第三章 上机

练习题

1. 一个 b 序列的长度为 n，其元素恰好是 1～n 的某个排列，编写一个实验程序判断 b 序列是否为以 1,2,…,n 为进栈序列的出栈序列。如果不是，输出相应的提示信息；如果是，输出由该进栈序列通过一个栈得到 b 序列的过程。

2. 改进用栈求解迷宫问题的算法，累计如图 2.20 所示的迷宫的路径条数，并输出所有迷宫路径。

3. 括号匹配问题：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(与常见的算术表达式一样)任何一个左括号都从内到外与它右边距离最近的右括号匹配。编写一个实验程序，找到无法匹配的左括号和右括号，输出原来的字符串，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。例如，输出样例如下：

```
((ABCD(x)
$$
)(rttyy())sss)(
?          ?  $
```

4. 修改《教程》3.2 节中的循环队列算法，使其容量可以动态扩展。当进队时，若容量满按两倍扩大容量；当出队时，若当前容量大于初始容量并且元素的个数只有当前容量的 1/4，缩小当前容量为一半。通过测试数据说明队列容量变化的情况。

5. 采用不带头结点只有一个尾结点指针 rear 的循环单链表存储队列，设计出这种链队的进队、出队、判队空和求队中元素个数的算法。

6. 对于如图 2.21 所示的迷宫图，编写一个实验程序，先采用队列求一条最短迷宫路径长度 minlen(路径中经过的方块个数)，再采用栈求所有长度为 minlen 的最短迷宫路径。在搜索所有路径时进行这样的优化操作：当前路径尚未到达出口但长度超过 minlen，便结束该路径的搜索。

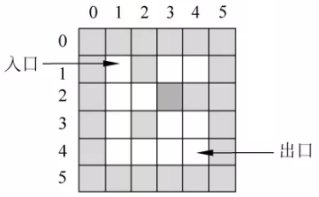


图 2.20　一个迷宫的示意图

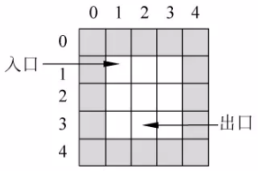


图 2.21　一个迷宫的示意图

解题思路

1. 解：判断 b 序列是否为以 1,2,…,n 为进栈序列的出栈序列的过程参见《教程》第 3 章中的例 3.9。对应的实验程序 Exp2-1.py

```
from SqStack import SqStack
def isSerial(b):                    # 判断算法
    n=len(b)
```

```
    ops=""
    st=SqStack()                        # 建立一个顺序栈
    i,j=1,0
    while i<=n:                         # 遍历 a 序列
        st.push(i)
        ops+="   "+str(i)+" 进栈\n"
        i+=1                            #i 后移
        while not st.empty() and st.gettop()==b[j]:
            e=st.pop()                  # 出栈
            ops+="   "+str(e)+" 出栈\n"
            j+=1                        #j 后移
    if st.empty():                      # 栈空返回 ops
        return ops
    else:
        return ""

# 主程序
#n=input(" 整数:")
print()
print(" 测试 1")
b=[1,3,4,2]
ret=isSerial(b)
if ret=="":
    print(+" ",b,end='  ')
    print(" 不是合法出栈序列")
else:
    print(" ",b,end=' ')
    print(" 是合法出栈序列, 操作如下:")
    print(ret)

print(" 测试 2")
b=[4,2,1,3]
ret=isSerial(b)
if ret=="":
    print(" ",b,end='')
    print(" 不是合法出栈序列")
else:
    print(" ",b,end='')
    print(" 是合法出栈序列, 操作如下:")
    print(ret)
```

