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，除了需要掌握常用的数据结构工具以外，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读者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。本章的学习要点和学习目标如下：

（1）掌握穷举法的原理和穷举法算法的框架。

（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法算法的方法。

（4）掌握递归法的原理和实现递归法算法的方法。

（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、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和解的特性可以采用不同的列举方法,常用以下几种列举方法。

① 顺序列举: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。排列中的元素是有顺序的。

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。组合中的元素是无顺序的。

穷举法的主要作用如下。

① 从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高的今天,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的规模不大,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 穷举法可以作为某类问题时间性能的底线,用来衡量同样问题的更高效率的算法。

穷举法的缺点主要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进行分析和归纳,寻找简化规律,精简穷举循环,优化穷举策略,以提高算法性能。

2. 穷举法算法的框架

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的所有可能的情况,而选择语句判断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 设计穷举法算法,编写相应的程序,执行和调试穷举法程序,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 $p(x_i,y_j)$，对应的穷举法算法的基本框架如下：

```
def Exhaustive(x,n,y,m):                #穷举法算法的框架
    for i in range(1,n+1):              #枚举 x 的所有可能的值
        for j in range(1,m+1):          #枚举 y 的所有可能的值
            ...
            if p(x[i],y[j]):            #检测约束条件是否成立
            ...
```

从中看出，x 和 y 的所有可能的搜索范围是笛卡儿积，即$([x_1,y_1],[x_1,y_2],\cdots,[x_1,y_m],\cdots,[x_n,y_1],[x_n,y_2],\cdots,[x_n,y_m])$，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，可能解对应于某些叶子结点。在第 5 章中将更详细地讨论解空间树的相关概念。

扫一扫

视频讲解

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

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（在本问题中也可以是先 y 后 x）。

显然，x 和 y 的取值范围都是 $0\sim a$，约束条件 $p(x,y)$ 为$(x+y==b)$和$(2x+4y=b)$。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：

```
1  def solve1(a,b):
2      for x in range(0,a+1):
3          for y in range(0,a+1):
4              if x+y==a and 2*x+4*y==b:
5                  print("x=%d,y=%d"%(x,y))
```

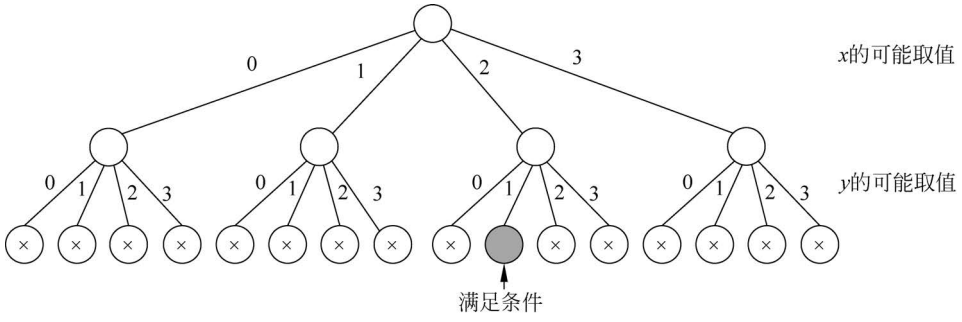

图 3.1 $a=3$、$b=8$ 的解空间树

从图 3.1 看到，解空间树中共有 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 $0\sim3$，y 的取值范围是 $0\sim2$。对应的解空间树如图 3.2 所示，共 17 个结点。

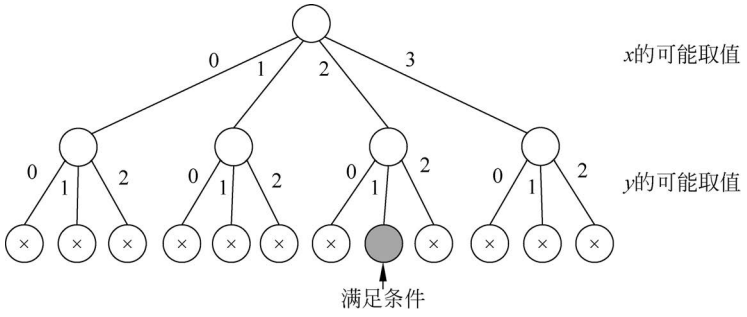

图 3.2 $a=3$、$b=8$ 的优化解空间树

对应的优化算法如下：

```
1  def solve2(a,b):
2      for x in range(0,min(a,b//2)+1):
3          for y in range(0,min(a,b//4)+1):
4              if x+y==a and 2*x+4*y==b:
5                  print("x=%d,y=%d"%(x,y))
```

所以尽管穷举法算法通常性能较差，但能够以它为基础进行优化继而得到高性能的算法，优化的关键是可以找出求解问题的优化点，不同的问题的优化点是不同的，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的设计方法，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)的最大连续子序列和为 20，序列(−6,2,4,−7,5,3,2,−1,6,−9,10,−2)的最大连续子序列和为 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

