DSA-假期Week1 OOP, stack, queue, etc.

Updated 2306 GMT+8 Jan 20, 2025

2025 spring, Complied by Hongfei Yan

Logs:

> 假期按照4周计算，对应学期中4个月，即每周要完成1个月的工作量。计划是第一周线性结构（链表，stack, queue, Big-O，排序等），第二周树，第三周图，第四周综合。假期每日选做争取累计满100个，1/3是树的题目，1/3是图，其他占1/3。
>
> cs201数算（计算机基础2/2）2025pre每日选做，https://github.com/GMyhf/2025spring-cs201/blob/main/pre_problem_list_2025spring.md
>
> 《Python数据结构与算法分析》v2与v3基本一致的，新版程序更规范了。我们按照v3讲，书上不足的地方，会有其他材料补充。https://runestone.academy/ns/books/published/pythonds3/index.html

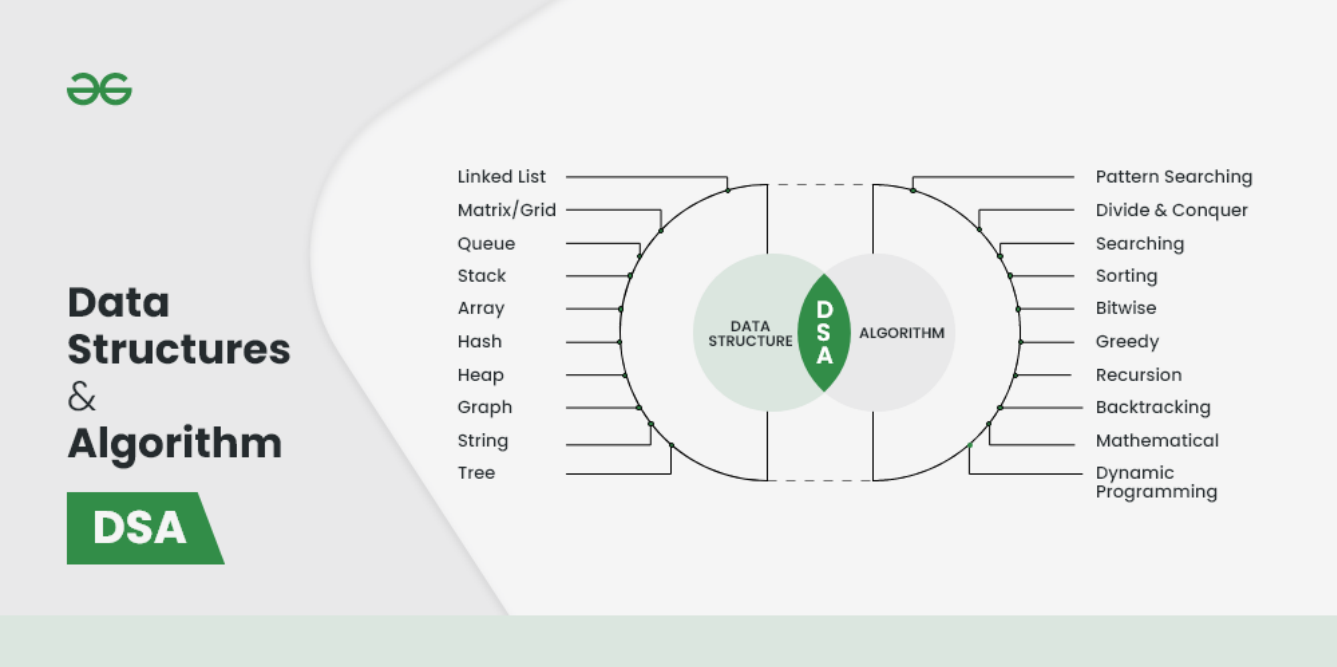
0 数据结构与算法DSA

https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/?ref=outind

What is DSA?

数据结构与算法（DSA）是两个独立却又紧密相连的领域相结合的产物。它被视为每位计算机科学学生必备的关键技能之一。实践中常常发现，那些对数据结构和算法有深刻理解的人往往比他人更具编程实力，也因此能够在众多求职者中脱颖而出，成功通过各大科技公司的面试。

> DSA is defined as a combination of two separate yet interrelated topics – Data Structure and Algorithms. DSA is one of the most important skills that every computer science student must have. It is often seen that people with good knowledge of these technologies are better programmers than others and thus, crack the interviews of almost every tech giant.



⚠️:

数据结构与算法（DS Algo）其他平行班确实会花很多时间讲dp, greedy, searching，但不是我们数算的重点，因为算法是计概课程（Algo DS）的主要内容，四大类的算法贪心greedy、递归recursion/回溯backtracking、动态规划dp、搜索searching，我们在计概中覆盖到了。如果同学递归题目做的少，可以找相关题目多练习。

What is Data Structure?

数据结构是指一种特定的存储和组织数据的方式，旨在我们的设备中高效且有效地管理和利用数据。采用数据结构的核心理念在于最小化时间和空间复杂度，即通过占用尽可能少的内存空间并以最短的时间来执行数据操作，从而实现高效的性能。

> A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexities. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

What is Algorithm?

算法是指为了解决特定类型的问题或执行某种特定计算而设计的一系列明确定义的步骤。用更简单的话来说，算法就是一组按部就班的操作，通过这些操作来完成一项具体的任务。

> Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

How to start learning DSA?

首先要做的是将整个学习过程分解为一系列需按顺序完成的小任务。从零开始系统地学习数据结构与算法（DSA）的过程可以分为以下四个阶段：

1. **理解时间和空间复杂度**：掌握评估算法效率的关键概念。

2. **学习各数据结构的基础**：熟悉不同数据结构的特点和使用场景。
3. **掌握算法的基础知识**：了解常用算法的工作原理及其应用。
4. **练习DSA相关的题目**：通过实践巩固所学知识，提高解决问题的能力。

> The first and foremost thing is dividing the total procedure into little pieces which need to be done sequentially. The complete process to learn DSA from scratch can be broken into 4 parts:
>
> 1. Learn about Time and Space complexities
> 2. Learn the basics of individual Data Structures
> 3. Learn the basics of Algorithms
> 4. Practice Problems on DSA

逻辑视图、物理视图和数据结构

计算机科学并不仅是研究计算机本身。尽管计算机在这一学科中是非常重要的工具，但也仅仅只是工具而已。计算机科学的研究对象是问题、解决问题的过程，以及通过该过程得到的解决方案。给定一个问题，计算机科学家的目标是开发一个能够逐步解决该问题的**算法**。算法是具有有限步骤的过程，依照这个过程便能解决问题。因此，算法就是解决方案。

可以认为计算机科学就是研究算法的学科。但是必须注意，某些问题并没有解决方案。可以将计算机科学更完善地定义为：研究问题及其解决方案，以及研究目前无解的问题的学科。

在研究问题解决过程的同时，计算机科学也研究**抽象**。抽象思维使得我们能分别从逻辑视角和物理视角来看待问题及其解决方案。举一个常见的例子。

试想你每天开车去上学或上班。作为车的使用者，你在驾驶时会与它有一系列的交互：坐进车里，插入钥匙，启动发动机，换挡，刹车，加速以及操作方向盘。从抽象的角度来看，这是从逻辑视角来看待这辆车，你在使用由汽车设计者提供的功能来将自己从某个地方运送到另一个地方。这些功能有时候也被称作**接口**。

