第3章 必备技能——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、迭代法和递归法等基本算法设计方法及其递推式的计算，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）掌握递推式的各种计算方法。

（6）综合运用穷举法、归纳法、迭代法和递归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，再根据问题提出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 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。

③ 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。

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，尤其是处在计算机的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的运算速度是可以接受的，此时设计一个更高效率的算法不值得。

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底限，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) {                    //穷举法框架
    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。

【例 3-1】 鸡兔同笼问题。现有一个笼子，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。

扫一扫 视频讲解

解： 由于有鸡和兔两种动物，每只鸡有 2 条腿，每只兔有 4 条腿，设置两个循环变量，x 表示鸡的只数，y 表示兔的只数，那么穷举对象就是 x 和 y，假设穷举对象的顺序是先 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，根结点是分支对应 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)
                System.out.printf("x=%d,y=%d\n",x,y);
        }
    }
}
```

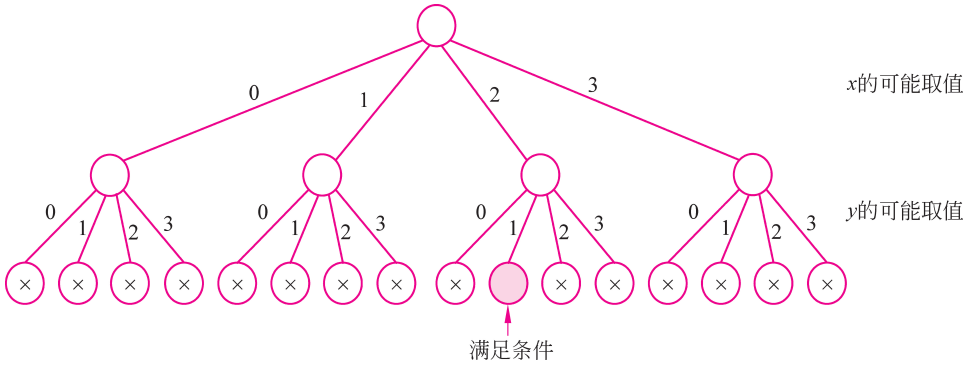

图 3.1 $a=3,b=8$ 的解空间树

从图 3.1 中看到,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。对应的解空间树如图 3.2 所示,共 17 个结点。

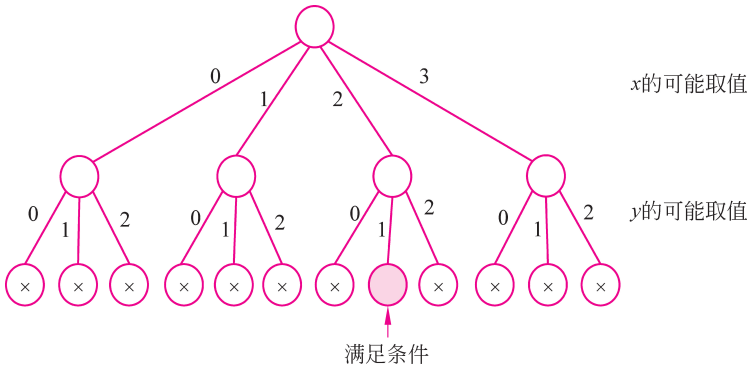

图 3.2 $a=3,b=8$ 的优化解空间树

对应的优化算法如下:

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

所以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,这就需要读者通过大量实训掌握一些基本算法设计技巧。3.1.2 节和 3.1.3 节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。

扫一扫 视频讲解

3.1.2 最大连续子序列和

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. 问题求解 1

对于含有 n 个整数的序列 $a[0..n-1]$,其中任何连续子序列为 $a[i..j](i\leqslant j,0\leqslant i\leqslant n-1,i\leqslant j\leqslant n-1)$,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。这种解法是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)来得到结果,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 $a[i..j](0\leqslant i\leqslant j\leqslant 5)$ 的所有元素之和如图 3.3 所示(行号为 i,列号为 j),其过程如下:

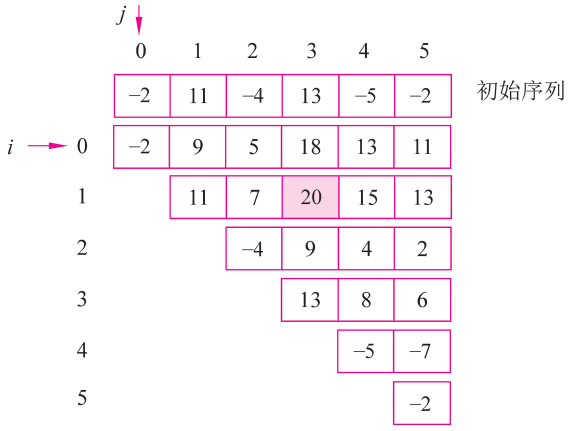

图 3.3 所有 $a[i..j]$ 子序列($0\leqslant i\leqslant j\leqslant 5$)的元素和

(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。

对应的算法如下:

```
int maxSubSum1(int a[]) {                          //解法 1
    int n=a.length;
    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++)
                cursum+=a[k];
            maxsum=Math.max(maxsum,cursum);        //通过比较求最大值 maxsum
        }
    }
    return maxsum;
}
```

【算法分析】 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. 问题求解 2

采用前缀和方法，用 Sum(i)表示子序列 $a[0..i-1]$的元素和，即 a 中前 i 个元素的和，显然有如下递推关系：

$$\text{Sum}(0)=0$$

$$\text{Sum}(i)=\text{Sum}(i-1)+a[i-1] \quad \text{当 } i>0 \text{ 时}$$

假设 $j \geqslant i$，则有：

$$\text{Sum}[i]=a[0]+a[1]+\cdots a[i-1]$$

$$\text{Sum}[j]=a[0]+a[1]+\cdots a[i-1]+a[i]+\cdots a[j-1]$$

两式相减得到 Sum[j]−Sum[i]=$a[i]$ +…$a[j-1]$，这样 i 从 0 到 $n-1$、$j-1$ 从 i 到$n-1$(即 j 从 $i+1$ 到 n) 循环可以求出所有连续子序列 $a[i..j]$之和，通过比较求出最大值 maxsum 即可。对应的算法如下：

```
static int maxSubSum2(int a[]) {                    //解法 2
    int n=a.length;
    int Sum[]=new int[n+1];
    Sum[0]=0;
    for(int i=1;i<=n;i++)
        Sum[i]=Sum[i-1]+a[i-1];
    int maxsum=0,cursum;
    for(int i=0;i<n;i++) {
        for(int j=i+1;j<=n;j++) {
            cursum=Sum[j]-Sum[i];
            maxsum=Math.max(maxsum,cursum);          //通过比较求最大值 maxsum
        }
    }
    return maxsum;
}
```

【算法分析】 maxSubSum2(a,n)算法中主要是两重循环，所以有：

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

4. 问题求解 3

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 $a[i..j]$的元素和(即表示以 $a[i]$为起始元素的前缀和)，显然有如下递推关系：

$$\text{Sum}(a[i..j])=0 \quad \text{当 } j<i \text{ 时}$$

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

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 maxSubSum3(int a[]) {                          //解法 3
    int n=a.length;
    int maxsum=0,cursum;
    for(int i=0;i<n;i++) {
        cursum=0;
        for(int j=i;j<n;j++) {
            cursum+=a[j];
            maxsum=Math.max(maxsum,cursum);        //通过比较求最大值 maxsum
        }
    }
    return maxsum;
}
```

【算法分析】 maxSubSum3(a,n)算法中只有两重循环,所以有:

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

5. 问题求解 4

对前面的解法 2 继续优化。将 maxsum 和 cursum 初始化为 0,用 i 遍历 a,置 cursum+=$a[i]$,也就是说 cursum 累计到 $a[i]$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 遍历完后返回 maxsum 即可。对应的算法如下:

```
int maxSubSum4(int a[]) {                          //解法 4
    int n=a.length;
    int maxsum=0,cursum=0;
    for(int i=0;i<n;i++) {
        cursum+=a[i];
        if(cursum>=maxsum)                         //通过比较求最大值 maxsum
            maxsum=cursum;
        if(cursum<0)                               //若 cursum<0,最大连续子序列从下一个位置开始
            cursum=0;
    }
    return Math.max(maxsum,0);
}
```

【算法分析】 maxSubSum4(a,n)算法中只有一重循环,所以时间复杂度为 $O(n)$。

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 的优化点是采用前缀和数组,解法 3 的优化点是找出 $a[i..j-1]$和 $a[i..j]$子序列的相关性,解法 4 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

3.1.3 实战——最大子序列和(LeetCode53★)

1. 问题描述

给定一个含 $n(1\leqslant n\leqslant 10^5)$个整数的数组 nums,设计一个算法找到一个具有最大和的连续子数组(子数组最少包含一个元素),返回其最大和。例如,nums={-2,1,-3,4,-1,2,1,-5,4},答案为 6,对应的连续子数组是{4,-1,2,1}。要求设计如下方法:

```
public int maxSubArray(int[] nums) { }
```

