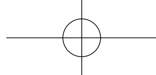

PYTHON

第 5 章　递归与回溯算法

递归是一种重要的算法结构和编程技巧，常用来实现回溯法、分治法（见第 9 章）和动态规划算法（见第 10 章），在分形几何学等众多领域中也有广泛应用。递归算法的主要思路是，在保证问题性质不变的前提下把大问题转化为小问题（这个过程为“递”），不断地减小问题规模，直到问题规模小到可以直接处理，然后再一层一层地返回并合成得到原问题的解（这个过程为“归”）。

在程序设计中，如果一个函数中又调用这个函数自己，自己又调用自己，……，这样的函数称为递归函数。在具体实现时，递归算法有个限制，那就是递归深度不能太深。虽然在 Python 中可以通过内置模块 sys 的函数 setrecursionlimit() 来修改最大递归深度，但也不能设置得太大，因为线程栈的大小还受操作系统的限制，并且参数也不能超过 C 语言整数大小的限制，否则会抛出异常并提示“OverflowError: Python int too large to convert to C int”。

回溯是一种重要的算法思想，根据问题描述和给定的数据创建一棵隐式的树或图并进行深度优先搜索（这个技术在后面多个章节的例题中都有应用），沿着一条路往前试探，能进则进，不能进就换旁边一条路再试，无路可换就往回退一步到上一个路口再换一条路继续尝试，不停地选择、尝试和撤回。回溯法类似于穷举法，试图在所有可能的解中寻找最优解，但区别在于回溯法会构造约束函数（不同问题的约束函数也不同，具体问题具体分析）进行剪枝，提前结束不可能得到答案的搜索。如果没有剪枝，回溯法就变成了穷举法。

具体编程实现时，回溯往往通过递归来实现（也可以使用非递归实现），时间复杂度一般为指数级别，不会记录已经计算的子问题解，已经计算过的结果无法得到有效复用，存在大量的重复计算，除非采用额外的技巧。

回溯法代码可以抽象为下面的框架：

```
result = []
def backtrack(路径, 选择列表)
    if 路径满足结束条件
        result.add(路径)
        return
```

```
for 遍历下一步可能的每个选择
    做选择，如果需要的话可以在这里进行剪枝，某些情况下不再递归调用函数
    backtrack( 更新后的路径 , 更新后的选择列表 )
    撤销选择
```

5.1　数学类问题算法设计与应用

例 5-1　计算 3 的 n 次方。

在 Python 中使用乘法运算符“*”、幂运算符“**”或者内置函数 pow() 可以解决这个问题，这里主要介绍递归算法的应用和优化。下面的代码把幂运算转换成了递归函数和加法，并且使用空间换时间来减少重复计算。标准库 functools 中的修饰器函数 lru_cache() 用来给函数增加辅助缓存，用来保存一定数量的中间结果，需要再次计算时如果缓冲区中已经存在就直接使用，效率提升非常明显。如果缓冲区满了就把最近使用次数最少的一个数据删除，腾出空间来存储新的数据。

```
from functools import lru_cache

# 缓冲区大小范围为 1~128，可以删除下一行 @lru_cache(maxsize=64) 并比较速度差别
@lru_cache(maxsize=64)
def func(n):
    if n == 0:
        return 1
    # 下面表达式写成 func(n-1) * 3 的效率更高，这里主要体现缓冲区的作用
    return func(n-1) + func(n-1) + func(n-1)

print(func(5), func(25), func(225), sep=',')
```

例 5-1

例 5-2　计算斐波那契数列中第 n 个数字。

已知斐波那契数列中的数字具有以下特点：

$$F(n)=\begin{cases}1, & n=1,2\\ F(n-1)+F(n-2), & n>2\end{cases}$$

根据上面公式使用递归算法计算数列中的值，属于从上向下的设计，存在大量重复计算。以 $F(5)$ 为例，其计算过程可以使用图 5-1 中的二叉树来描述。从图中容易看出，这样的求解过程存在大量的重复计算，效率非常低。缓解这一问题的一种方法是利用缓存适当记忆中间结果，基于记忆的搜索是动态规划算法的实现方式之一，详见第 10 章。

