第 3 章 Python 数学与算法

CHAPTER 3

数学是人类对事物的抽象结构与模式进行严格描述的一种通用手段，可以应用于现实世界的任何问题，所有的数学对象本质上都是人为定义的。从这个意义上，数学属于形式科学，而不是自然科学。不同的数学家和哲学家对数学的确切范围和定义有一系列的看法。在人类历史发展和社会生活中，数学发挥着不可替代的作用，同时也是学习和研究现代科学技术必不可少的基本工具。本章将介绍与数学相关的几种算法。

3.1 枚举算法

枚举算法也叫穷举算法，其最大特点是在面对任何情况时会尝试每一种解决方法。在进行归纳推理时，如果逐个考查了某类事件的所有可能情况后得出一般结论，那么这个结论是可靠的，这种归纳方法叫作枚举法。枚举算法的思想是：将问题的所有可能的答案一一列举，然后根据条件判断答案是否合适，最后保留合适的，丢弃不合适的。Python 中一般用 while 循环或 if 语句实现。

使用枚举算法的基本思路：

(1) 确定枚举对象、枚举范围和判定条件。

(2) 逐一列举可能的解，验证每个解是否是问题的解。

一般情况下，按照下面三个步骤进行：

(1) 解的可能范围，不能遗漏任何一个真正解，也要避免有重复解。

(2) 判断是否是真正解的方法。

(3) 使可能解的范围降至最小，以便提高解决问题的效率。

【例 3-1】 枚举算法计算 24 点游戏。

解析： 24 点是一款经典的棋牌类益智游戏，要求 4 个数字的运算结果等于 24。即用加、减、乘、除把牌面上的数算成 24。每张牌必须只能用一次，例如：抽出的牌为 3、8、8、9，那么(9－8)×8×3＝24。

```
#枚举算法:24 点游戏
import itertools

#Python 计算 24 点游戏
def twentyfour(cards):
```

```
    """
        (1)itertools.permutations(可迭代对象):
        通俗地讲,就是返回可迭代对象的所有数学全排列方式.
        itertools.permutations("1118") -> 即将数字 1118 进行全排列组合
        (2)itertools.product( * iterables, repeat = 1)
        iterables 是可迭代对象,repeat 指定 iterable 重复几次
        返回一个或者多个 iterables 中的元素的笛卡儿积的元组
        product(list1, list2) ->即依次取出 list1 中的每 1 个元素,与 list2 中的每 1 个元素,组
成元组
    """
    for num in itertools.permutations(cards):
        for ops in itertools.product(" +- * /", repeat = 3):
            #({0}{4}{1}){5}({2}{6}{3}) ->即在{0}{1}{2}{3}放上数字,{4}{5}{6}放上运算符
            # 号,只能放三个,四个数字中间只能放三个运算符
            # 带括号有 8 种方法
            #1. (ab)cd
            bsd1 = '({0}{4}{1}){5}{2}{6}{3}'.format( * num, * ops)
            #2. a(bc)d
            bsd2 = '{0}{4}({1}{5}{2}){6}{3}'.format( * num, * ops)
            #3. ab(cd)
            bsd3 = '{0}{4}{1}{5}({2}{6}{3})'.format( * num, * ops)
            #4. (ab)(cd)
            bsd4 = '({0}{4}{1}){5}({2}{6}{3})'.format( * num, * ops)
            #5. ((ab)c)d
            bsd5 = '(({0}{4}{1}){5}{2}){6}{3}'.format( * num, * ops)
            #6.  (a(bc))d
            bsd6 = '({0}{4}({1}{5}{2})){6}{3}'.format( * num, * ops)
            #7.  a((bc)d)
            bsd7 = '{0}{4}(({1}{5}{2}){6}{3})'.format( * num, * ops)
            #8.  a(b(cd))
            bsd8 = '{0}{4}({1}{5}({2}{6}{3}))'.format( * num, * ops)
            # print([bsd1, bsd2, bsd3, bsd4, bsd5, bsd6, bsd7, bsd8])
            for bds in [bsd1, bsd2, bsd3, bsd4, bsd5, bsd6, bsd7, bsd8]:
                try:
                    if abs(eval(bds) - 24.0) < 1e-20:
                        return "24 点结果 = " + bds
                except ZeroDivisionError:              #零除错误
                    continue
    return "Not fond"

cards = ['2484', '1126', '1127', '1128', '2484', '1111']
for card in cards:
    print(twentyfour(card))
```

运行程序,结果解析:cards 中的牌数,最终通过加减乘除得出 24,而 1111 这 4 个数不能通过加、减、乘、除得到 24。返回值如下:

```
24 点结果 = (2 + 4) * (8 - 4)
24 点结果 = ((1 + 1) + 2) * 6
24 点结果 = (1 + 2) * (1 + 7)
24 点结果 = (1 + (1 * 2)) * 8
24 点结果 = (2 + 4) * (8 - 4)
Not fond
```

3.2　递推算法

在解决许多数学问题中，根据已知条件，利用计算公式进行若干步重复的运算即可求解答案，这种方法被称为递推算法。根据推导问题的方向，可将递推算法分为顺推法和逆推法。

1. 递推与递归的比较

相对于递归算法，递推算法免除了数据进出栈的过程，也就是说，不需要函数不断地向边界值靠拢，而直接从边界出发，直到求出函数值。例如，阶乘函数：$f(n)=n\times f(n-1)$。

在 $f(3)$的运算过程中，递归的数据流动过程如下：

```
f(3){f(i) = f(i - 1) * i} --> f(2) --> f(1) --> f(0){f(0) = 1} --> f(1) --> f(2) -- f(3){f(3) = 6}
```

而递推如下：

```
f(0) --> f(1) --> f(2) --> f(3)
```

由此可见，递推的效率要高一些，在可能的情况下应尽量使用递推。但是递归作为比较基础的算法，它的作用不能忽视。所以，在把握这两种算法的时候应该特别注意。

【例 3-2】 钓鱼比赛：6 位同学钓鱼比赛，他们钓到的鱼数量都不相同。问第 1 位同学钓了多少条时，他指着旁边的第 2 位同学说比他多钓了 2 条，追问第 2 位同学，他说比第 3 位同学多钓了 2 条；如此，都说比另一位同学多钓了 2 条；最后问到第 6 位同学时，他说自己钓了 3 条。问第一位同学共钓鱼多少条？

```
'''递推算法'''
k = 3
for i in range(1,6):
    k += 2
print(k)

'''递归如下'''
def fish(n):
    if n == 6:
        return 3
    else:
        return fish(n + 1) + 2
print(fish(1))
```

2. 顺推法

所谓顺推法是从已知条件出发，逐步推算出要解决的问题的方法叫顺推。斐波那契数列、汉诺塔问题是经典的顺推法。

1）斐波那契数列

斐波那契数列指的是这样一个数列：

1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,17711,28657,46368,…

这个数列从第 3 项开始，每一项都等于前两项之和。设它的函数为 $f(n)$，已知 $f(1)=$

1，$f(2)=1$，$f(n)=f(n-2)+f(n-1)(n>=3, n\in \mathbf{N})$，则通过顺推可以知道，$f(3)=f(1)+f(2)=2$，$f(4)=f(2)+f(3)=3$……直至得到要求的解。