2. 问题求解

本题是求 nums 的最大连续子序列和,但该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,因此不能简单采用 3.1.2 节中解法 4 的优化点,需要做两点修改:

(1) 将表示结果的 maxsum 初始化为 nums[0]而不是 0。

(2) 在求当前以 nums[i]结尾的最大连续子序列和 cursum 时分为两个部分,一是 cursum+nums[i],另外一部分是只有 nums[i](即无论 nums[i]是否小于 0,它都有可能构成一个仅含一个元素的最大连续子序列),然后在这样的两个部分中取最大值得到 cursum,即 cursum=max(cursum+nums[i],nums[i])。

在每次得到一个连续子序列和时通过比较将最大值存放到 maxsum 中。对应的程序如下:

```
class Solution {
    public int maxSubArray(int[] nums) {
        int n=nums.length;
        if(n==1) return nums[0];
        int maxsum=nums[0],cursum=0;
        for(int i=0;i<n;i++) {
            cursum=Math.max(cursum+nums[i],nums[i]);
            maxsum=Math.max(maxsum,cursum);
        }
        return maxsum;
    }
}
```

上述程序提交时通过,运行时间为 1ms,内存消耗为 48.7MB。

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，分为第一数学归纳法和第二数学归纳法两种。

(1) 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。

(2) 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 3-2】 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一定的规则可供遵循。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”，而大、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 按推导问题的方向研究最初、最原始的若干问题。

② 按推导问题的方向寻求问题间的转换规律，即递推关系，使问题逐步转化成较低层级或简单的且能解决的问题或者已解决的问题。

递推算法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 3.4(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 3.4(b)所示。

图 3.4 顺推法和逆推法

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，往往先对大量个别事实进行观察，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。3.2.2 节～3.2.4 节通过几个示例讨论归纳法在算法设计中的应用。

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，每次循环时将 $R[i]$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；$f(R,i-1)$ 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：

$f(R,i)\equiv$ 不做任何事情　　当 $i=1$ 时

$f(R,i)\equiv f(R,i-1)$；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？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 $R[0..k]$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(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(int R[]) {               //迭代算法：直接插入排序
    int n=R.length;
```

```
        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。

扫一扫

视频讲解

3.2.3 实战——不同路径(LeetCode62★★)

1. 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。设计一个算法求机器人到达网格的右下角(标记为"Finish")总共有多少条不同的路径。例如,$m=3, n=3$,对应的网格如图 3.5 所示,答案为 6。要求设计如下方法:

```
public int uniquePaths(int m, int n) {}
```

2. 问题求解

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 3.5,这里 $x=2, y=2$,所有路径长度为 4,6 条不同的路径如下:

① 右右下下
② 右下右下
③ 右下下右
④ 下右右下
⑤ 下右下右
⑥ 下下右右

图 3.5 3×3 的网格

归纳起来,不同路径条数等于从 $x+y$ 个选择中挑选 x 个"下"或者 y 个"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答案。对应的程序如下:

```
class Solution {
    public int uniquePaths(int m, int n) {              //求解算法
        return comp(m-1,n-1);
    }
    int comp(int x,int y) {
        int a=x+y,b=Math.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，运行时间为 0ms，内存消耗 35.1MB。

说明： 在求 C_{x+y}^{x} 时如果先计算 $(x+y)!$，再计算 $x!y!$，最后求二者相除的结果，当 x、y 较大时会发生溢出。

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。求猴子第 1 天一共摘了多少桃子。

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：

$$f(10)=1$$
$$f(9)=2(f(10)+1)$$
$$f(8)=2(f(9)+1)$$
$$\cdots$$

即 $f(i)=2(f(i+1)+1)$。这样得到递推关系如下：

$$f(i)=1 \quad \text{当 } i=n \text{ 时}$$
$$f(i)=2(f(i+1)+1) \quad \text{其他}$$

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 天的桃子数为 1,求第 1 天的桃子数
    int ans=1;
    for(int i=n-1;i>=1;i--)
        ans=2*(ans+1);
    return ans;
}
```

上述算法求出 $f(1)$为 1534。

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() {                    //迭代法框架
    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 节中所有的算法均是采用迭代法实现的。

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(例如在直接插入排序中,排序表前面的部分必须是有序的)。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 3.6 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:如果在循环的第一次迭代开始之前正确,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。