Psyzw3-7.indd　102　　2025/5/8　17:01:02

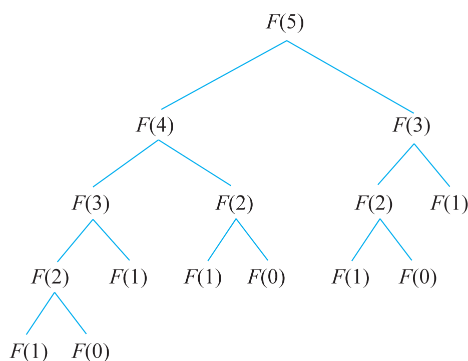

图 5-1　生成斐波那契数列的递归算法对应的二叉树

```
from functools import lru_cache

@lru_cache(maxsize=64)
def fib(n):
    if n <= 2:
        return 1
    return fib(n-1) + fib(n-2)

print(fib(5), fib(25), fib(225), fib(555), sep=',')
```

下标的代码使用列表作为自定义缓冲区来保存中间数据，感兴趣的读者参考其中的思路改为使用字典作为缓冲区。

```
def fib(n):
    # 第一个元素不使用
    buffer = [0] * (n+1)
    buffer[1] = 1
    def nested(n):
        if n < 2:
            return buffer[n]
        if buffer[n] == 0:
            buffer[n] = nested(n-1) + nested(n-2)
        return buffer[n]
    nested(n)
    return buffer[n]
```

例 5-3　使用递归算法计算杨辉三角形中的数字。

杨辉三角形的递推算法实现见第 4 章，这里使用递归算法重新实现。杨辉三角形中的数字与组合数、二项式系数有关，是一个非常有价值的三角形，把“数形结合”引入了计算数学，通过二项式展开式系数计算公式来计算系数称为“式算”，通过杨辉三角形计算

Psyzw3-7.indd　103　　2025/5/8　17:01:05

系数称为“图算”。假设行号和列号都从 0 开始，那么杨辉三角形第 m 行的数字就是二项式 $(x+y)^m$ 展开后的每项系数，第 m 行 n 列的数字就是组合数 C_m^n。

根据帕斯卡公式 $C_n^i=C_{n-1}^i+C_{n-1}^{i-1}$，可以把 C_n^i 看作二叉树的根，把 C_{n-1}^i 和 C_{n-1}^{i-1} 分别看作左右子节点，这两个节点又可以按照同样的规律得到各自的左右子节点，如图 5-2 所示。随着二叉树的向下扩展，左子节点最终会变成 $C_i^i=1$，右子节点最终会变成 $C_{n-i}^0=1$，这两个特殊情况可以作为递归结束条件。

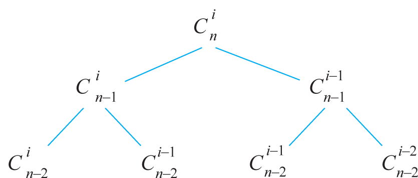

图 5-2　帕斯卡公式示意图

从图 5-2 可知，这棵二叉树上有两个相同的节点 C_{n-2}^{i-1}，并且它们分裂的方式也完全相同，也就是说存在两棵一模一样的子树，这就意味着存在重复计算。对于类似的问题，适合使用标准库 functools 提供的修饰器函数 lru_cache() 来优化。

```
from functools import lru_cache

@lru_cache(maxsize=64)
def cni(n, i):
    if n==i or i==0:
        return 1
    return cni(n-1,i) + cni(n-1,i-1)

def yanghui(num):
    for n in range(num):
        for i in range(n+1):
            print(str(cni(n, i)).ljust(4), end=' ')
        print()
yanghui(8)
```

例 5-4　连接若干自然数为一个大自然数。

在例 4-13 中用来连接自然数的几个函数都属于递推和迭代算法，下面的函数使用递归法和分治法（见第 9 章）解决同样的问题，并且速度比例 4-13 中 func2()、func3()、func4()、func5() 略快，原因在于减少了大数相乘的次数，转换为若干次小自然数相乘。例如，对于 [1,2,3,4]，连接过程为 [1*10**1+2,3*10**1+4]==>[12,34]==>[12*10**2+34]==>1234。

下面函数延续了例 4-13 中的函数编号，读者可以很容易地放到一起来测试和比较几个函数的执行速度。

Psyzw3-7.indd 104　　2025/5/8 17:01:05

```python
from math import log10

def func7(data):
    n = len(data)
    if n == 0:
        return 0
    if n == 1:
        return data[0]
    if n == 2:
        return data[0]*10**int(log10(data[1])+1) + data[1]
    middle = n // 2
    pre, aft = func7(data[:middle]), func7(data[middle:])
    return pre*10**int(log10(aft)+1) + aft
```

