第3章 基本算法设计方法

为了设计出解决问题的好算法，除了要掌握常用的数据结构工具外，还需要掌握算法设计方法。算法设计与分析课程主要讨论分治法、回溯法、分支限界法、贪心法和动态规划，称为五大算法策略，在学习这些算法策略之前必须具有一定的算法设计基础，本章讨论穷举法、归纳法、递归法和迭代法几种基本算法的设计方法及其递推式的计算。本章的学习要点和学习目标如下：

（1）掌握穷举法的原理和算法框架。

（2）掌握归纳法的原理和从求解问题找出递推关系的方法。

（3）掌握迭代法的原理和实现迭代算法的方法。

（4）掌握递归法的原理和实现递归算法的方法。

（5）掌握各种经典算法的设计过程。

（6）掌握递推式的各种计算方法。

（7）综合运用穷举法、归纳法、递归法和迭代法解决一些复杂的实际问题。

3.1 穷 举 法

3.1.1 穷举法概述

1. 什么是穷举法

穷举法又称枚举法或者列举法，是一种简单而直接地解决问题的方法。其基本思想是先确定有哪些穷举对象和穷举对象的顺序，按穷举对象的顺序逐一列举每个穷举对象的所有情况，再根据问题的约束条件检验哪些是问题的解，哪些应予排除。

在用穷举法解题时，针对穷举对象的数据类型而言，常用的列举方法如下。

① 顺序列举：是指答案范围内的各种情况很容易与自然数对应甚至就是自然数，可以按自然数的变化顺序去列举。

② 组合列举：当答案的数据形式为一些元素的组合时，往往需要用组合列举。组合是无序的。

③ 排列列举：有时答案的数据形式是一组数的排列，列举出所有答案所在范围内的排列，为排列列举。

穷举法的作用如下：

① 理论上讲穷举法可以解决可计算领域中的各种问题，尤其是处在计算机计算速度非常快的今天，穷举法的应用领域是非常广阔的。

② 在实际应用中通常要解决的问题规模不大，用穷举法设计的算法其运算速度是可以接受的，此时不值得设计一个更高效率的算法。

③ 穷举法算法一般逻辑清晰，编写的程序简洁明了。

④ 穷举法算法一般不需要特别证明算法的正确性。

⑤ 穷举法可作为某类问题时间性能的上界，用来衡量同样问题的更高效率的算法。

穷举法的主要缺点是设计的大多数算法的效率都不高，主要适合问题规模比较小的问题的求解。为此在采用穷举法求解时应根据问题的具体情况分析归纳，寻找简化规律，精简穷举循环，优化穷举策略。

2. 穷举法算法框架

穷举法算法一般使用循环语句和选择语句实现，其中循环语句用于枚举穷举对象所有可能的情况，而选择语句用于判定当前的条件是否为所求的解。其基本流程如下：

① 根据问题的具体情况确定穷举变量（简单变量或数组）。

② 根据确定的范围设置穷举循环。

③ 根据问题的具体要求确定解满足的约束条件。

④ 设计穷举法程序，并执行和调试，对执行结果进行分析与讨论。

假设某个问题的穷举变量是 x 和 y，穷举顺序是先 x 后 y，均为顺序列举，它们的取值范围分别是 $x\in(x_1,x_2,\cdots,x_n)$，$y\in(y_1,y_2,\cdots,y_m)$，约束条件为 $p(x_i,y_j)$，对应穷举法算法的基本框架如下：

```
void Exhaustive(x,n,y,m)
{   for (int i=1;i<=n;i++)                    //枚举 x 的所有可能的值
        for (int j=1;j<=m;j++)                //枚举 y 的所有可能的值
        {   ...
            if (p(x[i],y[j]))
                输出一个解;
            ...
        }
}
```

从中看出，x 和 y 所有可能的搜索范围是笛卡儿积，即($[x_1,y_1]$,$[x_1,y_2]$,…,$[x_1,y_m]$,…,$[x_n,y_1]$,$[x_n,y_2]$,…,$[x_n,y_m]$)，这样的搜索范围可以用一棵树表示，称为解空间树，它包含求解问题的所有解，求解过程就是在整个解空间树中搜索满足约束条件 $p(x_i,y_j)$ 的解。

扫一扫

视频讲解

【例 3-1】 鸡兔同笼问题。现有一笼子，里面有鸡和兔子若干只，数一数，共有 a 个头、b 条腿，设计一个算法求鸡和兔子各有多少只？

解 由于有鸡和兔两种动物，每只鸡有两条腿，每只兔有 4 条腿，设置两个变量，x 表示鸡的只数、y 表示兔的只数，那么穷举变量就是 x 和 y，假设穷举变量的顺序是先 x 后 y(在本问题中也可以先 y 后 x)。

显然，x 和 y 的取值范围都是 0～a，约束条件 $p(x,y)=(x+y==b)$ && $(2x+4y=b)$。以 $a=3$，$b=8$ 为例，对应的解空间树如图 3.1 所示，根结点的分支对应 x 的各种取值，第二层结点的分支为 y 的各种取值，其中“×”结点是不满足约束条件的结点，带阴影的结点是满足约束条件的结点，所以结果是 $x=2/y=1$。对应的算法如下：

```
void solve1(int a,int b)
{   for(int x=0;x<=a;x++)
        for(int y=0;y<=a;y++)
        {   if(x+y==a && 2*x+4*y==b)
                printf("x=%d,y=%d\n",x,y);
        }
}
```

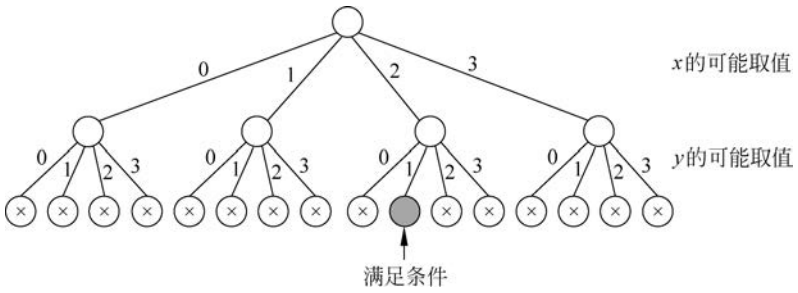


图 3.1 $a=3$，$b=8$ 的解空间树

从图 3.1 中看到，在解空间树中共 21 个结点，显然结点个数越多时间性能越差，可以稍做优化，鸡的只数最多为 min(a,$b/2$)，兔的只数最多为 min(a,$b/4$)。仍以 $a=3$，$b=8$ 为例，x 的取值范围是 0～3，y 的取值范围是 0～2，对应解空间树如图 3.2 所示，共 17 个结点。

对应的优化算法如下：

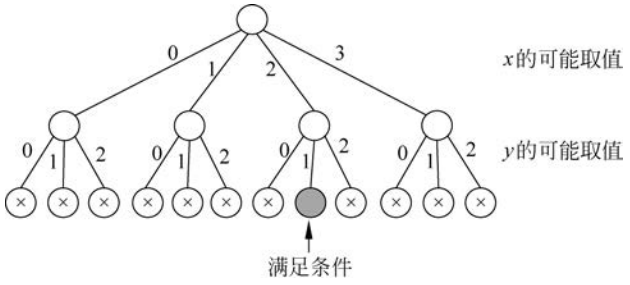


图 3.2　$a=3,b=8$ 的优化解空间树

```
void solve2(int a,int b)
{   for(int x=0;x<=min(a,b/2);x++)
        for(int y=0;y<=min(a,b/4);y++)
        {   if(x+y==a && 2*x+4*y==b)
                printf("x=%d,y=%d\n",x,y);
        }
}
```

尽管穷举法算法通常性能较差，但可以以它为基础进行优化继而得到高性能的算法，优化的关键是能够找出求解问题的优化点，不同的问题优化点是不相同的，这就需要大家通过大量实训掌握一些基本算法设计技巧。后面通过两个应用讨论穷举法算法设计方法以及穷举法算法的优化过程。

3.1.2　最大连续子序列和

扫一扫

视频讲解

1. 问题描述

给定一个含 $n(n\geqslant 1)$ 个整数的序列，要求求出其中最大连续子序列的和。例如序列(−2,11,−4,13,−5,−2)的最大子序列和为 20，序列(−6,2,4,−7,5,3,2,−1,6,−9,10,−2)的最大子序列和为 16。规定一个序列的最大连续子序列和至少是 0，如果小于 0，其结果为 0。

2. 解法 1

设含 n 个整数的序列为 $a[0..n-1]$，其中连续子序列为 $a[i..j]$($i\leqslant j$，$0\leqslant i\leqslant n-1$，$i\leqslant j\leqslant n-1$)，求出它的所有元素之和 cursum，并通过比较将最大值存放在 maxsum 中，最后返回 maxsum。这种解法通过穷举所有连续子序列(一个连续子序列由起始下标 i 和终止下标 j 确定)来得到，是典型的穷举法思想。

例如，对于 $a[0..5]=\{-2,11,-4,13,-5,-2\}$，求出的 $a[i..j]$($0\leqslant i\leqslant j\leqslant 5$)的所有元素和如图 3.3 所示(行号为 i，列号为 j)，其过程如下：

(1) $i=0$，依次求出 $j=0,1,2,3,4,5$ 的子序列和分别为−2,9,5,18,13,11。

(2) $i=1$，依次求出 $j=1,2,3,4,5$ 的子序列和分别为 11,7,20,15,13。

(3) $i=2$，依次求出 $j=2,3,4,5$ 的子序列和分别为−4,9,4,2。

(4) $i=3$，依次求出 $j=3,4,5$ 的子序列和分别为 13,8,6。

(5) $i=4$，依次求出 $j=4,5$ 的子序列和分别为−5,−7。

(6) $i=5$，求出 $j=5$ 的子序列和为−2。

其中 20 是最大值，即最大连续子序列和为 20。

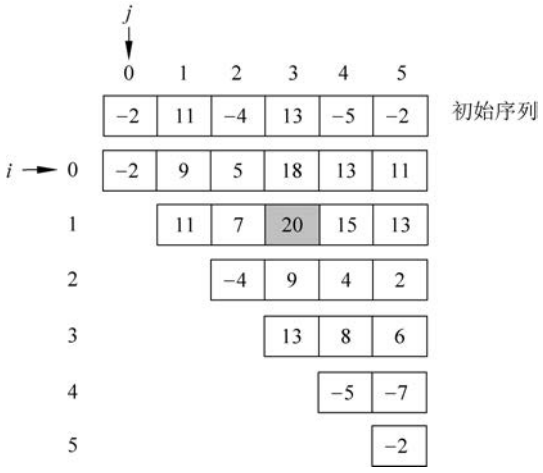


图 3.3 所有 $a[i..j]$子序列($0 \leqslant i \leqslant j \leqslant 5$)的元素和

对应的算法如下：

```
int maxSubSum1(vector<int> &a)                    //解法 1
{   int n=a.size();
    int maxsum=0,cursum;
    for (int i=0;i<n;i++)                         //两重循环穷举所有的连续子序列
    {   for (int j=i;j<n;j++)
        {   cursum=0;
            for (int k=i;k<=j;k++)                //求 a[i..j]子序列元素和 cursum
                cursum+=a[k];
            maxsum=max(maxsum,cursum);            //通过比较求最大连续子序列之和
        }
    }
    return maxsum;
}
```

【算法分析】 在 maxSubSum1(a,n)算法中用了三重循环，所以有：

$$T(n)=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j}1=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j-i+1)=\frac{1}{2}\sum_{i=0}^{n-1}(n-i)(n-i+1)=O(n^3)$$

3. 解法 2

对前面的解法 1 进行优化。当 i 取某个起始下标时，依次求 $j=i,i+1,\cdots,n-1$ 对应的子序列和，实际上这些子序列是相关的。用 Sum($a[i..j]$)表示子序列 $a[i..j]$元素和，初始时置 Sum($a[i..j]$)=0，显然有如下递推关系：

$$\text{Sum}(a[i..j])=\text{Sum}(a[i..j-1])+a[j] \quad \text{当 } j \geqslant i \text{ 时}$$

这样在连续求 $a[i..j]$子序列和($j=i,i+1,\cdots,n-1$)时没有必要使用循环变量为 k 的第 3 重循环，优化后的算法如下：

```
int maxSubSum2(vector<int> & a)                   //解法 2
{   int n=a.size();
    int maxsum=0,cursum;
    for (int i=0;i<n;i++)
    {   cursum=0;
```

```
        for (int j=i;j<n;j++)                    //连续求 a[i..j]子序列元素和 cursum
        {   cursum+=a[j];
            maxsum=max(maxsum,cursum);           //通过比较求最大 maxsum
        }
    }
    return maxsum;
}
```

【算法分析】 在 maxSubSum2(a,n)算法中只有两重循环，所以有：

$$T(n)=\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}1=\sum_{i=0}^{n-1}(n-i)=\frac{n(n+1)}{2}=O(n^2)$$

4. 解法 3

对前面的解法 2 继续优化。maxsum 和 cursum 初始化为 0，用 i 遍历 a，置 cursum+=$a[i]$，也就是说 cursum 累积到 $a[i]$时的元素和，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。