另一方面，修车工看待车辆的角度与司机截然不同。他不仅需要知道如何驾驶，而且更需要知道实现汽车功能的所有细节：发动机如何工作，变速器如何换挡，如何控制温度，等等。这就是所谓的物理视角，即看到表面之下的实现细节。

使用计算机也是如此。大多数人用计算机来写文档、收发邮件、浏览网页、听音乐、存储图像以及打游戏，但他们并不需要了解这些功能的实现细节。大家都是从逻辑视角或者使用者的角度来看待计算机。计算机科学家、程序员、技术支持人员以及系统管理员则从另一个角度来看待计算机。他们必须知道操作系统的原理、网络协议的配置，以及如何编写各种脚本来控制计算机。他们必须能够控制用户不需要了解的底层细节。

上面两个例子的共同点在于，用户（或称客户）只需要知道接口是如何工作的，而并不需要知道实现细节。这些接口是用户用于与底层复杂的实现进行交互的方式。

1 为何学习数据结构及抽象数据类型

为了控制问题及其求解过程的复杂度，计算机科学家利用抽象来帮助自己专注于全局，从而避免迷失在众多细节中。通过对问题进行建模，可以更高效地解决问题。模型可以帮助计算机科学家更一致地描述算法要用到的数据。

如前所述，过程抽象将功能的实现细节隐藏起来，从而使用户能从更高的视角来看待功能。**数据抽象**的基本思想与此类似。**抽象数据类型**（有时简称为ADT）从逻辑上描述了如何看待数据及其对应运算而无须考虑具体实现。这意味着我们仅需要关心数据代表了什么，而可以忽略它们的构建方式。通过这样的抽象，我们对数据进行了一层封装，其基本思想是封装具体的实现细节，使它们对用户不可见，这被称为信息隐藏。

图1-1展示了抽象数据类型及其原理。用户通过抽象数据类型提供的操作来与接口交互。抽象数据类型是与用户交互的外壳。真正的实现则隐藏在内部。用户并不需要关心各种实现细节。

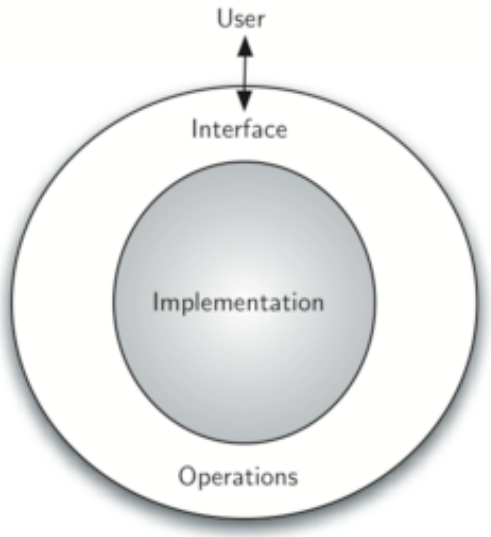


图1-1 抽象数据类型

抽象数据类型的实现，通常被称为**数据结构**。它是**物理视图**和**逻辑视图**之间的桥梁，因为数据结构既定义了数据的逻辑特性（如栈或队列的操作），也决定了这些特性的具体实现方式（如使用数组或链表）。逻辑视图和物理视图的分离使我们能够在不透露模型实际构建细节的情况下定义复杂的数据模型来解决相应的问题。这提供了与实现无关的逻辑视图。由于通常会有多种不同的方法来实现一个抽象数据类型，这种实现独立性允许程序员在不改变用户与数据交互方式的前提下切换实现的具体细节。因此，用户可以专注于问题解决的过程。
The implementation of an abstract data type, often referred to as a **data structure**, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types. As we discussed earlier, the separation of these two perspectives will allow us to define the complex data models for our problems without giving any indication as to the details of how the model will actually be built. This provides an **implementation-independent** view of the data. Since there will usually be many different ways to implement an abstract data type, this implementation independence allows the programmer to switch the details of the implementation without changing the way the user of the data interacts with it. The user can remain focused on the problem-solving process.

> 在计算机科学和软件工程中，“物理视图”、“数据视图”和“数据结构”是三个相关但含义不同的术语。
>
> **物理视图 (Physical View)**：
>
> - 物理视图指的是数据如何在计算机内存或存储设备上实际组织和表示的细节。它涉及到具体的数据布局、存储格式、访问方法等。例如，在实现一个数组时，物理视图会包括数组元素在内存中的连续存储方式，或者在一个链表中节点是如何通过指针相互链接的。
>
> **逻辑视图（也称为数据视图, Logical or Data View）**：

> - 逻辑视图（或数据视图）是指从用户或应用程序的角度看到的数据组织形式。它是抽象数据类型（ADT）的一部分，描述了数据应该具有的行为和操作，而不涉及这些操作是如何具体实现的。换句话说，逻辑视图定义了数据模型的功能特性，而不是其实现细节。例如，栈（stack）的逻辑视图只关心压入（push）和弹出（pop）操作，而不需要知道内部是用数组还是链表来实现。
>
> **数据结构 (Data Structure)**：
>
> - 数据结构是用来组织、管理和存储数据的方式，以便能够高效地访问和修改数据。它可以看作是物理视图和逻辑视图之间的桥梁，因为数据结构既定义了数据的逻辑特性（如栈或队列的操作），也决定了这些特性的具体实现方式（如使用数组或链表）。因此，数据结构的选择直接影响到程序的性能和复杂度。
>
> 原文中的 "physical view of the data" 更贴近于“数据的物理视图”，即关注的是数据在底层硬件上的表示和存储方式；而“implementation-independent view of the data” 则更倾向于描述“逻辑视图”或“数据视图”，强调的是与具体实现无关的数据抽象。

```
数据结构
├── 逻辑结构
│   ├── 集合结构
│   ├── 线性结构：线性表、栈、队列、串
│   ├── 非线性结构：数组、树、堆、图
│   └── 树形结构
└── 物理结构
    ├── 顺序存储
    ├── 链式存储
    ├── 索引存储 (Indexing)：B树、B+树等，适合有序数据和范围查询
    └── 散列存储 (Hashing)：在单一键值查找上表现优异
```

2 为何学习算法

计算机科学家通过经验来学习：观察他人如何解决问题，然后亲自解决问题。接触各种问题解决技巧并学习不同算法的设计方法，有助于解决新的问题。通过学习一系列不同的算法，可以举一反三，从而在遇到类似的问题时，能够快速加以解决。

各种算法之间往往差异巨大。例如求平方根的例子，完全可能有多种方法来实现计算平方根的函数。算法一可能使用了较少的资源，算法二返回结果所需的时间可能是算法一的10倍。我们需要某种方式来比较这两种算法。尽管这两种算法都能得到结果，但是其中一种可能比另一种“更好”——更高效、更快，或者使用的内存更少。随着对算法的进一步学习，你会掌握比较不同算法的分析技巧。这些技巧只依赖于算法本身的特性，而不依赖于程序或者实现算法的计算机的特性。

最坏的情况是遇到难以解决的问题，即没有算法能够在合理的时间内解决该问题。因此，至关重要的一点是，要能区分有解的问题、无解的问题，以及虽然有解但是需要过多的资源和时间来求解的问题。

在选择算法时，经常会有所权衡。除了有解决问题的能力之外，计算机科学家也需要知晓如何评估一个解决方案。总之，问题通常有很多解决方案，如何找到一个解决方案并且确定其为优秀的方案，是需要反复练习、熟能生巧的。