【例 3-3】 利用顺推法求解斐波那契数列。

```
class Solution:
    #递归
    def Fibonacci(self, n):
        if n == 1:
            return 1
        return self.Fibonacci(n-1) + self.Fibonacci(n-2)

    #非递归
    def Fibonacci2(self, n):
        a, b = 0, 1
        for i in range(n):
            a, b = b, a+b
        return b

f = Solution()
print(f.Fibonacci(10))
print(f.Fibonacci2(10))
```

2）汉诺塔问题

汉诺塔（又称河内塔）问题源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。

【例 3-4】 利用顺推法求解汉诺塔问题。

```
def move(n,a,b,c):
    if n == 1:
        print(a,'-->',c)

    else:
        move(n-1,a,c,b)
        move(1,a,b,c)
        move(n-1,b,a,c)

move(3,'a','b','c')
```

运行程序，输出如下：

```
a --> c
a --> b
c --> b
a --> c
b --> a
b --> c
a --> c
```

3. 逆推法

逆推法从已知问题的结果出发，用迭代表达式逐步推算出问题的开始条件，即顺推法的逆过程。

3.3　模拟算法

所谓模拟法，就是编写程序模拟现实世界中事物的变化过程，从而完成相应任务的方法。模拟法对算法设计的要求不高，需要按照问题描述的过程编写程序，程序按照问题要求的流程运行，从而求得问题的解。

1. 转盘赌选择概述

转盘赌选择策略是先将个体的相对适应值 $\frac{f_i}{\sum f_i}$ 记为 p_i，然后根据选择概率$\{p_i, i=1,2,\cdots,N\}$，按图 3-1 所示把圆盘分成 N 份，其中第 i 扇形的中心角为 $2\pi p_i$。在进行选择时，可以假想转动如图 3-1 所示的圆盘，如果某参照点落入第 i 个扇形内，则选择个体 i。这种选择策略可以实现如下：先生成一个[0,1]内的随机数，如果 $p_1+p_2+\cdots+p_{i-1}<r\leqslant p_1+p_2+\cdots+p_i$，则选择个体 i。显然，小扇区的面积越大，参照点落入其中的概率也越大，即个体的适应值越大，它被选择到的机会也就越多，其基因被遗传到下一代的可能性也越大。

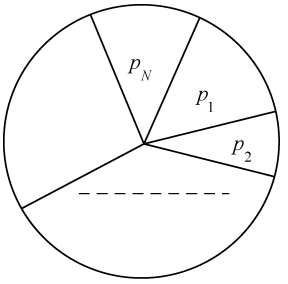


图 3-1　转盘器选择

2. 实现转盘赌游戏

假设各奖项在轮盘上所占比例为

- '一等奖'：0～0.08
- '二等奖'：0.08～0.3
- '三等奖'：0.3～1.0

递进的两个题目如下：

(1) 转动轮盘(随机产生一个 0～1 的数)1 万次，输出每个奖项的分布。

提示：使用字典来完成，首先定义一个字典 salary 来表示几等奖和它的中奖概率，再构造一个字典 now 来表示中几等奖的情况(键为几等奖，值为次数)。

```
from random import random
#各类奖项在轮盘上所占比例
salary = {'一等奖': (0, 0.08),
          '二等奖': (0.08, 0.3),
          '三等奖': (0.3, 1.0)}
print(salary)
#创建一个新字典
new = {}
n = 10000
for i in range(10000):
    num = random()
    if 0 <= num < 0.08:
        new['一等奖'] = new.get('一等奖', 0) + 1
    elif 0.08 <= num < 0.3:
        new['二等奖'] = new.get('二等奖', 0) + 1
    else:
```

```
            new['三等奖'] = new.get('三等奖', 0) + 1
    #根据值进行排序
    new = sorted(new.items(), key = lambda x: x[1], reverse = True)
    print("中奖情况分布:")
    for i in new:
        print(i)
```

运行程序,输出如下:

```
{'一等奖': (0, 0.08), '二等奖': (0.08, 0.3), '三等奖': (0.3, 1.0)}
中奖情况分布:
('三等奖', 6952)
('二等奖', 2248)
('一等奖', 800)
```

(2) 模拟一个用户转动轮盘的过程。

① 输出用户转动 10 次轮盘的奖项分布情况。

② 已知积分:转到一等奖,得 5 分,转到二等奖,得 3 分;转到三等奖,得 1 分。输出用户转动 10 次的得分情况;

③ 根据积分领奖品(自己构造奖项就可以)。

- 如果大于或等于 30 分,输出,奖励奥运会吉祥物。
- 如果大于或等于 20 分,小于 30 分,输出,奖励饮水壶。
- 如果大于或等于 10 分,小于 20 分,输出,奖励水杯。
- 如果大于或等于 0,小于 10 分,输出,奖励小奖品。

```
from random import random

def lunpan(n):
    """输出奖项分布"""
    #各类奖项在轮盘上所占比例
    salary = {'一等奖': (0, 0.08),
              '二等奖': (0.08, 0.3),
              '三等奖': (0.3, 1.0)}
    #创建一个新字典
    new = {}
    for i in range(n):
        num = random()
        if 0 <= num < 0.08:
            new['一等奖'] = new.get('一等奖', 0) + 1
        elif 0.08 <= num < 0.3:
            new['二等奖'] = new.get('二等奖', 0) + 1
        else:
            new['三等奖'] = new.get('三等奖', 0) + 1
    #根据值进行排序
    new = sorted(new.items(), key = lambda x: x[1], reverse = True)
    return new

def score(new):
    """对积分进行统计"""
    count = 0
    for line in new:
        if line[0] == "一等奖":
```

```
            count += line[1] * 5
        elif line[0] == "二等奖":
            count += line[1] * 3
        else:
            count += line[1]
    return count

def prize(score):
    if score >= 30:
        print("奖励奥运会吉祥物!")
    elif 20 <= score < 30:
        print("奖励饮水壶!")
    elif 10 <= score < 20:
        print("奖励水杯!")
    elif 0 <= score < 10:
        print("奖励小奖品!")

if __name__ == "__main__":
    print("欢迎参加轮盘赌游戏:")
    n = 10
    #调用 lunpan()函数,返回分布情况
    new = lunpan(n)
    #输出奖项分布
    print("中奖情况分布:")
    for i in new:
        print(i)
    #调用 score()函数对积分进行统计
    score = score(new)
    print("积分:{}分".format(score))
    #调用 prize()函数
    prize(score)
```

运行程序,输出如下:

```
欢迎参加轮盘赌游戏:
中奖情况分布:
('三等奖', 9)
('二等奖', 1)
积分:12 分
奖励水杯!
```

3.4　逻辑推理

解决逻辑推理问题的关键是,根据题目中给出的各种已知条件,提炼出正确的逻辑关系,并将其转换为用 Python 语言描述的逻辑表达式。Python 语言提供基本的关系运算符和逻辑运算符,可以用来构建各种逻辑表达式。在解决逻辑推理问题时,一般使用枚举法,也就是使用循环结构将各种方案列举出来,再逐一判断根据题目建立的逻辑表达式是否成立,最终找到符合题意的答案。