2. 解：修改《教程》中 3.1.6 节用栈求解迷宫问题的 mgpath()算法，用 cnt 累计找到的迷宫路径条数(初始为 0)。在找到一条路径后并不返回，而是将 cnt 增加 1，输出该迷宫路径，然后出栈栈顶方块 b 并将该方块的 mg 值恢复为 0，继续前面的过程，直到栈空为止，最后返回 cnt。对应的实验程序 Exp2-2.py 如下：

```
from SqStack import SqStack
cnt=0                                   # 累计迷宫路径条数
class Box:                              # 方块类
    def __init__(self,i1,j1,di1):       # 构造方法
        self.i=i1                       # 方块的行号
        self.j=j1                       # 方块的列号
        self.di=di1                     #di 是下一可走相邻方位的方位号

def mgpath(xi,yi,xe,ye):                # 求一条从 (xi,yi) 到 (xe,ye) 的迷宫路径
    global mg                           # 迷宫数组为全局变量
    global cnt
    st=SqStack()                        # 定义一个顺序栈
    dx=[-1,0,1,0]                       #x 方向的偏移量
    dy=[0,1,0,-1]                       #y 方向的偏移量
    e=Box(xi,yi,-1)                     # 建立入口方块对象
    st.push(e)                          # 入口方块进栈
    mg[xi][yi]=-1                       # 为避免来回找相邻方块，将进栈的方块置为-1
    while not st.empty():               # 栈不空时循环
        b=st.gettop()                   # 取栈顶方块，称为当前方块
        if b.i==xe and b.j==ye:         # 找到了出口，输出栈中所有方块构成一条路径
            cnt+=1
            print("  迷宫路径"+str(cnt)+": ",end='');    # 输出一条迷宫路径
            for k in range(len(st.data)):
                print("["+str(st.data[k].i)+','+str(st.data[k].j)+"]",end=' ')
            print()
            b=st.pop()                  # 退栈
            mg[b.i][b.j]=0              # 让该位置变为其他路径可走方块
        else:
            find=False                          # 否则继续找路径
            di=b.di
            while di<3 and find==False:         # 找 b 的一个相邻可走方块
                di+=1                           # 找下一个方位的相邻方块
                i,j=b.i+dx[di],b.j+dy[di]       # 找 b 的 di 方位的相邻方块 (i,j)
                if mg[i][j]==0:                 #(i,j) 方块可走
                    find=True
            if find:                            # 找到了一个相邻可走方块 (i,j)
                b.di=di                         # 修改栈顶方块的 di 为新值
                b1=Box(i,j,-1)                  # 建立相邻可走方块 (i,j) 的对象 b1
                st.push(b1)                     #b1 进栈
                mg[i][j]=-1                     # 为避免来回找相邻方块，将进栈的方块置为-1
            else:                               # 没有路径可走，则退栈
                mg[b.i][b.j]=0                  # 恢复当前方块的迷宫值
                st.pop()                        # 将栈顶方块退栈
    return cnt                              # 没有找到迷宫路径，返回 False

# 主程序
```

```
mg=[[1,1,1,1,1,1],[1,0,1,0,0,1],[1,0,0,1,1,1],[1,0,1,0,0,1],[1,0,0,0,0,1],[1,1,1,1,1,1]]
xi,yi=1,1
xe,ye=4,4
print()
cnt=mgpath(xi,yi,xe,ye)          #(1,1)->(4,4)
if cnt==0:
    print("  不存在迷宫路径")
else:
    print("  共计"+str(cnt)+" 条迷宫路径")
```

3. 解：对于字符串 s，设对应的输出字符串为 mark，采用栈 st 来产生 mark。遍历字符 $s[i]$，当遇到'('时将其下标 i 进栈，当遇到')'时，若栈中存在匹配的'('，置 mark$[i]$='',否则置 mark$[i]$='?'。当 s 遍历完毕时，若 st 栈不空，则 st 栈中的所有左括号都是没有右括号匹配的，将相应位置 j 的 mark 值置为'$'。对应的实验程序 Exp2-3.py 如下：

```
from collections import deque
def solve(s):
    n=len(s)
    mark=[None]*n                # 输出字符串
    st=deque()                   # 用双端队列作为栈
    for i in range(n):
        if s[i]=='(':            # 遇到'('则入栈
            st.append(i)         # 将数组下标暂存在栈中
            mark[i]=' '          # 对应输出字符串暂且为' '
        elif s[i]==')':          # 遇到')'
            if not st:           # 栈空，如果没有'('相匹配
                mark[i]='?'      # 对应输出字符串改为'?'
            else:                # 有'('相匹配
                mark[i]=' '      # 对应输出字符串改为' '
                st.pop()         # 栈顶位置左括号与其匹配，弹出已经匹配的左括号
        else:                    # 其他字符与括号无关
            mark[i]=' '          # 对应输出字符串改为' '
    while st:                    # 若栈非空，则有没有匹配的左括号
        mark[st[-1]]='$'         # 对应输出字符串改为'$'
        st.pop()
    print("    表达式:",s)
    print("    结  果:",''.join(mark))

# 主程序
print()
print("  测试 1")
s="((ABCD(x)"
solve(s)
print("  测试 2")
s=")(rttyy())sss)("
solve(s)
```

4. 解：用全局变量 Initcap 存放初始容量，队列中增加 capacity 属性表示队列的当前容量，增加 updatecapacity(newcap)方法用于将当前容量改为 newcap。其过程如下：