设有含 n 个整数的序列 $a[0..n-1]$，其连续子序列为 $a[i..j](i\leqslant j,0\leqslant i\leqslant n-1,i\leqslant j\leqslant n-1)$，求出它的所有元素之和 cursum，并通过比较将最大值存放到 maxsum 中，最后返回 maxsum。这种解法是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)，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。

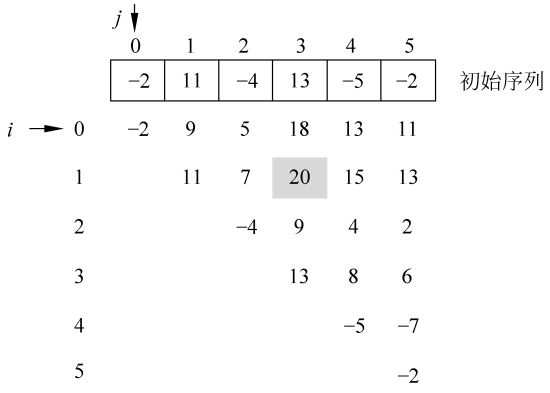

图 3.3 $a[i..j](0\leqslant i\leqslant j\leqslant 5)$的所有元素和

对应的算法如下：

```
def maxSubSum1(a):                                  #解法 1
    n,maxsum=len(a),0
    for i in range(0,n):                            #用三重循环穷举所有的连续子序列
        for j in range(i,n):
            cursum=0
            for k in range(i,j+1):cursum+=a[k]
            maxsum=max(maxsum,cursum)               #通过比较求最大 maxsum
    return maxsum
```

【算法分析】 在 maxSubSum1(a)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 presum[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：

presum[0]=0

presum[i]=presum[$i-1$]+$a[i-1]$　　当 $i>0$ 时

假设 $j\geqslant i$，则有：

presum[i]=$a[0]+a[1]+\cdots+a[i-1]$

presum[j]=$a[0]+a[1]+\cdots+a[i-1]+a[i]+\cdots+a[j-1]$

两式相减得到 presum[j]−presum[i]=$a[i]+\cdots+a[j-1]$，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 maxsum 即可。对应的算法如下：

```
def maxSubSum2(a):                                  #解法 2
    n=len(a)
    presum=[0]*(n+1)
    presum[0]=0
    for i in range(1,n+1):
```

```
6            presum[i]=presum[i-1]+a[i-1]
7        maxsum=0
8        for i in range(0,n):
9            for j in range(i+1,n+1):
10               cursum=presum[j]-presum[i]
11               maxsum=max(maxsum,cursum)              #通过比较求最大 maxsum
12       return maxsum
```

【算法分析】 在 maxSubSum2(a)算法中主要用了两重循环,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,显然有如下递推关系:

Sum($a[i..j]$)=0　　当 $j<i$ 时

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

这样在连续求以 $a[i]$开始的 $a[i..j]$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:

```
1    def maxSubSum3(a):                                 #解法 3
2        n,maxsum=len(a),0
3        for i in range(0,n):
4            cursum=0
5            for j in range(i,n):
6                cursum+=a[j]
7                maxsum=max(maxsum,cursum)              #通过比较求最大 maxsum
8        return maxsum
```

【算法分析】 在 maxSubSum3(a)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 3 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,将其存放到 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。在 a 遍历完后返回 maxsum 即可。对应的算法如下:

```
1    def maxSubSum4(a):                                 #解法 4
2        n,maxsum,cursum=len(a),0,0
```

```
    for i in range(0,n):
        cursum+=a[i]
        maxsum=max(maxsum,cursum)    #通过比较求最大 maxsum
        if cursum<0:cursum=0         #若 cursum<0,最大连续子序列从下一个位置开始
    return maxsum
```

【算法分析】 在 maxSubSum4(a)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(子数组中至少包含一个元素),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}。

