2-9 之间的一个整数  
    if(num/r!=0) fun3(num/r,r);  
    System.out.print(num%r);  
}
```

4.

```
public static void fun4(int num)  
{  
    Stack a=new SequenceStack();  
    while(num!=0) {  
        int k=num % 16;  
        a.push(k);  
        num/=16;  
    }  
    while(!a.isEmpty())  
    {  
        int x=(Integer)a.pop();  
        if(x<10) System.out.print(x);  
        else {  
            switch (x)  
            {  
                case 10: System.out.print('A'); break;  
                case 11: System.out.print('B'); break;  
                case 12: System.out.print('C'); break;  
                case 13: System.out.print('D'); break;  
                case 14: System.out.print('E'); break;  
            }  
        }  
    }  
}
```

```

        case 15: System.out.print('F');
    }
}
System.out.println();
}

```

5.2.4 算法设计题

1. 根据所给出的方法声明设计一个递归算法，返回 1 至 n 之间的所有整数平方的和。

```
public static int f1(int n);
```

2. 采用辗转相除和递归的方法求出两个正整数 a 和 b 的最大公约数。

```
public static int f2(int a, int b);
```

3. 编写一个递归函数，把一个元素类型为 `Integer` 的栈中的所有元素，按照从栈底到栈顶的次序输出。

```
public static void f3(Stack sck);
```

4. 斐波那契 (Fibonacci) 数列的定义为：它的第 1 项和第 2 项分别为 0 和 1，以后各项为其前两项之和。若斐波那契数列中的第 n 项用 $\text{Fib}(n)$ 表示，则计算公式为：

$$\text{Fib}(n) = \begin{cases} n-1 & (n=1 \text{ 或 } n=2) \\ \text{Fib}(n-1) + \text{Fib}(n-2) & (n>2) \end{cases}$$

试编写出计算 $\text{Fib}(n)$ 的递归算法和非递归算法。

5. 采用递归方法求两个正整数 a 和 b 的最小公倍数， k 的初值应为 2。

```
public static int f5(int a, int b, int k);
```

6. 根据代数中的二项式定理，二项式 $(x+y)^n$ 的展开式的系数序列可以表示成图 5-1 那样的三角形，其中除每一行最左和最右两个系数等于 1 以外，其余各系数均等于上一行左右两系数之和。这个系数三角形称做杨辉三角形。

$(x+y)^0$										1
$(x+y)^1$									1	1
$(x+y)^2$								1	2	1
$(x+y)^3$							1	3	3	1
$(x+y)^4$						1	4	6	4	1
$(x+y)^5$					1	5	10	10	5	1
$(x+y)^6$				1	6	15	20	15	6	1
$(x+y)^7$			1	7	21	35	35	21	7	1
$(x+y)^8$	1	8	28	56	70	56	28	8	1	

图 5-1 杨辉三角形

设 $\text{binomial}(n,k)$ 表示杨辉三角形中第 n 行 ($n \geq 0$) 的第 k 个系数 ($0 \leq k \leq n$), 按照二项式定理, $\text{binomial}(n,k)$ 可递归定义为:

$$\text{binomial}(n,k) = \begin{cases} 1 & (k=0 \text{ 或 } k=n) \\ \text{binomial}(n-1, k-1) + \text{binomial}(n-1, k) & (0 < k < n) \end{cases}$$

- (1) 写出计算 $\text{binomial}(n, k)$ 的递归算法。
- (2) 利用二维数组写出计算 $\text{binomial}(n, k)$ 的非递归算法。
- (3) 分析递归算法和非递归算法的时间复杂度和空间复杂度。

7. 编写一个递归算法, 输出自然数 $1 \sim n$ 这 n 个元素的全排列, $1 \sim n$ 被依次保存在整型数组 $a[0] \sim a[n-1]$ 中, s 的初值应为 0。