1. 各类逻辑推理

逻辑推理的种类按推理过程的思维方向划分:一类是从特殊到一般的推理,推理形式

主要有归纳、类比；另一类是从一般到特殊的推理，推理形式主要有演绎。

(1) 归纳推理是由特殊的前提推出普遍性结论的推理，主要有完全归纳法、不完全归纳法、简单枚举法、科学归纳法等类型。

(2) 类比推理是从特殊性前提推出特殊性结论的一种推理，也就是说，从一个对象的属性推出另一对象也可能具有这个属性，这种思维形式在创造学中称为“相似思维”。

(3) 演绎推理是由普遍性的前提推出特殊性结论的推理，有三段论、假言推理和选言推理等形式。

2. 逻辑推理与因果关系的区别

逻辑推理与因果关系的区别主要有以下几点：

(1) 两者最根本的区别是逻辑推理不考虑时间因素，而因果关系却必须考虑时间因素。

(2) 逻辑推理的条件是有限的，而在任何一个因果关系中，“条件”实际上是无限的。在逻辑推理中，有时一个条件即可推出一个结论，有时多个条件才能推出一个结论。但即使多个条件推出一个结论，这些条件的个数也都是有限的。但现实中的因果关系却大不相同，与结果现象有关的条件实际上是无限(多)的，无法把它们穷举出来。

(3) 逻辑推理中(主要指演绎推理)，条件必然蕴涵结论。但在因果关系中，原因并不必然蕴涵结论，而只有在“条件”都已经具备的情况下，原因的出现才会引起结果的发生。

(4) 因果关系是“现实”关系，只有在原因现象和结果现象已经发生之后，我们才说原因A和结果B之间存在“因果关系”。而“逻辑推理”是一种“理论”推导，它不需要任何现实性做支撑，条件就必然蕴涵结论。

3. Python 解决逻辑推理问题

逻辑推理解决离散数学问题是非常有用的方法，下面通过两个例子演示使用 Python 实现命题逻辑等值演算应用。

【例 3-5】 陈教授是哪里人。

解析：在某次研讨会的中间休息时间，3名参会者根据王教授的口音对他是哪个省(区、市)的人判断如下：

A：陈教授不是四川人，是广东人。

B：陈教授不是广东人，是四川人。

C：陈教授既不是广东人，也不是重庆人。

听完这3人的判断后，陈教授笑着说，你们3人中有一人说得全对，有一人说对了一半，另一人说得全不对。下面分析陈教授到底是哪里人。

```
'''p: 是广东人; q: 是四川人; r: 是重庆人'''
ls = [0, 1]
for p in ls:
    for q in ls:
        for r in ls:
            A1 = not p and q
            A2 = (not p and not q)or(p and q)
            A3 = p and not q   #A的三种情况
            B1 = p and not q
            B2 = (p and q)or(not p and not q)
            B3 = not p and q   #B的三种情况
```

```
C1 = not q and not r
C2 = (not q and r)or(q and not r)
C3 = q and r   #C的三种情况
if ((A1 and B2 and C3) or (A1 and B3 and C2) or (A2 and B1 and C3) or\
   (A2 and B3 and C1) or (A3 and B1 and C2) or (A3 and B2 and C1)) == 1\
    and p + q + r == 1:   #成立一项
    if p == 1:
        print('陈教授是广东人')
    if q == 1:
        print('陈教授是四川人')
    if r == 1:
        print('陈教授是重庆人')
```

运行程序,输出如下:

```
陈教授是四川人
```

【例 3-6】 谁是班委。

在某班班委成员的选举中,已知刘小红、张三强、丁小仙 3 名同学被选进了班委会。该班的 3 名同学猜测如下:

A:刘小红为班长,张三强为生活委员。

B:丁小仙为班长,刘小红为生活委员。

C:张三强为班长,刘小红为学习委员。

班委会分工名单公布后发现,3 人都恰好猜对了一半,问:刘小红、张三强、丁小仙各任何职。

```
'''班长:b 生活委员:s 刘小红:w 张三强:t 丁金生:d'''
L = ['b', 's', 'x']
for w in L:
   for t in L:
      for d in L:
         W1 = (w == 'b')          #刘小红为班长
         L1 = (t == 's')          #张三强为生活委员
         D1 = (d == 'b')          #丁小仙为班长
         W2 = (w == 's')          #刘小红为生活委员
         L2 = (t == 'b')          #张三强为班长
         W3 = (w == 'x')          #刘小红为学习委员
         A1 = W1 and not L1
         A2 = not W1 and L1       #A
         B1 = D1 and not W2
         B2 = not D1 and W2       #B
         C1 = L2 and not W3
         C2 = not L2 and W3       #C
         if ((A1 and B1 and C1) or (A1 and B1 and C2) or (A1 and B2 and C1) or (A1 and B2 and C2) or \
            (A2 and B1 and C1) or (A2 and B1 and C2) or (A2 and B2 and C1) or (A1 and B2 and C2)) == 1 \
                and W1 + D1 + L2 == 1 and L1 + L2 == 1:   #排除互斥项
            if W1 == 1:
               print('刘小红为班长')
            if L1 == 1:
               print('张三强为生活委员')
            if D1 == 1:
               print('丁小仙为班长')
            if W2 == 1:
```

```
        print('刘小红为生活委员')
    if L2 == 1:
        print('张三强为班长')
    if W3 == 1:
        print('刘小红为学习委员')
```

运行程序，输出如下：

```
张三强为生活委员
丁小仙为班长
刘小红为学习委员
```

3.5 冒泡排序

冒泡排序是一种简单的排序算法，它也是一种稳定排序算法。冒泡排序算法的基本思想：从序列中未排序区域的最后一个元素开始，依次比较相邻的两个元素，并将小的元素与大的交换位置。这样经过一轮排序，最后的元素被移出未排序区域，成为已排序区域的第一个元素。同样，也可以按从大到小的顺序排列。