1 Python基础及OOP

本节为之前提到的思想提供更详细的例子。目标是复习Python并且强化一些会在后续各章中变得非常重要的概念。
Python是一门现代、易学、面向对象的编程语言。它拥有强大的内建数据类型以及简单易用的控制语句。由于Python是一门解释型语言，因此只需要查看和描述交互式会话就能进行学习。解释器会显示提示符>>>，然后计算你提供的Python语句。例如，以下代码显示了提示符、print函数、结果，以及下一个提示符。

```
>>> print("Algorithms and Data Structures")
>>> Algorithms and Data Structures
>>>
```

1.1 数据

Python支持面向对象编程范式。这意味着Python认为数据是问题解决过程中的关键点。在Python以及其他所有面向对象编程语言中，类都是对数据的构成（状态）以及数据能做什么（行为）的描述。由于类的使用者只能看到数据项的状态和行为，因此类与抽象数据类型是相似的。在面向对象编程范式中，数据项被称作对象。一个对象就是类的一个实例。

1 内建原子数据类型

我们首先看原子数据类型。Python有两大内建数据类实现了整数类型和浮点数类型，相应的Python类就是int和float。标准的数学运算符，即+、-、＊、/以及＊＊（幂），可以和能够改变运算优先级的括号一起使用。其他非常有用的运算符包括取余（取模）运算符%，以及整除运算符//。注意，当两个整数相除时，其结果是一个浮点数，而整除运算符截去小数部分，只返回商的整数部分。

Python通过bool类实现对表达真值非常有用的布尔数据类型。布尔对象可能的状态值是True或者False，布尔运算符有and、or以及not。

布尔对象也被用作相等（==）、大于（>）等比较运算符的计算结果。此外，结合使用关系运算符与逻辑运算符可以表达复杂的逻辑问题。表2-1展示了关系运算符和逻辑运算符。

表2-1 关系运算符和逻辑运算符

Operation Name	Operator	Explanation
less than	<	Less than operator
greater than	>	Greater than operator
less than or equal	<=	Less than or equal to operator
greater than or equal	>=	Greater than or equal to operator
equal	==	Equality operator
not equal	!=	Not equal operator
logical and	and	Both operands True for result to be True
logical or	or	One or the other operand is True for the result to be True
logical not	not	Negates the truth value, False becomes True, True becomes False

标识符在编程语言中被用作名字。Python中的标识符以字母或者下划线（_）开头，区分大小写，可以是任意长度。采用能表达含义的名字是良好的编程习惯，这使程序代码更易阅读和理解。

当一个名字第一次出现在赋值语句的左边部分时，会创建对应的Python变量。赋值语句将名字与值关联起来。变量存的是指向数据的引用，而不是数据本身。

赋值语句改变了变量的引用，这体现了Python的动态特性。同样的变量可以指向许多不同类型的数据。

2 内建集合数据类型

除了数值类和布尔类，Python还有众多强大的内建集合类。列表、字符串以及元组是概念上非常相似的有序集合，但是只有理解它们的差别，才能正确运用。集（set）和字典是无序集合。

列表是零个或多个指向Python数据对象的引用的有序集合，通过在方括号内以逗号分隔的一系列值来表达。空列表就是[ ]。列表是异构的，这意味着其指向的数据对象不需要都是同一个类，并且这一集合可以被赋值给一个变量。

由于列表是有序的，因此它支持一系列可应用于任意Python序列的运算，如表2-2所示。

表2-2 可应用于任意Python序列的运算

Operation Name	Operator	Explanation
indexing	[ ]	Access an element of a sequence
concatenation	+	Combine sequences together
repetition	*	Concatenate a repeated number of times
membership	in	Ask whether an item is in a sequence
length	len	Ask the number of items in the sequence
slicing	[:]	Extract a part of a sequence

需要注意的是，列表和序列的下标从0开始。myList[1:3]会返回一个包含下标从1到2的元素列表（并没有包含下标为3的元素）。
如果需要快速初始化列表，可以通过重复运算来实现，如下所示。

```
>>> myList = [0] * 6
>>> myList
[0, 0, 0, 0, 0, 0]
```

非常重要的一点是，重复运算返回的结果是序列中指向数据对象的引用的重复。下面的例子可以很好地说明这一点。

```
>>> myList = [1,2,3,4]
>>> A = [myList]*3
>>> A
[[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
>>> myList[2] = 45
>>> A
[[1, 2, 45, 4], [1, 2, 45, 4], [1, 2, 45, 4]]
```

变量A包含3个指向myList的引用。myList中的一个元素发生改变，A中的3处都随即改变。
列表支持一些用于构建数据结构的方法，如表2-3所示。后面的例子展示了用法。

表2-3 Python列表提供的方法

Method Name	Use	Explanation
`append`	`alist.append(item)`	Adds a new item to the end of a list
`insert`	`alist.insert(i,item)`	Inserts an item at the ith position in a list
`pop`	`alist.pop()`	Removes and returns the last item in a list
`pop`	`alist.pop(i)`	Removes and returns the ith item in a list
`sort`	`alist.sort()`	Modifies a list to be sorted
`reverse`	`alist.reverse()`	Modifies a list to be in reverse order
`del`	`del alist[i]`	Deletes the item in the ith position
`index`	`alist.index(item)`	Returns the index of the first occurrence of `item`
`count`	`alist.count(item)`	Returns the number of occurrences of `item`
`remove`	`alist.remove(item)`	Removes the first occurrence of `item`

你会发现，像pop这样的方法在返回值的同时也会修改列表的内容，reverse等方法则仅修改列表而不返回任何值。pop默认返回并删除列表的最后一个元素，但是也可以用来返回并删除特定的元素。这些方法默认下标从0开始。你也会注意到那个熟悉的句点符号，它被用来调用某个对象的方法。

range是一个常见的Python函数，我们常把它与列表放在一起讨论。range会生成一个代表值序列的范围对象。使用list函数，能够以列表形式看到范围对象的值。

范围对象表示整数序列。默认情况下，它从0开始。如果提供更多的参数，它可以在特定的点开始和结束，并且跳过中间的值。在第一个例子中，range(10)从0开始并且一直到9为止（不包含10）；在第二个例子中，range(5,10)从5开始并且到9为止（不包含10）；range(5,10,2)的结果类似，但是元素的间隔变成了2（10还是没有包含在其中）。

字符串是零个或多个字母、数字和其他符号的有序集合。这些字母、数字和其他符号被称为字符。常量字符串值通过引号（单引号或者双引号均可）与标识符进行区分。

由于字符串是序列，因此之前提到的所有序列运算符都能用于字符串。此外，字符串还有一些特有的方法，表2-4列举了其中一些。

表2-4 Python字符串提供的方法

Method Name	Use	Explanation
`center`	`astring.center(w)`	Returns a string centered in a field of size `w`
`count`	`astring.count(item)`	Returns the number of occurrences of `item` in the string
`ljust`	`astring.ljust(w)`	Returns a string left-justified in a field of size `w`
`lower`	`astring.lower()`	Returns a string in all lowercase
`rjust`	`astring.rjust(w)`	Returns a string right-justified in a field of size `w`
`find`	`astring.find(item)`	Returns the index of the first occurrence of `item`
`split`	`astring.split(schar)`	Splits a string into substrings at `schar`

split在处理数据的时候非常有用。split接受一个字符串，并且返回一个由分隔字符作为分割点的字符串列表。

列表和字符串的主要区别在于，列表能够被修改，字符串则不能。列表的这一特性被称为可修改性。列表具有可修改性，字符串则不具有。

由于都是异构数据序列，因此元组与列表非常相似。它们的区别在于，元组和字符串一样是不可修改的。元组通常写成由括号包含并且以逗号分隔的一系列值。与序列一样，元组允许之前描述的任一操作。