```
public static void permute(int[] a, int s, int n);
```

8. 利用堆栈编写出求解迷宫问题的非递归算法, 其中, `sck` 栈用来保存可通行路径上的坐标, `maze` 用来保存迷宫数据, `mark` 用来保存访问标记, `move` 用来保存位移量数据。栈 `sck` 中的实际元素类型为 `Place` 类, 该类的定义为:

```
class Place {           //定义描述迷宫中当前位置的类
    int x, y, d;        //x 和 y 分别表示当前位置的行坐标和列坐标,
                        //d 表示移动到下一步的方向
    Place(int xx, int yy, int dd) {x=xx; y=yy; d=dd;}
}
```

求解迷宫问题的非递归算法的声明如下, 要求查找出 `maze` 迷宫中从入口(1,1)到出口的一条通路, 并通过 `sck` 栈带回。

```
public static boolean seekPathN(Stack sck, int [][]maze,
                                int [][]mark, int [][]move)
```

9. 编写出解决汉诺塔问题的非递归算法。

```
public static void HanoiTowerN(int n, char a, char b, char c)
```

10. 判断任意 n 个字符串能否首尾相接成为一个字符串。若一个字符串的尾字符等于另外一个字符串的首字符, 则认为这两个字符串能够首尾相接形成一个字符串。

```
public static boolean frontRear(String[] a, boolean[] mark,
                                int i, int k)
```

5.3 练习题参考解答

5.3.1 单选题

1. B 2. C 3. B 4. D 5. C
6. C 7. A 8. A 9. D 10. A
11. C 12. A 13. B 14. C

5.3.2 运算题

1. 所有可能的出栈序列: a,b,c; a,c,b; b,a,c; b,c,a; c,b,a;

所有不存在的出栈序列: c,a,b;

2. (3,5); (3,5,2); (3,7); (3,7,4); (3,1.75); (3,1.75,6); (3,10.5); (13.5)。

3. (@,+,(-,*); (@,+,(+); (@,+); (@,+,/); (@,+,*); ()。

4. 队列长度的计算公式为 $(\text{rear}-\text{front}+N)\%N$

解析: 当 rear 大于等于 front 时, 队列长度为 rear-front, 也可以表示为 $(\text{rear}-\text{front}+N)\%N$; 当 rear 小于 front 时, 队列被分成两个部分, 前部分在数组尾部, 其元素个数为 $N-1-\text{front}$, 后部分在数组首部, 其元素个数为 rear+1, 两者相加为 rear-front+N。所以, 在任何情况下, 队列长度的计算公式为 $(\text{rear}-\text{front}+N)\%N$ 。

5.3.3 算法分析题

1. 求出数组 a 中 n 个元素之和。

2. 计算并返回满足不等式 $1^2+2^2+3^2+\dots+n^2<100$ 的最大 n 值。

3. 根据十进制正整数 num 转换成 r 进制数输出。

4. 把一个十进制正整数 num 转换为一个十六进制数输出。

5.3.4 算法设计题

1.

```
public static int f1(int n)
{
    if(n==0) return 0;
    else return n*n+f1(n-1);
}
```

2.

```
public static int f2(int a, int b)
{
    if(a%b==0) return b;
    else return f2(b,a%b);
}
```

3.

```
public static void f3(Stack sck)
{
    if(sck.isEmpty()) return;
    else {
        int x=(Integer)sck.pop();
        f3(sck);
    }
}
```

```
        System.out.print(x+" ");
    }
}

4.
public static int Fib(int n)
{
    if(n<=0) {System.out.println("参数 n 值错误,返回 0!"); return 0;}
    if(n==1 || n==2) return n-1;
    else return Fib(n-1)+Fib(n-2);
}
```

```
public static int Fib1(int n)
{
    if(n<=0) {System.out.println("参数 n 值错误,返回 0!"); return 0;}
    int a=0,b=1,c=0; //c 代表当前项,
                    //a 和 b 分别代表当前项前面的第 2 项和第 1 项
    if(n==1 || n==2) return n-1;
    for(int i=3; i<=n; i++) {
        c=a+b;          //求出当前项
        a=b;           //把前面第 1 项赋给前面第 2 项
        b=c;           //把当前项赋给前面第 1 项
    }
    return c;          //返回所求的第 n 项的值
}
```