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。

在上述过程中先置 cursum+=$a[i]$，后判断 cursum 的两种情况。a 遍历完毕返回 maxsum 即可。对应的算法如下：

```
int maxSubSum3(vector<int> & a)              //解法 3
{   int n=a.size();
    int maxsum=0,cursum=0;
    for (int i=0;i<n;i++)
    {   cursum+=a[i];
        maxsum=max(maxsum,cursum);           //通过比较求最大的 maxsum
        if(cursum<0)                         //若 cursum<0,最大连续子序列从下一个位置开始
            cursum=0;
    }
    return maxsum;
}
```

【算法分析】 在 maxSubSum3(a,n)算法中只有一重循环，所以设计复杂度为 $O(n)$。

从中看出，尽管仍采用穷举法思路，但可以通过各种优化手段降低算法的时间复杂度。解法 2 的优化点是找出 $a[i..j-1]$和 $a[i..j]$子序列的相关性，解法 3 的优化点是进一步判断 cursum 的两种情况。

思考题： 对于给定的整数序列 a，不仅要求出其中最大连续子序列的和，还需要求出这个具有最大连续子序列和的子序列(给出其起始和终止下标)，如果有多个具有最大连续子序列和的子序列，求其中任意一个子序列。

扫一扫

视频讲解

【例 3-2】 素数个数问题。给定两个均含 n 个正整数的数组 a 和 b，其中整数元素的范围是 2～20000，求出每对 $a[i]$和 $b[i]$(0≤i<n)之间的素数个数(含 $a[i]$和 $b[i]$)。

解法 1： 采用穷举法。设计 isPrime(x)算法判断 x 是否为素数，Count1(a,b)求整数 a～b 中的素数个数。在此基础上设计求解算法 solve(a,b,n)求每对 $a[i]$和 $b[i]$(0≤i<n)之

间的素数个数。对应的算法如下：

```
bool isPrime(int x)                                    //判断 x 是否为素数
{   for (int i=2;i<=(int)sqrt(x);i++)
        if (x%i==0)                                    //x 能够被 i 整除
            return false;
    return true;
}
int Count1(int a,int b)                                //求 a 到 b 的素数个数
{   if (a>b) return 0;
    int cnt=0;
    for(int x=a;x<=b;x++)
        if(isPrime(x))
            cnt++;
    return cnt;
}
void solve1(int a[],int b[],int n)                     //求解算法
{   for(int i=0;i<n;i++)
        printf("   %d-%d 之间的素数个数=%d\n",a[i],b[i],Count1(a[i],b[i]));
}
```

【算法分析】 设最大整数元素为 $m=20000$，isPrime 算法的最坏时间复杂度为 $O(\sqrt{m})$，Count1 算法的最坏时间复杂度为 $O(m\sqrt{m})$，则 solve1 算法的最坏时间复杂度为 $O(mn\sqrt{m})$。

解法 2：对上述穷举法算法进行优化。由于最大整数元素为 $m=20000$，设计一个整数数组 prime，其中 prime[i]表示整数 i 是否为素数(prime[i]=1 表示 i 是素数，prime[i]=0 表示 i 不是素数)，初始时置 prime 的所有元素为 1，采用素数筛选法(若 i 是素数，则 i 的倍数一定不是素数)求出所有的非素数。

再将 prime 转换为前缀和，即将 prime[i]由原来表示整数 i 是否为素数转换为表示小于或等于 i 的素数个数，转换公式如下：

$$\text{prime}[i]+=\text{prime}[i-1]$$

这样 $a\sim b$ 中的素数个数为 prime[b]−prime[$a-1$]。对应的算法如下：

```
int prime[MAXD];                                       //prime[i]=1 表示 i 是素数
void init()                                            //求出 prime 数组
{   for(int i=0;i<=MAXD;i++)
        prime[i]=1;
    prime[0]=prime[1]=0;
    for(int i=2;i<=MAXD;i++)
    {   if(prime[i])                                   //若 i 是素数
        {   for(int j=2*i;j<=MAXD;j+=i)                //则 i 的倍数都不是素数
                prime[j]=0;
        }
    }
}
void solve2(int a[],int b[],int n)                     //求解算法
{   Init();
    for(int i=2;i<=MAXD;i++)                           //prime[i]累计小于或等于 i 的素数个数
        prime[i]+=prime[i-1];
    for(int i=0;i<n;i++)
        printf(" %d-%d 之间的素数个数=%d\n",a[i],b[i],prime[b[i]]-prime[a[i]-1]);
}
```

【算法分析】 同样设最大整数元素为 $m=20000$，Init 算法的时间复杂度为 $O(mk)$，实际上 k 是一个较小数，可以看成 $O(m)$，这样 solve2 算法的时间复杂度为 $O(m+n)$。其中的优化点是利用了前缀和数组，前缀和数组有很多用途，善于利用前缀和数组可以大幅度提高算法的时间性能。

3.1.3 字符串匹配

1. 问题描述

对于两个字符串 s 和 t，若 t 是 s 的子串，返回 t 在 s 中的位置（t 的首字符在 s 中对应的下标），否则返回 -1。

2. BF 算法

BF 算法称为暴力算法，采用穷举法的思路。假设 $s="s_0s_1\cdots s_i\cdots s_{n-1}"$，$t="t_0t_1\cdots t_j\cdots t_{m-1}"$，BF 算法的匹配过程如下。

① 第 0 趟：$i=0/j=0$ 从 s_i/t_j 开始比较，若 $s_i=t_j$，执行 $i{+}{+}/j{+}{+}$ 继续比较，若 j 超界，即 $j\geqslant m$，表示匹配成功，返回 $i-m(0)$；若 $s_i\neq t_j$，本趟匹配失败（称为 s_i/t_j 为失配处），置 $i=i-j+1/j=0$。

② 第 1 趟：$i=1/j=0$ 从 s_i/t_j 开始比较，若 $s_i=t_j$，执行 $i{+}{+}/j{+}{+}$ 继续比较，若 j 超界，即 $j\geqslant m$，表示匹配成功，返回 $i-m(1)$；若 $s_i\neq t_j$，本趟匹配失败，置 $i=i-j+1/j=0$。

③ 第 k 趟：$i=k/j=0$ 从 s_i/t_j 开始比较，若 $s_i=t_j$，执行 $i{+}{+}/j{+}{+}$ 继续比较，若 j 超界，即 $j\geqslant m$，表示匹配成功，返回 $i-m(k)$；若 $s_i\neq t_j$，本趟匹配失败，置 $i=i-j+1/j=0$。

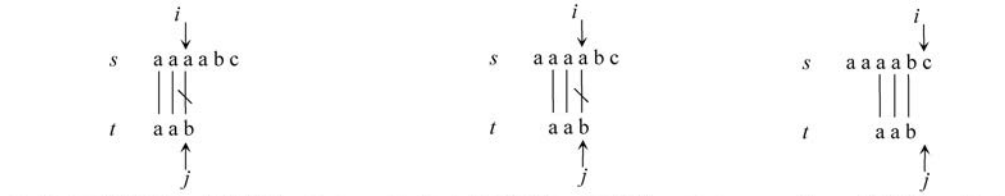
以此类推，若 $k\geqslant n-1$，说明 t 不是 s 的子串，返回 -1。

例如，$s="aaaabc"$，$t="aab"$，$n=6$，$m=3$，两个字符串的匹配过程如图 3.4 所示，在成功时有 $i=5/j=3$，返回 $i-m=2$。图中两个字符之间连接线（含比较不相同的连接线）的条数就是字符比较的次数，共比较 9 次，显然比较次数越多算法的性能越差。

对应的 BF 算法如下：

```
int BF(string s,string t)                    //字符串匹配
{   int n=s.size(),m=t.size();
    int i=0,j=0;
    while (i<n && j<m)
    {   if (s[i]==t[j])                      //比较的两个字符相同时
        {   i++;
            j++;
        }
        else                                 //比较的两个字符不相同时
        {   i=i-j+1;                         //i 回退到原 i 的下一个位置
            j=0;                             //j 从 0 开始
        }
    }
    if (j>=m) return i-m;                    //t 的字符比较完毕表示 t 是 s 的子串
    else return -1;                          //否则表示 t 不是 s 的子串
}
```

【算法分析】 BF 算法的主要时间花费在字符比较上。若 s 和 t 中的字符个数分别为 n 和 m，并且 t 是 s 的子串，最好的情况是从 $s[0]$ 的比较（第 0 趟）成功，此时 $T(n,m)=$



(a) 从s_0/t_0开始比较，失配处为$i=2/j=2$　(b) 从s_1/t_0开始比较，失配处为$i=3/j=2$　(c) 从s_2/t_0开始比较，成功

图 3.4　采用 BF 算法实现字符串匹配的过程

$O(m)$；最坏的情况是 s 末尾的 m 个字符为 t，前面 $n-m$ 趟比较均失败，并且每次需要比较 m 次，此时 $T(n,m)=O(nm)$。容易求出平均时间复杂度也是 $O(nm)$。

3. 从 BF 算法到 KMP 算法

前面讨论的 BF 算法性能低下，原因是遍历 s 的 i 在当前趟失败时会回退。KMP 算法是 BF 算法的优化，那么优化点在哪里呢？优化点就是保证 i 不回退（i 要么后移，要么停下来不动），那么是如何做到这一点的呢？

仍以 $s=$"aaaabc"，$t=$"aab"为例，BF 算法的第 0 趟的匹配如图 3.4(a)所示，失配处为 $i=2/j=2$，说明"s_0s_1"="t_0t_1"一定成立，显然"s_1"="t_1"，再观察 t 中失配处的 t_2，发现它的前面有一个子串"t_1"（这个子串最多从 t_1 开始且必须以 t_{j-1} 结尾）与 t 开头的子串相同（这个子串必须从 t_0 开始最多以 t_{j-2} 结尾），即"t_1"="t_0"，这样可以推出"s_1"="t_0"。也就是说，尽管第 0 趟匹配失败了，但同时得到了"s_1"="t_0"的信息。

现在再看 BF 算法，第 0 趟匹配失败后下一趟（即第 1 趟）从 s_1/t_0 开始重复匹配，实际上前面已经推出"s_1"="t_0"，所以下一趟没有必要做 s_1/t_0 的比较，直接从 s_2/t_1 开始比较，如图 3.5 所示，这样可能减少匹配的趟数，即使没有减少匹配的趟数，也会减少字符比较的次数，从而通过字符串匹配的性能。

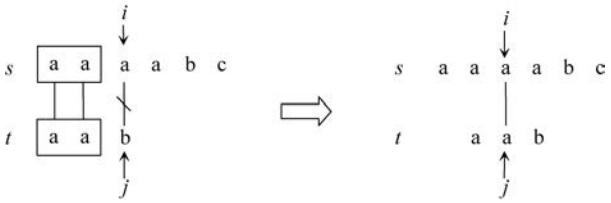


图 3.5　下一趟从 s_2/t_1 开始比较

从中看出，在字符串匹配之前提取出 t 中形如"s_1"="t_0"的信息是优化的关键，由于每个位置 j 都有这样的信息，即位置 j 的前面最多有多少个字符与 t 开头的字符相同，为此用 next 存放这样的信息，next[j]=k 表示 t_j 前面最多有 k 个字符与 t 开头的字符相同。求模式串 t 的 next 数组的公式如下：

$$\text{next}[j]=\begin{cases}-1 & \text{当 } j=0 \text{ 时}\\ \max\{k \mid 0<k<j \text{ 且 } "t_0t_1\cdots t_{k-1}"="t_{j-k}t_{j-k+1}\cdots t_{j-1}"\} & \text{当此集合非空时}\\ 0 & \text{其他情况}\end{cases}$$

next 数组的求解过程如下：

(1) next[0]=−1（$j=0$，属于特殊情况或者说 j 到头了）。

（2）next[1]=0（$j=1$，t 在 1～$j-1$ 的位置上没有字符）。

（3）如果 next[j]=k，表示有"$t_0t_1\cdots t_{k-1}$"="$t_{j-k}t_{j-k+1}\cdots t_{j-1}$"（$j-k\geqslant 1$，或者说 $j>k$）：

① 若 $t_k=t_j$，即有"$t_0t_1\cdots t_{k-1}t_k$"="$t_{j-k}t_{j-k+1}\cdots t_{j-1}t_j$"，显然有 next[$j+1$]=$k+1$。

② 若 $t_k\neq t_j$，说明 t_j 之前不存在长度为 next[j]+1 的子串和 t_0 开头的子串相同，那么是否存在一个长度较短的子串和开头字符起的子串相同呢？设 k'=next[k]（回退），则下一步应该将 t_j 与 $t_{k'}$ 比较：若 $t_j=t_{k'}$，则说明 t_j 之前存在长度为 next[k']+1 的子串和 t_0 开头的子串相同；否则以此类推找更短的子串，直到不存在可匹配的子串，此时置 next[$j+1$]=0。所以当 $t_k\neq t_j$ 时置 k=next[k]。

对应的求模式串 t 的 next 数组的算法如下：

```
void getnext(string& t,vector<int>& next)
{   int j,k;
    j=0;k=-1;                          //j 遍历 t,k 记录 t[j]之前与 t 开头相同的字符个数
    next[0]=k;                         //设置 next[0]值
    while (j<t.size()-1)               //求 t 所有位置的 next 值
    {   if (k==-1||t[j]==t[k])         //k 为-1 或比较的字符相等时
        {   j++;k++;                   //j、k 依次移到下一个字符
            next[j]=k;                 //设置 next[j]为 k
        }
        else k=next[k];                //k 就近回退
    }
}
```