5.2　其他类问题算法设计与应用

例 5-5　使用递归算法求解小明爬楼梯问题，问题描述与分析见例 4-20。

```python
from functools import lru_cache

@lru_cache(maxsize=64)
def climb_stairs2(n):
    first3 = {1:1, 2:2, 3:4}
    return (first3.get(n) or
            climb_stairs2(n-1) + climb_stairs2(n-2) + climb_stairs2(n-3))
```

例 5-6　使用递归法判断回文。

回文的其他判断方法见例 2-17 和例 3-34，这里给出递归法的两种不同实现。

```python
def is_palindrome3(text):
    if len(text) <= 1:
        return True
    if text[0] != text[-1]:
        return False
    return is_palindrome3(text[1:-1])

def is_palindrome4(text, start=None, end=None):
    if start==None or end==None:
        start, end = 0, len(text)-1
    if start >= end:
```


Psyzw3-7.indd 105　2025/5/8 17:01:05

```
        return True
    if text[start] != text[end]:
        return False
    return is_palindrome4(text, start+1, end-1)
```

例 5-7　求解汉诺塔问题中盘子的移动过程。

汉诺塔由法国数学家弗朗索瓦·爱德华·阿纳托尔·卢卡斯（François Édouard Anatole Lucas）于 1883 年提出，其灵感来自一个传说。从前有座山，山上有个庙，庙里有一个梵塔，塔内有 3 个柱子 A、B、C，在 A 柱子上有 64 个盘子，盘子大小不等，大的盘子在下，小的盘子在上。有一个和尚想把这 64 个盘子从 A 柱子移到 C 柱子，每次只能移动一个盘子，移动过程中可以利用 B 柱子，任何时刻 3 个柱子上的盘子都必须始终保持大盘在下、小盘在上的顺序。如果只有一个盘子，则不需要利用 B 柱子，直接将盘子从 A 移动到 C 即可。和尚应该如何完成这个任务呢?

假设庙里有很多和尚，对其从 1 开始编号。1 号和尚的想法是让 2 号和尚帮忙把最上面的 63 个盘子按照规则从 A 柱子拿到 B 柱子，然后自己把最下面的那个盘子从 A 柱子拿到 C 柱子，最后再请 2 号和尚帮忙把 B 柱子上的 63 个盘子拿到 C 柱子上，就完成任务了，如图 5-3~ 图 5-6 所示。

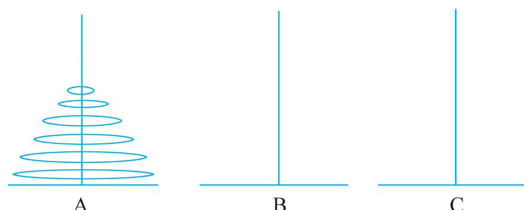

图 5-3　汉诺塔问题示意图（初始状态）

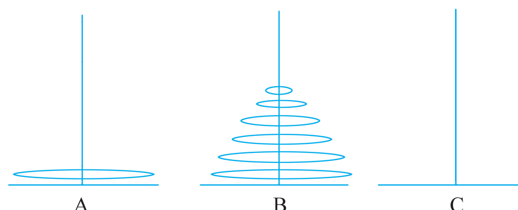

图 5-4　移动 A 柱子上面的 n–1 个盘子到 B 柱子，可以借助 C 柱子

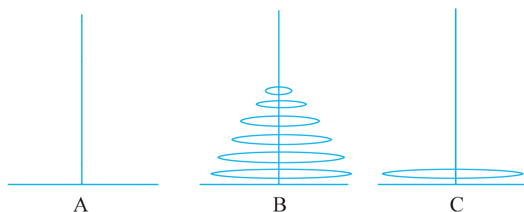

图 5-5　移动 A 柱子上最下面的 1 个盘子到 C 柱子

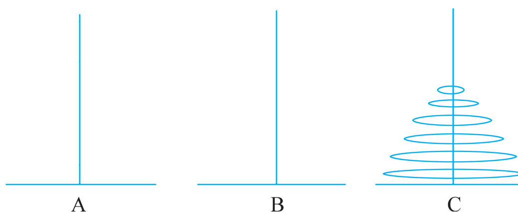

图 5-6　移动 B 柱子的 n–1 个盘子到 C 柱子，可以借助 A 柱子

2 号和尚面对 63 个盘子也很为难，于是找来 3 号和尚请他帮忙把最上面的 62 个盘子按照规则拿到 C 柱子，然后自己把第 63 个盘子拿到 B 柱子，3 号和尚再帮忙把 C 柱子上的 62 个盘子拿到 B 柱子，这样 2 号和尚也完成任务了。以此类推，每个和尚都请别人帮自己把上面的 n–1 个盘子拿到正确的柱子上，自己只需要处理第 n 个盘子就可以了。问

