# 第3章 基本算法设计方法

## 3.1 单项选择题及其参考答案

**1.** 穷举法的适用范围是_____。

A. 一切问题　　B. 解的个数极多的问题

C. 解的个数有限且可一一列举　　D. 不适合设计算法

**答**：穷举法作为一种算法设计基本方法并非万能的，主要适合于解个数有限且可一一列举的问题的求解。答案为C。

**2.** 如果一个4位数恰好等于它的各位数字的4次方和，则这个4位数称为玫瑰花数。例如$1634=1^4+6^4+3^4+4^4$，则1634是一个玫瑰花数。若想求出4位数中所有的玫瑰花数，可以采用的问题解决方法是_____。

A. 递归法　　B. 穷举法

C. 归纳法　　D. 都不适合

**答**：4位数的范围是1000～9999，可以一一枚举。答案为B。

**3.** 有一个数列，递推关系是$a_1=\frac{1}{2}$，$a_{n+1}=\frac{a_n}{a_n+1}$，则求出的通项公式是_____。

A. $a_n=\frac{1}{n+1}$　　B. $a_n=\frac{1}{n}$

C. $a_n=\frac{1}{2n}$　　D. $a_n=\frac{n}{2}$

**答**：采用不完全归纳法，$a_1=\frac{1}{2}$，$a_2=\frac{1}{3}$，$a_3=\frac{1}{4}$，$a_4=\frac{1}{5}$，…，$a_n=\frac{1}{n+1}$，可以采用数学归纳法证明这个结论的正确性。答案为A。

**4.** 猜想 $1=1,1-4=-(1+2),1-4+9=1+2+3,\cdots$的第 5 个式子是_____。

A. $1^2+2^2-3^2-4^2+5^2=1+2+3+4+5$

B. $1^2+2^2-3^2+4^2-5^2=-(1+2+3+4+5)$

C. $1^2-2^2+3^2-4^2+5^2=-(1+2+3+4+5)$

D. $1^2-2^2+3^2-4^2+5^2=1+2+3+4+5$

**答**：推导如下。

$n=1,1^2=1$

$n=2,1-4=1-2^2=-(1+2)$

$n=3,1-4+9=1^2-2^2+3^2=1+2+3$

$n=4,1^2-2^2+3^2-4^2=-(1+2+3+4)$

$n=5,1^2-2^2+3^2-4^2+5^2=1+2+3+4+5$

答案为 D。

**5.** 对于迭代法，下面的说法不正确的是_____。

A. 需要确定迭代模型

B. 需要建立迭代关系式

C. 需要对迭代过程进行控制，要考虑什么时候结束迭代过程

D. 不需要对迭代过程进行控制

**答**：迭代法先确定迭代变量，再对迭代过程进行控制。答案为 D。

**6.** 设计递归算法的关键是_____。

A. 划分子问题　　B. 提取递归模型

C. 合并子问题　　D. 求解递归出口

**答**：递归模型是递归算法的核心，反映递归算法的本质。答案为 B。

**7.** 若一个问题的求解既可以用递归算法，也可以用迭代算法，则往往用 ① 算法，因为 ② 。

① A. 先递归后迭代　　B. 先迭代后递归

C. 递归　　D. 迭代

② A. 迭代的效率比递归高　　B. 递归宜于问题分解

C. 递归的效率比迭代高　　D. 迭代宜于问题分解

**答**：一般情况下求解同一个问题的迭代算法比递归算法的效率高。答案是①D，②A。

**8.** 递归函数 $f(n)=f(n-1)+n(n>1)$的递归出口是_____。

A. $f(-1)=0$　　B. $f(1)=1$

C. $f(0)=1$　　D. $f(n)=n$

**答**：$f(n)=f(n-1)+n(n>1)$是递归体，递归出口对应 $n=1$ 的情况。答案为 B。

**9.** 递归函数 $f(n)=f(n-1)+n(n>1)$的递归体是_____。

A. $f(-1)=0$　　B. $f(1)=1$

C. $f(n)=n$　　D. $f(n)=f(n-1)+n$

**答**：$f(n)=f(n-1)+n(n>1)$本身就是递归体。答案为 D。

**10.** 有以下递归算法，$f(123)$的输出结果是_____。

```
void f(int n)
{   if(n>0)
    {   printf("%d",n%10);
        f(n/10);
    }
}
```

A. 321　　B. 123

C. 6　　D. 以上都不对

**答**：该算法属于先合后递的递归算法。在执行 $f(123)$时先输出 $123\%10=3$，再调用 $f(12)$输出 2，最后调用 $f(1)$输出 1。答案为 A。

**11.** 有以下递归算法，$f(123)$的输出结果是_____。

```
void f(int n)
{   if(n>0)
    {   f(n/10);
        printf("%d",n%10);
    }
}
```

A. 321　　B. 123

C. 6　　D. 以上都不对

**答**：该算法属于先递后合的递归算法。执行过程是 $f(123)\rightarrow f(12)\rightarrow f(1)$，返回时依次输出 1，2 和 3。答案为 B。

**12.** 整数单链表 $h$ 是不带头结点的，结点类型 ListNode 为(val,next)，则以下递归算法中隐含的递归出口是_____。

```
void f(ListNode * h)
{   if(h!=NULL)
    {   printf("%d ",h->val);
        f(h->next);
    }
}
```

A. if(h!=NULL) return;　　B. if(h==NULL) return 0;

C. if(h==NULL) return;　　D. 没有递归出口

**答**：任何递归算法都包含递归出口，该递归算法是正向输出单链表 $h$ 中的结点值，递归出口是 $h$ 为空的情况。答案为 C。

**13.** $T(n)$表示输入规模为 $n$ 时的算法效率，以下算法中性能最优的是_____。

A. $T(n)=T(n-1)+1, T(1)=1$　　B. $T(n)=2n^2$

C. $T(n)=T(n/2)+1, T(1)=1$　　D. $T(n)=3n\log_2 n$

**答**：对于选项 A，采用直接展开法求出 $T(n)=\Theta(n)$。对于选项 B，$T(n)=\Theta(n^2)$。对于选项 C，采用主方法，$a=1,b=2,f(n)=O(1),n^{\log_b a}=1$ 与 $f(n)$ 的阶相同，则 $T(n)=\Theta(\log_2 n)$。对于选项 D，$T(n)=\Theta(n\log_2 n)$。答案为 C。

## 3.2 问答题及其参考答案

**1.** 采用穷举法解题时的常用列举方法有顺序列举、排列列举和组合列举，问求解以下问题应该采用哪一种列举方法？

（1）求 $m\sim n$ 的所有素数。

（2）在数组 $a$ 中选择出若干元素，它们的和恰好等于 $k$。

（3）有 $n$ 个人合作完成一个任务，他们采用不同的排列顺序，则完成该任务的时间不同，求最优完成时间。

**答**：（1）采用顺序列举。

（2）采用组合列举。

（3）采用排列列举。

**2.** 许多系统用户登录时需要输入密码，为什么还需要输入已知的验证码？

**答**：一般密码长度有限，密码由数字和字母等组成，可以采用穷举法枚举所有可能的密码，对每个密码进行试探。如果加入验证码，就会延迟每次试探的时间，从而使得这样破解密码变成几乎不可能。

**3.** 什么是递归算法？递归模型由哪两个部分组成？

**答**：递归算法是指直接或间接地调用自身的算法。递归模型由递归出口和递归体两个部分组成。

**4.** 比较迭代算法与递归算法的异同。

**答**：迭代算法与递归算法的相同点是都是解决“重复操作”的机制，不同点是递归算法往往比迭代算法耗费更多的时间（调用和返回均需要额外的时间）与存储空间（用来保存不同次调用情况下变量的当前值的栈空间），每个迭代算法原则上总可以转换成与它等价的递归算法，反之不然。

**5.** 有一个含 $n(n>1)$ 个整数的数组 $a$，写出求其中最小元素的递归定义。

**答**：设 $f(a,i)$ 表示 $a[0..i]$（共 $i+1$ 个元素）中的最小元素，为大问题，则 $f(a,i-1)$ 表示 $a[0..i-1]$（共 $i$ 个元素）中的最小元素，为小问题。对应的递归定义如下：

$f(a,i)=a[0]$ 当 $i=0$ 时

$f(a,i)=\min(f(a,i-1),a[i])$ 其他

则 $f(a,n-1)$ 求数组 $a$ 中 $n$ 个元素的最小元素。

**6.** 有一个含 $n(n>1)$ 个整数的数组 $a$，写出求所有元素和的递归定义。

**答：** 设 $f(a,i)$ 表示 $a[0..i]$（共 $i+1$ 个元素）中所有元素的和，为大问题，则 $f(a,i-1)$ 表示 $a[0..i-1]$（共 $i$ 个元素）中所有元素的和，为小问题。对应的递归定义如下：

$f(a,i)=a[0]$　　当 $i=0$ 时

$f(a,i)=f(a,i-1)+a[i]$　　其他

则 $f(a,n-1)$ 求数组 $a$ 中 $n$ 个元素的和。

**7.** 利用整数的后继函数 succ 写出 $x+y$（$x$ 和 $y$ 都是正整数）的递归定义。

**答：** 设 $f(x,y)=x+y$，对应的递归定义如下。

$f(x,y)=y$　　当 $x=0$ 时

$f(x,y)=x$　　当 $y=0$ 时

$f(x,y)=f(\text{succ}(x),\text{succ}(y))+2$　　其他

**8.** 有以下递归算法，则 $f(f(7))$ 的结果是多少？

```
int f(int n)
{   if(n<=3)
        return 1;
    else
        return f(n-2)+f(n-4)+1;
}
```

**答：** 先求 $f(7)$，如图 3.1(a)所示，求出 $f(7)=5$ 后，再求 $f(5)$，如图 3.1(b)所示，求出 $f(5)=3$ 后，所以 $f(f(7))$ 的结果是 3。

**9.** 有以下递归算法，则 $f(3,5)$ 的结果是多少？

```
int f(int x,int y)
{   if(x<=0 || y<=0)
        return 1;
    else
        return 3*f(x-1,y/2);
}
```