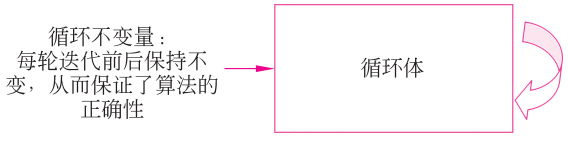

图 3.6 利用循环不变量证明算法的正确性

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”。

【例 3-3】 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：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 $R[0..0]$中只有一个元素，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。

这样证明了上述直接插入排序算法的正确性。

3.3.2 节～3.3.5 节将重点放在迭代法算法的设计上而不是算法的正确性证明上，通过几个经典应用予以讨论。

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，每次循环在 $R[i..n-1]$无序区采用简单比较找到最小元素 $R[\mathrm{min}j]$，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：

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

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(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.length;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]
        int tmp=R[minj];                        //交换
        R[minj]=R[i]; R[i]=tmp;
    }
}
void SelectSort1(int R[]) {                     //迭代法：简单选择排序
    for(int i=0;i<R.length-1;i++)               //进行 n-1 趟排序
        Select(R,i);
}
```

扫一扫

视频讲解

3.3.3 实战——多数元素(LeetCode169★)

1. 问题描述

见 2.12.3 节，这里采用迭代法求解。

2. 问题求解

依题意 nums 中一定存在多数元素。通过观察可以归纳出这样的结论，删除 nums 中任意两个不同的元素，则删除后多数元素依然存在且不变。假设 nums 中的多数元素为 c，即 c 出现的次数 cnt 大于 $n/2$，现在删除 nums 中任意两个不同的元素得到 nums1(含 $n-2$ 个元素)：

① 若两个元素均不是 c，则 c 在 nums1 中出现的次数仍然为 cnt，由于 cnt$>n/2>(n-2)/2$，所以 c 是 nums1 中的多数元素。

② 若两个元素有一个是 c，则 c 在 nums1 中出现的次数为 cnt-1，由于$(\text{cnt}-1)/(n-2)>(n/2-1)/(n-2)=1/2$，也就是说 c 在 nums1 中出现的次数超过一半，所以 c 是 nums1 中的多数元素。

既然上述结论成立，设候选多数元素为 c$=$nums[0]，计数器 cnt 表示 c 出现的次数(初始为 1)，i 从 1 开始遍历 nums，若两个元素(nums[i],c) 相同，cnt 增 1，否则 cnt 减 1，相当于删除这两个元素(nums[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 nums[$i+1$]开始重复查找，即重置 c$=$nums[$i+1$]，cnt$=1$。遍历结束后 c 就是 nums 中的多数元素。对应的迭代法算法如下：

```
class Solution {
    public int majorityElement(int[] nums) {
        int n=nums.length;
        if(n==1) return nums[0];
```

```
        int c=nums[0],cnt=1;
        int i=1;
        while(i<n) {
            if(nums[i]==c)              //选择两个元素(nums[i],c)
                cnt++;                  //相同时累加次数
            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=nums[i];
                cnt++;
            }
            i++;
        }
        return c;
    }
}
```

上述程序提交时通过，运行时间为 1ms，内存消耗为 44.3MB。对应算法的时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

说明：与 2.12.3 节的程序相比，上述程序的时间性能得到大幅度提高。

扫一扫

视频讲解

3.3.4 求幂集

1. 问题描述

给定正整数 $n(n\geqslant 1)$，请给出求{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 1～3 的幂集的过程如图 3.7 所示。

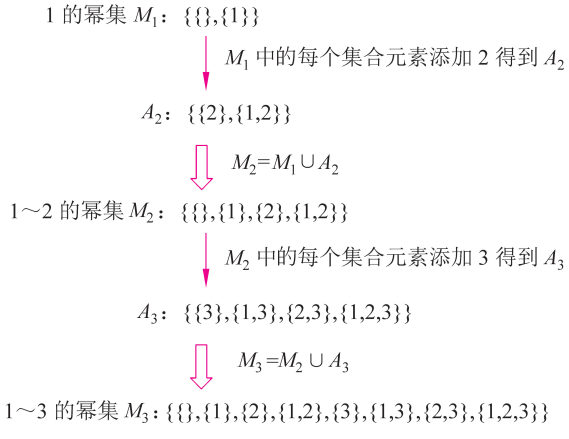

图 3.7 求{1,2,3}的幂集的过程

对应的步骤如下：

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 $M_3=\{\{\},\{1\},\{2\},\{1,2\},\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$。

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}是{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添加整数 i 的结果，即：

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

则：

$$M_i=M_{i-1}\cup A_i$$

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：

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

$$M_i=M_{i-1}\cup A_i \quad \text{当 } i>1 \text{ 时}$$

幂集是一个两层集合，采用 List<List<Integer>>类对象存放，其中每个 List<Integer>元素表示幂集中的一个集合。大问题即求{1～i}的幂集，用 Mi 变量表示，小问题即求{1～$i-1$}的幂集，用 Mi_1 变量表示，首先置 $Mi_1=\{\{\},\{1\}\}$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 $i+1$。对应的迭代法算法如下：