例如，t="aab"，$m=3$，求 t 的 next 数组的结果如表 3.1 所示。

表 3.1　求 t="aab"的 next 数组

j	0	1	2
$t[j]$	a	a	b
next[j]	-1	0	1

在求出模式串 t 的 next 数组后，实现两个字符串 s 和 t 匹配的 KMP 算法的过程是，用 i 和 j（均从 0 开始）分别遍历目标串 s 和模式串 t。

① 若 $j=-1$，这是一种特殊情况，由于只有 next[0]=-1，说明此前执行了 j=next[0]，也就是说失配处是 s_i/t_0，下一趟只能从 s_{i+1}/t_0 比较开始，即 i 和 j 分别增 1 进入下一趟。

② 若 $s_i=t_j$，两个字符相同，则 i 和 j 分别增 1 继续比较。

③ 否则失配处为 s_i/t_j，i 不变，j 退回到 j=next[j]的位置（即模式串右滑），再比较 s_i 和 t_j，若相等，i、j 各增 1，否则 j 再次退回到下一个 j=next[j]的位置（即模式串 t 右滑 $j-k$ 个位置，这里 k=next[j]），以此类推，直到出现下列两种情况之一：一种情况是 j 退回到某个 j=next[j]位置时有 $s_i=t_j$，则采用②的处理方式，另外一种情况是 $j=-1$，则采用①的处理方式。

例如，s="aaaabc"，t="aab"，采用 KMP 算法的匹配过程如图 3.6 所示，共比较 7 次。

对应的 KMP 算法如下：

```
int KMP(string s,string t)                    //KMP 算法
{   int n=s.size();
    int m=t.size();
    vector<int> next(m,-1);
```

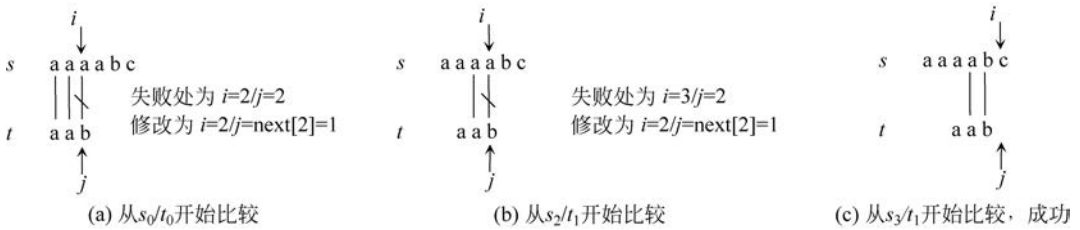


图 3.6 采用 KMP 算法实现字符串匹配的过程

```
    getnext(t,next);
    int i=0,j=0;
    while(i<n && j<m)
    {   if (j==-1 || s[i]==t[j])          //j 为-1 或者比较的字符相同,i 和 j 同时向后移动
        {   i++;
            j++;
        }
        else j=next[j];                   //比较的字符不相同,j 寻找之前匹配的位置
    }
    if (j>=m) return i-m;                 //j 超界说明 t 是 s 的子串
    else return -1;                       //否则说明 t 不是 s 的子串
}
```

【算法分析】 设主串 s 的长度为 n，子串 t 的长度为 m，在 KMP 算法中求 next 数组的时间复杂度为 $O(m)$，在后面的匹配中因目标串 s 的下标 i 不减(即不回溯)，比较次数可记为 n，所以 KMP 算法的平均时间复杂度为 $O(n+m)$。

【例 3-3】 有两个字符串 s 和 t，设计一个算法求 t 在 s 中出现的次数。例如，s="abababa"，t="aba"，则 t 在 s 中出现两次(不考虑子串重叠的情况)。

扫一扫

视频讲解

解法 1：采用 BF 算法的思路。用 cnt 记录 t 在 s 中出现的次数(初始时为 0)。当在 s 中找到 t 的一次出现时置 cnt++，此时 $j=t$ 的长度，i 指向 s 中本次出现 t 子串的下一个字符，所以为了继续查找 t 子串的下一次出现，只需要置 $j=0$ 即可。对应的算法如下：

```
int Count1(string s,string t)             //解法 1
{   int cnt=0;                            //累计出现次数
    int n=s.size(),m=t.size();
    int i=0,j=0;
    while (i<n && j<m)
    {   if (s[i]==t[j])                   //比较的两个字符相同时
        {   i++;
            j++;
        }
        else                              //比较的两个字符不相同时
        {   i=i-j+1;                      //i 回退
            j=0;                          //j 从 0 开始
        }
        if(j>=m)
        {   cnt++;                        //出现次数增 1
            j=0;                          //j 从 0 开始继续比较
        }
    }
    return cnt;
}
```

解法 2：采用 KMP 算法的思路。先求出 t 的 next 数组(同前面的 getnext 算法)，在 s 中找到 t 的一次出现时置 cnt++，同样 i 不变只需要置 $j=0$ 即可。对应的算法如下：

```
int Count2(string s,string t)                    //解法 2
{   int cnt=0;                                   //累计出现次数
    int n=s.size(),m=t.size();
    vector<int> next(m,-1);
    getnext(t,next);
    int i=0,j=0;
    while(i<n)
    {   if (j==-1||s[i]==t[j])                   //j=-1 或者两个字符相同,i 和 j 同时向后移动
        {   i++;
            j++;
        }
        else j=next[j];                          //两个字符不相同,j 寻找之前匹配的位置
        if (j>=m)                                //成功匹配一次
        {   cnt++;
            j=0;                                 //匹配成功后 t 从头开始比较
        }
    }
    return cnt;
}
```

扫一扫

视频讲解

思考题：两个字符串 s 和 t，设计一个算法求 t 在 s 中重叠出现的次数。例如，s="aaaaa"，t="aa"，则 t 在 s 中出现 4 次(考虑子串重叠的情况)。

3.1.4 实战——查找单词(POJ1501)

1. 问题描述

给定一个字母矩阵，在其中查找单词的位置。

输入格式：输入的第一行为正方形的字母矩阵的长度 n(以字符为单位，$1\leqslant n\leqslant 100$)，接下来的 n 行输入字母矩阵，每行仅包含 n 个大写字母。随后是一个单词列表，每个单词占一行，最多 100 个单词，每个单词不超过 100 个字符，并且只包含大写字母。输入的最后一行包含一个 0 字符。

输出格式：在字母矩阵中查找单词列表中的每个单词，如果一个单词中的所有字母都可以在字母矩阵中的单个(单向)水平、垂直或对角线中找到，则该单词查找成功。单词不会出现环绕，但在水平或者对角线上可以从右到左。若一个单词查找成功(测试数据保证每个单词最多只能查找成功一次)，在一行中输出其在字母矩阵中第一个和最后一个字母的坐标，坐标是以逗号分隔的整数对，其中第一个整数指定行号，第二个整数指定列号(行、列号均从 1 开始)，两组坐标之间以一个空格分隔。如果一个单词没有找到，则输出"Not found"字符串。

输入样例：

```
5
EDEEE
DISKE
ESEEE
```

```
ECEEE
EEEEE
DISC
DISK
DISP
0
```

输出样例：

```
1,2 4,2
2,1 2,4
Not found
```

2. 问题求解

依题意，用二维字符数组 map 存放字母矩阵，在其中查找单词 str(长度为 len)时只能依次按 8 个方位(用 dir 数组表示 8 个方位的偏移量)搜索，没有回退，所以采用穷举法，i 和 j 枚举行、列坐标，d 枚举 8 个方位，k 遍历 str，其中 str[k]字符的坐标是(i+dir[d][0]*k，j+dir[d][1]*k)，若 str 的全部字符均查找到，即 k=len−1 成立，则说明查找成功。所有情况下都没有找到 str 说明查找失败。最后输出结果。对应的程序如下：

```
#include<iostream>
#include<cstring>
#define MAXN 105
using namespace std;
int dir[8][2]={{0,1},{0,-1},{1,0},{-1,0},{-1,-1},{-1,1},{1,-1},{1,1}};
char map[MAXN][MAXN];
char str[MAXN];
int main()
{   int n,i,j,x,y;
    scanf("%d",&n);
    for(i=0;i<n;i++)                                    //输入字母矩阵
        scanf("%s",map[i]);
    while(scanf("%s",str))                              //输入若干个单词
    {   if(str[0]=='0') break;                          //输入"0"结束
        int len=strlen(str);
        bool flag=false;
        for(i=0;i<n;i++)                                //穷举每个行
        {   for(j=0;j<n;j++)                            //穷举每个列
            {   for(int d=0;d<8;d++)                    //穷举每个方位
                {   for(int k=0;k<len;k++)              //遍历 str 单词
                    {   x=i+dir[d][0]*k;                //求出 str[k]字母的坐标(x,y)
                        y=j+dir[d][1]*k;
                        if(x<0 || x>=n || y<0 || y>=n || map[x][y]!=str[k])
                            break;                      //坐标超界或者不相同退出 k 的循环
                        if(k==len-1) flag=true;         //查找成功，置 flag 为 true
                    }
                    if(flag) break;
                }
                if(flag) break;
            }
            if(flag) break;
        }
```

```
            if(flag)                                    //输出查找结果
                printf("%d,%d %d,%d\n",i+1,j+1,x+1,y+1);
            else
                printf("Not found\n");
        }
        return 0;
    }
```

上述程序提交时通过，执行时间为 16ms，内存消耗为 156KB。

3.2 归 纳 法

3.2.1 归纳法概述

1. 什么是数学归纳法

谈到归纳法很容易想到数学归纳法，数学归纳法是一种数学证明方法，典型地用于确定一个表达式在所有自然数范围内是成立的，分为两种。

第一数学归纳法的原理：若$\{P(1),P(2),P(3),P(4),\cdots\}$是命题序列且满足以下两个性质，则所有命题均为真。

① $P(1)$为真。

② 任何命题均可以从它的前一个命题推导得出。

例如，采用第一数学归纳法证明 $1+2+\cdots+n=n(n+1)/2$ 成立的过程如下：

当 $n=1$ 时，左式$=1$，右式$=(1\times2)/2=1$，左、右两式相等，等式成立。

假设当 $n=k-1$ 时等式成立，有 $1+2+\cdots+(k-1)=k(k-1)/2$。

当 $n=k$ 时，左式$=1+2+\cdots+(k-1)+k=k(k-1)/2+k=k(k+1)/2$，等式成立。即证。

第二数学归纳法的原理：若$\{P(1),P(2),P(3),P(4),\cdots\}$是满足以下两个性质的命题序列，则对于其他自然数，该命题序列均为真。

① $P(1)$为真。

② 任何命题均可以从它的前面所有命题推导得出。

用数学归纳法进行证明主要是两个步骤：

① 证明当取第一个值时命题成立。

② 假设当前命题成立，证明后续命题也成立。

数学归纳法的独到之处便是运用有限个步骤就能证明无限多个对象，而实现这一目的的工具就是递推思想。第①步是证明归纳基础成立，归纳基础成为后面递推的出发点，没有它递推成了无源之水；第②步是证明归纳递推成立，借助该递推关系，命题成立的范围就能从开始向后面一个数一个数地无限传递到以后的每一个正整数，从而完成证明，因此递推是实现从有限到无限飞跃的关键。

【例 3-4】 给定一棵非空二叉树，采用数学归纳法证明如果其中有 n 个叶子结点，则双分支结点的个数恰好为 $n-1$，即 $P(n)=n-1$。

证明： 当 $n=1$ 时，这样的二叉树恰好只有一个结点，该结点既是根结点又是叶子结点，

没有分支结点，则 $P(1)=0$ 成立。

假设叶子结点的个数为 $k-1$ 时成立，即 $P(k-1)=(k-1)-1=k-2$。由二叉树的结构可知，想要在当前的二叉树中增加一个叶子结点，对其中某种类型的结点的操作如下。

① 双分支结点：无法增加孩子结点，不能达到目的。

② 单分支结点：可以增加一个孩子结点(为叶子结点)，此时该单分支结点变为双分支结点，也就是说叶子结点和双分支结点均增加一个，这样 $P(k)=P(k-1)+1=k-2+1=k-1$，结论成立。

③ 叶子结点：增加一个孩子结点，总的叶子结点个数没有增加，不能达到目的。

④ 叶子结点：增加两个孩子结点(均为叶子结点)，此时该叶子结点变为双分支结点，也就是说叶子结点和双分支结点均增加一个，这样 $P(k)=P(k-1)+1=k-2+1=k-1$，结论成立。

凡是能够达到目的的操作都会使结论成立，根据第一数学归纳法的原理，问题即证。

2. 什么是归纳法

从广义上讲，归纳法是人们在认识事物的过程中所使用的一种思维方法，通过列举少量的特殊情况，经过分析和归纳推理寻找出基本规律。归纳法要比枚举法更能反映问题的本质，但是从一个实际问题中总结归纳出基本规律并不是一件容易的事情，而且归纳过程通常也没有固定的规则可供遵循。归纳法包含不完全归纳法和完全归纳法，不完全归纳法是根据事物的部分特殊事例得出的一般结论的推理方法，即从特殊出发，通过实验、观察、分析、综合和抽象概括出一般性结论的一种重要方法。完全归纳法是根据事物的所有特殊事例得出的一般结论的推理方法。