**答：** 求 $f(3,5)$ 的过程如图 3.2 所示，求出 $f(3,5)=27$。

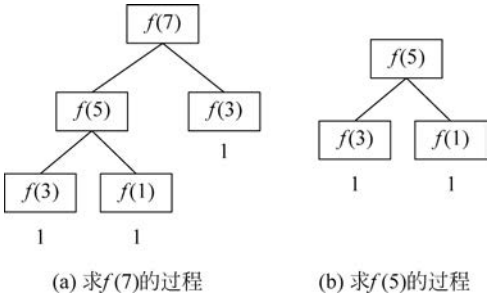


图 3.1　求 $f(f(7))$ 的过程

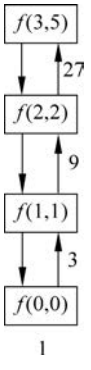


图 3.2　求 $f(3,5)$ 的过程

**10.** 采用直接展开法求以下递推式：

$T(1)=1$

$T(n)=T(n-1)+n$　　当 $n>1$ 时

**答**：求 $T(n)$的过程如下。

$$\begin{aligned}T(n)&=T(n-1)+n=[T(n-2)+(n-1)]+n=T(n-2)+n+(n-1)\\&=T(n-3)+n+(n-1)+(n-2)\\&=\cdots\\&=T(1)+n+(n-1)+\cdots+2\\&=n+(n-1)+\cdots+2+1=n(n+1)/2=\Theta(n^2)\end{aligned}$$

**11.** 采用递归树方法求解以下递推式：

$T(1)=1$

$T(n)=4T(n/2)+n$　　当 $n>1$ 时

**解**：构造的递归树如图 3.3 所示，第 1 层的问题规模为 $n$，第 2 层的子问题的问题规模为 $n/2$，以此类推，当展开到第 $k+1$ 层时，其规模为 $n/2^k=1$，所以递归树的高度为 $\log_2 n+1$。

第 1 层有一个结点，其时间为 $n$，第 2 层有 4 个结点，其时间为 $4(n/2)=2n$，以此类推，第 $k$ 层有 $4^{k-1}$ 个结点，每个子问题的规模为 $n/2^{k-1}$，其时间为 $4^{k-1}(n/2^{k-1})=2^{k-1}n$。叶子结点的个数为 $n$ 个，其时间为 $n$。将递归树每一层的时间加起来，可得：

$T(n)=n+2n+\cdots+2^{k-1}n+\cdots+n\approx n\times 2^{\log_2 n}=\Theta(n^2)$。

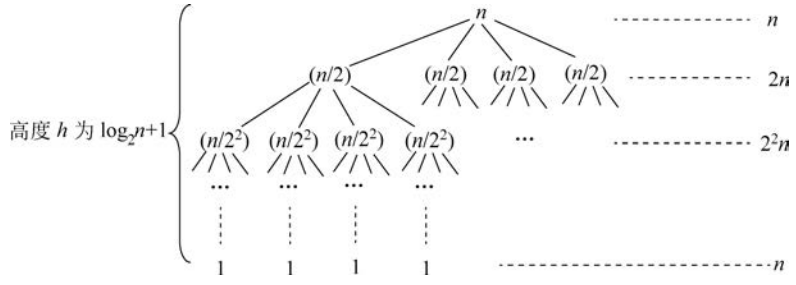


图 3.3　一棵递归树

**12.** 采用主方法求解以下递推式：

(1) $T(n)=4T(n/2)+n$

(2) $T(n)=4T(n/2)+n^2$

(3) $T(n)=4T(n/2)+n^3$

**答**：(1) 这里 $a=4,b=2,f(n)=n$。$n^{\log_b a}=n^{\log_2 4}=n^2$，显然 $f(n)$的阶小于 $n^{\log_b a}$，满足主方法中的情况①，所以 $T(n)=\Theta(n^{\log_b a})=\Theta(n^2)$。

(2) 这里 $a=4,b=2,f(n)=n^2$。$n^{\log_b a}=n^{\log_2 4}=n^2$，显然 $f(n)$与 $n^{\log_b a}$ 同阶，满足主方法中的情况②，$T(n)=\Theta(n^{\log_b a}\log_2 n)=\Theta(n^2\log_2 n)$。

(3) 这里 $a=4,b=2,f(n)=n^3$。$n^{\log_b a}=n^{\log_2 4}=n^2$，显然 $f(n)$的阶大于 $n^{\log_b a}$。另外，对于足够大的 $n$，$af(n/b)=4\times n^3/8=n^3/2\leqslant cf(n)$，这里 $c=1/2$，满足正规性条件，则有 $T(n)=\Theta(f(n))=\Theta(n^3)$。

**13.** 有以下算法，分析其时间复杂度。

```
void f(int n)
{   for(int i=1;i<=n;i++)
    {   for(int j=1;j<=i;j++)
            printf("%d %d %d\n",i,j,n);
    }
    if(n>0)
    {   for(int i=1;i<=4;i++)
            f(n/2);
    }
}
```

**答**：算法中两重 for 的执行次数 $=\sum_{i=1}^{n}\sum_{j=1}^{i}1=\sum_{i=1}^{n}i=\frac{n(n+1)}{2}=\Theta(n^2)$。对应的时间递推式如下：

$T(n)=1$　　当 $n=0$ 时

$T(n)=4T(n/2)+n^2$　　其他

采用主方法，$a=4,b=2,f(n)=n^2,n^{\log_b a}=n^2$，与 $f(n)$ 的阶相同，则 $T(n)=\Theta(n^2\log_2 n)$。

**14.** 分析《教程》3.4.3 节中求 1～$n$ 的全排列的递归算法 perm21($n$,$n$)的时间复杂度。

**答**：perm21($n$,$n$)用于求 1～$n$ 的全排列 $P_n$，设其执行时间为 $T(n)$，它首先调用 perm2($n$,$n-1$)求出 1～$n-1$ 的全排列 $P_{n-1}$，该小问题的执行时间为 $T(n-1)$，再对于 $P_{n-1}$ 中每个集合元素(共$(n-1)!$个集合元素)的每个位置(共 $n$ 个位置)插入 $n$，合并结果得到 $P_n$，执行次数为 $n(n-1)!=n!$。所以递推式如下：

$T(1)=1$　　求 $P_1$ 为常量

$T(n)=T(n-1)+n!$　　当 $n>1$ 时

令 $T(n)=n!g(n)$(因为 1～$n$ 全排列中的排列个数为 $n!$)，则

$$T(n-1)=(n-1)!g(n-1)$$

代入 $T(n)=T(n-1)+n!$ 中得到：

$$n!g(n)=(n-1)!g(n-1)+n!$$

两边乘以 $n$：

$$nn!g(n)=n(n-1)!g(n-1)+nn!=n!g(n-1)+nn!$$

两边除以 $n!$：

$$ng(n)=g(n-1)+n$$

得到如下递推式：

$g(1)=1$

$g(n)=g(n-1)/n+1$　　当 $n>1$ 时

采用直接展开法，$g(n)=g(n-1)/n+1=g(n-2)/(n(n-1))+1/n+1=\cdots\leqslant 2$。

$T(n)=n!g(n)\leqslant 2n!$

因此有 $T(n)=O(n!)$。

**15***. 有以下多项式：

$$f(x,n)=x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots+(-1)^n\frac{x^{2n+1}}{(2n+1)!}$$

给出求 $f(x,n)$值的递推式，分析其求解的时间复杂度。

**答**：为了简单，省略 $x$ 参数。

$f(1)=x$，　$g(1)=x$

$$f(2)=x-\frac{x^3}{3!}=f(1)+(-1)\times g(1)\times\frac{x^2}{3\times 2},\quad g(2)=(-1)\times g(1)\times\frac{x^2}{2\times 3}$$

$$f(3)=x-\frac{x^3}{3!}+\frac{x^5}{5!}=f(2)+(-1)\times g(2)\times\frac{x^2}{4\times 5},\quad g(3)=(-1)\times g(2)\times\frac{x^2}{4\times 5}$$

可以推出求 $f(n)$的递推式如下：

$$f(1)=x,\quad g(1)=x$$

$$g(n)=(-1)\times g(n-1)\times\frac{x^2}{(2n-2)(2n-1)}$$

$$f(n)=f(n-1)+g(n)$$

设求 $f(n)$的执行时间为 $T(n)$，求 $f(n)$需要求出 $g(n)$，但它们是同时计算的，也就是说 $T(n)$表示的是求 $f(n)$和 $g(n)$的时间，对应的执行时间递推式如下：

$T(1)=O(1)$

$T(n)=T(n-1)+O(1)$　　当 $n>1$ 时

可以推出 $T(n)=O(n)$。

## 3.3 算法设计题及其参考答案

**1**. 有 3 种硬币若干个，面值分别是 1 分、2 分、5 分，如果要凑够 1 毛 5，设计一个算法求有哪些组合方式，共多少种组合方式。

**解**：采用穷举法，设所需 1 分、2 分和 5 分硬币的个数分别为 $i$、$j$ 和 $k$，显然有 $0\leqslant i\leqslant 15$，$0\leqslant j\leqslant 7$，$0\leqslant k\leqslant 3$，约束条件为 $i+2j+5k=15$，用 cnt 表示组合方式数。对应的算法如下：

```
int solve()
{   int cnt=0;
    for(int i=0;i<=15;i++)
    {   for(int j=0;j<=7;j++)
        {   for(int k=0;k<=3;k++)
            {   if (i+(2*j)+(5*k)==15)
                {   printf("一分硬币%d个,两分硬币%d个,五分硬币%d个\n",i,j,k);
                    cnt++;
                }
            }
        }
    }
    return cnt;
}
```

**2.** 有一个整数序列是0,5,6,12,19,32,52,…,其中第1项为0,第2项为5,第3项为6,以此类推,采用迭代算法和递归算法求该数列的第 $n(n\geqslant 1)$ 项。

**解**:设 $f(n)$ 为数列的第 $n$ 项,则

$f(1)=0$

$f(2)=5$

$f(3)=6=f(1)+f(2)+1$

$f(4)=12=f(2)+f(3)+1$

…

可以归纳出当 $n>2$ 时有 $f(n)=f(n-2)+f(n-1)+1$。对应的迭代算法如下:

```
int sequence1(int n)                    //迭代算法
{   int a=0,b=5,c;
    if(n==1)
        return a;
    else if(n==2)
        return b;
    else
    {   for(int i=3;i<=n;i++)
        {   c=a+b+1;
            a=b;
            b=c;
        }
        return c;
    }
}
```

对应的递归算法如下:

```
int sequence2(int n)                    //递归算法
{   if(n==1)
        return 0;
    else if(n==2)
        return 5;
    else
        return sequence2(n-2)+sequence2(n-1)+1;
}
```

**3.** 给定一个正整数 $n(1\leqslant n\leqslant 100)$,采用迭代算法和递归算法求 $s=1+(1+2)+(1+2+3)+\cdots+(1+2+\cdots+n)$。

**解**:设 curs=1+2+…+($i$-1),则 curs+$i$ 便是 1+2+…+$i$,用 ans 累加所有的 curs(初始为0)。对应的迭代算法如下:

```
int Sum1(int n)                         //迭代算法
{   int ans=0;
    int curs=0;
    for(int i=1;i<=n;i++)
    {   curs+=i;
        ans+=curs;
    }
    return ans;
}
```