① 当参数 newcap 正确时(newcap>n)，建立长度为 newcap 的列表 tmp。

② 出队 data 中的所有元素并依次存放到 tmp 中(从 tmp[1]开始)。

③ 置 data 为 tmp，队头指针 front 为 0，队尾指针 rear 为 n，新容量为 newcap。

在进队中队满和出队中满足指定的条件时调用 updatecapacity(newcap)方法。对应的实验程序 Exp2-4.py

```
Initcap=3                                   # 全局变量, 初始容量为 3
class CSqQueue:                             # 非循环队列类
    def __init__(self):                     # 构造方法
        self.data=[None]*Initcap            # 存放队列中元素
        self.capacity=Initcap
        self.front=0                        # 队头指针
        self.rear=0                         # 队尾指针

    def size(self):                         # 返回队中元素个数
        return ((self.rear-self.front+self.capacity)%self.capacity)

    def getcap(self):                       # 返回队容量
        return self.capacity

    def updatecapacity(self,newcap):        # 修改循环队列的容量为 newcap
        n=self.size()
        assert newcap>n                     # 检测 newcap 参数的错误
        print(" 原容量 =%d, 原元素个数 =%d, 修改容量 =%d" %(self.capacity,n,newcap),end='')
        tmp=[None]*newcap                   # 新建存放队列元素的空间
        head=(self.front+1)%self.capacity
        for i in range(n):                  # 出队所有元素存放到 tmp 列表中
            tmp[i+1]=self.data[head]        # 从 tmp[1] 开始, tmp[0] 暂不用
            head=(head+1)%self.capacity
        self.data=tmp                       #tmp 用作 data
        self.front=0                        # 重置 front
        self.rear=n                         # 重置 rear
        self.capacity=newcap                # 重置 capacity


    def empty(self):                        # 判断队列是否为空
        return self.front==self.rear

    def push(self,e):                       # 元素 e 进队
        print("    进队"+str(e),end=': ')
        if (self.rear+1)%self.capacity==self.front:
            self.updatecapacity(2*self.capacity) # 队满时倍增容量
        print()
```

```
        self.rear=(self.rear+1)%self.capacity
        self.data[self.rear]=e

    def pop(self):                              # 出队元素
        assert not self.empty()                 # 检测队空
        self.front=(self.front+1) % self.capacity
        x=self.data[self.front]                 # 取队头元素
        print("    出队"+str(x),end=': ')
        n=self.size()
        if self.capacity>Initcap and n==self.capacity//4:
            self.updatecapacity(self.capacity//2)   # 满足要求则容量减半
        print()
        return x
    def gethead(self):                          # 取队头元素
        assert not self.empty()                 # 检测队空
        head=(self.front+1)%MaxSize
        return self.data[head]

# 主程序
print()
qu=CSqQueue()
print("  (1) 进队 1,2")
qu.push(1)
qu.push(2)
print("  元素个数 =%d, 容量 =%d" %(qu.size(),qu.getcap()))
print("  (2) 进队 3-13:")
for i in range(3,14):
    qu.push(i)
print("  元素个数 =%d, 容量 =%d" %(qu.size(),qu.getcap()))
print("  (3) 出队所有元素:")
while not qu.empty():
    qu.pop()
print("  (4) 元素个数 =%d, 容量 =%d" %(qu.size(),qu.getcap()))
```

5. 解：用只有尾结点指针 rear 的循环单链表作为队列存储结构，如图 2.26 所示，其中每个结点的类型为 LinkNode(同前面链队的结点类)。

在这样的链队中，队列为空时 rear＝None，进队在链表的表尾进行，出队在链表的表头进行。例如，在空链队中进队 a、b、c 元素的结果如图 2.27(a)所示，出队两个元素后的结果如图 2.27(b)所示。

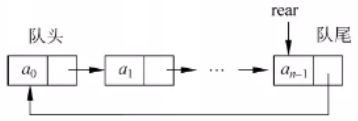
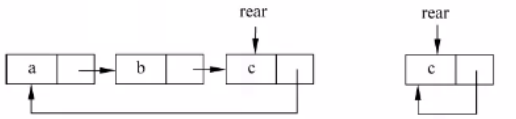


图 2.26　用只有尾结点指针的循环单链表作为队列存储结构



(a) 元素a、b、c进队　　(b) 出队两个元素

图 2.27　链队的进队和出队操作

```
class LinkNode:                          # 链队结点类
    def __init__(self,data=None):        # 构造方法
        self.data=data                   #data 域
        self.next=None                   #next 域

class LinkQueue1:                        # 本例的链队类
    def __init__(self):                  # 构造方法
        self.rear=None                   # 队尾指针
    def empty(self):                     # 判断队是否为空
        return self.rear==None

    def push(self,e):                    # 元素 e 进队
        s=LinkNode(e)                    # 创建新结点 s
        if self.rear==None:              # 原链队为空
            s.next=s                     # 构成循环单链表
            self.rear=s
        else:
            s.next=self.rear.next        # 将 s 结点插入到 rear 结点之后
            self.rear.next=s
            self.rear=s                  # 让 rear 指向 s 结点
    def pop(self):                       # 出队操作
        assert not self.empty()          # 检测空链队
        if self.rear.next==self.rear:    # 原链队只有一个结点
            e=self.rear.data             # 取该结点值
            self.rear=None               # 置为空队
        else:                            # 原链队有多个结点
```

```
            e=self.rear.next.data                  # 取队头结点值
            self.rear.next=self.rear.next.next     # 删除队头结点
        return e
    def gethead(self):                             # 取队顶元素操作
        assert not self.empty()                    # 检测空链队
        if self.rear.next==self.rear:              # 原链队只有一个结点
            e=self.rear.data                       # 该结点也是头结点
        else:                                      # 原链队有多个结点
            e=self.rear.next.data                  #rear.next 为头结点
        return e

# 主程序
print()
qu=LinkQueue1()
print("  (1)1-5 进队")
for i in range(1,6):
    qu.push(i)
print("  (2) 队头 ="+str(qu.gethead()))
print("  (3) 出队 3 次:")
for i in range(3):
    print("     出队元素 ="+str(qu.pop()))
print("  (4) 进队 6-8")
for i in range(6,9):
    qu.push(i)
print("  (5) 出队所有元素")
while not qu.empty():
    print("     出队元素 ="+str(qu.pop()))
```

