《人工智能》课程系列

Part II: Python 算法基础*

武汉纺织大学数学与计算机学院

杜小勤

2018/07/31

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: October 7, 2019。

1 引言

1.1 计算机科学

什么是计算机科学，它的定义是什么？即便计算机科学及其相关理论与技术已经发展了将近一个世纪，我们也难以给出一个明确统一的定义。部分原因在于，它涉及的理论与技术非常广泛。

如果试着从下面 2 个方面来划分研究领域的话，那么情形也许会变得简单明朗一些：以计算机作为研究对象；以计算机作为工具。由此，可以划分出第一个研究领域：研究计算机本身的软硬件理论与技术——硬件的开发与设计 (CPU、存储器、外部设备等)、软件的开发与设计 (操作系统、编译器、各类工具软件等)。第二个主要领域是，将计算机视为工具并借助它进行各类问题的求解。该领域的理论与技术包罗万象，诸如人工智能、机器学习、计算机游戏等诸多领域都可以划归到此类中。

但是，在这些研究领域之外，还存在着一个非常特殊的领域，你很难将它划归到哪个类别中。它无处不在，渗透于各个领域中，极大地影响着计算机自身的

运行及求解问题的效率。

这个领域就是计算机算法，它是整个计算机领域的灵魂。在一些特定的情况下，它甚至可以与计算机科学领域相提并论。硬件、操作系统、编译器、计算机网络等软硬件的设计都离不开算法。从早期的计算机到现代计算机，从原始的编译器到现代编译器，从早期的人工智能 (例如，搜索与规划技术) 到现代的人工智能 (例如，神经网络与深度学习技术)，算法一直是这些领域的主要与重要的研究对象。

基于此，我们从计算机算法的角度，给出计算机科学的定义——计算机科学是研究如何利用计算机算法来有效求解问题的学科[1]。因此，计算机科学家的目标是研究与开发出针对问题的有效算法并在计算机上编程实现，以解决该问题。

在计算机发展的早期，利用计算机求解的问题主要集中于数值计算问题，范围较窄。随着计算机应用范围的扩大，需要求解的问题包罗万象，已经深入到了各个领域。当然，计算机系统本身的研究也在其中。

1.2 计算机算法

与计算机科学的定义一样，也很难给出一个完整统一的计算机算法定义。从数据处理的角度看，我们可以给出一个非正式的定义，计算机算法是一个计算机处理数据的计算过程——给定输入数据，能够生成输出数据。因此，计算机算法可以看作是一个将输入数据变换到输出数据的计算序列。从问题求解的角度看，计算机算法也可以看作是求解具有明确需求的计算问题的工具。

一个算法被认为是正确的，如果对于每一个输入数据，它都能够在有限的计算步骤后停止运行，并输出正确的数据，我们就称算法解决了该问题。一个算法被认为是不正确的，如果对于某些输入，它根本不会停止运行，或者，可能输出非预期的输出。如果非预期的输出能够控制在允许的范围内，这些算法不一定是无用的。然而，一般情况下，我们只关注那些正确的算法。

在设计与实现算法的过程中，需要一种合理有效的方式对输入、输出及中间数据进行存储与管理，以便算法能够快速地存取与修改它们，用于管理数据的模块被称为数据结构。不存在单一的数据结构，能够有效应对各类算法所需的数据管理任务。因此，需要了解几种常见的数据结构的优点与缺点。

[1]在此定义下，计算机科学中的“计算机”有误导之嫌，因为定义已明确表明，计算机不是该领域的主要研究对象，而是作为一个运行的工具而存在。

算法的复杂度 (或复杂性) 由它所消耗的计算资源来度量，所需要的资源越多，算法的复杂度就越高；反之，所消耗的资源越少，复杂度就越低。对于计算资源来讲，算法运行的时间与消耗的空间是最重要的 2 个因素。因此，算法复杂度分为时间复杂度与空间复杂度。

如何度量时间复杂度呢？可以采用 2 种方式：

- 实际测时法

  使用计时程序实际统计给定数据下算法的运行时间。但是，这种方法的缺点比较明显：先要实现算法；统计量依赖于计算机的软硬件环境；不易对算法进行理论分析；

- 分析估算法

  这种分析方法撇开特定的软硬件因素 (例如，实现算法的语言、编译器、CPU 速度等)，或者，它并不比较绝对的运行时间，而是将注意力集中于算法与问题本身——选取算法中的基本操作，并考虑问题的规模，以该基本操作重复执行的次数作为时间复杂度[2]。

下面通过示例来表明，时间复杂度是一个比计算机软硬件的差异性更重要的因素。现在考虑 2 个排序算法。一个是插入排序，它的时间复杂度是 c_1n^2，其中 c_1 是常量，n 是待排序的数据个数。另一个是归并排序，它的时间复杂度是 c_2nlog_2n，其中 c_2 是常量，一般情况下有 $c_1 < c_2$。对于小规模问题，插入排序通常要比归并排序快。但是，对于大规模问题，情形截然不同。无论 c_1 与 c_2 的大小如何变化，在这 2 种算法的时间复杂度曲线之间总存在一个交点，它是运行时间的分水岭——在该交点之后，归并排序有很大的优势。

假设现有 2 台计算机 A 和 B，A(CPU) 的速度为每秒十亿条指令，B(CPU) 的速度为每秒 1 千万条指令，A 的速度是 B 的 100 倍。考虑到编译器与算法实现人员的差异性，并且为了凸显时间复杂度的作用，假设插入排序的实现质量要高于归并排序。设插入排序的常量因子为 $c_1 = 2$，归并排序的常量因子为 $c_2 = 50$，那么对 100 万个数据进行排序，它们的计算时间分别为：

$$\begin{aligned} \frac{2 * (10^6)^2 instructions}{10^9 instructions/second} &= 2000 seconds \\ \frac{50 * 10^6 log_2 10^6 instructions}{10^7 instructions/second} &\approx 100 seconds \end{aligned} \tag{1}$$

[2]时间复杂度是问题规模 n 的函数。

可以看出，即便 B 的 CPU 速度及算法运行环境不如 A，但是归并排序的运行速度仍然大约是插入排序的 20 倍。当问题规模进一步变大时，假如有 1000 万个数据，那么归并排序的优势将更加明显：插入排序需要 2.3 天，而归并排序仅需要 20 分钟。显然，对算法的时间复杂度进行分析，有助于我们预估算法的性能。

类似于算法的时间复杂度，空间复杂度也是问题规模 n 的函数。如果程序所需的空间是常量，那么我们称该算法是原地工作的。当然，这种算法是最受欢迎的。但是，一些算法，尤其是递归算法，并不满足这个条件，请看计算阶乘的例子：

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

每次递归调用函数 factorial，都会创建一个新栈，直到调用返回时才会释放栈空间。因此，递归调用到最深处时，算法将会创建 n 个栈。对于 n 较小的情况，这当然不是什么问题，但是，当 n 超过某个数值时，该算法却不能正常工作：

```
>>> len(str(factorial.factorial(998)))
2562
```

上例计算出了 998 阶乘的结果——一个有 2562 位数的整数！这是一个天文数字。如果接着输入：

```
>>> factorial.factorial(999)
...
 factorial
    if n == 1:
RecursionError: maximum recursion depth exceeded in comparison
```

此时触发了 Python 的异常处理代码，表明算法无法正常工作[3]！可见，对算法进行空间复杂度的分析，有助于我们预估算法的空间使用量，并采取措施解决空间利用率问题。

[3]实际上，Python 的设计者对程序的递归深度做了限制，具体的深度值与 Python 的版本有关，典型的深度值为 1000(可以使用 sys 包的 getrecursionlimit 函数获取缺省值，使用 setrecursionlimit 修改缺省值)。此外，对于算法出现的栈溢出问题，可以使用尾递归技术来解决。

上述 2 个例子表明，对于算法的时间和空间性质，最重要的是算法的数量级，这是算法代价的主要部分。代价函数中的常量因子通常可以忽略不计。例如，通常认为 c_1n^2 和 c_2n^2 属于同一个数量级。

通常情况下，我们使用渐近表示的方法来讨论算法的时间和空间与输入规模之间的关系。为什么需要考虑问题的规模呢？原因在于，当问题规模非常小时，几乎所有的算法都是非常高效的。而一般情况下，我们更关心的是算法处理大规模输入时的性能，尤其是在人工智能与深度学习蓬勃发展的今天。为了量化算法性能，我们使用渐近复杂度来描述当输入规模接近无限大时算法的复杂度，它有以下规则：

- 如果代价函数是乘法项的和，保留增长最快的项，去掉其它项；
- 如果只剩下乘法项，去掉所有常数项；

最常用的渐近表示被称为“大 O(Big O)”表示法，它给出了代价函数渐近增长的上界。例如，假设算法的复杂度函数为 $f(n) \in O(n^2)$，它表示代价函数的增长速度不会超过二次多项式 n^2。而当我们说代价函数的复杂度函数 $f(n) = O(n^2)$，其含义是，对于最坏情况下的复杂度来讲，n^2 既是上界，也是下界，这被称为“紧界”。

虽然可以选择任意合适的函数作为复杂度函数，但是，在算法和数据结构领域，一般使用下面的一组渐近复杂度函数：

$$O(1), O(logn), O(n), O(nlogn), O(n^2), O(n^3), O(2^n), O(n!), \tag{2}$$

其中，$O(1)$ 表示常量复杂度，$O(logn)$ 表示对数复杂度，$O(n)$ 表示线性复杂度，$O(nlogn)$ 表示对数线性复杂度，$O(n^2)$ 表示平方复杂度，$O(n^3)$ 表示立方复杂度，$O(2^n)$ 表示指数 (几何) 复杂度，$O(n!)$ 表示阶乘 (组合) 复杂度。它们随输入规模的函数曲线如图1-1所示。

1.3 抽象性

在计算机的世界里，抽象性似乎是必然的。无论是何种类型的数据，它的最原始最底层的表示仍然是诸如“00110100...”这样的二进制串。在较高层，这些数据可以被解释成字符串、整型或浮点型等数据类型。当然，这些数据类型属于基本的 (或简单的) 数据类型。对于复杂的问题来讲，这些基本的数据类型不足以应

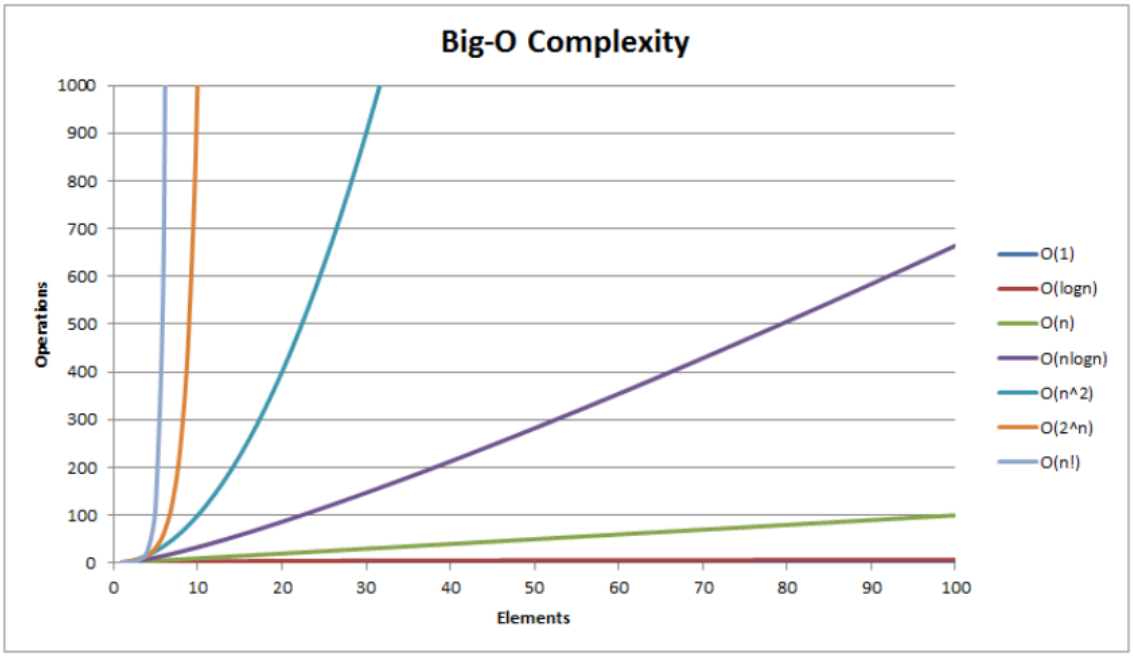

图 1-1: 几种渐近复杂度的函数曲线

对，必须引入复杂的数据类型。实际上，从二进制串，到基本数据类型，再到复杂数据类型，它们形成了一个抽象层次。有了较高层次的抽象，我们就可以方便地利用它们解决更高层次的问题。

抽象是一种信息屏蔽或隐藏机制，它能够让我们将注意力只集中于与当前问题紧密相关的那些因素，而忽略掉那些无关的因素。抽象机制无处不在，它是一种简化问题、提高问题求解效率的有效武器。例如，对于普通的办公人员来讲，将计算机当作文字处理工具，只需要知道基本的使用方法，而不用知道这些文字处理软件是如何编写的。即使是那些能够胜任程序编写任务的程序员，也无需知道计算机软硬件的方方面面。

有 2 种抽象机制：过程抽象 (Procedural Abstraction) 和数据抽象 (Data Abstraction)。过程抽象允许我们只关注过程的使用方法，而无需知道它是如何实现的。例如，假设在问题求解的过程中，我们需要模块 min：返回一组数据中的最小值。无论是设计阶段，还是实现阶段，我们都可以将该模块当作一个标准模块[4]，从而无需考虑它的实现细节。在这种情况下，我们可以集中更多的精力去解

[4]事实也的确如此。在概念阶段，对 min 的使用也不会产生任何歧义；在实现阶段，各种程序设计语言也提供了相应的函数实现。

决更复杂更重要的事情。数据抽象也大体如此，它将数据类型的定义 (包括属性与操作) 与具体实现进行了分离。数据抽象提供了统一的操作接口，系统设计人员、模块设计人员、程序员只需按照统一的接口规范开展自己的工作，互不干扰。

大多数情况下，抽象性与层次性是紧密关联的。例如，计算机网络中的 5 层或 7 层网络模型，由低到高，一层比一层更抽象，最底层是实际的物理层，而最高层是应用层，更接近实际的用户。再举一个例子，常用的整数运算，最底层是硬件实现的整数运算，然后是汇编语言层提供的整数算术运算指令，再往上是高级语言提供的运算符或语句，最后是支持大整数运算的语句。

1.4 抽象数据类型

抽象数据类型 (Abstract Data Type, ADT) 把数据类型定义为抽象的数据类型，只为它们定义可用的操作 (接口)，并不指定数据表示与操作实现的具体细节。ADT 的最大特点是提供统一的接口，并且独立于实现。

实际上，ADT 也是计算机领域中广泛用于程序模块设计的技术。设计者在设计程序模块时，首先给出一个清晰的边界，通过一套接口描述模块提供的功能，但并不限制功能的实现方法。对于模块使用者来讲，也只需要通过接口使用该模块，并不需要知道模块的实现细节[5]。另一方面，模块的实现者根据模块的规范实现模块的所有功能，从形式与实际效果上满足模块的要求。

ADT 提供的接口可以分成四类：

- Constructors

  负责创建和初始化 ADT 的实例对象；

- Accessors

  负责返回 ADT 实例对象中的属性 (不做任何修改)；

- Mutators

  负责修改 ADT 实例对象中的属性；

[5]ADT 可以被看作是一种黑盒。设计人员与模块用户都不需要知道模块的实现细节，这两者在同一层次上形成一种生产者-消费者的关系，即他们并不依赖模块的实现细节。而模块的实现细节，则由处于较低层次的模块实现者负责实现。

- Iterators

  负责遍历 ADT 实例对象中的所有属性；

1.5 数据结构

数据结构是抽象数据类型的具体实现，数据的实现需要程序设计语言中的基本数据类型及其 “粘合剂”[6]，而操作的实现则需要算法。

在实现的过程中，需要特别关注数据结构的实现效率——不仅要考虑如何把抽象的数据类型映射到适合计算机表达和操作的存储形式，还要考虑操作的具体执行方式与执行效率 (这是算法要考虑的问题)。我们把抽象数据类型到数据结构的映射称为抽象数据结构的物理实现。

尽管数据结构要考虑具体的实现细节问题，但是我们仍然需要给出数据结构的抽象定义与分类[7]。原因在于，数据结构中存在一些通用的模式，正确的分类有助于我们正确合理地使用它们。

从逻辑上看，数据结构包含元素集合及其关系，它是一个二元组：

$$D = (E, R) \tag{3}$$

其中，E 是数据结构 D 的元素集合，$R \in E \times E$ 是元素之间的某种关系。对于不同类型的数据结构，元素之间的关系具有不同的性质。

下面是一些典型的数据结构：

1. 集合结构

   数据元素之间没有需要特别关注的明确关系，即 R 是空集。这是最简单的数据结构；

2. 序列结构

   数据元素之间具有一种明确的先后顺序关系。有一个排在最前面的元素，也有一个排在最后面的元素，中间的每个元素都有一个唯一的前驱元素，也有一个唯一的后继元素，所有的元素排列成一个线性序列。关系 R 是一种线性顺序关系。该结构也被称为线性结构，它的实例化对象被称为序列对象。在实际应用中，序列结构有几种变形，例如环形结构 (首尾相接)、ρ 形结构；

[6]粘合剂可以把一些基本的数据对象组合起来，形成更复杂的数据对象。

[7]即便研究具体的对象，以抽象的方式对它们进行分析、总结与归纳，总是正确的思考方式。

3. 层次结构

   数据元素位于不同的层次上，上层元素可以关联一个或多个下层元素。关系 R 形成一种明确的层次性；

4. 树形结构

   它是层次结构中最简单的一种关系。其特点是只有 1 个最上层数据元素 (根)，其余的数据元素有且只有一个上层元素；

5. 图结构

   数据元素之间可以形成任意复杂的相互关系。上述结构都是图结构的特例，它们是受限的图结构；

上述数据结构，规定了数据元素之间的关系，被称为结构性数据结构。还有一类数据结构，没有规定数据元素之间的关系，而是从功能的角度规定了数据结构应该具备什么功能。其中，有一种被称为“容器”的特殊数据结构，没有明确指定数据元素之间要满足的关系，而是规定它在数据元素的读与写方面应该满足的要求。这类数据结构被称为是功能性数据结构，常见的例子有：栈、队列、优先队列、字典等。

由于只有功能的要求，这类数据结构可以采用任何技术实现。在实际实现时，常常将这类数据结构映射到某种结构性数据结构，有时也会开发一些专门的实现技术。

1.6 数据的内存映射

在程序运行时，无论多么复杂的数据对象，都需要在计算机的线性内存区域进行存储。为了存储对象，需要根据当时的内存环境，在空闲的内存中确定一块或若干块区域，以便将数据存入其中。

不同的程序设计语言有不同的处理方式。例如，Python 专门有一个存储管理模块，它负责对象的空间分配，当对象不再被使用时，其空间将被自动回收。这一机制的实现依赖于对象的标识 (ID 号)，在对象存续期间，其 ID 号是不会改变的[8]。

有了对象的 ID 号，就可以通过它直接访问该对象。如果是简单的对象，那么这种映射是直接的。如果是一个复杂的对象，它包含了一组数据元素，那么通常

[8]可以使用内建函数 id、内置操作 is 或 is not 判断是否为同一对象。

有 2 种内存映射模型。第一种是连续存储模型，即在内存空间中安排一个连续的内存空间，将所有的数据元素连续存放在一起。对这种对象的访问，可以通过下标的方式进行。在 Python 中，属于这种存储模型的数据类型有：字符串、列表、元组及字典。第 2 种是链接存储模型，即在每个数据元素里额外地保存着其它数据元素的关联信息，对数据元素的遍历需要这些关联信息。

2 基本的 ADT 及其实现

2.1 Stack

2.1.1 ADT 及其实现

栈是一种线性数据结构，只能在一端 (称为栈顶) 存取数据元素，它是一种 LIFO(Last-In-First-Out) 结构。抽象栈的操作定义如下：

- Stack()

  创建新栈 (空栈)，不需要参数，返回空栈；

- push(item)

  元素入栈，需要 1 个参数 item，不返回任何值；

- pop()

  元素出栈，不需要参数，返回栈顶元素，栈被修改；

- top() 或 peek()

  返回栈顶元素但不移除它，不需要参数，栈未被修改；

- `is_empty()`

  检查栈是否为空，不需要参数，返回布尔值；

- size()

  返回栈的元素个数，不需要参数，返回整数值；

在 Python 中实现栈数据结构，可以利用它提供的基本数据类型——列表来实现。将列表的尾端作为栈顶，调用 append 和 pop 方法进行入栈和出栈处理：

```python
# Completed implementation of a Stack ADT
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def is_empty(self):
        return self.items == []

    def peek(self):
        return self.items[-1]

    def top(self):
        return self.items[-1]

    def size(self):
        return len(self.items)

def main():
    s = Stack()
    print(s.is_empty())
    s.push(1)
    s.push(2)
    s.push(3)
    s.push(4)
    print(s.size())
```

```
    print(s.top())
    print(s.is_empty())
    while not s.is_empty():
        print(s.pop())

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

测试结果如下:

```
True
4
4
False
4
3
2
1
```

如果我们选择将列表的首端作为栈顶，那么就要选用列表的 insert 和 pop 方法 (显式指定位置 0):

```
# Completed implementation of a Stack ADT
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.insert(0, item)

    def pop(self):
        return self.items.pop(0)

    def is_empty(self):
```

```
13          return self.items == []
14
15      def peek(self):
16          return self.items[0]
17
18      def top(self):
19          return self.items[0]
20
21      def size(self):
22          return len(self.items)
23
24  def main():
25      s = Stack()
26      print(s.is_empty())
27      s.push(1)
28      s.push(2)
29      s.push(3)
30      s.push(4)
31      print(s.size())
32      print(s.top())
33      print(s.is_empty())
34      while not s.is_empty():
35          print(s.pop())
36
37  if __name__ == '__main__':
38      main()
```

执行程序，可以发现结果没有任何变化——改变 ADT 的实现并不会影响使用 ADT 接口的那些代码！但是，要注意一点：即使两种实现在逻辑上是等价的，但它们的性能是完全不同的，前者 push 和 pop 方法的时间复杂度是 $O(1)$，而后者的时间复杂度却是 $O(n)$。

2.1.2 括号匹配算法

括号匹配算法检测括号是否成对出现，并且按从左到右的顺序，左括号必须要与同层次的右括号匹配。例如，“(())”符合要求，而“)()(”、“(()()”不符合要求。经过仔细分析，可以发现，在匹配过程中，最内层左括号必须与最内层右括号匹配。类似地，最外层左括号必须与最外层右括号匹配，中间的括号也遵循一样的模式。对于这种相同层次且具有 LIFO 模式的匹配问题，可以考虑使用栈作为数据结构，算法如下[9]：

```
def parentheses_check(str):
    s = Stack()
    balanced = True
    while str is not empty and balanced:
        read symbol from str
        if symbol is '(':
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                s.pop()
    if (s.is_empty() and balanced):
        return True
    else:
        return False
```

将它转换到 Python 程序是直接的：

```
1 from stack import Stack
2
3 def parentheses_check(str):
4     s = Stack()
5     balanced = True
```

[9]为使算法描述显得简洁，尽可能使用 ADT 方法及那些“见其名知其意”的基本模块 (函数)。

```
6       index = 0
7       while index < len(str) and balanced:
8           symbol = str[index]
9           if symbol == "(":
10              s.push(symbol)
11          else:
12              if s.is_empty():
13                  balanced = False
14              else:
15                  s.pop()
16
17          index = index + 1
18
19      if s.is_empty() and balanced:
20          return True
21      else:
22          return False
23
24  def main():
25      print(parentheses_check('((()))'))
26      print(parentheses_check('(()'))
27      print(parentheses_check('((()))'))
28      print(parentheses_check('((()()(()))))'))
29
30  if __name__ == '__main__':
31      main()
```

2.1.3 通用符号匹配算法

下面考虑能够处理几种通用符号 (例如“{}”、“[]”、“()” 等) 匹配问题的算法。实际上，该算法与括号匹配算法没有本质上的区别。但是，问题稍微变得复杂一点，除了要考虑左右符号匹配之外，还要考虑符号类型是否匹配。首先，对

原算法进行改进：

```
def symbol_check(str):
    s = Stack()
    balanced = True
    while str is not empty and balanced:
        read symbol from str
        if symbol is in '([{':
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False
    if (s.is_empty() and balanced):
        return True
    else:
        return False

def matches(open, close):
    return open and close are pairs
```

将算法转换成相应的程序：

```
from stack import Stack

def symbol_check(str):
    s = Stack()
    balanced = True
    index = 0
    while index < len(str) and balanced:
```

```
        symbol = str[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.is_empty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False

        index = index + 1

    if s.is_empty() and balanced:
        return True
    else:
        return False

def matches(open, close):
    opens = '([{'
    closes = ')]}'
    return opens.index(open) == closes.index(close)

def main():
    print(symbol_check('{[(((()))][()]}'))
    print(symbol_check('{[]}{(()}'))
    print(symbol_check('[[](((()))]'))
    print(symbol_check('(((){()[()](())})))'))

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

2.1.4 进制转换算法

现在考虑如何将十进制整数转换成二进制整数的问题[10]。基本原理是，对待转换的整数使用除 2 取余法获取各位上的 0 或 1 数字：最先得到的是最低位，最后得到的是最高位，最后将二进制字符串按从高到低的顺序依次连接起来——这又是一个 LIFO 模式，可以使用栈数据结构。算法如下：

```
def binary(dec):
    s = Stack()
    while dec > 0:
        rem = dec % 2
        s.push(rem)
        dec = dec // 2

    bin_str = ''
    while not s.is_empty():
        bin_str = bin_str + str(s.pop())

    return bin_str
```

对应的程序如下：

```
1  from stack import Stack
2  def binary(dec):
3      s = Stack()
4      while dec > 0:
5          rem = dec % 2
6          s.push(rem)
7          dec = dec // 2
8
9      bin_str = ''
10     while not s.is_empty():
11         bin_str = bin_str + str(s.pop())
```

[10]转换到其它进制，可以采用类似的办法。

```
12
13      return bin_str
14
15  def main():
16      print(10, binary(10))
17      print(15, binary(15))
18      print(256, binary(256))
19      print(1024, binary(1024))
20
21  if __name__ == '__main__':
22      main()
```

可以看出，由于 Python 语言的简洁性，使用它来描述算法也是相当称职的[11]。对算法稍加修改，可以用于其它进制的转换：

```
def anybase(dec, base):
    s = Stack()
    while dec > 0:
        rem = dec % base
        s.push(rem as SPECIAL_DIGIT)
        dec = dec // base

    bin_str = ''
    while not s.is_empty():
        bin_str = bin_str + str(s.pop())

    return bin_str
```

注意，算法中的 “rem as SPECIAL_DIGIT” 表示需要将 rem 转换成相应的进制基本数字，例如，16 进制中的基本数字是 “0123456789ABCDEF”。程序如下：

[11] 从效率的角度考虑，程序的第 9-11 行是有问题的。因为 bin_str 是字符串，它是不可变类型，每次执行第 11 行，都会创建一个新字符串对象。正确的做法是，使用列表代替字符串对象，利用 append 方法添加字符，最后调用''.join 方法将所有列表元素连接起来生成字符串对象。但是，考虑到本问题中字符串的长度并不长，影响可以忽略不计。

```python
from stack import Stack
def anybase(dec, base):
    digits = '0123456789ABCDEF'
    s = Stack()
    while dec > 0:
        rem = dec % base
        s.push(digits[rem])
        dec = dec // base

    bin_str = ''
    while not s.is_empty():
        bin_str = bin_str + str(s.pop())

    return bin_str

def main():
    print(10, anybase(10, 16))
    print(15, anybase(15, 8))
    print(255, anybase(255, 16))
    print(256, anybase(256, 16))
    print(1024, anybase(1024, 2))

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

2.1.5 表达式转换算法

算术表达式包括 2 种元素：运算符和操作数。运算符具有一定的优先级与结合性，规定 * 和/的优先级高于 + 和-，在优先级相同的情况下，按从左到右的顺序进行计算。例如，表达式 $A+B*C+D$ 应该被解释成 $((A+(B*C))+D)$。实际上，这种表示方法被称为中缀表示法。在程序求值的过程中，需要额外地增加括号，以解决运算符的优先级问题。

还有 2 种表示法，它们分别是前缀表示法和后缀表示法。例如，$A+B*C+D$ 的前缀表达式是 $++A*BCD$，后缀表达式是 $ABC*+D+$。仔细分析，可以看出，这 2 种表示法并不需要括号表达优先级，因为运算符出现的位置已经反映了优先级。在前缀表达式中，最先计算 $*BC$[12]，得到中间结果，设名称为 E，接着计算 $+AE$，得到中间结果，设名称为 F，最后计算 $+FD$，得到正确的结果。同样的分析，表明后缀表达式也具有这个特点。

下面介绍将中缀表达式转换到后缀表达式的算法。

首先要明确一点，表达式转换后，无论是前缀表达式还是后缀表达式，操作数的相对位置是不会发生变化的，而运算符的相对位置可能会发生变化。例如，对于表达式 $A*B+C$，其前缀表达式是 $+*ABC$，后缀表达式是 $AB*C+$。其次，需要讨论一下，算法需要用到的数据结构。由于运算符的位置可能会发生变化，因此需要一种机制先暂时保存运算符，等到某种条件成立时，再取出该运算符。基于以上分析，我们可以选用栈数据结构来保存运算符。下面来模拟一下处理过程。对于 $A*B+C$，按如下步骤处理：得到字符 A，直接输出操作数 A，得到字符 $*$，直接入栈，得到字符 B，直接输出操作数 B，得到字符 $+$，此时需要比较运算符优先级，$+$ 优先级低于栈顶 $*$，弹出 $*$ 并输出，$+$ 入栈，得到字符 C，直接输出操作数 C，此时整个字符串为空，一一弹出栈顶运算符并输出，最终得到后缀表达式为 $AB*C+$。

对于表达式含有括号() 的情况，可以进行类似的处理。我们规定，左右括号也是一种运算符，它的优先级最低；左括号只有碰到右括号，才能出栈（但它们并不输出到后缀表达式中）。例如，对于中缀表达式 $(A+B)*C$，按如下相似步骤进行：得到字符(，直接入栈，得到字符 A，直接输出操作数 A，得到字符 $+$，它的优先级高于(，$+$ 入栈，得到字符 B，直接输出操作数 B，得到字符)，优先级低于栈顶 $+$，$+$ 出栈并输出，此时) 与栈顶运算符(相遇，(弹出 (但不输出)，得到字符 $*$，$*$ 入栈，得到字符 C，直接输出操作数 C，整个字符串无可处理字符，于是一一弹出栈顶元素并输出，最终得到后缀表达式为 $AB+C*$。

综合考虑上述各类情况，给出中缀表达式转换到后缀表达式的算法：

```
def infix_postfix(expr):
    s = Stack()
    result = ''
```

[12]运算符在前，后跟 2 个操作数。

```
    expr = expr.upper()
    while expr is not empty:
        read token from expr
        if token is operand:
            result = result + token
        elif token is (:
            s.push(token)
        elif token is ):
            while True:
                opr = s.pop()
                if opr == '(':
                    break
                result = result + opr
        elif token is in '*/+-':
            if not s.is_empty() and token has higher precedence over s.top():
                s.push(token)
            else:
                while not s.is_empty() and token has lower or equal precedence
                      over s.top():
                    opr = s.pop()
                    result = result + opr
                s.push(token)
    while not s.is_empty():
        opr = s.pop()
        result = result + opr
    return result
```

程序如下：

```
from stack import Stack
def infix_postfix(expr):
    s = Stack()
    result = ''
```

```
    expr = expr.upper()
    index = 0
    priority = {'*':3, '/':3, '+':2, '-':2, '(':1}
    while index < len(expr):
        token = expr[index]
        if token in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' or token in '0123456789':
            result = result + token
        elif token == '(':
            s.push(token)
        elif token == ')':
            while True:
                opr = s.pop()
                if opr == '(':
                    break
                result = result + opr
        elif token in '*/+-':
            if not s.is_empty() and priority[token] > priority[s.top()]:
                s.push(token)
            else:
                while not s.is_empty() and priority[token] <= priority[s.top()]:
                    opr = s.pop()
                    result = result + opr
                s.push(token)

        index = index + 1

    while not s.is_empty():
        opr = s.pop()
        result = result + opr
    return result

```

```
36 def main():
37     print(infix_postfix('A + B * C'))
38     print(infix_postfix('(A+B)*c'))
39     print(infix_postfix('(A+B) * (C +D)'))
40     print(infix_postfix('(A+B)+(C+D)*D+A*D'))
41
42 if __name__ == '__main__':
43     main()
```

实验结果如下:

```
ABC*+
AB+C*
AB+CD+*
AB+CD+D*+AD*+
```

2.1.6 后缀表达式求值算法

后缀表达式求值时，也需要用到栈数据结构。例如，仔细分析后缀表达式'ABC*+'，可以总结出如下计算步骤：如果遇到操作数，直接存入栈；否则，遇到运算符，则弹出栈顶最上面的 2 个元素，将该运算符作用于它们。算法如下:

```
def postfix_eval(expr):
    s = Stack()
    token_list = expr.split()
    for token in token_list:
        if token is in '*/+-':
            op2 = s.pop()
            op1 = s.pop()
            s.push(do_math(op1,op2,token))
        else: #token is operand
            s.push(float(operand))
    return s.pop()
def do_math(op1,op2,op):
```

```
    if op == '*':
        return op1 * op2
    elif op == '/':
        return op1 / op2
    elif op == '+':
        return op1 + op2
    elif op == '-':
        return op1 - op2
```

注意：上面的算法做了一个假定，即 token 之间都是由空格分隔开的。对于一般情况的处理，留下作为一道编程练习题。程序如下：

```
from stack import Stack

def postfix_eval(expr):
    s = Stack()
    token_list = expr.split()
    print(token_list)
    for token in token_list:
        if token in '*/+-':
            op2 = s.pop()
            op1 = s.pop()
            s.push(do_math(op1, op2, token))
        else:
            s.push(float(token))
    return s.pop()

def do_math(op1, op2, op):
    if op == '*':
        return op1 * op2
    elif op == '/':
        return op1 / op2
    elif op == '+':
```

```
22          return op1 + op2
23      elif op == '-':
24          return op1 - op2
25
26  def main():
27      print(postfix_eval('7 8 * 7 +'))
28      print(postfix_eval('7 8 + 7 *'))
29      print(postfix_eval('2 3 + 1 4 + 3 * + 2 4 * +'))
30      print(postfix_eval('4 2 / 8 4 + 2 / + 2 4 * +'))
31
32  if __name__ == '__main__':
33      main()
```

2.1.7 练习

1. 完善前面的表达式转换程序，能够检测表达式中的错误，并引发异常；
2. 编写中缀表达式转换到前缀表达式的转换程序，能够检测表达式中的错误并引发异常；
3. 完善前面的后缀表达式求值程序，能够处理等价表达式，例如“7,3*”、“7, 3 *” 与 “7 3 *” 等，能够检测表达式中的错误并引发异常；
4. 编写前缀表达式求值程序，能够检测表达式中的错误并引发异常；

2.2 Queue

2.2.1 ADT 及其实现

Queue(队列) 是一种被称为 FIFO(First-In-First-Out) 的线性数据结构，在一端 (前端) 移除元素，在另一端 (末端) 加入元素。抽象队列的操作定义如下：

- Queue()

  创建新队列，不需要参数，返回空队列；

- enqueue(item)

向末端添加一个元素，需要参数 item，不返回任何值；

- dequeue()

  从前端移除一个元素，不需要参数，返回元素，队列被修改；

- is_empty()

  测试队列是否为空，不需要参数，返回布尔值；

- size()

  返回队列中的元素个数，不需要参数；

与 Stack 的实现一样，我们再次使用 Python 中的列表对象实现队列，列表的末端作为队列的前端 (移除元素)，列表的首端作为队列的末端 (加入元素)。因此，enqueue 的时间复杂度将是 O(n)，dequeue 的时间复杂度将是 O(1)。类 Queue 的实现如下：

```
1  class Queue:
2      def __init__(self):
3          self.items = []
4
5      def enqueue(self, item):
6          self.items.insert(0,item)
7
8      def dequeue(self):
9          return self.items.pop()
10
11     def is_empty(self):
12         return self.items == []
13
14     def size(self):
15         return len(self.items)
16
17 def main():
```

```
18      q = Queue()
19      q.enqueue(1)
20      q.enqueue(2)
21      q.enqueue(3)
22      q.enqueue(4)
23      q.enqueue(5)
24      print(q.size())
25      while not q.is_empty():
26          print(q.dequeue())
27      print(q.size())
28
29  if __name__ == '__main__':
30      main()
```

2.2.2 打印服务模拟程序

本程序模拟打印任务的随机生成、打印机的打印服务，并统计平均等待时间。它需要如下类：Printer、Task，其中 Printer 用于模拟打印机，Task 用于模拟提交的打印任务。它也需要一个过程 simulation，用于模拟打印任务与打印机之间的交互。下面将一一展开描述。

Printer 的 ADT 描述如下：

- Printer(ppm)

  创建一个 Printer 对象，需要参数 ppm(每分钟打印页数)，返回创建的对象；

- tick()

  模拟打印时间的流逝 (秒)；

- busy()

  返回打印机的当前状态；

- start_task(task)

  启动打印任务，需要参数 task 对象；

Task 的 ADT 描述如下：

- Task(time)

  创建一个 Task 对象，需要参数 time(时间戳)，返回创建的对象；

- wait_time(cur_time)

  计算打印等待时间，需要参数 cur_time，返回等待时间；

打印模拟算法如下：

```
def simulation(serve_time, ppm):
    p = Printer(ppm)
    q = Queue()
    waitingtimes = []
    for cur_time in range(serve_time):
        if new task is arrived:
            t = Task(cur_time)
            q.enqueue(t)

        if (not p.busy()) and (not q.is_empty()):
            nexttask = q.dequeue()
            waitingtimes.append(nexttask.wait_time(cur_time))
            p.start_task(nexttask)

        p.tick()

    calculate average waiting time from waitingtimes
    print average waiting time
```

完整的打印模拟程序如下：

```
1 import random
2 from queue import Queue
3
```

```
4  class Printer:
5      def __init__(self, ppm):
6          self.pagerate = ppm
7          self.currentTask = None
8          self.timeRemaining = 0
9
10     def tick(self):
11         if self.currentTask != None:
12             self.timeRemaining = self.timeRemaining - 1
13             if self.timeRemaining <= 0:
14                 self.currentTask = None
15
16     def busy(self):
17         if self.currentTask != None:
18             return True
19         else:
20             return False
21
22     def start_task(self,newtask):
23         self.currentTask = newtask
24         self.timeRemaining = newtask.get_pages() * 60/self.pagerate
25
26 class Task:
27     def __init__(self,time):
28         self.timestamp = time
29         self.pages = random.randrange(1,21)
30
31     def get_stamp(self):
32         return self.timestamp
33
34     def get_pages(self):
```

```
35          return self.pages
36
37      def wait_time(self, currenttime):
38          return currenttime - self.timestamp
39
40  def simulation(serve_time, ppm):
41
42      p = Printer(ppm)
43      q = Queue()
44      waitingtimes = []
45
46      for cur_time in range(serve_time):
47
48        if newPrintTask():
49            t = Task(cur_time)
50            q.enqueue(t)
51
52        if (not p.busy()) and (not q.is_empty()):
53          nexttask = q.dequeue()
54          waitingtimes.append(nexttask.wait_time(cur_time))
55          p.start_task(nexttask)
56
57        p.tick()
58
59      averageWait=sum(waitingtimes)/len(waitingtimes)
60      print("Average Wait %6.2f secs %3d tasks remaining."%(averageWait,q.size()))
61
62  def newPrintTask():
63      num = random.randrange(1,181)
64      if num == 180:
65          return True
```

```
66      else:
67          return False
68
69  def main():
70      for i in range(10):
71          simulation(3600,5)
72
73  if __name__ == '__main__':
74      main()
```

2.3 Deque

2.3.1 ADT 及其实现

Deque 被称为双端队列，它也是一种线性数据结构。与栈、队列不同的是，元素的添加与移除可以在任一端进行。从这个角度看，双端队列具有栈与队列的全部优点。

Deque 的 ADT 定义如下：

- Deque()

  创建一个空的 Deque 对象，不需要参数，返回空的对象；

- add_front(item)

  将元素 item 添加到前端，需要参数 item，不返回任何值；

- add_rear(item)

  将元素 item 添加到尾端，需要参数 item，不返回任何值；

- remove_front()

  从前端移除一个元素，不需要参数，返回该元素，双端队列被修改；

- remove_rear()

  从尾端移除一个元素，不需要参数，返回该元素，双端队列被修改；

- is_empty()

  测试双端队列是否为空，不需要参数，返回布尔值；

- size()

  返回双端队列中元素的个数，不需要参数，返回整型值；

下面使用 Python 的列表对象来实现双端队列，规定列表的尾端为双端队列的前端，程序如下：

```
class Deque:
    def __init__(self):
        self.items = []

    def add_front(self, item):
        self.items.append(item)

    def add_rear(self, item):
        self.items.insert(0, item)

    def remove_front(self):
        return self.items.pop()

    def remove_rear(self):
        return self.items.pop(0)

    def is_empty(self):
        return self.items == []

    def size(self):
       return len(self.items)

def main():
    dq = Deque()
```

```
25      dq.add_front(1)
26      dq.add_front(2)
27      dq.add_front(3)
28      dq.add_rear(1)
29      dq.add_rear(2)
30      dq.add_rear(3)
31      print(dq.size())
32      while not dq.is_empty():
33          print(dq.remove_front())
34
35      dq.add_front(6)
36      dq.add_front(7)
37      dq.add_front(8)
38      while not dq.is_empty():
39          print(dq.remove_rear())
40
41  if __name__ == '__main__':
42      main()
```

在这种实现方式下，前端操作的时间复杂度是 $O(1)$，而尾端操作的时间复杂度是 $O(n)$。

2.3.2 回文数检测算法

回文数是一种特殊的字符序列——从左到右与从右到左的字符序列是完全一样的。例如，“12321”、“radar” 和 “madam” 就是回文数。如果把回文数放进双端队列中，检测时，同时从前端与尾端移除字符并进行比较，若相同，继续进行下一步的比较；否则，直接退出，表明不是回文数。前一分支一直执行到双端队列为空或仅剩一个元素为止，此时可判断出它是回文数。算法如下：

```
def pal_check(str):
    dq = Deque()
    for ch in str:
```

```
            dq.add_rear(ch)
        result = True
        while dq.size() > 1 and result == True:
            ch1 = dq.remove_front()
            ch2 = dq.remove_rear()
            if ch1 != ch2:
                result = False
        return result
```

程序如下：

```
from deque import Deque
def pal_check(str):
    dq = Deque()
    for ch in str:
        dq.add_rear(ch)
    result = True
    while dq.size() > 1 and result == True:
        ch1 = dq.remove_front()
        ch2 = dq.remove_rear()
        if ch1 != ch2:
            result = False
    return result

def main():
    print(pal_check('1234321'))
    print(pal_check('12344321'))
    print(pal_check('12345321'))
    print(pal_check('radar'))
    print(pal_check('madam'))

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

2.4 Heap

堆 (Heap) 是一种满足特定条件的二叉树，可以被有效地应用于堆排序和优先级队列中[13]。它满足下面 2 个条件：

- 堆序 (Heap-Order)

  任一节点的值小于或等于其子节点的值 (最小堆)，或者，任一节点的值大于或等于其子节点的值 (最大堆)；

- 完全二叉树 (Complete Binary Tree)

  除最后一层外，其余所有层中的节点都是满的，没有空缺；而在最后一层，节点按从左到右的顺序排列，中间也没有空缺；

依据这 2 个条件，还可以推出其它性质：

- 从根节点到叶节点的任一路径上，各节点的值按规定的顺序 (从小到大或从大到小) 排列；
- 根节点的值最小 (最小堆) 或最大 (最大堆)，访问的时间复杂度为 $O(1)$；
- 不同路径上的元素，顺序没有要求；

根据完全二叉树的性质，堆的高度为 $O(logn)$。此外，堆也可以很方便地存入连续的存储结构中，并且可以通过下标很方便地访问所有节点：根节点存放于索引 0 处，它的左子节点存放于索引 $2*0+1$ 处，右子节点存放于索引 $2*0+2$ 处，依此类推——假设 i 为节点 e 的索引，那么 e 的两个子节点的索引分别为 $2*i+1$ 和 $2*i+2$。

图2-2为堆的一个示例，它满足堆的所有性质。下面给出堆的 ADT：

- Heap()

  构造函数，返回空的堆对象；

- heapify(items)

  将序列数据 items 进行堆化，使它成为一个堆；

[13]不要与内存堆概念相混淆。

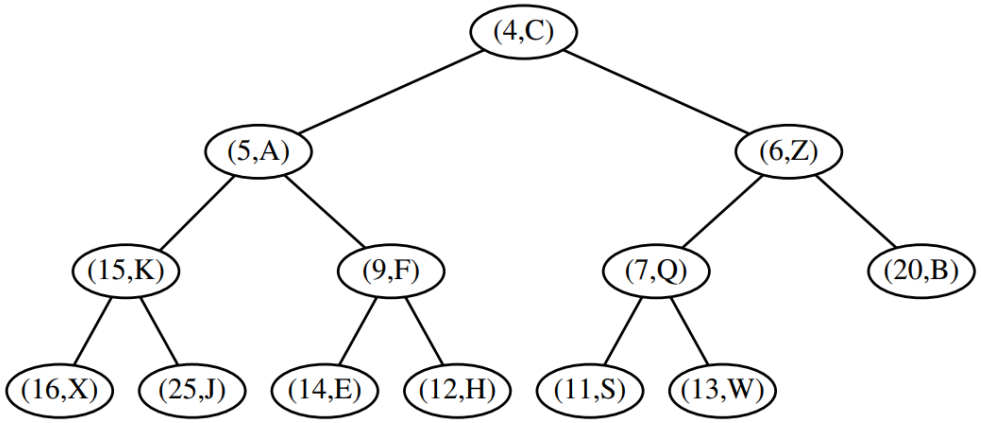

图 2-2: 具有 13 个元素的堆示例

- heappush(item)

  将元素 item 加入堆中；

- heappop()

  将堆顶元素弹出 (其余元素仍然构成堆)，返回该元素：最小堆，返回最小值；最大堆，返回最大值；

- is_empty()

  返回堆是否为空；

- size()

  返回堆的大小；

下面来讨论主要函数的实现方法。

在 heappush(item) 函数中，将一个新元素 item 加入到堆中，为使新数据结构仍维持堆的性质，需要进行一次向上筛选操作：首先，将 item 放入堆的末尾 (新结构仍然构成一棵完全二叉树，但不一定是一个堆)；然后，将新元素沿着它的父节点、父节点的父节点、...、根节点所形成的一条路径，向上进行筛选——将新元素与父节点进行比较，如果不满足堆序要求，就进行交换，此过程一直持续到满足堆序为止。处理完毕，新元素所处的路径上的所有节点均满足堆序，而其余路径没有发生变化。至此，整个完全二叉树已恢复为一个堆。显然，该操作的时间复杂度为 $O(logn)$。

在 heappop() 函数中，需要将堆顶元素弹出。此时，堆顶元素为空，需要补充一个元素，可以将原堆的最后一个元素填入 (新结构仍然构成一棵完全二叉树，但不一定是一个堆。为描述方便，假设该元素为 e)，为使新数据结构仍维持堆的性质，需要进行一次向下筛选操作：将新的根节点 e 与它的 2 个子节点进行比较，选取其中的最小者 (假设为最小堆)，让该最小者与 e 进行交换，元素 e 下移，再将 e 与它的 2 个新子节点进行比较并 (必要时) 交换，此过程一直持续到 e 不再移动为止。处理完毕，整个完全二叉树已恢复成一个堆。显然，该操作的时间复杂度也为 $O(logn)$。

函数 heapify(items) 的作用是将序列数据 items(假设是一个 list 或数组对象，元素个数为 n) 进行堆化。设最后一个非叶节点的索引为 l，显然，从索引 $l+1$ 到索引 $n-1$ 的所有叶节点都是堆 (单节点堆)。对整个序列数据进行堆化的方法如下：首先对以节点 l 为根节点的准堆执行一次向下筛选的堆化处理，接着对以节点 $l-1$ 为根节点的准堆执行一次向下筛选的堆化处理，...，此过程一直持续到以节点 0 为根节点的准堆执行一次向下筛选的堆化处理为止。至此，整个序列数据的堆化工作结束。稍后将分析 heapify 的时间复杂度。

下面给出各个操作的算法。首先，给出向上筛选 sift_up 和向下筛选 sift_down 的算法。然后，在此基础上，给出 heappush、heappop 和 heapify 的算法。

向上筛选 sift_up 的算法如下：

```
Input:
    self: class instance
    items: sequence data
    item: new element
    pos: the item's position in items
Output:
    items is changed
def sift_up(self, items, item, pos):
    parent_pos = (pos-1)//2
    while parent_pos >= 0 and item < items[parent_pos]:
        items[pos] = items[parent_pos]
        pos, parent_pos = parent_pos, (parent_pos-1)//2
    items[pos] = item
```

向下筛选 sift_down 的算法如下：

```
Input:
    self: class instance
    items: sequence data
    item: new element
    pos: the item's position in items
    len: the items' size
Output:
    items is changed
def sift_down(self, items, item, pos, len):
    end = len - 1
    child_pos = pos*2 + 1
    while child_pos <= end:
        if child_pos+1 <= end and
           items[child_pos+1] < items[child_pos]:
            child_pos = child_pos+1
        if item < items[child_pos]:
            break;
        items[pos] = items[child_pos]
        pos, child_pos = child_pos, 2*child_pos+1
    items[pos] = item
```

插入元素 heappush 的算法如下：

```
Input:
    self: class instance
    item: new element
Output:
    self.items is changed
def heappush(self, item):
    self.items.append(None)
    self.sift_up(self.items, item, len(self.items)-1)
```

弹出堆顶元素 heappop 的算法如下：

```
Input:
    self: class instance
Output:
    self.items is changed
def heappop(self):
    if self.is_empty():
        return None
    top = self.items[0]
    item = self.items.pop()
    if not self.is_empty():
        self.sift_down(self.items, item, 0)
    return top
```

序列数据堆化 heapify 的算法如下：

```
Input:
    self: class instance
    items: sequence data
Output:
    items is changed
def heapify(self, items):
    if len(items) <= 1:
        return
    for x is the index from the last Non-leaf node to root:
        self.sift_down(items, items[x], x)
```

下面分析 heapify 算法的时间复杂度。设序列数据的元素个数为 n，则堆的高度 h 为 $O(logn)$。为讨论方便，假设堆是一棵满二叉树。该算法以自底向上的方式进行堆化调整，第 h 层，总共有 2^h 个叶节点，已经是 (单节点) 堆，无需调整；第 $h-1$ 层，总共有 2^{h-1} 个非叶节点，需要执行向下筛选 (此句以下略)，每个至多比较和移动 1 次，总共需 $1*2^{h-1}$ 次操作；第 $h-2$ 层，总共有 2^{h-2} 个非叶节点，每个至多比较和移动 2 次，总共需 $2*2^{h-2}$ 次操作；第 $h-3$ 层，总共有 2^{h-3}

个非叶节点，每个至多比较和移动 3 次，总共需 $3*2^{h-3}$ 次操作；...；第 1 层，总共有 2^1 个非叶节点，每个至多比较和移动 $h-1$ 次，总共需 $(h-1)*2^1$ 次操作；第 0 层，总共有 2^0 个非叶节点，每个至多比较和移动 h 次，总共需 $h*2^0$ 次操作。因此，该算法的总操作次数为：

$$t = 1*2^{h-1} + 2*2^{h-2} + 3*2^{h-3} + ... + (h-2)*2^2 + (h-1)*2^1 + h*2^0 \quad (4)$$

两端同时乘以 2，得到：

$$2t = 1*2^h + 2*2^{h-1} + 3*2^{h-2} + ... + (h-2)*2^3 + (h-1)*2^2 + h*2^1 \quad (5)$$

继续化简，得到：

$$t = 2t - t = 2^h + 2^{h-1} + 2^{h-2} + ... + 2^3 + 2^2 + 2^1 - h = 2*2^h - h - 2 \quad (6)$$

将 $h = O(logn)$ 代入，得到：

$$t = 2*O(n) - O(logn) - 2 \quad (7)$$

因此，该算法的时间复杂度为 $O(n)$。

下面给出堆的程序：

```
import operator

class Heap():
    def __init__(self, opr='<'):
        self.items = []
        if opr == '<':
            self.opr = operator.lt
        else:
            self.opr = operator.gt

    def sift_up(self, items, item, pos):
        parent_pos= (pos-1)//2
        while parent_pos >= 0 and self.opr(item, items[parent_pos]):
            items[pos] = items[parent_pos]
```

```
15              pos, parent_pos = parent_pos, (parent_pos-1)//2
16          items[pos] = item
17
18      def sift_down(self, items, item, pos, len):
19          end = len - 1
20          child_pos = pos*2 + 1
21          while child_pos <= end:
22              if child_pos+1 <= end and \
23                 self.opr(items[child_pos+1], items[child_pos]):
24                  child_pos = child_pos+1
25              if self.opr(item, items[child_pos]):
26                  break;
27              items[pos] = items[child_pos]
28              pos, child_pos = child_pos, 2*child_pos+1
29          items[pos] = item
30
31      def heappush(self, item):
32          self.items.append(None)
33          self.sift_up(self.items, item, len(self.items)-1)
34
35      def heappop(self):
36          if self.is_empty():
37              return None
38          top = self.items[0]
39          item = self.items.pop()
40          if not self.is_empty():
41              self.sift_down(self.items, item, 0, len(self.items))
42          return top
43
44      def heapify(self, items):
45          if len(items) <= 1:
```

```
46              return
47          n = len(items)-1
48          index = (n-1)//2
49          for x in range(index, -1, -1):
50              self.sift_down(items, items[x], x, len(items))
51
52      def is_empty(self):
53          return self.items == []
54
55      def size(self):
56          return len(self.items)
57
58  def main():
59      d = [3,5,20,4,6,1,50,30,40,15,2]
60      h = Heap()
61      for item in d:
62          h.heappush(item)
63      print(h.size())
64      while not h.is_empty():
65          print(h.heappop(), end=' ')
66      print()
67      print(h.size())
68
69      h.heapify(d)
70      h.items = d
71      while not h.is_empty():
72          print(h.heappop(), end=' ')
73      print()
74
75  if __name__ == '__main__':
76      main()
```

在 Heap 的构造函数中，可以选择构造最小堆或最大堆。运行结果如下：

```
11
1 2 3 4 5 6 15 20 30 40 50
0
1 2 3 4 5 6 15 20 30 40 50
```

2.5 PriorityQueue

优先队列 (Priority Queue) 是一种类似于队列的数据结构，但是它的每个元素都有一个关联的优先级。每次从优先队列中取出的是具有最高优先级的元素；如果几个元素有相同的优先级，那么它们的访问顺序取决于队列中的顺序。因此，优先队列具有最高级先出 (First-In Largest-Out, FILO) 的特点。通常情况下，从效率的角度考虑，可以采用堆数据结构来实现优先队列。

下面是优先队列的 ADT：

- PriorityQueue()

  创建一个优先队列对象，不需要参数，返回空的对象；

- enqueue(item, priority)

  依据给定的优先级 *priority*，将元素 *item* 加入到优先队列中；

- dequeue()

  返回并删除具有最高优先级的元素，优先队列被修改；

- is_empty()

  判断优先队列是否为空；

- size()

  返回优先队列的元素个数；

程序如下：

```
1 import heapq
2
```

```python
class PriorityQueue:
    def __init__(self):
        self.elements = []

    def enqueue(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def dequeue(self):
        return heapq.heappop(self.elements)[1]

    def is_empty(self):
        return self.elements == []

    def size(self):
        return len(self.elements)

def main():
    pq = PriorityQueue()
    pq.enqueue(3, 3)
    pq.enqueue(2, 2)
    pq.enqueue(1, 1)
    pq.enqueue(4, 4)
    pq.enqueue(5, 5)
    print(pq.size())
    while not pq.is_empty():
        print(pq.dequeue())
    print(pq.size())

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

注意，在程序中，优先级数值越小，优先级别越高。

下面分析一下插入元素与删除元素的时间复杂度。由于采用堆来实现优先队列，所以这 2 种操作的时间取决于堆的相关操作：

- 把新元素加入到优先队列 (堆) 后，需要执行一次向上筛选操作，才能确保新的优先队列仍然是一个堆。而在向上筛选的过程中，比较和交换的次数不会超过二叉树中最长路径的长度。因此，插入元素的时间复杂度是 $O(logn)$；
- 从优先队列中删除元素，需要执行：弹出堆顶元素 (优先级别最高)；从堆末尾取一个元素作为二叉树的根 (此时，新的优先队列不是一个堆，需要重新将它调整为堆)；执行一次向下筛选。前 2 步的时间复杂度都是 $O(1)$，最后一步从根节点开始，每步操作需要做 2 次比较，总的操作次数不会超过二叉树中最长路径的长度。因此，删除元素的时间复杂度是 $O(logn)$；

2.6 List

从前面的内容可以看出，Stack、Queue 和 Deque 的底层都是使用 Python 的列表对象实现的，列表是一种非常重要的数据结构。本小节仅讨论 Python 中列表的实现机制，而不会具体给出它的 ADT 及实现技术。

我们知道，Python 的列表对象可以包含不同类型的元素，对它们的访问操作可以通过索引在常数时间内完成。这是如何做到的呢？

实际上，Python 使用了一种所谓的“间接操作[14]”技术来实现列表元素的常数时间访问。一般来说，使用间接操作访问元素，需要先访问另一个元素。事实上，Python 中的变量，就是通过间接操作来完成对它所指向对象的访问。使用变量访问列表中的元素时，实际上会进行两次间接操作。

在 Python 中，列表被表示成一个固定大小的指针对象序列，示意图如图2-3所示。图中展示了列表对象 1 的内存示意，它的每个元素实际上是一个指针对象，最左侧的元素包含一个指向整数的指针，该整数是列表的长度，而其它的元素包含了指向其它对象的指针。通过这种间接操作的方式，实现了 2 个功能：元素的访问可以在常数时间内完成；列表的元素可以是不同类型的对象。

[14]计算机科学家发现该技术可以解决许多问题。人们常说，计算机科学中的所有问题都可以通过添加另一层的间接操作来解决。请参考文献 [5]，P114。

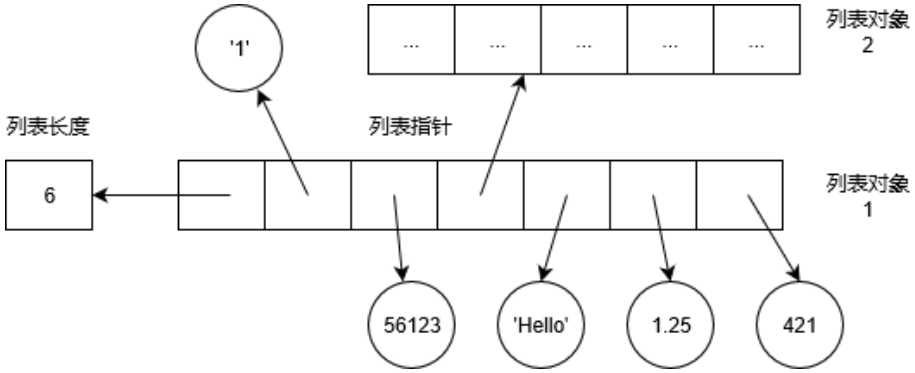

图 2-3: 列表的底层实现

2.7 Array

数组 (Array) 是一种常见的数据元素容器，广泛应用于各个问题中。大多数程序设计语言支持多维度的数组，本部分内容将讨论数组的 ADT 及其实现问题。

2.7.1 一维数组的 ADT 及其实现

一维数组 (one-dimensional array) 可以表示为一个连续的存储空间，并且可以通过索引或下标的方式对它的元素进行访问。整个一维数组被看作一个对象，使用数组名表示该对象。

数组对象与列表 (list) 对象非常相似，两者都是顺序数据，并且都可以根据索引或下标进行访问。但是，两者有以下几个差别：

- 数组的方法较少，而列表的方法较多；
- 数组不能动态改变，而列表可以；

既然列表要比数组强大许多，而且 Python 核心已经提供了列表类型，那么为什么还要讨论数组的 ADT 及其实现呢？原因在于，在一些应用场合，数据元素的个数固定，且只需要对数据元素进行存取操作，此时，选用数组作为数据结构，处理效率较高，是一种明智的选择。此外，列表需要更多的额外空间，整个分配的空间最多可达实际所需容量的 2 倍。如果使用列表对象存放固定数目的元素，那么一半的空间将会被浪费掉。

下面给出数组的 ADT：

- Array(size)

创建一个一维数组，大小为 size(>0)，所有的元素被初始化为 None；

- length()

  返回数组的元素个数；

- getitem(index)

  返回数组中索引为 index 的元素，index 必须在有效范围内；

- setitem(index, value)

  修改数组中索引为 index 的元素值，新值为 value，index 必须在有效范围内；

- clearing(value)

  将数组中所有元素的值设置为 value；

- iterator()

  创建并返回数组迭代器 (用于遍历数组中的元素)；

在实现数组 ADT 时，可以使用 Python 提供的 ctypes 库，它是 Python 标准库的一部分。该库提供了与 C 兼容的数据类型，并允许调用 DLL 或共享库中的函数。因此，ctypes 库为 Python 语言包装 (wrapping)DLL 中的函数提供了方便，它是 Python 与 C 之间的一座桥梁。

下面是使用 ctypes 的一个例子：

```
1  import ctypes
2
3  ArrayType = ctypes.py_object * 10
4  slots = ArrayType()
5  for i in range(len(slots)):
6      slots[i] = None
7  slots[0] = 10
8  slots[3] = 20
9  for i in range(len(slots)):
10     print(slots[i])
```

注意，slots 的元素在引用之前必须要初始化。运行结果如下：

```
10
None
None
20
None
None
None
None
None
None
```

下面给出数组 ADT 的实现：

```
import random
import ctypes

class Array:
    def __init__(self, size):
        assert size > 0, 'Array size must be > 0'
        self.size = size
        PyArrayType = ctypes.py_object * size
        self.elements = PyArrayType()
        self.clear(None)

    def __len__(self):
        return self.size

    def __getitem__(self, index):
        assert index >= 0 and index < len(self), \
               'Array subscript out of range'
        return self.elements[index]

    def __setitem__(self, index, value):
```

```
21          assert index >= 0 and index < len(self), \
22                 'Array subscript out of range'
23          self.elements[index] = value
24
25      def clear(self, value):
26          for i in range(len(self)):
27              self.elements[i] = value
28
29      def __iter__(self):
30          return ArrayIterator(self.elements)
31
32  class ArrayIterator:
33      def __init__(self, theArray):
34          self.arrayRef = theArray
35          self.curNdx = 0
36
37      def __iter__(self):
38          return self
39
40      def __next__(self):
41          if self.curNdx < len(self.arrayRef):
42              entry = self.arrayRef[self.curNdx]
43              self.curNdx = self.curNdx + 1
44              return entry
45          else:
46              raise StopIteration
47
48  def main():
49      a = Array(10)
50      for i in range(len(a)):
51          a[i] = random.random()
```

```
52
53      for i in range(len(a)):
54          print(a[i])
55
56  if __name__ == '__main__':
57      main()
```

2.7.2 二维数组的 ADT 及其实现

二维数组有行与列 2 个维度，访问其中的元素就需要 2 个索引。它的 ADT 定义如下：

- Array2D(nrows, ncols)

  创建一个 nrows 行 ncols 列的二维数组，它的所有元素被初始化为 None；

- numRows()

  返回二维数组的行数；

- numCols()

  返回二维数组的列数；

- clear(value)

  将所有元素设置为 value；

- getitem(r, c)

  返回二维数组中第 r 行第 c 列的元素，r 和 c 必须在有效范围内；

- setitem(r, c, value)

  修改二维数组中第 r 行第 c 列的元素值为 value，r 和 c 必须在有效范围内；

通常，有 2 种方式可以实现二维数组：第 1 种方式，把二维数组转换成一个一维数组 (按行或列的顺序，将元素排成一排放进一维数组中)；第 2 种方式，把二维数组看作 (一维) 数组的 (一维) 数组，即第 1 组的每一个一维数组存放每行的元素，第 2 组只有一个一维数组，它的每个元素分别指向第 1 组的每个一维数组，如图2-4所示。

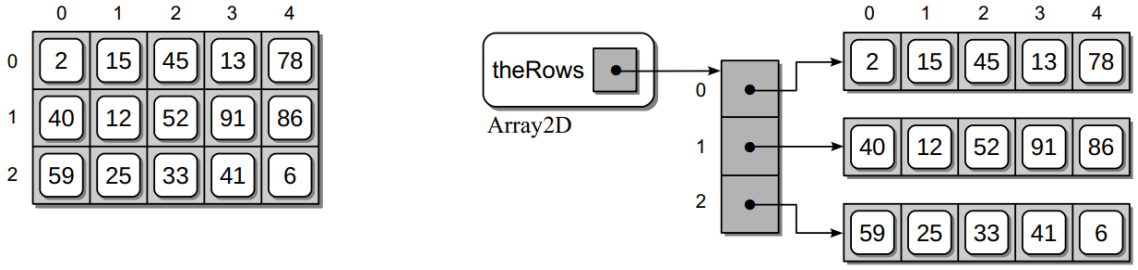

图 2-4: 二维数组的表示：数组的数组

二维数组的 ADT 实现如下：