设 $f(n)=1+(1+2)+(1+2+3)+\cdots+(1+2+\cdots+n)$,则 $f(n-1)=1+(1+2)+$

(1+2+3)+…+(1+2+…+$n$-1),两式相减得到 $f(n)-f(n-1)=(1+2+\cdots+n)=n(n+1)/2$,则递归模型如下:

$f(1)=1$

$f(n)=f(n-1)+n(n+1)/2$　　当 $n>1$ 时

对应的递归算法如下:

```
int Sum2(int n)                          //递归算法
{   if(n==1)
        return 1;
    else
        return Sum2(n-1)+n*(n+1)/2;
}
```

**4.** 一个数列的首项 $a_1=0$,后续奇数项和偶数项的计算公式分别为 $a_{2n}=a_{2n-1}+2$,$a_{2n+1}=a_{2n-1}+a_{2n}-1$,设计一个递归算法求数列的第 $n$ 项。

**解**:设 $f(m)$计算数列的第 $m$ 项。当 $m$ 为偶数时,不妨设 $m=2n$,则 $2n-1=m-1$,所以有 $f(m)=f(m-1)+2$;当 $m$ 为奇数时,不妨设 $m=2n+1$,则 $2n-1=m-2$,$2n=m-1$,所以有 $f(m)=f(m-2)+f(m-1)-1$。对应的递归算法如下:

```
int sequence(int m)                      //递归算法
{   if (m==1)
        return 0;
    else if (m%2==0)
        return sequence(m-1)+2;
    else
        return sequence(m-2)+sequence(m-1)-1;
}
```

**5.** 设计一个递归算法用于翻转一个非空字符串 $s$。

**解**:设 $f(\text{str},i)$返回 $s[i..n-1]$(共 $n-i$ 个字符)的翻转字符串,为大问题,$f(\text{str},i+1)$返回 $s[i..n-1]$(共 $n-i-1$ 个字符)的翻转字符串,为小问题,$i\geqslant n$ 时空串的翻转结果是空串。对应的递归模型如下:

$f(s,i)=$""　　当 $i\geqslant n$ 时

$f(s,i)=f(s,i+1)+\text{s}[i]$　　其他情况

对应的递归算法如下:

```
string reverse1(string s,int i)          //被 reverse 算法调用
{   if (i>=s.size())
        return "";
    else
        return reverse1(s,i+1)+s[i];
}
string reverse(string s)                 //递归算法
{
    return reverse1(s,0);
}
```

**6.** 对于不带头结点的非空整数单链表 $h$,设计一个递归算法求其中值为 $x$ 的结点的个数。

**解**:设 $f(h,x)$返回单链表 $h$ 中值为 $x$ 的结点的个数,为大问题,$f(h->\text{next},x)$返回

子单链表 $h->next$ 中值为 $x$ 的结点的个数，为小问题，空单链表的结点个数为 0。对应的递归模型如下：

$f(h,x)=0$　　当 $h=$NULL 时

$f(h,x)=f(h->next,x)+1$　　当 $h->val=x$ 时

$f(h,x)=f(h->next,x)$　　其他

对应的递归算法如下：

```
int Count(ListNode * h,int x)
{   if(h==NULL)
        return 0;
    else if(h->val==x)
        return Count(h->next,x)+1;
    else
        return Count(h->next,x);
}
```

**7.** 对于不带头结点的非空单链表 $h$，设计一个递归算法删除其中第一个值为 $x$ 的结点。

**解：** 设 $f(h,x)$删除单链表 $h$ 中第一个值为 $x$ 的结点，为大问题，$f(h->next,x)$删除子单链表 $h->next$ 中第一个值为 $x$ 的结点，为小问题。对应的递归模型如下：

$f(h,x)\equiv$ 不做任何事情　　当 $h=$NULL 时

$f(h,x)\equiv$ 删除 $h$ 结点，$h=h->next$　　当 $h->val=x$ 时

$f(h,x)\equiv f(h->next,x)$　　其他

对应的递归算法如下：

```
void Delfirstx(ListNode * &h,int x)          //递归算法:删除单链表 h 中第一个值为 x 的结点
{   if (h==NULL) return;
    if (h->val==x)
    {   ListNode *p=h;
        h=h->next;
        free(p);
    }
    else
        Delfirstx(h->next,x);
}
```

**8.** 对于不带头结点的非空单链表 $h$，设计一个递归算法删除其中所有值为 $x$ 的结点。

**解：** 设 $f(h,x)$删除单链表 $h$ 中所有值为 $x$ 的结点，为大问题，$f(h->next,x)$删除子单链表 $h->next$ 中所有值为 $x$ 的结点，为小问题。对应的递归模型如下：

$f(h,x)\equiv$ 不做任何事情　　当 $h=$NULL 时

$f(h,x)\equiv$ 删除 $h$ 结点，$h=h->next$；$f(h,x)$　　当 $h->val=x$ 时

$f(h,x)\equiv f(h->next,x)$　　其他

对应的递归算法如下：

```
void Delallx(ListNode * &h,int x)            //递归算法:删除单链表 h 中所有值为 x 的结点
{   if (h==NULL) return;
    if (h->val==x)
    {   ListNode *p=h;
        h=h->next;
        free(p);
```

```
        Delallx(h,x);
    }
    else
        Delallx(h->next,x);
}
```

**9.** 假设二叉树采用二叉链存储结构存放，结点值为整数，设计一个递归算法求二叉树 $b$ 中所有叶子结点值的和。

**解**：设 $f(b)$返回二叉树 $b$ 中所有叶子结点值的和，为大问题，$f(b->\text{left})$和 $f(b->\text{right})$分别返回二叉树 $b$ 左、右子树中所有叶子结点值的和，为两个小问题。对应的递归模型如下：

$f(b)=0$　　当 $b=$NULL 时

$f(b)=b->\text{val}$　　当 $b$ 结点为叶子结点时

$f(b)=f(b->\text{left})+f(b->\text{right})$　　其他

对应的递归算法如下：

```
int LeafSum(TreeNode *b)            //递归算法:求二叉树 b 中所有叶子结点值的和
{   if (b==NULL) return 0;
    if (b->left==NULL && b->right==NULL)
        return b->val;
    int lsum=LeafSum(b->left);
    int rsum=LeafSum(b->right);
    return lsum+rsum;
}
```

**10.** 假设二叉树采用二叉链存储结构存放，结点值为整数，设计一个递归算法求二叉树 $b$ 中第 $k$(1≤$k$≤二叉树 $b$ 的高度)层所有结点值的和(根结点层次为 1)。

**解**：设 $f(b,h,k)$返回二叉树 $b$ 中第 $k$ 层所有结点值的和(初始时 $b$ 指向根结点，$h$ 置为 1 表示结点 $b$ 的层次)。其递归模型如下：

$f(b,h,k)=0$　　当 $b=$NULL 时

$f(b,h,k)=b->\text{val}$　　当 $h=k$ 时

$f(b,h,k)=f(b->\text{left},h+1,k)+f(b->\text{right},h+1,k)$　　当 $h<k$ 时

$f(b,h,k)=0$　　其他

对应的递归算法如下：

```
int LevelkSum1(TreeNode *b,int h,int k)      //被 LevelkSum 算法调用
{   if(b==NULL)
        return 0;
    if(h==k)
        return b->val;
    if(h<k)
    {   int lsum=LevelkSum1(b->left,h+1,k);
        int rsum=LevelkSum1(b->right,h+1,k);
        return lsum+rsum;
    }
    else
        return 0;
}
int LevelkSum(TreeNode *b,int k)             //递归算法:求二叉树 b 中第 k 层所有结点值的和
{
```

```
    return LevelkSum1(b,1,k);
}
```

**11.** 设计将十进制正整数 $n$ 转换为二进制数的迭代算法和递归算法。

**解**：设 $f(n)$为 $n$ 的二进制数。求出 $n\%2$ 和 $n/2$，$n\%2$ 作为结果二进制数的最高位，$f(n/2)$作为小问题。假设 $f(n/2)$已经求出，将 $n\%2$ 作为其最高位得到 $f(n)$的结果。对应的递归模型如下：

$f(n)$ = 空　　当 $n\leqslant 0$ 时

$f(n)$ = $n\%2 \oplus f(n/2)$　　当 $n>0$ 时

其中 $x\oplus y$ 表示 $x$ 作为 $y$ 的最高位。

采用数组存放转换的二进制数，每个元素表示一个二进制位，由于需要将 $n\%2$ 的二进制位插入最前面，所以改为用 deque<int>容器存放转换的二进制数。对应的迭代法算法如下：

```
deque<int> trans1(int n)                    //迭代算法
{   deque<int> ans;
    while(n>0)
    {   int d=n%2;                          //求出二进制位 d
        ans.push_front(d);                  //将 d 作为高位的元素
        n/=2;                               //新值取代旧值
    }
    return ans;
}
```

对应的递归算法如下：

```
deque<int> trans2(int n)                    //递归算法
{   if(n<=0) return {};
    deque<int> ans=trans2(n/2);             //先递后合
    int d=n%2;
    ans.push_back(d);
    return ans;
}
```

**12.** 在《教程》3.2.2 节中采用迭代算法实现直接插入排序，请设计等效的递归算法。

**解**：直接插入排序递归算法的设计思路参考《教程》3.2.2 节和 3.4.2 节。采用先递后合和先合后递两种递归算法如下：

```
void Insert(vector<int> &R,int i)           //将 R[i]有序插入 R[0..i-1]中
{   int tmp=R[i];
    int j=i-1;
    do                                      //找 R[i]的插入位置
    {   R[j+1]=R[j];                        //将大于 R[i]的元素后移
        j--;
    } while(j>=0 && R[j]>tmp);              //直到 R[j]<=tmp 为止
    R[j+1]=tmp;                             //在 j+1 处插入 R[i]
}
/*** 先递后合算法 ***************************/
void InsertSort21(vector<int> &R,int i)     //递归的直接插入排序
{   if(i==0) return;
    InsertSort21(R,i-1);
```

```
    if (R[i]<R[i-1])                    //反序时
        Insert(R,i);
}
void InsertSort2(vector<int> &R)        //递归算法:直接插入排序
{   int n=R.size();
    InsertSort21(R,n-1);
}
/*** 先合后递算法 ******************************/
void InsertSort31(vector<int> &R,int i)  //递归的直接插入排序
{   int n=R.size();
    if(i<1 || i>n-1) return;
    if (R[i]<R[i-1])                    //反序时
        Insert(R,i);
    InsertSort31(R,i+1);
}
void InsertSort3(vector<int> &R)        //递归算法:直接插入排序
{
    InsertSort31(R,1);
}
```