假设待排序序列为[5,1,4,2,8]，如果采用冒泡排序对其进行升序(由小到大)排序，则整个排序过程如下：

(1) 第一轮排序，此时整个序列中的元素都位于待排序序列，依次扫描每对相邻的元素，并对顺序不正确的元素对交换位置，整个过程如图 3-2 所示。

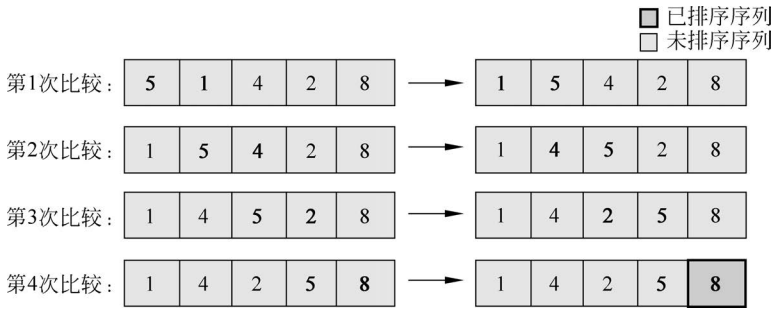


图 3-2　第一轮排序

从图 3-2 可以看到，经过第一轮冒泡排序，从待排序序列中找出了最大数 8，并将其放到了待排序序列的尾部，并入已排序序列中。

(2) 第二轮排序，此时待排序序列只包含前 4 个元素，依次扫描每对相邻元素，对顺序不正确的元素对交换位置，整个过程如图 3-3 所示。

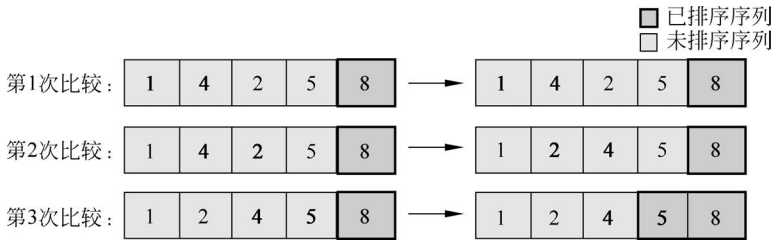


图 3-3　第二轮排序

从图 3-3 可以看到，经过第二轮冒泡排序，从待排序序列中找出了最大数 5，并将其放到了待排序序列的尾部，并入已排序序列中。

（3）第三轮排序，此时待排序序列包含前 3 个元素，依次扫描每对相邻元素，对顺序不正确的元素对交换位置，整个过程如图 3-4 所示。

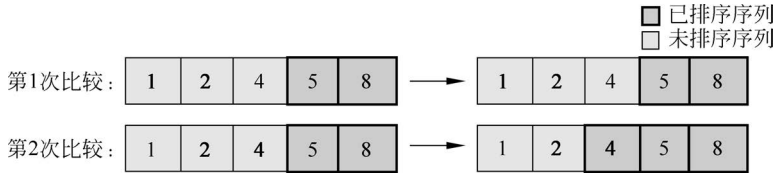


图 3-4　第三轮排序

经过第三轮冒泡排序，从待排序序列中找出了最大数 4，并将其放到了待排序序列的尾部，并入已排序序列中。

（4）第四轮排序，此时待排序序列包含前 2 个元素，对其进行冒泡排序的整个过程如图 3-5 所示。

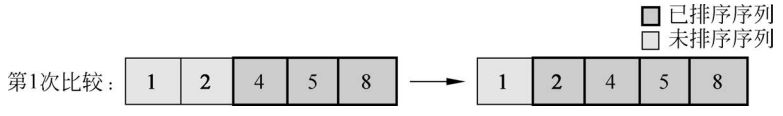


图 3-5　第四轮排序

（5）当进行第五轮冒泡排序时，由于待排序序列中仅剩 1 个元素，无法再进行相邻元素的比较，因此直接将其并入已排序序列中，此时的序列就认定为已排序好的序列，如图 3-6 所示。

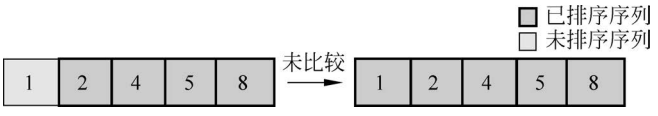


图 3-6　第五轮序列

【例 3-7】 利用冒泡排序对数列[39,22,41,19,32,15]进行排序。

排序步骤如下：

（1）单次循环查找最大值。

依次两两比较列表中元素，将最大值移到列表末尾，并打印最大值和列表的内容。

```
List = [39,22,41,19,32,15]                          #原来列表中的内容
for i in range(len(List) - 1):                      #循环
    if List[i] > List[i + 1]:                       #比较当前值和下一个元素
        #若前面元素较大,则交换位置,将较大的元素后移
        List[i], List[i + 1] = List[i + 1], List[i]
print('最大值为：',List[ - 1])                       #打印最大值
print('依次排序后的列表为：',List)                    #打印列表
最大值为：41
依次排序后的列表为：[22, 39, 19, 32, 15, 41]
```

（2）使用两层循环实现冒泡排序。

```
List = [39,22,41,19,32,15]                          #原来列表中的内容
for j in range(len(List) - 1):                      #外层 for 循环控制循环次数
```

```
    for i in range(len(List) - 1 - j):                    # 内层 for 循环控制比较次数
        if List[i] > List[i + 1]:                         # 比较当前值和下一个元素
            List[i], List[i + 1] = List[i + 1], List[i]   # 将较大的元素进行后移
print('排序后的列表为: ',List)                             # 打印排序后的列表
```

运行程序,输出如下:

```
排序后的列表为: [15, 19, 22, 32, 39, 41]
```

(3) 使用函数封装冒泡排序实现的过程,并传参控制正序或倒序排列。

```
List = [39,22,41,19,32,15]                                # 原来列表中的内容
def bubble_sort(List,flag = True):
    for j in range(len(List) - 1):                        # for 循环控制循环次数
        for i in range(len(List) - 1 - j):                # for 循环控制比较次数
            if List[i] > List[i + 1]:                     # 比较当前值和下一个元素
                # 较大的元素进行后移
                List[i], List[i + 1] = List[i + 1], List[i]
    if flag:                                              # 如果要求正序排列
        return List                                       # 直接返回排序后的结果
    else:
        return List[::-1]                                 # 返回反序后的列表
print('正序排列后的列表为: ',bubble_sort(List, True))
print('反序排列后的列表为: ',bubble_sort(List, False))
```

运行程序,输出如下:

```
正序排列后的列表为: [15, 19, 22, 32, 39, 41]
反序排列后的列表为: [41, 39, 32, 22, 19, 15]
```

3.6 选择排序

选择排序是一种简单直观的排序算法。它的工作原理是每次从待排序的数据元素中选出最小(或最大)的一个元素,存放在序列的起始位置,所以称为选择排序。

选择排序算法的基本思想:先从序列的未排序区域中选择一个最小的元素,把它与序列中的第 1 个元素交换位置;再从剩下的未排序区域中选出一个最小的元素,把它与序列中的第 2 个元素交换位置……如此反复操作,直到序列中的所有元素按升序排列完毕。

例如,对无序表[56,12,80,92,20]采用选择排序算法进行排序,具体过程如下:

(1) 第一次遍历时,从下标为 1 的位置即 56 开始,找出关键字值最小的记录 12,同下标为 0 的关键字 56 交换位置。

12	56	80	92	20

(2) 第二次遍历时,从下标为 2 的位置即 56 开始,找出最小值 20,同下标为 2 的关键字 56 互换位置。

12	20	80	92	56

(3) 第三次遍历时,从下标为 3 的位置即 80 开始,找出最小值 56,同下标为 3 的关键字 80 互换位置。

12	20	56	92	80

(4) 第四次遍历时,从下标为 4 的位置即 91 开始,找出最小值 80,同下标为 4 的关键字 92 互换位置。

12	20	56	80	92

(5) 至此简单选择排序算法完成,无序列变为有序表。

【例 3-8】 利用选择排序算法对序列[56,12,80,92,20]进行排序。

实现步骤如下:

步骤 1:找出序列中的最大值,然后跟最后一个元素交换位置。

步骤 2:循环执行步骤 1。

步骤 3:优化函数并封装成函数。

```
#步骤 1
lst = [56,12,80,92,20]
index = 0                          #定义一个存放最大值的索引值的变量,默认为 0
for i in range(len(lst) - 1):
    if lst[i + 1] > lst[index]:
        index = i + 1
#结束完循环之后 index 为最大值的索引值
lst[index], lst[len(lst) - 1] = lst[len(lst) - 1], lst[index]
print(lst)                         #此时将最大值放到了列表最后
[56, 12, 80, 20, 92]

#步骤 2
lst = [56,12,80,92,20]
for i in range(len(lst) - 1):
    index = 0                      #定义一个存放最大值的索引值的变量,默认为 0
    for j in range(len(lst) - 1 - i):
        #当循环第一次时,i = 0,需要循环整个列表才能找出最大值
        #当循环第二次时,i = 1,不需要循环最后一个元素,因为最后一个元素一定是最大的
        #...
        if lst[j + 1] > lst[index]:
            index = j + 1
        #结束完循环之后 index 为最大值的索引值,将最大值放置参加循环的最后一位
    lst[index], lst[len(lst) - 1 - i] = lst[len(lst) - 1 - i], lst[index]
print(lst)
[12, 20, 56, 80, 92]

#步骤 3
def sort(lst):
    n = len(lst)
    for i in range(n - 1):
        index = 0
        for j in range(n - 1 - i):
            if lst[j + 1] > lst[index]:
                index = j + 1
        lst[index], lst[n - 1 - i] = lst[n - 1 - i], lst[index]
    return lst
lst = [56,12,80,92,20]
print(sort(lst))
[12, 20, 56, 80, 92]
```

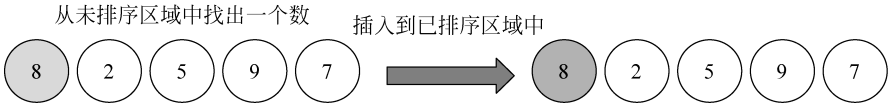
3.7 插入排序

插入排序是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

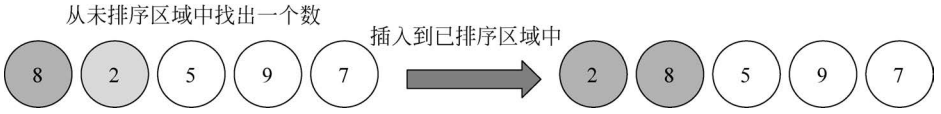
把 n 个待排序的元素看成为一个有序表和一个无序表。开始时有序表中只包含 1 个元素，无序表中包含有 $n-1$ 个元素，排序过程中每次从无序表中取出第一个元素，将它插入到有序表中的适当位置，使之成为新的有序表，重复 $n-1$ 次可完成排序过程。

例如，有一组初始化数组[8,2,5,9,7]，把数组中的数据分成两个区域，已排序区域和未排序区域，初始化的时候所有的数据都处在未排序区域中，已排序区域是空的。

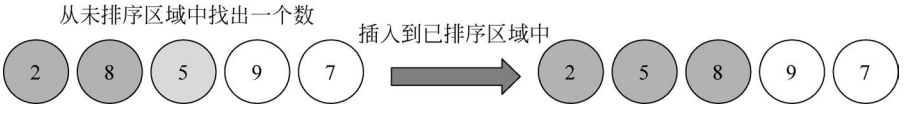
(1) 第一轮排序，从未排序区域中随机拿出一个数字，既然是随机，那么我们就获取第一个，然后插入到已排序区域中，如果已排序区域是空，那么就不做比较，默认自身已经是有序的了。



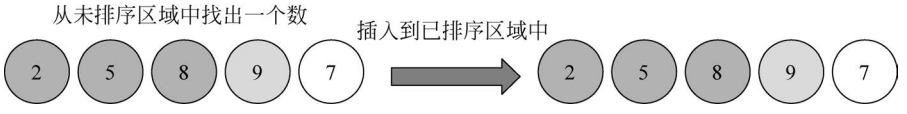
(2) 第二轮排序，继续从未排序区域中拿出一个数，插入到已排序区域中，这个时候要遍历已排序区域中的数字遍历做比较，比大比小取决于我们是想升序排还是想倒序排，这里按升序排。



(3) 第三轮排序，排 5。



(4) 第四轮排序，排 9。



(5) 第五轮排序，排 7。

(6) 排序结束。

【例 3-9】 利用插入排序对序列[11,11,22,33,33,36,39,44,55,66,69,77,88,99]进行排序。

```
def insertion_sort(arr):
    """插入排序"""
```

```
    #第一层 for 表示循环插入的遍数
    for i in range(1, len(arr)):
        #设置当前需要插入的元素
        current = arr[i]
        #与当前元素比较的比较元素
        pre_index = i - 1
        while pre_index >= 0 and arr[pre_index] > current:
            #当比较元素大于当前元素则把比较元素后移
            arr[pre_index + 1] = arr[pre_index]
            #往前选择下一个比较元素
            pre_index -= 1
        #当比较元素小于当前元素,则将当前元素插入在其后面
        arr[pre_index + 1] = current
    return arr
print('排序后的结果为:')
insertion_sort([11, 11, 22, 33, 33, 36, 39, 44, 55, 66, 69, 77, 88, 99])
```

运行程序,输出如下:

```
排序后的结果为:
[11, 11, 22, 33, 33, 36, 39, 44, 55, 66, 69, 77, 88, 99]
```

3.8　快速排序

快速排序采用分治的策略。它的基本思想是:通过一趟排序将要排序的数据分割成独立的两部分,其中一部分的所有数据都比另外一部分的所有数据都要小,然后再按此方法对这两部分数据分别进行快速排序,整个排序过程可以递归进行,以此达到整个数据变成有序序列的目的。

如图 3-7 所示,假设最开始的基准数据为数组第一个元素 23,则首先用一个临时变量去存储基准数据,即 tmp=23。然后分别从数组的两端扫描数组,设两个指示标志:low 指向起始位置,high 指向末尾。

(1) 先从后半部分开始,如果扫描到的值大于基准数据就让 high 减 1,如果发现有元素比该基准数据的值小(如图 3-7 中的 18≤tmp),就将 high 位置的值赋值给 low 位置,结果如图 3-8 所示。

(2) 开始从前往后扫描,如果扫描到的值小于基准数据就让 low 加 1,如果发现有元素大于基准数据的值(如图 3-8 中的 46≥tmp),就再将 low 位置的值赋值给 high 位置的值,指针移动并且数据交换后的结果如图 3-9 所示。