```
5.
public static int f5(int a, int b, int k)
{
    while(a>=k && b>=k) {
        if(a%k==0 && b%k==0)
            return k*f5(a/k, b/k, k);
        else k++;
    }
    return a*b;
}
```

6.

(1)

```
public static int binomial(int n, int k)
    //求出指数为 n 的二项展开式中第 k 项 (0≤k≤n) 系数的递归算法
{
    if(k==0 || k==n)
        return 1;
    else
```

```

        return binomial(n-1,k-1)+binomial(n-1,k);
    }

(2)
public static int binomialN(int n, int k)
    //求出指数为 n 的二项展开式中第 k 项 (0<=k<=n) 系数的非递归算法
{
    int i,j;
    //定义并创建具有 (n+1)*(n+1) 个元素的二维整型数组 a
    int[][] a=new int[n+1][n+1];
    //外循环每循环一次计算出指数为 i 的二项展开式的所有系数,
    //内循环每循环一次计算出此展开式中第 j 项的系数
    for(i=0; i<=n; i++)
        for(j=0; j<=i; j++)
            if(j==0 || j==i) a[i][j]=1;
            else a[i][j]=a[i-1][j-1]+a[i-1][j];
    //返回指数为 n 的二项展开式中第 k 项的系数
    return a[n][k];
}

```

(3) 递归算法的每次递归调用都引起两次向下的递归调用, 并且每次都使 n 值减 1 和 k 值减 1, 当 k 等于 0 或 k 等于 n 时结束递归, 所以此递归算法的时间复杂度和空间复杂度分别为 $O(2^n)$ 和 $O(n)$ 。非递归算法的时间复杂度和空间复杂度均为 $O(n^2)$ 。

7. 分析: 由排列组合的知识可知, n 个元素的全排列共有 $n!$ 种。例如, 对于 1、2、3 这 3 个元素, 其全排列为 123、132、213、231、321、312, 共 $3!=6$ 种。 $n!$ 种可分解为 $n*(n-1)!$ 种, 而 $(n-1)!$ 种又可分解为 $(n-1)*(n-2)!$ 种, 依次类推。对于 n 个元素, 可把它们分别放入到 n 个位置上, 让第 1 个位置依次取每一个元素, 共有 n 种不同的取法, 对其后 $n-1$ 个位置上的 $n-1$ 个元素, 共有 $(n-1)!$ 种不同的排列, 所以总共有 $n*(n-1)!$ 种不同的排列; 同样, 对于从第 2 个位置开始的所有元素, 让第 2 个位置依次取除第 1 个位置上的元素之外的剩余 $n-1$ 个元素, 共有 $n-1$ 种不同的取法, 对其后 $n-2$ 个位置上的 $n-2$ 个元素, 共有 $(n-2)!$ 种不同的排列, 依次类推; 当进行到第 n 个位置时, 只有一种取法, 因为前 $n-1$ 个位置已经固定了 $n-1$ 个元素, 剩余的一个元素被放在这个位置上。

若用一个数组 $a[n]$ 来保存 $1\sim n$ 之间的 n 个自然数, 对于 $i=0$ 到 $n-1$, 每次使 $a[0]$ 同 $a[i]$ 交换 ($i=0,1,2,\dots,n-1$) 后, 对 $a[1]\sim a[n-1]$ 中的 $n-1$ 个元素进行全排列, 然后再交换 $a[0]$ 与 $a[i]$ 的值, 使它恢复为此次排列前的状态; 同样, 对于 $a[1]\sim a[n-1]$ 区间内的 $n-1$ 个元素, 每次使 $a[1]$ 同 $a[i]$ 交换 ($i=1,2,\dots,n-1$) 后, 对 $a[2]\sim a[n-1]$ 区间内的 $n-2$ 个元素进行全排列, 然后再把交换的元素交换回来; 依次类推, 直到对 $a[n-1]$ 进行全排列时, 输出整个数组的值, 即得到一种排列结果。

对 n 个元素的全排列是一个递归过程, 具体描述如下:

```

public static void permute(int a[], int s, int n)
    //对 a[0]~a[n-1] 中依次保存的 1~n 数字进行全排列, s 的初值应为 0
{
    int i, temp;

```

```

//当递归排序到最后一个元素时结束递归, 输出 a 中保存的一种排列
if (s==n-1) {
    for(i=0; i<n ;i++) System.out.print(a[i]+" ");
    System.out.println();
}
//其他情况需要递归排列
else
    for (i=s; i<n; i++) { //循环 n-s 次, 每次使 a[s]取一个新值
        //交换 a[s]与 a[i]的元素值
        temp=a[s]; a[s]=a[i]; a[i]=temp;
        //对 a[s+1]~a[n-1]中的元素进行递归排序
        permute(a, s+1, n);
        //恢复 a[s]与 a[i]的原有值
        temp=a[s]; a[s]=a[i]; a[i]=temp;
    }
}

```

此算法的时间复杂度为 $O(n!)$, 因为共需要进行 $n!$ 次递归调用; 空间复杂度为 $O(n)$, 因为系统栈的最大深度为 n 。

8. 非递归算法包含在下面类中。

```

public class Lianxi5_2
{
    public static boolean seekPathN(Stack sck, int [][]maze, int [][]mark,
        int [][]move)
        //查找 maze 迷宫中从入口 (1,1) 到出口的一条通路
    {
        //将入口点访问标记置为 1, 表示将访问它
        mark[1][1]=1;
        //定义 temp 为进出栈的变量
        Place temp;
        //将入口点位置及方向初值-1 压入栈
        temp=new Place(1,1,-1);
        sck.push(temp);
        //循环处理, 直到栈空为止
        while(!sck.isEmpty())
        {
            //退栈, 即从查找的路径中退回一步
            temp=(Place) (sck.pop());
            //沿着原位置的下一个方向前进
            int x=temp.x; int y=temp.y; int d=temp.d+1;
            while(d<4)
            {
                //到达出口, 将出口点坐标压入栈后返回
                if(x==maze.length-2 && y==maze[0].length-2) {

```

```

        temp=new Place(maze.length-2, maze[0].length-2, 0);
        sck.push(temp);
        return true;
    }
    //按方向 d 前进, 求出下一个位置的坐标
    int g=x+move[d][0]; int h=y+move[d][1];
    //对未访问过的可通行的下一个位置, 应将当前位置进栈,
    //并将下一个位置变为当前位置, 否则沿下一个方向寻找出口
    if(maze[g][h]==0 && mark[g][h]==0)
    {
        mark[g][h]=1;
        temp=new Place(x,y,d);
        sck.push(temp);
        x=g; y=h; d=0;    //进入一个新位置后, 重新从初始方向东开始
    }
    else
        d++;    //沿下一个方向前进
    }
}
//不存在从入口到出口的通路返回假
return false;
}