**13.** 在《教程》3.3.2 节中采用迭代算法实现简单选择排序,请设计等效的递归算法。

**解**:简单选择排序递归算法的设计思路参考《教程》3.3.2 节和 3.4.2 节。采用先递后合和先合后递两种递归算法如下:

```
void Select(vector<int> & R,int i)      //在 R[i..n-1]中选择最小元素交换到 R[i]位置
{   int minj=i;                         //minj 表示 R[i..n-1]中最小元素的下标
    for (int j=i+1;j<R.size();j++)      //在 R[i..n-1]中找最小元素
    {   if (R[j]<R[minj])
            minj=j;
    }
    if (minj!=i)                        //若最小元素不是 R[i]
        swap(R[minj],R[i]);             //交换
}
/*** 先递后合算法 ******************************/
void SelectSort21(vector<int> & R,int i)  //递归的简单选择排序
{   if (i==-1) return;                  //满足递归出口条件
    SelectSort21(R,i-1);
    Select(R,i);
}
void SelectSort2(vector<int> & R)       //递归的简单选择排序
{
    SelectSort21(R,R.size()-2);
}
/*** 先合后递算法 ******************************/
void SelectSort31(vector<int> & R,int i)  //递归的简单选择排序
{   int n=R.size();
    if (i==n-1) return;                 //满足递归出口条件
    Select(R,i);
    SelectSort31(R,i+1);
```

```
}
void SelectSort3(vector<int> & R)          //递归的简单选择排序
{
    SelectSort31(R,0);
}
```

**14.** 在《教程》3.4.2 节中采用递归算法实现冒泡排序，请设计等效的迭代算法。

**解：** 冒泡排序迭代算法的设计思路参考《教程》3.4.2 节和 3.3.2 节。对应的算法如下：

```
void Bubble(vector<int> & R,int i,bool& exchange)  //在 R[i..n-1]中冒泡最小元素到 R[i]位置
{   int n=R.size();
    for (int j=n-1;j>i;j--)                        //无序区元素比较,找出最小元素
    {   if (R[j-1]>R[j])                           //当相邻元素反序时
        {   swap(R[j],R[j-1]);                     //R[j]与 R[j-1]进行交换
            exchange=true;                         //本趟排序发生交换置 exchange 为 true
        }
    }
}
void BubbleSort1(vector<int> & R)                  //迭代算法:冒泡排序
{   int n=R.size();
    bool exchange;
    for (int i=0;i<n-1;i++)                        //进行 n-1 趟排序
    {   exchange=false;                            //本趟排序前置 exchange 为 false
        Bubble(R,i,exchange);
        if (exchange==false)                       //本趟未发生交换时结束算法
            return;
    }
}
```

**15.** 在《教程》3.3.4 节中采用迭代算法求 1～$n$ 的幂集，请设计等效的递归算法。

**解：** 用 $M_i$ 表示 1～$i$ 的幂集。对应的递归模型如下：

$M_1=\{\{\},\{1\}\}$

$M_i=M_{i-1}\bigcup A_i$　　当 $i>1$ 时

其中 $A_i$=appendi($M_{i-1}$,$i$)。幂集用 vector<vector<int>>容器存放，其中每个 vector<int>类型的元素表示幂集中的一个集合。大问题是求{1～$i$}的幂集，小问题是求{1～$i-1$}的幂集。采用先递后合和先合后递两种递归算法如下：

```
vector<vector<int>> appendi(vector<vector<int>> Mi_1,int i)
//向 Mi_1 中每个集合元素的末尾添加 i
{   vector<vector<int>> Ai=Mi_1;
    for(int j=0;j<Ai.size();j++)
        Ai[j].push_back(i);
    return Ai;
}
/*** 先递后合算法 ****************************/
vector<vector<int>> pset(int n,int i)              //递归算法
{   if(i==1)
        return {{},{1}};
    else
    {   vector<vector<int>> Mi_1=pset(n,i-1);      //递归求出 Mi_1
```

```
        vector<vector<int>> Mi=Mi_1;                    //Mi 置为 Mi_1
        vector<vector<int>> Ai=appendi(Mi_1,i);
        for(int j=0;j<Ai.size();j++)                    //将 Ai 中的所有集合元素添加到 Mi 中
            Mi.push_back(Ai[j]);
        return Mi;                                      //返回 Mi
    }
}
vector<vector<int>> subsets2(int n)                     //递归算法
{
    return pset(n,n);
}
/*** 先合后递算法 ****************************/
vector<vector<int>> pset(vector<vector<int>> M,int n,int i)   //递归算法
{   vector<vector<int>> A=appendi(M,i);                 //求 A
    for(int j=0;j<A.size();j++)                         //将 A 中的所有集合元素添加到 M 中
        M.push_back(A[j]);
    if(i==n)                                            //已经求出结果时返回 M
        return M;
    else                                                //否则递归调用
        return pset(M,n,i+1);
}
vector<vector<int>> subsets3(int n)                     //递归算法
{   vector<vector<int>> M={{},{1}};                     //M 存放{1-n}的幂集,初始时置为 M1
    if(n==1)
        return M;
    else
        return pset(M,n,2);
}
```

**16.** 在《教程》3.4.3 节中采用递归算法求 1～$n$ 的全排列，请设计等效的迭代算法。

**解**：全排列是一个两层集合，采用 vector<vector<int>>容器存放。首先置 $P_i=P_1=\{\{1\}\}$，$i$ 从 2 到 $n$ 循环：置 $P_{i-1}=P_i$，清空 $P_i$，在 $P_{i-1}$ 中每个集合元素的每个位置插入 $i$，将结果添加到 $P_i$ 中，最后返回 $P_i$。对应的迭代法算法如下：

```
vector<int> Insert(vector<int> s,int i,int j)           //在 s 的位置 j 插入 i
{   vector<int>::iterator it=s.begin()+j;               //求出插入位置
    s.insert(it,i);                                     //插入整数 i
    return s;
}
vector<vector<int>> CreatePi(vector<int> s,int i)       //在 s 集合中 i-1 到 0 的位置插入 i
{   vector<vector<int>> tmp;
    for (int j=s.size();j>=0;j--)                       //在 s 的每个位置插入 i
    {   vector<int> s1=Insert(s,i,j);
        tmp.push_back(s1);                              //添加到 Pi 中
    }
    return tmp;
}
vector<vector<int>> Perm1(int n)                        //用迭代法求 1～n 的全排列
{   vector<vector<int>> Pi;                             //存放 1～i 的全排列
    Pi.push_back({1});
    vector<vector<int>> Pi_1;                           //存放 1～i-1 的全排列
    for (int i=2;i<=n;i++)                              //迭代循环:添加 2～n
    {   Pi_1=Pi;                                        //新值取代旧值
        Pi.clear();
        for (auto it=Pi_1.begin();it!=Pi_1.end();it++)
```

```
        {   vector<vector<int>> tmp=CreatePi(*it,i); //在 it 集合中插入 i 得到 tmp
            for(int k=0;k<tmp.size();k++)
                Pi.push_back(tmp[k]);              //将 tmp 的全部元素添加到 Pi 中
        }
    }
    return Pi;
}
```

**17.** 在《教程》3.4.3 节中求 1～$n$ 的全排列的递归算法采用的是先递后合，请设计等效的先合后递的递归算法。

**解：** 在先合后递的递归算法中，$P$ 首先置为 $P_1=\{\{1\}\}$，再依次产生 $P_2,\cdots,P_n$。也就是说当 $i<=n$ 时，将 $P$ 看成 $P_{i-1}$，在 $P_{i-1}$ 中每个集合元素的每个位置插入 $i$ 得到 $P_i$，再递归添加 $i+1$；若 $i>n$，说明已经生成 1～$n$ 的全排列 $P$，返回 $P$ 即可。对应的递归算法如下：

```
vector<vector<int>> perm31(vector<vector<int>> P,int n,int i)   //先合后递的递归算法
{   if(i<=n)
    {   vector<vector<int>> Pi;
        for (auto it=P.begin();it!=P.end();it++)                 //由 P 产生 Pi
        {   vector<vector<int>> tmp=CreatePi(*it,i);             //在 it 集合中插入 i 得到 tmp
            for(int k=0;k<tmp.size();k++)
                Pi.push_back(tmp[k]);                           //将 tmp 的全部元素添加到 Pi 中
        }
        return perm31(Pi,n,i+1);
    }
    else return P;
}
vector<vector<int>> Perm3(int n)                 //用递归法求 1－n 的全排列
{   vector<vector<int>> P={{1}};                 //P 存放{1－n}的全排列，初始时置为 P1
    if(n==1)
        return P;
    else
        return perm31(P,n,2);
}
```

**18.** 给定一个整数数组 $a$，打印一个和三角形，其中第一层包含数组元素，以后每一层的元素数比上一层少一个，该层的元素是上一层中连续两个元素的和，设计一个算法求最高层的整数。例如，$a=\{1,2,3,4,5\}$，对应的和三角形如下：

```
          48
       20   28
    8   12   16
 3   5   7   9
1   2   3   4   5
```

求出的最高层的整数为 48。

**解法 1：** 设 $f(a)$为数组 $a$ 的和三角形中最高层的整数，为大问题。假设 $a$ 中含 $n$ 个整数，分为两种情况：

① 当 $n=1$ 时，$a[0]$就是 $a$ 的和三角形中最高层的整数，直接返回 $a[0]$。

② 当 $n>1$ 时，由 $a$ 中两两元素相加得到数组 $b$，$b$ 中的元素个数为 $n-1$，求 $f(b)$ 是对应的小问题，此时返回 $f(b)$ 即可。

对应的递归算法如下：

```
int solve1(vector<int> &a)                    //解法 1:递归算法
{   int n=a.size();
    if (n==1)
        return a[0];
    else
    {   vector<int> b(n-1);
        for (int i=0;i<n-1;i++)
            b[i]=a[i]+a[i+1];
        return solve1(b);
    }
}
```

**解法 2**：采用迭代法。定义一个队列 qu，先将 $a$ 中的所有元素进队，当队中元素个数大于 1 时循环：对于队中的 $n$ 个元素，出队 $n$ 次，每次求出相邻元素和后进队，共进队 $n-1$ 次。当队中只有一个元素时返回该元素。对应的迭代算法如下：

```
int solve2(vector<int> &a)                    //解法 2:迭代算法
{   int n=a.size();
    if (n==1)
        return a[0];
    else
    {   queue<int> qu;
        for(int i=0;i<n;i++)
            qu.push(a[i]);                    //a 中的所有元素进队
        while(qu.size()>1)                    //循环到队列中只有一个元素时为止
        {   n=qu.size();
            int x=qu.front(); qu.pop();       //出队首元素
            for(int i=1;i<n;i++)              //循环 n-1 次
            {   int y=qu.front(); qu.pop();   //出队元素 y
                qu.push(x+y);                 //x+y 和进队
                x=y;                          //替换
            }
        }
        return qu.front();                    //返回结果
    }
}
```