```
class Solution {
    List<List<Integer>>deepcopy(List<List<Integer>>A) {      //返回 A 的深复制
        List<List<Integer>>B=new ArrayList<>();
        for(List<Integer>x: A)
            B.add(new ArrayList<>(x));
        return B;
    }
    List<List<Integer>>appendi(List<List<Integer>>Mi_1,int e) {
                                          //向 Mi_1 中每个集合元素的末尾添加 e
        List<List<Integer>>Ai=Mi_1;       //浅复制
        for(List<Integer>x: Ai)
            x.add(e);
        return Ai;
    }
    public List<List<Integer>>subsets(int n) {      //迭代法：求 1 到 n 的幂集
        List<List<Integer>>Mi=new ArrayList<>();    //存放幂集
        List<List<Integer>>Mi_1=new ArrayList<>();
        Mi_1.add(new ArrayList<>());                //添加一个{}
        ArrayList<Integer>tmp=new ArrayList<>();
        tmp.add(1);
```

```
            Mi_1.add(tmp);                          //再添加一个{1},即 Mi_1={{},{1}}
            if(n==1) return Mi_1;                   //处理特殊情况
            for(int i=2;i<=n;i++) {
                Mi=deepcopy(Mi_1);
                List<List<Integer>>Ai=appendi(Mi_1,i);
                for(List<Integer>x: Ai)             //将 Ai 中的所有集合元素添加到 Mi 中
                    Mi.add(x);
                Mi_1=deepcopy(Mi);                  //用新值替换旧值
            }
            return Mi;
        }
    }
```

扫一扫

视频讲解

3.3.5 实战——子集(LeetCode78★★)

1. 问题描述

给定一个含 $n(1\leqslant n\leqslant 10)$ 个整数的数组 nums,其中所有元素互不相同。设计一个算法求该数组的所有可能的子集(幂集),结果中不能包含重复的子集,但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}}。要求设计如下方法:

```
public List<List<Integer>>subsets(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 $1\sim n$ 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 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:

$$M_i=M_{i-1}\cup A_i$$

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:

$$M_0=\{\{\},\{\text{nums}[0]\}\}$$

$$M_i=M_{i-1}\cup A_i \qquad \text{当 } i>0 \text{ 时}$$

对应的迭代法算法如下:

```
class Solution {
    public List<List<Integer>>subsets(int[] nums) {     //迭代法:求 nums 的幂集
        List<List<Integer>>Mi=new ArrayList<>();        //存放幂集
        List<List<Integer>>Mi_1=new ArrayList<>();
```

```
        Mi_1.add(new ArrayList<>());          //添加一个{}
        ArrayList<Integer>tmp=new ArrayList<>();
        tmp.add(nums[0]);
        Mi_1.add(tmp);                        //再添加一个{nums[0]}
        if(nums.length==1) return Mi_1;       //处理特殊情况
        for(int i=1;i<nums.length;i++) {
            Mi=deepcopy(Mi_1);
            List<List<Integer>>Ai=appendi(Mi_1,nums[i]);
            for(List<Integer>x: Ai)           //将 Ai 中的所有集合元素添加到 Mi 中
                Mi.add(x);
            Mi_1=deepcopy(Mi);                //用新值替换旧值
        }
        return Mi;
    }
    List<List<Integer>>appendi(List<List<Integer>>Mi_1,int e) {
        //向 Mi_1 中每个集合元素的末尾添加 e
        List<List<Integer>>Ai=Mi_1;           //浅复制
        for(List<Integer>x: Ai)
            x.add(e);
        return Ai;
    }
    List<List<Integer>>deepcopy(List<List<Integer>>A) {    //返回 A 的深复制
        List<List<Integer>>B=new ArrayList<>();
        for(List<Integer>x: A)
            B.add(new ArrayList<>(x));
        return B;
    }
}
```

上述程序提交时通过,执行用时为 1ms,内存消耗为 38.5MB。

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 p 算法定义中调用 p 算法,称为直接递归算法;若 p 算法定义中调用 q 算法,而 q 算法定义中又调用 p 算法,称为间接递归算法。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。