2. 问题求解

本题是求 nums 的最大连续子序列和,但该序列中至少包含一个元素,也就是说最大连续子序列和可能为负数。例如 nums={-1,-2},结果为-2,因此需要在 3.1.2 节中解法 4 的基础上将答案 maxsum 初始化为 nums[0]而不是 0。对应的算法如下:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n,maxsum,cursum=len(nums),nums[0],0
        for i in range(0,n):
            cursum+=nums[i]
            maxsum=max(maxsum,cursum)      #通过比较求最大 maxsum
            if cursum<0:cursum=0 #若 cursum<0,最大连续子序列从下一个位置开始
        return maxsum
```

上述程序提交时通过,运行时间为 128ms,内存消耗为 29.8MB。

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,大家很容易会想到数学归纳法,数学归纳法是一种数学证明方法,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。

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,问题即证。

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有两个步骤:

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。从中看出凡是能够达到目的的操作都会使结论成立,因此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以遵循。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 表示问题的规模（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，使问题逐渐转化成较低层级或简单的且能解决的问题或已解决的问题。

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 3.4(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？数学归纳法虽然不是归纳法，但是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 直接插入排序

1. 问题描述

有一个整数序列 $a[0..n-1]$，采用直接插入排序实现 a 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 $a[i]$有序插入 $a[0..i-1]$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 $a=(2,5,4,1,3)$，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 $f(a,i)$用于实现 $a[0..i]$(共 $i+1$ 个元素)的递增排序，它是一个大问题，则 $f(a,$

$i-1$)实现 $a[0..i-1]$(共 i 个元素)的排序,它是一个小问题。对应的递推关系如下:

$f(a,i) \equiv$ 不做任何事情　　当 $i=0$ 时

$f(a,i) \equiv f(a,i-1)$;将 $a[i]$有序插入 $a[0..i-1]$中　　其他

显然 $f(a,n-1)$用于实现 $a[0..n-1]$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 a 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 $f(a,k-1)$用于实现 $a[0..k-1]$的递增排序。当 $n=k+1$ 时对应 $f(a,k)$,先调用 $f(a,k-1)$将 $a[0..k-1]$排序,再将 $a[k]$有序插入 $a[0..k-1]$中,这样 $a[0..k]$变成递增有序序列,所以 $f(a,k)$实现 $a[0..k]$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:

```
1  def Insert(a,i):                          #将 a[i]有序插入 a[0..i-1]中
2      tmp=a[i]
3      j=i-1
4      while True:                           #找 a[i]的插入位置
5          a[j+1]=a[j]                       #将关键字大于 a[i]的元素后移
6          j-=1
7          if not (j>=0 and a[j]>tmp):
8              break                         #循环到 a[j]<=tmp 为止
9      a[j+1]=tmp                            #在 j+1 处插入 a[i]
10
11 def InsertSort1(a):                       #迭代算法:直接插入排序
12     n=len(a)
13     for i in range(1,n):
14         if a[i]<a[i-1]:Insert(a,i)        #反序时调用 Insert
```

在实际中有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")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。

Start		
		Finish

图 3.5　3×3 的网格

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。

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}。

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

在上式中分子、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四舍五入得到整数答案。对应的算法如下：

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        return self.comp(m-1,n-1)

    def comp(self,x,y):
        a,b=x+y,min(x,y)
        ans=1.0
        while b>0:
            ans*=1.0*a/b
            a-=1;b-=1
        return round(ans)
```

上述程序提交时通过，执行时间为 40ms，内存消耗为 14.9MB。

说明：在求 C_{x+y}^{x} 时，如果先计算 $(x+y)!$，再计算 $x!\ y!$，最后求两者相除的结果，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 天一共摘了多少个桃子。

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，题目中隐含了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：

```
1  def peaches(n):                #第 n 天的桃子数为 1,求第 1 天的桃子数
2      ans=1
3      for i in range(n-1,0,-1):
4          ans=2*(ans+1)
5      return ans
```

用上述算法求出 $f(1)$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：

```
def Iterative():                  #迭代法算法的框架
    为迭代变量赋初值
    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，排序表的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，有助于大家理解算法的正确性。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。

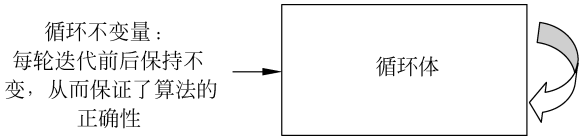

图 3.6　利用循环不变量证明算法的正确性

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，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”。

【例 3-3】　采用循环不变量证明 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“$a[0..i-1]$是递增有序的”。

初始化：在循环时 i 从 1 开始，循环之前 $a[0..0]$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 $a[i]$排序的这一趟，之前 $a[0..i-1]$是递增有序的：

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

② 如果 $a[i]<a[i-1]$，即反序，则在 $a[0..i-1]$中从后向前找到第一个 $a[j]\leqslant a[i]$，将 $a[j+1..i-1]$均后移一个位置，并且将原 $a[i]$放在 $a[j+1]$位置，这样结束后循环不变量 $a[0..i]$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 $a[0..n-1]$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 3.2.2 节中直接插入排序算法的正确性。

后面的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 $a[0..n-1]$，采用简单选择排序实现 a 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，$a[0..i-1]$是有序区，$a[i..n-1]$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 $a[i..n-1]$无序区采用简单比较找到最小元素 $a[\text{minj}]$，通过交换将其放到 $a[i]$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 $a=(2,5,4,1,3)$，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 $f(a,i)$用于实现 $a[i..n-1]$(共 $n-i$ 个元素)的递增排序,它是一个大问题,则 $f(a,i-1)$ 实现 $a[i-1..n-1]$(共 $n-i-1$ 个元素)的排序,它是一个小问题。对应的递推关系如下:

$f(a,i)$ ≡ 不做任何事情　　当 $i=n-1$ 时

$f(a,i)$ ≡ 在 $a[i..n-1]$中选择最小元素交换到 $a[i]$位置; $f(a,i+1)$　　否则

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