public static void main(String[] args)
{
    //定义并创建一个顺序栈 sck, 用来记录路径上的坐标点
    Stack sck=new SequenceStack();
    //定义和创建迷宫矩阵
    int [][]maze={{1,1,1,1,1,1,1,1,1,1}, {1,0,0,1,1,0,1,0,1,1},
        {1,1,0,0,1,1,0,0,0,1}, {1,0,0,0,0,0,0,1,1,1},
        {1,1,1,0,1,1,0,0,0,1}, {1,0,0,0,0,0,1,0,1,1},
        {1,1,0,1,0,0,0,0,0,1}, {1,1,1,1,1,1,1,1,1,1}};
    //定义并创建一个访问迷宫的标记数组 mark, 每个元素被初始化为 0
    int [][]mark=new int[8][10];
    for(int i=0; i<8; i++) for(int j=0; j<10; j++) mark[i][j]=0;
    //定义并创建一个二维整型数组 move, 用 4 个方向的位移量坐标进行初始化
    int [][]move={{0,1},{1,0},{0,-1},{-1,0}};
    //调用求解迷宫的非递归算法
    boolean b=seekPathN(sck,maze,mark,move);
    if(b) {    //输出在 sck 栈中保存的可通行路径
        Stack s=new SequenceStack();    //创建一个辅助栈 s
        while(!sck.isEmpty()) s.push(sck.pop());    //用 s 逆序保存 sck
        System.out.println("从迷宫入口到出口的一条搜索路径为:");
        while(!s.isEmpty()) {    //按从入口到出口次序输出坐标
            Place c=(Place)(s.pop());

```

```

        System.out.println("(" + c.x + ", " + c.y + ")");
    }
}
else System.out.println("此迷宫不存在从入口到出口的任何通路!");
}
}

```

程序调试和运行结果:

```

D:\tsinghua>javac Place.java
D:\tsinghua>javac Lianxi5_2.java
D:\tsinghua>java Lianxi5_2

```

从迷宫入口到出口的一条搜索路径为:

```

(1,1)
(1,2)
(2,2)
(2,3)
(3,3)
(3,4)
(3,5)
(3,6)
(4,6)
(4,7)
(5,7)
(6,7)
(6,8)

```

9.

```

public static void HanoiTowerN(int n, char a, char b, char c)
{
    //定义栈中的元素类型, 它含有保存参数表中每个参数的域
    class Items {
        int n;
        char a,b,c;
        Items(int nn, char aa, char bb, char cc) {
            n=nn; a=aa; b=bb; c=cc;
        }
    }
    //定义顺序存储的栈 sck
    Stack sck=new SequenceStack();
    //定义临时变量 temp, 作为出入栈使用
    Items temp;
    //算法主体, 进行汉诺塔运算, 当栈非空或 n 大于等于 1 时循环
    while(!sck.isEmpty() || n>=1)
    {
        while(true) {

```