在算法设计中归纳法常用于建立数学模型，通过归纳推理得到求解问题的递推关系，也就是采用递推关系表达寻找出的基本规律，从而将复杂的运算化解为若干重复的简单运算，以充分发挥计算机擅长重复处理的特点。

在应用归纳法时一般用 n 表示问题规模(n 为自然数)，并且具有这样的递推性质：能从已求得的问题规模为 $1\sim n-1$ 或者 $n/2$ 等的一系列解构造出问题规模为 n 的解。前者均称为“小问题”，后者称为“大问题”，大、小问题的解法相似，只是问题规模不同。

利用归纳法产生递推关系的基本流程如下：

① 按推导问题方向研究最初、最原始的若干问题。

② 按推导问题方向寻求问题间的转换规律，即递推关系，使问题逐次转化成较低层级或简单的且能解决的或已解决的问题。

根据推导问题的方向分为顺推法和逆推法两种。所谓顺推法是从已知条件出发逐步推算出要解决问题的结果，如图 3.7(a)所示。所谓逆推法是从已知问题的结果出发逐步推算出问题的开始条件，如图 3.7(b)所示。

前面讨论的数学归纳法和归纳法有什么关系呢？实质上数学归纳法与归纳法没有逻辑联系，按著名数学家波利亚的说法是数学归纳法这个名字是随便起的。数学归纳法虽不是归纳法，但它与归纳法有着一定程度的关联，在结论的发现过程中，往往先通过对大量个别事实的观察，通过不完全归纳法归纳形成一般性的结论，最终利用数学归纳法对结论的正确性予以证明。后面通过几个示例讨论归纳法在算法设计中的应用。

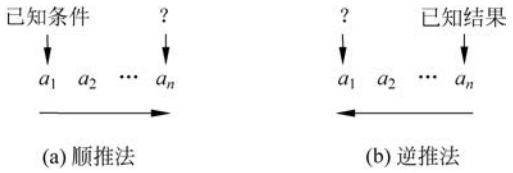


图 3.7　顺推法和逆推法

3.2.2　直接插入排序

1. 问题描述

有一个整数序列 $R[0..n-1]$，采用直接插入排序实现 R 的递增有序排序。直接插入排序的过程是 i 从 1 到 $n-1$ 循环，将 $R[i]$有序插入 $R[0..i-1]$中。

2. 问题求解

采用不完全归纳法产生直接插入排序的递推关系。例如 $R=(2,5,4,1,3)$，这里 $n=5$，用[ ]表示有序区，各趟的排序结果如下：

```
初始:  ([2],5,4,1,3)
i=1:   ([2,5],4,1,3)
i=2:   ([2,4,5],1,3)
i=3:   ([1,2,4,5],3)
i=4:   ([1,2,3,4,5])
```

设 $f(R,i)$用于实现 $R[0..i]$(共 $i+1$ 个元素)的递增排序，它是大问题，则 $f(R,i-1)$实现 $R[0..i-1]$(共 i 个元素)的排序，它是小问题。对应的递推关系如下：

$f(R,i)$≡不做任何事情　　当 $i=0$ 时
$f(R,i)$≡$f(R,i-1)$；将 $R[i]$有序插入 $R[0..i-1]$中；　　其他

显然 $f(R,n-1)$用于实现 $R[0..n-1]$的递增排序。这样采用不完全归纳法得到的结论(直接插入排序的递推关系)是否正确呢？对于排序元素个数 n 采用数学归纳法证明如下：

① 证明归纳基础成立。当 $n=1$ 时直接返回，由于此时 R 中只有一个元素，它是递增有序的，所以结论成立。

② 证明归纳递推成立。假设 $n=k$ 时成立，也就是说 $f(R,k-1)$用于实现 $R[0..k-1]$的递增排序。当 $n=k+1$ 时对应 $f(R,k)$，先调用 $f(R,k-1)$将 $R[0..k-1]$排序，再将 $R[k]$有序插入 $R[0..k-1]$中，这样 $R[0..k]$就变成递增有序序列了，所以 $f(R,k)$实现 $R[0..k]$的递增排序，结论成立。

根据第一数学归纳法的原理，问题即证。按照上述直接插入排序的递推关系得到对应的算法如下：

```
void Insert(vector<int> &R,int i)                    //插入操作: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];                                 //将关键字大于 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]
}
void InsertSort1(vector<int> &R)                      //直接插入排序
{   int n=R.size();
    for (int i=1;i<n;i++)
    {   if (R[i]<R[i-1])                              //反序时
            Insert(R,i);
    }
}
```

在实际中有一些采用不完全归纳法得到的结论明显是正确的，或者已经被人们证明是正确的，可以在算法设计中直接使用这些结论。

3.2.3 楼梯问题

1. 问题描述

一个楼梯共有 n 个台阶，规定每一步只能跨一个或两个台阶。设计一个算法求登上第 n 级台阶有多少种不同走法。

2. 问题求解

采用归纳法中的顺推法。设 $f(n)$表示登上第 n 级台阶的不同的走法数。

① 当 $n=1$ 时，只有跨一个台阶的一种走法，即 $f(1)=1$。

② 当 $n=2$ 时，可以走两步每步跨一个台阶，也可以走一步跨两个台阶，这样共有两种走法，即 $f(2)=2$。

③ 当 $n=3$ 时，考虑第一步，第一步跨两个台阶，剩下一个台阶(剩下一个台阶的不同走法数为 $f(1)$)，对应的不同走法数为 $f(1)$；第一步跨一个台阶，剩下两个台阶(剩下两个台阶的不同走法数为 $f(2)$)，对应的不同走法数为 $f(2)$。采用加法原理有 $f(3)=f(1)+f(2)$。

④ 当 $n=4$ 时，考虑第一步，第一步跨两个台阶，剩下两个台阶，对应的不同走法数为 $f(2)$；第一步跨一个台阶，剩下 3 个台阶，对应的不同走法数为 $f(3)$；采用加法原理有 $f(4)=f(2)+f(3)$。

以此类推，得到如下递推关系：

$$
\begin{aligned}
&f(1)=1\\
&f(2)=2\\
&f(n)=f(n-2)+f(n-1) \qquad \text{当 } n>2 \text{ 时}
\end{aligned}
$$

其中 $f(n)$是大问题，$f(n-1)$和 $f(n-2)$是两个小问题。实际上该模型是斐波那契数列的变形。对应的算法如下：

```
int Count(int n)                                      //求登上第 n 级台阶的不同的走法数
{   int a=1,b=2,c;                                    //a、b、c 分别对应 f(n-2)、f(n-1)、f(n)
    if(n==1)
        return a;
    else if(n==2)
        return b;
    else
```

```
    {   for(int i=3;i<=n;i++)
        {   c=a+b;
            a=b;
            b=c;
        }
        return c;
    }
}
```

3.2.4 猴子摘桃子问题

1. 问题描述

猴子第 1 天摘下若干个桃子，当即吃了一半又一个。第 2 天又把剩下的桃子吃了一半又一个，以后每天都吃前一天剩下的桃子的一半又一个，到第 10 天猴子想吃的时候只剩下一个桃子。给出求猴子第 1 天一共摘了多少桃子的递归模型。

2. 问题求解

采用归纳法中的逆推法。设 $f(i)$表示第 i 天的桃子数，假设第 n(题目中 $n=10$)天只剩下一个桃子，即 $f(n)=1$。另外题目中隐含有前一天的桃子数等于后一天桃子数加 1 的两倍：

$f(10)=1$
$f(9)=2(f(10)+1)$
$f(8)=2(f(9)+1)$
…

即 $f(i)=2(f(i+1)+1)$。这样得到递推关系如下：

$f(i)=1$　　当 $i=n$ 时
$f(i)=2(f(i+1)+1)$　　其他

其中 $f(i)$是大问题，$f(i+1)$是小问题。最终结果是求 $f(1)$。对应的算法如下：

```
int peaches(int n)              //第 n 天桃子数为 1,求第 1 天桃子数
{   int ans=1;
    for(int i=n-1;i>=1;i--)
        ans=2*(ans+1);
    return ans;
}
```

扫一扫

视频讲解

3.2.5 实战——骨牌铺方格(HDU2046)

1. 问题描述

在一个 $2\times n$ 的长方形方格中，用一个 1×2 的骨牌铺满。输入 n，输出铺放方案的总数。例如 $n=3$ 时为 2×3 方格，骨牌的铺放方案有如图 3.8 所示的 3 种。

输入格式：输入数据由多行组成，每行包含一个整数 n，表示该测试实例的长方形方格的规格是 $2\times n(0<n\leqslant 50)$。

输出格式：对于每个测试实例，请输出铺放方案的总数，每个实例的输出占一行。

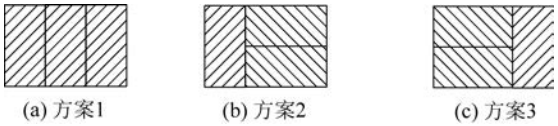


图 3.8　$n=3$ 时的 3 种铺放方案

输入样例：

```
1
3
2
```

输出样例：

```
1
3
2
```

2. 问题求解

设 $f(n)$表示用 1×2 的骨牌铺满 $2\times n$ 的长方形方格的铺放方案总数。

当 $n=1$ 时，用一块 1×2 的骨牌铺满，即 $f(1)=1$。

当 $n=2$ 时，用两块 1×2 的骨牌横向或者纵向铺满，即 $f(2)=2$。

当 $n>2$ 时，$2\times n$ 的长方形方格可以看成由高度为 2 的 n 个方格组成，编号依次是 $1\sim n$，铺放分为如下情况：

(1) 先铺好方格 1，剩下 $2\sim n$ 共 $n-1$ 个方格有 $f(n-1)$种铺放方案，如图 3.9(a)所示，采用乘法原理，情况 1 的铺放方案总数$=1*f(n-1)=f(n-1)$。

(2) 先铺好方格 1 和方格 2，剩下 $3\sim n$ 共$n-2$ 个方格有 $f(n-2)$种铺放方案。前面两个方格对应两种铺放方案：

① 一种如图 3.9(b)所示，对应铺放方案总数$=1*f(n-2)=f(n-2)$，但该铺放方案包含在情况 1 中。

② 另外一种如图 3.9(c)所示，对应铺放方案总数$=1*f(n-2)=f(n-2)$，该铺放方案没有包含在情况 1 中。

采用加法原理，铺放方案总数 $f(n)=f(n-1)+f(n-2)$。

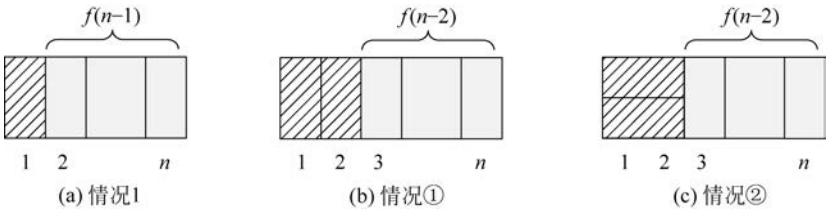


图 3.9　各种铺放情况

合并起来得到如下递推关系：

$f(1)=1$
$f(2)=2$
$f(n)=f(n-1)+f(n-2)$　　当 $n>2$ 时

从中看出，上述递推关系与3.2.3节的楼梯问题完全相同，可以采用如下求解算法（由于 n 的最大值可以是50，$f(n)$必须采用long long数据类型）：

```
#include<iostream>
using namespace std;
long long Count(int n)                    //求铺放方案的总数
{   long long a=1,b=2,c;                  //a、b、c分别对应f(n-2)、f(n-1)、f(n)
    if(n==1)
        return a;
    else if(n==2)
        return b;
    else
    {   for(int i=3;i<=n;i++)
        {   c=a+b;
            a=b;
            b=c;
        }
        return c;
    }
}
int main()
{   int n;
    while (~scanf("%d", &n))
        printf ("%lld\n",Count(n));
    return 0;
}
```

上述程序提交通过，执行时间为31ms，内存消耗为1732KB。可以进一步提高速度，定义一个数组 a（大小为55），$a[i]$存放 $f(i)$，先求出 a 中的所有元素，再对于每个测试实例 n 直接输出 $a[n]$即可，对应的程序如下：

```
#include<iostream>
using namespace std;
int main()
{   int n;
    long long a[55]={0,1,2};
    for (int i=3;i<=51;i++)
        a[i]=a[i-1]+a[i-2];
    while (~scanf("%d", &n))
        printf ("%lld\n",a[n]);
    return 0;
}
```

上述程序提交通过，执行时间为0ms，内存消耗为1744KB。

3.3 迭 代 法

3.3.1 迭代法概述

迭代法也称辗转法，是一种不断用变量的旧值推出新值的过程。通过让计算机对一组

指令(或一定步骤)进行重复执行,在每次执行这组指令(或这些步骤)时都从变量的原值推出它的一个新值。

如果说归纳法是一种建立求解问题数学模型的方法,则迭代法是一种算法实现技术。一般先采用归纳法产生递推关系,在此基础上确定迭代变量,再对迭代过程进行控制,基本的迭代法算法框架如下:

```
void Iterative()
{   迭代变量赋初值;
    while(迭代条件成立)
    {   根据递推关系式由旧值计算出新值;
        新值取代旧值,为下一次迭代做准备;
    }
}
```