```
1  def Select(a,i):                    #一趟排序:在 a[i..n-1]中选择最小元素交换到 a[i]位置
2      n,minj=len(a),i                 #minj 表示 a[i..n-1]中最小元素的下标
3      for j in range(i+1,n):          #在 a[i..n-1]中找最小元素
4          if a[j]<a[minj]:minj=j
5      if minj!=i:                     #若最小元素不是 a[i]
6          a[minj],a[i]=a[i],a[minj]       #交换
7
8  def SelectSort1(a):                 #迭代法:简单选择排序
9      for i in range(0,len(a)):       #进行 n-1 趟排序
10         Select(a,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 若删除的两个元素均不是 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 若删除的两个元素中有一个是 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。对应的迭代算法如下:

```
1  class Solution:
2      def majorityElement(self, nums: List[int]) -> int:
```

```
    n=len(nums)
    if n==1:return nums[0]
    c,cnt=nums[0],1
    i=1
    while i<n:
        if nums[i]==c:          #选择两个元素(R[i],c)
            cnt+=1              #相同时累加次数
        else:
            cnt-=1              #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+=1
                c=nums[i];cnt+=1
        i+=1
    return c
```

上述程序提交时通过，运行时间为 44ms，内存消耗为 16.8MB。对应算法的时间复杂度为 $O(n)$，空间复杂度为 $O(1)$。

扫一扫

视频讲解

3.3.4 求幂集

1. 问题描述

给定一个正整数 $n(n\geqslant 1)$，给出求{1,2,…,n}的幂集的递推关系和迭代算法。例如，当 $n=3$ 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 3.7 所示，其中 M_1、M_2 和 M_3 分别表示{1}、{1,2}和{1,2,3}的幂集。

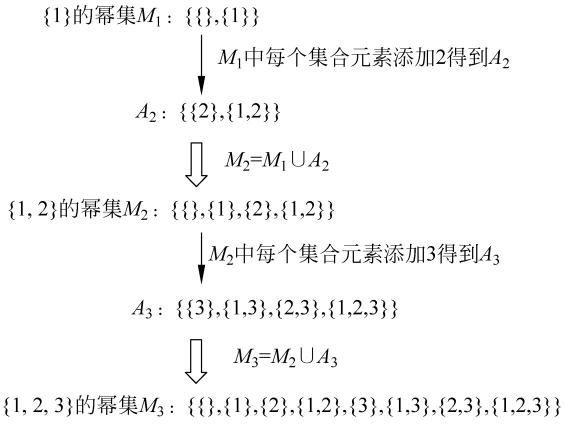

图 3.7 求{1,2,3}的幂集的过程

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$\{1,2,\cdots,i\}(i\geqslant 1$,共 i 个元素)的幂集(是一个两层集合),为大问题,则 M_{i-1} 为$\{1,2,\cdots,i-1\}$(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

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

则

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

这样求$\{1,2,\cdots,n\}$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,采用双层表存放,其中每个列表元素表示幂集中的一个集合。大问题即求$\{1,2,\cdots,i\}$的幂集,用 Mi 变量表示,小问题即求$\{1,2,\cdots,i-1\}$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 增加为 $i+1$。对应的迭代法算法如下:

```
1  import copy
2  def appendi(Mi_1,e):                    #向 Mi_1 中每个集合元素的末尾添加 e
3      Ai=Mi_1
4      for x in Ai:x.append(e)
5      return Ai
6
7  def subsets(n):                         #迭代法:求{1,2,…,n}的幂集
8      Mi_1=[[],[1]]                       #Mi_1 初始化为{1}的幂集
9      if n==1:return Mi_1                 #处理特殊情况
10         for i in range(2,n+1):
11             Mi=copy.deepcopy(Mi_1)
12             Ai=appendi(Mi_1,i)
13             for x in Ai:Mi.append(x)    #将 Ai 中的所有集合元素添加到 Mi 中
14             Mi_1=copy.deepcopy(Mi)
15     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中所有可能的子集(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$\{\{\},\{1\},\{2\},\{1,2\},\{3\},\{1,3\},\{2,3\},\{1,2,3\}\}$。

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$\{1,2,\cdots,n\}$的幂集改为求 nums

[0..$n-1$]的幂集，即设 M_i 为 nums[0..i]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

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

则

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

这样求 nums 的幂集的递推关系如下：

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

$M_i = M_{i-1} \cup A_i$　　当 $i>0$ 时

对应的迭代法算法如下：

