第 3 章习题

3.1 对一个栈的输入序列 $a_1, a_2, a_3, \dots, a_n,$，称由此栈依次出栈后所得到的元素序列为栈的合法输出序列。例如，假设栈 S 的一个输入序列为 1, 2, 3, 4, 5，则可得到多个输出序列，例如，1, 2, 3, 4, 5 就是一个合法的输出序列，同理，5, 4, 3, 2, 1 和 3, 2, 1, 4, 5 也分别是其合法的输出序列。分别求解下列问题：

（1）判断序列 1, 3, 4, 5, 2 是否是合法的输出序列。

（2）对输入序列 1, 2, 3, 4, 5，求出其所有的合法的输出序列。

（3*）设计算法以判断对输入序列 1, 2, 3, … , n，序列 $a_1, a_2, a_3, \dots, a_n$ 是否是该栈的合法的输出序列（假设输出序列在数组 A 中）。

【解】

（1）手工判断栈的输出序列的合法性是各类考试中经常出现的题型。我们用一个更具典型性的序列来介绍判定方法，假设入栈顺序为 1、2、3、4、5，判定序列 4、3、5、2、1 的合法性。

从左往右依次考察输出序列中的每个数字，序列中 4 为第一个出栈数字，说明比它小的数字 1、2、3 一定在栈中，此后无论怎样操作一定会是 3 出栈、再 2 出栈、再 1 出栈，即比当前数字 4 小的数字，不在其左边出现的话，在其右边一定会按降序出现，即一定先出现 3，后出现 2，再出现 1。次输出序列中 3、2、1 在 4 的右边降序出现合法。再分析输出序列中第二个数字 3，比它小的数字 1、2 没有出现在 3 的左边，在其右边降序出现，也合法。再检查 5，比它小的 1、2、3、4，其中 4、3 出现在其左边，无关升降，1、2 在其右边降序合法。再看 2，只有 1 比它小，在其右边，也合法。所以，此序列是栈的一个合法输出序列。

总结判定方法：对给定的输出序列，从左往右依次检查序列中的每个数字。对当前检查数字，比它小的数字，或者出现在这个数字的左边；如果出现在其右边，这些数字一定按降序出现，为合法。如果输出序列中每个数字都满足这个条件即为合法序列，否则非法。

判定序列 4、3、5、1、2 是否合法序列？对 4，比它小的 1、2、3 均出现在其右边，但 1、2 不是降序，所以此序列非法。

再来分析题中给出的序列 1、3、4、5、2。对 1 没有比它小的数；对 3，比它小的 1 出现在其左边，还有一个 2 在其右边，合法；对 4，1、3 在其左，2 在右，合法；对 5，1、3、4 在左，只有 2 在其右，合法，所以此为栈的一个合法输出序列。其出入栈操作序列为 push(1)pop(1)push(2)push(3)pop(3)push(4)pop(4)push(5)pop(5)pop(2)。

（2）对 n 个元素的入栈序列，合法的出栈序列数为 Catalan 数，等于

$$\frac{C_{2n}^{n}}{n+1}=\frac{(2n)!}{(n+1)\times n!\times n!}=\frac{(2n)!}{(n+1)!\times n!}$$