实际上 3.2 节中所有的算法均采用迭代法实现。

如果一个算法已经采用迭代法实现,那么如何证明算法的正确性呢?由于迭代法算法包含循环,对循环的证明引入循环不变量的概念。所谓**循环不变量**是指在每轮迭代开始前后要操作的数据必须保持的某种特性(比如在直接插入排序中,排序表前面部分必须是有序的)。循环不变量是进行循环的必备条件,因为它保证了循环进行的有效性,有助于用户理解算法的正确性。如图 3.10 所示,对于循环不变量必须证明它的 3 个性质。

初始化:在循环的第一轮迭代开始之前应该是正确的。

保持:如果在循环的第一轮迭代开始之前正确,那么在下一次迭代开始之前它也应该保持正确。

终止:当循环结束时,循环不变量给了用户一个有用的性质,它有助于表明算法是正确的。

这里的推理与数学归纳法相似,在数学归纳法中要证明某一性质是成立的,必须首先证明归纳基础成立,这里就是证明循环不变量在第一轮迭代开始之前是成立的。证明循环不变量在各次迭代之间保持成立,就类似于在数学归纳法中证明归纳递推成立。循环不变量的第三项性质必须成立,与数学归纳法不同,在归纳法中归纳步骤是无穷地使用,在循环结束时才终止"归纳"。

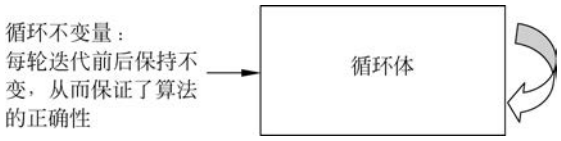


图 3.10 利用循环不变量证明算法的正确性

【例 3-5】 采用循环不变量方法证明 3.2.2 节中直接插入排序算法的正确性。

证明:在直接插入排序算法中循环不变量为 $R[0..i-1]$是递增有序的。

初始化:循环时 i 从 1 开始,循环之前 $R[0..0]$只有一个元素,显然是有序的,所以循环不变量在循环开始之前是成立的。

保持:需要证明每一轮循环都能使循环不变量保持成立。对于 $R[i]$排序的这一趟,之前 $R[0..i-1]$是递增有序的。

① 如果 $R[i]\geqslant R[i-1]$，即正序，则该趟结束，结束后循环不变量 $R[0..i]$显然是递增有序的。

② 如果 $R[i]<R[i-1]$，即反序，则在 $R[0..i-1]$中从后向前找到第一个 $R[j]\leqslant R[i]$，将 $R[j+1..i-1]$均后移一个位置，并且将原 $R[i]$放在 $R[j+1]$位置，这样结束后循环不变量 $R[0..i]$显然也是递增有序的。

终止：循环结束时 $i=n$，在循环不变量中用 i 替换 n，就有 $R[0..n-1]$包含原来的全部元素，但现在已经排好序了，也就是说循环不变量也是成立的。

这样就证明了上述直接插入排序算法的正确性。

后面的重点放在了迭代法算法设计上而不是算法的正确性证明上，通过几个经典应用进行讨论。

3.3.2 简单选择排序

1. 问题描述

有一个整数序列 $R[0..n-1]$，采用简单选择排序实现 R 的递增有序排序。简单选择排序的过程是 i 从 0 到 $n-2$ 循环，$R[0..i-1]$是有序区，$R[i..n-1]$是无序区，并且前者的所有元素均小于或等于后者的任意元素，在 $R[i..n-1]$无序区中通过简单比较找到最小元素 $R[\text{minj}]$，通过交换将其放在 $R[i]$位置。

2. 问题求解

采用不完全归纳法产生简单选择排序的递推关系。例如 $R=(2,5,4,1,3)$，这里 $n=5$，用[]表示有序区，各趟的排序结果如下：

```
初始:  ([]2,5,4,1,3)
i=0:   ([1],5,4,2,3)
i=1:   ([1,2],4,5,3)
i=2:   ([1,2,3],5,4)
i=3:   ([1,2,3,4],5)
```

设 $f(R,i)$用于实现 $R[i..n-1]$(共 $n-i$ 个元素)的递增排序，它是大问题，则 $f(R,i+1)$实现 $R[i+1..n-1]$(共 $n-i-1$ 个元素)的排序，它是小问题。对应的递推关系如下：

$$
\begin{aligned}
&f(R,i) \equiv \text{不做任何事情} && \text{当 } i=n-1 \text{ 时}\\
&f(R,i) \equiv \text{在 } R[i..n-1] \text{中选择最小元素交换到 } R[i] \text{位置}; && \text{否则}\\
&\qquad\quad f(R,i+1);
\end{aligned}
$$

显然 $f(R,0)$用于实现 $R[0..n-1]$的递增排序。对应的算法如下：

```
void Select(vector<int> & R,int i)          //选择操作：在 R[i..n-1]中选择最小元素交换到 R[i]
{   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
}
```

```
void SelectSort1(vector<int> & R)//迭代法：简单选择排序
{   int n=R.size();
    for (int i=0;i<n-1;i++)//进行 n-1 趟排序
        Select(R,i);
}
```

3.3.3 求多数元素

1. 问题描述

给定一个含 $n(n\geqslant 1)$ 个正整数的序列 R，求其中的一个多数元素。所谓多数元素是指出现次数大于 $n/2$ 次的元素，如果 R 中存在多数元素则返回该元素，否则返回 -1。例如 $R=\{1,2,1\}$，返回 1；若 $R=\{1,2,1,2\}$，返回 -1。

2. 问题求解

先考虑 R 中存在多数元素的情况。可以遍历 R 求出每个不同整数的出现次数(采用哈希表存放)，找到出现次数大于 $n/2$ 的元素即可，对应的时间复杂度和空间复杂度均为 $O(n)$，下面讨论一种时间复杂度为 $O(n)$、空间复杂度为 $O(1)$ 的方法。

通过观察可以归纳出这样的结论：删除序列 R 中任意两个不同的元素，若存在多数元素，则删除后多数元素依然存在且不变。这个结论是很容易证明的，假设 R 中存在多数元素 x，即 x 出现的次数 c 大于 $n/2$，现在删除 R 中任意两个不同的元素得到 R_1(含 $n-2$ 个元素)：

① 若其中两个元素均不是 x，则 x 在 R_1 中出现的次数仍然为 c，由于 $c>n/2>(n-2)/2$，所以 x 是 R_1 中的多数元素。

② 若两个元素中有一个是 x，则 x 在 R_1 中出现的次数仍然为 $c-1$，由于 $(c-1)/(n-2)>(n/2-1)/(n-2)=1/2$，也就是说 x 在 R_1 中出现的次数超过一半，所以 x 是 R_1 中的多数元素。

既然上述结论成立，就可以从头开始，设候选多数元素为 $c=R[0]$，计数器 cnt 表示 c 出现的次数(初始为 1)，i 从 1 开始遍历 R，若两个元素($R[i]$,c)相同，cnt 增 1，否则 cnt 减 1，相当于删除这两个元素($R[i]$删除一次，c 也只删除一次)，如果 cnt 为 0，说明前面没有找到多数元素，从 $R[i+1]$开始重复查找，即重置 $c=R[i+1]$，cnt=1。

找到候选多数元素 c 后，当 R 中存在多数元素时，c 就是所求结果，若 R 中不存在多数元素，c 不是正确的结果，所以还需要遍历 R 求出 c 出现的次数 cnt，若 cnt$>n/2$，则 c 是多数元素，返回 c，否则返回 -1。对应的迭代算法如下：

```
int candidate(vector<int> &R)              //求候选多数元素
{   int c=R[0],cnt=1;
    int i=1;
    while(i<R.size())
    {   if(R[i]==c)                        //选择两个元素(R[i],c)
            cnt++;                         //相同时累加
        else
            cnt--;                         //不相同时递减 cnt,相当于删除这两个元素
        if(cnt==0)                         //cnt 为 0 时对剩余元素从头开始查找
        {   i++;
            c=R[i];
```

```
                cnt++;
            }
            i++;
        }
        return c;
    }
    int majority(vector<int> & R)//迭代法：求多数元素
    {   if(R.size()==0 || R.size()==1)
            return -1;
        int c=candidate(R);
        int cnt=0;
        for(int i=0; i<R.size(); i++)
            if(R[i]==c) cnt++;
        if(cnt>R.size()/2)
            return c;
        else
            return -1;
    }
```

扫一扫

视频讲解

3.3.4 求幂集

1. 问题描述

给定正整数 $n(n\geqslant 1)$，给出求{1～n}的幂集的递推关系和迭代算法。例如，$n=3$ 时{1,2,3}的幂集合为{{},{1},{2},{1,2},{3},{1,3},{2,3},{1,2,3}}(子集的顺序任意)。

2. 问题求解

以 $n=3$ 为例，求{1,2,3}的幂集的过程如图 3.11 所示，步骤如下：

① {1}的幂集 $M_1=\{\{\},\{1\}\}$。

② 在 M_1 的每个集合元素的末尾添加 2 得到 $A_2=\{\{2\},\{1,2\}\}$，将 M_1 和 A_2 的全部元素合并起来得到 $M_2=\{\{\},\{1\},\{2\},\{1,2\}\}$。

③ 在 M_2 的每个集合元素的末尾添加 3 得到 $A_3=\{\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$，将 M_2 和 A_3 的全部元素合并起来得到 $M_3=\{\{\},\{1\},\{2\},\{1,2\},\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$。

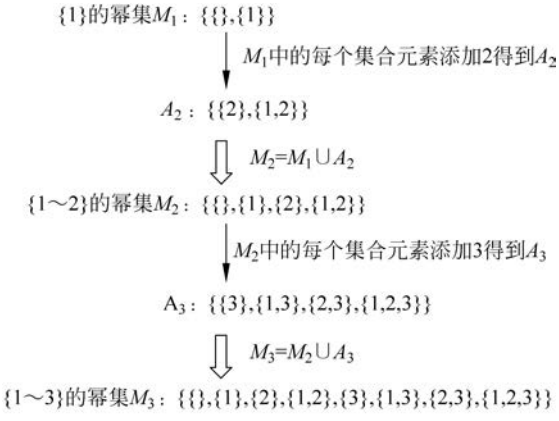


图 3.11 求{1,2,3}的幂集的过程

归纳起来，设 M_i 表示{1～i}($i\geqslant 1$，共 i 个元素)的幂集(是一个两层集合)，为大问题，则 M_{i-1} 为{1～$i-1$}(共 $i-1$ 个元素)的幂集，为小问题。显然有 $M_1=\{\{\},\{1\}\}$。

考虑 $i>1$ 的情况，假设 M_{i-1} 已经求出，定义运算 appendi(M_{i-1},i)返回在 M_{i-1} 中每个集合元素的末尾插入整数 i 的结果，即：

$$\text{appendi}(M_{i-1},i)=\bigcup_{s\in M_{i-1}}\text{push\_back}(s,i)$$

则：

$$M_i=M_{i-1}\cup A_i,\quad \text{其中 } A_i=\text{appendi}(M_{i-1},i)$$

这样求{1～n}的幂集的递推关系如下：

$M_1=\{\{\},\{1\}\}$
$M_i=M_{i-1}\cup A_i$　　当 $i>1$ 时

幂集是一个两层集合，采用 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子集。大问题即求{1～i}的幂集用 M_i 变量表示，小问题即求{1～$i-1$}的幂集用 M_i_1 变量表示，首先置 M_i_1={{},{1}}表示 M_1，迭代变量 i 从 2 到 n 循环，每次迭代将完成的问题规模由 i 增加为 $i+1$。对应的迭代法算法如下：

```
vector<vector<int>> appendi(vector<vector<int>> Mi_1,int i)
//向 Mi_1 中的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;
}
vector<vector<int>> subsets1(int n)              //迭代法:求{1～n}的幂集
{   vector<vector<int>> Mi;                      //存放{1～n}的幂集
    vector<vector<int>> Mi_1={{},{1}};           //初始时存放 M1
    if(n==1) return Mi_1;                        //处理特殊情况
    for(int i=2;i<=n;i++)                        //迭代循环
    {   vector<vector<int>> Ai=appendi(Mi_1,i);
        Mi=Mi_1;
        for(int j=0;j<Ai.size();j++)             //将 Ai 的所有集合元素添加到 Mi 中
            Mi.push_back(Ai[j]);
        Mi_1=Mi;                                 //新值取代旧值
    }
    return Mi;
}
```

扫一扫

视频讲解

3.3.5 实战——子集(LeetCode78)

1. 问题描述

给定一个整数数组 nums，长度范围是 1～10，其中所有元素互不相同。求该数组所有可能的子集(幂集)，结果中不能包含重复的子集，可以按任意顺序返回幂集。例如，nums=[1,2,3]，结果为[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]。要求设计如下成员函数：