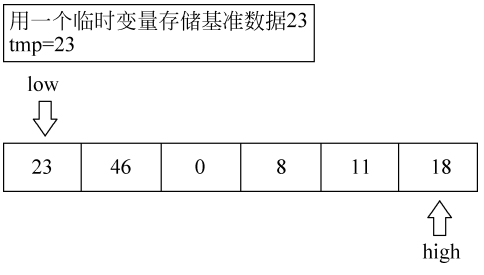


图 3-7　最开始的基准数据

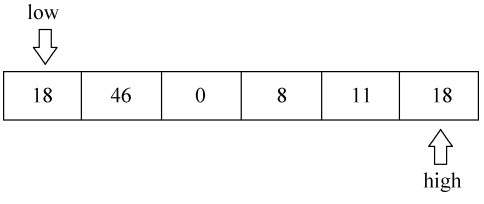


图 3-8　值大于基准数据 high 减 1

(3) 然后再开始从后向前扫描,原理同上,发现图 3-9 的 11≤tmp,则将 high 位置的值赋值给 low 位置的值,结果如图 3-10 所示。

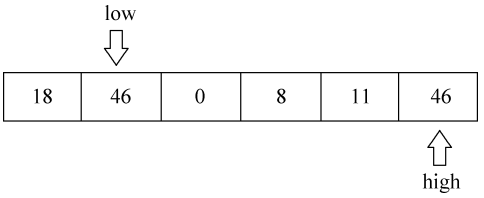


图 3-9 值小于基准数据 low 加 1

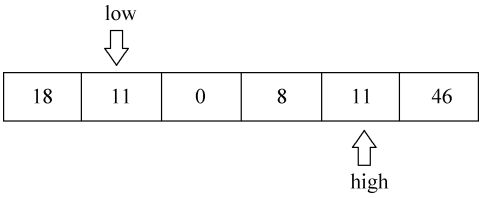


图 3-10 high 位置的赋值给 low 位置的值

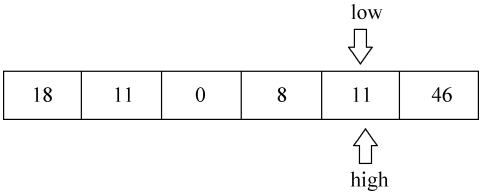


图 3-11 从前往后遍历

(4) 然后再开始从前往后遍历,直到 low=high 结束循环,此时 low 或 high 的下标就是基准数据 23 在该数组中的正确索引位置,如图 3-11 所示。

快速排序的本质就是把比基准数大的都放在基准数的右边,把比基准数小的放在基准数的左边,这样就找到了该数据在数组中的正确位置。

【例 3-10】 利用快速排序对序列[50,36,61,95,73,13,27,50]进行排序。

```
def QuickSort(myList,start,end):
    #判断 low 是否小于 high,如果为 false,则直接返回
    if start < end:
        i,j = start,end
        #设置基准数
        base = List1[i]
        while i < j:
      #如果列表后边的数,比基准数大或相等,则前移一位直到有比基准数小的数出现
            while (i < j) and (List1[j] >= base):
                j = j - 1
            #如找到,则把第 j 个元素赋值给第个元素 i,此时表中 i,j 个元素相等
            List1[i] = List1[j]
            #同样的方式比较前半区
            while (i < j) and (List1[i] <= base):
                i = i + 1
            List1[j] = List1[i]
        #做完第一轮比较之后,列表被分成了两个半区,并 i = j,需要将这个数设置回 base
        List1[i] = base
        #递归前后半区
        QuickSort(List1, start, i - 1)
        QuickSort(List1, j + 1, end)
    return List1

List1 = [50,36,61,95,73,13,27,50]
print("快速排序:")
QuickSort(List1,0,len(List1)-1)
print(List1)
```

运行程序,输出如下:

```
快速排序:
[13, 27, 36, 50, 50, 61, 73, 95]
```

3.9　二分查找

二分查找又称折半查找，优点是比较次数少，查找速度快，平均性能好，占用系统内存较少；其缺点是要求待查表为有序表，且插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。

1. 二分查找思想

二分查找算法的基本思想：假设序列中的元素是按从小到大的顺序排列的，以序列的中间位置为界将序列一分为二，再将序列中间位置的元素与目标数据比较。如果目标数据等于中间位置的元素，则查找成功，结束查找过程；如果目标数据大于中间位置的元素，则在序列的后半部分继续查找；如果目标数据小于中间位置的元素，则在序列中的前半部分继续查找。当序列不能被定位时，则查找失败，并结束查找过程。

中间位置的计算公式如下：

$$\text{中间位置} \approx (\text{结束位置} - \text{起始位置}) \div 2 + \text{起始位置}$$

注意：对计算结果进行向下取整。

2. 算法分析

根据二分查找算法的基本思想，结合图 3-12 将该算法的工作过程描述如下。

第一次查找：起始位置为 0，结束位置为 7，中间位置为(7－0)/2＋0≈3，序列中索引位置为 3 的元素是 7。目标值 17 大于 7，则继续查找元素 7 右侧的数据。

第二次查找：起始位置为 4，结束位置为 7，中间位置为(7－4)/2＋4≈5，序列中索引位置为 5 的元素是 13。目标值 17 大于 13，则继续查找元素 13 右侧的数据。

第三次查找：起始位置为 6，结束位置为 7，中间位置为(7－6)/2＋6≈6，序列中索引位置为 6 的元素是 17。正好与目标值 17 相等，则将目标位置 6 返回，整个查找过程结束。

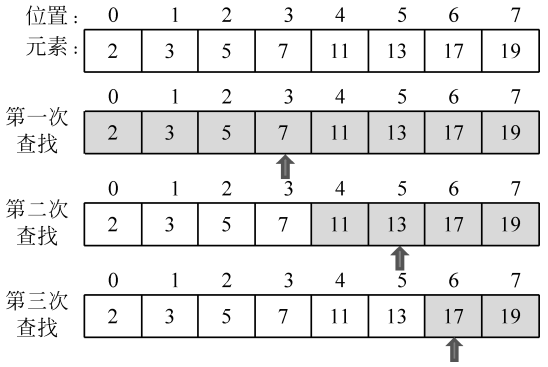


图 3-12　二分查找算法的工作过程

通过分析上述查找过程，将二分查找算法的编程思路描述如下：

(1) 根据待查找序列的起始位置和结束位置计算出一个中间位置。

(2) 如果目标数据等于中间位置的元素，则查找成功，返回中间位置。

(3) 如果目标数据小于中间位置的元素，则在序列的前半部分继续查找。

(4) 如果目标数据大于中间位置的元素，则在序列的后半部分继续查找。

(5) 重复进行以上步骤，直到待查找序列的起始位置大于结束位置，即待查找序列不可定位时，则查找失败。

【例 3-11】 利用二分查找法查找序列[2,3,5,7,11,13,17,19]中值为 17 的位置。

```
#返回 x 在 arr 中的索引,如果不存在则返回 - 1
```

```
def binarySearch(arr, l, r, x):
    #基本判断
    if r >= l:
        mid = int(l + (r - l)/2)
        #元素的中间位置
        if arr[mid] == x:
            return mid
        #元素小于中间位置的元素,只需要再比较左边的元素
        elif arr[mid] > x:
            return binarySearch(arr, l, mid - 1, x)
        #元素大于中间位置的元素,只需要再比较右边的元素
        else:
            return binarySearch(arr, mid + 1, r, x)
    else:
        #不存在
        return -1
#测试数组
arr = [2,3,5,7,11,13,17,19]
x = 10
#函数调用
result = binarySearch(arr, 0, len(arr) - 1, 17)
if result != -1:
    print("元素在数组中的索引为 %d" % result)
else:
    print("元素不在数组中")
```