本题中入栈序列 5 个元素，可以计算出合法的出栈序列数为 42 个。

对于个数较少的入栈序列可用合法的 push 和 pop 操作来枚举出所有的合法出栈序列。如果入栈序列元素较多，要使用栈和递归技术编写程序来输出所有出栈序列。

【一种求解算法描述】

```
//求解所有可能出栈序列
void legalSequence(Stack<int,100> S, elementType In[], elementType Out[], int len, int i, int j)
{
    //In[]入栈序列，也可以用栈来保存，但要注意顺序。还可用 C++的 vector 保存。
    //Out[]出栈序列，也可以用栈来保存，但要注意输出循序。还可用 C++的 vector 保存。
    //len 序列长度
    //i 入栈序列元素指针，指示当前处理的入栈序列元素
    //j 出栈序列元素指针，指示当前获取的出栈序列元素

    //每次操作有可能有 2 中操作，要么出栈，要么入栈，且 2 种操作是或的关系。但，出栈、入栈递归返回后要恢复递归前状态

    elementType x;

    if(S.empty() && j>=len)  //递归出口，获得了一个出栈序列
    {
        seqNum++;            //序列数加 1
        seqPrint(Out, len);  //打印序列
    }
    else if(!S.empty() && i<len)  //栈不空，入栈序列中还有数据
    {
            //选择出栈
        S.getTop(x);
        S.pop();
        Out[j]=x;
        j++;
        legalSequence(S, In, Out, len, i, j);
        j--;  //递归返回，恢复到出栈前的状态
        S.push(x);
            //选择入栈
        S.push(In[i]);
        i++;
        legalSequence(S, In, Out, len, i, j);
        i--;     //恢复到入栈前的状态
        S.pop();
    }
```

```
    else if(!S.empty() && i>=len)  //栈不空，入栈序列数据已经处理结束
    {
        //此时只能选择出栈操作
        S.getTop(x);
        S.pop();
        Out[j]=x;
        j++;
        legalSequence(S, In, Out, len, i, j);
        j--;   //恢复到出栈前状态
        S.push(x);
    }
    else if(S.empty() && i<len)  //栈空，入栈序列未处理结束
    {
        //此时，只能选择入栈操作
        S.push(In[i]);
        i++;
        legalSequence(S, In, Out, len, i, j);
        i--;   //恢复到入栈前的状态
        S.pop();
    }

}
```

（3）用数组 In[]保存入栈序列，Out[]保存输出序列，分别用指针 i，j 指示。用一个栈模拟进出站的操作。对每个输出元素 Out[j]做如下操作：

如果 In[i]<Out[j]，将 In[i]入栈，指针 i 后移，即 i++；

如果 In[i]==Out[j]，2 个指针同时后移，即 i++，j++；

如果栈顶元素==Out[j]，栈顶元素出栈，j 后移，即 j++；

循环结束后，如果栈空，且入栈序列处理完毕，则输出序列是一个合法的出栈序列，否则不是。

【算法描述】

```
bool legalSequence(elementType In[], elementType Out[], int len)
{
    //In[]保存入栈序列
    //Out[]保存输出序列
    int i=0,j=0;  //i 为入栈序列指针；j 为输出序列指针
    elementType x;
    Stack<int,100> S;  //初始化顺序栈
    while(j<len)
    {
        S.getTop(x);
```

```
            if(In[i]<Out[j] && i<len)  //入栈序列当前元素小于输出序列当前元素
            {
                S.push(In[i]);   //输入序列当前元素入栈
                i++;             //入栈序列指针后移
            }
            else if(In[i]==Out[j])    //入栈序列当前元素等于输出序列当前元素
            {
                i++;   //指针同时后移
                j++;
            }
            else if(x==Out[j])   //栈顶元素等于输出序列当前元素
            {
                S.pop();    //出栈
                j++;        //输出序列指针后移
            }
            else
                break;
        }
        if(S.empty() && i>=len)  //栈空，且入栈序列处理完毕，则输出序列为合法出栈序列，返回 true
            return true;
        else
            return false;
    }
```