6. 解：迷宫图用 mg 数组表示，先用队列求一条最短迷宫路径长度 minlen，再恢复 mg，用栈 st 求所有长度为 minlen 的最短迷宫路径并且输出，由于 st 中恰好存放路径中的所有方块，当求栈顶方块 b 扩展时，若 len(st)≥minlen，便恢复方块 b 的 mg 值并出栈，结束该路径的搜索，从而保证找到的路径一定是最短路径。队列和栈均采用双端队列 deque 实现。对应的实验程序 Exp2-6.py 如下：

```
from collections import deque
dx=[-1,0,1,0]                              #x 方向的偏移量
dy=[0,1,0,-1]                              #y 方向的偏移量

class Box1:                                    # 方块类，用作队列元素类型
    def __init__(self,i1,j1):                  # 构造方法
        self.i=i1                              # 方块的行号
        self.j=j1                              # 方块的列号
        self.pre=None                          # 前驱方块

def minpathlen(xi,yi,xe,ye):                   # 求 (xi,yi) 到 (xe,ye) 的一条最短路径长度
```

```
    global mg                                   # 迷宫数组为全局变量
    qu=deque()                                  # 定义一个队列
    b=Box1(xi,yi)                               # 建立入口结点 b
    qu.appendleft(b)                            # 结点 b 进队
    mg[xi][yi]=-1                               # 进队方块 mg 值置为-1
    while len(qu)!=0:                           # 队不空时循环
        b=qu.pop()                              # 出队一个方块 b
        if b.i==xe and b.j==ye:                 # 找到了出口，输出路径
            p=b                                 # 从 b 出发回推导出迷宫路径并输出
            apath=[]
            while p!=None:                      # 找到入口为止
                apath.append("["+str(p.i)+","+str(p.j)+"]")
                p=p.pre
            return len(apath)                   # 返回找到的一条路径长度
        for di in range(4):                     # 循环扫描每个相邻方位的方块
            i,j=b.i+dx[di],b.j+dy[di]           # 找 b 的 di 方位的相邻方块 (i,j)
            if mg[i][j]==0:                     # 找相邻可走方块
                b1=Box1(i,j)                    # 建立后继方块结点 b1
                b1.pre=b                        # 设置其前驱方块为 b
                qu.appendleft(b1)               #b1 进队
                mg[i][j]=-1                     # 进队的方块置为-1
    return -1                                   # 未找到任何路径时返回-1

class Box2:                                     # 方块类，用作栈元素类型
    def __init__(self,i1,j1,di1=None):          # 构造方法
        self.i=i1                               # 方块的行号
        self.j=j1                               # 方块的列号
        self.di=di1                             #di 是下一可走相邻方位的方位号

def mgpath(xi,yi,xe,ye,minlen):                 # 求从 (xi,yi) 到 (xe,ye) 的所有最短迷宫路径
    global mg                                   # 迷宫数组为全局变量
    cnt=0                                       # 累计路径条数
    st=deque()                                  # 以双端队列作为栈
    e=Box2(xi,yi,-1)                            # 建立入口方块对象
    st.append(e)                                # 入口方块进栈
    mg[xi][yi]=-1                               # 为避免来回找相邻方块，将进栈的方块置为-1
    while len(st)>0:                            # 栈不空时循环
        b=st[-1]                                # 取栈顶方块，称为当前方块
        if b.i==xe and b.j==ye:
            cnt+=1                              # 找到一条最短路径
            st1=st.copy()                       # 由 st 复制产生 st1
            apath=[]
            b1=st1.pop()
            while True:                         # 出栈 st1 所有方块得到 apath
                apath.append([b1.i,b1.j])
                if len(st1)==0: break
```

```
            b1=st1.pop()
        apath.reverse()                     # 逆置 apath 得到正向迷宫路径
        print("    路径%d" %(cnt),apath)
        b=st.pop()                          # 退栈
        mg[b.i][b.j]=0                      # 让该位置变为其他路径可走方块
    elif len(st)<minlen:                    # 不是出口且路径长度小于 minlen 时
        find=False                          # 否则继续找路径
        di=b.di
        while di<3 and find==False:         # 找 b 的一个相邻可走方块
            di+=1                           # 找下一个方位的相邻方块
            i,j=b.i+dx[di],b.j+dy[di]       # 找 b 的 di 方位的相邻方块 (i,j)
            if mg[i][j]==0:                 #(i,j) 方块可走
                find=True
        if find:                            # 找到了一个相邻可走方块 (i,j)
            b.di=di                         # 修改栈顶方块的 di 为新值
            b1=Box2(i,j,-1)                 # 建立相邻可走方块 (i,j) 的对象 b1
            st.append(b1)                   #b1 进栈
            mg[i][j]=-1                     # 为避免来回找相邻方块, 将进栈的方块置为-1
        else:                               # 没有路径可走, 则退栈
            mg[b.i][b.j]=0                  # 恢复当前方块的迷宫值
            st.pop()                        # 将栈顶方块退栈
    else:
        mg[b.i][b.j]=0                  # 恢复当前方块的迷宫值
        st.pop()                        # 将栈顶方块退栈


# 主程序
mg=[[1,1,1,1,1],[1,0,0,0,1],[1,0,0,0,1],[1,0,0,0,1],[1,1,1,1,1]]
xi,yi=1,1
xe,ye=3,2
minlen=minpathlen(xi,yi,xe,ye)
for i in range(len(mg)):                    # 恢复迷宫数组
    for j in range(len(mg[i])):
        if mg[i][j]==-1: mg[i][j]=0
print()
print("  所有 [%d,%d] 到 [%d,%d] 的最短迷宫路径:" %(xi,yi,xe,ye))
mgpath(xi,yi,xe,ye,minlen)                  #(1,1)->(3,2)
```