运行程序,输出如下:

```
元素在数组中的索引为 6
```

3.10 勾股树

勾股树是根据勾股定理绘制的可以无限重复的图形,重复多次之后呈现为树状。勾股树最早是由古希腊数学家毕达哥拉斯绘制,因此又称之为毕达哥拉斯树。这种图形在数学上称为分形图,它们中的一个部分与其整体或者其他部分都十分相似,分形图内任何一个相对独立的部分,在一定程度上都是整体的再现和缩影。这就是分形图的自相似特性。

勾股定理的定义:在平面上的一个直角三角形中,两个直角边边长的平方加起来等于斜边长的平方。用数学语言表达为 $a^2+b^2=c^2$,用图形表达如图 3-13 所示。

以图 3-13 中的勾股定理图为基础,让两个较小的正方形按勾股定理继续“生长”,又能画出新一代的勾股定理图,如此一直画下去,最终得到一棵完全由勾股定理图组成的树状图形(图 3-14)。

1. 算法分析

利用分形图的自相似特性,先构造出分形图的基本图形,再不断地对基本图形进行复制,就能绘制出分形图。针对勾股树分形图,其绘制步骤如下:

(1) 先画出图 3-13 所示的勾股定理图形作为基本图形,将这一过程封装为一个绘图函数,以便进行递归调用。

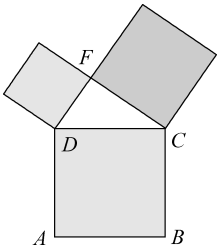


图 3-13　勾股定理图

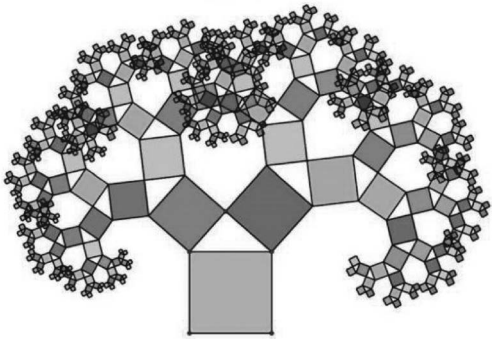


图 3-14　勾股树

（2）在绘制两个小正方形之前，分别以直角三角形两条直角边作为下一代勾股定理图形中直角三角形的斜边，以递归方式调用绘制函数画出下一代的基本图形。

（3）重复执行前两步，最终可绘制出一棵勾股树的分形图。由于是递归调用，需要递归的终止条件，此处设置某一代勾股定理图的直角三角形的斜边小于某个数值时就结束递归。

2. 经典应用

通过 Python 分别实现分形树与勾股树的绘制。

1）分形树

分形树是分形几何中的一小种类型，一棵分形树相当于一棵“满二叉树”。通常都用递归来实现，递归条件通常分两排：一排是用长度递减，直到长度不满足某个条件时退出；另一排则是按层数递归，相当于“满二叉树”的层序遍历。前一排的长度递归相当于“满二叉树”的先序遍历，从根出发先左子树后右子树，每一棵子树都按这种“先根后左右”的顺序遍历。

【例 3-12】 绘制分形树。

```
import turtle

def bintree(size):
    angle = 60                          #分叉的角度
    if size > 5:                        #长度退出条件
        turtle.forward(size)
        turtle.left(angle)
        bintree(size / 1.6)
        turtle.right(angle * 2)
        bintree(size / 1.6)
        turtle.left(angle)
        turtle.backward(size)

def main():
    turtle.speed(0)
    turtle.hideturtle()
    turtle.penup()
    turtle.left(90)
    turtle.backward(100)
    turtle.showturtle()
    turtle.pendown()
```

```
    turtle.pensize(2)
    turtle.color('green')
    bintree(150)
    turtle.done()

if __name__ == '__main__':
    main()
```

运行程序，效果如图 3-15 所示。

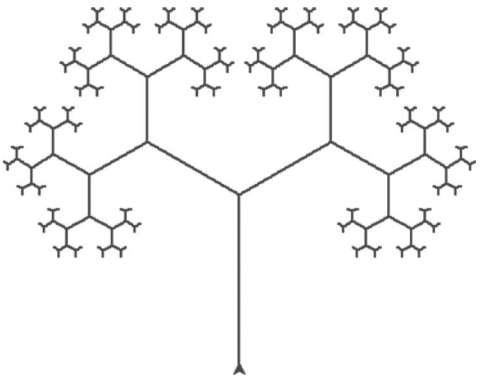


图 3-15　分形树

在以上代码中，长度以等比数列递减，公比为 1/1.6；也可以改成等差数列形式。此方式缺点是：树的层数不能直接控制，需要用初始长度、递减公式和退出条件计算得出。

2）勾股树

勾股树，其实就是分形树的一种，只是不像上例一样简单地画 2 个分叉，而是画直角三角形加上各边上的正方形，就像平面几何中勾股定理证明时画的示意图。

【例 3-13】 绘制勾股树。

```
from turtle import *

def Square(self,length):
    for _ in range(5):
        self.forward(length)
        self.right(90)

def Triangle(self,length):
    self.left(45)
    self.forward(length/2 ** 0.5)
    self.right(90)
    self.forward(length/2 ** 0.5)
    self.right(135)
    self.forward(length)

def Move2Right(self,length):
    self.back(length)
    self.right(45)
    self.forward(length/2 ** 0.5)
    self.right(90)

def Recursive(n, tracer, length):
    if n < 1: return
    tracers = []
    for left in tracer:
        if n < 3: left.pencolor('green')
        else: left.pencolor('brown')
        Square(left, length)
        Triangle(left, length)
        right = left.clone()
        left.right(45)
        Move2Right(right, length)
```

```
            tracers.append(left)
            tracers.append(right)
        Recursive(n-1, tracers, length/2**0.5)

def Setup(self, length, speed):
    self.hideturtle()
    self.speed(speed)
    self.penup()
    self.goto(-length*0.5, -length*1.8)
    self.seth(90)
    self.pensize(2)
    self.pendown()

def main(level, length, speed=-1):
#level: 树的层数
#length: 最底层正方形的边长
#speed: 1～10,画笔速度递增; =0 时速度最快; =-1 时关闭画笔踪迹
setup(800,600)
    title('Fractal Tree')
    if speed==-1: tracer(0)
    else: tracer(1)
    t = Turtle()
    Setup(t, length, speed)
    from time import sleep
    sleep(2)
    Recursive(level, list([t]), length)
    done()
    bye()

if __name__ == '__main__':
    main(6,150,10)
```