集合（set）是由零个或多个不可修改的Python数据对象组成的无序集合。集不允许重复元素，并且写成由花括号包含、以逗号分隔的一系列值。空集由set()来表示。集是异构的。

尽管集是无序的，但它还是支持之前提到的一些运算，如表2-5所示。

表2-5 Python集合支持的运算

Operation Name	Operator	Explanation
membership	in	Set membership
length	len	Returns the cardinality of the set
`|`	`aset | otherset`	Returns a new set with all elements from both sets
`&`	`aset & otherset`	Returns a new set with only those elements common to both sets
`-`	`aset - otherset`	Returns a new set with all items from the first set not in second
`<=`	`aset <= otherset`	Asks whether all elements of the first set are in the second

集支持一系列方法，如表2-6所示。在数学中运用过集合概念的人应该对它们非常熟悉。Note that `union`, `intersection`, `issubset`, and `difference` all have operators that can be used as well.

Note that `union`, `intersection`, `issubset`, and `difference` all have operators that can be used as well.

表2-6 Python集合提供的方法

Method Name	Use	Explanation
`union`	`aset.union(otherset)`	Returns a new set with all elements from both sets
`intersection`	`aset.intersection(otherset)`	Returns a new set with only those elements common to both sets
`difference`	`aset.difference(otherset)`	Returns a new set with all items from first set not in second
`issubset`	`aset.issubset(otherset)`	Asks whether all elements of one set are in the other
`add`	`aset.add(item)`	Adds item to the set
`remove`	`aset.remove(item)`	Removes item from the set
`pop`	`aset.pop()`	Removes an arbitrary element from the set
`clear`	`aset.clear()`	Removes all elements from the set

字典是无序结构，由相关的元素对构成，其中每对元素都由一个键和一个值组成。这种键-值对通常写成键：值的形式。字典由花括号包含的一系列以逗号分隔的键-值对表达，

可以通过键访问其对应的值，也可以向字典添加新的键-值对。访问字典的语法与访问序列的语法十分相似，只不过是使用键来访问，而不是下标。添加新值也类似。

需要谨记，字典并不是根据键来进行有序维护的。键的位置是由散列来决定的，后续章节会详细介绍散列。len函数对字典的功能与对其他集合的功能相同。

字典既有运算符，又有方法。表2-7和表2-8分别展示了它们。keys、values和items方法均会返回包含相应值的对象。可以使用list函数将字典转换成列表。在表2-8中可以看到，get方法有两种版本。如果键没有出现在字典中，get会返回None。然而，第二个可选参数可以返回特定值。

表2-7 Python字典支持的运算

Operator	Use	Explanation
`[]`	`myDict[k]`	Returns the value associated with `k`, otherwise its an error
`in`	`key in adict`	Returns `True` if key is in the dictionary, `False` otherwise
`del`	del `adict[key]`	Removes the entry from the dictionary

表2-8 Python字典提供的方法

Method Name	Use	Explanation
`keys`	`adict.keys()`	Returns the keys of the dictionary in a dict_keys object
`values`	`adict.values()`	Returns the values of the dictionary in a dict_values object
`items`	`adict.items()`	Returns the key-value pairs in a dict_items object
`get`	`adict.get(k)`	Returns the value associated with `k`, `None` otherwise
`get`	`adict.get(k,alt)`	Returns the value associated with `k`, `alt` otherwise

1.2基本语法

1 输入与输出

程序经常需要与用户进行交互，以获得数据或者提供某种结果。目前的大多数程序使用对话框作为要求用户提供某种输入的方式。尽管Python确实有方法来创建这样的对话框，但是可以利用更简单的函数。Python提供了一个函数，它使得我们可以要求用户输入数据并且返回一个字符串的引用。这个函数就是input。

input函数接受一个字符串作为参数。由于该字符串包含有用的文本来提示用户输入，因此它经常被称为提示字符串。

不论用户在提示字符串后面输入什么内容，都会被存储在aName变量中。使用input函数，可以非常简便地写出程序，让用户输入数据，然后再对这些数据进行进一步处理。

需要注意的是，input函数返回的值是一个字符串，它包含用户在提示字符串后面输入的所有字符。如果需要将这个字符串转换成其他类型，必须明确地提供类型转换。

格式化字符串
print函数为输出Python程序的值提供了一种非常简便的方法。它接受零个或者多个参数，并且将单个空格作为默认分隔符来显示结果。通过设置sep这一实际参数可以改变分隔符。此外，每一次打印都默认以换行符结尾。这一行为可以通过设置实际参数end来更改。

更多地控制程序的输出格式经常十分有用。幸运的是，Python提供了另一种叫作格式化字符串的方式。格式化字符串是一个模板，其中包含保持不变的单词或空格，以及之后插入的变量的占位符。

2 控制结构

算法需要两个重要的控制结构：迭代和分支。Python通过多种方式支持这两种控制结构。程序员可以根据需要选择最有效的结构。

对于迭代，Python提供了标准的while语句以及非常强大的for语句。while语句会在给定条件为真时重复执行一段代码。

分支语句允许程序员进行询问，然后根据结果，采取不同的行动。绝大多数的编程语言都提供两种有用的分支结构：if else和if。

和其他所有控制结构一样，分支结构支持嵌套，一个问题的结果能帮助决定是否需要继续问下一个问题。

另一种表达嵌套分支的语法是使用elif关键字。将else和下一个if结合起来，可以减少额外的嵌套层次。注意，最后的else仍然是必需的，它用来在所有分支条件都不满足的情况下提供默认分支。

列表可以可以不通过迭代结构和分支结构来创建，这种方式被称为列表解析式。通过列表解析式，可以根据一些处理和分支标准轻松创建列表。

> Returning to lists, there is an alternative method for creating a list that uses iteration and selection constructs known as a **list comprehension**. A list comprehension allows you to easily create a list based on some processing or selection criteria.

3 异常处理

在编写程序时通常会遇到两种错误。第一种是语法错误，也就是说，程序员在编写语句或者表达式时出错。

第二种是逻辑错误，即程序能执行完成但返回了错误的结果。这可能是由于算法本身有错，或者程序员没有正确地实现算法。有时，逻辑错误会导致诸如除以0、越界访问列表等非常严重的情况。这些逻辑错误会导致运行时错误，进而导致程序终止运行。通常，这些运行时错误被称为异常。

许多初级程序员简单地把异常等同于引起程序终止的严重运行时错误。然而，大多数编程语言都提供了让程序员能够处理这些错误的方法。此外，程序员也可以在检测到程序执行有问题的情况下自己创建异常。
当异常发生时，我们称程序“抛出”异常。可以用try语句来“处理”被抛出的异常。

4 定义函数

通常来说，可以通过定义函数来隐藏任何计算的细节。函数的定义需要一个函数名、一系列参数以及一个函数体。函数也可以显式地返回一个值。

1.3 面向对象编程

Python是一门面向对象的编程语言。到目前为止，我们已经使用了一些内建的类来展示数据和控制结构的例子。面向对象编程语言最强大的一项特性是允许程序员（问题求解者）创建全新的类来对求解问题所需的数据进行建模。