```
1  import random
2  from myarray import Array
3
4  class Array2D:
5      def __init__(self, numRows, numCols):
6          self.theRows = Array(numRows)
7
8          for i in range(numRows):
9              self.theRows[i] = Array(numCols)
10
11     def numRows(self):
12         return len(self.theRows)
13
14     def numCols(self):
15         return len(self.theRows[0])
16
17     def clear(self, value):
18         for row in range(self.numRows()):
19             self.theRows[row].clear(value)
20
21     def __getitem__(self, ndxTuple):
```

```
22          assert len(ndxTuple) == 2, 'Invalid number of array subscripts.'
23          row = ndxTuple[0]
24          col = ndxTuple[1]
25          assert row >= 0 and row < self.numRows() and \
26                 col >= 0 and col < self.numCols(), \
27                 "Array subscript out of range."
28          the1dArray = self.theRows[row]
29          return the1dArray[col]
30
31      def __setitem__(self, ndxTuple, value):
32           assert len(ndxTuple) == 2, 'Invalid number of array subscripts.'
33           row = ndxTuple[0]
34           col = ndxTuple[1]
35           assert row >= 0 and row < self.numRows() and \
36                  col >= 0 and col < self.numCols(), \
37                  'Array subscript out of range.'
38           the1dArray = self.theRows[row]
39           the1dArray[col] = value
40
41  def main():
42      a = Array2D(10, 5)
43      for r in range(a.numRows()):
44          for c in range(a.numCols()):
45              a[r, c] = random.random()
46
47      for r in range(a.numRows()):
48          for c in range(a.numCols()):
49              print(a[r, c], end=' ')
50          print()
51
52  if __name__ == '__main__':
```

```
53      main()
```

2.7.3 应用：LifeGrid 的生命游戏

生命游戏 (Game of Life) 由英国数学家 John H. Conway 设计，用来模拟“生命”的兴衰与更替。实际上，它是一个零玩家游戏，由 Martin Gardner 在 1970 年 10 月发行的《科学美国人-数学游戏专栏》中刊出。此后，生命游戏受到了广泛的关注与研究，研究者们可以借此观察简单的规则是如何演化出复杂的系统或模式的。生命游戏是元胞自动机 (Cellular Automata) 的早期例子。

生命游戏在一个无边界的网格 (Grid) 中进行演化。网格本身由许多小单元 (cell) 均匀地组成，这些小单元 (细胞)，要么是死亡的细胞 (空单元)，要么是活着的细胞 (填充单元)。死亡或活着的细胞在下一时刻的生死状态由下面几条规则确定：

1. 如果细胞是活的，并且它的相邻邻居[15]中有 2 个或 3 个是活的，那么在下一时刻，该细胞仍然是活的；

2. 如果细胞是活的，并且它的周围没有活着的细胞，或者只有一个活着的细胞，那么在下一时刻，该细胞将会死去；

3. 如果细胞是活的，并且它的周围有 4 个及以上活着的细胞，那么在下一时刻，该细胞也将会死去；

4. 如果细胞是死的，并且它的周围刚好有 3 个活着的细胞，那么在下一时刻，该细胞将会获得重生。在其它情况下，死亡细胞的状态将不会发生变化；

下面看几个有趣的游戏演化图例。在这些例子中，第 1 幅图显示的是初始的细胞状态，每次演化时，所有的细胞都是同时基于上述规则进行演化的。在图例2-5中，经过 2 步演化，所有的细胞都消亡了。在图例2-6中，只经过 1 步演化，细胞状态就变成了固定的模式，不再进一步演化。在图例2-7中，经过 2 步演化，细胞状态就还原到初始状态，此后，呈现出周期性的演化模式。

显然，生命游戏需要一个对象来存储所有细胞的状态，并且需要提供某些抽象操作，我们将该对象抽象出来形成类 LifeGrid。下面是类 LifeGrid 的 ADT：

[15]每个细胞总共有 8 个相邻邻居，它们分布于水平、垂直和斜线方向上。

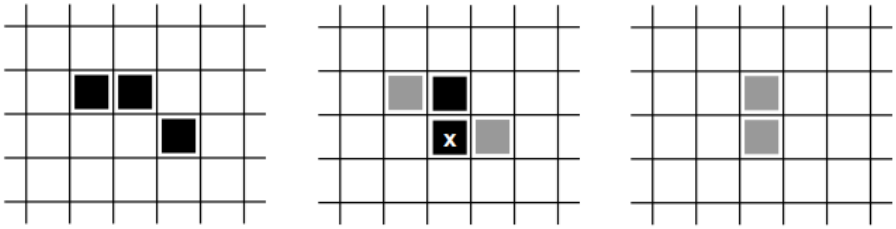

图 2-5: 消亡模式

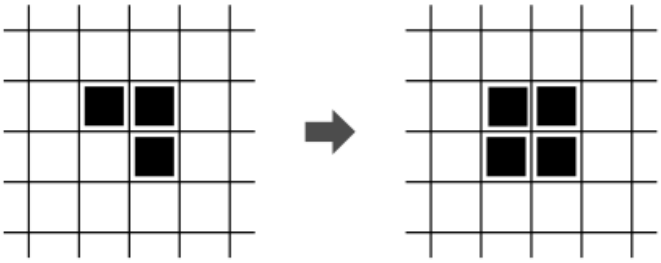

图 2-6: 固定模式

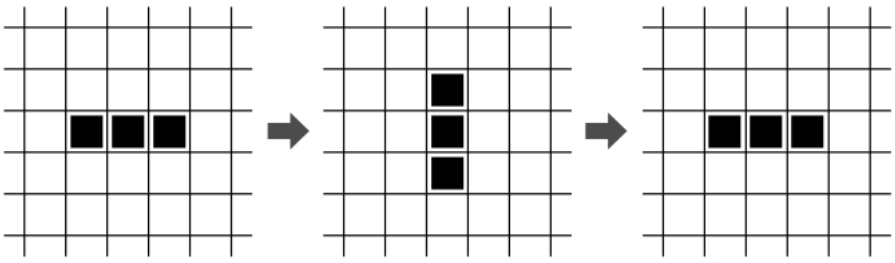

图 2-7: 周期模式

- LifeGrid(nrows, ncols)

  创建一个 nrows 行 ncols 列的 LifeGrid 对象，所有的细胞初始化为“死亡”；

- numRows()

  返回 LifeGrid 的行数；

- numCols()

  返回 LifeGrid 的列数；

- configure(coordList)

  coordList 是一个 2-tuples 序列数据，每个 2-tuples 指定了某行某列处的细胞是活的。其它所有行列索引不在 coordList 里面的细胞是死的；

- clearCell(row, col)

  设置行 row 列 col 处的细胞为死的状态，row 和 col 必须在有效的范围内；

- setCell(row, col)

  设置行 row 列 col 处的细胞为活的状态，row 和 col 必须在有效的范围内；

- isLiveCell(row, col)

  返回行 row 列 col 处的细胞状态：True，活；False，死。row 和 col 必须在有效的范围内；

- numLiveNeighbors(row, col)

  返回行 row 列 col 处活的 (直接) 邻居细胞的数目。默认情况下，边界外的细胞是死细胞。row 和 col 必须在有效的范围内；

下面给出类 LifeGrid 的实现：

```
from myarray2d import Array2D

class LifeGrid:
    DEAD_CELL = 0
    LIVE_CELL = 1
```

```
6
7       def __init__(self, numRows, numCols):
8           self.grid = Array2D(numRows, numCols)
9           self.configure(list())
10
11      def numRows(self):
12          return self.grid.numRows()
13
14      def numCols(self):
15          return self.grid.numCols()
16
17      def configure(self, coordList):
18          for row in range(self.numRows()):
19              for col in range(self.numCols()):
20                  self.clearCell(row, col)
21
22          for coord in coordList:
23              self.setCell(coord[0], coord[1])
24
25      def isLiveCell(self, row, col):
26          return self.grid[row, col] == LifeGrid.LIVE_CELL
27
28      def clearCell(self, row, col):
29          self.grid[row, col] = LifeGrid.DEAD_CELL
30
31      def setCell(self, row, col):
32          self.grid[row, col] = LifeGrid.LIVE_CELL
33
34      def numLiveNeighbors(self, row, col):
35          pos = [(row-1, col-1), (row-1, col), (row-1, col+1),\
36                 (row, col-1),                 (row, col+1),\
```

```
37                 (row+1, col-1), (row+1, col), (row+1, col+1)]
38         sum = 0
39         for p in pos:
40             if 0 <= p[0] < self.numRows() and \
41                0 <= p[1] < self.numCols():
42                 sum += self.grid[p[0], p[1]]
43 
44         return sum
```

下面是生命游戏的实现:

```
1  from lifegrid import LifeGrid
2  
3  INIT_CONFIG = [(1, 2), (2, 1), (2, 2), (2, 3)]
4  
5  GRID_WIDTH = 5
6  GRID_HEIGHT = 5
7  
8  NUM_GENS = 12
9  
10 def main():
11     gamegrid = LifeGrid(GRID_WIDTH, GRID_HEIGHT)
12     gamegrid.configure(INIT_CONFIG)
13 
14     draw(gamegrid)
15     for i in range(NUM_GENS):
16         evolve(gamegrid)
17         draw(gamegrid)
18 
19 def evolve(gamegrid):
20     liveCells = list()
21 
22     for row in range(gamegrid.numRows()):
```

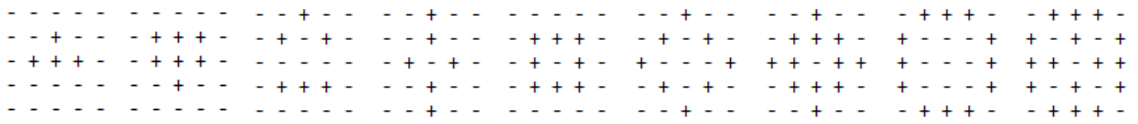

图 2-8: 生命游戏的运行片段示例

```
23          for col in range(gamegrid.numCols()):
24              nlives = gamegrid.numLiveNeighbors(row, col)
25              if (nlives == 2 and gamegrid.isLiveCell(row, col)) or \
26                 (nlives == 3):
27                  liveCells.append((row, col))
28
29      gamegrid.configure(liveCells)
30
31  def draw(gamegrid):
32      print()
33      for row in range(gamegrid.numRows()):
34          for col in range(gamegrid.numCols()):
35              if gamegrid.isLiveCell(row, col):
36                  print('+', end=' ')
37              else:
38                  print('-', end=' ')
39          print()
40      print()
41
42  if __name__ == '__main__':
43      main()
```

程序运行的一个片段如图2-8所示。从第 9 次迭代开始，第 8 幅图与第 9 幅图交替出现，生命游戏进入周期模式。

2.7.4 多维数组

多维数组 (Multi-Dimensional Array) 也是一种数据容器。通常，维度大于等于 2 的数组被称为是多维数组。与一维数组不同的是，它的元素分布于二维平面及以上的多维度空间中，例如二维平面、三维空间、四维空间等。

下面给出它的 ADT 定义：

- MultiArray(d_1, d_2, ..., d_n)

  创建一个多维度数组，它的元素被初始化为 None。构造函数支持可变参数，但是它的参数个数必须大于 1(二维及以上的数组)，每个参数的值必须大于 0(不允许某个维度为 0)。d_1 表示最高维度，d_n 表示最低维度，中间参数表示的维度，依此类推；

- dims()

  返回多维数组的维度；

- length(dim)

  返回指定维度的长度。维度编号从 1 开始，1 表示最高维度。例如，在三维数组中，1 表示深度，2 表示行，3 表示列；

- clear(value)

  设置所有元素的值为 value；

- getitem(i_1, i_2, ..., i_n)

  返回指定索引处的元素值。各维度的索引分别由 i_1、i_2、...、i_n 指定，它们必须在有效的范围内；

- setitem(i_1, i_2, ..., i_n, value)

  设置指定索引处的元素值为 value。各维度的索引分别由 i_1、i_2、...、i_n 指定，它们必须在有效的范围内；

现在考虑多维数组的实现问题。在2.7.2节中，提到过二维数组的 2 种实现方式：二维数组被处理成一维数组形式、数组的数组方式。前者又有 2 种处理方式：行优先 (row-major)、列优先 (column-major)。在行优先方式中，先存入第 1 行，

再存入第 2 行，依此类推；在列优先方式中，先存入第 1 列，再存入第 2 列，依此类推。实际上，这 2 种方式的区别在于，是最高维元素优先保存，还是最低维元素优先保存。但是，不论是行优先，还是列优先，在某个维度内，元素都是按照索引顺序从低到高依次存入的。行优先方式优先保存最高维元素，而列优先方式优先保存最低维元素。在大多数高级程序设计语言中，使用行优先方式[16]。我们也将采用行优先存储方式。

对于多维数组，在实现时，一种简便的方法是将它处理成一维数组。主要原因在于，计算机的存储结构本身是一种自然的一维线性结构。

将多维数组转换到一维数组，最关键的问题是，如何根据元素的多维索引计算出元素在一维数组中的位置，以便进行随机存取。下面将详细讨论该问题。

在二维数组 MultiArray(d_1, d_2) 中，d_1 是最高维长度，d_2 是最低维长度。设二维数组索引为 $(0,0)$(第 0 行第 0 列) 的元素，其一维地址是 0，那么二维数组索引为 (i_1, i_2)(第 i_1 行第 i_2 列) 的元素，其一维地址将是：

$$OFFSET_2(i_1, i_2) = i_1 * d_2 + i_2 \tag{8}$$

在三维数组 MultiArray(d_1, d_2, d_3) 中，d_1 是最高维长度，d_3 是最低维长度。设三维数组索引为 $(0,0,0)$ 的元素，其一维地址是 0，那么三维数组索引为 (i_1, i_2, i_3) 的元素，其一维地址将是：

$$OFFSET_3(i_1, i_2, i_3) = i_1 * d_2 * d_3 + i_2 * d_3 + i_3 \tag{9}$$

依此类推，在 n 维数组 MultiArray(d_1, d_2, d_3,...,d_n) 中，d_1 是最高维长度，d_n 是最低维长度。设 n 维数组索引为 $(0,0,0,...,0)$ 的元素，其一维地址是 0，那么 n 维数组索引为 $(i_1, i_2, i_3, ..., i_n)$ 的元素，其一维地址将是：

$$\begin{aligned} OFFSET_n(i_1, i_2, i_3, ..., i_n) &= i_1 * d_2 * d_3 * ... * d_n + i_2 * d_3 * ... * d_n + ... + i_{n-1} * d_n + i_n \\ &= i_1 * f_1 + i_2 * f_2 + ... + i_{n-1} * f_{n-1} + i_n * f_n = \sum_{k=1}^{n} i_k * f_k \end{aligned} \tag{10}$$

其中，

$$f_k = \begin{cases} 1 & if\ k = n \\ \prod_{j=k+1}^{n} d_j & if\ 1 \le k < n. \end{cases} \tag{11}$$

[16]Fortran 是少数几种使用列优先方式的高级语言之一。

下面给出 MultiArray 的 ADT 实现：

```
import random
from myarray import Array

class MultiArray:
    def __init__(self, *dimensions):
        assert len(dimensions) > 1, \
               'The Multi-Array must have 2 or more dimensions.'
        self.dimensions = dimensions
        size = 1
        for d in self.dimensions:
            assert d > 0, 'Dimensions must be > 0.'
            size = size*d

        self.elements = Array(size)
        self.factors = Array(len(self.dimensions))
        self.computeFactors()

    def numDims(self):
        return len(self.dimensions)

    def length(self, dim):
        assert 1 <= dim <= len(self.dimensions),\
               'Dimensions component out of range.'
        return self.dimensions[dim-1]

    def clear(self, value):
        self.elements.clear(value)

    def __getitem__(self, ndxTuple):
        assert len(ndxTuple) == self.numDims(),\
```

```
31                 'Invalid # of array subscripts.'
32         index = self.computeIndex(ndxTuple)
33         assert index is not None, 'Array subscript out of range.'
34         return self.elements[index]
35
36     def __setitem__(self, ndxTuple, value):
37         assert len(ndxTuple) == self.numDims(), \
38                'Invalid # of array subscripts.'
39         index = self.computeIndex(ndxTuple)
40         assert index is not None, 'Array subscript out of range.'
41         self.elements[index] = value
42
43     def computeIndex(self, ndxTuple):
44         offset = 0
45         for i in range(len(ndxTuple)):
46             if ndxTuple[i] < 0 or ndxTuple[i] >= self.dimensions[i]:
47                 return None
48             else:
49                 offset = offset + ndxTuple[i] * self.factors[i]
50         return offset
51
52     def computeFactors(self):
53         ndims = self.numDims()
54         self.factors[ndims-1] = 1
55         for i in range(ndims-1):
56             self.factors[i] = 1
57             for j in range(i+1, ndims):
58                 self.factors[i] *= self.dimensions[j]
59
60 def main():
61     ma = MultiArray(4, 3, 2)
```

```
62      for i in range(ma.length(1)):
63          for j in range(ma.length(2)):
64              for k in range(ma.length(3)):
65                  ma[i,j,k] = random.random()
66      for i in range(ma.length(1)):
67          for j in range(ma.length(2)):
68              for k in range(ma.length(3)):
69                  print(ma[i,j,k], end=' ')
70              print()
71          print()
72          print()
73      print(ma[2,2,1])
74
75  if __name__ == '__main__':
76      main()
77
```

某次运行的结果如下：

```
0.5503359424562829 0.2261963413392819
0.9258264068951316 0.5947682441582823
0.5180065980424153 0.30648129997278184


0.5102887146045798 0.14912238285281743
0.5111164419954669 0.5993335627982831
0.3616434195122785 0.3699968286717461


0.8195610432076301 0.6426249646459391
0.6806015500875922 0.3577610185961503
0.6160298352269856 0.15076511940860693
```

```
0.6173714632396361 0.2084537555781263
0.5223859878122135 0.5078123208681911
0.5281325308094426 0.02299140858776516


0.15076511940860693
```

2.8 Matrix

矩阵 (Matrix) 是一类非常重要的数学概念和工具，广泛应用于线性代数、计算机图形学及机器学习等领域中。它的 ADT 定义如下：

- Matrix(nrows, ncols)

  创建一个 nrows 行 ncols 列的矩阵，它的每个元素被初始化为 0；

- numRows()

  返回矩阵的行数；

- numCols()

  返回矩阵的列数；

- getitem(row, col)

  返回矩阵第 row 行第 col 列处的元素，row 和 col 必须在允许的范围内；

- setitem(row, col, scalar)

  设置矩阵第 row 行第 col 列处元素的值为 scalar，row 和 col 必须在允许的范围内；

- scaleBy(scalar)

  矩阵的每个元素分别乘以 scalar，矩阵被该操作修改；

- transpose()

  返回矩阵的转置；

- add(rhsMatrix)

  返回本矩阵与矩阵 rhsMatrix 的和，两个矩阵的维度必须相同；

- subtract(rhsMatrix) 返回本矩阵与矩阵 rhsMatrix 的差，两个矩阵的维度必须相同；

- multiply(rhsMatrix)

  返回本矩阵与矩阵 rhsMatrix 的乘积，两个矩阵的维度必须满足矩阵乘法规则；

利用二维数组来实现矩阵的 ADT，程序如下：

```
from myarray2d import Array2D

class Matrix:
    def __init__(self, numRows, numCols):
        self.theGrid = Array2D(numRows, numCols)
        self.theGrid.clear(0)

    def numRows(self):
        return self.theGrid.numRows()

    def numCols(self):
        return self.theGrid.numCols()

    def __getitem__(self, ndxTuple):
        return self.theGrid[ndxTuple[0], ndxTuple[1]]

    def __setitem__(self, ndxTuple, scalar):
        self.theGrid[ndxTuple[0], ndxTuple[1]] = scalar

    def scaleBy(self, scalar):
        for r in range(self.numRows()):
```

```
22              for c in range(self.numCols()):
23                  self[r, c] *= scalar
24
25      def transpose(self):
26          newMatrix = Matrix(self.numCols(), self.numRows())
27          for r in range(self.numRows()):
28              for c in range(self.numCols()):
29                  newMatrix[c, r] = self.theGrid[r, c]
30          return newMatrix
31
32      def __add__(self, rhsMatrix):
33          assert rhsMatrix.numRows() == self.numRows() and \
34                 rhsMatrix.numCols() == self.numCols(), \
35                 'Matrix sizes not compatible for the add operation.'
36
37          newMatrix = Matrix(self.numRows(), self.numCols())
38
39          for r in range(self.numRows()):
40              for c in range(self.numCols()):
41                  newMatrix[r, c] = self[r, c] + rhsMatrix[r, c]
42          return newMatrix
43
44      def __sub__(self, rhsMatrix):
45          assert rhsMatrix.numRows() == self.numRows() and \
46                 rhsMatrix.numCols() == self.numCols(), \
47                 'Matrix sizes not compatible for the subtract operation.'
48
49          newMatrix = Matrix(self.numRows(), self.numCols())
50
51          for r in range(self.numRows()):
52              for c in range(self.numCols()):
```

```
53                 newMatrix[r, c] = self[r, c] - rhsMatrix[r, c]
54         return newMatrix
55 
56     def __mul__(self, rhsMatrix):
57         assert rhsMatrix.numRows() == self.numCols(), \
58                'Matrix sizes not compatible for the multiply operation.'
59 
60         newMatrix = Matrix(self.numRows(), rhsMatrix.numCols())
61 
62         for r in range(self.numRows()):
63             for c in range(rhsMatrix.numCols()):
64                 for t in range(self.numCols()):
65                     newMatrix[r, c] += self[r, t] * rhsMatrix[t, c]
66         return newMatrix
67 
68     def print(self):
69         for r in range(self.numRows()):
70             for c in range(self.numCols()):
71                 print(self[r, c], end=' ')
72             print()
73 
74 def main():
75     A = Matrix(3, 2)
76     A[0, 0] = 6
77     A[0, 1] = 9
78     A[1, 0] = 7
79     A[1, 1] = 1
80     A[2, 0] = 8
81     A[2, 1] = 0
82     B = Matrix(3, 2)
83     B[0, 0] = 0
```

```
84      B[0, 1] = 1
85      B[1, 0] = 2
86      B[1, 1] = 3
87      B[2, 0] = 4
88      B[2, 1] = 5
89      (A+B).print()
90      (A-B).print()
91      (B*A.transpose()).print()
92
93  if __name__ == '__main__':
94      main()
95
```

2.9 Sparse Matrix

在稀疏矩阵 (Sparse Matrix) 中，包含了大量的 0 值元素。如果采用常规 Matrix 数据结构存储稀疏矩阵，那么就会浪费很多的空间，因为非零元素的个数 $k \ll m \times n$，其中 m 和 n 分别表示矩阵的行数与列数。

针对这个问题，可以使用 Python 中的列表对象来存储稀疏矩阵。它的 ADT 定义与常规 Matrix 一样，在此不再赘述。下面直接给出它的 ADT 实现：

```
1   class SparseMatrix:
2       def __init__(self, numRows, numCols):
3           self._numRows = numRows
4           self._numCols = numCols
5           self.elements = list()
6
7       def numRows(self):
8           return self._numRows
9
10      def numCols(self):
11          return self._numCols
```

```
12
13      def __getitem__(self, ndxTuple):
14          index = self.findPosition(ndxTuple[0], ndxTuple[1])
15          if index == None:
16              return 0.0
17          else:
18              return self.elements[index].value
19
20      def __setitem__(self, ndxTuple, value):
21          index = self.findPosition(ndxTuple[0], ndxTuple[1])
22          if index != None:
23              if value != 0.0:
24                  self.elements[index].value = value
25              else:
26                  self.elements.pop(index)
27          else:
28              if value != 0.0:
29                  element = SparseMatrixElement(\
30                                  ndxTuple[0], ndxTuple[1], value)
31                  self.elements.append(element)
32
33      def scaleBy(self, value):
34          if value == 0.0:
35              self.elements.clear()
36          for element in self.elements:
37              element.value *= value
38
39      def transpose(self):
40          newmatrix = SparseMatrix(self.numCols(), self.numRows())
41          for element in self.elements:
42              row, col, value = element.col, element.row, element.value
```

```
43              newelement = SparseMatrixElement(row, col, value)
44              newmatrix.elements.append(newelement)
45          return newmatrix
46
47      def __add__(self, rhsMatrix):
48          assert rhsMatrix.numRows() == self.numRows() and \
49                 rhsMatrix.numCols() == self.numCols(), \
50                 'Matrix sizes not compatible for the add operation.'
51
52          newMatrix = SparseMatrix(self.numRows(), self.numCols())
53
54          for r in range(self.numRows()):
55              for c in range(self.numCols()):
56                  newMatrix[r, c] = self[r, c] + rhsMatrix[r, c]
57          return newMatrix
58
59      def __sub__(self, rhsMatrix):
60          assert rhsMatrix.numRows() == self.numRows() and \
61                 rhsMatrix.numCols() == self.numCols(), \
62                 'Matrix sizes not compatible for the subtract operation.'
63
64          newMatrix = SparseMatrix(self.numRows(), self.numCols())
65
66          for r in range(self.numRows()):
67              for c in range(self.numCols()):
68                  newMatrix[r, c] = self[r, c] - rhsMatrix[r, c]
69          return newMatrix
70
71      def __mul__(self, rhsMatrix):
72          assert rhsMatrix.numRows() == self.numCols(), \
73                 'Matrix sizes not compatible for the multiply operation.'
```

```
74
75          newMatrix = SparseMatrix(self.numRows(), rhsMatrix.numCols())
76
77          for r in range(self.numRows()):
78              for c in range(rhsMatrix.numCols()):
79                  for t in range(self.numCols()):
80                      newMatrix[r, c] += self[r, t] * rhsMatrix[t, c]
81          return newMatrix
82
83      def findPosition(self, row, col):
84          for index, element in enumerate(self.elements):
85              if row == element.row and col == element.col:
86                  return index
87          return None
88
89      def print(self):
90          for r in range(self.numRows()):
91              for c in range(self.numCols()):
92                  print(self[r, c], end=' ')
93              print()
94
95  class SparseMatrixElement:
96      def __init__(self, row, col, value):
97          self.row = row
98          self.col = col
99          self.value = value
100
101 def main():
102     A = SparseMatrix(3, 2)
103     A[0, 0] = 6
104     A[1, 0] = 7
```

```
105     A[1, 1] = 1
106     B = SparseMatrix(3, 2)
107     B[0, 1] = 1
108     B[1, 0] = 2
109     B[2, 1] = 5
110     (A+B).print()
111     (A-B).print()
112     (B*A.transpose()).print()
113
114 if __name__ == '__main__':
115     main()
```

可以看出，SparseMatrix 与常规 Matrix 实现的主要区别在于 ___getitem___ 与 ___setitem___ 方法，其它方法的实现没有任何区别[17]。

2.10 Singly Linked List

在前文中，我们介绍了 Array 和 List 类型。它们有各自的特点。Array 类型操作简单，计算机硬件对其有着天然的支持；List 类型在 Array 类型的基础上进行了扩展，支持较多的操作，并且能够动态地扩展大小。在日常应用中，它们是使用率较高的 2 种数据类型。

然而，这 2 种数据类型存在一些缺点：

- 插入与删除元素的操作需要移动其它的元素；
- Array 类型不能扩展。List 虽然可以扩展，但是，它的初始分配空间是实际所需空间的 2 倍；并且，在扩展时，需要执行原始数据的复制工作。在数据量较大时，代价很高；
- Array 和 List 类型需要连续的内存空间。在某些情况下，特别是当数据量巨大时，该条件难以得到满足；

[17]实际上，本例提供的 SparseMatrix 没有对矩阵的 +、-、* 等操作进行优化。此部分的优化工作，将在后续版本中进行完善。在实际使用时，可以使用 scipy 库提供的稀疏矩阵模块。

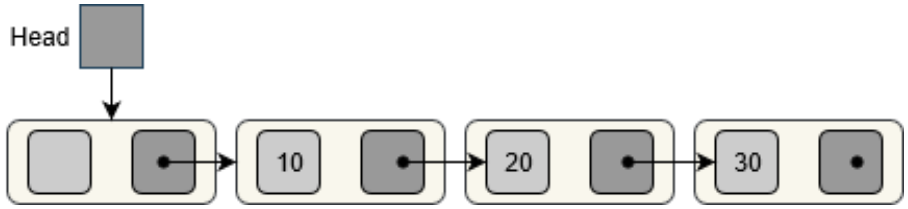

图 2-9: 简单的单链表示意图

本节将引入单链表 (Singly Linked List) 结构[18]，它不具有上述问题。但是，它的缺点在于，不能使用常量时间访问元素。在实际应用中，需要根据实际问题，权衡利弊，做出合理正确的选择。

单链表是一种最基本的链表结构，它的基本单元被称为节点 (Node)，用于存放数据元素，并且节点中也包含一个指向其它节点的链接 (Link)。

如图2-9所示，展示了一个简单的单链表。下面是它的程序实现：

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def main():
    head = ListNode(None)
    a = ListNode(10)
    b = ListNode(20)
    c = ListNode(30)
    head.next = a
    a.next = b
    b.next = c
    node = head.next
    while node:
        print(node.data)
        node = node.next
```

[18] 本讲义的后续版本将会讨论链表结构的其它各种变体，例如双链表、环形链表、多链接链表等。

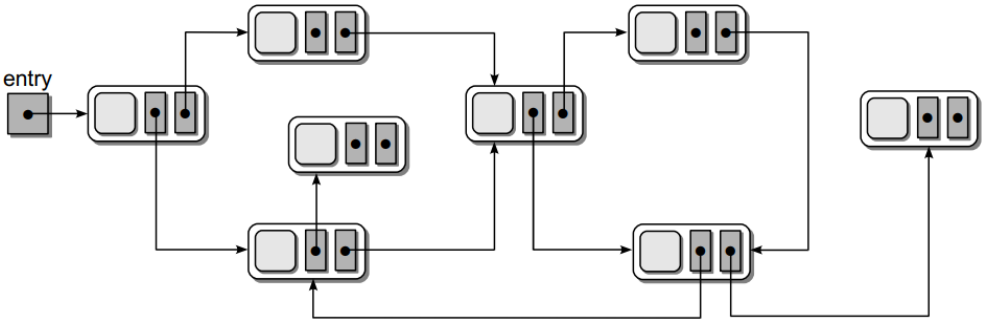

图 2-10: 一个较复杂的链表结构

```
18
19 if __name__ == '__main__':
20     main()
```

单链表拥有一个头节点 (Head Node) 和一个尾节点 (Tail Node)。单链表也允许为空，当头节点没有链接到其它任何节点时 (head.next=None)。

如果节点中包含多个链接，那么这些节点可以形成任意复杂的链接结构，如图2-10所示。树形结构 (Tree Structure) 是另一种链表结构，将在后面进行介绍。

下面给出单链表的 ADT 定义：

- SingleLinkedList()

  创建一个空的单链表；

- addAtFirst(item)

  在单链表的头部增加一个元素 item；

- addAtLast(item)

  在单链表的尾部增加一个元素 item；

- removeAtFirst()

  从单链表的头部删除一个元素，返回该元素；

- size()

  返回单链表的元素个数；

下面的程序给出了单链表的 ADT 实现：

```python
class SingleLinkedList:
    def __init__(self):
        self.head = ListNode(0)
        self.tail = self.head.next

    def addAtFirst(self, item):
        node = ListNode(item)
        node.next = self.head.next
        self.head.data += 1
        self.head.next = node
        if node.next == None:
            self.tail = node

    def addAtLast(self, item):
        if len(self) == 0:
            self.addAtFirst(item)
            return
        node = ListNode(item)
        self.tail.next = node
        self.tail = node
        self.head.data += 1

    def removeAtFirst(self):
        assert len(self) >= 1, 'Linked list empty.'
        node = self.head.next
        self.head.next = node.next
        self.head.data -= 1
        if len(self) == 0:
            self.tail = self.head.next

    def __len__(self):
```

```
32          return self.head.data
33
34      def print(self):
35          node = self.head.next
36          while node:
37              print(node.data)
38              node = node.next
39
40  class ListNode:
41      def __init__(self, data):
42          self.data = data
43          self.next = None
44
45  def main():
46      sll = SingleLinkedList()
47      sll.addAtLast(4)
48      sll.addAtFirst(1)
49      sll.addAtFirst(2)
50      sll.addAtLast(0)
51      sll.print()
52      sll.addAtLast(3)
53      sll.print()
54      sll.removeAtFirst()
55      sll.print()
56
57  if __name__ == '__main__':
58      main()
```

2.11 Set

集合 (Set) 是一种数据容器[19]，与数学上的集合概念一样，不要求数据元素是有序的，而只要求数据元素具有唯一性。另外，要求该容器提供各种基本的集合操作。

下面给出集合的 ADT：

- Set()