```
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        Mi=[]                                   #存放幂集
        Mi_1=[[],[nums[0]]]
        n=len(nums)
        if n==1:return Mi_1                     #处理特殊情况
        for i in range(1,n):
            Mi=copy.deepcopy(Mi_1)
            Ai=self.appendi(Mi_1,nums[i])
            for x in Ai:Mi.append(x)            #将 Ai 中的所有集合元素添加到 Mi 中
            Mi_1=copy.deepcopy(Mi)              #用新值替换旧值
        return Mi

    def appendi(self,Mi_1,e):                   #向 Mi_1 中每个集合元素的末尾添加 e
        Ai=Mi_1                                 #浅复制
        for x in Ai:x.append(e)
        return Ai
```

上述程序提交时通过，执行用时为 44ms，内存消耗为 15.3MB。

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，所以下面主要讨论直接递归算法。

递归算法通常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(例如 C/C++、Java 和 Python 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化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,在设计递归法算法时首先用归纳法建立递推关系,这里称为递归模型,在此基础上直接转换为递归法算法。

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:

$f(1)=1$

$f(n)=n\times f(n-1)$ 当 $n>1$ 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 $f(s)$(即 $f(s)$用于求解大问题)进行分析,假设出合理的小问题 $f(s')$(即 $f(s')$用于求解小问题)。

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(例如 $f(1)$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. 递归法算法的框架

在递归模型中递归体是核心,用于将小问题的解通过合并操作产生大问题的解,通常递归框架分为两种。

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

```
def 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的解,即先合后递,也就是在递去的过程中解决问题,其框架如下:

```
def 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
        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 3-4】 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 r1 和 r2 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(r1,r2)表示二叉树 r1 和 r2 是否相同,它们的左、右子树的判断是两个小问题,如图 3.8 所示。依题意,对应的递归模型如下:

f(r1,r2)=True　　当 r1 和 r2 均为空时

f(r1,r2)=False　　当 r1 和 r2 中一个为空另一个非空时

f(r1,r2)=False　　当 r1 和 r2 均不空但是结点值不相同时

f(r1,r2)=f(r1.left,r2.left) && f(r1.right,r2.right)　　其他

对应的递归算法如下:

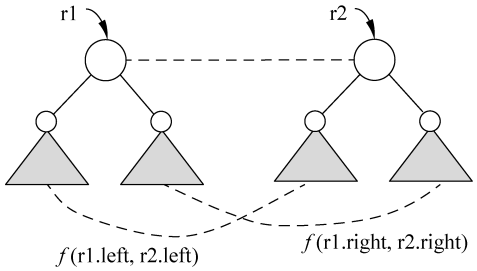

图 3.8　二叉树的两个小问题

```
1   def same(r1,r2):                    #递归算法:判断 r1 和 r2 是否相同
2       if r1==None and r2==None:
3           return True
4       elif r1==None or r2==None:
5           return False
6       if r1.val!=r2.val:
7           return False
8       leftans=same(r1.left,r2.left)   #递归调用 1
9       rightans=same(r1.right,r2.right) #递归调用 2
10      return leftans and rightans
```

后面通过几个应用进一步讨论递归法算法的设计过程。

3.4.2 冒泡排序

1. 问题描述

有一个整数序列 $a[0..n-1]$,采用冒泡排序实现 a 的递增有序排序。冒泡排序的过程是,i 从 0 到 $n-2$ 循环,$a[0..i-1]$是有序区,$a[i..n-1]$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 $a[i..n-1]$无序区采用冒泡方式将最小元素放在 $a[i]$位置。其迭代算法如下:

```
1  def Bubble(a,i):                    #一趟排序:在 a[i..n-1]中冒泡最小元素到 a[i]中
2      exchange=False
3      for j in range(len(a)-1,i,-1):  #无序区中的元素比较,找出最小元素
4          if a[j-1]>a[j]:             #当相邻元素反序时
5              a[j],a[j-1]=a[j-1],a[j]  #a[j]与 a[j-1]进行交换
6              exchange=True           #本趟排序发生交换置 exchange 为真
7      return exchange                 #返回是否存在交换
8
9  def BubbleSort1(a):                 #迭代算法:冒泡排序
10     for i in range(0,len(a)-1):     #进行 n-1 趟排序
11         if not Bubble(a,i):return   #本趟未发生交换时结束算法
```

现在要求采用递归实现冒泡排序算法。

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 $a=(2,5,4,1,3)$,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)　　#初始有序区为空

$i=0$:([**1**],2,5,4,3)　　#调用 Bubble 从 $a[0..4]$中冒泡最小元素到 $a[0]$

$i=1$:([1,**2**],3,5,4)　　#调用 Bubble 从 $a[1..4]$中冒泡最小元素到 $a[1]$

$i=2$:([1,2,**3**],4,5)　　#调用 Bubble 从 $a[2..4]$中冒泡最小元素到 $a[2]$

$i=3$:([1,2,3,**4**],5)　　#调用 Bubble 从 $a[3..4]$中冒泡最小元素到 $a[3]$

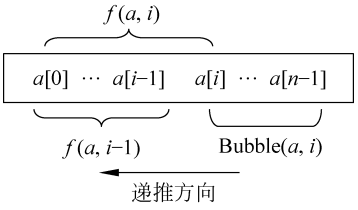

图 3.9　大、小问题的表示及其递推分析 1

解法 1:采用先递后合的递归算法。设 $f(a,i)$用于实现 $a[0..i]$(共 $i+1$ 个元素)的递增排序,为大问题,则 $f(a,i-1)$ 实现 $a[0..i-1]$(共 i 个元素)的排序,为小问题,如图 3.9 所示。

当执行 $f(a,i-1)$求解小问题后,$a[0..i-1]$变为全局有序区,在 $a[i..n-1]$中冒泡最小元素到 $a[i]$位置(调用前面的 Bubble 算法实现),则 $a[0..i]$变为更大的全局有序区,这就是大问题的解,即 $f(a,i)$。其递推方向是从后向前。当 $i=-1$ 时,$a[0..i]$为空,看成有序的。对应的递推模型如下:

$f(a,i)\equiv$ 不做任何事情　　当 $i=-1$ 时

$f(a,i)\equiv f(a,i-1)$; Bubble(a,i);　　否则

显然 $f(a,n-1)$用于实现 $a[0..n-1]$递增排序,由于这样排序后最后一个元素 $a[n-1]$一定是最大元素,所以调用 $f(a,n-2)$就可以实现 $a[0..n-1]$的递增排序。对应的递归算法(未考虑一趟没有发生交换时提前结束的情况)如下:

```
1  def BubbleSort21(a,i):              #递归冒泡排序 1
2      if i==-1:return                 #满足递归出口条件
3      BubbleSort21(a,i-1)             #递归调用
4      Bubble(a,i)
5
```

```
6   def BubbleSort2(a):                    #冒泡排序
7       BubbleSort21(a,len(a)-2)
```

解法 2：采用先合后递的递归算法。设 $f(a,i)$用于实现 $a[i..n-1]$(共 $n-i$ 个元素)的递增排序，它是大问题，则 $f(a,i+1)$ 实现 $a[i+1..n-1]$(共 $n-i-1$ 个元素)的排序，它是小问题，如图 3.10 所示。

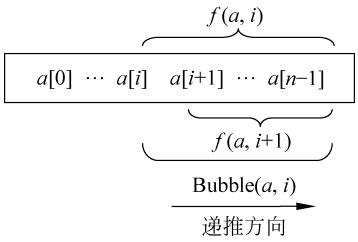

图 3.10 大、小问题的表示及其递推分析 2

当执行 $f(a,i+1)$求解小问题后，$a[i+1..n-1]$变为全局有序区，在 $a[i..n-1]$中冒泡最小元素到 $a[i]$位置(调用前面的 Bubble 算法实现)，则 $a[i..n-1]$变为更大的全局有序区，这就是大问题的解，即 $f(a,i)$。其递推方向是从前向后。当 $i=n-1$ 时，$a[n-1..n-1]$中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：

$f(a,i)\equiv$ 不做任何事情　　当 $i=n-1$ 时

$f(a,i)\equiv$ Bubble(a,i)；$f(a,i+1)$；　　否则

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：

```
1   def BubbleSort31(a,i):                 #递归冒泡排序 2
2       if i==len(a)-1:return              #满足递归出口条件
3       if Bubble(a,i):BubbleSort31(a,i+1) #一趟中发生交换时递归调用
4
5   def BubbleSort3(a):                    #冒泡排序
6       BubbleSort31(a,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 $1\sim 3$ 的全排列的过程如图 3.11 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 $1\sim 2$ 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 $1\sim 3$ 的全排列{{1,2,3},{1,3,2},{3,1,2},{2,1,3},{2,3,1},{3,2,1} }。

归纳起来，设 P_i 表示 $1\sim i(i\geqslant 1$，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 $1\sim i$ 的某个排列)，为大问题，则 P_{i-1} 为 $1\sim i-1$(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 $s_0s_1\cdots s_{i-2}$(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 $i-2$，…，

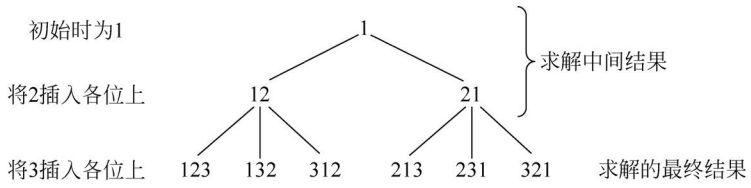

图 3.11　求 1～3 的全排列的过程

位置 0),定义 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

$$\text{CreatePi}(s,i) = \bigcup_{0 \leqslant j \leqslant \text{len}(s)} \text{Insert}(s,i,j)$$