第四章 上机

练习题

1. 编写一个实验程序，假设串用 Python 字符串类型表示，求字符串 s 中出现的最长的可重叠的重复子串。例如，s="ababababa"，输出结果为"abababa"。

2. 编写一个实验程序，假设串用 Python 字符串类型表示，给定两个字符串 s 和 t，求串 t 在串 s 中不重叠出现的次数，如果不是子串则返回 0。例如，s="aaaab"，t="aa"，则 t 在 s 中出现两次。

3. 编写一个实验程序，假设串用 Python 字符串类型表示，给定两个字符串 s 和 t，求串 t 在串 s 中不重叠出现的次数，如果不是子串则返回 0，注意在判断子串时是与大小写无关的。例如，s="aAbAabaab"，t="aab"，则 t 在 s 中出现 3 次。

4. 求马鞍点问题。如果矩阵 a 中存在一个元素 $a[i][j]$满足这样的条件：$a[i][j]$是第 i 行中值最小的元素，且又是第 j 列中值最大的元素，则称之为该矩阵的一个马鞍点。设计一个程序，计算出 $m\times n$ 的矩阵 a 的所有马鞍点。

5. 对称矩阵压缩存储的恢复。一个 n 阶对称矩阵 $\boldsymbol{A}$ 采用一维数组 a 压缩存储，压缩方式为按行优先顺序存放 $\boldsymbol{A}$ 的下三角和主对角线的各元素，完成以下功能：

① 由 $\boldsymbol{A}$ 产生压缩存储 a。

② 由 b 来恢复对称矩阵 $\boldsymbol{C}$。

通过相关数据进行测试。

解题思路

1. 解：采用简单匹配算法的思路，先给最长重复子串的下标 maxi 和长度 maxl 赋值为 0。设 s="$a_0\cdots a_{n-1}$"，i 从头扫描 s，对于当前字符 a_i，判定其后是否有相同的字符，j 从 $i+1$ 开始遍历 s 后面的字符，若有 $a_i==a_j$，再判定 a_{i+1} 是否等于 a_{j+1}，a_{i+2} 是否等于 a_{j+2}，…，直到找到一个不同的字符为止，即找到一个重复出现的子串，把其下标 i 与长度 l 记下来，将 l 与 maxl 相比较，保留较长的重复子串 maxi 和 maxl。再从 a_{j+1} 之后查找重复子串。最后的 maxi 与 maxl 即记录下最长可重叠重复子串的起始下标与长度，由其构造字符串 t 并返回。对应的实验程序 Exp2-1.py 如下：

```
def maxsubstr(s):                       # 求 s 中出现的最长的可重叠的重复子串
    maxi,maxl=0,0
    i=0
    while i<len(s):                     #i 遍历 s
        j=i+1                           # 从 i+1 开始
        while j<len(s):                 # 查找与 si 开头的相同重复子串
            if s[i]==s[j]:              # 遇到首字符相同
                l=1
                while j+l<len(s) and s[i+l]==s[j+l]:
                    l+=1                # 找到重复子串 (i,l)
                if l>maxl:              # 存放较长子串 (maxi,maxl)
                    maxi=i
                    maxl=l
```

```
                j+=l                    #j 跳过 l 个字符继续在后面找子串
            else: j+=1                  # 首字符不相同，j 递增
        i+=1                            #i 后移
    t=""                                # 构造最长重复子串
    if maxl==0:
        t="None"
    else:
        for i in range(maxl):
            t+=s[maxi+i]
    return t
# 主程序
print()
print(" 测试 1")
s="ababcabccabdce"
print(" s: "+s)
print(" s 中最长重复子串: "+maxsubstr(s))
print(" 测试 2")
s="abcd"
print(" s: "+s)
print(" s 中最长重复子串: "+maxsubstr(s))
print(" 测试 3")
s="abababababa"
print(" s: "+s)
print(" s 中最长重复子串: "+maxsubstr(s))
```