  创建一个空的集合对象；

- length()

  返回集合中元素的个数 (基数/cardinality)[20]；

- contains(element)

  判断元素 element 是否在集合中[21]；

- add(element)

  将元素 element 添加到集合中：如果该元素已经存在，不执行任何动作；否则，将该元素添加到集合中；

- remove(element)

  将元素 element 从集合中删除掉。如果该元素不存在，则引发一个异常；

- equals(otherset)

  判断本集合与 otherset 集合是否相同 (两个集合中的数据元素完全一致。易知，2 个空集合也是相同的)[22]；

- isSubsetOf(otherset)

  判断本集合是否是集合 otherset 的子集；

- union(otherset)

[19]Python 已经提供了集合类型，本节提供它的 ADT，探讨它的实现方式。
[20]通过定义成类的方法 ___len___(self) 形式，可以通过内建函数 len 调用。
[21]通过定义成类的方法 ___contains___(self) 形式，可以通过关键字 in 调用。
[22]通过定义成类的方法 ___eq___(self,otherset) 形式，可以通过 == 或!= 调用。

创建一个新集合，它是本集合与集合 otherset 的并集。原有的集合不会被修改；

- intersect(otherset)

  创建一个新集合，它是本集合与集合 otherset 的交集。原有的集合不会被修改；

- difference(otherset)

  创建一个新集合，它是本集合与集合 otherset 的差集。原有的集合不会被修改；

- iterator()

  创建并返回一个迭代器，用于遍历集合中的元素；

在实现集合 ADT 时，可以考虑使用数组、列表或字典类型。数组存在的问题是，它的大小是固定的；而字典存在的问题是，浪费一半的空间。显然，对于此次实现，列表是较合适的数据结构。但是，列表中的元素是允许重复的。因此，实现时，需要对这种情况进行特别的处理。

下面给出集合 ADT 的实现：

```
class Set:
    def __init__(self):
        self.theElements = list()

    def __len__(self):
        return len(self.theElements)

    def __contains__(self, element):
        return element in self.theElements

    def add(self, element):
        if element not in self:
            self.theElements.append(element)
```

```
14
15      def remove(self, element):
16          assert element in self, 'The element must be in the set.'
17          self.theElements.remove(element)
18
19      def __eq__(self, otherset):
20          if len(self) != len(otherset):
21              return False
22          else:
23              return self.isSubsetOf(otherset)
24
25      def isSubsetOf(self, otherset):
26          for element in self:
27              if element not in otherset:
28                  return False
29          return True
30
31      def union(self, otherset):
32          newset = Set()
33          newset.theElements.extend(self.theElements)
34          for element in otherset:
35              if element not in newset:
36                  newset.theElements.append(element)
37          return newset
38
39      def interset(self, otherset):
40          newset = Set()
41          for element in otherset:
42              if element in self:
43                  newset.theElements.append(element)
44          return newset
```

```
45
46      def difference(self, otherset):
47          newset = Set()
48          for element in self:
49              if element not in otherset:
50                  newset.theElements.append(element)
51          return newset
52
53      def __iter__(self):
54          return SetIterator(self.theElements)
55
56      def print(self):
57          for element in self:
58              print(element, end=' ')
59          print()
60
61  class SetIterator:
62      def __init__(self, theElements):
63          self.setRef = theElements
64          self.curNdx = 0
65
66      def __iter__(self):
67          return self
68
69      def __next__(self):
70          if self.curNdx < len(self.setRef):
71              entry = self.setRef[self.curNdx]
72              self.curNdx = self.curNdx + 1
73              return entry
74          else:
75              raise StopIteration
```

```
76
77  def main():
78      s = Set()
79      s.add('A1')
80      s.add('A2')
81      s.add('A1')
82      s.add('A3')
83      t = Set()
84      t.add('A1')
85      t.add('A2')
86      t.add('A4')
87      t.add('A5')
88      t.add('A3')
89      s.print()
90      s.union(t).print()
91      s.interset(t).print()
92      s.difference(t).print()
93      print(s.isSubsetOf(t))
94
95  if __name__ == '__main__':
96      main()
```

2.12 Map

映射 (Map) 或字典 (Dictionary) 是另一种数据容器，提供了一种映射能力：将 (唯一) 键 (Key) 映射到 (对应的) 值 (Value)，即建立 key-value 之间的对应关系。

Python 的字典数据类型提供了类似的功能，它使用了 hash 表技术。因此，需要为 key 对象定义 ___hash___ 方法，以生成 hash 编码。实际上，这限制了字典类型的使用范围。本节定义的 Map 数据类型放宽了 key 的限制——只要求 key 能够进行比较即可。对于使用哈希技术实现的 HashMap 数据类型，将在2.13中进行讨论。

下面给出 Map 的 ADT 定义[23]：

- Map()

  创建一个空的 Map 对象；

- length()

  返回 Map 容器中 key-value 对的数目；

- contains(key)

  判断 key 是否在 Map 容器内，返回：True，在；False，不在；

- add(key, value)

  向容器内添加 key-value 对：如果 key 已经在容器中，使用 value 替换原值，并返回 False；否则，将 key-value 加入到容器中，并返回 True；

- remove(key)

  删除容器内的 key-value 对。如果 key 不存在，则引发一个异常；

- valueOf(key)

  返回与 key 关联的 value。如果 key 不存在，则引发一个异常；

- iterator()

  创建并返回一个迭代器，用于遍历容器中的所有 key；

基于前述理由，本节考虑使用 List 来实现 Map 数据类型，存储示意图如图2-11所示。

下面给出 Map 的 ADT 实现：

```
1 class Map:
2     def __init__(self):
3         self.entryList = list()
4 
5     def __len__(self):
```

[23]某些 Map 的 ADT 方法，存在着对应的 Python 语句，例如，___contains___ 方法与 in 语句对应。这种情况在 Set 中提到过，此处不再赘述。

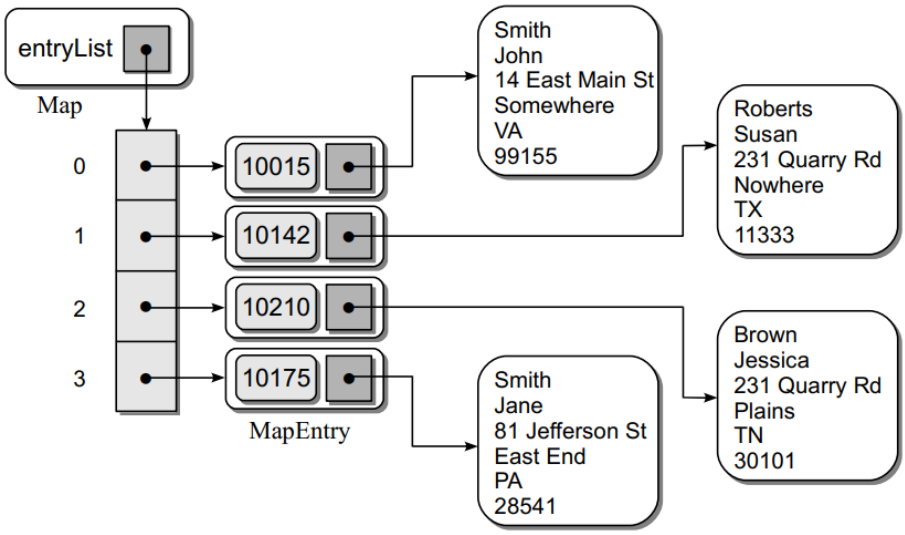

图 2-11: Map 的 List 实现示意图

```
6           return len(self.entryList)
7
8       def __contains__(self, key):
9           index = self.findPosition(key)
10          return index is not None
11
12      def add(self, key, value):
13          index = self.findPosition(key)
14          if index is not None:
15              self.entryList[index].value = value
16              return False
17          else:
18              entry = MapEntry(key, value)
19              self.entryList.append(entry)
20              return True
21
22      def valueOf(self, key):
23          index = self.findPosition(key)
24          assert index is not None, 'Invalid map key.'
```

```
25          return self.entryList[index].value
26
27      def remove(self, key):
28          index = self.findPosition(key)
29          assert index is not None, 'Invalid map key.'
30          self.entryList.pop(index)
31
32      def __iter__(self):
33          return MapIterator(self.entryList)
34
35      def findPosition(self, key):
36          for i in range(len(self)):
37              if self.entryList[i].key == key:
38                  return i
39          return None
40
41      def print(self):
42          for i in range(len(self)):
43              print('key=', self.entryList[i].key, \
44                    'value=', self.entryList[i].value)
45
46  class MapEntry:
47      def __init__(self, key, value):
48          self.key = key
49          self.value = value
50
51  class MapIterator:
52      def __init__(self, entryList):
53          self.mapRef = entryList
54          self.curNdx = 0
55
```

```
56      def __iter__(self):
57          return self
58
59      def __next__(self):
60          if self.curNdx < len(self.mapRef):
61              entry = self.mapRef[self.curNdx]
62              self.curNdx = self.curNdx + 1
63              return entry
64          else:
65              raise StopIteration
66
67  def main():
68      m = Map()
69      m.add(1, 'a')
70      m.add(2, 'b')
71      m.add(3, 'c')
72      m.add(4, 'd')
73      m.print()
74      print()
75      m.remove(3)
76      m.print()
77      print()
78      print(m.valueOf(4))
79
80  if __name__ == '__main__':
81      main()
```

2.13 Hash Table

在日常应用中，需要对特定的数据进行搜索。如果序列数据是无序的，那么可以执行线性顺序搜索，时间复杂度是 $O(n)$；而如果序列数据是有序的，则可以执行二分搜索，时间复杂度是 $O(logn)$。这 2 种搜索都是基于 key 值比较的方法。

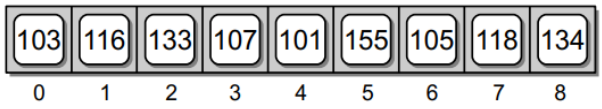

图 2-12: 数组中连续存放的序列数据

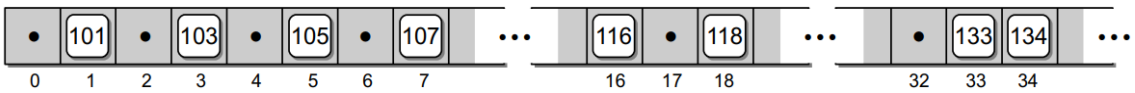

图 2-13: 数组中散列存放的序列数据

相关的研究已经表明，基于比较的搜索方法，它的最佳时间复杂度就是 $O(logn)$。

为了进一步提高搜索效率，必须考虑比较之外的搜索方法。本节考虑这样一种技术——哈希表 (Hash Table)。

2.13.1 Hashing 技术

哈希 (Hashing) 技术也被称为散列技术，它的作用是将 key 映射到 (存放数据的) 数组中的索引，从而为元素的存取提供了方便。在最佳情况下，元素的存取可以在常量时间内完成。

如图2-12所示，一个序列数据连续存放于数组中。对于此例，因为没有建立起 key 与元素索引之间的关系，也没有对数据进行排序，所以只能使用线性顺序方法进行搜索。

但是，如果将数组的容量 n 扩大，例如 $n = 100$，并且，不是依次连续存放元素，而是利用某个函数，例如 $key\%n$ 计算出元素存放的索引，该索引将元素“散列”到数组的各处，如图2-13所示。实际上，这种方法的效率更高——在存取时，依据同样的函数 (被称为散列或哈希函数)，计算出元素所在的位置 (索引)，就可以实现常量时间的存取操作了。当然，实际情况不可能如此完美，还需要解决面临的其它问题。

使用哈希技术建立起来的容器 (例如数组) 被称为哈希表 (Hash Table)，将 key 映射到索引的函数被称为哈希函数 (Hash Function) 或散列函数。

哈希表面临的主要问题是哈希冲突或散列冲突，即不同的 key 被散列到相同的索引。虽然选择不同的哈希函数可以减少冲突，但冲突不可避免。

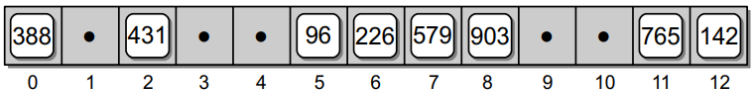

图 2-14: 序列数据的哈希表示意图

Linear Probing 线性探测 (Linear Probing) 技术是解决哈希冲突的方法之一。当冲突发生时，需要为冲突元素找到一个空单元。线性探测是一种最简单的方法，它从冲突发生的位置开始，逐单元向后进行探测，直到找到一个空单元，然后将该元素存入其中。

基本的线性探测方法使用如下公式进行探测：

$$probe = (home + i)\%M \tag{12}$$

其中，$i = 1, 2, ..., M-1$，$home$ 是哈希函数计算出的原始索引位置。

例如，对于数据序列 “765, 431, 96, 142, 579, 226, 903, 388”，假设需要建立长度为 $M = 13$ 的哈希表，使用的哈希函数是 $h(key) = key\%M$。

前 5 个元素的哈希值 (哈希函数计算出的索引值) 为：$h(765) = 11$、$h(431) = 2$、$h(96) = 5$、$h(142) = 12$、$h(579) = 7$，它们都是唯一的。但是，当 $key = 226$ 时，$h(226) = 5$，产生了哈希冲突。

线性探测方法简单地向后探测空单元，将 226 存入索引为 6 的空单元。随后，对于元素 903，$h(903) = 6$，该单元刚被占据，只能再次向后进行线性探测，并将 903 存入找到的第 1 个空单元。对于元素 388 的存放位置，依次类推。最后，建立的哈希表如图2-14所示。

Searching 搜索 key 的执行过程，与插入 key 的过程是相似的。首先，通过哈希函数计算索引，判断该索引处的元素是否与目标一致，如果一致，则停止搜索；否则，使用线性探测方法进行搜索：或者，找到目标；或者，碰到空单元；或者，整个哈希表都被搜索完毕。在后两种情况下，表明没有搜索到目标 key 值，key 不在哈希表中。

Deleting 在删除某个单元时，不能简单地将它置为空单元；否则，可能会破坏之前建立的 (因防止哈希冲突而建立的) 元素链，从而导致线性探测方法失效。

原因在于，该单元当初在进入哈希表时，假设它遇到了哈希冲突，那么，依据

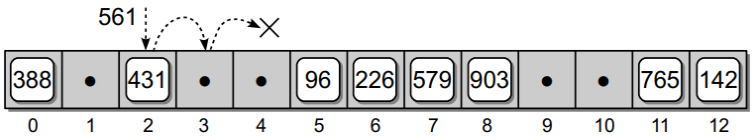

图 2-15: key 簇对冲突概率与探测长度的影响

冲突解决方法，它将被安置在冲突索引 (设该索引为 i) 后的某个位置 (设该索引为 j)。此后，在插入新元素时，$[i, j]$ 之间的索引位置，还可能会引发哈希冲突，假设 k 是因冲突而被安置的最后元素的索引。此时，在删除索引 j 处的单元时，如果简单地将其置为空单元，那么就会破坏 $[i, k]$ 之间的元素链，使线性探测方法失效。

正确的处理方法是，给待删除单元设置一个特殊的标志——既表明该单元已经被删除，又不至于让线性探测方法在该处终止探测。

Clustering 聚集 (Clustering) 是一种由于哈希冲突及解决冲突而导致的 key 相互关联与聚拢的现象，而这些 key 本来是互不相干的[24]。它们所形成的集合被称为 key 簇 (Cluster)。如果聚集现象发生在原始的哈希位置 (即使用哈希函数计算出来的索引位置) 附近，则称这种聚集为一次聚集 (Primary Clustering)。

随着簇的增长，冲突概率及线性探测的长度均会变大，如图2-15所示。索引为 4 的位置是空单元，它被新元素选中的概率为 $\frac{1}{13}$。索引为 9 的位置也是空单元，而它被新元素选中的概率不再是 $\frac{1}{13}$，而是 $\frac{5}{13}$。因为索引 $5-8$ 的单元形成了一个簇，所有经过哈希散列到此的元素都会选择索引为 9 的空单元。簇的形成，增大了潜在冲突的概率，同时也导致了探测长度的增长。

从上面分析可知，簇的长度越长，导致的问题越显著。为了减轻这 2 个问题，可以不采用逐步 (步长为 1) 线性的探测方法，而改用其它的探测方法，这样就可以有效地减少簇的长度。

Modified Linear Probing 对于前面的序列数据，使用基本的线性探测方法 (公式12) 将产生 6 次冲突，如图2-16所示。

[24]对于这些 key，可以使用哈希函数计算出索引。如果它们的索引互不相同，则称这些 key 是互不相干的。

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 8 => 9
h(142) => 12          h(388) => 11  => 12 => 0
```

图 2-16: 基本的线性探测方法示例

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 8
h(96)  => 5           h(903) => 6
h(142) => 12          h(388) => 11  => 1
```

图 2-17: 改进的线性探测方法示例

如果将步长修改为 $c > 1$[25]，那么探测公式如下：

$$probe = (home + i * c)\%M \tag{13}$$

设 $c = 3$，使用改进的线性探测方法仅仅产生 2 次冲突，如图2-17所示。使用改进的线性探测方法，建立的哈希表如图2-18所示。

比较图2-14和图2-18，可以发现，虽然 $c = 3$ 时的冲突次数减少了，但是最大簇的规模没有发生变化。

Quadratic Probing 改进的线性探测方法只是加大了探测步长，探测距离仍然是均匀的，因而最大簇的规模并没有发生变化。一种较好的改进方法是，使用非线性的平方探测 (Quadratic Probing) 方法：

$$probe = (home + i^2)\%M \tag{14}$$

该方法逐步增加 2 次探测之间的距离，可以减轻聚集现象的发生。虽然冲突次数增加到 7，如图2-19所示，但是簇的规模变小了，如图2-20所示。但是，平方探测方法不能确保探测到所有的单元。

[25]注意，选取的步长要确保一次完整的探测能够覆盖所有的单元。

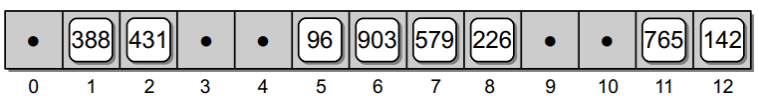

图 2-18: 改进的线性探测方法建立的哈希表

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 6
h(96)  => 5           h(903) => 6   => 7  => 10
h(142) => 12          h(388) => 11  => 12 => 2  => 7 => 1
```

图 2-19: 平方探测方法示例

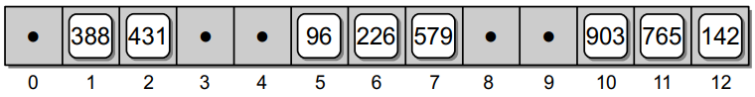

图 2-20: 平方探测方法建立的哈希表

Double Hashing 上述各种探测方法都存在一个问题——二次聚集 (Secondary Clustering)，指的是多个 key 映射到同一个位置后引起的探测序列相同的现象。引起这种现象的根源在于探测公式中的 $home$，对于这些 key 而言，$home$ 是相同的，它会产生相同的探测序列。

一种较好的解决方法是，引入第 2 个哈希函数，这被称为双哈希 (Double Hashing) 技术。当冲突发生时，第 2 个哈希函数将产生一个依赖于 key 的步长：

$$probe = (home + i * hp(key))\%M \tag{15}$$

虽然对于同一个 key 来讲，$hp(key)$ 是常量，但是对于映射到同一个位置的不同 key 而言，$hp(key)$ 将是不同的。注意，第 2 级哈希函数 $hp(key)$ 的选取要与主要哈希函数不同，以确保 $hp(key)$ 能够发挥作用。$hp(key)$ 产生的数值 c 应该满足 $0 < c < M$。一个简单的函数是：

$$hp(key) = 1 + key\%P \tag{16}$$

其中，P 是小于 M 的某个常量。

例如，对于前面的序列数据，定义的第 2 个哈希函数如下：

$$hp(key) = 1 + key\%8 \tag{17}$$

产生的结果如图2-21所示，该方法仅引起了 2 次哈希冲突。产生的哈希表如图2-22所示。双哈希方法能够同时减少一次聚集与二次聚集现象，是最常用的探测方法之一。为了确保哈希表的每个单元都能够被探测到，哈希表的大小应该是一个质数。

```
h(765) => 11          h(579) => 7
h(431) => 2           h(226) => 5   => 8
h(96)  => 5           h(903) => 6
h(142) => 12          h(388) => 11  => 3
```

图 2-21: 双哈希方法示例

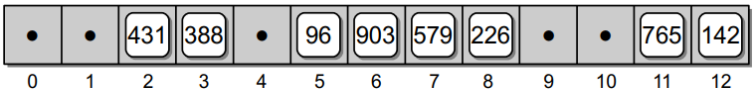

图 2-22: 双哈希方法建立的哈希表

Rehashing 在大多数情况下，哈希表的大小预先是无法知道的，而且也不一定是固定的，这就需要程序能够动态地扩展哈希表。这种情形在其它的数据类型中也同样存在，例如，C++ 中的 vector 类型与 Python 中的 list 类型都支持动态扩展空间。

但是，在哈希表中，元素的位置取决于哈希函数与哈希表的大小。因此，如果哈希表的大小发生了变化，那么就不能简单地将原哈希表中的数据复制到新哈希表中，而必须要再次应用哈希技术计算出元素的新位置，然后将元素放入，这被称为再哈希 (Rehashing)。

随着哈希表中元素的增多，发生哈希冲突的概率也将增大。这意味着，不能等到哈希表完全或几乎满的情况下，再进行扩展。那么，在什么时机扩充哈希表比较合适呢？

我们可以使用负荷因子 (Load Factor)——它是已用单元个数与整个单元数的比值——来确定哈希表的扩展时机。理论分析已经表明，负荷因子在 $\frac{1}{2} - \frac{2}{3}$ 时，哈希表的主要方法 (搜索) 的平均复杂度取得最佳值。通常情况下，取负荷因子为 $\frac{2}{3}$。当负荷因子达到该值时，就需要对哈希表进行扩展。

关于哈希表初始容量的确定，常规的做法是，取实际所需容量 m 的 2 倍，即 $2m$；并且，探测方法要求容量大小为质数。因此，还需要计算出比 $2m$ 大的第 1 个质数，将它作为哈希表的初始容量。在某些情况下，也可以将 $2m+1$ 作为哈希表的初始容量。

Efficiency Analysis 哈希表的操作效率取决于哈希函数、哈希表的容量及解决冲突的探测方法。

哈希表的主要操作包括：搜索、插入、删除。其中，后 2 个操作要依赖于搜索操作，无论是将新元素插入哈希表，还是从哈希表中删除已有元素，都需要搜索定位到某个空单元或已有单元。一旦定位成功，只需要常量时间执行插入与删除操作。

下面讨论一下搜索操作的时间复杂度。假设哈希表的容量为 m，当前的元素个数为 n。在最好情况下，元素经过哈希函数计算出索引位置，就直接定位成功，没有冲突发生，时间复杂度为 $O(1)$。在最坏情况下，需要探测整个哈希表，时间复杂度为 $O(m)$。乍一看，与线性搜索相比，哈希表好像没有什么优势。

但是，实际上，在平均情况下，哈希表有很大的优势。假设元素均匀地分布于整个哈希表中，那么，在平均情况下，搜索操作的时间复杂度将依赖于平均探测的长度，而平均探测长度又依赖于负荷因子。

设负荷因子为 $\alpha = \frac{n}{m} < 1$，使用线性探测方法，对于成功的搜索操作，平均的比较次数为[26]：

$$\frac{1}{2}[1 + \frac{1}{(1-\alpha)^2}] \tag{18}$$

而对于不成功的搜索操作，平均的比较次数为：

$$\frac{1}{2}[1 + \frac{1}{1-\alpha}] \tag{19}$$

如果使用平方探测或双哈希方法，对于成功的搜索操作，平均的比较次数为：

$$\frac{-log(1-\alpha)}{\alpha} \tag{20}$$

对于不成功的搜索操作，平均的比较次数为：

$$\frac{1}{1-\alpha} \tag{21}$$

为了建立直观认识，图2-23列出了线性探测与平方探测的平均比较次数。如图所示，当负荷因子超过 $\frac{2}{3}$ 时，平均比较次数变得很大，尤其是不成功的搜索。另外，平方探测与双哈希方法允许使用非常高的负荷因子。综合以上分析，当哈希表的负荷因子在 $\frac{1}{2} - \frac{2}{3}$ 时，搜索操作的平均时间复杂度为 $O(1)$。与线性搜索和二分搜索相比，哈希表提供了一种非常有效的解决方案。

[26]关于成功与不成功搜索的平均比较次数的详细分析，请参考 Donald E. Knuth 的《The Art of Computer Programming》系列丛书。

Load Factor	0.25	0.5	0.67	0.8	0.99
Successful search:					
Linear Probe	1.17	1.50	2.02	3.00	50.50
Quadratic Probe	1.66	2.00	2.39	2.90	6.71
Unsuccessful search:					
Linear Probe	1.39	2.50	5.09	13.00	5000.50
Quadratic Probe	1.33	2.00	3.03	5.00	100.00

图 2-23: 不同探测方法的平均比较次数

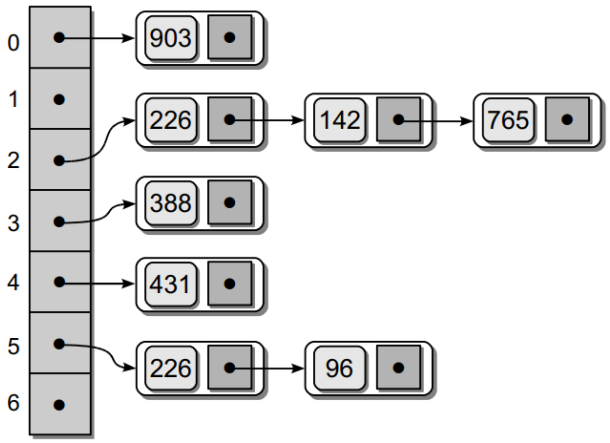

图 2-24: 使用分离链接技术建立的哈希表

2.13.2 分离链接技术

分离链接 (Separate Chaining) 是另一种处理哈希冲突的方法——不是使用探测技术在其它的单元中安置发生冲突的元素，而是将所有因映射到同一单元而发生冲突的元素安置到同一个链表 (Linked List 或 Chain) 中。如图2-24所示，展示了使用分离链接技术建立的哈希表。

在为所有冲突元素建立了单独的链接表之后，元素的搜索、插入与删除操作变得非常简单。其核心的操作，依然是搜索操作：首先利用哈希函数定位链接表引用所在的单元位置，然后顺着链接表，一直往下搜索，直到找到目标，或者没有找到目标，再执行相关操作。

使用分离链接的哈希技术，也被称为开放哈希 (Open Hashing)。相应地，前面的哈希技术，也被称为封闭哈希 (Closed Hashing)。此处的“开放”与“封闭”指

的是元素是存储在主哈希表中 (Closed)，还是存储在单独的链接表中 (Open)。

需要注意的是，还有 2 个经常使用的相关术语：开放寻址 (Open Addressing) 和封闭寻址 (Closed Addressing)。此处的“开放”与“封闭”指是否使用探测技术：如果使用，即为开放寻址；否则，即为封闭寻址。显然，开放哈希使用封闭寻址方式，封闭哈希使用开放寻址方式。

在分离链接的哈希表中，尽管哈希表容量的作用不那么突出，但它仍然是一个影响效率的重要因素——如果容量过小，独立的链接表可能就会太大，这会影响哈希表的整体性能。

下面讨论一下分离链接哈希表的时间复杂度。在最坏情况下，所有的元素集中在一条链接表中，其时间复杂度与线性搜索的一样，即 $O(n)$。然而，与封闭哈希一样，分离链接哈希表的平均时间复杂度也是非常好的。在平均情况下，n 个元素均匀地分布于 m 个链接表中 (主哈希表有 m 个单元)，每个链接表的长度为 $\frac{n}{m}$，它刚好是哈希表的负荷因子 α。对于成功的搜索，比较的平均次数为：

$$1 + \frac{\alpha}{2} \tag{22}$$

对于不成功的搜索，比较的次数为：

$$1 + \alpha \tag{23}$$

与封闭哈希相比，分离链接哈希表允许的负荷因子要大一些。当 $\alpha < 2$ 时，对于成功的搜索，它的平均比较次数小于 2；对于不成功的搜索，它的平均比较次数小于 3。而在相同的负荷因子下，分离链接哈希表也要优于封闭哈希方法。但是，分离链接哈希表的问题在于，它需要额外的空间存储节点的链接信息。

2.13.3 Hash 函数

哈希表的效率在很大程度上依赖于哈希函数 (Hash Function)。“完美”的哈希函数能够将每个元素散列到不同的单元中。在实际应用中，这几乎是不可能的。但是，我们可以设计出相对好的哈希函数，它尽可能地将每个元素均匀地散列到不同的单元中。下面给出哈希函数的选取准则：

- 计算简单；
- 计算结果总是确定的；

- 如果 key 由多个部分组成，每个部分都对计算结果起作用；
- 哈希表的容量应该是一个质数，这有利于元素的分布及减少冲突；

整型 key 值是哈希函数的完美映射对象。但是，对于其它类型的 key 值，可以先将它们转换到整型，再利用哈希函数完成映射。下面将给出几种常用的哈希函数。

Division 除法 (Division) 是一种最简单的哈希映射函数，采用除数取余法：

$$h(key) = key\%M \tag{24}$$

它的优点在于，计算的结果总是落在有效的索引范围 $[0, M-1]$ 中，M 表示哈希表的容量。

Truncation 截断 (Truncation) 方法，从大整型 key 值中取出某些位组成单元索引，并且要确保该索引在合法有效的范围内。例如，假设哈希表的容量为 10000，$key = 4527109243$，那么可以从 key 中取出 4 位数字 (随机或指定) 组成索引，例如 793。

Folding 折叠 (Folding) 方法，将 key 值拆分成几个部分，然后将各部分相加或相乘，再将所得结果进行除数取余或截断以获得最终的索引。

Zobrist Hashing Zobrist 哈希[27]是一种用于棋类游戏 (例如国际象棋和围棋等) 的特殊哈希函数，它以发明者 Albert Lindsey Zobrist 的名字命名。

以 19×19 围棋棋盘为例。首先，随机生成 3 个 19×19 的整数矩阵，它们分别表示 BLACK、WHITE、EMPTY 方的随机矩阵。然后，计算空棋盘的哈希编码 $HASH$：遍历 EMPTY 矩阵，将所有单元值进行 XOR(异或) 计算。现假设 BLACK 方在 (i, j) 处落子，那么，此时的哈希编码被更新为：$HASH = HASH\ XOR\ EMPTY[i, j]\ XOR\ BLACK[i, j]$。第 1 个 XOR，去掉了 $EMPTY[i, j]$ 的值；第 2 个 XOR，考虑了 $BLACK[i, j]$ 的值。此后，棋局的哈希编码 $HASH$ 也是按类似方式进行更新。Zobrist 哈希编码的优点在于，每次无需重复遍历整个矩阵计算棋局的编码，而只需以增量的方式更新编码即可。

[27]详见Wikipedia: Zobrist hashing。

Hashing Strings 为了应用上述哈希函数，字符串 key 必须以某种方式转换到整型数值。有多种方式可以进行转换。例如，可以将字符串中每个字符的 ASCII 值进行相加。但是，这种方式只适用于容量较小的哈希表。对于较大容量的哈希表，这种方式没有办法直接将较短字符串的 key 值映射到具有较大索引的单元 (只能通过探测的方式)。

另一种转换方式，具有一定的通用性，它采用如下多项式：

$$s_0 a^{n-1} + s_1 a^{n-2} + ... + s_{n-2} a + s_{n-1} \tag{25}$$

其中，a 是非零常量，s_i 是字符串的第 i 个字符，n 是字符串的长度。例如，如果 $a = 27$，“hashing” 的转换结果将是 41746817200。

2.13.4 HashMap 的 ADT 及其实现

哈希表的最常见应用是实现 Map 数据类型。实际上，Python 的字典 (dict) 是一种封闭哈希表。

在2.12中，我们已经讨论了常规技术实现的 Map 数据类型。在本节中，我们将讨论使用哈希技术实现的 HashMap 数据类型。我们的实现方法非常类似于 Python 中字典的实现方法：建立封闭哈希表，使用双哈希探测方法。

HashMap 的 ADT 定义与 Map 的一样，在此不再赘述。下面直接给出 HashMap 的实现程序：

```
1  from myarray import Array
2
3  class MapEntry:
4      def __init__(self, key, value):
5          self.key = key
6          self.value = value
7
8  class HashMap:
9      UNUSED = None
10     EMPTY = MapEntry(None, None)
11
12     def __init__(self):
```