3.2 如果顺序栈中的第二个分量是记录元素个数的变量而不是栈顶指针，应如何实现各算法？

【解】

【结构定义和算法描述】

```
class Stack
{
    public:
        Stack();
        virtual ~Stack(){};

        bool empty();
        bool full();
        bool getTop(elementType &x);
        bool push(elementType x);
        bool pop();
```

```
        void print();  //打印栈内元素
    private:
        int count;      //Using the number of the data elements, count, replaces
the stack top pointer, top.
        elementType data[MaxLen];
};

//1. 初始化
Stack::Stack()
{
    count=0;  //初始化空栈，元素个数为 0
}

//2. 判断栈空
bool Stack::empty()
{
    if(count==0)
        return true;
    else
        return false;
}

//3. 判断栈满
bool Stack::full()
{
    if(count==MaxLen)
        return true;
    else
        return false;
}

//4. 取栈顶元素
bool Stack::getTop(elementType &x)
{
    if(empty())
        return false;  //栈空，取栈顶失败
    else
    {                  //取得栈顶，返回 true；取得的值有 x 传递。
        x=data[count-1];
        return true;
    }
```

```
}

//5. 入栈
bool Stack::push(elementType x)
{
    if(full())
        return false; //栈满，不能入栈，返回 false
    else
    {
        data[count]=x;  //数据入栈
        count++;        //增加元素数，和上一语句不能颠倒次序
        return true;
    }
}

//6. 出栈
bool Stack::pop()
{
    if(empty())     //空栈，没元素出栈，返回 false
        return false;
    else
    {
        count--;
        return true;
    }
}

//7. 打印栈内元素
void Stack::print()
{
    int i;
    if(empty())
        cout<<"空栈！"<<endl;
    else
    {
        cout<<"站内元素（从栈底到栈顶）"<<endl;
        for(i=0;i<count;i++)
            cout<<data[i]<<"\t";
        cout<<endl;
    }
}
```

3.3 设计出链栈的各基本问题求解的算法，并分析其时间复杂度。

3.4 对一个合法的数学表达式来说，其中的各大小括号“{”，“}”，“[”，“]”，“(”和“)”应是相互匹配的。设计算法对以字符串形式读入的表达式 S，判断其中的各括号是否是匹配的。

【解】

【算法思想】

括号匹配的四种可能性：

① 左右括号配对次序不正确

② 右括号多于左括号

③ 左括号多于右括号

④ 左右括号匹配正确

算法实现思想如下：

顺序扫描算数表达式（表现为一个字符串），当遇到三种类型的左括号时候让该括号进栈；当扫描到某一种类型的右括号时，比较当前栈顶元素是否与之匹配，若匹配，退栈继续判断；若当前栈顶元素与当前扫描的括号不匹配，则左右括号配对次序不正确；若字符串当前为某种类型的右括号而堆栈已经空，则右括号多于左括号；字符串循环扫描结束时，若堆栈非空（即堆栈尚有某种类型的左括号），则说明左括号多于右括号；否则，括号配对正确。

【算法描述】

```
bool bracketMatch2(string str)
{
    Stack<char,100> S;  //初始化一个字符串顺序栈
    int len;
    int i=0;
    int result=false;
    char x;             //保存栈顶字符（某种左括号）
    bool tag=true;
    len=str.length();   //求字符串长度
    while(i<len && tag==true)
    {
        switch(str[i])
        {
        case '(':
        case '[':
        case '{':
            S.push(str[i]);
            break;
        case ')':
```

```
        if(S.empty())
        {
            tag=false;
            result=false;
            break;
        }
        S.getTop(x);  //取当前栈顶
        if(x=='(')
        {
            S.pop();
            break;
        }
        else
        {
            tag=false;
            result=false;
            break;
        }
    case ']':
        if(S.empty())
        {
            tag=false;
            result=false;
            break;
        }
        S.getTop(x);  //取当前栈顶
        if(x=='[')
        {
            S.pop();
            break;
        }
        else
        {
            tag=false;
            result=false;
            break;
        }
    case '}':
        if(S.empty())
        {
            tag=false;
```

```
                result=false;
                break;
            }
            S.getTop(x);  //取当前栈顶
            if(x=='{')
            {
                S.pop();
                break;
            }
            else
            {
                tag=false;
                result=false;
                break;
            }

        default:
            break;
        }
        i++;
    }
    if(S.empty() && tag==true)
    {
        result=true;
    }
    else
        result=false;

    return result;
}
```