2. 解：采用两种解法。用 cnt 累计 t 在串 s 中不重叠出现的次数(初始值为 0)。

① 基于 BF 算法：在找到子串后不是退出，而是 cnt 增加 1，i 增加 t 的长度并继续查找，直到整个字符串查找完毕。

② 基于 KMP 算法：当匹配成功时，cnt 增加 1，并且置 j 为 0 重新开始比较。

```
MaxSize=100
# 基于 BF 算法
def StrCount1(s,t):                     #BF 算法求解
    i,cnt=0,0
    while i<len(s)-len(t)+1:
        j,k=i,0
        while j<len(s) and k<len(t) and s[j]==t[k]:
            j,k=j+1,k+1
        if k==len(t):                   # 找到一个子串
            cnt+=1                      # 累加出现的次数
            i=j                         #i 从 j 开始
        else: i+=1                      #i 增加 1
    return cnt
# 基于 KMP 算法
def GetNext(t,next):                    # 由模式串 t 求出 next 值
    j,k=0,-1
```

```
    next[0]=-1
    while j<len(t)-1:
        if k==-1 or t[j]==t[k]:      #j 遍历后缀，k 遍历前缀
            j,k=j+1,k+1
            next[j]=k
        else:
            k=next[k]                    #k 置为 next[k]
def StrCount2(s,t):
    i,j=0,0
    cnt=0
    next=[0]*MaxSize
    GetNext(t,next)                  # 求 next 数组
    while i<len(s) and j<len(t):
        if j==-1 or s[i]==t[j]:
            i,j=i+1,j+1
        else:
            j=next[j]
        if j>=len(t):                # 找到一个子串
            cnt+=1                   # 累加出现的次数
            j=0
    return cnt

# 主程序
print()
print(" 测试 1")
s="aaaab"
t="aa"
print("   s: "+s+" t:"+t)
print("    BF: t 在 s 中出现次数 =%d" %(StrCount1(s,t)))
print("   KMP: t 在 s 中出现次数 =%d" %(StrCount2(s,t)))
print(" 测试 2")
s="abcabcdabcdeabcde"
t="abcd"
print("   s: "+s+" t:"+t)
print("    BF: t 在 s 中出现次数 =%d" %(StrCount1(s,t)))
print("   KMP: t 在 s 中出现次数 =%d" %(StrCount2(s,t)))
print(" 测试 3")
s="abcABCDabc"
t="abcd"
print("   s: "+s+" t:"+t)
print("    BF: t 在 s 中出现次数 =%d" %(StrCount1(s,t)))
print("   KMP: t 在 s 中出现次数 =%d" %(StrCount2(s,t)))
```

3. 解：由于在判断子串时是大小写无关的，所以将 s 和 t 中两个字符是否相同的条件改为 s[i].lower()==t[j].lower()，用 cnt 累计 t 在串 s 中不重叠出现的次数（初始值为 0）。采用两种解法。

① 基于 BF 算法：在找到子串后不是退出，而是 cnt 增加 1，i 增加 t 的长度并继续查找，直到整个字符串查找完毕。

② 基于 KMP 算法：当匹配成功时，cnt 增加 1，并且置 j 为 0 重新开始比较。

```
MaxSize=100
def StrCount1(s,t):                    #BF 算法求解
    i,cnt=0,0
    while i<len(s)-len(t)+1:
        j,k=i,0
        while j<len(s) and k<len(t) and s[j].lower()==t[k].lower():
            j,k=j+1,k+1
        if k==len(t):                          # 找到一个子串
            cnt+=1                             # 累加出现的次数
            i=j                                #i 从 j 开始
        else: i+=1                             #i 增加 1
    return cnt

def GetNext(t,next):                           # 由模式串 t 求出 next 值
    j,k=0,-1
    next[0]=-1
    while j<len(t)-1:
        if k==-1 or t[j].lower()==t[k].lower():
            j,k=j+1,k+1
            next[j]=k
        else:
            k=next[k]                          #k 置为 next[k]

def StrCount2(s,t):                            #KMP 算法
    i,j=0,0
    cnt=0
    next=[0]*MaxSize
    GetNext(t,next)                            # 求 next 数组
    while i<len(s) and j<len(t):
        if j==-1 or s[i].lower()==t[j].lower():
            i,j=i+1,j+1
        else:
            j=next[j]
        if j>=len(t):                          # 找到一个子串
            cnt+=1                             # 累加出现的次数
            j=0
    return cnt
```

```
# 主程序
print()
print(" 测试 1")
s="aAbAabaab"
t="aab"
print("   s: "+s+" t:"+t)
print("    BF: t 在 s 中出现次数 =%d" %(StrCount1(s,t)))
print("   KMP: t 在 s 中出现次数 =%d" %(StrCount2(s,t)))
print(" 测试 2")
s="abcabcdabcdeabcde"
t="ABCD"
print("   s: "+s+" t:"+t)
print("    BF: t 在 s 中出现次数 =%d" %(StrCount1(s,t)))
print("   KMP: t 在 s 中出现次数 =%d" %(StrCount2(s,t)))
print(" 测试 3")
s="abcABCDabc"
t="abcd"
print("   s: "+s+" t:"+t)
print("    BF: t 在 s 中出现次数 =%d" %(StrCount1(s,t)))
print("   KMP: t 在 s 中出现次数 =%d" %(StrCount2(s,t)))
```