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums) {  }
};
```

2. 问题求解

将数组 nums 看成一个集合(所有元素互不相同)，求 nums 的所有可能的子集就是求 nums 的幂集，与 3.3.4 节的思路完全相同，仅需要将求 1～n 的幂集改为求 nums[0..$n-1$]的幂集。定义运算 appendi(M_{i-1},nums[i])返回在 M_{i-1} 中每个集合元素的末尾插入元素的 nums[i]的结果，即：

$$\text{appendi}(M_{i-1},\text{nums}[i]) = \bigcup_{s\in M_{i-1}} \text{push\_back}(s,\text{nums}[i])$$

则：

$$M_i = M_{i-1} \cup A_i, \quad \text{其中 } A_i = \text{appendi}(M_{i-1},\text{nums}[i])$$

这样求 nums 的幂集的递推关系如下：

$M_1 = \{\{\},\{\text{nums}[0]\}\}$
$M_i = M_{i-1} \cup A_i$　　当 $i>0$ 时

对应的迭代法算法如下：

```
class Solution {
public:
    vector<vector<int>> subsets(vector<int>& nums)      //迭代法:求 nums 的幂集
    {   vector<vector<int>> Mi;                         //存放幂集
        vector<vector<int>> Mi_1={{},{nums[0]}};        //存放 M1
        if(nums.size()==1) return Mi_1;                 //处理特殊情况
        for(int i=1;i<nums.size();i++)
        {   vector<vector<int>> Ai=appendi(Mi_1,nums[i]);
            Mi=Mi_1;
            for(int j=0;j<Ai.size();j++)                //将 Ai 的所有集合元素添加到 Mi 中
                Mi.push_back(Ai[j]);
            Mi_1=Mi;
        }
        return Mi;
    }
    vector<vector<int>> appendi(vector<vector<int>> Mi_1,int e)
    //向 Mi_1 中的每个集合元素的末尾添加 e
    {   vector<vector<int>> Ai=Mi_1;
        for(int j=0;j<Ai.size();j++)
            Ai[j].push_back(e);
        return Ai;
    }
};
```

上述程序提交时通过，执行用时为 0ms，内存消耗为 7.1MB。

3.4 递 归 法

3.4.1 递归法概述

1. 什么是递归

递归算法是指在算法定义中又调用自身的算法。若在 p 算法定义中调用 p 算法，称之

为直接递归算法；若在 p 算法定义中调用 q 算法，而在 q 算法定义中又调用 p 算法，称之为间接递归算法。任何间接递归算法都可以等价地转换为直接递归算法，所以本节主要讨论直接递归算法。

递归算法通常是把一个大的复杂问题层层转换为一个或多个与原问题相似的规模较小的问题来求解，具有思路清晰和代码少的优点。目前主流的计算机语言如 C/C++、Java 等都支持递归，在内部通过系统栈实现递归调用。一般来说，能够用递归解决的问题应该满足以下 3 个条件：

① 需要解决的问题可以转换为一个或多个子问题来求解，而这些子问题的求解方法与原问题完全相同，只是在数量规模上不同。

② 递归调用的次数必须是有限的。

③ 必须有结束递归的条件来终止递归。

与迭代法类似，递归法也是一种算法实现技术，在设计递归算法时首先采用归纳法建立递推关系，这里称为递归模型，然后在此基础上直接转换为递归算法。

2. 递归模型的一般格式

递归模型总是由递归出口和递归体两部分组成。递归出口表示递归到何时结束（对应最初、最原始的问题），递归体表示求解时的递推关系。一个简化的递归模型如下：

$$f(s_1)=m$$
$$f(s_n)=g(f(s_{n-1}),c)$$

其中 g 是一个非递归函数，m 和 c 为常量。例如，为了求 $n!$，设 $f(n)$ 表示 $n!$，对应的递归模型如下：

$$f(1)=1$$
$$f(n)=n*f(n-1) \qquad \text{当 } n>1 \text{ 时}$$

3. 提取求解问题的递归模型

结合算法设计的特点，提取求解问题的递归模型的一般步骤如下：

① 对大问题 $f(s)$ 进行分析，假设出合理的“小问题”$f(s')$。

② 假设小问题 $f(s')$ 是可解的，在此基础上确定大问题 $f(s)$ 的解，即给出 $f(s)$ 与 $f(s')$ 之间的递推关系，也就是提取递归体（与数学归纳法中假设 $i=n-1$ 时等式成立，再求证 $i=n$ 时等式成立的过程相似）。

③ 确定一个特定情况（如 $f(1)$ 或 $f(0)$）的解，由此作为递归出口（与数学归纳法中求证 $i=1$ 或 $i=0$ 时等式成立相似）。

4. 递归算法框架

在递归模型中递归体是核心，用于将求得的小问题解通过合并操作构成大问题的解，由求小问题解和合并操作的次序分为两种基本递归框架。

（1）先求小问题解后做合并操作，即先递后合，也就是在归来的过程中解决问题，其框架如下。

```
void recursion1(n)                          //先递后合的递归框架
{   if (满足出口条件)
        直接解决;
    else
    {   recursion1(m);                      //递去,递到最深处
        merge();                            //归来时执行合并操作
    }
}
```

(2) 先做合并操作再求小问题解，即先合后递，也就是在递去的过程中解决问题，其框架如下。

```
void recursion2(n)                          //先合后递的递归框架
{   if (满足出口条件)
        直接解决;
    else
    {   merge();                            //合并,递去
        recursion2(m);                      //递到最深处后,再不断地归来
    }
}
```

对于复杂的递归问题，例如在递去和归来过程中都包含合并操作，一个大问题分解为多个子问题等，其求解框架一般是上述基本框架的叠加。

扫一扫

视频讲解

【例 3-6】 假设二叉树采用二叉链存储，设计一个算法判断两棵二叉树 $t1$ 和 $t2$ 是否相同，所谓相同是指它们的形态相同并且对应的结点值相同。

解 像树和二叉树等递归数据结构特别适合采用递归算法求解。对于本例，设 $f(t1,t2)$ 表示二叉树 $t1$ 和 $t2$ 是否相同，它们的左、右子树的判断是两个小问题，如图 3.12 所示。依题意，对应的递归模型如下：

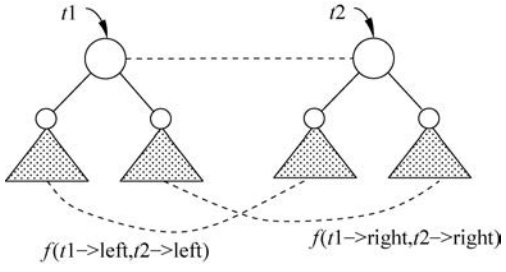


图 3.12　二叉树的两个小问题

$f(t1,t2)=\text{true}$　　当 $t1$ 和 $t2$ 均为空时
$f(t1,t2)=\text{false}$　　当 $t1$、$t2$ 中的一个为空另外一个非空时
$f(t1,t2)=\text{false}$　　当 $t1$ 和 $t2$ 均不空但是结点值不相同时
$f(t1,t2)=f(t1\text{->left},t2\text{->left})\ \&\&\ f(t1\text{->right},t2\text{->right})$　　其他

对应的递归算法如下：

```
bool same(TreeNode * t1,TreeNode * t2)          //递归算法:判断 t1 和 t2 是否相同
{   if(t1==NULL && t2==NULL)
        return true;
```

```
    else if(t1==NULL || t2==NULL)
        return false;
    if(t1 -> val!=t2 -> val)
        return false;
    bool left=same(t1 -> left,t2 -> left);          //递归调用 1
    bool right=same(t1 -> right,t2 -> right);       //递归调用 2
    return left && right;
}
```

【例 3-7】 有 $n+2$ 个实数 $a_0,a_1,\cdots,a_n$ 和 x，设计一个算法求多项式 $P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0$ 的值。

解 多项式 $P_n(x)$是由 n 项构成的，可以直接对每一项分别求值，通过累加得到最后的结果。对应的算法如下：

```
double solve1(double a[],int n,double x)          //解法 1：求多项式的值
{   double p=0.0, p1;
    for (int i=n;i>=0;i--)
    {   p1=1.0;
        for (int j=1;j<=i;j++)
            p1 * =x;
        p+=p1 * a[i];
    }
    return p;
}
```

solve1 算法十分低效，因为它需要做 $n+(n-1)+\cdots+1=n(n+1)/2$ 次乘法和 n 次加法，时间复杂度为 $O(n^2)$。通过归纳法可以导出一种快得多的方法：

$$\begin{aligned}P_n(x)&=a_nx^n+a_{n-1}x^{n-1}+\cdots+a_1x+a_0\\&=((\cdots(((a_nx+a_{n-1})x+a_{n-2})x+a_{n-3})\cdots)x+a_1)x+a_0\end{aligned}$$

设

$$P_0(x)=a_n$$

则：

$$\begin{aligned}P_1(x)&=P_0(x)x+a_{n-1}\\P_2(x)&=P_1(x)x+a_{n-2}\\&\vdots\\P_i(x)&=P_{i-1}(x)x+a_{n-i}\\&\vdots\\P_n(x)&=P_{n-1}(x)x+a_0\end{aligned}$$

这种求值的安排称为 Horner 规则。用 $f(a,n,x,i)$表示求 $P_i(x)$的值，为大问题，用 $f(a,n,x,i-1)$表示求 $P_{i-1}(x)$的值，为小问题。对应的递归模型如下：

```
f(a,n,x,i)=a[n]                          当 i=0 时
f(a,n,x,i)=x * f(a,n,x,i-1)+a[n-i]       其他
```

对应的递归算法如下：

```
double Horner(double a[],int n,double x,int i)          //递归算法
{   if (i==0)
        return a[n];
    else
        return x * Horner(a,n,x,i-1)+a[n-i];
}
double solve2(double a[],int n,double x)                //解法 2：求多项式的值
{
    return Horner(a,n,x,n);
}
```

在 solve2 算法中执行 n 次乘法和 n 次加法，时间复杂度为 $O(n)$，从而显著地改进算法的性能。下面再通过几个应用进一步讨论递归法算法设计的过程。

3.4.2 冒泡排序

1. 问题描述

有一个整数序列 $R[0..n-1]$，采用冒泡排序实现 R 的递增有序排序。冒泡排序的过程是，i 从 0 到 $n-2$ 循环，$R[0..i-1]$是有序区，$R[i..n-1]$是无序区，并且前者的所有元素均小于或等于后者的任意元素，在 $R[i..n-1]$无序区中通过冒泡方式将最大元素放在 $R[i]$位置。

2. 问题求解

采用不完全归纳法产生冒泡排序的递推关系。例如 $R=(2,5,4,1,3)$，这里 $n=5$，用[ ]表示有序区，各趟的排序结果如下：

```
初始:   ([]2,5,4,1,3)
i=0:    ([1],2,5,4,3)
i=1:    ([1,2],3,5,4)
i=2:    ([1,2,3],4,5)
i=3:    ([1,2,3,4],5)
```

3. 先递后合的递归算法

设 $f(R,i)$用于实现 $R[0..i]$(共 $i+1$ 个元素)的递增排序，它是大问题，而 $f(R,i-1)$实现 $R[0..i-1]$(共 i 个元素)的排序，它是小问题。当 $i=-1$ 时，$R[0..i]$为空，看成是有序的。对应的递推关系如下：

```
f(R,i) ≡ 不做任何事情                                  当 i=-1 时
f(R,i) ≡ f(R,i-1);                                     否则
         在 R[i..n-1]中冒出最小元素到 R[i]位置;
```

显然 $f(R,n-1)$用于实现 $R[0..n-1]$递增排序，由于这样排序后最后一个元素 $R[n-1]$一定是最大元素，所以调用 $f(R,n-2)$就可以实现 $R[0..n-1]$的递增排序。对应的递归算法如下：

```
void Bubble1(vector<int> & R,int i)      //冒泡操作: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
}
void BubbleSort21(vector<int> & R,int i)    //被 BubbleSort2 调用
{   if (i==-1) return;                      //满足递归出口条件
    BubbleSort21(R,i-1);                    //递归调用
    Bubble1(R,i);                           //合并操作
}
void BubbleSort2(vector<int> & R)           //递归算法:冒泡排序
{   int n=R.size();
    BubbleSort21(R,n-2);
}
```

4. 先合后递的递归算法

设 $f(R,i)$用于实现 $R[i..n-1]$(共 $n-i$ 个元素)的递增排序,它是大问题,则 $f(R,i+1)$实现 $R[i+1..n-1]$(共 $n-i-1$ 个元素)的排序,它是小问题。当 $i=n-1$ 时,$R[n-1..n-1]$仅包含最后一个元素,它一定是最大元素,排序结束。对应的递推关系如下:

$$
\begin{aligned}
&f(R,i) \equiv \text{不做任何事情} && \text{当 } i=n-1 \text{ 时}\\
&f(R,i) \equiv \text{在 } R[i..n-1] \text{中冒出最小元素到 } R[i] \text{位置;} \\
&\qquad\quad f(R,i+1); && \text{否则}
\end{aligned}
$$

显然 $f(R,0)$用于实现 $R[0..n-1]$的递增排序。对应的递归算法如下:

```
void BubbleSort31(vector<int> & R,int i)    //被 BubbleSort3 调用
{   int n=R.size();
    if (i==n-1) return;                     //满足递归出口条件
    Bubble1(R,i);                           //合并操作
    BubbleSort31(R,i+1);                    //递归调用
}
void BubbleSort3(vector<int> & R)           //递归算法:冒泡排序
{
   BubbleSort31(R,0);
}
```

扫一扫

视频讲解

3.4.3 求全排列

1. 问题描述

给定正整数 $n(n\geqslant 1)$,给出求 $1\sim n$ 的全排序的递归模型和递归算法。例如,$n=3$ 时全排列是{{1,2,3},{1,3,2},{3,1,2},{2,1,3},{2,3,1},{3,2,1}}。

2. 问题求解

以 $n=3$ 为例,求 1～3 的全排列的过程如图 3.13 所示,步骤如下:

① 1 的全排列是{{1}}。

② {{1}}中只有一个元素{1},在{1}前后位置分别插入 2 得到{1,2},{2,1},合并起来得到 1～2 的全排列{{1,2},{2,1}}。

③ {{1,2},{2,1}}中有两个元素，在{1,2}中的 3 个位置插入 3 得到{1,2,3},{1,3,2},{3,1,2}，在{2,1}中的 3 个位置插入 3 得到{2,1,3},{2,3,1},{3,2,1}，合并起来得到 1～3 的全排列{{1,2,3},{1,3,2},{3,1,2},{2,1,3},{2,3,1},{3,2,1}}。

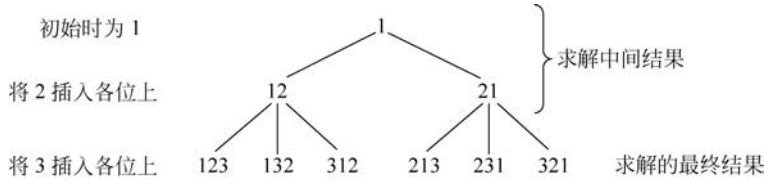


图 3.13　求 1～3 的全排列的过程

归纳起来，设 P_i 表示 1～i($i \geqslant 1$，共 i 个元素)的全排列(是一个两层集合，其中每个集合元素表示 1～i 的某个排列)，为大问题，则 P_{i-1} 为 1～$i-1$(共 $i-1$ 个元素)的全排列，为小问题。显然有 $P_1=\{\{1\}\}$。

考虑 $i>1$ 的情况，假设 P_{i-1} 已经求出，对于 P_{i-1} 中的任意一个集合元素 s，s 表示为 $s_0 s_1 \cdots s_{i-2}$(长度为 $i-1$，下标从 0 开始)，其中有 i 个插入位置(即位置 $i-1$、位置 $i-2$、……、位置 0)，定义运算 Insert(s,i,j)返回 s 的序号为 j($0 \leqslant j \leqslant i-1$)的位置上插入元素 i 后的集合元素，定义 CreatePi(s,i)返回 s 中每个位置插入 i 的结果，即

$$\text{CreateP}_i(s,i)=\bigcup_{0 \leqslant j \leqslant s.\text{size}()} \text{Insert}(s,i,j)$$

则

$$P_i=\bigcup_{s \in P_{i-1}} \text{CreateP}_i(s,i)$$

求 1～n 的全排序的递归模型如下：

$P_1=\{\{1\}\}$
$P_i=\bigcup_{s \in P_{i-1}} \text{CreatePi}(s,i)$　　当 $i>1$ 时

用 vector < vector < int >>容器存放 1～n 的全排列。对应的递归算法如下：

```
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 集合中的 i-1 到 0 位置插入 i
{    vector < vector < int >> tmp;
     for (int j=s.size();j>=0;j--)                           //在 s(含 i-1 个整数)的每个位置插入 i
     {    vector < int > s1=Insert(s,i,j);
          tmp.push_back(s1);                                 //添加到 tmp 中
     }
     return tmp;
}
vector < vector < int >> perm21(int n,int i)                 //被 Perm2 调用
{    if(i==1)                                                //递归出口
          return {{1}};
     else
     {    vector < vector < int >> Pi;
          vector < vector < int >> Pi_1=perm21(n,i-1);       //求出 Pi_1
```

```
        for (auto it=Pi_1.begin();it!=Pi_1.end();it++)    //在 Pi 的每个集合元素的各个位置插入 i
        {    vector<vector<int>> tmp=CreatePi(*it,i);
             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;
    }
}
vector<vector<int>> Perm2(int n)                        //用递归法求 1～n 的全排列
{
    return perm21(n,n);
}
```

扫一扫

视频讲解

3.4.4 实战——展开字符串(HDU1274)

1. 问题描述

在纺织 CAD 系统的开发过程中经常会遇到纱线排列的问题。该问题的描述是这样的：常用纱线的品种一般不会超过 25 种，可以用小写字母表示不同的纱线，例如 abc 表示 3 根纱线的排列。重复可以用数字和括号表示，例如 2(abc)表示 abcabc，1(a)=1a 表示 a，2ab 表示 aab。如果括号的前面没有表示重复的数字出现，则可认为是 1 被省略了，例如 cd(abc)=cd1(abc)=cdabc。这种表示方法非常简单紧凑，也易于理解，但是计算机却不能理解。为了使计算机接受，必须将简单紧凑的表达式展开。某 ACM 队接受了此项任务。如果你就是该 ACM 队的一员，请把这个程序编写完成。已知条件是输入的简单紧凑表达式的长度不超过 250 个字符，括号前表示重复的数不超过 1000，不会出现除了数字、括号、小写字母以外的任何其他字符；不会出现括号不配对等错误的情况(错误处理已由 ACM 队的其他队员完成了)。

输入格式：本题有多个测试用例，第一行输入测试用例个数 t，接着是 t 行表达式，表达式是按照前面介绍的意义书写的。

输出格式：输出时含有 t 行，每行对应一个输入的表达式。

输入样例：

```
2
1(1a2b1(ab)1c)
3(ab2(4ab))
```

输出样例：

```
abbabc
abaaaabaaaabababaaaabaaaababaaaabaaaab
```

2. 问题求解

采用 string 容器 s 存放紧凑表达式，ans 存放 s 展开的字符串。用整型变量 i 遍历 s，一边遍历一边展开，如果 s 中不包括括号，称 s 为简单紧凑表达式，其展开过程十分容易，例

如"2ab"看成"2a0b","2a"展开为"aa","0b"展开为"b",合起来结果是"aab"。也就是说,每个字符 c 都看成"nc",其中 n 是数字串,c 是单个字母,展开后合并即可。现在 s 中包含括号(所有括号一定是正确匹配的),可以将 s 看成形如"$n_1(s_1)\cdots n_2(s_2)\cdots$",其中 s_1 和 s_2 称为子紧凑表达式,从中看出这是递归定义的,所以特别适合采用递归法求解。

设计递归算法 unfold(s),递归出口是 s 为简单紧凑表达式的情况;将每个(si)的展开(即 unfold(si))作为小问题,非括号部分直接展开,再依次合并起来得到大问题 unfold(s)的结果。例如样例 s="3(ab2(4ab))",求展开过程如图 3.14 所示,先调用 unfold(s),将 s 看成"3($s1$)",递归调用 unfold($s1$),将 $s1$ 看成"ab2($s2$)",执行中递归调用 unfold($s2$),$s2$="4ab"是一个简单紧凑表达式,求出展开结果为"aaaab",再返回到 unfold($s1$),求出展开结果为"abaaaabaaaab",最后返回到 unfold(s),求出整个 s 的展开结果为"abaaaabaaaab abaaaabaaaab abaaaabaaaab"。

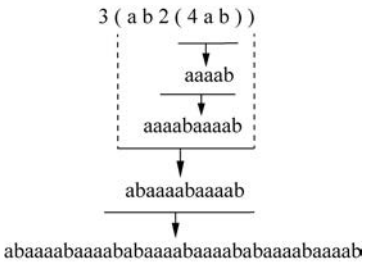


图 3.14 "3(ab2(4ab))"的展开过程

准确地说,unfold(s)的功能是求从 s 的起始下标 i(i 设置为全局变量,初始为 0)开始的紧凑表达式(s 结束为止)或者子紧凑表达式(遇到一个')'为止)的展开结果字符串。对应的递归法程序如下:

```
#include<iostream>
#include<string>
using namespace std;
int i;                                  //用于遍历 s,全局变量
string unfold(string s)                 //递归算法
{   string ans="";
    int n=0;
    for(;s[i];i++)
    {   if(isdigit(s[i]))               //遇到数字字符
            n=n*10+s[i]-'0';            //将若干连续的数字转换为整数 n
        else if(isalpha(s[i]))          //遇到字母,形如"2a"中的'a'
        {   if(n>0)                     //前面有数字的情况
            {   while(n--)              //展开 s[k]字符 n 次
                    ans+=s[i];
            }
            else ans+=s[i];             //前面没有数字的情况,相当于展开 s[k]字符一次
            n=0;                        //重置 n
        }
        else if(s[i]=='(')              //遇到'('
        {   i++;                        //跳过'('
            string tmp=unfold(s);       //递归调用展开括号中的子串得到 tmp
            if(n>0)                     //前面有数字的情况
            {   while(n--)              //展开 tmp 字符串 n 次
                    ans+=tmp;
            }
            else ans+=tmp;              //前面没有数字的情况,相当于展开 tmp 字符串一次
            n=0;                        //重置 n
        }
        else                            //遇到')'
            return ans;                 //结束并返回子紧凑表达式的 ans
```

```
    }
    return ans;                          //s 处理完毕返回 ans
}
int main()
{   int t;
    cin >> t;
    while(t--)
    {   string s;
        cin >> s;
        i=0; //从头开始遍历 s
        cout << unfold(s) << endl;
    }
    return 0;
}
```

上述程序提交时通过,执行用时为 15ms,内存消耗为 1808KB。

3.5 递推式计算

递归算法的执行时间可以用递推式(也称为递归方程)表示,这样求解递推式对算法分析来说极为重要。本节介绍几种求解简单递推式的方法,对于更复杂的递推式可以采用数学上的生成函数和特征方程求解。

3.5.1 直接展开法

求解递推式最自然的方法是将其反复展开,即直接从递归式出发,一层一层地往前递推,直到最前面的初始条件为止,就得到了问题的解。

【例 3-8】 求解汉诺塔问题的递归算法如下,分析移动 n 盘片的时间复杂度。