**19**. 给定一个含 $n$ 个元素的整数序列 $a$，设计一个算法求其中两个不同元素相加的绝对值的最小值。

**解法 1**：采用穷举法，任意两个不同元素求相加的绝对值，比较求最小值 ans，算法的时间复杂度为 $O(n^2)$。对应的算法如下：

```
int minabs1(vector<int> &a)                   //解法 1
{   int n=a.size();
    int ans=0x3f3f3f3f;                       //初始置为∞
```

```
    for(int i=0;i<n-1;i++)
    {   for(int j=i+1;j<n;j++)
        {   ans=min(ans,abs(a[i]+a[j]));
            if(ans==0) return ans;      //当结果为0时不必继续
        }
    }
    return ans;
}
```

**解法 2**：改进穷举法算法，如果 $a$ 中元素全部是正数，只需要找到其中两个不同的最小元素，答案就是它们相加的结果，对应的时间复杂度为 $O(n)$。但这里 $a$ 中可能有负数，那么在这种情况下就变成了求差的绝对值，而差的绝对值最小的两个整数一定是大小最相近的，为此先对数组 $a$ 递增排序，用 low 和 high 前后遍历，求 sum=a[low]+a[high]，将最小绝对值保存在 ans 中，如果 sum>0 除去 $a$[high]，如果 sum<0 除去 $a$[low]。算法的时间主要花费在排序上，时间复杂度为 $O(n\log_2 n)$。对应的算法如下：

```
int minabs2(vector<int> &a)             //解法 2
{   int n=a.size();
    int ans=0x3f3f3f3f;                 //初始置为∞
    sort(a.begin(),a.end());            //递增排序
    int low=0,high=n-1;
    while(low<high)
    {   int sum=a[low]+a[high];
        ans=min(ans,abs(sum));
        if(ans==0) return ans;          //当结果为0时不必继续
        if(sum>0) high--;
        if(sum<0) low++;
    }
    return ans;
}
```

# 3.4 上机实验题及其参考答案

## 3.4.1 求最长重复子串

编写一个实验程序 exp3-1，采用穷举法求字符串 $s$ 中最长的可重叠重复的子串。例如，$s$="aaa"，结果是"aa"。

**解**：采用穷举法，用 maxlen 表示 $s$ 中最长的可重叠重复子串的长度(初始为 0)，maxi 表示其起始下标。用 $i$ 遍历字符串 $s$，$j$ 从 $i+1$ 开始找到 $s$[$i$,curlen]=$s$[$j$,curlen]的重复子串($s$[$i$,curlen]表示 $s$ 中从 $i$ 下标开始长度为 curlen 的子串)，若 curlen>maxlen，则置 maxlen=curlen，max$i$=$i$。最后将 $s$[maxi,maxlen]存放在 ans 中，并且返回 ans。对应的程序如下：

```
#include<iostream>
```

```
#include <vector>
#include <string>
using namespace std;
string longestsubstr(string s)
{   int n=s.size();
    int i=0,j;
    int maxlen=0,maxi,curlen;
    while(i<n)
    {   j=i+1;
        while(j<n)
        {   curlen=0;
            while(j<n && s[i+curlen]==s[j+curlen])
                curlen++;
            if(curlen>maxlen)
            {   maxi=i;
                maxlen=curlen;
            }
            j++;
        }
        i++;
    }
    string ans="";
    int cnt=0;
    for(int i=maxi;cnt<maxlen;i++,cnt++)
        ans+=s[i];
    return ans;
}
int main()
{   vector<string> ss{"aababcabcd","aaaaaaaaa","aaaaabaaaabac"};
    cout << "实验结果" << endl;
    for(int i=0;i<ss.size();i++)
    {   cout << "   串" << ss[i] << "的结果: \t";
        cout << longestsubstr(ss[i]) << endl;
    }
    return 0;
}
```

上述实验程序的输出结果如图 3.4 所示。


图 3.4 exp3-1.cpp 的执行结果

**思考题**：如何求字符串 $s$ 中最长的不重叠的子串。

## 3.4.2 求子矩阵元素和

编写一个实验程序 exp3-2，给定一个 $m$ 行 $n$ 列的二维矩阵 $\boldsymbol{a}(2\leqslant m,n\leqslant 100)$，其中所有元素为整数。其大量的运算是求左上角为 $a[i,j]$、右下角为 $a[s,t](i<s,j<t)$ 的子矩阵

的所有元素之和。请设计高效的算法求给定子矩阵的所有元素之和，并用相关数据进行测试。

**解**：建立一个 $m$ 行 $n$ 列的二维数组 $b$，$b[i,j]$ 为 $a$ 中左上角为 $a[0,0]$、右下角为 $a[i,j]$ 的子矩阵的所有元素之和，设计算法 Sum 由数组 $a$ 求出数组 $b$，时间复杂度为 $O(m\times n)$。在求出 $b$ 数组后，设计算法 submat 求数组 $a$ 中左上角为 $a[i,j]$、右下角为 $a[s,t](i\leqslant s,j\leqslant t)$ 的子矩阵（用 $(i,j)-(s,t)$ 表示这样的子矩阵）的所有元素的和，它可以利用数组 $b$ 来实现，如图 3.5 所示，$(i,j)-(s,t)$ 子矩阵的所有元素之和为 $b[s][t]-b[s][j-1]-b[i-1][t]+b[i-1][j-1]$，另外考虑两种特殊情况，$i=0$ 时如图 3.6 所示，$j=0$ 时如图 3.7 所示，显然该算法的时间复杂度为 $O(1)$。

这样尽管 sum 算法的时间复杂度为 $O(m\times n)$，但只需要执行一次（除非数组 $a$ 发生改变），而 submat 算法需要大量应用，所以这种设计是十分经济的。

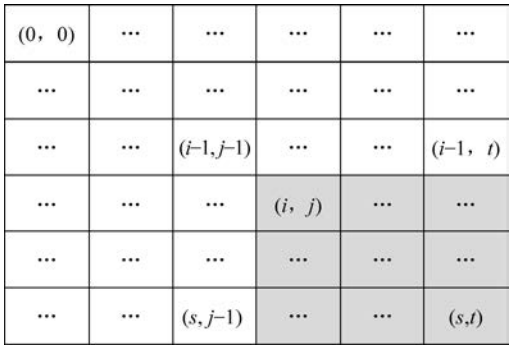


图 3.5　子矩阵和 $=b[s][t]-b[s][j-1]-b[i-1][t]+b[i-1][j-1]$


图 3.6　$i=0$ 时子矩阵和 $=b[s][t]-b[s][j-1]$

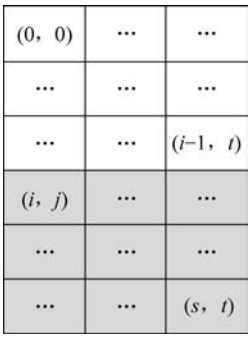


图 3.7　$j=0$ 时子矩阵和 $=b[s][t]-b[i-1][t]$

对应的程序如下：

```
#include<iostream>
#include<vector>
using namespace std;
vector<vector<int>> Sum(vector<vector<int>> &a)          //由矩阵 a 求矩阵 b
{   int m=a.size();
    int n=a[0].size();
    vector<vector<int>> b(m,vector<int>(n));
    b[0][0]=a[0][0];
    for (int i=1;i<m;i++)                                //求 b 的第 0 列
        b[i][0]=b[i-1][0]+a[i][0];
```

```
    for (int j=1;j<n;j++)                                    //求 b 的第 0 行
        b[0][j]=b[0][j-1]+a[0][j];
    for (int i=1;i<m;i++)                                    //求 b[i][j]
        for (int j=1;j<n;j++)
            b[i][j]=a[i][j]+b[i-1][j]+b[i][j-1]-b[i-1][j-1];
    return b;
}
int submat(vector<vector<int>> &b,int i,int j,int s,int t)  //求[i,j]-[s,t]子矩阵元素和
{   int sum;
    if (i==0 && j==0)
        return b[s][t];
    else if(i==0)
        sum=b[s][t]-b[s][j-1];
    else if(j==0)
        sum=b[s][t]-b[i-1][t];
    else
        sum=b[s][t]-b[s][j-1]-b[i-1][t]+b[i-1][j-1];
    return sum;
}
int main()
{   vector<vector<int>> a={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    vector<vector<int>> q={{0,0,1,1},{0,0,2,1},{1,1,2,3},{0,0,2,3},{0,1,2,3}};
    int m=a.size();
    int n=a[0].size();
    printf("a:\n");
    for(int i=0;i<m;i++)
    {   for(int j=0;j<n;j++)
            printf("%4d",a[i][j]);
        printf("\n");
    }
    vector<vector<int>> b=Sum(a);
    cout << "实验结果" << endl;
    for(int i=0;i<q.size();i++)
    {   printf("  [%d,%d]-[%d,%d]子矩阵元素和=",q[i][0],q[i][1],q[i][2],q[i][3]);
        printf("%d\n",submat(b,q[i][0],q[i][1],q[i][2],q[i][3]));
    }
    return 0;
}
```

上述实验程序的输出结果如图 3.8 所示。

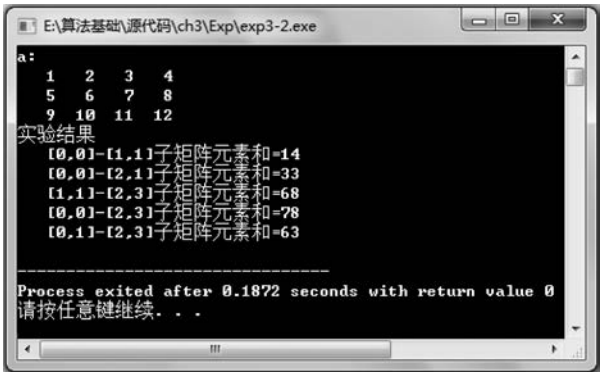


图 3.8　exp3-2.cpp 的执行结果

**思考题**：当数组 $a$ 中的元素 $a[i][j]$ 发生改变时，如何设计高效的算法修改对应的 $b$ 数

组(注意仅影响 $b[i..m-1..j..n-1]$的部分)。

## 3.4.3 求 $n$ 阶螺旋矩阵

编写一个实验程序 exp3-3,采用非递归和递归算法创建一个 $n(1\leqslant n\leqslant 10)$阶螺旋矩阵并输出。例如,$n=4$ 时的螺旋矩阵如下:

```
1     2     3     4
12    13    14    5
11    16    15    6
10    9     8     7
```