3.5 对表达式 5+3*(12+4)/4-8，依次画出在求解过程中的各步骤中的栈的状态。

【解】

构建 2 个栈，分别用于保存运算符（含定界符）和操作数。以'#'作为整个表达式的定界符。

3.6*设计算法以求解所读入的表达式的值。假设数据类型为整型，并且仅包含加减乘除四则运算。

【解】算法思想和步骤：

中缀表达式直接求值需要借助 2 个栈：操作符栈（oS）和操作数栈（nS）。算法如下：

（1）自左往右扫描中缀表达式；

（2）如果是操作数，压入操作数栈 nS；

（3）如果是算符（包括括号）：

a）如果栈空，或'('，当前算符直接入算符栈 oS；

b）如果是')'，将算符栈 oS 中'('之前全部算符逐次弹出，对弹出的每个算符，从操作数栈 nS 弹出 2 个操作数（先弹出的是右操作数，后弹出的是左操作数），进行运算，运算结果压入操作数栈 nS，最后释放一对括号；

c）其它算符：

如果 oS 栈顶算符优先级大于等于当前算符，弹出栈顶算符，从操作数栈 nS 弹出 2 个操作数（先弹出的是右操作数，后弹出的是左操作数），进行运算，运算结果压入操作数栈，重复此操作直到栈顶算符优先级低于当前算符。

最后将当前算符压入栈 oS（此时，当前算符优先级高于栈顶算符）。

（4）表达式扫描结束，弹出栈中所有剩下算符，每弹出一个算符，都要完成上述相同的操作数处理。

（5）输出操作数栈 nS 中最后剩下的一个操作数即为计算结果。

【算法描述】

计算中缀表达式需要用到操作符和操作数 2 个栈，算符栈为字符，操作数栈为数字，一套栈代码处理两种类型数据，在 C++中可以用模板完成，在 C 中处理起来复杂一点，要么做两套栈代码，要么用 C 的联合（union）来处理，这里就用了联合定义栈的数据类型 elementType。

```
typedef union ele      //因为要用到字符栈、数值栈，
{                  //用联合来处理，否则要单独定义 2 个栈，分别处理算符和操作数
    char ch;       //用于字符栈，保存字符
    double num;    //用于操作数栈，保存操作数
} elementType;
```