```
void Hanoi(int n,char x,char y,char z)
{   if (n==1)
        printf("将盘片%d 从%c 搬到%c\n",n,x,z);
    else
    {   Hanoi(n-1,x,z,y);
        printf("将盘片%d 从%c 搬到%c\n",n,x,z);
        Hanoi(n-1,y,x,z);
    }
}
```

解 设调用 Hanoi(n,x,y,z)的执行时间为 $T(n)$,由其执行过程得到以下求执行时间的递归关系(递推关系式)。

$$\begin{aligned} &T(n)=1 && \text{当 } n=1 \text{ 时}\\ &T(n)=2T(n-1)+1 && \text{当 } n>1 \text{ 时} \end{aligned}$$

则:

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

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

所以移动 n 盘片的时间复杂度为 $O(2^n)$。

【例 3-9】 分析以下递推式的时间复杂度。

$T(0)=0$
$T(n)=nT(n-1)+n!$ 当 $n\geqslant 1$ 时

解 $T(n)=nT(n-1)+n!=n[(n-1)T(n-2)+(n-1)!]+n!$

$=n(n-1)T(n-2)+2n!=n!(T(n-2)/(n-2)!+2)$

构造一个辅助函数 $g(n)$，令 $T(n)=n!g(n)(T(0)=g(0)=0)$，代入后有 $n!g(n)=n(n-1)!g(n-1)+n!$，简化为 $g(n)=g(n-1)+1$。

展开后有

$$g(n)=g(0)+\sum_{i=1}^{n}1=0+n=n$$

因此 $T(n)=n!g(n)=nn!=O(nn!)$。

3.5.2 递归树方法

用递归树求解递推式的基本过程是，展开递推式，构造对应的递归树，然后把每一层的时间进行求和，从而得到算法执行时间的估计，再用时间复杂度形式表示。

【例 3-10】 分析以下递推式的时间复杂度：

$T(n)=1$ 当 $n=1$ 时
$T(n)=2T(n/2)+n^2$ 当 $n>1$ 时

解 对于 $T(n)$画出一个结点如图 3.15(a)所示，将 $T(n)$展开一次的结果如图 3.15(b)所示，再展开 $T(n/2)$的结果如图 3.15(c)所示，以此类推，构造的递归树如图 3.16 所示，从中可看出在展开过程中子问题的规模逐步缩小，当到达递归出口时，即当子问题的规模为 1 时，递归树不再展开。

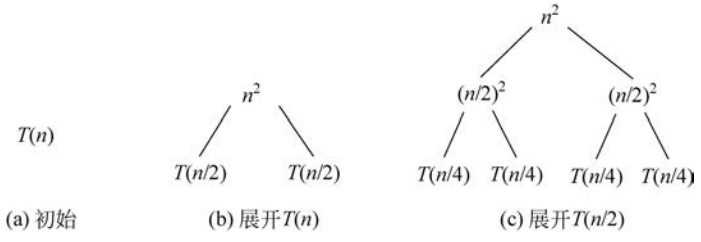


图 3.15 展开两次的结果

显然在递归树中，第 1 层的问题规模为 n，第 2 层的问题规模为 $n/2$，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^2，第 2 层有两个结点，其时间为 $2(n/2)^2=n^2/2$，以此类

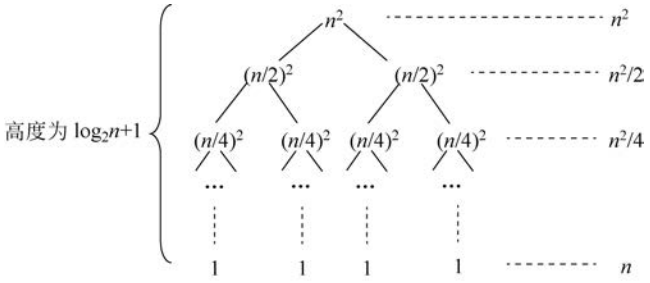


图 3.16　一棵递归树

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

$$T(n)=n^2+n^2/2+\cdots+n^2/2^{k-1}+\cdots+n=O(n^2)$$

【例 3-11】　分析以下递推式的时间复杂度：

$$T(n)=1 \qquad \text{当 } n=1 \text{ 时}$$
$$T(n)=T(n/3)+T(2n/3)+n \qquad \text{当 } n>1 \text{ 时}$$

解　构造的递归树如图 3.17 所示，不同于图 3.16 所示的递归树中所有叶子结点在同一层，这棵递归树的叶子结点的层次可能不同，从根结点出发到达叶子结点有很多路径，最左边的路径是最短路径，每走一步问题规模就减少为原来的 1/3，最右边的路径是最长路径，每走一步问题规模就减少为原来的 2/3。

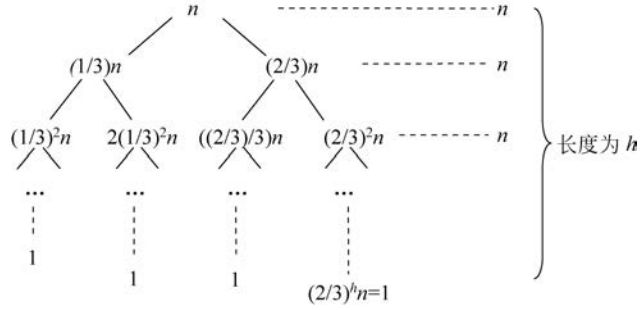


图 3.17　一棵递归树

最坏的情况是考虑右边最长的路径。设右边最长路径的长度为 h（指路径上经过的分支线数目），则有 $n(2/3)^h=1$，求出 $h=\log_{3/2}n$。

因此这棵递归树有 $\log_{3/2}n+1$ 层，每层结点的数值和为 n，所以 $T(n)\leqslant n(\log_{3/2}n+1)=O(n\log_{3/2}n)=O(n\log_2 n)$。

该递推式的时间复杂度是 $O(n\log_2 n)$。

3.5.3　主方法

主方法提供了求解如下形式递推式的一般方法：

$$T(1)=c$$
$$T(n)=aT(n/b)+f(n) \qquad \text{当 } n>1 \text{ 时}$$

其中 $a \geqslant 1, b > 1$ 为常数，n 为非负整数，$T(n)$表示算法的执行时间，该算法将规模为 n 的原问题分解成 a 个子问题，每个子问题的大小为 n/b，$f(n)$表示分解原问题和合并子问题解得到答案的时间。例如，对于递推式 $T(n)=3T(n/4)+n^2$，有 $a=3, b=4, f(n)=n^2$。

主方法的求解对应如下主定理。

主定理： 设 $T(n)$是满足上述定义的递推式，$T(n)$的计算如下。

① 若对于某个常数 $\varepsilon>0$，有 $f(n)=O(n^{\log_b a-\varepsilon})$，称为 $f(n)$多项式小于 $n^{\log_b a}$（即 $f(n)$与 $n^{\log_b a}$ 的比值小于或等于 $n^{-\varepsilon}$），则 $T(n)=\Theta(n^{\log_b a})$。

② 若 $f(n)=\Theta(n^{\log_b a})$，即 $f(n)$多项式的阶等于 $n^{\log_b a}$，则 $T(n)=\Theta(n^{\log_b a}\log_2 n)$。

③ 若对于某个常数 $\varepsilon>0$，有 $f(n)=O(n^{\log_b a+\varepsilon})$，称为 $f(n)$多项式大于 $n^{\log_b a}$（即 $f(n)$与 $n^{\log_b a}$ 的比值大于或等于 n^{ε}），并且满足 $af(n/b)\leqslant cf(n)$，其中 $c<1$，则 $T(n)=\Theta(f(n))$。

主定理涉及的 3 种情况中都是以 $f(n)$与 $n^{\log_b a}$ 作比较，递推式解的渐近阶由这两个函数中的较大者决定。情况①是函数 $n^{\log_b a}$ 阶较大，则 $T(n)=\Theta(n^{\log_b a})$；情况③是函数 $f(n)$阶较大，则 $T(n)=\Theta(f(n))$；情况②是两个函数阶一样大，则 $T(n)=\Theta(n^{\log_b a}\log_2 n)$，即以 n 的对数作为因子乘上 $f(n)$与 $T(n)$的同阶。

此外有一些细节不能忽视，情况①中 $f(n)$不仅必须比 $n^{\log_b a}$ 阶小，而且必须是多项式比 $n^{\log_b a}$ 小，即 $f(n)$必须渐近地小于 $n^{\log_b a}$ 与 $n^{-\varepsilon}$ 的积；情况③中 $f(n)$不仅必须比 $n^{\log_b a}$ 阶大，而且必须是多项式比 $n^{\log_b a}$ 大，即 $f(n)$必须渐近地大于 $n^{\log_b a}$ 与 n^{ε} 的积，同时还要满足附加的“正规性”条件，即 $af(n/b)\leqslant cf(n)$，该条件的直观含义是 a 个子问题再分解和再合并所需要的时间最多与原问题分解和合并所需要的时间同阶，这样 $T(n)$就由 $f(n)$确定。也就是说，如果不满足正规性条件，采用这种递归分解和合并求解的方法是不合适的，即时间性能差。

还有一点很重要，即上述 3 类情况并没有覆盖所有可能的 $f(n)$。在情况①和②之间有一个间隙，即 $f(n)$小于但不是多项式小于 $n^{\log_b a}$。类似地，在情况②和③之间也有一个间隙，即 $f(n)$大于但不是多项式大于 $n^{\log_b a}$。如果函数 $f(n)$落在这两个间隙之一中，或者虽有 $f(n)=O(n^{\log_b a+\varepsilon})$，但正规性条件不满足，那么主方法则无能为力。

【例 3-12】 采用主定理求以下递推式的时间复杂度。

$$\begin{aligned} &T(n)=1 && \text{当 } n=1 \text{ 时}\\ &T(n)=4T(n/2)+n && \text{当 } n>1 \text{ 时}\end{aligned}$$

解 这里 $a=4, b=2, n^{\log_b a}=n^2, f(n)=n=O(n^{\log_b a-\varepsilon})$，由于 $\varepsilon=1$，即 $f(n)$多项式地小于 $n^{\log_b a}$，满足情况①，所以 $T(n)=\Theta(n^{\log_b a})=\Theta(n^2)$。

【例 3-13】 采用主方法求以下递推式的时间复杂度。

$$\begin{aligned} &T(n)=1 && \text{当 } n=1 \text{ 时}\\ &T(n)=3T(n/4)+n\log_2 n && \text{当 } n>1 \text{ 时}\end{aligned}$$

解 这里 $a=3, b=4, f(n)=n\log_2 n$。$n^{\log_b a}=n^{\log_4 3}=O(n^{0.793})$，显然 $f(n)$的阶大于

$n^{0.793}$（因为 $f(n)=n\log_2 n>n^1>n^{0.793}$），如果能够证明主定理中情况③成立，则按该情况求解。对于足够大的 n，$af(n/b)=3(n/4)\log_2(n/4)=(3/4)n\log_2 n-3n/2\leqslant(3/4)n\log_2 n=cf(n)$，这里 $c=3/4$，满足正规性条件，则有 $T(n)=\Theta(f(n))=\Theta(n\log_2 n)$。

【例 3-14】 采用主定理和直接展开法求以下递推式的时间复杂度。

$$\begin{aligned} &T(n)=1 && \text{当 } n=1 \text{ 时}\\ &T(n)=2T(n/2)+(n/2)^2 && \text{当 } n>1 \text{ 时}\end{aligned}$$

解 采用主定理，这里 $a=2,b=2,n^{\log_b a}=n^{\log_2 2}=n$，$f(n)=n^2/4$，$f(n)$ 多项式地大于 $n^{\log_b a}$。对于足够大的 n，$af(n/b)=2f(n/2)=2(n/2/2)^2=n^2/8\leqslant cn^2/4=cf(n)$，$c\leqslant 1/2$ 即可。也就是说，满足正规性条件，按照主方法的情况③，有 $T(n)=\Theta(f(n))=\Theta(n^2)$。

采用直接展开法求解，不妨设 $n=2^{k+1}$，即 $\dfrac{n}{2^k}=2$。

$$\begin{aligned} T(n)&=2T\left(\frac{n}{2}\right)+\left(\frac{n}{2}\right)^2=2\left(2T\left(\frac{n}{2^2}\right)+\frac{n^2}{2^4}\right)+\left(\frac{n}{2}\right)^2=2^2T\left(\frac{n}{2^2}\right)+\frac{n^2}{2^3}+\frac{n^2}{2^2}\\ &=2^2\left(2T\left(\frac{n}{2^3}\right)+\frac{n^2}{2^6}\right)+\frac{n^2}{2^3}+\frac{n^2}{2^2}=2^3T\left(\frac{n}{2^3}\right)+\frac{n^2}{2^4}+\frac{n^2}{2^3}+\frac{n^2}{2^2}\\ &=\cdots=2^kT\left(\frac{n}{2^k}\right)+\frac{n^2}{2^{k+1}}+\frac{n^2}{2^k}+\cdots+\frac{n^2}{2^2}\\ &=\frac{n}{2}\times 2+n^2\left(\frac{1}{2^{k+1}}+\frac{1}{2^k}+\cdots+\frac{1}{2^2}\right)=n+n^2\left(\frac{1}{2}-\frac{1}{n}\right)=\frac{n^2}{2}=\Theta(n^2)\end{aligned}$$

两种方法得到的结果是相同的。以上介绍的递推式求解方法在第 4 章有关分治算法的分析中大量用到。

可以这样简化主定理，如果递推式如下：

$$\begin{aligned} &T(1)=c\\ &T(n)=aT(n/b)+cn^k && \text{当 } n>1 \text{ 时}\end{aligned}$$

其中 a、b、c、k 都是常量，有：

① 若 $a>b^k$，则 $T(n)=O(n^{\log_b a})$。

② 若 $a=b^k$，则 $T(n)=O(n^k\log_b n)$。

③ 若 $a<b^k$，则 $T(n)=O(n^k)$。

3.6 练 习 题

3.6.1 单项选择题

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

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

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. $a_n=\frac{n}{2}$

4. 猜想 $1=1$，$1-4=-(1+2)$，$1-4+9=1+2+3$，…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. $1^2-2^2+3^2-4^2+5^2=1+2+3+4+5$

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

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

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

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. $f(n)=n$

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

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

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

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

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. $T(n)=3n\log_2 n$

3.6.2 问答题

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 个人合起来做一个任务，他们不同的排列顺序完成该任务的时间不同，求最优完成时间。

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？

3. 什么是递归算法？递归模型由哪两个部分组成？

4. 比较迭代算法与递归算法的异同。

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。

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。

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。

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;
}
```

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);
}
```

10. 采用直接展开法求以下递推式：

$$T(1)=1$$
$$T(n)=T(n-1)+n \qquad \text{当 } n>1 \text{ 时}$$

11. 采用递归树方法求解以下递推式：

$$T(1)=1$$
$$T(n)=4T(n/2)+n \qquad \text{当 } n>1 \text{ 时}$$

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

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);
    }
}
```

14. 分析 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。

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。

3.6.3 算法设计题

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。

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。

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 $s=1+(1+2)+(1+2+3)+\cdots+(1+2+\cdots+n)$。

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。

5. 设计一个递归算法用于翻转一个非空字符串 s。

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。

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。

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。

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。

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($1\leqslant k\leqslant$二叉树 b 的高度)层所有结点值的和(根结点层次为 1)。

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。

12. 在 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。

13. 在 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。

14. 在 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。

15. 在 3.3.4 节中采用迭代算法求 $1\sim n$ 的幂集，请设计等效的递归算法。

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

17. 在 3.4.3 节中求 $1\sim n$ 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。

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。

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。

3.7 上机实验题

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"。

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。

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
```

4. 验证汉诺塔问题

本书的例 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 盘片时搬动盘片的总次数的公式，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。

3.8 在线编程题

1. LeetCode344——反转字符串
2. LeetCode206——反转链表
3. LeetCode24——两两交换链表中的结点
4. LeetCode62——不同路径
5. HDU1003——最大子序列和
6. HDU1143——三平铺问题
7. POJ2231——奶牛的总音量
8. POJ1050——最大子矩形