**解**:采用递归求解时,设 $f(x,y,\text{start},n)$用于创建左上角为$(x,y)$、起始元素值为 start 的 $n$ 阶螺旋矩阵,共 $n$ 行 $n$ 列,它是大问题;则 $f(x+1,y+1,\text{start},n-2)$用于创建左上角为$(x+1,y+1)$、起始元素值为 start 的 $n-2$ 阶螺旋矩阵,共 $n-2$ 行 $n-2$ 列,它是小问题,如图 3.9 所示为 $n=4$ 时的大问题和小问题。对应的递归模型如下:

$f(x,y,\text{start},n)$≡不做任何事情　　当 $n\leqslant 0$ 时

$f(x,y,\text{start},n)$≡产生只有一个元素的螺旋矩阵　当 $n=1$ 时

$f(x,y,\text{start},n)$≡产生$(x,y)$的那一圈;　　当 $n>1$ 时

$f(x+1,y+1,\text{start},n-2)$

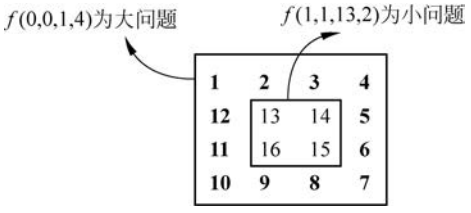


图 3.9　$n=4$ 时的大问题和小问题

非递归算法则是采用循环语句代替递归调用。对应的程序如下:

```
#include<iostream>
using namespace std;
#define N 15
int s[N][N];
int n;
void CreateaLevel(int &start,int ix,int iy,int ex,int ey)          //产生一圈的螺旋矩阵元素
{   if (ix==ex)                                                    //该圈只有一个元素时
        s[ix][iy]=start++;
    else
    {   int curx=ix;
        int cury=iy;
        while (curx!=ex)                                           //上一行
        {   s[iy][curx]=start++;
            curx++;
        }
        while (cury!=ey)                                           //右一列
        {   s[cury][ex]=start++;
            cury++;
```

```
        }
        while (curx!=ix)                                    //下一行
        {   s[ey][curx]=start++;
            curx--;
        }
        while (cury!=iy)                                    //左一列
        {   s[cury][ix]=start++;
            cury--;
        }
    }
}
void Spiral1(int n)                                         //非递归创建螺旋矩阵
{   int start=1;
    int ix=0,iy=0;
    int ex=n-1,ey=n-1;
    while (ix<=ex && iy<=ey)
        CreateaLevel(start,ix++,iy++,ex--,ey--);
}
void Spiral2(int x,int y,int start,int n)                   //递归创建螺旋矩阵
{   if (n<=0)                                               //递归结束条件
        return;
    if (n==1)                                               //矩阵大小为1时
    {   s[x][y] = start;
        return;
    }
    for (int i=x; i<x+n-1; i++)                             //上一行
        s[y][i]=start++;
    for (int j=y; j<y+n-1; j++)                             //右一列
        s[j][x+n-1]= start++;
    for (int i=x+n-1; i>x; i--)                             //下一行
        s[y+n-1][i]= start++;
    for (int j=y+n-1; j>y; j--)                             //左一列
        s[j][x]=start++;
    Spiral2(x+1,y+1,start,n-2);                             //递归调用
}
void Display()                                              //输出螺旋矩阵
{   for (int i=0; i<n; i++)
    {   for (int j=0; j<n; j++)
            printf("%4d", s[i][j]);
        printf("\n");
    }
}
int main()
{   n=5;
    printf("非递归方法建立的%d阶螺旋矩阵:\n",n);
    Spiral1(n);
    Display();
```

```
    n=5;
    printf("递归方法建立的%d阶螺旋矩阵:\n",n);
    Spiral2(0,0,1,n);
    Display();
    return 0;
}
```

上述程序的执行结果如图 3.10 所示。

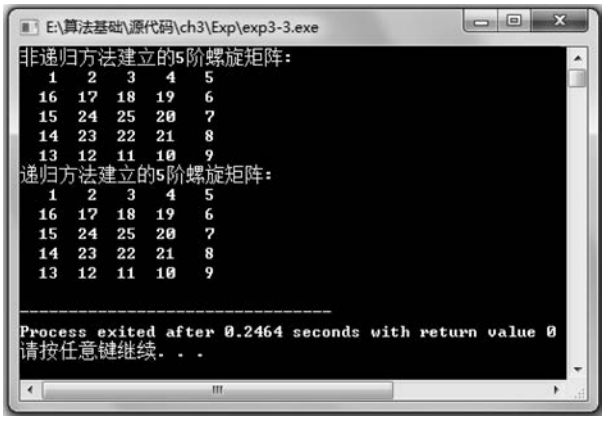


图 3.10　exp3-3.cpp 的执行结果

## 3.4.4　验证汉诺塔问题

《教程》的例 3-8 中给出了求解汉诺塔问题的递归算法，请针对该算法推导出移动 $n$ 盘片时搬动盘片总次数的公式，再用递归算法求出搬动盘片的总次数，前者称为公式求解结果，后者称为递归算法求解结果，判断两者是否相同。编写一个实验程序 exp3-4 完成上述功能，并用相关数据进行测试。

**解**：设 Hanoi($n$,$x$,$y$,$z$)中的搬动盘片总次数为 $C(n)$。根据递归算法有以下递归式：

$C(n)=1$　　当 $n=1$ 时

$C(n)=2C(n-1)+1$　　当 $n>1$ 时

则：

$$\begin{aligned}C(n)&=2[2C(n-2)+1]+1=2^2C(n-2)+1+2^1\\&=2^3C(n-3)+1+2^1+2^2\\&=\cdots\\&=2^{n-1}C(1)+1+2^1+2^2+\cdots+2^{n-2}\\&=2^n-1\end{aligned}$$

所以移动 $n$ 盘片时搬动盘片的总次数为 $2^n-1$。对应的验证程序如下：

```
#include<iostream>
#include<cmath>
using namespace std;
int Hanoi(int n,char x,char y,char z)
{   if (n==1)
        return 1;                    //搬一次盘片:将盘片 n 从 x 搬到 z
    else
```

```
    {   int cnt=Hanoi(n-1,x,z,y);
        cnt++;                          //搬一次盘片:将盘片 n 从 x 搬到 z
        cnt+=Hanoi(n-1,y,x,z);
        return cnt;
    }
}
int main()
{   printf("实验结果\n");
    for(int n=2;n<=10;n++)
    {   int cnt1=(int)pow(2,n)-1;
        int cnt2=Hanoi(n,'x','y','z');
        printf("  公式求解结果=%4d 递归算法求解结果=%4d 两者%s\n",
            cnt1,cnt2,(cnt1==cnt2? "相等":"不相等"));
    }
    return 0;
}
```

上述程序的执行结果如图 3.11 所示。


图 3.11 exp3-4.cpp 的执行结果

## 3.5 在线编程题及其参考答案

### 3.5.1 LeetCode344——反转字符串

问题描述：将输入的字符串反转过来。不要给另外的数组分配额外的空间。要求设计如下函数：

```
class Solution {
public:
    void reverseString(vector<char>& s) {}
};
```

**解法 1**：采用迭代算法。将 s 的两端字符交换，直到未交换区间为空或者只有一个字符时为止。对应的程序如下：

```
class Solution {
public:
    void reverseString(vector<char>& s)          //迭代算法
```

```
    {   int i=0,j=s.size()-1;
        while(i<j)
        {   swap(s[i],s[j]);
            i++; j--;
        }
    }
};
```

上述程序提交时通过，执行时间为 20ms，内存消耗为 22.7MB。

**解法 2**：采用递归算法。设 $f(s,i,j)$用于反转 s[$i..j$]，先交换 s[$i$]和 s[$j$]，子问题为 $f(s,i+1,j-1)$。对应的程序如下：

```
class Solution {
public:
    void reverseString(vector<char>& s)                    //递归算法
    {   int n=s.size();
        if(n==0 || n==1)
            return;
        rev(s,0,n-1);
    }
    void rev(vector<char> &s,int i,int j)
    {   if(i>j || i==j) return;
        swap(s[i],s[j]);
        rev(s,i+1,j-1);
    }
};
```

上述程序提交时通过，执行时间为 24ms，内存消耗为 22.7MB。

## 3.5.2 LeetCode206——反转链表

问题描述：反转一个不带头结点的单链表 head。例如 head 为[1,2,3,4,5]，反转后为[5,4,3,2,1]。要求设计如下函数：

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {  }
};
```

**解法 1**：采用迭代算法。先建立一个反转单链表的头结点 rh，用 $p$ 遍历单链表 head，将结点 $p$ 采用头插法插入 rh 的表头。最后返回 rh->next。对应的程序如下：

```
class Solution {
public:
    ListNode* reverseList(ListNode* head)                  //迭代算法
    {   ListNode* rh=new ListNode;                         //建立一个头结点
        ListNode* p=head;
        while(p!=NULL)
        {   ListNode* q=p->next;                           //临时保存结点 p 的后继结点
            p->next=rh->next;
            rh->next=p;
            p=q;
        }
        return rh->next;
    }
};
```

上述程序提交时通过，执行时间为 8ms，内存消耗为 8MB。

**解法 2**：采用递归算法。设 $f$(head)的功能是反转单链表 head 并且返回反转单链表的首结点 rh，其过程如图 3.12 所示。对应的程序如下：

```
class Solution {
public:
    ListNode * reverseList(ListNode * head)                    //递归算法
    {   if (head==NULL || head->next==NULL)
        return head;
        ListNode * rh=reverseList(head->next);
        head->next->next=head;
        head->next=NULL;
        return rh;
    }
};
```

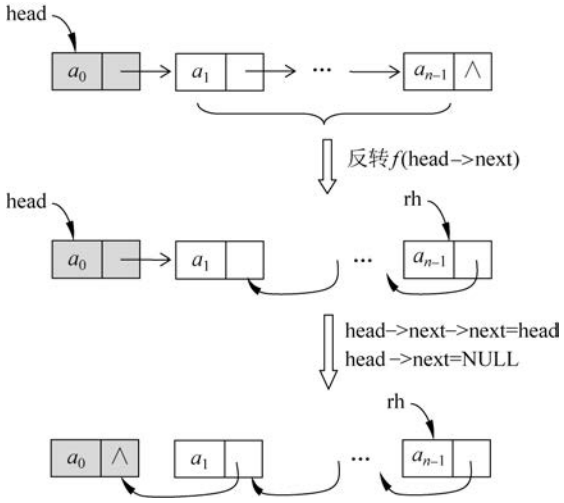


图 3.12　递归反转单链表 head 的过程

上述程序提交时通过，执行时间为 8ms，内存消耗为 8.4MB。

## 3.5.3　LeetCode24——两两交换链表中的结点

问题描述：给定一个不带头结点的单链表 head，两两交换其中相邻的结点，并返回交换后的链表。注意，不能只单纯地改变结点内部的值，而是需要实际地进行结点交换。例如 head 为[1,2,3,4,5]，两两交换后变为[2,1,4,3,5]。要求设计如下函数：