我们之前使用了抽象数据类型来对数据对象的状态及行为进行逻辑描述。通过构建能实现抽象数据类型的类，可以利用抽象过程，同时为真正在程序中运用抽象提供必要的细节。每当需要实现抽象数据类型时，就可以创建新类。

1 Fraction类

要展示如何实现用户定义的类，一个常用的例子是构建实现抽象数据类型Fraction的类。我们已经看到，Python提供了很多数值类。但是在有些时候，需要创建“看上去很像”分数的数据对象。

像 $\frac{3}{5}$ 这样的分数由两部分组成。上面的值称作分子，可以是任意整数。下面的值称作分母，可以是任意大于0的整数（负的分数带有负的分子）。尽管可以用浮点数来近似表示分数，但我们在此希望能精确表示分数的值。

Fraction对象的表现应与其他数值类型一样。我们可以针对分数进行加、减、乘、除等运算，也能够使用标准的斜线形式来显示分数，比如3/5。此外，所有的分数方法都应该返回结果的最简形式。这样一来，不论进行何种运算，最后的结果都是最简分数。

在Python中定义新类的做法是，提供一个类名以及一整套与函数定义语法类似的方法定义。以下是一个方法定义框架。

> 在 Python 中，新创建的类默认继承自 `object` 类。`object` 是所有类的基类，也称为顶级基类或根类。这意味着在 Python 中，如果没有显式指定一个类的基类，它将自动成为 `object` 类的子类。
>
> 在类中定义的方法（包括 `__str__` 方法）的第一个参数应该是 self，它表示对当前对象的引用。这样，Python 在调用方法时会自动将该对象作为第一个参数传递给方法。
>
> 当我们在多个类中定义了 `__str__` 方法时，通过 self 参数，Python 可以确定应该执行哪个类的 `__str__` 方法。

```
class Fraction

    #the methods go here
```

所有类都应该首先提供构造方法。构造方法定义了数据对象的创建方式。要创建一个Fraction对象，需要提供分子和分母两部分数据。在Python中，构造方法总是命名为`__init__`（即在init的前后分别有两个下划线）。

代码 Fraction类及其构造方法

```
def __init__(self, top, bottom):

    self.num = top
    self.den = bottom
```

注意，形式参数列表包含3项。self是一个总是指向对象本身的特殊参数，它必须是第一个形式参数。然而，在调用方法时，从来不需要提供相应的实际参数。如前所述，分数需要分子与分母两部分状态数据。构造方法中的self.num定义了Fraction对象有一个叫作num的内部数据对象作为其状态的一部分。同理，self.den定义了分母。这两个实际参数的值在初始时赋给了状态，使得新创建的Fraction对象能够知道其初始值。

要创建Fraction类的实例，必须调用构造方法。使用类名并且传入状态的实际值就能完成调用（注意，不要直接调用`__init__`）。

```
myfraction = Fraction(3,5)
```

以上代码创建了一个对象，名为myfraction，值为3/5。图2-1展示了这个对象。

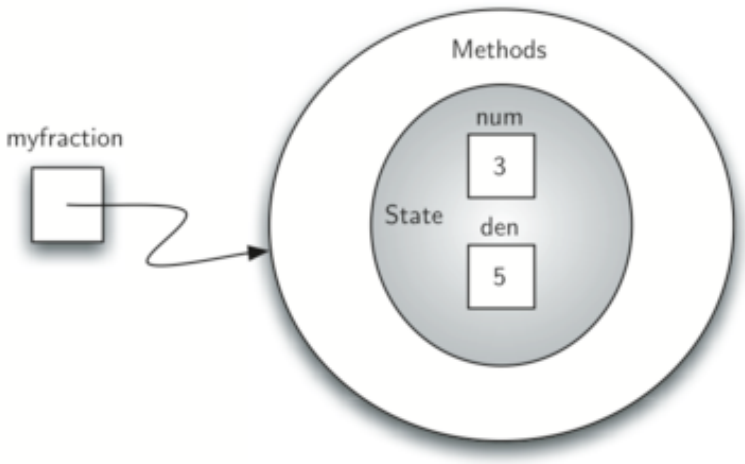


图2-1 Fraction类的一个实例

接下来实现这一抽象数据类型所需要的行为。考虑一下，如果试图打印Fraction对象，会发生什么呢?

```
>>> myf = Fraction(3,5)
>>> print(myf)
<__main__.Fraction instance at 0x409b1acc>
```

Fraction对象myf并不知道如何响应打印请求。print函数要求对象将自己转换成一个可以被写到输出端的字符串。myf唯一能做的就是显示存储在变量中的实际引用（地址本身）。这不是我们想要的结果。

有两种办法可以解决这个问题。一种是定义一个show方法，使得Fraction对象能够将自己作为字符串来打印。代码show方法展示了该方法的实现细节。如果像之前那样创建一个Fraction对象，可以要求它显示自己（或者说，用合适的格式将自己打印出来）。不幸的是，这种方法并不通用。为了能正确打印，我们需要告诉Fraction类如何将自己转换成字符串。要完成任务，这是print函数所必需的。

代码 show方法

```
def show(self):
     print(self.num,"/",self.den)

>>> myf = Fraction(3,5)
>>> myf.show()
3 / 5
>>> print(myf)
<__main__.Fraction instance at 0x40bce9ac>
>>>
```

Python的所有类都提供了一套标准方法，但是可能没有正常工作。其中之一就是将对象转换成字符串的方法`__str__`。这个方法的默认实现是像我们之前所见的那样返回实例的地址字符串。我们需要做的是为这个方法提供一个“更好”的实现，即重写默认实现，或者说重新定义该方法的行为。

为了达到这一目标，仅需定义一个名为`__str__`的方法，并且提供新的实现。除了特殊参数self之外，该方法定义不需要其他信息。新的方法通过将两部分内部状态数据转换成字符串并在它们之间插入字符/来将分数对象转换成字符串。一旦要求Fraction对象转换成字符串，就会返回结果。注意该方法的各种用法。

代码`__str__`方法

```
def __str__(self):
    return str(self.num)+"/"+str(self.den)
>>> myf = Fraction(3,5)
>>> print(myf)
3/5
>>> print("I ate", myf, "of the pizza")
I ate 3/5 of the pizza
>>> myf.__str__()
'3/5'
>>> str(myf)
'3/5'
>>>
```

可以重写Fraction类中的很多其他方法，其中最重要的一些是基本的数学运算。我们想创建两个Fraction对象，然后将它们相加。目前，如果试图将两个分数相加，会得到下面的结果。

```
>>> f1 = Fraction(1,4)
>>> f2 = Fraction(1,2)
f1+f2

Traceback (most recent call last):
  File "<pyshell#173>", line 1, in -toplevel-
    f1+f2
TypeError: unsupported operand type(s) for +:
          'instance' and 'instance'
>>>
```

如果仔细研究这个错误，会发现加号+无法处理Fraction的操作数。
可以通过重写Fraction类的`__add__`方法来修正这个错误。该方法需要两个参数。第一个仍然是self，第二个代表了表达式中的另一个操作数。

```
f1.__add__(f2)
```