```

//若 n 等于 1 则直接由 a 柱搬到 c 柱, 同时置 n=0 并退出此层循环
    if (n==1) {System.out.println(a+"→"+c); n=0; break;}
//按照参数值生成一个元素进栈保存
    temp=new Items(n,a,b,c);
    sck.push(temp);
//按原递归调用 HanoiTower(n-1,a,c,b) 的要求给每个参数重新赋值
    n=n-1; char d=b; b=c; c=d;          //n 值减 1, a 值不变, b 和 c 交换值
}
if(!sck.isEmpty()) {
//执行原递归调用 HanoiTower(n-1,a,c,b) 返回后应做退栈处理,
//分别把栈顶元素的每个域的值赋给对应的参数
    temp=(Items)(sck.pop());
    n=temp.n; a=temp.a; b=temp.b; c=temp.c;
//执行原递归调用 HanoiTower(n-1,a,c,b) 后的输出语句
    System.out.println(a+"→"+c);
//按原末尾递归调用 HanoiTower(n-1,b,a,c) 的要求给每个参数重新赋值
    n=n-1; char d=a; a=b; b=d;          //n 值减 1, c 值不变, a 和 b 交换值
}
}
}
}

```

10. 分析: 此题同求解迷宫问题的思路类似, 也是一个递归求解的过程。开始时要用一个字符串同另外 $n-1$ 个其中的一个字符串首尾相连, 再接着同剩余 $n-2$ 个其中的一个字符串首尾相连, 依次类推。当无法由 k 个已连接成的字符串与剩余 $n-k$ 个中的任一个字符串连接时, 则需要退回一步, 由 $k-1$ 个已连接成的字符串与剩余的未试探过的任一个字符串连接, 若仍不能找到连接的字符串, 还需要后退, 由 $k-2$ 个已连接成的字符串与剩余的未试探过的任一个字符串连接, 依次类推。当连接了 n 个字符串之后, 表明连接成功, 应返回真; 若无法连接下去, 应返回假。

该算法需要一个字符数组 a , 用来保存 n 个字符串; 还需要一个整型数组 $mark$, 用它的每个元素标记 a 数组中的对应字符串是否已被连接, 若已被连接则其值为真, 否则为假, 这样能够避免一个字符串重复被试探和连接; 还需要一个参数 i , 用它来表示最近被连接上的数组 a 中的一个字符串的下标; 还需要设置一个参数 k , 用来表示已连接成的字符串的个数, 初值应为 1, 每首尾连接一个新字符串之后 k 增 1, 当 k 等于 n 时表示连接成功应返回真。这里的 n 就是字符串数组 a 的长度, 即所包含的字符串的个数, 此算法需要判断它们是否能够首尾连接成一个字符串。

当下标为 i 的字符串同下标为 j 的字符串连接时, 只有当 i 不等于 j , 即不是同一个字符串, $mark[j]$ 的值为假, 即没有被连接, 并且 $a[i].charAt(a[i].length()-1)$ 等于 $a[j].charAt(0)$, 即下标为 i 的字符串的尾字符与下标为 j 的字符串的首字符相同时, 这两个字符串才能够连接, 否则不能被连接。下标为 i 的字符串与下标为 j 的字符串连接成功后, 应置 $mark[j]$ 的值为真, 应接着进行递归调用, 把 j 的值传送给值参 i , 表明下标为 j 的字符串被重新连接上, 把 $k+1$ 的值传送给值参 k , 进行第 $k+1$ 个字符串的连接。当此次递归调用成功返回真, 应输出 j 的值并返回真, 接着依次向上返回输出 j 的值, 依次类推, 这样将使输出的字符串的下标值与字符串连接的次序正好相反。当此次递归调用不成功返回假时, 则表明

从下标为 j 的字符串向下无法继续连接到最后一个字符串, 即此路走不通, 应将 $\text{mark}[j]$ 重置为假, 以便使第 k 个字符串 $a[i]$ 与其他未连接的字符串相连接。

根据以上分析, 编写出此题的算法如下:

```
public class Lianxi5_4
{
    public static boolean frontRear(String[] a, boolean[] mark, int i, int k)
    {
        if(k==a.length) return true;
        for(int j=0; j<a.length; j++)
        { //试探第 k 个字符串 a[i] 与第 k+1 个字符串 a[j] 的连接
            if(i!=j && mark[j]==false && a[i].charAt(a[i].length()-1)==a[j].
                charAt(0)) {
                mark[j]=true;
                if(frontRear(a,mark,j,k+1)) {
                    System.out.print(" "+j);
                    return true;
                }
                else mark[j]=false;
            }
        }
        return false; //第 k 个字符串与第 k+1 个字符串连接不成功返回假
    }

    public static void main(String[] args)
    {
        //定义和创建一个字符串数组 a
        String[] a={"ab", "ba", "cderfg", "admnc", "bzxca", "asdfcvb"};
        //定义和创建一个逻辑数组 mark, 用来标记对应数组 a 中的字符串是否被连接
        boolean[] mark=new boolean[a.length];
        //依次以数组 a 中的每个字符串作为第 1 个字符串进行试探连接
        for(int i=0; i<a.length; i++) {
            //初始化 mark 数组中的每个元素的初始值为 false, 表示未连接
            for(int j=0; j<mark.length; j++) mark[j]=false;
            //要试探连接的第 1 个字符串为 a[i], 将其对应的标记置为 true
            mark[i]=true;
            //若以 a[i] 为第 1 个字符串连接成功则输出 i 值和连接成功信息
            if(frontRear(a,mark,i,1)) {
                System.out.println(" "+i);
                System.out.println("以"+a[i]+"为首字符串连接成功!");
            }
        } //循环结束后能够得到所有可能的连接
    }
}
```