另外需要几个辅助函数：

```
//判定一个算符是否运算符
int isOp(char ch)
{
    char op[]={'+','-','/','*','(',')'};
    int i;
    for(i=0;i<6;i++)
    {
        if(ch==op[i])
            return 1;
    }
    return 0;
}

//定义算符优先级别，用以比较算符优先级
```

```
int level(char ch)
{
    switch(ch)
    {
    case '+':
    case '-':
        return 1;
        break;
    case '*':
    case '/':
        return 2;
        break;
    default:
        return 0;       //运算符为括号'('或')'时，级别为 0
        break;
    }
}

//判定是否数字字符或小数点
bool isNum(char ch)
{
    char numList[]={'0','1','2','3','4','5','6','7','8','9','.'};
    int i=0;
    while(numList[i]!='\0')
    {
        if(numList[i]==ch)
            return true;
        i++;
    }
    return false;
}

//算术运算，计算 2 个操作数，返回结果
double calc(double n1,double n2,char ch)
{
    switch(ch)
    {
    case '+':
        return n1+n2;
        break;
    case '-':
```

```
            return n1-n2;
            break;
        case '*':
            return n1*n2;
            break;
        case '/':
            if(n2==0)
            {
                cout<<"0 除溢出。"<<endl;
                return 0;
            }
            return n1/n2;
            break;
        default:
            return NULL;
            break;
        }
    }

    //中缀表达式求值
    void inExCalc(char inEx[])
    {
        seqStack S1,S2;      //运算符栈、操作数栈
        int i,j;
        char ch1,ch2;
        char nStr[MAXLEN];  //处理一个操作数
        elementType x;
        double n1=0,n2=0;   //存放 2 个操作数

        initialStack(S1);
        initialStack(S2);


        i=0;
        while(i<strlen(inEx))
        {
            ch2=inEx[i];
            if(isOp(ch2))  //ch2 是算符，包括括号
            {
                if(stackEmpty(S1) || ch2=='(')   //算符栈为空，或 ch2 为'(', ch2
直接入栈
```

```
                {
                    x.ch=ch2;
                    push(S1,x);
                }
                else if(ch2==')')    //弹出算符栈 S1 中，'('之前的所有运算符，完成运算，最后释放一对括号
                {
                    getTop(S1,x);
                    while(x.ch!='(')
                    {
                        ch1=x.ch;

                        pop(S2,x);
                        n2=x.num;
                        pop(S2,x);
                        n1=x.num;

                        x.num=calc(n1,n2,ch1);    //计算中间结果，入操作数栈
                        push(S2,x);

                        pop(S1,x);          //栈顶运算符出栈
                        getTop(S1,x);     //取下一个栈顶算符
                    }
                    pop(S1,x);              //释放'('
                }
                else      //ch2 为其它算符
                {
                    getTop(S1,x);
                    ch1=x.ch;
                    while(level(ch1)-level(ch2)>=0)  //栈顶算符 ch1 优先级大于或等于 ch2，弹出进行运算
                    {
                        pop(S1,x);
                        ch1=x.ch;

                        pop(S2,x);
                        n2=x.num;
                        pop(S2,x);
                        n1=x.num;
                        x.num=calc(n1,n2,ch1);    //计算中间结果，入操作数栈
                        push(S2,x);
```

```
                if(!stackEmpty(S1))
                {
                    getTop(S1,x);
                    ch1=x.ch;
                }
                else
                    break;
            }
            x.ch=ch2;   //此时，ch2 算符优先级高于栈顶，算符入栈
            push(S1,x);
        }
        i++;
    }
    else    //ch2 不是算符，是操作数
    {
        nStr[0]='\0';    //nStr 初始化为空串
        j=0;
        while(isNum(inEx[i]))
        {
            nStr[j]=inEx[i];    //获取一个操作数
            j++;
            i++;
        }
        nStr[j]='\0';

        x.num=strtod(nStr,NULL);
        //cout<<x.num<<endl;
        push(S2,x);    //一个操作数入栈
    }
}

//S1 中剩下算符弹出完成计算
while(!stackEmpty(S1))
{
    pop(S1,x);
    ch1=x.ch;
    pop(S2,x);
    n2=x.num;
    pop(S2,x);
    n1=x.num;
```

```
            x.num=calc(n1,n2,ch1);
            push(S2,x);
        }

        //弹出最终计算结果
        pop(S2,x);
        cout<<"中缀表达式计算结果：="<<x.num<<endl;
    }
```