```
13          self.table = Array(47)
14          self.count = 0
15          self.maxCount = len(self.table) - len(self.table)//3
16
17      def __len__(self):
18          return self.count
19
20      def __contains__(self, key):
21          slot = self.findSlot(key, False)
22          return slot is not None
23
24      def add(self, key, value):
25          if key in self:
26              slot = self.findSlot(key, False)
27              assert slot is not None, 'Invalid map key.'
28              self.table[slot].value = value
29              return False
30          else:
31              slot = self.findSlot(key, True)
32              assert slot is not None, 'Not enough memory.'
33              self.table[slot] = MapEntry(key, value)
34              self.count += 1
35              if self.count == self.maxCount:
36                  self.rehash()
37              return True
38
39      def valueOf(self, key):
40          slot = self.findSlot(key, False)
41          assert slot is not None, 'Invalid map key.'
42          return self.table[slot].value
43
```

```
44      def remove(self, key):
45          slot = self.findSlot(key, False)
46          assert slot is not None, 'Invalid map key.'
47          self.table[slot] = HashMap.EMPTY
48          self.count -= 1
49
50      def __iter__(self):
51          return HashMapIterator(self.table)
52
53      def findSlot(self, key, forInsert):
54          slot = self.hash1(key)
55          step = self.hash2(key)
56
57          M = len(self.table)
58          refcount = 0
59          if forInsert:
60              while self.table[slot] is not HashMap.UNUSED and \
61                    self.table[slot] is not HashMap.EMPTY and \
62                    refcount <= self.count:
63                  slot = (slot + step) % M
64                  refcount += 1
65              if refcount > self.count:
66                  return None
67              else:
68                  return slot
69          else:
70              while self.table[slot] is not HashMap.UNUSED and \
71                    refcount <= self.count:
72                  if self.table[slot] is not HashMap.EMPTY and \
73                     self.table[slot].key == key:
74                      return slot
```

```
75                  slot = (slot + step) % M
76                  refcount += 1
77              return None
78
79      def rehash(self):
80          origTable = self.table
81          newSize = len(self.table)*2 + 1
82          self.table = Array(newSize)
83
84          self.count = 0
85          self.maxCount = newSize - newSize//3
86
87          for entry in origTable:
88              if entry is not HashMap.UNUSED and entry is not HashMap.EMPTY:
89                  slot = self.findSlot(entry.key, True)
90                  assert slot is not None, 'Not enough memory.'
91                  self.table[slot] = entry
92                  self.count += 1
93
94      def hash1(self, key):
95          return abs(hash(key)) % len(self.table)
96
97      def hash2(self, key):
98          return 1 + abs(hash(key)) % (len(self.table) - 2)
99
100     def print(self):
101         for entry in self:
102             if entry is not HashMap.UNUSED and \
103                entry is not HashMap.EMPTY:
104                 print('Key=', entry.key, 'Value=', entry.value)
105
```

```
106 class HashMapIterator:
107     def __init__(self, table):
108         self.mapRef = table
109         self.curNdx = 0
110 
111     def __iter__(self):
112         return self
113 
114     def __next__(self):
115         if self.curNdx < len(self.mapRef):
116             entry = self.mapRef[self.curNdx]
117             self.curNdx = self.curNdx + 1
118             return entry
119         else:
120             raise StopIteration
121 
122 def main():
123     hm = HashMap()
124     for n in range(26):
125         hm.add(chr(ord('A')+n), n)
126     hm.print()
127     print(len(hm))
128     for n in range(0, 26, 2):
129         hm.remove(chr(ord('A')+n))
130     hm.print()
131     print(len(hm))
132 
133 if __name__ == '__main__':
134     main()
```

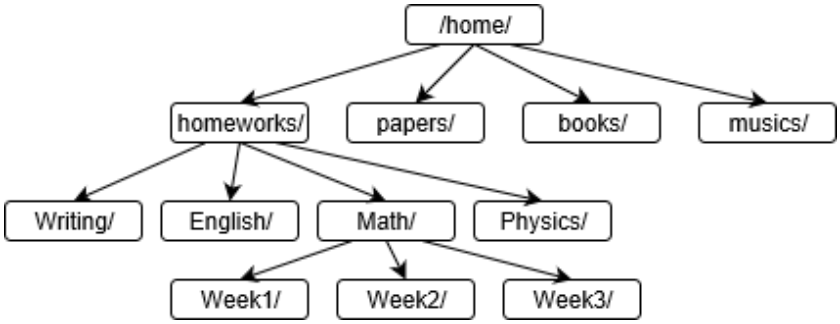

图 2-25: 一棵倒长的“树”

2.14 Binary Tree

2.14.1 相关概念

前面介绍的许多数据类型属于线性类型，例如 Array、List、Stack、Queue 等，它们的元素之间具有前后关系。本节要介绍的二叉树 (Binary Tree) 属于非线性类型，它们的元素之间具有层次关系。二叉树和树类型，被广泛应用于计算机科学、计算机图形学、人工智能与机器学习等领域，是一类非常重要的数据类型。

一般情况下，树被绘制成一棵倒长的自然树形式，如图2-25所示。

树由节点 (Node) 和边 (Edge) 组成，形成一个层次结构。除了最上层节点 (根节点，Root Node) 之外，每个节点有 1 个父节点 (Parent Node)，有 0 个或多个孩子节点 (Child Node)。例如，图2-25中，“/home/” 是根节点，“Math/” 的父节点是 “homeworks/”，它的孩子节点有 “Week1/”、“Week2/” 和 “Week3/”。

正式地，定义树 T 为一个节点集合，它满足下面的条件：

- 如果 T 为空，那么它没有任何节点；
- 如果 T 非空，那么它有一个根节点，根节点没有父节点；
- 如果 T 非空，且根节点有 0 个或多个孩子节点与之相连，那么每个孩子节点 (如果存在) 是其相应子树的根；

根的定义方式有多种，上面是根的递归定义。

除了根节点、父节点、孩子节点、子树等概念之外，还有外部节点 (叶节点) 与内部节点概念。叶节点指的是没有孩子节点的节点，内部节点指的是至少有一个孩子节点的节点。

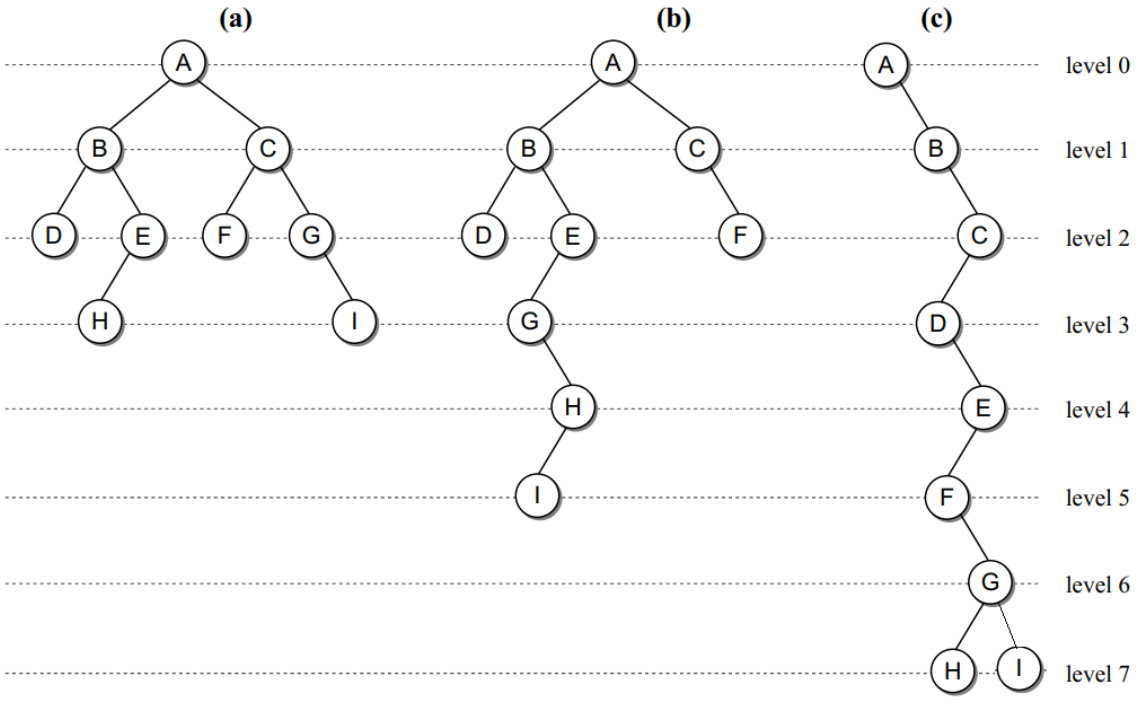

图 2-26: 不同形状的二叉树

此外，除非特别说明，总是从生物学的角度来命名树中节点。例如，兄弟节点指的是有共同直接父节点的那些节点；祖先节点指的是当前节点的父节点、父节点的父节点、...、根节点的那些节点——从当前节点开始，沿着父节点-父节点的父节点-...-根节点这条路径一直向上回溯所碰到的每一个节点。

二叉树 (Binary Tree) 是树的一种，它的每个节点至多有 2 个孩子节点，或者，它的根节点至多有 2 棵子树。

二叉树的形状多种多样。即使节点的个数相同，节点的父子层次关系不同，形状也大不相同，如图2-26所示。

下面给出几个常用术语：

- Level/Depth

  节点的层次 (Level) 或深度 (Depth) 定义了节点与根节点之间的远近关系。根节点的层次总是 0，它的孩子节点的层次总是 1；其它节点的层次，依此类推。例如，图2-26(a) 中，G 节点的层次/深度为 2；(b) 中，G 节点的层次/深度为 3；(c) 中，G 节点的层次/深度为 6；

- Height

  树的高度 (Height) 定义为：$max(Level(Node_i)) + 1$。例如，图2-26(a) 中，树的高度为 4；(b) 中，树的高度为 6；(c) 中，树的高度为 8；

- Width

  树的宽度 (Width) 定义为各层中最大的节点个数。例如，图2-26(a) 中，树的宽度为 4；(b) 中，树的宽度为 3；(c) 中，树的宽度为 2；

- Size

  树的大小 (Size) 定义为树中节点的个数。例如，图2-26(a)、(b)、(c) 中，树的大小都是 9。

  假设树的大小为 n，那么树的最大高度和最小高度分别是多少呢？当树中每层只有一个节点时，树具有最大高度 n。当树中每一层具有最大可能个数的节点时，例如，第 0 层，1 个节点；第 1 层，2 个节点；第 2 层，4 个节点；...，即可能除了最后一层外，其余各层皆“满员”，那么，树具有最小高度 $\lfloor log_2^n \rfloor + 1$；

在执行与树类型相关的任务中，树的高度会对算法的性能产生非常重要的影响。实际上，一些算法需要特殊结构的树。下面给出几种二叉树的定义[28]：

- 满二叉树 (Full Binary Tree)

  满二叉树的每个内部节点 (非叶子节点) 都有 2 个孩子节点[29]。满二叉树的形状有多种，如图2-27所示；

- 完美二叉树 (Perfect Binary Tree)

  完美二叉树是一种特殊的满二叉树，它的所有叶节点都在同一层，或者说，完美二叉树中每一层的节点数都达到了该层的最大容量，如图2-28所示；

- 完全二叉树 (Complete Binary Tree)

  完全二叉树被定义为：除最后一层外，其余各层节点形成的二叉树是一棵完美二叉树；在最后一层，节点按从左到右的顺序排列，中间没有跳跃。图2-29展示了一棵完全二叉树；

[28]不同的读物，概念的定义可能不一样，阅读时请注意区分。

[29]在有些书籍中，将满二叉树定义成“每层均包含最多节点个数的二叉树”。

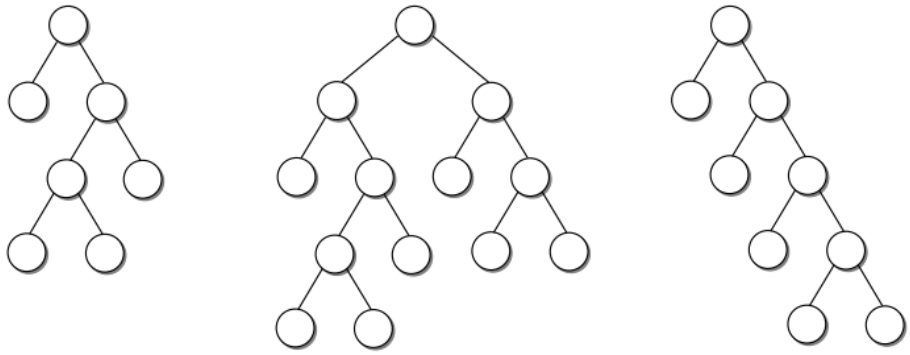

图 2-27: 不同形状的满二叉树

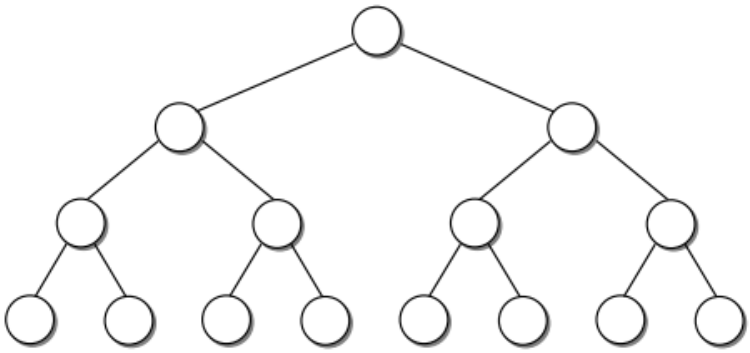

图 2-28: 完美二叉树示例

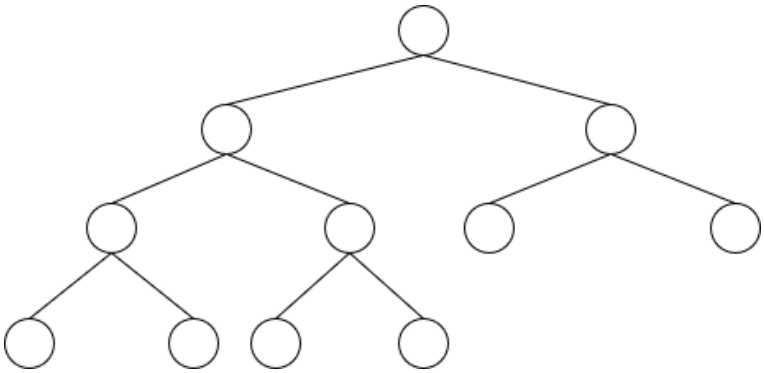

图 2-29: 完全二叉树示例

2.14.2 二叉树的 ADT

在二叉树的 ADT 中，有一些操作是依赖于具体应用的，仅当涉及到具体实例时，才给出相应的 ADT 描述。此处，先给出二叉树的基本 ADT 定义：

- BinaryTree(data)

  创建一棵没有左右子树的二叉树，节点值为 data；

- set_child(left, right)

  设置左右子树；

- is_empty()

  判断二叉树是否为空；

- length()

  返回二叉树中节点的个数；

- data()

  返回二叉树中根节点的数据；

- left()

  返回二叉树的左子树；

- right()

  返回二叉树的右子树；

- traversal()

  遍历二叉树中各节点；

2.14.3 二叉树的遍历

二叉树的遍历是一个重要的问题。在线性结构中，例如链接表，遍历元素是相当容易的——从头节点开始，顺着链接通过每一个节点，一直到最后一个节点。

但是，在二叉树中，节点之间不再是线性组织在一起的，而是呈现出“分叉”结构。正是这种分叉结构，导致从根节点到任一节点的访问路径不再是单一的。那

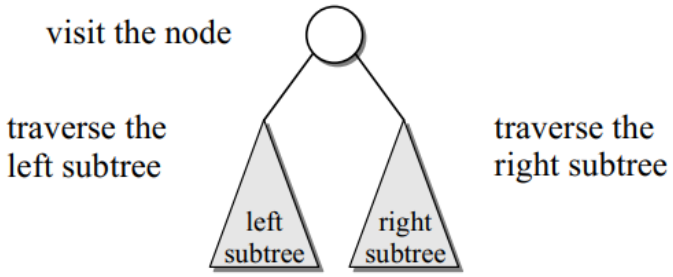

图 2-30: 二叉树的递归遍历

么，如何找到一种合理的遍历方式，从根节点开始，一次访问一个元素，且每个元素只被访问一次呢？

如果简单地从根节点开始，顺着一条路径往下访问，一直到叶节点为止，那么这种方式仅仅只能访问到该条路径上的节点，而不能遍历树中的每个节点。

然而，如果从递归的角度看待二叉树，就会找到一种非常优雅的解决方案，如图2-30所示。从本质上讲，访问一棵二叉树与访问它的左右子树没有什么区别，只是节点规模变小了——这是典型的递归问题。因此，递归算法是遍历二叉树的有力武器。

依据根节点与左右子树的不同访问次序，可以将二叉树的递归遍历算法分成三类：Preorder(先根)、Inorder(中根)、Postorder(后根)。

Preorder Traversal 该遍历方式首先访问根节点，然后分别递归遍历左子树和右子树。下面是 Preorder 的遍历算法：

```
def preorder(tree):
    if tree is not None:
        visit(tree.data)
        preorder(tree.left)
        preorder(tree.right)
```

如图2-31所示，虚线展示了 Preorder 遍历节点的过程。节点的 Preorder 遍历顺序为：A、B、D、E、H、C、F、G、I、J。

Inorder Traversal 该遍历方式首先递归遍历左子树，然后访问根节点，最后递归遍历右子树。下面是 Inorder 的遍历算法：

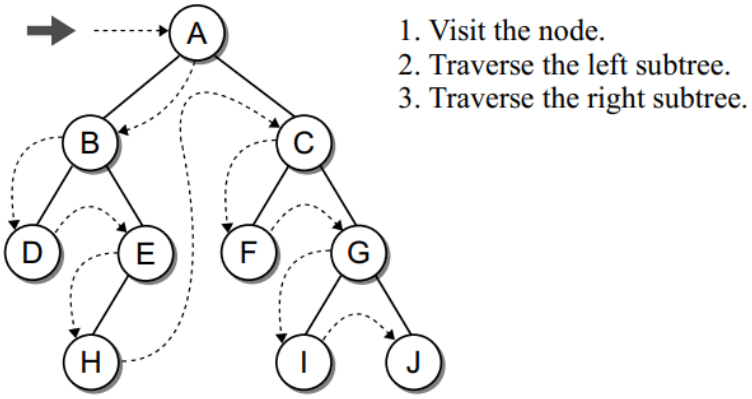

图 2-31: 二叉树的 Preorder 遍历

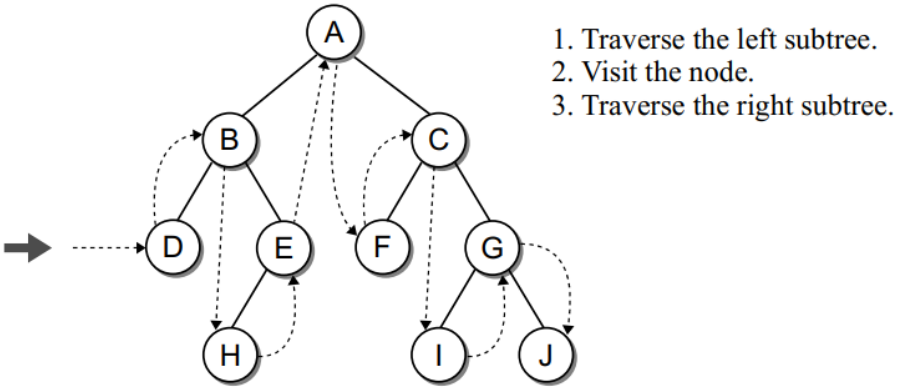

图 2-32: 二叉树的 Inorder 遍历

```
def Inorder(tree):
if tree is not None:
    Inorder(tree.left)
    visit(tree.data)
    Inorder(tree.right)
```

如图2-32所示，虚线展示了 Inorder 遍历节点的过程。节点的 Inorder 遍历顺序为：D、B、H、E、A、F、C、I、G、J。

Postorder Traversal 该遍历方式首先递归遍历左子树，然后递归遍历右子树，最后访问根节点。下面是 Postorder 的遍历算法：

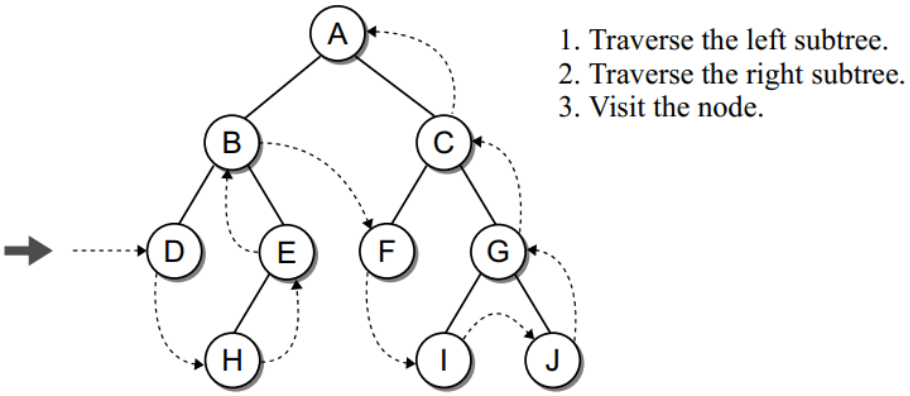

图 2-33: 二叉树的 Postorder 遍历

```
def Postorder(tree):
if tree is not None:
    Postorder(tree.left)
    Postorder(tree.right)
    visit(tree.data)
```

如图2-33所示，虚线展示了 Postorder 遍历节点的过程。节点的 Postorder 遍历顺序为：D、H、E、B、F、I、J、G、C、A。

Breadth-First Traversal 前面讨论的 Preorder、Inorder、Postorder 遍历算法均属于深度优先遍历 (Depth-First Traversal)，还有一种宽度优先遍历 (Breadth-First Traversal)，如图2-34所示。对于此例，它的访问顺序是：A、B、C、D、E、F、G、H、I、J。

在实现宽度优先遍历算法时，不能再使用递归算法，需要设计新的方法。例如，可以考虑使用队列 (Queue) 来存放待访问节点：访问根节点 A 时，将它的 2 个子节点 B 和 C 放入队尾；访问节点 B 时，再将它的 2 个子节点 D 和 E 放入队尾；其余的节点，依此类推。该算法的特点是，每次从队头取一个节点访问，并将它的子节点放入队尾。

下面给出宽度优先遍历的算法：

```
def BFT(tree):
    q = Queue()
    q.enqueue(tree)
```

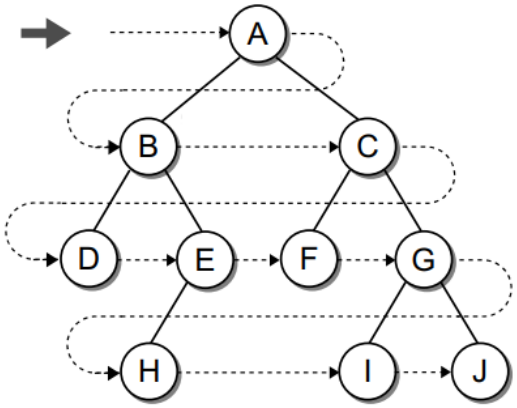

图 2-34: 二叉树的宽度优先遍历

```
    while not q.is_empty():
        node = q.dequeue()
        visit(node)
        if node.left is not None:
            q.enqueue(node.left)
        if node.right is not None:
            q.enqueue(node.right)
```

2.14.4 实现

下面给出二叉树的 ADT 实现:

```
1  class BinaryTree:
2      def __init__(self, data):
3          self.root = data
4          self.left = None
5          self.right = None
6          if data is None:
7              self.count = 0
8          else:
9              self.count = 1
10
11     def is_empty(self):
```

```
12          return self.root is None
13
14      def __len__(self):
15          return self.count
16
17      def data(self):
18          return self.root
19
20      def left(self):
21          return self.left
22
23      def right(self):
24          return self.right
25
26      def set_child(self, left, right):
27          self.left = left
28          self.right = right
29          if left is not None:
30              lcount = len(left)
31          else:
32              lcount = 0
33          if right is not None:
34              rcount = len(right)
35          else:
36              rcount = 0
37          self.count = lcount + rcount + 1
38
39      def traversal_preorder(self):
40          self.preorder(self)
41          print()
42
```

```
43      def traversal_inorder(self):
44          self.inorder(self)
45          print()
46
47      def traversal_postorder(self):
48          self.postorder(self)
49          print()
50
51      def preorder(self, tree):
52          if tree is not None:
53              print(tree.root, end=' ')
54              self.preorder(tree.left)
55              self.preorder(tree.right)
56
57      def inorder(self, tree):
58          if tree is not None:
59              self.inorder(tree.left)
60              print(tree.root, end=' ')
61              self.inorder(tree.right)
62
63      def postorder(self, tree):
64          if tree is not None:
65              self.postorder(tree.left)
66              self.postorder(tree.right)
67              print(tree.root, end=' ')
68
69  def main():
70      A = BinaryTree('A')
71      B = BinaryTree('B')
72      C = BinaryTree('C')
73      D = BinaryTree('D')
```

```
74      E = BinaryTree('E')
75      F = BinaryTree('F')
76      G = BinaryTree('G')
77      H = BinaryTree('H')
78      I = BinaryTree('I')
79      J = BinaryTree('J')
80
81      E.set_child(H, None)
82      B.set_child(D, E)
83      A.set_child(B, C)
84      C.set_child(F, G)
85      G.set_child(I, J)
86
87      A.traversal_preorder()
88      A.traversal_inorder()
89      A.traversal_postorder()
90
91  if __name__ == '__main__':
92      main()
```

程序的运行结果如下：

```
A B D E H C F G I J
D B H E A F C I G J
D H E B F I J G C A
```

3 递归

递归是函数通过直接或间接的方式调用自身的一种行为。它为执行重复性、相似性[30]的任务提供了一种优雅而强有力的解决方案。

[30]例如，我们说整体任务与子任务之间具有相似性，是指它们的解决方案是类似的，只是在问题的规模上不一样。

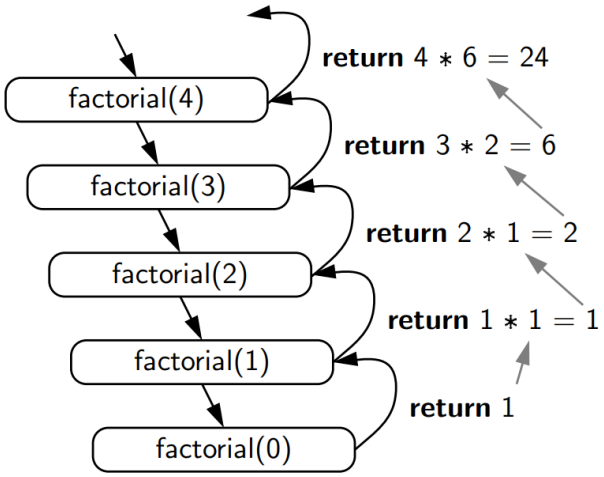

图 3-35: 阶乘的递归调用示例

实际上，只有少数几种语言 (例如 Scheme、Smalltalk 等) 没有提供显式的循环结构，它们依赖于递归实现循环。现代程序设计语言以统一的方式处理函数的一般调用与递归调用，即在调用方式上，它们没有本质上的区别。

3.1 几个典型的例子

3.1.1 阶乘

数学上，阶乘有一种简洁的递归定义形式：

$$n! = \begin{cases} 1 & if\ n = 0 \\ n \cdot (n-1)! & if\ n \geq 1. \end{cases} \tag{26}$$

实际上，该递归定义可以直接转换成递归函数：

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     return n * factorial(n - 1)
```

阶乘递归调用的一个示例如图3-35所示。下面估算它的时间复杂度。为计算 $factorial(n)$，需要连续递归调用到 $factorial(n-1)$、$factorial(n-2)$、...、$factorial(0)$，总共 $n+1$ 次，而每次递归调用需要 1 个常量计算。因此，它的时间复杂度是 $O(n)$。

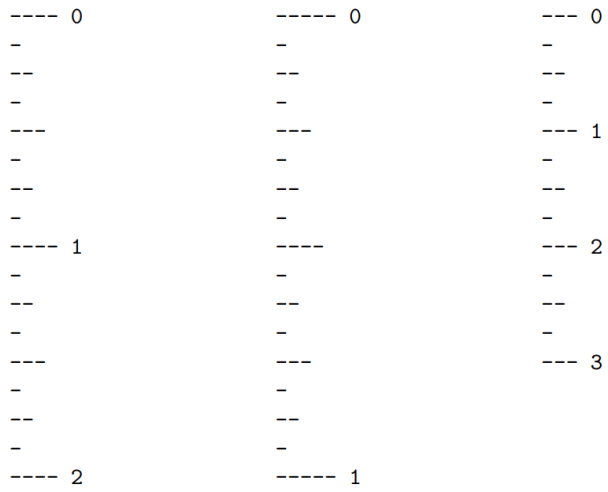

图 3-36: Ruler 的绘制实例

3.1.2 Ruler 的绘制

几种 Ruler 的绘制实例如图3-36所示。仔细分析 Ruler 的绘制实例，可以发现，该图形呈现出分形特征——图形的整体与部分具有相似性，或者说，图形在不同的层级上具有自相似或自递归结构。虽然也可以使用迭代的方式绘制 Ruler，但是递归的方式显得更加自然简洁。算法如下：

```
def draw_ruler(num_units, len_tick):
    draw a tick with len_tick and '0'
    for j in range(1, num_units + 1):
        recur_draw(len_tick - 1)
        draw a tick with len_tick and 'j'


def recur_draw(len_tick):
    if (len_tick > 0):
        recur_draw(len_tick - 1)
        draw a tick with len_tick
        recur_draw(len_tick - 1)
```

可以看出，递归函数 recur_draw 要比 factorial 复杂一些，原因在于前者的递归调用有 2 个分支。

程序如下：

```
def draw_ruler(num_units, len_tick):
    draw_tick(len_tick, '0')
    for j in range(1, num_units + 1):
        recur_draw(len_tick - 1)
        draw_tick(len_tick, str(j))

def recur_draw(len_tick):
    if (len_tick > 0):
        recur_draw(len_tick - 1)
        draw_tick(len_tick)
        recur_draw(len_tick - 1)

def draw_tick(len_tick, label = ''):
    tick = '-'*len_tick
    tick = tick + label
    print(tick)

def main():
    draw_ruler(2, 4)
    print()
    print()
    draw_ruler(1, 5)
    print()
    print()
    draw_ruler(3, 3)

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

下面分析一下 Ruler 绘制算法的时间复杂度。首先，分析recur_draw 递归函数。该函数的每次递归调用会产生 2 个分支，而每次递归调用的基本操作是绘制 1 个 tick，当len_tick=1 时，产生 1 个基本操作；当len_tick=2 时，产生 3 个

基本操作；运用归纳法可以证明，当以参数 $n-1$ 调用函数recur_draw 时，会产生 $2^{n-1}-1$ 个基本操作。再来分析函数draw_ruler，它总共产生m=num_units 次循环，总的基本操作次数将是 $m(2^{n-1}-1+1)+1=m2^{n-1}+1$ 次。因此，该算法的时间复杂度是 $O(m2^n)$。实际上，可以采用更严格的递归方程来分析算法的复杂度，后面将详细介绍。

3.1.3 二分搜索

假设需要从一列数据中搜索某个目标值，如果数据是无序的，那么可以执行如下顺序搜索算法：

```
def search(l, e):
    for i in range(len(l)):
        if l[i] == e:
            return True
    return False
```

最坏情况下，时间复杂度是 $O(n)$。但是，假如数据是有序的，那么可以对算法进行如下改进：

```
def search(l, e):
    for i in range(len(l)):
        if l[i] == e:
            return True
        elif l[i] > e:
            return False
    return False
```

算法在搜索到大于 e 的数据时停止。该算法可以降低平均时间复杂度，但是不会降低最坏情况下的时间复杂度。此时，可以使用二分搜索降低最坏情况下的时间复杂度，算法如下：

```
def search(l, e):
    if len(l) == 0:
        return False
```

```
    else:
        return bin_search(l, e, 0, len(l) - 1)
def bin_search(l, e, low, high):
    if low == high:
        return l[low] == e
    elif low > high:
        return False
    else:
        mid = (low + high) // 2
        if l[mid] == e:
            return True
        elif l[mid] > e:
            return bin_search(l, e, low, mid - 1)
        else:
            return bin_search(l, e, mid + 1, high)
```

二分搜索的程序如下：

```
def search(l, e):
    if len(l) == 0:
        return False
    else:
        return bin_search(l, e, 0, len(l) - 1)

def bin_search(l, e, low, high):
    if low == high:
        return l[low] == e
    elif low > high:
        return False
    else:
        mid = (low + high) // 2
        if l[mid] == e:
            return True
```