则

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

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

用双层表存放 1～n 的全排列。对应的递归算法如下:

```
import copy
def Insert(s,i,j):                          #在 s 的位置 j 插入 i
    tmp=copy.deepcopy(s)
    tmp.insert(j,i)                         #位置 j 插入整数 i
    return tmp

def CreatePi(s,i):                          #在 s 集合中的 i-1 到 0 位置插入 i
    tmp=[]
    for j in range(len(s),-1,-1):           #在 s(含 i-1 个整数)的每个位置插入 i
        s1=Insert(s,i,j)
        tmp.append(s1)                      #将 s1 添加到 Pi 中
    return tmp

def perm11(n,i):                            #递归算法
    if i==1:
        return [[1]]
    else:
        Pi=[]                               #存放 1～i 的全排列
        Pi_1=perm11(n,i-1)                  #求出 Pi_1
        for x in Pi_1:
            tmp1=CreatePi(x,i)              #在 x 集合中插入 i 得到 tmp1
            for y in tmp1:Pi.append(y)      #将 tmp1 的全部元素添加到 Pi 中
    return Pi

def perm1(n):                               #用递归法求 1～n 的全排列
    return perm11(n,n)
```

扫一扫

视频讲解

3.4.4　实战——字符串解码(LeetCode394★★)

1. 问题描述

给定一个经过编码的有效字符串 s,设计一个算法返回 s 解码后的字符串,编码规则是

用“k[encoded_string]”表示方括号内的 encoded_string（仅包含小写字母）正好重复 k（k 保证为正整数）次。例如，s="3[a]2[bc]"，答案为"aaabcbc"，若 s="abc3[cd]xyz"，答案为"abccdcdcdxyz"。

2. 问题求解

用 ans 存放 s 展开的字符串（初始为空），用整型变量 i 从 0 开始遍历 s（将 i 设计为全局变量），一边遍历一边展开。采用递归法求解，设 $f(s)$求字符串 s 解码后的字符串。

（1）递归出口：对于不包含数字和括号的字符串 s，直接连接到 ans 中。例如，s="abc"，则 ans="abc"。

（2）递归体：依题意，s 是一个合法的字符串，分为以下几种情况。

① s="k[encoded_string]"（以字符']'结尾），其中 encoded_string 是一个合法的字符串，先提取整数 k，再调用 f(encoded_string)求出小问题的结果，则 ans 为 k 个 f(encoded_string)的连接。例如 s="3[a]"，则 ans="aaa"。

② s="$s_1\cdots s_n$"，其中 s_i 是合法的子串，则 ans=$f(s_1)+\cdots+f(s_n)$。例如，s="abc3[cd]xyz"，则 ans="abc"+"cdcdcd"+"xyz"="abccdcdcdxyz"。

对应的递归程序如下：

```
class Solution:
    def decodeString(self, s: str) -> str:                    #求解算法
        self.i=0                                               #类变量 i 从 0 开始遍历 s
        return self.unfold(s)

    def unfold(self,s):                                        #递归算法
        ans=""
        while self.i<len(s) and s[self.i]!=']':                #处理到']'为止
            if s[self.i]>='a' and s[self.i]<='z':              #遇到字母
                ans+=s[self.i]; self.i+=1
            else:
                k=0
                while self.i<len(s) and s[self.i]>='0' and s[self.i]<='9':
                    k=k*10+ord(s[self.i])-ord('0');self.i+=1   #数字符转为整数 k
                self.i+=1                                      #数字字符后面为'[',则跳过该'['
                tmp=self.unfold(s)                             #求子串解码结果 tmp
                self.i+=1                                      #后面是一个']',跳过该']'
                while k>0:                                     #连接 tmp 字符串 k 次
                    ans+=tmp;k-=1
        return ans                                             #s 处理完毕返回 ans
```

上述程序提交时通过，执行用时为 32ms，内存消耗为 15MB。

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）来表示，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,直到达到最前面的初始条件为止,就得到了问题的解。