3.7*设计算法以求解所读入的表达式的值。假设数据类型为浮点型，并且仅包含加减乘除四则运算。

【解】算法同上题，上题既可以处理整数，也能处理浮点数。

3.8 用一个数组、头指针和元素个数合在一起所构成的结构来存储顺序队列，设计算法以实现队列的各运算。

【解】相关运算如下

```
//定义相关符号
#define MaxLen 100
typedef int elementType;

//-----------------------------------------------------------------
//顺序队列存储结构定义--注意定义的不同
typedef struct sQueue
{
    elementType data[MaxLen];
    int front;   //队头指向队头元素的前一个单元，初始化指向单元 -1
    int queueLength;
}seqQueue;

//1. 初始化顺序队列
void initQueue(seqQueue* Q) //参数号可用传值、引用
{
    Q->front=-1;   //空队列
    Q->queueLength=0;
}

//-----------------------------------------------------------------
//2. 判队空
bool queueEmpty(seqQueue &Q)
{
```

```
    if(Q.queueLength==0)
        return true;   //队空，返回 true
    else
        return false;   //队不空
}

//3. 判队满--存在假溢出情况
bool queueFull(seqQueue &Q)
{
    if(Q.queueLength==MaxLen)
    {
        cout<<"<-- 当前队满 -->"<<endl;
        return true;   //队满，返回 true
    }
    else if(Q.front==MaxLen-1)
    {
        cout<<"<-- 假溢出 -->"<<endl;
        return true;
    }
    else
        return false; //不满，返回 false
}

//4. 取队头元素
void queueFront(seqQueue &Q, elementType &x)
{
    if(queueEmpty(Q))
        cout<<"队空，不能取队头元素！"<<endl;
    else
        x=Q.data[(Q.front+1)];
}

//5. 入队（插入）
void enQueue(seqQueue* Q, elementType x)
{
    if(queueFull(*Q))
        cout<<"队列已满，不能完成入队操作！"<<endl;
    else
    {
        //Q->front++;
        Q->queueLength++;   //增加长度
```

```
            Q->data[Q->queueLength-1]=x;   //填入数据 x

        }
    }

    //6. 出对（删除）
    void outQueue(seqQueue* Q)
    {
        if(queueEmpty(*Q))
            cout<<"空队列，没有元素可供出队！"<<endl;
        else
        {
            Q->front++;
            Q->queueLength--;
        }
    }
```

3.9 对教材中所讨论的循环队列及其约定，给出求解队列中元素个数的表达式。

【解】

设循环队列为 Q，数组最大长度为 MaxLen，头指针和尾指针分别为 front 和 rear

元素个数=(Q.rear-Q.front+MaxLen) % MaxLen

3.10 如果对循环队列采用设置运算标志的方法来区分队列的满和空的状态，试给出对应的各运算的实现。

3.11 如果采用带尾指针的单循环链表作为队列的存储结构，设计算法以实现队列的各运算。

【解】

```
//带尾指针单循环链表实现链队列，以下 Q 为单循环链表尾指针
#include "..\scrLinkedList.h"       //带尾指针单循环链表
typedef linkedList linkedQueue;
//--1.链队初始化-----------------------------------------------
void initialQueue(linkedQueue &Q)
{
    Q=new node;   //产生头结点，指针为 R;
    Q->next=Q;
}
//--2.判断队空-------------------------------------------------
bool queueEmpty(linkedQueue &Q)
```

```
{
    return (Q==Q->next);
}
//--3.取队头元素-----------------------------------------------
bool getFront(linkedQueue &Q, elementType &x)
{
    if(queueEmpty(Q))
        return false;   //空队列，无法取队头元素
    else
    {
        x=(Q->next->next)->data;
        return true;
    }
}
//--4.入队-----------------------------------------------------
void enQueue(linkedQueue &Q, elementType x)
{
    node* p=new node;   //申请内存，产生新节点
    p->data=x;
    p->next=Q->next;
    Q->next=p;
    Q=p;
}
//--5.出队-----------------------------------------------------
bool outQueue(linkedQueue &Q, elementType &x)
{
    node* u; //用以指向删除节点
    if(queueEmpty(Q))
        return false;       //队空，无法执行出队操作
    else
    {
        x=Q->next->next->data;       //取出队头元素
        u=Q->next->next;              //u 指向队头
        Q->next->next=u->next;
        if(Q==u)     //删除最后一个节点，成为空队列
            Q=Q->next;
        delete u;                    //删除原队头，释放内存
        return true;
    }
}
```

3.12 写出下面程序或调用的结果:

(1)
```
void  P1(int W);
      {  int A,B;
         A=W-1;  B=W+1;
         cout<<A<<B;
      }
void  P2(W int );
   {  int A,B;
      A=2*W;  B=W*W;
      P1(A);  P1(B);
      cout<<A<<B;
   }
调用  P2(5);
```

【解】执行结果为:

9 11 24 26 10 25

(2)
```
int  Hcf(int M,int N)
   { int H;
      while (N!=0)
          {
              H=M % N;
              M=N; N=H
          } ;
      cout<<M;  return M;
   } ;
调用  cout<<Hcf(100, 350);
      cout<<Hcf(200, 49);
```

【解】结果为:

50 50 1 1

(3)
```
int Hcf(int M, int N)
   {   int H;
      while (N!=0)
          {
              H=M mod N;
              M=N; N=H
          }
      return M;
   }
```

```
void  reduce(int M1, int N1; int *M2, int *N2)
    {  int R;
       R=Hcf(M1,N1);
        *M2=M1 / R;  *N2=N1/ R;
      cout<<M1<<'/'<<N1<<'='<<M2<<'/'<<N2;
    }
调用 reduce(100,200, X,Y);
     reduce(300,550, M,N);
```

【解】

100/200 = 1/2

300/550 = 6/11

3.13 阅读下列程序，并写出其运行结果:

(1)
```
void  P(int W)
  {
    if (W>0)
      {   P(W-1);
          cout<<W;
      }
  }
调用 P(4);
```

【解】结果为:

1 2 3 4

(2)
```
void  P(int W)
  {
      if (W>0)
        {   cout<<W;
            P(W-1);
        }
  }
调用 P(4);
```

【解】结果为:

4 3 2 1

(3)
```
void  P(int W)
```

```
        {
          if (W>0)
            {  cout<<W;
               P(W-1);
               cout<<W;
            }
        }
    调用 P(4);
```

【解】结果为：

4 3 2 1 1 2 3 4

```
(4)   void  P(int W)
        {
            if (W>0)
              {   P(W-1);
                  P(W-1);
                  cout<<W;
               }
        }
    调用 P(4);
```

【解】结果为：

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

```
(5)   int   F(int N)
        {
            if ( N==0) F=0;
            else  if (N==1)
                   return 1;
                  else  return F(N-1)+2*F(N-2);
        }
    调用 cout<<F(5);
```

【解】结果为：

11

```
(6)   void  P(int N,int *F);
        {
```

```
        if (N==0)
         *F=0;
        else  {  P(N-1, *F); F=F+N;  }
    }
  调用 P(4,*M); cout<<*M
```

解：结果为：

10

3.14 求解下面各调用的结果，并指出算法的功能。

(1)
```
void  PrintRV(int N)
    {
        if (N>0)
          {  cout<<N % 10;
             PrintRV(N / 10);
          }
    } ;
  调用 PrintRV(12345);
```

【解】结果为：

54321　10 进制数逆序输出

(2)
```
void  PC(int M, int N; int *K );
    {
        if (N==0)
          *K=1;
        else {  PC(M-1,N-1,*K); *K=*K*M / N;  }
    } ;
  调用  PC(10,4,*M); cout<<*M;
```

【解】结果为：

210　功能：$\frac{m}{n}\times\frac{m-1}{n-1}\times\frac{m-2}{n-2}\times\cdots\times 1$

(3)
```
int   SS(int N)
    {
        if (N==0 ) return 100;
      else  return SS(N-1)+N*N;
```

```
    }
调用  cout<<SS(5);
```

【解】结果为：

155　功能：$100+N^2+(N-1)^2+...+1$

(4)
```
int   ACM(int M, int N)
    {
      if (M==0) return N+1;
      else  if (N==0) return ACM(M-1,1);
            else return ACM(M-1,ACM(M,N-1));
    }
调用  cout<<ACM(2,2);
```

【解】结果为：

7

3.15 对下面的函数定义，证明下面有关程序功能描述的正确性。

(1)
```
void  P(int N)
    {
    if (N>0)
      {  P(N-1);
         cout<<N;
         P(N-1);
      }
    }
```

a. 在调用 P(N)所产生的输出序列中，当且仅当其序号为奇数时，输出项为 1。

b. 在调用 P(N)所产生的输出序列中，输出 2 的项数为 2^{n-2}。

(2)下面函数的设计目标是求整型数组 A 中下标从 i 到 j 的各元素的最大值，请判断该函数的正确性。另外，函数中所定义的局部变量是否是必需的？

```
int Max(int i, int j)
    { int M1,M2,Mid;
      if (i==j ) return A[I];
      else { Mid=(i+j) / 2;
             M1=Max(i,Mid);
             M2=Max(Mid+1,j);
             if (M1>M2 )return M1;
                   else return M2;
           }
```