4. 解：对于二维数组 $a[m][n]$，先求出每行的最小值元素放入 min 数组中，再求出每列的最大值元素放入 max 数组中。若 $\min[i]=\max[j]$，则该元素 $a[i][j]$ 便是马鞍点，找出所有这样的元素并输出。

```
def MinMax(a):
    m=len(a)                          # 行数
    n=len(a[0])                       # 列数
    min=[0]*m
    max=[0]*n
    for i in range(m):                # 计算每行最小元素，放入 min[i] 中
        min[i]=a[i][0]
        for j in range(1,n):
            if a[i][j]<min[i]:
                min[i]=a[i][j]
    for j in range(n):                # 计算每列最大元素，放入 max[j] 中
        max[j]=a[0][j]
        for i in range(1,m):
            if a[i][j]>max[j]:
                max[j]=a[i][j]
    res=[]                            # 存放马鞍点
    for i in range(m):                # 判定是否为马鞍点
        for j in range(n):
            if min[i]==max[j]:        # 找到一个马鞍点
                res.append([i,j,a[i][j]])
    return res
```

```
def disp(a):                            # 输出二维数组
    for i in range(len(a)):
        for j in range(len(a[i])):
            print("%4d" %(a[i][j]),end=' ')
        print()

# 主程序
a=[[1,3,2,4],[15,10,1,3],[4,5,3,6]]
print("\n a:"); disp(a)
print(" 所有马鞍点: ")
ans=MinMax(a)
for i in range(len(ans)):
    print("   (%d,%d): %d" %(ans[i][0],ans[i][1],ans[i][2]))
```

5. 解：设 A 为 n 阶对称矩阵，若其压缩数组 a 中有 m 个元素，则有 $n(n+1)/2=m$，即 $n^2+n-2m=0$，求得 $n=(\text{int})(-1+\text{sqrt}(1+8m))/2$。

A 的下三角或者主对角线 $A[i][j](i\geqslant j)$ 元素值存放在 $b[k]$ 中，则 $k=i(i+1)/2+j$，显然 $i(i+1)/2\leqslant k$，可以求出 $i\leqslant(-1+\text{sqrt}(1+8k))/2$，则 $i=\text{int}((-1+\text{sqrt}(1+8k))/2)$，$j=k-i(i+1)/2$。由此设计由 k 求出 i、j 下标的算法 getij(k)。

```
import math
def disp(A):                            # 输出二维数组 A
    for i in range(len(A)):
        for j in range(len(A[i])):
            print("%4d" %(A[i][j]),end=' ')
        print()
def compression(A,a):                   # 将 A 压缩存储到 a 中
    for i in range(len(A)):
        for j in range(i+1):
            k=i*(i+1)//2+j
            a[k]=A[i][j]
def getij(k):                       # 由 k 求出 i、j 下标
    ans=[0,0]
    i=int((-1+math.sqrt(1+8*k))/2)
    j=k-i*(i+1)//2
    ans[0],ans[1]=i,j
    return ans

def Restore(b,C):                   # 由 b 恢复成 C
    m=len(b)
    n=int((-1+math.sqrt(1+8*m))/2)
    for k in range(m):                          # 求主对角线和下三角部分元素
        ans=getij(k)
        i=ans[0]
        j=ans[1]
```

```
            C[i][j]=b[k]
        for i in range(n):       # 求上三角部分元素
            for j in range(i+1,n):
                C[i][j]=C[j][i]
# 主程序
print("\n ********* 测试 1**********")
n=3
A=[[1,2,3],[2,4,5],[3,5,6]]
C=[[None]*n for i in range(n)]
a=[0]*(n*(n+1)//2)
print(" A:"); disp(A)
print(" A 压缩得到 a")
compression(A,a)
print(" a:")
for i in range(len(a)):
    print(" "+str(a[i]),end=' ')
print()
print(" 由 a 恢复得到 C")
Restore(a,C)
print(" C:"); disp(C)
print("\n ********* 测试 2**********")
n=4
B=[[1,2,3,4],[2,5,6,7],[3,6,8,9],[4,7,9,10]]
D=[[None]*n for i in range(n)]
b=[0]*(n*(n+1)//2)
print(" B:"); disp(B)
print(" B 压缩得到 b")
compression(B,b)
print(" b:")
for i in range(len(a)):
    print(" "+str(a[i]),end=' ')
print()
print(" 由 b 恢复得到 D")
Restore(b,D)
print(" D:"); disp(D)
```

第五章 上机

练习题

1. 求楼梯走法数问题。一个楼梯有 n 个台阶，上楼可以一步上一个台阶，也可以一步上两个台阶。编写一个实验程序，求上楼梯共有多少种不同的走法，并用相关数据进行测试。

2. 假设 L 是一个带头结点的非空单链表，设计以下递归算法：

(1) 逆置单链表 L。

(2) 求结点值为 x 的结点个数。

并用相关数据进行测试。

3. 输入一个正整数 $n(n>5)$，随机产生 n 个 1～99 的整数，采用递归算法求其中的最大整数和次大整数。

解题思路

1. 解：设 $f(n)$表示上 n 个台阶的楼梯的走法数，显然 $f(1)=1$，$f(2)=2$(一种走法是一步上一个台阶、走两步，另外一种走法是一步上两个台阶)。