以上代码会要求Fraction对象f1将Fraction对象f2加到自己的值上。可以将其写成标准表达式：f1 + f2。
两个分数需要有相同的分母才能相加。确保分母相同最简单的方法是使用两个分母的乘积作为分母。

$$\frac{a}{b}+\frac{c}{d}=\frac{ad}{bd}+\frac{cb}{bd}=\frac{ad+cb}{bd}$$

代码`__add__`方法返回一个包含分子和分母的新Fraction对象。可以利用这一方法来编写标准的分数数学表达式，将加法结果赋给变量，并且打印结果。值得注意的是，第3行中的\称作续行符。当一条Python语句被分成多行时，需要用到续行符。

代码`__add__`方法

```
def __add__(self,otherfraction):

    newnum = self.num*otherfraction.den + \
                self.den*otherfraction.num
    newden = self.den * otherfraction.den

    return Fraction(newnum,newden)
>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
6/8
>>>
```

虽然这一方法能够与我们预想的一样执行加法运算，但是还有一处可以改进。1/4+1/2的确等于6/8，但它并不是最简分数。最好的表达应该是3/4。为了保证结果总是最简分数，需要一个知道如何化简分数的辅助方法。该方法需要寻找分子和分母的最大公因数（greatest common divisor, GCD），然后将分子和分母分别除以最大公因数，最后的结果就是最简分数。

要寻找最大公因数，最著名的方法就是欧几里得算法，第8章将详细讨论。欧几里得算法指出，对于整数m和n，如果m能被n整除，那么它们的最大公因数就是n。然而，如果m不能被n整除，那么结果是n与m除以n的余数的最大公因数。代码gcd函数提供了一个迭代实现。注意，这种实现只有在分母为正的时候才有效。对于Fraction类，这是可以接受的，因为之前已经定义过，负的分数带有负的分子，其分母为正。

代码 gcd函数

```
def gcd(m,n):
    while m%n != 0:
        oldm = m
        oldn = n

        m = oldn
        n = oldm%oldn
    return n

print(gcd(20,10))
```

> import math
>
> print(math.gcd(16,12))

现在可以利用这个函数来化简分数。为了将一个分数转化成最简形式，需要将分子和分母都除以它们的最大公因数。对于分数6/8，最大公因数是2。因此，将分子和分母都除以2，便得到3/4。

代码 改良版`__add__`方法

```
def __add__(self,otherfraction):
    newnum = self.num*otherfraction.den + self.den*otherfraction.num
    newden = self.den * otherfraction.den
    common = gcd(newnum,newden)
    return Fraction(newnum//common,newden//common)

>>> f1=Fraction(1,4)
>>> f2=Fraction(1,2)
>>> f3=f1+f2
>>> print(f3)
3/4
>>>
```

Fraction对象现在有两个非常有用的方法，如图2-2所示。为了允许两个分数互相比较，还需要添加一些方法。假设有两个Fraction对象，f1和f2。只有在它们是同一个对象的引用时，f1 == f2才为True。这被称为浅相等，如图2-3所示。在当前实现中，分子和分母相同的两个不同的对象是不相等的。

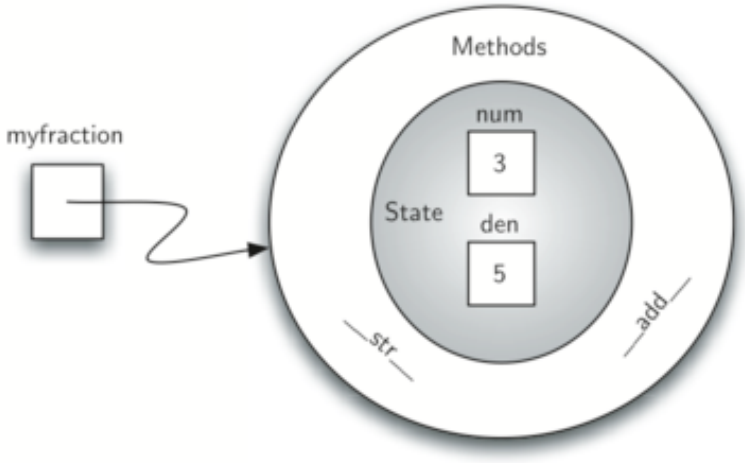


图2-2 包含两个方法的Fraction实例

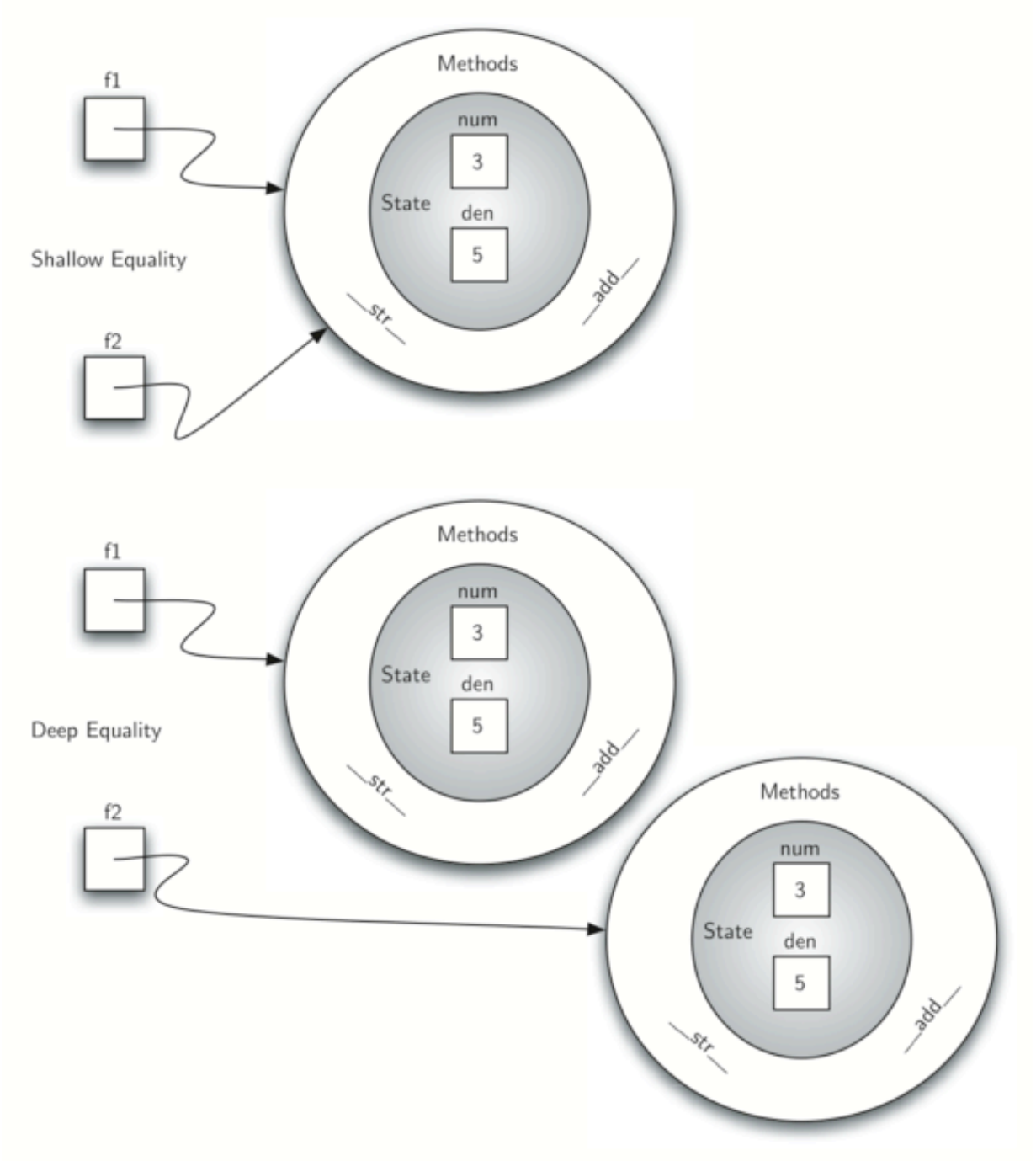


图2-3 浅相等与深相等

通过重写`__eq__`方法，可以建立深相等——根据值来判断相等，而不是根据引用。`__eq__`是又一个在任意类中都有的标准方法。它比较两个对象，并且在它们的值相等时返回True，否则返回False。
在Fraction类中，可以通过统一两个分数的分母并比较分子来实现`__eq__`方法，如代码清单1所示。需要注意的是，其他的关系运算符也可以被重写。例如，`__le__`方法提供判断小于等于的功能。