}

3.16 已知 ACKMANN 函数定义如下，试分别编写出求解该函数的递归过程和递归函数。

$$\begin{cases} Ack(m,n)=n+1 & m=0 \\ Ack(m,n)=Ack(m-1,1) & n=0 \\ Ack(m,n)=Ack(m-1,Ack(m,n-1)) & m>0,n>0 \end{cases}$$

【解】

```
int ACK(int m, int n)
{
    if(m==0)
        return n+1;
    else if(n==0)
        return ACK(m-1,1);
    else
        return ACK(m-1,ACK(m,n-1));
}
```

3.17 将下面递归程序转换为等价的非递归程序。

```
(1) void  P(int n)
    {
      if (n>0)
        {  cout<<n;
           P(n-1);
           cout<<n;
        }
    }
```

【解】

```
//循环求解
void P(int n)
{
    int *A;
    int i;
    A=new int[n+1];  //动态数组,0 单元不用

    for(i=n;i>0;i--)
    {
        cout<<i;
        A[i]=i;
    }
    for(i=1;i<=n;i++)
```

```
        cout<<A[i];
    cout<<endl;
    delete A;
}
//利用栈转换为非递归
void P1(int n)
{
    seqStack S;
    elementType m;
    initialStack(S);
    while(n>0 || !stackEmpty(S))
    {
        if(n>0)
        {
            cout<<n;
            push(S,n);
            n=n-1;
        }
        else
        {
            pop(S,m);    //出栈，栈顶元素存到 m
            cout<<m;
        }
    }
    cout<<endl;
}
```

(2)
```
void  P(int n)
    {
      if (n>0)
       {  P(n-1);
          cout<<n;
          P(n-1);
       }
    }
```

【解】

```
//递归程序--本题类似二叉树中序遍历
void P(int n)
{
    if(n>0)
    {
```

```
        P(n-1);
        cout<<n;
        P(n-1);
    }
}

//利用栈转换为非递归
void P1(int n)
{
    seqStack S;
    elementType m;
    initialStack(S);
    while(n>0 || !stackEmpty(S))
    {
        if(n>0)    //n>0，入栈
        {
            push(S,n);
            n=n-1;
        }
        else      //n<=0，但栈不空，取栈顶打印，出栈
        {
            pop(S,n);    //出栈，栈顶元素存到 n
            cout<<n;
            n=n-1;
        }
    }
    cout<<endl;
}
```

(3) int f(int n)

```
    {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return f(n-1)+f(n-2);
    }
```

【解】

```
// 循环求解
int f(int n)
{
    int f1,f2,tem,i;
    if(n==0)
```

```
        return 0;
    else if(n==1)
        return 1;
    else
    {
        f1=1;
        f2=0;
        for(i=2;i<=n;i++)
        {
            tem=f1;
            f1=f1+f2;  //f(n-1)+f(n-2)
            f2=tem;    //跟新 f2
        }
        return f1;
    }
}
//利用栈转换为非递归
int f(int n)
{
    int f1,f2,tem;
    seqStack S;
    elementType m;
    initialStack(S);
    if(n==0)
        return 0;
    else if(n==1)
        return 1;
    while(n>1)
    {
        push(S,n);
        n=n-1;
    }
    f2=0;
    f1=1;
    while(!stackEmpty(S))
    {
        tem=f1;
        f1=f1+f2;
        f2=tem;
        S.data[S.top]=f1;
        pop(S,f1);
```

```
    }
    return f1;
    cout<<endl;
}
```