```
16          elif l[mid] > e:
17              return bin_search(l, e, low, mid - 1)
18          else:
19              return bin_search(l, e, mid + 1, high)
20
21  def main():
22      l = [1, 3, 24, 57, 64, 80, 90, 95, 100, 150, 164, 180, 190, 200]
23      print(search(l, 80))
24      print(search(l, 1))
25      print(search(l, 200))
26      print(search(l, -10))
27      print(search(l, 400))
28
29  if __name__ == '__main__':
30      main()
```

下面分析该算法的时间复杂度。仔细分析该算法，可以发现，每次递归调用时，它的基本操作 (比较操作) 是常量的，因此，该算法的运行时间与递归调用的次数成正比。每次递归调用，待搜索元素的个数为 $high - low + 1$。折半搜索后，剩余元素的个数为：

$$(mid - 1) - low + 1 = \lfloor \frac{low + high}{2} \rfloor - low \leq \frac{high - low + 1}{2} \tag{27}$$

或者，

$$high - (mid + 1) + 1 = high - \lfloor \frac{low + high}{2} \rfloor \leq \frac{high - low + 1}{2} \tag{28}$$

即每次折半搜索后，剩余元素的个数至多为原来元素个数的一半。在最坏情况下 (搜索不成功)，产生最大的递归深度，设深度为 r，有关系式：

$$\begin{aligned} \frac{n}{2^r} &< 1 \\ r &> log_2^n \\ r &= \lfloor log_2^n \rfloor + 1 \end{aligned} \tag{29}$$

其中，n 为总的元素个数。因此，该算法的时间复杂度为 $O(logn)$。

顺序搜索的时间复杂度是 $O(n)$，而二分搜索的时间复杂度是 $O(logn)$。这是巨大的效率提升，假如序列数据的个数为 10 亿，那么二分搜索算法仅仅只需要执行约 30 步。

3.1.4 文件系统

无论是 Unix/Linux 操作系统，还是 Windows 系列操作系统，文件系统都被设计成树形结构，这也是一种分形！因此，同样也可以采用递归方式遍历它。

下面的算法统计指定路径中所有文件及路径的磁盘空间使用量：

```
def disk_usage(path):
    space = disk_size(path)
    if path is a directory:
        for child in path:
            space = space + disk_usage(child)
    return space
```

其中，`disk_size(path)` 返回 path 自身占据的磁盘空间 (磁盘空间直接使用量)。

下面使用 Python 的 os 模块来实现该算法，需要使用如下函数：

- os.path.getsize(path)

  返回 path 自身占据的磁盘空间；

- os.path.isdir(path)

  检测 path 是否为目录；

- os.listdir(path)

  返回目录 path 下所有的文件及子目录 (不会进入下一层)。函数返回一个字符串列表；

- os.path.join(path, filename)

  生成 filename 的全路径名；

完整的程序如下：

```python
import os
def disk_usage(path):
    space = os.path.getsize(path)
    if os.path.isdir(path):
        for child in os.listdir(path):
            child  = os.path.join(path, child)
            space = space + disk_usage(child)
    print('{0:>14}'.format(space), path)
    return space

def main():
    disk_usage('E:\dpy')

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

下面分析它的时间复杂度。设指定路径中所有文件及目录的个数为 n。由算法可知，函数disk_usage 被调用的次数为 n[31]。再来看每次递归调用时，函数是否执行了常量时间。答案是否定的，原因在于，对于path 是子目录的情况，for 循环将被执行，其执行次数与path 的子目录个数有关；而如果path 是文件，则执行时间就是常量的。在最坏情况下，for 的循环次数为 $n-1$。此时，似乎可以得出结论，算法的时间复杂度是 $O(n^2)$。实际上，这个结论并不准确——并不是每个目录都包含那么多的子项 (文件和目录)。在这种情况下，从整体上进行分析而不是从最坏情况入手，可以得到更准确的结果：除了第 1 次调用disk_usage 之外，其余的 $n-1$ 次调用都是由for 发起的，而每次的disk_usage 调用都会执行一个常量计算时间 (即循环外的磁盘空间统计步骤)，那么总的计算时间将是 $n-1+1=n$ 次 (常量计算)。因此，算法的时间复杂度是 $O(n)$。

[31] 不论是文件的访问，还是子目录的访问，都是由 for 循环中函数 disk_usage 发起的，并且这些文件和子目录都只被访问一次。

3.2 递归滥用问题

3.2.1 元素唯一性检测算法

元素唯一性检测算法可以检测出给定序列数据中的所有元素是否都不一样。

第一个算法直接检测所有不同的元素对 $(d[j], d[k])$ 是否相同，其中 $j < k$。算法如下：

```
def unique1(d):
    for j in range(len(d)):
        for k in range(j+1, len(d)):
            if d[j] == d[k]:
                return False
    return True
```

可以非常直接地将该算法转换到 Python 程序：

```
1  def unique1(d):
2      for j in range(len(d)):
3          for k in range(j+1, len(d)):
4              if d[j] == d[k]:
5                  return False
6      return True
7
8  def main():
9      print(unique1([1,2,3,4]))
10     print(unique1([1,2,3,1]))
11     print(unique1([1,2,3,4,5,6,2]))
12
13 if __name__ == '__main__':
14     main()
```

该算法使用 2 重循环，内层循环的执行次数之和是：

$$(n-1) + (n-2) + (n-3) + ... + 1 = \frac{n(n-1)}{2} \tag{30}$$

因此，它的时间复杂度是 $O(n^2)$。

第 2 个算法的思路是，首先对序列数据进行排序预处理，然后遍历整个序列数据，检测任意 2 个邻接元素是否相同。算法如下：

```
def unique2(d):
    s = sorted(d)
    for j in range(1, len(s)):
        if s[j-1] == s[j]:
            return False
    return True
```

相应的 Python 程序如下：

```
def unique2(d):
    s = sorted(d)
    for j in range(1, len(s)):
        if s[j-1] == s[j]:
            return False
    return True

def main():
    print(unique2([1,2,3,4]))
    print(unique2([1,2,3,1]))
    print(unique2([1,2,3,4,5,6,2]))

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

排序算法的时间复杂度是 $O(nlogn)$，单重循环的时间复杂度是 $O(n)$。因此，该算法的时间复杂度是 $O(nlogn)$。

第 3 个算法的思路是使用递归——无论序列数据中元素的多少，基本的检测过程是一样的。首先，归纳出递归的检测过程。当 $n = 1$ 时，直接返回 True；当 $n = 2$ 时，当且仅当 2 个元素不同时，返回 True；当 $n = 3$ 时，设序列为 (d_0, d_1, d_2)，可以将问题归约为序列 (d_0, d_1) 和序列 (d_1, d_2) 的检测问题，另外，还

需额外判断 d_0 与 d_2 是否相同；当 $n=4$ 时，将问题归约为序列 (d_0,d_1,d_2) 和序列 (d_1,d_2,d_3) 的检测问题，另外，还需额外判断 d_0 与 d_3 是否相同；...；当问题规模为 n 时，将问题归约为序列 (d_0,d_1,d_2,d_{n-2}) 和序列 (d_1,d_2,d_3,d_{n-1}) 的检测问题，另外，还需额外判断 d_0 与 d_{n-1} 是否相同。然后，给出简化的递归算法如下：

```
def unique3(d, start, stop):
    if stop == start:
        return True
    elif not unique3(d, start, stop-1):
        return False
    elif not unique3(d, start+1, stop):
        return False
    else:
        return d[start] != d[stop]
```

相应的 Python 程序如下：

```
1  def unique3(d, start, stop):
2      if stop == start:
3          return True
4      elif not unique3(d, start, stop-1):
5          return False
6      elif not unique3(d, start+1, stop):
7          return False
8      else:
9          return d[start] != d[stop]
10
11 def main():
12     print(unique3([1,2,3,4],0,3))
13     print(unique3([1,2,3,1],0,3))
14     print(unique3([1,2,3,4,5,6,2],0,6))
15
16 if __name__ == '__main__':
```

```
17      main()
```

然而，该算法的效率是有问题的。当 $n=1$ 时，消耗 1 个常量计算时间；当 $n=2$ 时，最坏情况下，消耗 3 个常量计算时间 (2 次 $n=1$ 的递归，1 次常量判断，即总共 3 个常量计算)；当 $n=3$ 时，最坏情况下，消耗 7 个常量计算 (2 次 $n=2$ 的递归，1 次常量判断，即总共 $2*3+1=7$ 个常量计算)。依此类推，可知，当元素个数为 n 时，最坏情况下，总共消耗 2^n-1 个常量计算。因此，该算法的时间复杂度是 $O(2^n)$！这是一个典型的递归滥用例子。

3.2.2 Fibonacci 数列

Fibonacci 数列的递归定义如下：

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1},\ n \geq 2 \end{aligned} \tag{31}$$

直接将其转换成递归算法：

```
def fib1(n):
    if n <= 1:
        return n
    else:
        return fib1(n-2) + fib1(n-1)
```

程序如下：

```
1  def fib1(n):
2      if n <= 1:
3          return n
4      else:
5          return fib1(n-2) + fib1(n-1)
6
7  def main():
8      for n in range(11):
9          print(fib1(n))
```

```
10
11 if __name__ == '__main__':
12     main()
```

该算法的运行效率相当糟糕。当 $n = 0$ 和 $n = 1$ 时，各需要 1 个常量计算时间；当 $n = 2$ 时，在不考虑 $fib1(0) + fib1(1)$ 计算的情况下，需要 2 个常量计算时间；当 $n = 3$ 时，在不考虑所有求和计算的情况下，需要 3 个常量计算时间；依此类推，当问题规模为 n 时，在不考虑所有求和计算的情况下，恰好需要 $Fibonacci(n)$(即 $Fibonacci$ 数列的第 n 个数) 个常量计算时间，而 $Fibonacci(n)$ 的增长是关于 n 的指数级[32]。因此，本递归算法虽然直接自然，但并不可取。

实际上，如果我们能够充分利用已经计算出的 F_{n-2} 和 F_{n-1}，那么算法的时间复杂度就能够大大降低。下面的算法将 2 次递归改成了 1 次递归调用，并且每次递归调用都返回 2 个值，算法如下：

```
def fib2(n):
    if n <= 1:
        return (n, 0)
    else:
        (a, b) = fib2(n-1)
        return (a+b, a)
```

程序如下：

```
1  def fib2(n):
2      if n <= 1:
3          return (n, 0)
4      else:
5          (a, b) = fib2(n-1)
6          return (a+b, a)
7
8  def main():
9      for n in range(11):
```

[32] 更准确地说，$Fibonacci(n)$ 是最接近 $\frac{1.618^n}{\sqrt{5}}$ 的整数。具体内容请参考《计算机程序的构造和解释》(Harold Abelson 等，裘宗燕译，机械工业出版社，2016 年)，P25。

```
10          print(fib2(n)[0])
11
12  if __name__ == '__main__':
13      main()
```

这是一个线性递归，时间复杂度为 $O(n)$：每次递归调用后问题的规模为 $n-1$，且都使用常量的计算时间。

还有一种算法，不使用递归调用，而是直接使用迭代：

```
def fib3(n):
    if n <= 1:
        return n
    a = 1
    b = 0
    for i in range(2, n+1):
        a, b = a + b, a
    return a
```

程序如下：

```
1   def fib3(n):
2       if n <= 1:
3           return n
4       a = 1
5       b = 0
6       for i in range(2, n+1):
7           a, b = a + b, a
8       return a
9
10  def main():
11      for n in range(11):
12          print(fib3(n))
13
```

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

显然，该算法的时间复杂度为 $O(n)$。

3.3 递归的分类

无论什么情况下，我们可以把递归调用形成的轨迹图看作是一种树形结构[33]。从树形结构分支的角度看，可以将递归分成三种类型：线性递归[34]、二分支递归、多分支 (三分支及以上) 递归。

3.3.1 线性递归

线性递归的主要特征是，每次递归调用时，至多只会产生一个新的递归调用[35]。到目前为止，属于线性递归的算法有：阶乘计算 (factorial)、Fibonacci 计算 (fib2)、二分搜索 (bin_search)。

对于序列数据的处理问题，一种有趣的想法是，将它设计成递归算法，例如二分搜索。下面来看另一个例子，递归计算一个序列数据中所有元素的和。

该算法的思路是：如果元素的个数为 1，直接返回该元素；否则，递归计算前 $n-1$ 个元素的和，然后计算它与最后 1 个元素的和，并返回结果。算法如下：

```
def lsum(d, n):
    if n == 1:
        return d[n-1]
    else:
        return lsum(d, n-1) + d[n-1]
```

将算法转换到相应的程序是非常直接的，程序如下：

```
1 def lsum(d, n):
2     if n == 1:
3         return d[n-1]
```

[33]即使是形如 factorial 那样的单分支递归调用形成的轨迹图，也是一种特殊的树形结构。

[34]注意，此处的“线性”不是指时间复杂度是线性的，而是说递归分支是单支的。

[35]注意，不要与递归调用出现的次数相混淆。例如，在二分搜索递归函数 bin_search 中，递归调用出现的次数为 2 次，但每次只会产生一次递归调用。

```
    else:
        return lsum(d, n-1) + d[n-1]

def main():
    print(lsum([1,2,3,4,5,6,7,8,9,10],10))
    l = range(101)
    print(lsum(l,len(l)))

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

算法的时间复杂度取决于递归深度，它的递归深度为 n，并且每次递归的计算时间是常量的。因此，它的时间复杂度是 $O(n)$。类似地，空间复杂度也是 $O(n)$。

下面使用递归算法解决另一个问题：将序列数据进行反转，即第 1 个元素与最后一个元素对调，第 2 个元素与倒数第 2 个元素对调，依此类推。递归算法如下：

```
def reverse(d, start, stop):
    if start < stop:
        d[start], d[stop] = d[stop], d[start]
        reverse(d, start+1, stop-1)
```

程序如下：

```
def reverse(d, start, stop):
    if start < stop:
        d[start], d[stop] = d[stop], d[start]
        reverse(d, start+1, stop-1)

def main():
    l = list(range(11))
    print(l)
    reverse(l, 0, len(l)-1)
    print(l)
```

```
11
12 if __name__ == '__main__':
13     main()
```

算法的运行时间取决于递归深度，它的递归深度为 $1+\lfloor\frac{n}{2}\rfloor$，每次递归的计算时间是常量的。因此，算法的时间复杂度是 $O(n)$。

下面再看一个实例，利用递归算法计算 x 的任意非负整数幂：

$$x^n = \begin{cases} 1 & if\ n = 0 \\ x \cdot x^{n-1} & if\ n \geq 1. \end{cases} \tag{32}$$

根据定义直接写出递归算法：

```
def power1(x, n):
    if n == 0:
        return 1
    else:
        return x * power1(x, n-1)
```

程序如下：

```
1  def power1(x, n):
2      if n == 0:
3          return 1
4      else:
5          return x * power1(x, n-1)
6
7  def main():
8      print(power1(0.3, 3))
9      print(power1(3, 3))
10     print(power1(2, 10))
11
12 if __name__ == '__main__':
13     main()
```

该算法的运行轨迹与阶乘相似，其时间复杂度也是 $O(n)$，空间复杂度也是 $O(n)$。实际上，仔细分析该问题，可以得到一个更快的算法。首先，将定义修改为：

$$x^n = \begin{cases} 1 & if\ n = 0 \\ (x^{\lfloor \frac{n}{2} \rfloor})^2 & if\ n \geq 1\ and\ n\ is\ even \\ x \cdot (x^{\lfloor \frac{n}{2} \rfloor})^2 & if\ n \geq 1\ and\ n\ is\ odd \end{cases} \tag{33}$$

改进后的递归算法如下：

```
def power2(x, n):
    if n == 0:
        return 1
    elif n is even:
        temp = power2(x, n // 2)
        return temp * temp
    else:
        temp = power2(x, n // 2)
        return x * temp * temp
```

程序如下：

```
def power2(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        temp = power2(x, n // 2)
        return temp * temp
    else:
        temp = power2(x, n // 2)
        return x * temp * temp

def main():
    print(power2(0.3, 3))
    print(power2(3, 3))
```

```
14      print(power2(2, 10))
15
16  if __name__ == '__main__':
17      main()
```

改进后的算法，每次递归，n 的值都减半，其处理轨迹与二分搜索类似，它的时间复杂度为 $O(logn)$，空间复杂度也是 $O(logn)$，这是有意义的改进！

3.3.2 二分支递归

在递归算法中，如果产生 2 个递归调用，那么该递归被称为二分支递归。在前面的例子中，Ruler 的绘制及 Fibonacci 数列 (fib1) 就属于此种类型。

下面使用二分支递归重新设计序列数据求和算法。对于一个序列数据，每次递归时可以将该序列一分为二，并返回这 2 部分的相加结果，此过程一直进行到序列中只包含 2 个元素和或 1 个元素为止。算法如下：

```
def bin_sum(d, start, stop):
    if start == stop:
        return d[start]
    elif start == stop - 1:
        return d[start] + d[stop]
    else:
        mid = (start + stop) // 2
        return bin_sum(d, start, mid) + bin_sum(d, mid + 1, stop)
```

程序如下：

```
1  def bin_sum(d, start, stop):
2      if start == stop:
3          return d[start]
4      elif start == stop - 1:
5          return d[start] + d[stop]
6      else:
7          mid = (start + stop) // 2
```

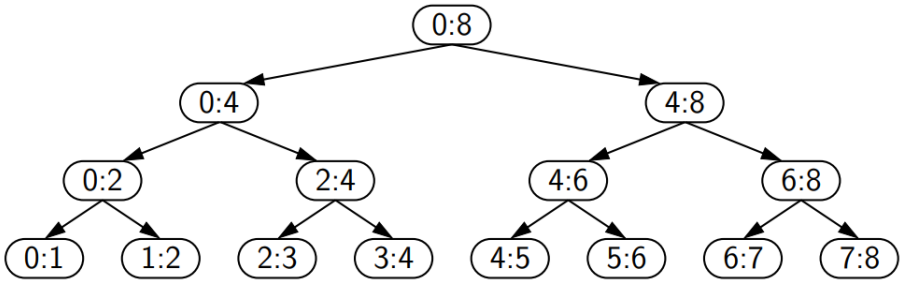

图 3-37: bin_sum 的递归调用轨迹示意图

```
8           return bin_sum(d, start, mid) + bin_sum(d, mid + 1, stop)
9
10  def main():
11      print(bin_sum([1,2,3,4,5,6,7,8,9,10],0,9))
12      l = list(range(101))
13      print(bin_sum(l,0,len(l)-1))
14
15  if __name__ == '__main__':
16      main()
```

该算法的执行轨迹图是一棵二叉树。为分析方便，假设它是一棵满二叉树，示意图如图3-37所示。在图中，每个圆角矩形框内均标识出了 start 和 stop 的数值。可以看出，该算法的递归深度为 $1+\lfloor log_2 n \rfloor$，它的空间复杂度是 $O(logn)$，与线性递归版本相比，这是一个有意义的改进。它的运行时间取决于树中节点的个数。对于满二叉树，它的节点个数为 $2^{1+\lfloor log_2 n \rfloor}-1=2(n-1)-1=2n-3$。它的时间复杂度是 $O(n)$。

下面再看一个子集求解的实例。以字符串为例，例如'abcdef'，找出该集合的所有子集 (幂集)。可以使用递归算法进行求解：对于任一字符，执行 2 分支递归——子集中包括该字符的递归与子集中不包括该字符的递归。算法如下：

```
def power_set(str):
    recur_set('', str)


def recur_set(sofar, rest):
    if rest == '':
```

```
        print(sofar)
    else:
        recur_set(sofar+rest[0], rest[1:])
        recur_set(sofar, rest[1:])
```

程序如下：

```
def power_set(str):
    recur_set('', str)

def recur_set(sofar, rest):
    if rest == '':
        if sofar == '':
            print('_')
        else:
            print(sofar)
    else:
        recur_set(sofar+rest[0], rest[1:])
        recur_set(sofar, rest[1:])

def main():
    power_set('abc')

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

算法每次递归产生 2 个分支，递归深度为 n，总共生成 $2^{n+1}-1$ 个节点，算法的时间复杂度为 $O(2^n)$，空间复杂度为 $O(n)$。

3.3.3 多分支递归

在递归算法中，如果产生三个及以上递归调用，那么该递归被称为多分支递归。在前面的例子中，统计目录磁盘空间的算法就属于此类——递归调用的次数取决于当前目录中的子目录个数。

对于多分支递归，它的递归调用轨迹图是一棵多分支的树。因此，它的调用次数可能随着树的深度加深而急剧地增长。

在一些情况下，我们需要以递归的方式遍历树中的每个节点[36]，除叶子节点之外的其它节点，被称为决策点 (Decision Point)。在回溯 (backtracking) 算法中，经常需要回到最近的某个决策点重新试探或选择其它的分支。

下面看一个排列求解的实例。该实例虽然针对字符，但是其求解算法具有通用性。假设有一个字符串，例如 “abcdef”，试找出由这些字符组成的所有字符串 (长度与原字符串的长度相同，且字符不重复使用，这是一个排列问题)。穷尽递归的算法如下：

```
def str_perm(str):
    recur_perm('', str)


def recur_perm(sofar, rest):
    if rest == '':
        print(sofar)
    else:
        for each char in rest:
            remaining = rest
            remove char from remaining
            recur_perm(sofar+char, remaining)
```

程序如下：

```
1 def str_perm(str):
2     recur_perm('', str)
3
4 def recur_perm(sofar, rest):
5     if rest == '':
6         print(sofar)
7     else:
8         for i in range(len(rest)):
```

[36]这种方式被称为 “穷尽递归 (Exhaustive Recursion)”。

```
9               remaining = rest[0:i] + rest[i+1:]
10              recur_perm(sofar+rest[i], remaining)
11
12  def main():
13      str_perm('abcd')
14
15  if __name__ == '__main__':
16      main()
```

为求解该问题，需要穷尽遍历所有可能的排列，总共有 $n!$ 个排列！这种基本的穷尽排列算法是许多递归算法的核心。

下面对该穷尽算法进行改写：如果在遍历的过程中找到了目标，那么终止遍历并返回；否则，继续遍历或回溯 (backtracking)。我们可以利用改进的算法搜索生成的字符串中是否出现了有效的单词，例如 “face”、“bed” 等。算法如下：

```
def FindWord(str, goal):
    return recur_find('', str, goal)


def recur_find(sofar, rest, goal):
    if rest == '':
        return goal is in sofar?sofar:''
    else:
        for each char in rest:
            remaining = rest
            remove char from remaining
            found = recur_find(sofar+char, remaining)
            if found:
                return found;
    return ''
```

程序如下：

```
1   def FindWord(str, goal):
2       return recur_find('', str, goal)
```

```
3
4  def recur_find(sofar, rest, goal):
5      if rest == '':
6          if goal in sofar:
7              return sofar
8          else:
9              return ''
10     else:
11         for i in range(len(rest)):
12             remaining = rest[0:i] + rest[i+1:]
13             found = recur_find(sofar+rest[i], remaining, goal)
14             if found:
15                 return found
16     return ''
17
18 def main():
19     print(FindWord('abcdef', 'face'))
20     print(FindWord('abcdef', 'bed'))
21     print(FindWord('abcdef', 'ace'))
22
23 if __name__ == '__main__':
24     main()
```

4 排序

排序是处理数据集的基本操作，它按照某种规则确定数据集中每个数据元素的顺序。例如，成绩统计中需要按照成绩的高低对学生进行排序；字典中需要按照字母顺序对单词进行排序，这些排序操作为人们解决问题提供了极大的方便。在计算机科学中，排序操作往往作为数据预处理的一个过程，能够提高算法的效率。例如，二分搜索中，如果预先对数据进行排序，则可以将算法的时间复杂度从 $O(n)$ 降低到 $O(logn)$，这是算法执行效率上的一个极大提升！

对于排序问题，一个自然的想法是，每次从数据中挑选出一个最小元素，直至所有的数据元素处理完毕[37]。显然，它的时间复杂度是 $O(n^2)$。实际上，可以对这种算法进行优化，获得更好的运行效率。本部分的内容将从最简单的排序算法开始，讨论各种优化的排序算法、时间复杂度和特点。

在讨论之前，需要了解以下几个内容：

- 键值 (key)

  在任何比较排序的算法中，用于确定元素排序顺序的值。对于简单的数据类型，元素值本身就是键值，而对于复杂的数据类型，可以指定某个分量或一组分量作为键值；

- 原址 (in-place) 排序

  排序时，不使用额外的空间。例如，排序时，仅仅交换或移动元素的位置；

- 稳定性

  这是排序算法的一个重要性质。稳定排序的算法，能够维持相同元素的相对位置不变；

- 适应性

  这是排序算法的另一个重要性质。如果排序算法对接近有序的数据集工作得更快，那么就表明该算法具有更好的适应性。具有适应性的算法也是有实际意义的，因为在实际应用中，常常需要处理准有序数据集；

4.1 $O(n^2)$ 算法

4.1.1 冒泡排序

冒泡排序 (Bubble Sort) 基于这样一种思路：如果数据集没有排好序，一定存在逆序的元素，那么通过交换逆序的元素对，可以让数据集更接近于有序；通过不断地减少数据集中的逆序，最终可以得到有序数据集。

具体的排序过程是，对数据集进行多次元素遍历，每次遍历时，比较相邻元素，并将较小或较大的元素前移 (元素交换)；一趟遍历后，最大或最小的元素被

[37]这种排序方法符合人们的直觉，在日常生活中被大量使用，被称为选择排序算法。

后移到了序列末尾；经过多趟遍历之后，如果数据集中没有任何元素移动，表明数据集达到稳定状态，排序过程终止。算法如下：

```
def bubble_sort(d):
    n = len(d)
    for i in range(n-1):
        found = False
        for j in range(1, n-i):
            if d[j-1] > d[j]:
                swap(d[j-1], d[j])
                found = True
        if not found:
            break
```

程序如下：

```
1  def bubble_sort(d):
2      n = len(d)
3      for i in range(n-1):
4          found = False
5          for j in range(1, n-i):
6              if d[j-1] > d[j]:
7                  d[j-1], d[j] = d[j], d[j-1]
8                  found = True
9          if not found:
10             break
11
12 def main():
13     d = [3,5,20,4,6,1,50,30,40,15,2]
14     print(d)
15     bubble_sort(d)
16     print(d)
17
```

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

下面分析冒泡排序算法的复杂度。在最坏情况下，数据集完全逆序，当 $i=0$ 时，内循环比较 $n-1$ 次，当 $i=1$ 时，内循环比较 $n-2$ 次，...，当 $i=n-2$ 时，内循环比较 1 次，总的比较次数为：

$$(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2} \tag{34}$$

最坏情况下的时间复杂度为 $O(n^2)$。最好情况下，数据集完全正序，内层循环比较 $n-1$ 次，$found=False$，排序提前结束，时间复杂度为 $O(n)$。

下面分析平均情况下算法的时间复杂度。设 C 为算法执行的比较次数，S 为算法执行的元素交换次数，从最坏情况分析可知，$C=S=\frac{n(n-1)}{2}$。设 i 和 j 是元素的索引，如果对于任意 $i<j$，有 $d[i]>d[j]$(或 $d[i]<d[j]$，如果元素按从大到小的顺序排序)，则称元素 $d[i]$ 与 $d[j]$ 是逆序的。在排序过程中，如果 i 与 j 直接相邻，则在一趟排序过程中，逆序元素对将被交换，成为正序元素对；如果 i 与 j 不直接相邻，那么在若干趟排序后，该逆序元素对也将被交换，成为正序元素对。因此，逆序元素对的个数就是元素的交换次数 S。算法的真正排序时间[38]取决于逆序对的个数或元素的交换次数 S。在数据集中，假设元素的位置随机出现，任意元素对为正序与逆序的概率相等，那么逆序元素对的个数平均为 $\frac{n(n-1)}{4}$，即平均情况下元素对交换的次数为 $S=\frac{n(n-1)}{4}$。因此，平均情况下，算法的时间复杂度仍然为 $O(n^2)$。

冒泡排序算法被认为是性能最差的排序算法之一，原因在于，只要存在逆序，每一趟都会进行元素对的交换。在数据集完全逆序的情况下，至始至终都要进行元素对的交换，代价很大。

实际上，即使在数据集完全逆序的情况下 (此时的逆序元素对有 $\frac{n(n-1)}{2}$ 个)，排序算法也并非一定要执行 $\frac{n(n-1)}{2}$ 次元素对的交换操作。例如，序列数据 $(5,4,3,2,1)$，使用冒泡算法排序时，需要执行 10 次元素对的交换操作。然而，如果使用“每次选取最小元素并交换”[39]的方法，则只需要经过 $(1,4,3,2,5)\rightarrow(1,2,3,4,5)$ 两次元素交换操作即可。这个例子表明，冒泡排序算法还有改进的空间。

[38]算法的实际排序时间是由比较次数决定的。例如，在最好情况下，即使数据集是完全有序的(没有任何元素对进行交换)，算法仍然要执行比较操作 $n-1$ 次。即便如此，对时间复杂度的量级也不会产生影响。

[39]这种算法就是下文要介绍的选择排序算法。

4.1.2 选择排序

选择排序 (Selection Sort) 算法的思路是，每次找到数据集中最小或最大的元素，一直进行到所有的元素都被挑选完为止。该算法与冒泡排序算法一样，需要进行多趟排序，但与之不同的是，选择排序算法在每趟结束后只进行一次元素交换，降低了交换代价！算法如下：

```
def select_sort(d):
    n = len(d)
    for i in range(n-1):
        find the smallest element in d[i:] and swap(d[i], smallest)
```

注意，选择排序算法在同一数据结构 d 中维持着一个有序集和待排序集，d 的前端是有序集，后端是待排序集。排序前，有序集为空，随着排序的进行，有序集增长，待排序集缩减，直至排序完成，待排序集为空。每次新选取的最小元素始终放在前端有序集的末尾。程序如下：

```
1  def select_sort(d):
2      n = len(d)
3      for i in range(n-1):
4          smallest = i
5          for j in range(i+1, n):
6              if d[j] < d[smallest]:
7                  smallest = j
8          if smallest != i:
9              d[smallest], d[i] = d[i], d[smallest]
10
11 def main():
12     d = [3,5,20,4,6,1,50,30,40,15,2]
13     print(d)
14     select_sort(d)
15     print(d)
16
```

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

下面分析算法的时间复杂度。显然，选择排序算法的比较次数与数据集的有序或无序程度无关，它的内外层循环次数始终是固定的。当 $i=0$ 时，内循环比较 $n-1$ 次，当 $i=1$ 时，内循环比较 $n-2$ 次，...，当 $i=n-2$ 时，内循环比较 1 次，总的比较次数为：

$$(n-1)+(n-2)+...+2+1=\frac{n(n-1)}{2} \tag{35}$$

该算法没有适应性，在任何情况下，时间复杂度都是 $O(n^2)$。但是，与冒泡排序算法相比，它的元素对交换次数是 $O(n)$，而后者是 $O(n^2)$。

选择排序算法不具有稳定性，原因在于：在每趟排序的过程中，从左到右搜索到的最小值 (可能存在多个相同的最小值)，将它们陆续交换到前端有序集的末尾时，它们的顺序并不会改变 (它们是稳定的)；但是，与它们对应的这些被交换的元素，在与最小值交换后，会越过中间的一批元素，如果这批元素中有与它们的值相同的元素，那么它们的顺序就出现了问题。因此，选择排序算法是不稳定的[40]。

这种基本的选择排序算法在各个方面都不如插入排序算法，其实际平均效率也低于插入排序算法。因此，在实际应用中，基本的选择排序算法用得并不多。

4.1.3 插入排序

插入排序 (Insertion Sort) 是一种简单的排序算法，它的平均与最坏时间复杂度都是 $O(n^2)$。它的原理是，反复将待排序元素插入到已排序的数据中。对于小规模数据集的排序，如果数据集已经是有序的，那么优先选择插入排序。插入排序的算法如下：

```
def insert_sort(d):
    for e in d:
        pos is the position of e in d
        newpos is the proper position for e in d[:pos]
        move the element in d[newpos:pos] backward and
        insert e in newpos
```

[40] 可以对基本的选择排序算法进行改进，使之具有稳定性。

注意，插入排序算法在同一个数据结构 d 中维持着一个有序集和待排序集，d 的前端是有序集，后端是待排序集。排序前，有序集为空或只包含第 1 个元素，随着排序的进行，有序集增长，待排序集缩减，直至排序完成，待排序集为空。在排序的过程中，对于每一个当前待排序的元素，需要在有序集中找到正确的位置，然后在有序集中移动该位置及后面的所有元素，腾出 1 个空间，并将该元素插入其中，从而完成一次排序。程序如下：