代码`__eq__`方法

```
def __eq__(self, other):
    firstnum = self.num * other.den
    secondnum = other.num * self.den

    return firstnum == secondnum
```

到目前为止我们完成了Fraction类的完整实现。剩余的算术方法及关系方法留作练习。

代码 Fraction类的完整实现

```
def gcd(m,n):
    while m%n != 0:
        oldm = m
        oldn = n

        m = oldn
        n = oldm%oldn
    return n

class Fraction:
     def __init__(self,top,bottom):
         self.num = top
         self.den = bottom

     def __str__(self):
         return str(self.num)+"/"+str(self.den)

     def show(self):
         print(self.num,"/",self.den)

     def __add__(self,otherfraction):
         newnum = self.num*otherfraction.den + \
                     self.den*otherfraction.num
         newden = self.den * otherfraction.den
         common = gcd(newnum,newden)
         return Fraction(newnum//common,newden//common)

     def __eq__(self, other):
         firstnum = self.num * other.den
         secondnum = other.num * self.den

         return firstnum == secondnum

x = Fraction(1,2)
y = Fraction(2,3)
print(x+y)
print(x == y)
```

2 继承Inheritance

最后一节介绍面向对象编程的另一个重要方面。继承使一个类与另一个类相关联，就像人们相互联系一样。孩子从父母那里继承了特征。与之类似，Python中的子类可以从父类继承特征数据和行为。父类也称为超类。

图2-4展示了内建的Python集合类以及它们的相互关系。我们将这样的关系结构称为继承层次结构。举例来说，列表是有序集合的子。因此，我们将列表称为子，有序集合称为父（或者分别称为子类列表和超类序列）。这种关系通常被称为IS-A关系（IS-A意即列表是一个有序集合）。这意味着，列表从有序集合继承了重要的特征，也就是内部数据的顺序以及诸如拼接、重复和索引等方法。

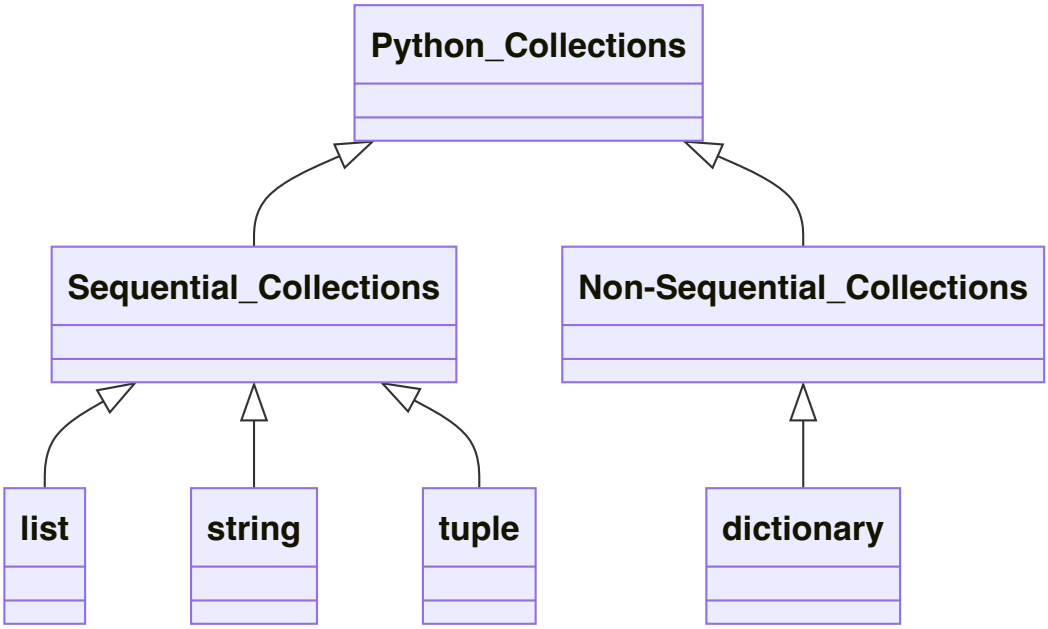


图2-4 Python容器的继承层次结构

列表、字符串和元组都是有序集合。它们都继承了共同的数据组织和操作。不过，根据数据是否同类以及集合是否可修改，它们彼此又有区别。子类从父类继承共同的特征，但是通过额外的特征彼此区分。

通过将类组织成继承层次结构，面向对象编程语言使以前编写的代码得以扩展到新的应用场景中。此外，这种结构有助于更好地理解各种关系，从而更高效地构建抽象表示。

> `namedtuple` 是 Python 的 `collections` 模块中提供的一种数据结构，它扩展了标准元组（tuple）的功能。与普通元组不同的是，`namedtuple` 允许你通过名称访问元素，而不仅仅是通过索引，这使得代码更具可读性和自解释性。

```python
from collections import namedtuple

# 定义一个名为 'Car' 的 namedtuple 类型，它有两个字段：'make' 和 'model'
Car = namedtuple('Car', ['make', 'model'])

# 创建一个 Car 实例
my_car = Car(make='Toyota', model='Corolla')

# 访问元素
print(my_car.make)  # 输出：Toyota
print(my_car.model)  # 输出：Corolla

# 也可以像普通 tuple 那样用索引访问
```

```python
print(my_car[0])  # 输出: Toyota
print(my_car[1])  # 输出: Corolla

# 尝试修改元素会引发错误，因为 namedtuple 是不可变的
# my_car.make = 'Honda'  # 这行代码将导致 AttributeError
```

`namedtuple` 特别适合用来创建轻量级的、不可变的数据对象，当你有一组相关的数据项，并且希望以一种更加面向对象的方式访问这些数据时，`namedtuple` 是一个非常方便的选择。

2 Time Complexities Big-O

使用数据结构与算法（DSA）的主要目的是为了有效地和高效地解决问题。你如何决定自己编写的程序是否高效呢？这通过复杂度来衡量。复杂度分为两种类型：

1. 时间复杂度：时间复杂度用于衡量执行代码所需的时间。
2. 空间复杂度：空间复杂度指的是成功执行代码功能所需的存储空间量。 在数据结构与算法中，你还会经常遇到**辅助空间**这个术语，它指的是程序中除了输入数据结构外使用的额外空间。

> Here comes one of the interesting and important topics. The primary motive to use DSA is to solve a problem effectively and efficiently. How can you decide if a program written by you is efficient or not? This is measured by complexities. Complexity is of two types:
>
> 1. Time Complexity: Time complexity is used to measure the amount of time required to execute the code.
> 2. Space Complexity: Space complexity means the amount of space required to execute successfully the functionalities of the code.
>   You will also come across the term **Auxiliary Space** very commonly in DSA, which refers to the extra space used in the program other than the input data structure.