Psyzw3-7.indd 106　　2025/5/8 17:01:08

题逐步简化，最后一个和尚只需要处理最顶上的一个盘子即可，不需要再请别人帮忙，可以直接完成任务。

这是一个非常巨大的工程，是一个不可能完成的任务。根据数学知识我们可以知道，移动 n 个盘子需要 2^n-1 步，64 个盘子需要 18446744073709551615 步。如果每步需要一秒的话，至少也需要 584942417355.072 年才能完成。

下面的程序使用递归算法实现了盘子移动过程的求解。

```
def hannoi1(num, src, dst, temp=None):
    # 确认参数类型和范围
    assert type(num)==int and num>0, 'num 必须为正整数'
    # 声明用来记录移动次数的变量为全局变量
    global times
    # 只有一个盘子需要移动，这也是函数递归调用的结束条件
    if num == 1:
        print(f'The {times} Times move:{src}==>{dst}')
        times = times + 1
    else:
        # 递归调用函数自身，先把除最后一个盘子之外的所有盘子移动到临时柱子上
        hannoi1(num-1, src, temp, dst)
        # 把最后一个盘子直接移动到目标柱子上
        hannoi1(1, src, dst)
        # 把除最后一个盘子之外的其他盘子从临时柱子上移动到目标柱子上
        hannoi1(num-1, temp, dst, src)

# 用来记录移动次数的变量
times = 1
# A 表示最初放置盘子的柱子，C 是目标柱子，B 是临时柱子
hannoi1(3, 'A', 'C', 'B')
```

下面程序改用字典并输出了每次移动后 3 根柱子上的盘子，算法核心仍然是递归。

```
def hannoi2(num, src, dst, temp=None):
    if num < 1:
        return
    global times
    # 递归调用函数自身，先把除最后一个盘子之外的所有盘子移动到临时柱子上
    hannoi2(num-1, src, temp, dst)
    # 移动最后一个盘子
    print(f'The {times} Times move:{src}==>{dst}')
    towers[dst].append(towers[src].pop())
    for tower in 'ABC':
        # 输出 3 根柱子上的盘子
```


Psyzw3-7.indd 107 2025/5/8 17:01:08

```
        print(tower, ':', towers[tower])
    times = times + 1
    # 把除最后一个盘子之外的其他盘子从临时柱子上移动到目标柱子上
    hannoi2(num-1, temp, dst, src)

times, n = 1, 3
# 初始状态，所有盘子都在 A 柱上
towers = {'A':list(range(n, 0, -1)), 'B':[], 'C':[]}
# n 表示盘子数量，A 表示最初放置盘子的柱子，C 是目标柱子，B 是临时柱子
hannoi2(n, 'A', 'C', 'B')
```

下面程序使用非递归算法重新求解汉诺塔问题，可以解除输出语句的注释方便理解。

```
def hannoi3(n):
    # top 表示 3 个柱上的盘子数量，tower 表示 3 个柱上的盘子，初始时盘子都在第一个柱上
    top, tower = [3,0,0], [[n+1-i,n+1,n+1] for i in range(n+1)]
    # print(*tower, sep='\n')
    b, bb, min_ = (n%2==1, True, 0)
    # 移动盘子，直到所有盘子都移动到中间的柱上
    while top[1] < n:
        # print(top)
        if bb:
            x = min_
            # n 为奇数时顺时针移动盘子，偶数时逆时针移动盘子
            y = (x+1) % 3 if b else (x+2) % 3
            min_, bb = y, False
        else:
            x, y, bb = (min_+1)%3, (min_+2)%3, True
            if tower[top[x]][x] > tower[top[y]][y]:
                x, y = y, x
        # 从 x 柱子上移动一个盘子到 y 柱子上
        print(chr(65+x), chr(65+y), sep='==>')
        tower[top[y]+1][y] = tower[top[x]][x]
        # 更新每个柱子上盘子数量
        top[x], top[y] = top[x] - 1, top[y] + 1
    # print(top)
    # print(*tower, sep='\n')

hannoi3(3)
```