```
class Solution {
public:
    ListNode * swapPairs(ListNode * head) {}
};
```

**解法 1**：采用迭代算法。先将前面的两个结点交换，交换后 head 指向 $a_1$ 的结点，last 指向 $a_0$ 的结点，然后让 $p$、$q$、$r$ 分别指向其后的 3 个相邻结点，如图 3.13 所示，若 $p$ 或者 $q$ 为空则结束，否则交换结点 $p$、$q$。

对应的程序如下：

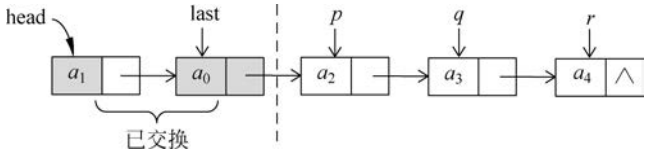


图 3.13　两两结点交换的过程

```
class Solution {
public:
    ListNode * swapPairs(ListNode * head)                    //迭代算法
    {   if (head==NULL || head->next==NULL)
            return head;                                     //head 为空或者只有一个结点的情况
        ListNode * p, * q, * r, * last;
        p=head;                                              //p 指向 a0
        q=head->next;                                        //q 指向 a1
        r=q->next;                                           //r 指向 a2
        head=q; p->next=r;                                   //交换 p 和 q 结点,head 指向新的首结点
        head->next=p;
        last=p;
        while(true)
        {   p=r;
            if (p==NULL || p->next==NULL)
                break;                                       //单链表 p 为空或者只有一个结点的情况
            q=p->next;
            r=q->next;
            last->next=q; p->next=r;                         //交换 p 和 q 结点
            q->next=p; p->next=r;
            last=p;                                          //重新设置 last
        }
        return head;                                         //返回交换后的单链表
    }
};
```

上述程序提交时通过,执行时间为 4ms,内存消耗为 7.3MB。

**解法 2**：采用递归算法。设 $f$(head)是大问题,用于两两交换链表 head 中的结点。

① 若单链表 head 为空或者只有一个结点(head=NULL 或者 head->next=NULL),交换后的结果单链表没有变化,返回 head。

② 否则,让 last 和 $p$ 分别指向 $a_1$ 和 $a_2$ 结点,如图 3.14 所示,显然 $f(p)$为小问题,用于两两交换链表 $p$ 中的结点。$f$(head)的执行过程是先交换 last 和 head 结点(让 head 指向 $a_1$ 结点,last 指向 $a_0$ 结点),再置 last->next=$f(p)$,最后返回 head。

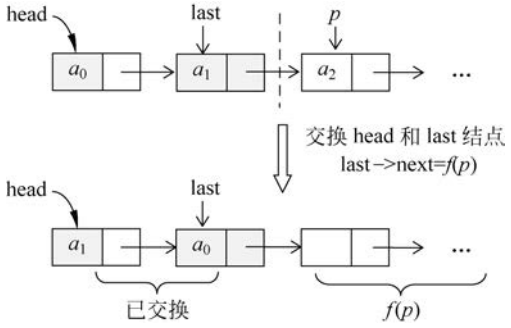


图 3.14　有两个或者两个以上结点时 $f$(head)的执行过程

对应的程序如下：

```
class Solution {
public:
    ListNode * swapPairs(ListNode * head)                    //递归算法
    {   if (head==NULL || head->next==NULL)
            return head;                                     //head 为空或者只有一个结点的情况
        ListNode * last=head->next;                          //last 指向 a1
        ListNode * p=last->next;                             //p 指向 a2
        last->next=head;                                     //交换 head 和 last 结点
        head=last;
        last=head->next;
        last->next=swapPairs(p);
        return head;
    }
};
```

上述程序提交时通过，执行时间为 4ms，内存消耗为 7.2MB。

## 3.5.4 LeetCode62——不同路径

问题描述：一个机器人位于一个 $m \times n$（$1 \leqslant m, n \leqslant 100$）网格的左上角（起始点标记为“Start”）。机器人每次只能向下或者向右移动一步。机器人试图到达网格的右下角（标记为“Finish”）。问总共有多少条不同的路径？要求设计如下函数：

```
class Solution {
public:
    int uniquePaths(int m,int n){   }
};
```

例如，$m=3, n=3$，对应的网格如图 3.15 所示，结果为 6。

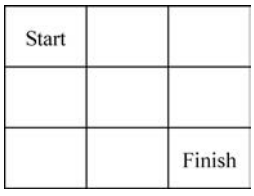

图 3.15　3×3 的网格

**解**：在从左上角到右下角的任意路径中，一定是向下走 $m-1$ 步向右走 $n-1$ 步，不妨置 $x=m-1, y=n-1$，路径长度为 $x+y$。例如对于图 3.15，这里 $x=2, y=2$，所有路径长度为 4，6 条不同的路径如下：

右右下下
右下右下
右下下右
下右右下
下右下右
下下右右

归纳起来，不同路径条数等于从 $x+y$ 个选择中挑选 $x$ 个“上”或者“右”的组合数，即 $C_{x+y}^{x}$ 或者 $C_{x+y}^{y}$（实际上从数学上推导也有 $C_{x+y}^{x}=C_{x+y}^{y}$），为了方便，假设 $x \leqslant y$，结果取 $C_{x+y}^{x}$。

$$C_{x+y}^{x}=\frac{(x+y)!}{x!y!}=\frac{(x+y)(x+y-1)\cdots(y-1)y!}{x!y!}=\frac{(x+y)\times\cdots\times(y-1)}{x\times(x-1)\times\cdots\times2\times1}$$

上式中的分子、分母均为 $x$ 个连乘，可以进一步转换为：

$$\frac{x+y}{x}\times\frac{x+y-1}{x-1}\times\cdots\times\frac{y-1}{1}$$

由于除法的结果是实数，而不同的路径数一定是整数，所以最后需要将计算的结果向上

取整得到整数结果。对应的程序如下：

```
class Solution {
public:
    int uniquePaths(int m, int n)
    {
        return comp(m-1,n-1);
    }
    int comp(int x,int y)
    {   int a=x+y,b=min(x,y);
        double ans=1.0;
        while(b>0)
            ans*=(double)(a--)/(double)(b--);
        ans+=0.5;
        return (int)ans;
    }
};
```

上述程序提交时通过，执行时间为 0ms，内存消耗为 5.8MB。

## 3.5.5 HDU1003——最大子序列和

问题描述：给定一个整数序列 $a$，请计算一个最大子序列和。例如，给定(6，−1，5，4，−7)，该序列中的最大子序列和为 6+(−1)+5+4=14。

输入格式：输入的第一行包含一个整数 $t$(1≤$t$≤20)，表示测试用例的数量，接下来是 $t$ 行，每行以一个数字 $n$(1≤$n$≤100000)开始，然后是 $n$ 个整数(整数的取值范围为−1000～1000)。

输出格式：对于每个测试用例，输出两行，第一行是"Case ＃:"，其中＃表示测试用例的编号(从 1 开始)，第二行包含 3 个整数，依次为序列中的最大子序列和、对应子序列的开始位置和结束位置。如果有多个结果，则输出第一个。每两个测试用例的输出之间输出一个空行，最后一个测试用例的输出的后面没有空行。

输入样例：

```
2
5 6 -1 5 4 -7
7 0 6 -1 1 -6 7 -5
```

输出样例：

```
Case 1:
14 1 4

Case 2:
7 1 6
```

**解**：算法原理参见《教程》3.1.2 节求最大连续子序列和问题的解法 3，但有以下两点不同。

① 这里的最大连续子序列至少包含一个元素，也就是说最大连续子序列和可能为负数。

② 需要求第一个最大连续子序列，由于最大连续子序列和相同的子序列可能有多个，这里是求第一个，也就是说起始位置最小的子序列。

第一个最大连续子序列表示为 $a[start..end]$，首先置 maxsum(最大连续子序列和)和 cursum(当前连续子序列 $a[prestart..i]$和)为 0，prestart＝0。用 $i$ 遍历 $a$，累计 $a[i]$到 cursum 中，分为两种情况：

① 若 cursum≥maxsum，说明 cursum 是一个更大的连续子序列和，将其存放在 maxsum 中，即置 maxsum＝cursum，[start，end]＝[prestart，$i$]。

② 若 cursum＜0，说明 cursum 不可能是一个更大的连续子序列和，从下一个 $i$ 开始继续遍历，所以置 cursum＝0，prestart 置为 $i+1$。

最后输出 maxsum，start 和 end。对应的程序如下：

```
#include<stdio.h>
#define INF 0x3f3f3f3f                    //∞
#define MAXN 100010
int a[MAXN];
int n;
int maxSubSum(int& start,int &end)        //求最大子序列和
{   int cursum=0;
    int maxsum=-INF;
    int prestart;
    start=end=prestart=0;
    for(int i=0;i<n;i++)
    {   cursum+=a[i];
        if(cursum>maxsum)
        {   start=prestart;
            end=i;
            maxsum=cursum;
        }
        if(cursum<0)
        {   prestart=i+1;
            cursum=0;
        }
    }
    return maxsum;
}
int main()
{   int t;
    scanf("%d",&t);
    for(int cas=1;cas<=t;cas++)
    {   scanf("%d",&n);
        for(int i=0;i<n;i++)
            scanf("%d",&a[i]);
        int start,end;
        int ans=maxSubSum(start,end);
        printf("Case %d:\n",cas);
        printf("%d %d %d\n",ans,start+1,end+1);
        if(cas!=t) printf("\n");
    }
    return 0;
}
```

上述程序提交时通过，执行时间为 46ms，内存消耗为 2116KB。

## 3.5.6 HDU1143——三平铺问题

问题描述：可以用多少种方式用 2×1 的多米诺骨牌平铺一个 3×$n$ 的矩形？如图 3.16

所示为一个 3×12 矩形的平铺示例。

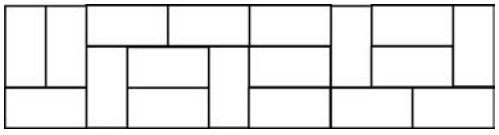

图 3.16　一个 3×12 矩形的平铺示例

输入格式：输入由几个测试用例组成，以包含－1 的行结束。每个测试用例是一行，其中包含一个整数 $0 \leqslant n \leqslant 30$。

输出格式：对于每个测试用例，输出一个整数，给出可能的平铺方案数。

输入样例：