```
1  def insert_sort(d):
2      for pos in range(1, len(d)):
3          value = d[pos]
4          while pos > 0 and value < d[pos-1]:
5              d[pos] = d[pos-1]
6              pos = pos-1
7          d[pos] = value
8
9  def main():
10     d = [3,5,20,4,6,1,50,30,40,15,2]
11     print(d)
12     insert_sort(d)
13     print(d)
14
15 if __name__ == '__main__':
16     main()
```

下面分析该算法的时间复杂度。在最坏情况下，数组完全逆序，插入第 2 个元素时要考察前 1 个元素，插入第 3 个元素时，要考虑前 2 个元素，……，插入第 n 个元素，要考虑前 $n-1$ 个元素。因此，最坏情况下的比较次数是 $1+2+...+(n-1)=\frac{n(n-1)}{2}$，时间复杂度为 $O(n^2)$。最好情况下，数组已经是有序的，每插入一个元素，只需要考查前一个元素。因此，插入排序的时间复杂度为 $O(n)$，表明该算法具有适应性。此外，该算法是稳定的，因为在检索新元素的插入位置时，一旦发现前面的元素与新元素相等，就停止检索，从而保证了相等的元素不会交换位置。

下面分析算法的平均时间复杂度。在排序过程中，假设元素插入各位置的概率相等，那么每次内层循环的平均比较次数为[41]：

$$\frac{1+2+...+pos}{pos} = \frac{pos(pos+1)}{2} \cdot \frac{1}{pos} = \frac{pos+1}{2} \tag{36}$$

内外循环总共的比较次数为：

$$\sum_{pos=1}^{n-1} \frac{pos+1}{2} = \frac{1}{2}(2+3+...+n) = \frac{1}{2}[\frac{n(n+1)}{2} - 1] \tag{37}$$

可以看出，平均时间复杂度仍然是 $O(n^2)$。

4.2 $O(nlogn)$ 算法

4.2.1 归并排序

归并排序 (Merge Sort) 是一种通用有效、基于比较的排序算法。它采用了分治 (Divide & Conquer) 策略，该策略由 John von Neumann 在 1945 年发明。早在 1948 年，Goldstine 和 von Neumann 就对归并排序进行了详细的描述与分析。

分治策略的典型实现方式是递归，我们在第3章已经见识了递归的优雅与强大。典型的分治策略由三个步骤组成：

1. Divide

   如果输入的规模小于某个阈值 (例如 1 个或 2 个)，直接解决该问题并返回；否则，将输入分成 2 个或多个较小规模的子集；

2. Conquer

   递归解决与这些子集对应的问题 (子问题)；

3. Combine

   将这些子问题的解合并成整个问题的解；

下面给出归并排序的分治策略，假设序列数据为 S，共有 n 个元素：

[41] 元素的索引为 0、1、2、...、*pos*-1，即总共有 *pos* 个元素。因此，比较次数分别为 1、2、3、...*pos*，即总共有 $1+2+3+...+pos$ 次比较。

1. Divide

   如果 S 只有 1 个元素，直接返回 S；否则，将 S 分成 2 个子集：$S1$ 和 $S2$，其中，$S1$ 包含 $\lfloor \frac{n}{2} \rfloor$ 个元素，$S2$ 包含 $\lceil \frac{n}{2} \rceil$ 个元素；

2. Conquer

   递归排序 $S1$ 和 $S2$；

3. Combine

   将 $S1$ 和 $S2$ 合并成有序序列；

下面给出归并排序算法：

```
def merge_sort(S):
    n = len(S)
    if n < 2:
        return
    mid = n // 2
    S1 = S[:mid]
    S2 = S[mid:]
    merge_sort(S1)
    merge_sort(S2)
    merge(S1, S2, S)


def merge(S1, S2, S):
    i = j = 0
    while i+j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] <= S2[j]):
            S[i+j] = S1[i]
            i = i+1
        else:
            S[i+j] = S2[j]
            j = j+1
```

可以非常直接地转换到 Python 程序：

```python
def merge_sort(S):
    n = len(S)
    if n < 2:
        return
    mid = n // 2
    S1 = S[:mid]
    S2 = S[mid:]
    merge_sort(S1)
    merge_sort(S2)
    merge(S1, S2, S)

def merge(S1, S2, S):
    i = j = 0
    while i+j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] <= S2[j]):
            S[i+j] = S1[i]
            i = i+1
        else:
            S[i+j] = S2[j]
            j = j+1

def main():
    d = [3,5,20,4,6,1,50,30,40,15,2]
    print(d)
    merge_sort(d)
    print(d)

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

该归并算法没有适应性，无论什么样的序列，都需要执行全部步骤。算法具有稳定性，对于具有相同关键字的元素，从函数 merge 可知，左元素的优先级要

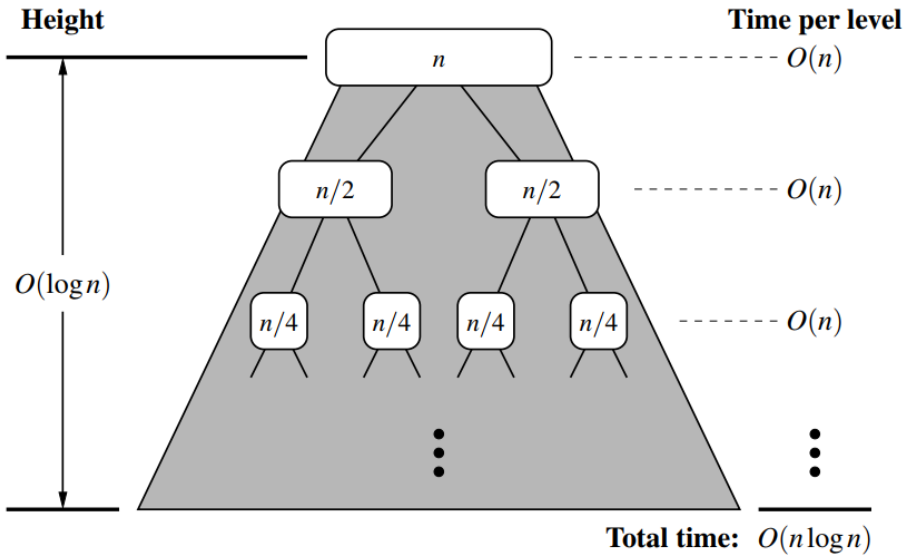

图 4-38: 归并排序的时间复杂度分析

高于右元素。

下面分析它的时间复杂度。首先，分析函数 merge 的时间复杂度。设 $S1$ 和 $S2$ 的元素个数分别为 $n1$ 和 $n2$，则该函数需要执行 $O(n1+n2)$ 个常量操作。对于归并排序算法的总时间复杂度，可以采取整体分析的方法，如图4-38所示。易知，归并排序树的高度为 $O(logn)$(每次将序列数据分成 2 个准等分子集 $S1$ 和 $S2$，分析方法与二分搜索类似)。而在树的每一层，将执行多次 merge 操作，每次的时间复杂度为 $O(n1+n2)$，它取决于元素的个数。在每一层中，总的元素个数为 n，则每层 merge 的时间复杂度是 $O(n)$。因此，归并算法的时间复杂度为 $O(nlogn)$。

下面使用递归方程来分析它的时间复杂度。设 $t(n)$ 表示最坏情况下该算法的时间复杂度，n 为序列大小，则归并递归算法的时间复杂度为：

$$t(n) = \begin{cases} b & if\ n \leq 1 \\ 2t(\frac{n}{2}) + cn & otherwise. \end{cases} \tag{38}$$

其中，b 和 c 为常量因子，cn 为函数 merge 所需时间。对于 $n > 1$ 的情况，连续地应用递归方程，得到：

$$\begin{aligned} t(n) = 2t(\frac{n}{2}) + cn = 2[2t(\frac{n}{2^2}) + c\frac{n}{2}] + cn = 2^2 t(\frac{n}{2^2}) + 2cn \\ = ... = 2^i t(\frac{n}{2^i}) + icn \end{aligned} \tag{39}$$

当 $n = 2^i$ 或 $i = log_2^n$ 时[42]，$t(n)$ 的递归进程结束，此时：

$$t(n) = nt(1) + cnlog_2^n = nb + cnlog_2^n. \tag{40}$$

因此，归并算法的时间复杂度为 $O(nlogn)$。

4.2.2 快速排序

快速排序 (Quicksort) 算法在 1959 年由 Tony Hoare 开发，并于 1961 年发表。该算法是运用分治策略的成功范例之一，它仍然是目前被广泛使用的排序算法。在实现好的情况下，速度大约是其它主要算法 (归并排序和堆排序) 的2~3 倍。

虽然归并排序与快速排序都是基于分治策略的算法，但是，两者还是有明显的区别。在归并排序中，2 个递归子集的划分是以索引为基础的——整个序列数据按位置被分成元素个数大致相同的 2 个递归子集；归并排序的主要工作量是在 merge 函数中，即工作量主要集中于递归完成后。在快速排序中，2 个递归子集是以枢轴 (Pivot) 元素为中心划分的——整个序列数据被枢轴元素划分成 2 个递归子集，第 1 个递归子集的每个元素都小于或等于枢轴元素，而第 2 个递归子集的每个元素都大于枢轴元素。与归并排序不同的是，快速排序的主要工作量是序列数据的划分，即工作量主要集中于递归前。

下面从分治策略的角度分析快速排序算法的主要步骤 (序列数据集为 S)：

1. Divide

   如果 S 的元素个数少于 2 个，直接返回；否则，从 S 中选取枢轴元素 x(可以任意选择。一般选取 S 中的第一个元素或最后一个元素)，将 S 分成 2 个子集：

   - L，它的每个元素小于或等于 x；
   - R，它的每个元素大于 x；

2. Conquer

   递归排序子集 L 和 R；

3. 将 L、x 和 R 连接起来组成有序序列；

[42]假设 n 是 2 的指数关系。对于一般情况，可以做类似的分析，结论不变。

下面给出快速排序算法：

```
Input:
    S: a sequence
    l: the leftmost index
    r: the rightmost index
def quick_sort(S, l, r):
    if l >= r:
        return
    pivot = S[r]
    l_forward = l
    r_backward = r
    while l_forward < r_backward:
        scan S from l_forward to r_backward-1, until S[l_forward] > pivot
        S[r_backward] = S[l_forward]
        scan S from r_backward to l_forward+1, until S[r_backward] <= pivot
        S[l_forward] = S[r_backward]
    S[l_forward] = pivot

    quick_sort(S, l, l_forward-1)
    quick_sort(S, l_forward+1, r)
```

程序如下：

```
1 def quick_sort(S, l, r):
2     if l >= r:
3         return
4     pivot = S[r]
5     l_forward = l
6     r_backward = r
7     while l_forward < r_backward:
8         while l_forward <= r_backward-1 and S[l_forward] <= pivot:
9             l_forward = l_forward + 1
```

```
10          S[r_backward] = S[l_forward]
11          while r_backward >= l_forward+1 and S[r_backward] > pivot:
12              r_backward = r_backward - 1
13          S[l_forward] = S[r_backward]
14      S[l_forward] = pivot
15
16      quick_sort(S, l, l_forward-1)
17      quick_sort(S, l_forward+1, r)
18
19  def main():
20      d = [3,5,20,4,6,1,50,30,40,15,2]
21      print(d)
22      quick_sort(d, 0, len(d)-1)
23      print(d)
24
25  if __name__ == '__main__':
26      main()
```

快速排序算法不具有适应性，无论数据序列如何，都需要执行所有步骤。该算法也是不稳定的，原因是，在递归调用前的子集划分过程中，存在元素的跳跃移动现象。该算法属于原址 (in-place) 排序。

下面分析算法的时间复杂度。在 2 次递归调用前，算法需要对序列数据进行划分，其主要工作量为元素的比较。在最坏情况下，排序树的深度为 $O(n)$，即排序树是一棵单分支的二叉树 (左分支树或右分支树，每次只去掉一个元素进行递归)，快速排序将蜕化为冒泡排序，时间复杂度为 $O(n^2)$。在平均情况下，根据二叉树理论，所有 n 个节点的二叉树的平均高度为 $O(logn)$。从整体上看，在树的每一层，需要进行比较的次数为 $O(n)$。因此，平均情况下，算法的时间复杂度为 $O(nlogn)$，如图4-39所示。更正式的分析方法，可以使用递归方程，请参考相关资料。

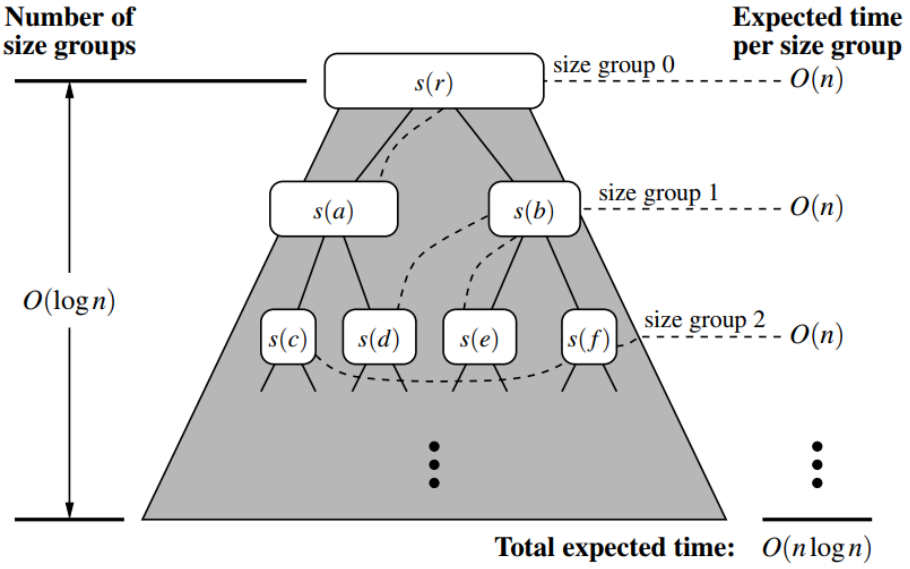

图 4-39: 快速排序的时间复杂度示意图

4.2.3 堆排序

堆排序算法利用堆作为存储序列数据的结构，初始时，从原始序列数据中构造出堆，然后弹出堆顶元素，将它存入堆的末尾，并对堆的其余元素进行一次向下筛选的操作，重新生成一个新堆 (比原堆少一个元素)，接着再弹出堆顶元素，重复上述过程，直至堆中只剩下一个元素为止。此时，整个数据序列，或者按从大到小的顺序排列 (最小堆)，或者按从小到大的顺序排列 (最大堆)。

堆排序算法如下：

```
def heap_sort(d):
    h = MaxHeap()
    h.heapify(d)
    n = len(d)
    for index in range(1, n):
        d[n-index], d[0] = d[0], d[n-index]
        h.sift_down(d, d[0], 0, n-index)
```

堆排序程序如下：

```
1 from heap import Heap
2
```

```
def heap_sort(d):
    h = Heap(opr='>')
    h.heapify(d)
    n = len(d)
    for index in range(1, n):
        d[n-index], d[0] = d[0], d[n-index]
        h.sift_down(d, d[0], 0, n-index)

def main():
    d = [3,5,20,4,6,1,50,30,40,15,2]
    print(d)
    heap_sort(d)
    print(d)

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

显然，该算法属于原址排序。构造堆的时间复杂度为 $O(n)$(参见2.4)，循环体总共执行 $n-1$ 次，每次执行比较次数为 $2\lfloor log_2^i \rfloor$，$i \in [2,n]$，总的比较次数大约为：

$$2[log_2^n + log_2^{n-1} + ... + log_2^3 + log_2^2] = 2log_2^{n!} \tag{41}$$

根据 Stirling 近似公式：

$$log_2^{n!} = nlog_2^n - nlog_2^e + O(log_2^n), \tag{42}$$

可知，在最坏情况下，堆排序的时间复杂度也为 $O(nlogn)$。与快速排序相比，这是堆排序的一大优点。

5 算法设计策略

本章将开始讨论问题求解的几种常见算法。对于许多问题，在这些方法中，很有可能至少有一种方法是适用的。

5.1 分治算法

5.1.1 基本形式

分治算法的思想是，将一个规模较大的问题分解成若干个规模较小的问题，如果这些较小规模的问题能够直接求解，那么整体问题也能够求解；否则，继续将这些子问题进行分解，直到都能够直接求解为止。

从实现方式来看，分治算法与递归是密不可分的。递归提供了一种逻辑直观的实现形式。分治算法产生了许多高效的算法。

分治算法的基本形式如下：

```
Input:
    P: problem
def divide-and-conquer(P)
    if P.size <= n0:
        solve(P)
    else:
        divide P into smaller problem:P=P1,P2,...Pk
        for p in P:
            ri = divide-and-conquer(p)
        return merge(r1,r2,...rk)
```

其中，$n0$ 表示问题 P 规模的阈值，当问题的规模小于该值时，问题可直接求解；$merge$ 表示将所有子问题的解合并成整体解。实践经验表明，子问题的规模大致相当时，算法的效果最好。在大多数应用中，可以只将问题分解成 2 个子问题，即取 $k = 2$。

分治算法的时间复杂度可以使用递归方程来分析。在分析时，一般假设 $solve$ 操作能够在常量时间 $O(1)$ 内完成，将问题分解及合并所需时间为 $f(n)$，则分治算法所需时间为：

$$T(n) = \begin{cases} O(1) & if\ n = n_0 \\ kT(\frac{n}{m}) + f(n) & otherwise,\ n > n_0. \end{cases} \tag{43}$$

其中，$\frac{n}{m}$ 为子问题规模。一般情况下，可以通过反复展开递归式来求解递归方程，

得到（为计算方便，假定 n 是 m 的整数幂）：

$$\begin{aligned} T(n) = kT(\frac{n}{m}) + f(n) = k[kT(\frac{n}{m^2}) + f(\frac{n}{m})] + f(n) = k^2T(\frac{n}{m^2}) + kf(\frac{n}{m}) + f(n) \\ = k^2[kT(\frac{n}{m^3}) + f(\frac{n}{m^2})] + kf(\frac{n}{m}) + f(n) \\ = k^3T(\frac{n}{m^3}) + k^2f(\frac{n}{m^2}) + kf(\frac{n}{m}) + f(n) = ... \\ = k^{log_m^n}T(1) + \sum_{j=0}^{log_m^n-1} k^j f(\frac{n}{m^j}) \\ = n^{log_m^k}T(1) + \sum_{j=0}^{log_m^n-1} k^j f(\frac{n}{m^j}) \end{aligned} \tag{44}$$

5.1.2 二分搜索

二分搜索是分治算法的典型例子，已经在3.1.3中讨论过。

5.1.3 归并排序

归并排序也是分治算法的典型例子，已经在4.2.1中讨论过。

5.1.4 快速排序

快速排序也是分治算法的典型例子，已经在4.2.2中讨论过。

5.1.5 大整数的乘法

在一些应用中，要处理很大的整数乘法。然而，大整数乘法无法直接在计算机硬件所允许的范围内计算，必须使用算法来计算它。

设 X 与 Y 都是 n 位二进制整数，现在需要计算它们的乘积 $X * Y$。最直接简单的算法是采用类似手工计算乘积的方法，但是这种算法的时间复杂度为 $O(n^2)$。

现在考虑使用分治算法，以设计出更加有效的乘积算法。首先，将 X 与 Y 都分成规模相同的 2 个部分[43]，每段长度都为 $\frac{n}{2}$。设 $X = A|B$，$Y = C|D$，则有：

$$\begin{aligned} X &= 2^{\frac{n}{2}}A + B \\ Y &= 2^{\frac{n}{2}}C + D \end{aligned} \tag{45}$$

[43]为分析方便，假设 n 是 2 的幂。

于是，X 与 Y 的乘积为：

$$\begin{aligned} XY &= (2^{\frac{n}{2}}A + B)(2^{\frac{n}{2}}C + D) \\ &= 2^n AC + 2^{\frac{n}{2}}(AD + BC) + BD \end{aligned} \tag{46}$$

从上式可以看出，该式需要执行 4 次 $\frac{n}{2}$ 位整数乘法，分别是 AC、AD、BC、BD，它们的复杂度为 $4T(\frac{n}{2})$。此外，还需执行 3 次不超过 $2n$ 位的整数加法以及 2 次移位操作，它们的时间复杂度为 $O(n)$，则该方法的总时间复杂度为：

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ 4T(\frac{n}{2}) + O(n) & otherwise,\ n > 1. \end{cases} \tag{47}$$

考虑公式44，其中，$k = 4$，$m = 2$，代入可得 $T(n) = O(n^2)$。显然，该方法与手工计算方式相比，没有改进。

要想改进算法的时间复杂度，需要减少 $\frac{n}{2}$ 位整数乘法的次数，继续对公式 XY 进行变换：

$$XY = 2^n AC + 2^{\frac{n}{2}}[(A - B)(D - C) + AC + BD] + BD \tag{48}$$

虽然上式在形式上显得更加复杂，但是实际上，它只需执行 3 次 $\frac{n}{2}$ 位整数乘法，它们分别是 AC、BD、$(A - B)(D - C)$，时间复杂度为 $3T(\frac{n}{2})$。此外，还需执行 6 次加减法和 2 次移位操作，时间复杂度为 $O(n)$。该方法的总时间复杂度为：

$$T(n) = \begin{cases} O(1) & if\ n = 1 \\ 3T(\frac{n}{2}) + O(n) & otherwise,\ n > 1. \end{cases} \tag{49}$$

显然，$T(n) = O(n^{log_2^3}) = O(n^{1.59})$，算法有较大的改进。

5.1.6 矩阵乘法

设 A 与 B 是 $n \times n$ 矩阵，要计算它们的乘积 $C = AB$。最常规的做法是利用三重 n 循环，每次计算 C 的一个元素 $C[i][j] = \sum_{k=1}^{n} A[i][k]B[k][j]$。计算 $C[i][j]$ 需要执行 n 次乘法和 $n - 1$ 次加法，时间复杂度为 $O(n)$，而 C 矩阵总共有 n^2 个元素。因此，该算法的时间复杂度为 $O(n^3)$。

下面讨论如何应用分治算法提高矩阵乘法的计算效率问题。早在 20 世纪 60 年代，Strassen 就提出了矩阵乘法的分治算法，将时间复杂度降低到

$O(n^{log_2^7}) = O(n^{2.81})$。其基本思想与大整数乘法类似，将大矩阵分成规模相同的小矩阵，并设法降低小矩阵的乘法次数。

为讨论方便，仍然假设 n 是 2 的幂。首先，分别将矩阵 A 与 B 分成 4 个 $\frac{n}{2} \times \frac{n}{2}$ 的小矩阵，利用子矩阵计算矩阵乘法：

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = AB = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \tag{50}$$

由此可得：

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{51}$$

上述矩阵分解过程一直进行到 2 阶矩阵为止：2 个 2 阶矩阵相乘，一共需要 8 次乘法和 4 次加法，时间复杂度为 $O(1)$。对于一般情况，需要计算 8 次 $\frac{n}{2}$ 阶矩阵乘法，时间复杂度为 $T(\frac{n}{2})$；还需要计算 4 次 $\frac{n}{2}$ 阶矩阵加法，时间复杂度为 $O(n^2)$，则总的时间复杂度为：

$$T(n) = \begin{cases} O(1) & if\ n = 2 \\ 8T(\frac{n}{2}) + O(n^2) & otherwise,\ n > 2. \end{cases} \tag{52}$$

显然，与大整数乘法一样，只单纯地进行分解，而没有对关键计算进行优化，时间复杂度不会有任何的降低，该算法的时间复杂度仍然是 $T(n) = O(n^{log_2^8}) = O(n^3)$。

为了改善时间复杂度，必须减少关键算法——乘法的次数。Strassen 提出了一种方法，将乘法次数降低到了 7 次，但是该方法相应地增加了加法与减法的次数：

$$\begin{aligned} M_1 &= A_{11}(B_{12} - B_{22}) \\ M_2 &= (A_{11} + A_{12})B_{22} \\ M_3 &= (A_{21} + A_{22})B_{11} \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ M_7 &= (A_{11} - A_{21})(B_{11} + B_{12}) \end{aligned} \tag{53}$$

然后，再使用下面的公式计算：

$$\begin{aligned} C_{11} &= M_5 + M_4 - M_2 + M_6 \\ C_{12} &= M_1 + M_2 \\ C_{21} &= M_3 + M_4 \\ C_{22} &= M_5 + M_1 - M_3 - M_7 \end{aligned} \tag{54}$$

改进的算法，总共使用了 7 次 $\frac{n}{2}$ 阶矩阵乘法和 18 次 $\frac{n}{2}$ 阶矩阵加法，总的时间复杂度为：

$$T(n) = \begin{cases} O(1) & if\ n = 2 \\ 7T(\frac{n}{2}) + O(n^2) & otherwise,\ n > 2. \end{cases} \tag{55}$$

因此，$T(n) = O(n^{log_2^7}) = O(n^{2.81})$，时间复杂度有较大的改进。

实际上，如果一个算法能够再进一步减少矩阵乘法的次数，那么时间复杂度还可能进一步得到改善。但是，对于 2 个 2 阶矩阵的乘法，Hopcroft 和 Kerr 于 1971 年已经证明，7 次乘法是必要的。这意味着，如果总是将大矩阵分解成 $2 \times 2 = 4$ 个 $\frac{n}{2}$ 阶矩阵的话，算法的时间复杂度不可能再进一步降低；除非，将矩阵分解成其它的形式，例如 $3 \times 3 = 9$ 或 $5 \times 5 = 25$ 个子矩阵。目前，最好的时间复杂度可以达到 $O(n^{2.376})$。

5.1.7 棋盘覆盖问题

设有一个 $2^k \times 2^k$ 个方格组成的棋盘，其中含有一个特殊的方格，如图5-40所示。现在需要使用如图5-41所示的 4 种 L 形骨牌覆盖该棋盘，要求：不能覆盖特殊方格、L 形骨牌不能重叠。显然，需要使用的 L 形骨牌的个数为 $\frac{4^k-1}{3}$。

乍一看，该问题好像与分治算法无关。原因在于，在将大棋盘分割成 4 个小棋盘之后，总是有 3 个棋盘与原棋盘的特征不相符——没有特殊方格。然而，实际上，可以在 3 个无特殊方格的棋盘边界上放置一个 L 形骨牌，如图5-42所示，那么就可以将这些棋盘转换成含有特殊方格的棋盘，从而就可以递归地应用这种策略，连续分割棋盘，直至 1×1 个方格组成的棋盘为止。

从算法策略可知，它的时间复杂度公式为：

$$T(k) = \begin{cases} O(1) & if\ k = 0 \\ 4T(k-1) + O(1) & otherwise,\ k > 0. \end{cases} \tag{56}$$

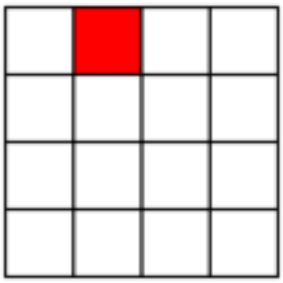

图 5-40: k=2 时的棋盘

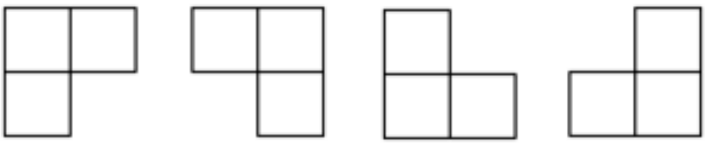

图 5-41: 4 个 L 形骨牌

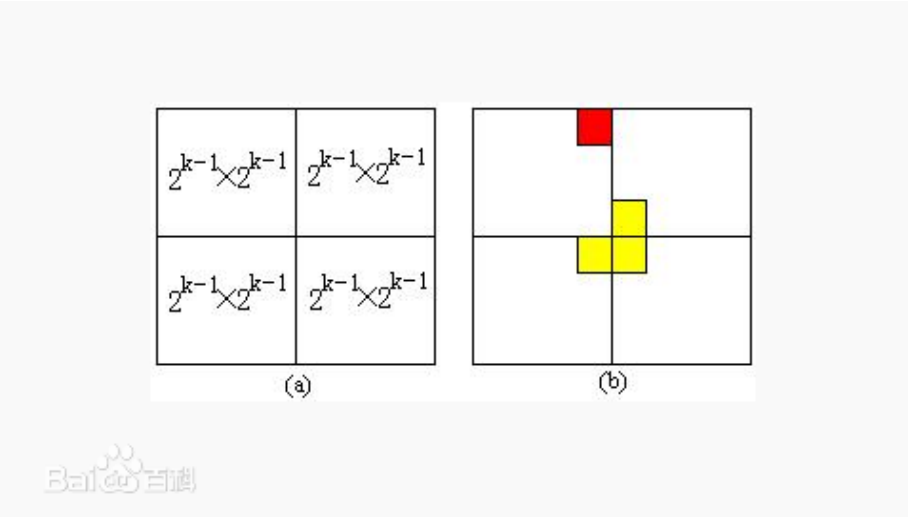

图 5-42: 递归地分割棋盘

连续代入 $T(k-1)$、$T(k-2)$、...、$T(0)$，解得 $T(k)=O(4^k)$。因为 $2^k\times 2^k$ 棋盘所需的 L 形骨牌个数为 $\frac{4^k-1}{3}$，所以该算法是渐近意义下最优的算法。

5.2 动态规划

5.2.1 一般方法

动态规划的基本思想与分治算法很相似，它也是需要将整个问题分解为若干个子问题，并分别求解子问题的最优解，最后从这些子问题的最优解得到整个问题的最优解。动态规划适用于求解最优化问题。

动态规划具有以下特点：

- 与分治算法相比，分解后得到的子问题往往不是独立的，而是相互关联的。如果仍然采用分治算法求解，那么这种关联性会使得算法消耗指数级的时间；
- 子问题在取得最优解时，整个问题也将取得最优解，反之亦然[44]；
- 动态规划的另一个特征是，专门使用记录表来保存所有子问题的最优解。在需要时，直接从表中获取解，从而避免了大量的重复计算；

动态规划于 20 世纪 50 年代由 Richard Bellman 提出，它的理论基础是最优性原理 (Principle of Optimality)。

在实际应用中，有一类被称为多阶段决策的马尔可夫决策进程 (Markov Decision Process, MDP)[45]问题：其过程被分成若干阶段，在任一阶段 i，其决策只依赖于阶段 $i-1$，而与之前的状态及如何到达阶段 i 的方式无关。

在多阶段决策的每一阶段，都面临多种选择，那么，从第 1 阶段到最后阶段，将会形成所有可能的决策序列。最优化求解该问题，就是要从所有的决策序列中选择一个能让问题获得最优解的决策序列——最优决策序列。

显然，利用穷举法直接从所有可能的决策序列中选取最优决策序列是最低效的算法。在许多实际应用中，穷举法也是不可行的。

Bellman 提出的最优性原理告诉我们，无论过程的初始状态和初始决策是什么，其余的决策必须相对于初始决策构成一个最优决策序列。该原理意味着，子

[44]在有些文献中，这条性质被称为“最优子结构性质”。

[45]MDP 是强化学习的理论基础之一。

问题与整体问题的最优性是一致的。如果实际问题符合最优性原理，那么动态规划方法就是可行的。

应用动态规划方法解决问题的关键在于，获取各阶段的最优递推关系式。有了最优递推关系式，就可以采取自底向上的方式构造出最优解 (最优决策序列)。利用最优性原理设计的动态规划方法，可以使得枚举量急剧下降，能够有效地解决实际问题。

通常，使用以下步骤设计动态规划算法：

1. 分析问题的最优解性质——整个问题的最优解与子问题的最优解是否一致；
2. 递归地定义最优解，构造最优递推关系式；
3. 以自底向上的方式构造最优解；

5.2.2 多矩阵乘法

现在考虑 n 个矩阵 $A_1, A_2, ..., A_n$ 的乘法问题。矩阵乘法满足结合律，因此，n 个矩阵的乘法有许多种不同的计算次序，而计算次序又决定了计算量。

例如，设有 4 个矩阵 A_1、A_2、A_3 与 A_4，它们的维度分别为 50×10、10×40、40×30 与 30×5。下面列出不同乘法顺序所需的计算次数：

- $(A_1((A_2A_3)A_4))$

  A_2A_3 所需计算次数为 $10 \times 40 \times 30 = 12000$，$(A_2A_3)A_4$ 所需计算次数为 $12000 + 10 \times 30 \times 5 = 13500$，$(A_1((A_2A_3)A_4))$ 所需计算次数为 $13500 + 50 \times 10 \times 5 = 16000$；

- $(A_1(A_2(A_3A_4)))$

  A_3A_4 所需计算次数为 $40 \times 30 \times 5 = 6000$，$A_2(A_3A_4)$ 所需计算次数为 $6000+10\times40\times5 = 8000$，$(A_1(A_2(A_3A_4)))$ 所需计算次数为 $8000+50\times10\times5 = 10500$；

- $((A_1A_2)(A_3A_4))$

  计算方法依此类推，不再赘述。所需计算次数为 36000；

- $(((A_1A_2)A_3)A_4)$

  所需计算次数为 87500；

- $((A_1(A_2A_3))A_4)$

  所需计算次数为 34500；

可见，不同的计算顺序对计算量的影响很大。如何找出最优的计算顺序呢？

穷举法是最容易想到的方法，但是它不是有效的算法。下面的分析表明，当问题规模很大时，穷举法根本就不可行。

设 $P(n)$ 表示 n 个矩阵乘法的不同计算顺序的个数，那么有 $P(1)=P(2)=1$，$P(3)=2$，$P(4)=5$，...。如何得到它的一般形式呢？对于 n 个矩阵的连乘，可以把它看作 2 个组成部分：假设第 1 部分有 k 个矩阵，那么它的不同计算顺序的个数为 $P(k)$；第 2 部分有 $n-k$ 个矩阵，它的不同计算顺序的个数为 $P(n-k)$；这 2 部分合起来总的不同计算顺序的个数为 $P(k)P(n-k)$。由此得到 $P(n)$ 的递推公式为：

$$P(n)=\begin{cases}1 & if\ n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k) & otherwise,\ n>1.\end{cases} \tag{57}$$

实际上，$P(n)$ 是 Catalan 数：$P(n)=C(n-1)$。Catalan 数的公式如下：

$$C(n)=\frac{1}{n+1}\begin{pmatrix}2n\\ n\end{pmatrix}=\Omega(\frac{4^n}{n^{\frac{3}{2}}}) \tag{58}$$

由此可见，$P(n)$ 随 n 呈指数级增长。

下面考虑使用动态规划来求解最优的计算次序问题。算法按如下步骤进行：

1. 分析问题解的性质——判断整体最优解与子问题的最优解是否一致

   为描述方便，将矩阵连乘积 $A_iA_{i+1}...A_j$ 记为 $A[i:j]$，那么 $A[1:n]$ 就表示 $A_1A_2...A_n$。

   假设已经为 $A[1:n]$ 找到了一个最优计算次序，并且假设该最优计算次序在 $1\le k<n$ 处，将 $A[1:n]$ 划分成 $A[1:k]$ 与 $A[k+1:n]$ 两个子问题，那么子问题 $A[1:k]$ 与 $A[k+1:n]$ 的计算次序也应该是最优的。如若不然，使用 $A[1:k]$ 的最优计算次序替换之，则 $A[1:n]$ 将会获得比最优计算次序还要好的计算次序，显然这是自相矛盾的。因此，子问题 $A[1:k]$ 与 $A[k+1:n]$ 的计算次序必然也是最优的。

上述分析表明，矩阵连乘问题的最优计算次序可以使用动态规划方法来求解。

2. 建立最优递推关系

此步骤是动态规划算法的核心。记 $m[i][j]$ 为计算 $A[i:j]$ 连乘积所需的最少计算次数。

当 $i=j$ 时，$A[i:j]=A_i$ 为单一矩阵，无需计算，$m[i][i]=0$，$i=1,2,3,...,n$。

当 $i<j$ 时，设 $i\leq k<j$，此时需要枚举所有的 k，并利用如下递归公式找到使 $m[i][j]$ 获得最小值的 k 值：

$$m[i][j]=\min_{i\leq k<j} m[i][k]+m[k+1][j]+d_i d_k d_j \tag{59}$$

实际上，该递归公式是根据最优性原理或整体最优解与子问题最优解的一致性进行计算的，其中 d_* 表示矩阵维度。需要注意的是，算法要保留最优解对应的 k 值，这些 k 值表示矩阵连乘需要断开的地方，它们被保存于数组 $s[i][j]$ 中。

3. 构造最优解

本步骤将从数组 $s[i][j]$ 中恢复矩阵乘法链需要添加括号 (在断开位置添加括号) 的地方，这些括号决定了矩阵的计算顺序。

下面给出多矩阵乘法的动态规划程序：

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 21 15:03:13 2018

@author: duxiaoqin
Functions:
    (1)Dynamic Programming: Matrix Multiply
"""

from myarray2d import Array2D