下面代码使用了同样的算法，但更加简洁，原始代码由国防科技大学刘万伟老师提供，本书略做修改。

Psyzw3-7.indd 108 2025/5/8 17:01:08

```
def hannoi4(n):
    # L 用来记录移动过程中每个盘子的当前位置
    # L[i] 值为 0、1、2 分别表示第 i 个盘子在 A、B、C 柱子上
    # 初始都在 A 柱子上，即 chr(65+0)
    L = [0] * n
    # n 个盘子一共需要移动 2^n-1 次才能完成
    for i in range(1, 2**n):
        # 假设盘子编号分别为 0,1,2,...,n-1
        # 第 i 步应该移动的盘子编号，正好是 i 的二进制形式中最后连续的 0 的个数
        b_i = bin(i)
        j = len(b_i[b_i.rfind('1')+1:])
        print(f'第{i}步：移动盘子{j+1},{chr(65+L[j])}->', end=' ')
        # 把 A、B、C 三根柱子摆成三角形，把第 j 个盘子移动到下一根柱子上
        # 根据 j 的奇偶性决定是顺时针移动还是逆时针移动
        L[j] = ((L[j]+1)%3 if j%2 == 0 else (L[j]+2)%3)
        # 下一根柱子，这里 65 是 A 的 ASCII 码
        print(chr(65+L[j]))

hannoi4(3)
```

例 5-8　求解八皇后问题。

八皇后问题是一个经典的回溯算法问题，由国际象棋棋手马克斯·贝瑟尔（Max Bezzel）于 1848 年提出，德国数学家约翰·卡尔·弗里德里希·高斯（德文名字为 Johann Carl Friedrich Gauß）对该问题做了大量研究。该问题核心要求为：在国际象棋棋盘（8 行 8 列）上摆放 8 个皇后，其中任意两个都不能位于同一行、同一列或同一斜线上。

例 5-8

```
def is_conflict(perm, last):
    # 如果行下标 last 的皇后与前面的皇后有冲突就返回 True，否则返回 False
    for i in range(last):
        if perm[i]==perm[last] or last-i==abs(perm[last]-perm[i]):
            return True
    return False

def queen8_1(n):
    # perm 中 -1 表示没有放置皇后，非负整数表示皇后位置的列下标
    result, perm, current = [], [-1]*n, 0
    while current >= 0:
        perm[current] = perm[current] + 1
        while perm[current]<n and is_conflict(perm,current):
            # 寻找一个不冲突的位置
            perm[current] = perm[current] + 1
```


Psyzw3-7.indd　109　　2025/5/8　17:01:08

```
            if perm[current] == n:
                # 所有位置都冲突，该行无法放置皇后，回退
                perm[current] = -1
                current = current - 1
            elif current < n-1:
                # 放置成功，继续尝试下一个皇后
                current = current + 1
            else:
                # 找到一个有效解，记录
                result.append(tuple(perm))
        return sorted(result)

print(queen8_1(8))
```

下面代码演示了另一种实现方式。

```
def is_valid(perm, col):
    # 当前皇后所在的行号，这里已经隐式保证每个皇后的行号不同
    row = len(perm)
    # 检查 (row,col) 这个位置是否可以放皇后，与 perm 中已经放好的皇后们是否有冲突
    for r, c in enumerate(perm):
        # 如果这一列已有皇后，或者某个皇后与当前皇后的水平与垂直距离相等
        # 就表示当前皇后位置不合法，不允许放置
        if c == col or abs(row-r) == abs(col-c):
            return False
    return True

def queen8_2(n, perm=()):
    # 参数 perm 为已经有皇后的列号，每个元素的下标对应行号
    # (3,5,6) 表示第 0 行皇后列下标为 3，第 1 行皇后列下标为 5，第 2 行皇后列下标为 6
    # 已是最后一个皇后，保存本次结果
    if len(perm) == n:
        return [perm]
    res = []
    for col in range(n):
        # 检查 perm 指定的位置已经有皇后的情况下，下一行的 col 列位置能不能放皇后
        if not is_valid(perm, col):
            continue
        for r in queen8_2(n, perm+(col,)):
            res.append(r)
    return res

# 形式转换，最终结果中包含每个皇后所在的行号和列号
```


Psyzw3-7.indd 110 2025/5/8 17:01:08