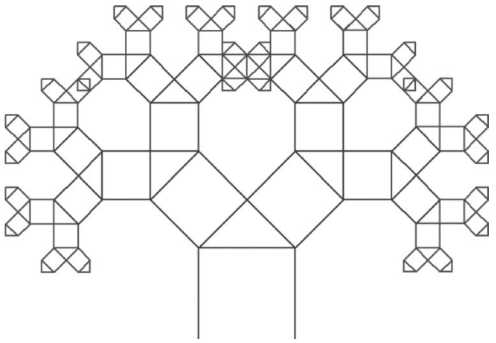


图 3-16　勾股树

运行程序,效果如图 3-16 所示。

3.11　玫瑰曲线

在数学中有一些美丽的曲线图形,如螺旋线、摆线、双纽线、蔓叶线、心脏线、渐开线、玫瑰曲线、蝴蝶曲线……这些形状各异、简繁有别的数学曲线图形为看似枯燥的数学公式披上了精彩纷呈的美丽衣裳。

在数学曲线的百花园中,玫瑰曲线算得上个中翘楚,它的数学方程简单,曲线变化众多,根据参数的变化能展现出姿态万千的优美形状。玫瑰曲线可用极坐标方程表示为

$$\rho = a\sin n\theta$$

也可以用参数方程表示为

$$\begin{cases} x = a\sin n\theta\cos\theta \\ y = a\sin n\theta\sin\theta \end{cases}$$

式中,参数 a 控制叶子的长度;参数 n 控制叶子的数量,并影响曲线闭合周期。当 n 为奇数

时，玫瑰曲线的叶子数为 n，闭合周期为 π，即参数 θ 的取值范围为 0～π，才能使玫瑰曲线闭合为完整图形。当 n 为偶数时，玫瑰曲线的叶子数为 $2n$，闭合周期为 2π，即参数 θ 的取值范围为 0～2π。

1. 算法分析

在数学世界中，像玫瑰曲线这样美丽的曲线图形实际上是由简单的函数关系生成的。利用曲线函数的参数方程，可以在平面直角坐标系中方便地绘制出它们的图形。

假如要利用玫瑰曲线的参数方程绘制三叶玫瑰曲线，则参数 n 的值可以设定为 3，参数 a 的值可以设定叶子的长度(如 100)。因为参数 $n=3$ 是奇数，所以三叶玫瑰曲线的闭合周期为 π。

绘制玫瑰曲线的编程思路：在一个循环结构中，让参数 θ 从 0 变化到 π，再利用玫瑰曲线的参数方程求出点坐标 x 和 y 的值，并通过绘图库绘制一系列连续的点，最终绘制出一个完整的玫瑰曲线图形。

2. 经典应用

根据上述算法分析中给出的编程思路，编程绘制玫瑰曲线的图形。下面通过两个实例演示利用 Python 绘制玫瑰曲线。

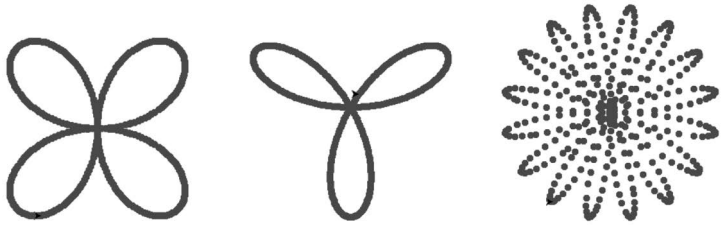
【例 3-14】 使用 turtle 模块(海龟绘图)绘制图案。

```
from turtle import *
from math import *

def rose(a,n):
    t = 0
    while t <= cycle:
        x = cos(t) * a * (sin(n * t))
        y = sin(t) * a * (sin(n * t))
        goto(x,y)
        dot(10)
        pd()
        t += 1
        pu()

speed(0)
tracer(100)
pencolor("blue")
pensize(5)
pu()
cycle = 360
a = 150
n = 2
rose(a,n)
hideturtle()
done()
```

运行程序，当 $n=2$ 时，得到效果如图 3-17(a)所示；当 $n=3$ 时，得到效果如图 3-17(b)所示；当 $n=8$ 时，得到效果如图 3-17(c)所示。



(a) n=2曲线　(b) n=3曲线　(c) n=8曲线

图 3-17　玫瑰曲线

可修改代码中的 n 值，绘制不同代玫瑰曲线。

【例 3-15】 利用海龟绘制盛开的玫瑰花。

```
import turtle as t

t.setup(800,800)
t.hideturtle()
t.speed(11)
t.penup()
t.goto(50, - 450)
t.pensize(5)
t.pencolor("black")
t.seth(140)
t.pendown()
t.speed(10)
t.circle( - 300,60)
t.fd(100)
＃1ye
t.seth(10)
t.fd(50)
t.fillcolor("green")
t.begin_fill()
t.right(40)
t.circle(120,80)
t.left(100)
t.circle(120,80)
t.end_fill()
t.seth(10)
t.fd(90)
t.speed(11)
t.penup()
t.fd( - 140)
t.seth(80)
＃2ye
t.pendown()
t.speed(10)
t.fd(70)
t.seth(160)
t.fd(50)
t.fillcolor("green")
t.begin_fill()
t.right(40)
t.circle(120,80)
```

```
t.left(100)
t.circle(120,80)
t.end_fill()
t.seth(160)
t.fd(90)
t.speed(11)
t.penup()
t.fd(-140)
t.seth(80)
t.pendown()
t.speed(10)
#
t.fd(100)
#1ban
t.seth(-20)
t.fillcolor("crimson")
t.begin_fill()
t.circle(100,100)
t.circle(-110,70)
t.seth(179)
t.circle(223,76)
t.end_fill()
#2ban
t.speed(11)
t.fillcolor("red")
t.begin_fill()

t.left(180)
t.circle(-223,60)
t.seth(70)
t.speed(10)
t.circle(-213,15)     #55
t.left(70)            #125
t.circle(200,70)
t.seth(-80)
t.circle(-170,40)
t.circle(124,94)
t.end_fill()

t.speed(11)
t.penup()
t.right(180)
t.circle(-124,94)
t.circle(170,40)
t.pendown()
t.speed(10)

t.seth(-60)
t.circle(175,70)
t.seth(235)
t.circle(300,12)
t.right(180)
t.circle(-300,12)
```

```
t.seth(125)
t.circle(150,60)
t.seth(70)
t.fd(-20)
t.fd(20)

t.seth(-45)
t.circle(150,40)
t.seth(66)
t.fd(-18.5)
t.fd(18.5)

t.seth(140)
t.circle(150,27)
t.seth(60)
t.fd(-8)
t.speed(11)
t.penup()
t.left(20.8)
t.fd(-250.5)

#3ban
t.pendown()
t.speed(10)
t.fillcolor("crimson")
t.begin_fill()
t.seth(160)

t.circle(-140,85)
t.circle(100,70)
t.right(165)
t.circle(-200,32)
t.speed(11)
t.seth(-105)
t.circle(-170,14.5)
t.circle(123,94)
t.end_fill()
```

运行程序，效果如图 3-18 所示。

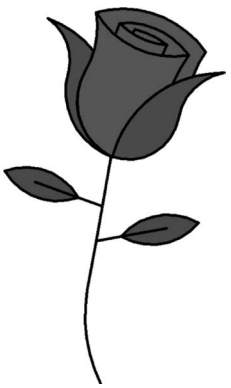


图 3-18　盛开的玫瑰花