```

```
"""
Input:
    dims: Matrix Dimensions
    m: the optimality of Matrix Chain's time complexity
    s: the optimal position of Matrix Chain
"""
def DP(dims, m, s):
    for up_diag in range(1, len(dims)):
        for i in range(0, len(dims)-up_diag):
            j = i + up_diag
            m[i, j] = m[i, i] + m[i+1, j] + \
                      dims[i][0] * dims[i+1][0] * dims[j][1]
            s[i, j] = i
            for k in range(i+1, j):
                temp = m[i, k] + m[k+1, j] + \
                      dims[i][0] * dims[k+1][0] * dims[j][1]
                if temp < m[i, j]:
                    m[i, j] = temp
                    s[i, j] = k

def traceBack(s, i, j, chain):
    if i == j:
        return
    traceBack(s, i, s[i, j], chain)
    traceBack(s, s[i, j]+1, j, chain)
    if i < s[i, j]:
        index = chain.index(chr(ord('A')+i))
        chain.insert(index, '(')
        index = chain.index(chr(ord('A')+s[i, j]))
        chain.insert(index+1, ')')

```

```
43      if s[i, j]+1 < j:
44          index = chain.index(chr(ord('A')+s[i, j]+1))
45          chain.insert(index, '(')
46          index = chain.index(chr(ord('A')+j))
47          chain.insert(index+1, ')')
48
49  def main():
50      #dims = [(50, 10), (10, 40), (40, 30), (30, 5)]
51      dims = [(30, 35), (35, 15), (15, 5), (5, 10), (10, 20), (20, 25)]
52      m = Array2D(len(dims), len(dims))
53      m.clear(0)
54      s = Array2D(len(dims), len(dims))
55      s.clear(-1)
56      DP(dims, m, s)
57      chain = ['(']
58      for index in range(0, len(dims)):
59          chain.append(chr(ord('A')+index))
60      chain.append(')')
61      traceBack(s, 0, len(dims)-1, chain)
62      print('The optimal matrix chain is:', end=' ')
63      print(''.join(chain))
64      for r in range(m.numRows()):
65          for c in range(m.numCols()):
66              if r <= c:
67                  print('%6d' % m[r, c], end=' ')
68              else:
69                  print(' '*6, end=' ')
70          print()
71      print()
72      for r in range(s.numRows()):
73          for c in range(s.numCols()):
```

```
74              if r <= c:
75                  print('%6d' % s[r, c], end=' ')
76              else:
77                  print(' '*6, end=' ')
78          print()
79
80  if __name__ == '__main__':
81      main()
```

程序的运行结果如下：

```
The optimal matrix chain is: ((A(BC))((DE)F))
0  15750   7875   9375  11875  15125
       0   2625   4375   7125  10500
              0    750   2500   5375
                     0   1000   3500
                            0   5000
                                   0

-1      0      0      2      2      2
       -1      1      2      2      2
              -1      2      2      2
                     -1      3      4
                            -1      4
                                   -1
```

实现算法的关键步骤是第 2 步，需要利用递归关系计算每个 $m[i][j]$。一般情况下，需要为计算 $m[i][j]$ 构建合理的计算顺序，以确保在计算 $m[i][j]$ 时，$m[i][k]$ 和 $m[k+1][j]$ 已经被计算过。对于本例，合理的计算顺序如图5-43所示，箭头方向为计算顺序。

在构造最优解的 traceBack 函数中，简单地利用递归程序分段确定矩阵乘法链需要添加括号 (断开) 的地方，这些括号确定了矩阵的计算顺序。

该算法的时间复杂度为 $O(n^3)$。相比于穷举法的指数级时间复杂度，动态规划

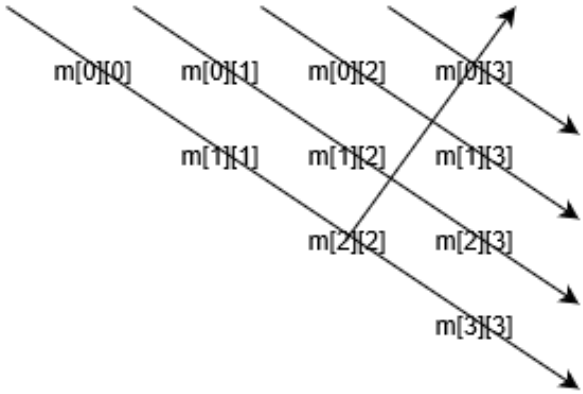

图 5-43: 最优递归数组的计算顺序

算法要有效得多。

5.2.3 0-1 背包问题

给定 n 种物品和一个背包，物品的重量为 $w = \{w_1, w_2, ..., w_n\}$，物品的价值为 $v = \{v_1, v_2, ..., v_n\}$，背包的容量为 C，求解的目标是：如何选择装入背包的物品，使得装入物品的总价值最大？在选择装入物品时，物品要么被装入，要么不被装入，且物品不能重复装入，这个问题被称为 0-1 背包问题。

该问题可以被进一步形式化。给定 $C > 0$，$w_i > 0$，$v_i > 0$，$1 \leq i \leq n$，要求找出 n 维 0-1 向量 $(x_1, x_2, ..., x_n)$，$x_i \in \{0, 1\}$，使得：

$$max \sum_{i=1}^{n} v_i x_i \tag{60}$$

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \leq C \\ x_i \in \{0, 1\}, 1 \leq i \leq n \end{cases} \tag{61}$$

利用反证法，可以证明该问题的整体最优解与子问题最优解之间具有一致性的特点。设 $(y_1, y_2, ..., y_n)$ 是所给 0-1 背包问题的一个最优解，那么 $(y_2, y_3, ..., y_n)$ 是下面子问题的一个最优解：

$$max \sum_{i=2}^{n} v_i x_i \tag{62}$$

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \leq C - w_1 y_1 \\ x_i \in \{0,1\}, 2 \leq i \leq n \end{cases} \tag{63}$$

如若不然，设 $(z_2, z_3, ..., z_n)$ 是上述子问题的一个最优解，那么有：

$$\begin{aligned} v_1 y_1 + \sum_{i=2}^{n} v_i z_i > \sum_{i=1}^{n} v_i y_i \\ w_1 y_1 + \sum_{i=2}^{n} w_i z_i \leq C \end{aligned} \tag{64}$$

这说明 $(z_1, z_2, ..., z_n)$ 比 $(y_1, y_2, ..., y_n)$ 更优，这与 $(y_1, y_2, ..., y_n)$ 是最优解的假设相矛盾。

下面给出动态规划算法求解 0-1 背包问题所需要的核心要素——递归关系。

首先，设 $m(i, j)$ 是下面子问题的最优值：

$$max \sum_{k=i}^{n} v_k x_k \tag{65}$$

$$\begin{cases} \sum_{k=i}^{n} w_k x_k \leq j \\ x_k \in \{0,1\}, i \leq k \leq n \end{cases} \tag{66}$$

由此，根据整体最优解与子问题最优解的一致性特点，建立如下的最优值递归公式：

$$m(i,j) = \begin{cases} max\{m(i+1,j), m(i+1,j-w_i) + v_i\} & j \geq w_i \\ m(i+1,j) & 0 \leq j < w_i \end{cases} \tag{67}$$

最优值的初始值为：

$$m(n,j) = \begin{cases} v_n & j \geq w_n \\ 0 & 0 \leq j < w_n \end{cases} \tag{68}$$

5.3 贪心算法

5.3.1 基本要素

贪心/贪婪算法 (Greedy Algorithm) 分阶段地工作，它总是做出在当前看来局部最优的选择，即贪心算法并没有从整体上进行最优地选择，而是在局部上做出最优选择。

显然，贪心算法不能对所有问题都能够得到整体最优解。但是，对一些符和某些特征的问题，贪心算法确实能够得到整体最优解，下文将详细描述。在另一些情况下，虽然贪心算法不能得到整体最优解，但是也能够得到满足要求的次优解。

下面先看一个能够得到整体最优解的问题。例如，考察找硬币的例子。假设有 4 种硬币，它们的面值分别为 0.25 元、0.1 元、0.05 元、0.01 元四种。现在需要找给某个顾客 0.63 元，并且要使得找出的硬币个数最少。

针对这个特定的问题，可以设计出这样一种贪心算法：首先，选出一个币值不超过 0.63 元的最大硬币，即 0.25 元，剩余 0.63-0.25=0.38 元；然后，再选出一个币值不超过 0.38 元的最大硬币，还是 0.25 元，剩余 0.38-0.25=0.13 元；接着，再选出一个币值不超过 0.13 元的最大硬币，即 0.1 元，剩余 0.13-0.1=0.03 元；最后，依照相同的方式，连续地选出 3 个 0.01 元的硬币。实际上，以上过程将得到一个整体最优解——找给顾客的硬币个数最少。

该问题具有一个重要的特征：整体最优解与局部 (子问题) 最优解是一致的[46]。在5.2中，我们提到过，如果一个问题具有该特征，那么该问题将能够使用动态规划方法来解决。

但是，动态规划问题需要采用自底向上的方式，通过递归公式层层推进，最终才能计算出整体最优解。换句话说，在动态规划算法中，每步所作出的选择往往要依赖于它的相关子问题的解；只有在这些子问题被求解之后，才能得到它的解。

与之形成对比的是，针对一些具有该特征的问题，贪心算法则更简单更有效。在贪心算法中，在定义了贪心准则之后，就可以在每步依据贪心准则直接做出最优的局部选择。每做出这样的贪心选择，就可以将原问题缩小为规模更小的子问题。然后，继续应用贪心算法于该子问题，直至没有新的子问题产生为止。

下面再看一个使用贪心算法不能得到最优解的例子。还是来看找硬币的例子，如果将硬币的面值修改为 0.01 元、0.05 元和 0.11 元三种，需要找给顾客 0.15 元。如果使用贪心算法，将找给顾客 1 个 0.11 元和 4 个 0.01 元的硬币，总共需要 5 个硬币。然而，此时的最优解却是找给顾客 3 个 0.05 元的硬币。对于这个问题，动态规划算法却可以有效地解决它，得到整体最优解。一个主要的原因在于，在考虑找硬币问题时，应该比较选择该硬币与不选择该硬币所导致的最终方案，然后再做出最佳选择。这个过程将会导出许多互相重叠的子问题，而这正是动态规

[46]注意，并不是所有具有该特征的找硬币问题，使用贪心算法都能够找到整体最优解，下文将举一个这样的例子，这与问题所选用的币值种类有关。

划算法能够求解问题并得到整体最优解的另一个重要特点。然而，贪心算法并没有比较并选择这个过程，不能获得整体最优解也就不足为奇了。

虽然新的找硬币例子仍然符合“整体最优解与局部最优解具有一致性”特征[47]，但是使用贪心算法却不能得到整体最优解。由此可见，“整体最优解与局部最优解具有一致性”特征只是贪心算法适用的一个必要条件。

对于一个具体的问题，贪心算法是否可用？是否能够得到整体最优解？这些问题难以得到肯定的回答。但是，从许多贪心算法适用的实际问题中，可以看到它们具有如下两个重要的性质：

- 整体最优解与子问题 (局部) 最优解具有一致性

  无论是动态规划算法，还是贪心算法，适合应用它们的问题必须具备该性质。该性质已经在5.2中讨论过；

- 贪心选择性质

  所谓贪心选择性质，指的是整体最优解可以通过每步的贪心选择来完成。这意味着，在贪心算法中，仅仅依赖当前状态就做出最优的选择 (局部最优)。因此，如果需要使用贪心算法求解出整体最优解，就必须要证明每步做出的贪心选择，最终确实能够导致整体的最优解。事实上，对一些问题 (例如，活动最优安排问题，下文中将详细介绍)，就可以使用数学归纳法与反证法加以证明。

  而在动态规划算法中，需要权衡所有步骤上的解，即仅仅依据当前状态而做出的最优选择未必就能导致整体的最优解。该算法以自底向上的方式，求解所有步骤上的解，并且在每一步骤都进行比较，以选取到目前步骤为止的“整体”最优解。当在最顶端进行比较并选择最优解时，获得的解才是整体最优解。

  在贪心算法中，如果已经证明了问题具有贪心选择性质，那么为求解整体最优解，通常可以采用自顶向下的方式进行简单的贪心选择，每做出一次贪心选择，就将所求问题简化为规模更小的子问题。

[47]可以使用归纳法与反证法证明问题具有该特征。假设第 j 个币值在最优硬币集合 A 中，设 $A_j = A - j$ 是拿出第 j 个硬币后的最优硬币集合 (它对应的零钱数是原零钱数减去第 j 个硬币的币值)，那么局部解 A_j 也应该是最优的。如若不然，假设存在一个局部最优解 B，那么 $B \cup j$ 将比 A 找的硬币数个数还要少，这与 A 是最优解的假设相矛盾。

下面讨论，对于给定问题，证明它具有贪心选择性质的一般方法。首先，考察问题的一个整体最优解，并证明可修改这个最优解，使其从贪心选择开始；在做出贪心选择后，原问题简化为规模更小的类似子问题；最后，采用数学归纳法证明，通过每一步的贪心选择，最终可以得到问题的整体最优解。其中，证明贪心选择后的问题简化为规模更小的子问题，依据的是该问题还具有“整体最优解与子问题 (局部) 最优解具有一致性”的特征。

5.3.2 活动最优安排问题

活动最优安排问题，是贪心算法能够得到整体最优解的典型例子。求解该问题，要求尽可能多地安排一系列活动，而这些活动需要争用某一公共资源。

假设有 n 个活动集合 $E = \{1, 2, 3, ..., n\}$，其中，每个活动共享同一资源，例如活动场所等；在任一时刻，只有一个活动能够使用该资源。每个活动 i 都有一个起始时刻 s_i 和结束时刻 f_i，且 $s_i < f_i$。一旦该活动在进行，则它在半开区间 $[s_i, f_i)$ 内占用资源，其它活动则不能被执行。如果活动 i 的半开区间 $[s_i, f_i)$ 与活动 j 的半开区间 $[s_j, f_j)$ 不存在重叠的区域，那么就称这 2 个活动是相容的。如果活动集合中所有的活动是相容的，那么就称该集合是相容活动集合。活动最优安排问题的目标就是要求解所给活动集合中最大的相容活动子集合。

在明确了问题的目标之后，打算使用贪心算法进行求解。首先，对原有问题进行改造调整：将各活动 (s_i, f_i) 按关键字 f_i 从小到大的顺序排列。然后，使用集合 A 保存所选择的活动：每次选择 f_i 最小 (贪心选择) 且与集合 A 相容的活动，将其加入到集合 A 中。该过程从第 1 个活动开始，一直到最后一个活动，最后将得到集合 A。下面将证明，A 就是最优的相容活动集合。

假设 $E = \{1, 2, 3, ..., n\}$ 为所给的活动集合，它已经按照 f_i 从小到大的顺序排序，那么活动 1 具有最早的完成时间。首先，证明该问题存在一个最优解，它可以从贪心选择活动 1 开始进行构造。设 $A \subseteq E$ 是一个最优解，且 A 中活动也是按 f_i 从小到大的顺序排列，假设 k 是 A 中的第 1 个活动。若 $k = 1$，则 A 就是从贪心选择开始的最优解；若 $k > 1$，则设 $B = (A - \{k\}) \cup \{1\}$，因为 $f_1 \leq f_k$，且 A 中活动是相容的，所以 B 中活动也是相容的，又由于 A 是最优解，则 B 也是最优解 (活动个数与 A 相同)。因此，对于活动集合 E，总存在从贪心选择开始的最优解。

下面将利用反证法证明，该问题具有“整体最优解与子问题最优解一致”的特

点。在第 1 步中，算法做出了贪心选择，选择了活动 1，那么原问题就简化为对 $E' = E - \{1\}$ 中活动的最优安排问题。设 A 是原问题的最优解，则 $A' = A - \{1\}$ 也应该是问题 E' 的最优解。如若不然，设 B' 是 E' 问题的最优解，它包含的活动个数要比 A' 多，那么 $B' \cup \{1\}$ 将是一个比最优解 A 的活动个数还要多的解，而这与 A 是原问题的最优解相矛盾的。因此，整体最优解与子问题最优解具有一致性的特点。这意味着，每一步所做出的贪心选择，都将问题简化为一个更小的且与原问题求解目标相似的子问题。

综上所述，对活动安排问题连续地应用贪心选择，一直到没有活动选择为止，最终将得到一个整体最优解。

下面给出它的程序实现：

```
# -*- coding: utf-8 -*-
"""
Created on Sun Sep 23 15:46:20 2018

@author: duxiaoqin
Functions:
    (1) Greedy Algorithm: tasks schedule
"""

def greedy(tasks):
    results = [tasks[0]]
    for task in tasks[1:]:
        if results[-1][1] <= task[0]:
            results.append(task)
    return results

def main():
    tasks = [(1, 4, 'A'), (3, 5, 'B'), (0, 6, 'C'), (5, 7, 'D'), \
             (3, 8, 'E'), (5, 9, 'F'), (6, 10, 'G'), (8, 11, 'H'), \
             (8, 12, 'I'), (2, 13, 'J'), (12, 14, 'K')]
    ts = greedy(tasks)
```

```
22      for task in ts:
23          print(task)
24
25  if __name__ == '__main__':
26      main()
```

在上面的程序中，$tasks$ 是一个列表，它的每个元素是一个元组，分别表示每项任务的开始时刻 s_i、结束时刻 f_i 及任务名称。$tasks$ 已经按关键字 f_i 排序。该算法的效率很高，在已经排序的情况下，时间复杂度为 $O(n)$。考虑到未排序情况，时间复杂度为 $O(nlogn)$。

如果想要成功地应用贪心算法，得到问题的整体最优解，需要注意以下几点：

- 定义贪心准则

  即贪心算法的选择标准；

- 证明“整体最优解与子问题最优解具有一致性”；

- 依据贪心准则，证明问题具有贪心选择性质；

贪心准则的选取也是一个非常重要的因素。对于同一个问题，贪心准则选取不当，将可能使问题失去贪心选择性质，从而使得贪心算法得不到问题的整体最优解。

5.3.3 与动态规划的比较

动态规划与贪心算法，都要求问题具有整体最优解与局部最优解一致性的特点，这是它们的共同点。下面是两者之间的区别：

- 如果问题具有整体最优解与局部最优解一致性的特点，并且能够给出最优值的递归公式，那么动态规划算法一定能够找到整体最优解；而贪心算法还需要证明问题要具有贪心选择性质，才能找到整体最优解；

- 如果两者都能够找到整体最优解，那么贪心算法更简单直接，以自顶向下的方式工作；而动态规划更复杂些，以自底向上的方式工作；

- 一般而言，动态规划的适用范围更广些；贪心算法需要定义合理的贪心准则，否则，算法可能不能找到整体最优解；

下面以 0-1 背包问题与背包问题为例，讨论 2 种算法的具体差别。

0-1 背包问题在5.2.3中讨论过。下面给出背包问题的定义，它与 0-1 背包问题很相似，区别在于，选择物品装入背包时，可以选择：不装入、装入部分或全部装入。给定 $C>0$，$w_i>0$，$v_i>0$，$1\leq i\leq n$，要求找出 n 维向量 $(x_1,x_2,...,x_n)$，$0\leq x_i\leq 1$。它的形式化定义如下：

$$max\sum_{i=1}^{n}v_ix_i \tag{69}$$

$$\begin{cases}\sum_{i=1}^{n}w_ix_i\leq C\\ 0\leq x_i\leq 1,1\leq i\leq n\end{cases} \tag{70}$$

虽然 0-1 背包问题与背包问题都具有整体最优解与局部最优解一致性的特点，但是，背包问题可以由贪心算法求解，而 0-1 背包问题不能由贪心算法求解。

针对背包问题，首先，定义贪心准则：计算每种物品的单位重量的价值 $\frac{v_i}{w_i}$。然后，依据贪心选择策略，迭代地将单位重量价值最高的物品装入背包，直至背包装满为止。当然，为了证明贪心算法的确能够找到背包问题的整体最优解，还必须证明背包问题具有贪心选择性质。

对于 0-1 背包问题，这种贪心选择策略就不再适用了。主要原因在于，0-1 背包问题的性质已发生变化——物品要么装入，要么不装入。因此，此时的贪心算法无法保证背包最终能够被装满。一旦不能装满背包，那么部分闲置的背包空间将使得单位重量的价值发生变化，贪心算法也就不能保证能够找到整体的最优解了。

然而，对于动态规划而言，在考虑 0-1 背包问题时，将会比较选择物品 i 和不选择物品 i 所导致的最终方案，然后再做出最优的选择，而在此过程中，将会产生许多互相重叠的子问题。因此，动态规划算法能够有效地解决 0-1 背包问题。

5.4 回溯法

5.4.1 基本框架

严格来说，回溯法是一种直接在解空间中按深度优先搜索策略进行搜索的“通用型”算法。

在使用回溯法时,应该明确定义问题的解空间。例如,对于 0-1 背包问题,假设物品的数量为 n,那么解空间就是 n 维二值向量组成的向量空间。例如,当 $n=3$ 时,问

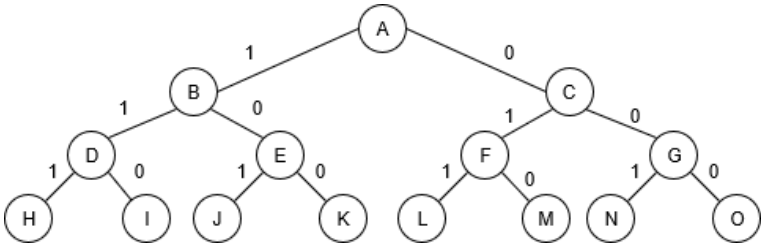

图 5-44: n=3 时 0-1 背包问题的解空间树

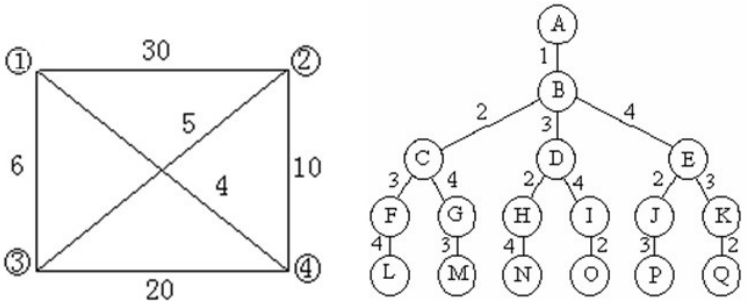

图 5-45: 旅行商问题的示例及解空间树

题的解空间是 $\{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)\}$。

在定义了解空间之后，需要按某种方式将解空间中的元素组织起来，以便能够让回溯法遍历整个解空间。一般情况下，可以将解空间组织成树或图的形式，如图5-44所示。从根节点到叶节点的路径定义了解空间中的一个向量。例如，从根节点 A 到叶节点 H 定义了一个解 $(1,1,1)$。解空间中叶节点的个数为 $O(2^n)$。

再来看另一个例子——旅行商问题。该问题描述的是，某商人要到若干个城市去推销商品，需要选择一条从所在地出发，只经过各个城市一次，最后返回所在地的路线，并且要使得总的旅费最少。

例如，图5-45中左图展示的是有 4 个城市的带权图，城市编号为 1、2、3、4，边权值为每 2 个城市之间的旅程代价。问题的求解目标是，在带权图中，要找出一条从节点 1 开始的、代价最少的周游路线。

该问题的解空间可以被组织成一棵树，如图5-45中右图所示。从树的根节点到叶节点定义了带权图的一条周游路线。例如，从根节点 A 到叶节点 L 定义了一条周游路线 1-2-3-4(-1)。解空间中叶节点的个数为 $O(n!)$。

通常情况下，回溯法涉及的解空间树有 2 种类型：子集树、排列树。

如果问题是从 n 个元素组成的集合 S 中，挑选出符合要求的子集时，那么解

空间就是由 n 维二值向量组成的向量空间，相应的解空间树被称为子集树。例如，n 个物品的 0-1 背包问题，它的解空间树就是一棵子集树。子集树是一棵完美二叉树，节点总数为 $2^{n+1}-1$，叶节点的个数为 2^n，遍历子集树的时间复杂度为 $O(2^n)$。

如果问题是确定 n 个元素中满足某种性质的排列时，那么解空间就是由 n 个元素的所有排列组成的空间，相应的解空间被称为排列树。排列树有 $O(n!)$ 个叶节点，遍历排列树的时间复杂度为 $O(n!)$。例如，旅行商问题的解空间就是一棵排列树。

回溯法在搜索解空间树时，通常采用 2 种策略避免无效的搜索，以提高回溯法的搜索效率。这 2 种方法分别是：

- 约束函数

  在扩展节点时，使用约束函数剪去不满足约束条件的子树。

  例如，如图5-44所示，对于 $n=3$ 的 0-1 背包问题，考虑下面的具体实例：$w=[16,15,15]$、$v=[45,25,25]$、$C=30$。从根节点 A 开始，选择节点 B，获得的价值为 45，背包剩余的容量为 $30-16=14$；在节点 B 处，可选扩展节点为 D 和 E，由于背包剩余容量只有 14，而扩展节点 D 需要消耗背包的容量为 15。因此，节点 D 不满足约束条件，可以被剪去；

- 限界函数

  在扩展节点时，使用限界函数剪去得不到最优解的子树。

  例如，如图5-45所示，左图已经标注了每条边的权值 (代价)。从根节点 A 出发，假设现在已经搜索了路径 $A-B-D-H-N$，获得了目前的最小代价 25，然后算法回溯到节点 D 处，准备扩展节点 I，此时的预估代价已经是 26，表明该路径已经不可能获得最小代价。因此，节点 I 不满足限界函数，可以被剪去。

下面给出回溯算法的基本框架：

```
def backtrack(root):
    if root is a goal:
        output(root)
    else:
        label root as discovered
```

```
        for child in root.getChild():
            if child is not labeled as discovered and
            constraint(child) and bound(child):
                backtrack(child)
```

该算法以深度优先方式搜索解空间，并在搜索的过程中，使用约束函数和限界函数 (统称为剪枝函数)，分别剪去不满足约束条件和无法得到最优解的分支。

5.5 分支限界法

5.5.1 基本框架

分支限界法 (Branch and Bound Algorithm) 类似于回溯法，也是在解空间树上直接搜索目标解。但是，它们之间有如下差别：

- 回溯法的求解目标是，找出解空间中满足约束条件的所有解，而分支限界法的求解目标是，找出满足约束条件的一个解，或者，在满足约束条件的解中，找出某种意义下的最优解；
- 两者的求解目标不同，导致它们的搜索方式也不相同。

  回溯法以深度优先的方式搜索解空间树，而分支限界法则以宽度优先或最低代价优先的方式搜索解空间树。分支限界法总是一次性地为扩展节点生成所有的子节点，舍弃掉那些不满足约束条件和限界函数的节点，然后在所有的可扩展节点中选择最有利的子节点进行扩展与搜索，以便尽快地找到一个最优解；

在实际应用中，可以选择普通队列 (FIFO) 或优先级队列存放待扩展节点。如果选择普通队列，则分支限界算法的搜索方式与宽度优先搜索方式非常相似，只是分支限界算法允许使用约束函数和限界函数 (参见5.4.1) 来舍弃一些不满足约束条件和无法获得最优解的子节点。

下面给出分支限界算法的基本框架：

```
def BBA(root):
    frontier = PriorityQueue()
    frontier.enqueue(root, 0)
```

```
while not frontier.is_empty():
    v = frontier.dequeue()
    if v is a goal:
        return v
    else:
        for child in v.getChild():
            cost = child.getCost()
            if constraint(child) and bound(child):
                frontier.put(child, cost)
return None
```

6 参考文献

1. Brad Miller, David Ranum. Problem Solving with Algorithms and Data Structures Using Python. Franklin Beedle & Associates. Sep. 22, 2013.

2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, Second Edition. The MIT Press. 2001.

3. 王晓东。算法设计与分析。清华大学出版社，2003 年 1 月第 1 版。

4. 严蔚敏，吴伟民。数据结构。清华大学出版社，1992 年。

5. John V.Guttag，梁杰译。编程导论，Introduction to Computation and Programming Using Python. 人民邮电出版社，2015。

6. 裘宗燕。数据结构与算法：Python 语言描述。机械工业出版社，2015 年 12 月。

7. Rance D. Necaise. Data Structures and Algorithms Using Python. John Wiley & Sons, Inc. 2011.

8. Marina von Steinkirch. An Introduction to Python & Algorithms. Summer, 2013.

9. Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. Data Structures & Algorithms in Python. John Wiley & Sons, Inc., 2013.

10. J Zelenski. Exhaustive recursion and backtracking. Feb. 1, 2008, https://see.stanford.edu/materials/icspacs106b/H19-RecBacktrackExamples.pdf.

11. 刘璟。计算机算法引论——设计与分析技术。科学出版社，2003 年 9 月。

12. Mark Allen Weiss. 冯舜玺，陈越译。数据结构与算法分析。机械工业出版社，2017 年 8 月。

13. 邹海明，余祥宣。计算机算法基础。华中理工大学出版社，1985 年 9 月第 1 版。

14. Priority Queue, https://en.wikipedia.org/wiki/Priority_queue.

15. Quicksort, https://en.wikipedia.org/wiki/Quicksort.

16. Merge Sort, https://en.wikipedia.org/wiki/Merge_sort.

17. Stirling's approximation, https://en.wikipedia.org/wiki/Stirling%27s_approximation.

18. Wikipedia: Zobrist hashing, https://en.wikipedia.org/wiki/Zobrist_hashing.