【例 3-5】 求解梵塔问题的递归算法如下,分析移动 n 盘片的时间复杂度。

```
1  def Hanoi(n,x,y,z):
2      if n==1:
3          print("将盘片%d从%c搬到%c"%(n,x,z))
4      else:
5          Hanoi(n-1,x,z,y)
6          print("将盘片%d从%c搬到%c"%(n,x,z))
7          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)。

$T(n)=1$ 当 $n=1$ 时

$T(n)=2T(n-1)+1$ 当 $n>1$ 时

则

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

所以移动 n 盘片的时间复杂度为 $O(2^n)$。

3.5.2 递归树方法

递归树方法是直接展开法的一种图形表述,用递归树求解递推式的基本过程是先展开递推式,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 3-6】 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 3.12(a)所示,将 $T(n)$展开一次的结果如图 3.12(b)所示,再展开 $T(n/2)$的结果如图 3.12(c)所示,以此类推,构造的递归树如图 3.13 所示。从中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。

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 $2^{k-1}(n/2^{k-1})^2=n^2/$

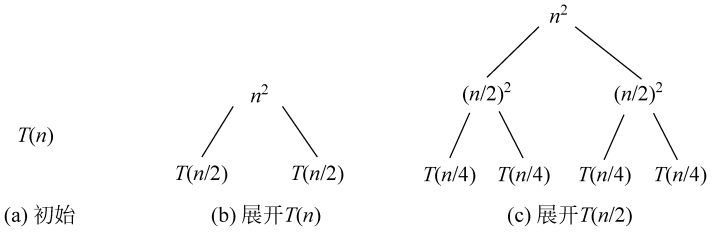

图 3.12 展开两次的结果

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：

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

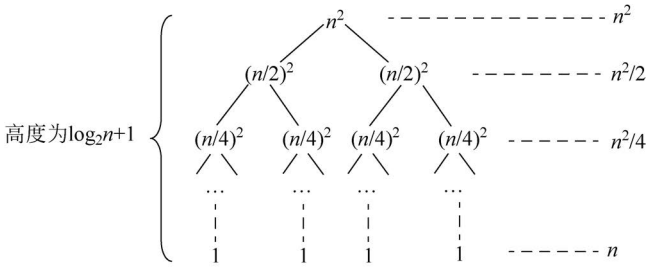

图 3.13 一棵递归树

【例 3-7】 分析以下递推式的时间复杂度：

$T(n)=1$ 当 $n=1$ 时

$T(n)=T(n/3)+T(2n/3)+n$ 当 $n>1$ 时

解 构造的递归树如图 3.14 所示，不同于图 3.13 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，每走一步问题的规模减小为原来的 1/3（问题规模变小的速度相对较快），最右边的路径是最长路径，每走一步问题的规模减小为原来的 2/3（问题规模变小的速度相对较慢）。

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的数目），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 $T(n)\leqslant n(\log_{3/2}n+1)=O(n\log_{3/2}n)=O(n\log_2 n)$，即该递推式的时间复杂度是 $O(n\log_2 n)$。

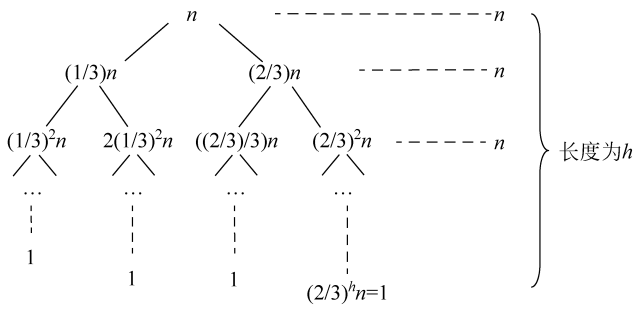

图 3.14 一棵递归树

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：

$T(1)=c$

$T(n)=aT(n/b)+f(n)$　　当 $n>1$ 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的解得到答案的时间。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 $f(n)$多项式地小于 $n^{\log_b a}$（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 $f(n)$多项式地大于 $n^{\log_b a}$（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 种情况都是拿 $f(n)$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 $n^{\log_b a}$ 的阶较大，则 $T(n)=\Theta(n^{\log_b a})$；情况③是函数 $f(n)$的阶较大，则 $T(n)=\Theta(f(n))$；情况②是两个函数的阶一样大，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 $n^{\log_b a}$ 的阶小，而且必须是多项式地比 $n^{\log_b a}$ 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 $n^{\log_b a}$ 的阶大，而且必须是多项式地比 $n^{\log_b a}$ 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 个子问题的再分解和再合并所需要的时间最多与原问题的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。

当然还有一点很重要，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 $f(n)$小于但不是多项式地小于 $n^{\log_b a}$。类似地，在情况②和③之间也有一个间隙，即 $f(n)$大于但不是多项式地大于 $n^{\log_b a}$。如果函数 $f(n)$落在这两个间隙之一中，或者虽然有 $f(n)=O(n^{\log_b a+\varepsilon})$，但是正规性条件不满足，那么主方法将无能为力。

【例 3-8】 采用主定理求以下递推式的时间复杂度：

$T(n)=1$　　当 $n=1$ 时

$T(n)=4T(n/2)+n$　　当 $n>1$ 时

解 这里 $a=4$，$b=2$，$n^{\log_b a}=n^2$，$f(n)=n=O(n^{\log_b a-\varepsilon})$，$\varepsilon=1$，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 3-9】 采用主方法求以下递推式的时间复杂度：

$T(n)=1$ 当 $n=1$ 时

$T(n)=3T(n/4)+n\log_2 n$ 当 $n>1$ 时

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的情况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 3-10】 采用主定理和直接展开法求以下递推式的时间复杂度：

$T(n)=1$ 当 $n=2$ 时

$T(n)=2T(n/2)+(n/2)^2$ 当 $n>1$ 时

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。如果递推式如下：

$T(1)=c$

$T(n)=aT(n/b)+cn^k$ 当 $n>1$ 时

其中 a、b、c、k 都是常量，可以这样简化主定理：

① 若 $a>b^k$，则 $T(n)=\Theta(n^{\log_b a})$。

② 若 $a=b^k$，则 $T(n)=\Theta(n^k\log_b n)$。

③ 若 $a<b^k$，则 $T(n)=\Theta(n^k)$。

以上介绍的递推式求解方法将在第 4 章有关分治法算法的分析中大量用到。

习题 3

扫一扫

练习题

扫一扫

自测题