```
2
8
12
-1
```

输出样例：

```
3
153
2131
```

**解**：设 $f(n)$表示用 2×1 的多米诺骨牌平铺一个 3×$n$ 矩形的方案数，假设平铺所用的 2×1 的多米诺骨牌个数为 $k$，则 $2k=3n$，当 $n$ 为奇数时该式右边为奇数，而左边为偶数，所以不成立，也就是说当 $n$ 为奇数时不能平铺，返回 0。下面仅考虑 $n$ 为偶数的情况。

当 $n=2$ 时，所有的平铺方案如图 3.17 所示，即 $f(2)=3$，不妨设 $f(0)=1$。

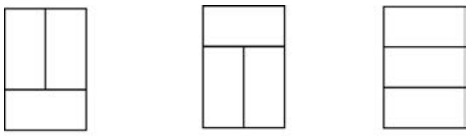


图 3.17　一个 3×2 矩形的 3 种平铺方案

当 $n>2$ 时，将 3×$n$ 矩形看成高度为 3、长度为 $n$ 的矩形，按分割线分割为(2，$n-2$)，(4，$n-4$)，(6，$n-6$)，…，($n$，0)。

① 当分割为(2，$n-2$)时，平铺方案数为 $f(2)\times f(n-2)$，即 $3f(n-2)$。

② 当分割为(4，$n-4$)时，前面长度为 4 的部分只能有两种(考虑第一列，第一种是一个横的在上面，然后一个竖的在左下方；第二种是一个横的在下面，然后一个竖的在左上方，两种情况是对称的)，其他情况与(2，$n-2$)重复，如图 3.18 所示，所以平铺方案数为 $2f(n-4)$。

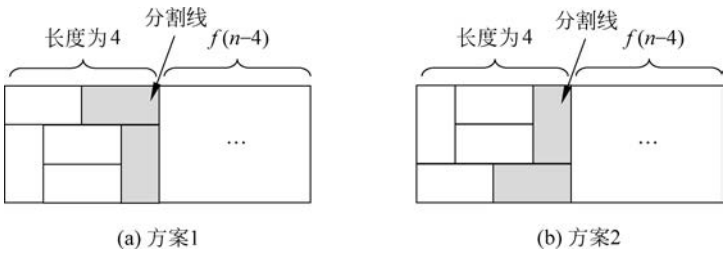


图 3.18　分割为(4，$n-4$)的情况

③ 当分割为(6,$n-6$)时,前面长度为 6 的部分也只能有两种,其他情况是重复的,如图 3.19 所示,所以平铺方案数为 $2f(n-6)$。

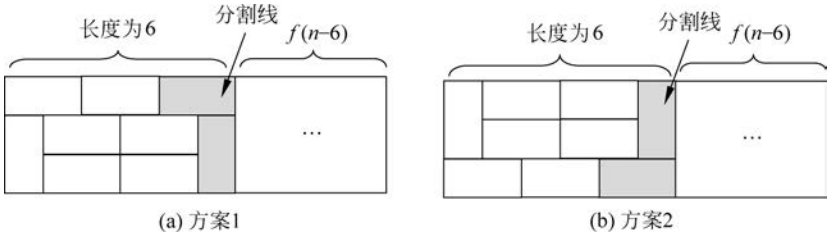


图 3.19 分割为(6,$n-6$)的情况

以此类推,利用加法原理得到 $f(n)=3f(n-2)+2f(n-4)+2f(n-6)+\cdots+2f(0)$。

用 $n-2$ 代入得到 $f(n-2)=3f(n-4)+2f(n-6)+2f(n-8)+\cdots+2f(0)$。

两式相减:$f(n)=4f(n-2)-f(n-4)$。

所以递推关系如下:

$f(0)=1$

$f(2)=3$

$f(n)=4f(n-2)-f(n-4)$ 当 $n\geqslant 4$(偶数)时

对应的程序如下:

```
#include <iostream>
using namespace std;
int main()
{   int n;
    int a[35];                    //a[i]存放 f(i)
    a[0]=1,a[2]=3;
    for(int i=4;i<=30;i+=2)  //求出 a 数组
        a[i]=4*a[i-2]-a[i-4];
    while(~scanf("%d",&n))
    {   if(n==-1) break;
        if(n%2==1)                //n 为奇数时
            printf("0\n");
        else                      //n 为偶数时
            printf("%d\n",a[n]);
    }
    return 0;
}
```

上述程序提交时通过,执行时间为 15ms,内存消耗为 1732KB。

## 3.5.7 POJ2231——奶牛的总音量

问题描述:John 收到邻居 Bob 的噪声投诉,称他的奶牛噪声太大。John 的 $n$ 头奶牛($1\leqslant n\leqslant 10000$)都在一个长长的一维牧场的不同位置吃草。每对奶牛之间可以同时对话,也就是说每头奶牛可以同时向其他 $n-1$ 头奶牛发出哞哞声。当奶牛 $i$ 对奶牛 $j$ 发出哞哞声时,这个音量必须等于它们之间的距离才能让奶牛 $j$ 完全听到哞哞声。请帮助 John 计算所有 $n\times(n-1)$个同时发出的哞哞声的总音量。

输入格式:第一行为 $n$,第二行到第 $n+1$ 行表示每只奶牛的位置(范围是 0～1000000000)。

输出格式：输出一行表示总音量。

输入样例：

```
5
1
5
3
2
4
```

输出样例：

```
40
```

**解**：题目大意是数轴上共有 $n$ 头奶牛，每头奶牛有一个位置，每头奶牛都要向其他所有奶牛发出一个哞哞声，因此一共有 $n\times(n-1)$ 个哞哞声，每个哞哞声的大小等于两头奶牛之间坐标差的绝对值，求所有哞哞声大小的和。

用一个数组 $a$ 存放所有奶牛的位置，题目就是求 $a$ 中任意两个元素差的绝对值之和。采用穷举法的程序如下：

```
#include<stdio.h>
#include<algorithm>
using namespace std;
typedef long long LL;
LL a[10005];
int main()
{   int n;
    while(~scanf("%d",&n))
    {   for(int k=0;k<n;k++)
            scanf("%lld",&a[k]);
        LL sum=0;
        for(int i=0;i<n;i++)
            for(int j=0;j<n;j++)
            {   if(i!=j)
                sum+=abs(a[i]-a[j]);
            }
        printf("%lld\n",sum);
    }
    return 0;
}
```

上述程序提交时出现超时。假设 $b[i][j]=|a[i]-a[j]|$，显然 $b$ 数组是对称的并且主对角线均为 0，只需要求出下三角部分元素和(或者上三角部分元素和)sum，输出 2sum 即可。对应的程序如下：

```
#include<stdio.h>
#include<algorithm>
using namespace std;
typedef long long LL;
LL a[10005];
int main()
{   int n;
    while(~scanf("%d",&n))
    {   for(int k=0;k<n;k++)
            scanf("%lld",&a[k]);
```

```
        LL sum=0;
        for(int i=0;i<n;i++)
            for(int j=0;j<i;j++)
                sum+=abs(a[i]-a[j]);
        printf("%lld\n",2*sum);
    }
    return 0;
}
```

上述程序提交时通过，执行时间为 593ms，内存消耗为 212KB。

## 3.5.8 POJ1050——最大子矩形

问题描述：给定一个由正整数或者负整数组成的二维数组，子矩形是位于整个数组内的大小为 1×1 或更大的任何连续子数组。矩形的总和是该矩形中所有元素的总和。在这个问题中，总和最大的子矩形称为最大子矩形。例如有以下数组：

```
  0  -2  -7  0
  9   2  -6  2
 -4   1  -4  1
-18   0   0  2
```

其最大子矩形是左下角：

```
 9  2
-4  1
-1  8
```

最大子矩形和是 15。

输入格式：输入由一个 $n\times n$ 的整数数组组成。输入以单独一行的单个正整数 $n$ 开头，表示二维数组的大小。后面是由空格(空格和换行符)分隔的 $n^2$ 个整数，这些是数组的 $n^2$ 个整数元素，以行优先顺序显示，即先从左到右为第一行中的所有整数，然后是第二行中的所有整数，以此类推。$n$ 可能大到 100，数组中的整数在 [-127,127] 范围内。

输出格式：输出最大子矩形的总和。

输入样例：

```
4
0 -2 -7 0 9 2 -6 2
-4 1 -4 1 -1

8 0 -2
```

输出样例：

```
15
```

**解**：利用《教程》3.2.2 节中求子矩阵元素和的思路，在获取 $n$ 和 $a$ 数组后，先执行 $b$=Sum($a$)求出数组 $b$，采用 $i$ 从 0 到 $n-1$，$j$ 从 0 到 $n-1$，$s$ 从 $i$ 到 $n-1$，$t$ 从 $j$ 到 $n-1$ 的四重循环，执行 curmax=submat($b$,$i$,$j$,$s$,$t$)求出每个矩阵($i$,$j$)-($s$,$t$)的元素和，比较求出最大值 ans，最后输出 ans。调用的程序如下：

```
#include<iostream>
#include<vector>
```

```
using namespace std;
#define INF 0x3f3f3f3f                                    //存放∞
vector<vector<int>> Sum(vector<vector<int>> &a)           //由矩阵 a 求矩阵 b
{   int n=a.size();
    vector<vector<int>> b(n,vector<int>(n));
    b[0][0]=a[0][0];
    for (int i=1;i<n;i++)                                 //求 b 的第 0 列
        b[i][0]=b[i-1][0]+a[i][0];
    for (int j=1;j<n;j++)                                 //求 b 的第 0 行
        b[0][j]=b[0][j-1]+a[0][j];
    for (int i=1;i<n;i++)                                 //求 b[i][j]
        for (int j=1;j<n;j++)
            b[i][j]=a[i][j]+b[i-1][j]+b[i][j-1]-b[i-1][j-1];
    return b;
}
int submat(vector<vector<int>> &b,int i,int j,int s,int t)   //求[i,j]-[s,t]子矩阵元素和
{   int sum;
    if (i==0 && j==0)
        return b[s][t];
    else if(i==0)
        sum=b[s][t]-b[s][j-1];
    else if(j==0)
        sum=b[s][t]-b[i-1][t];
    else
        sum=b[s][t]-b[s][j-1]-b[i-1][t]+b[i-1][j-1];
    return sum;
}
int main()
{   int n;
    cin>>n;
    vector<vector<int>> a(n,vector<int>(n));
    for(int i=0;i<n;i++)                                  //获取 a 数组
        for(int j=0;j<n;j++)
            cin>>a[i][j];
    vector<vector<int>> b=Sum(a);
    int ans=-INF;                                         //存放结果,初始为-∞
    for(int i=0;i<n;i++)                                  //4 重循环
    {   for(int j=0;j<n;j++)
        {   for(int s=i;s<n;s++)
            {   for(int t=j;t<n;t++)
                {   int curmax=submat(b,i,j,s,t);
                    ans=max(ans,curmax);
                }
            }
        }
    }
    cout<<ans<<endl;                                      //输出结果
    return 0;
}
```

上述程序提交时通过,执行时间为 309ms,内存消耗为 308KB。