对于大于 2 的 n 个台阶的楼梯，一种走法是第一步上一个台阶，剩余 $n-1$ 个台阶的走法数是 $f(n-1)$；另外一种走法是第一步上两个台阶，剩余 $n-2$ 个台阶的走法数是 $f(n-2)$，所以有 $f(n)=f(n-1)+f(n-2)$。

```
MAXN=100
dp=[0]*MAXN
def solve1(n):              # 解法 1
    if n==1: return 1
    if n==2: return 2
    return solve1(n-1)+solve1(n-2)
def solve2(n):              # 解法 2
    if dp[n]!=0:
        return dp[n]
    if n==1:
        dp[1]=1
        return dp[1]
    if n==2:
        dp[2]=2
        return dp[2]
    dp[n]=solve2(n-1)+solve2(n-2)
    return dp[n]
def solve3(n):              # 解法 3
    dp[1]=1
    dp[2]=2
    for i in range(3,n+1):
        dp[i]=dp[i-1]+dp[i-2]
    return dp[n]
```

```
def solve4(n):              # 解法 4
    a=1                     # 对应 f(n-2)
    b=2                     # 对应 f(n-1)
    c=0                     # 对应 f(n)
    if n==1: return 1
    if n==2: return 2
    for i in range(3,n+1):
        c=a+b
        a=b
        b=c
    return c

# 主程序
n=10
print("\n n=%d" %(n))
print(" 解法 1: %d" %(solve1(n)))
print(" 解法 2: %d" %(solve2(n)))
print(" 解法 3: %d" %(solve3(n)))
print(" 解法 4: %d" %(solve4(n)))
```

2. 解：(1) 设 $f(t,h)$用于逆置以结点 t 为首结点(看成是不带头结点的单链表 t)的单链表，并且返回逆置后的单链表的首结点 h，这是大问题，小问题 $f(t.\text{next},h)$用于逆置以结点 $t.\text{next}$ 为首结点的单链表，并且返回该逆置后的单链表的首结点 h。对应的递归模型如下：

$$
\begin{aligned}
&f(t,h) \equiv h=t && \text{当单链表 } t \text{ 只有一个结点时}\\
&f(t,h) \equiv h=f(t.\text{next},h); && \text{其他情况}\\
&\qquad \text{将结点 } t \text{ 作为尾结点 } t.\text{next} \text{ 的后继结点}\\
&\qquad \text{将结点 } t \text{ 作为逆置后单链表的尾结点}
\end{aligned}
$$

(2) 设 $f(t,x)$用于求以结点 t 为首结点的单链表中值为 x 的结点个数。对应的递归模型如下：

$$
\begin{aligned}
&f(t,x)=0 && \text{当单链表 } t \text{ 为空时}\\
&f(t,h)=1+f(t.\text{next},x) && \text{当 } t.\text{data}=x \text{ 时}\\
&f(t,h)=1+f(t.\text{next},x) && \text{当 } t.\text{data}\neq x \text{ 时}
\end{aligned}
$$

```
from LinkList import LinkList,LinkNode
def Reverse(L):                 # 算法 (1)
    L.head.next=Reverse1(L.head.next,L.head.next)
    return L
def Reverse1(t,h):
    if t.next==None:            # 以 t 为首结点的单链表只有一个结点
        h=t
        return h
    else:
        h=Reverse1(t.next,h)    # 逆置 t.next 单链表
        t.next.next=t           # 将 t 结点作为尾结点
```

```
        t.next=None              # 尾结点 next 置为空
        return h
def Countx(L,x):                   # 算法 (2)
    return Countx1(L.head.next,x)

def Countx1(p,x):
    if p==None:
        return 0
    if p.data==x:
        return 1+Countx1(p.next,x)
    else:
        return Countx1(p.next,x)

# 主程序
a=[1,2,3,2,2]
L=LinkList()
L.CreateListR(a)
print()
print("  L: ",end=''),L.display()
print("  递归逆置")
L=Reverse(L)
print("  L: ",end=''),L.display()
x=2
print("  值为%d的结点个数: %d" %(x,Countx(L,x)))
```

3. 解：设递归函数 $f(a,low,high)$返回[max1,max2]，其中 max1 为最大整数，max2 为次大整数。对应的递归模型如下：

$f(a,low,high)=[a[low],-1]$　　当 a[low..high]仅含一个整数时

$f(a,low,high)=[max1,max2]$　　当 a[low..high]仅含两个整数时

　　max1=max(a[low],a[high])

　　max2=min(a[low],a[high])

$f(a,low,high)=[max1,max2]$　　其他情况

　　mid=(low+high)/2

　　lres=$f(a,low,mid)$

　　rres=$f(a,mid+1,high)$

　　max1 为 lres 中的最大整数，max2 为次大整数

```
import random
def solve():
    print()
    n=int(input("  n: "))
    a=random.sample(range(0,100),n)
    print("  整数序列:",a)
    res=Max2(a,0,n-1)
    print("  最大整数: %d, 次大整数: %d" %(res[0],res[1]))
    print()
```

```
def Max2(a,low,high):
    if low==high:
        return [a[low],-1]
    if low+1==high:
        max1=max(a[low],a[high])
        max2=min(a[low],a[high])
        return [max1,max2]
    mid=(low+high)//2
    lres=Max2(a,low,mid)
    rres=Max2(a,mid+1,high)
    if lres[0]>rres[0]:
        max1=lres[0]
        max2=max(lres[1],rres[0])
    else:
        max1=rres[0]
        max2=max(rres[1],lres[0])
    return [max1,max2]


# 主程序
solve()
```