上述两种复杂度都是相对于输入参数来衡量的。但这里出现了一个问题。执行一段代码所需的时间取决于多个因素，例如：

- 程序中执行的操作数量，
- 设备的速度，以及
- 如果是在在线平台上执行的话，数据传输的速度。

那么我们如何确定哪一个更高效呢？答案是使用渐近符号。**渐近符号**是一种数学工具，它根据输入大小计算所需时间，并不需要实际执行代码。

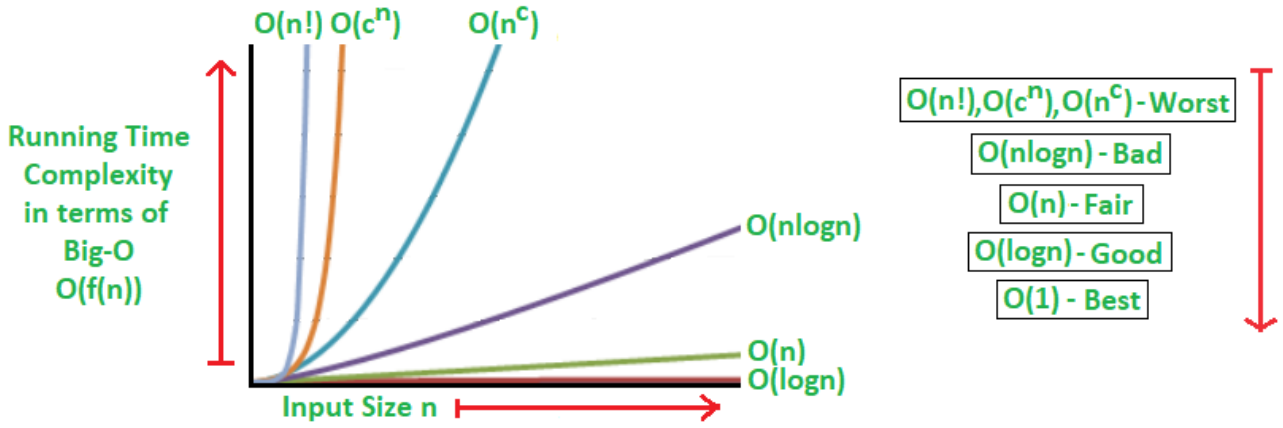
> Both of the above complexities are measured with respect to the input parameters. But here arises a problem. The time required for executing a code depends on several factors, such as:
>
> - The number of operations performed in the program,
> - The speed of the device, and also
> - The speed of data transfer if being executed on an online platform.

> So how can we determine which one is efficient? The answer is the use of asymptotic notation. **Asymptotic notation** is a mathematical tool that calculates the required time in terms of input size and does not require the execution of the code.

它忽略了依赖于系统的常数，并且只与整个程序中执行的模块化操作的数量有关。以下三种渐近符号最常用以表示算法的时间复杂度：

- **大O符号 (O)** – 大O符号特别描述了最坏情况下的情形。
- **欧米伽符号 (Ω)** – 欧米伽(Ω)符号特别描述了最好情况下的情形。
- **西塔符号 (θ)** – 这个符号代表了算法的平均复杂度。

> It neglects the system-dependent constants and is related to only the number of modular operations being performed in the whole program. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:
>
> - **Big-O Notation (O)** – Big-O notation specifically describes the worst-case scenario.
> - **Omega Notation (Ω)** – Omega(Ω) notation specifically describes the best-case scenario.
> - **Theta Notation (θ)** – This notation represents the average complexity of an algorithm.



算法的增长率

在代码分析中最常用的符号是**大O符号**，它给出了代码运行时间的上界（或者说是输入规模大小对应的内存使用量）。大O符号帮助我们理解当输入数据量增加时，算法的执行时间或空间需求将以怎样的速度增长。

> Rate of Growth of Algorithms
>
> The most used notation in the analysis of a code is the **Big O Notation** which gives an upper bound of the running time of the code (or the amount of memory used in terms of input size).

2.1 Analyzing algorithms

分析算法意味着预测该算法所需的资源。虽然有时我们主要关心像内存、通信带宽或计算机硬件这类资源，但是通常我们想要度量的是计算时间。一般来说，通过分析某个问题的几种候选算法，我们可以识别出最高效的那一个。这样的分析可能会指出不止一个可行的候选算法，但在这一过程中，我们通常可以淘汰几个较差的算法。

> **Analyzing** an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure. Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

在分析一个算法之前，我们必须有一个要使用的实现技术的模型，包括该技术的资源模型及其成本。我们将假设一种通用的单处理器计算模型——随机存取机（random-access machine, RAM）来作为我们的实现技术，算法可以用计算机程序来实现。在RAM模型中，指令是顺序执行的，没有并发操作。

> Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs. For most of this book, we shall assume a generic oneprocessor, **random-access machine (RAM)** model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs. In the RAM model, instructions are executed one after another, with no concurrent operations.

严格来说，我们应该精确地定义RAM模型的指令及其成本。然而这样做会很繁琐，并且不会对算法设计和分析提供太多的洞察力。但我们必须小心不要滥用RAM模型。例如，如果RAM有一个排序指令，那么我们就可以只用一条指令完成排序。这样的RAM将是不现实的，因为实际的计算机没有这样的指令。因此，我们的指导原则是实际计算机的设计方式。RAM模型包含了在实际计算机中常见的指令：算术运算（如加法、减法、乘法、除法、求余数、取底、取顶），数据移动（加载、存储、复制），以及控制（条件分支和无条件分支、子程序调用和返回）。每条这样的指令都需要固定的时间量。

> Strictly speaking, we should precisely define the instructions of the RAM model and their costs. To do so, however, would be tedious and would yield little insight into algorithm design and analysis. Yet we must be careful not to abuse the RAM model. For example, what if a RAM had an instruction that sorts? Then we couldsort in just one instruction. Such a RAM would be unrealistic, since real computers do not have such instructions. Our guide, therefore, is how real computers are designed. The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time.

RAM模型中的数据类型有整型和浮点实数型。虽然在此处我们通常不关心精度问题，但在某些应用中精度是至关重要的。我们也假设每个数据字的大小有限制。例如，在处理大小为n的输入时，我们通常假设对某个常数c≥1， 整数由`c lgn`位表示。我们要求c≥1是为了确保每个字能够容纳n的值，从而使我们能够索引各个输入元素，并且我们将c限制为常数以防止字长无限增长。（如果字长可以无限增长，我们就可以在一个字中存储大量数据并在恒定时间内对其进行操作——这显然是一种不切实际的情况。）

> The data types in the RAM model are integer and floating point (for storing real numbers). Although we typically do not concern ourselves with precision in this book, in some applications precision is crucial. We also assume a limit on the size of each word of data. For example, when working with inputs of size n, we typically assume that integers are represented by c lg n bits for some constant $c \geq 1$. We require $c \geq 1$ so that each word can hold the value of n, enabling us to index the individual input elements, and we restrict c to be a constant so that the word size does not grow arbitrarily. (If the word size could grow arbitrarily, we could store huge amounts of data in one word and operate on it all in constant time—clearly an unrealistic scenario.)

> 计算机字长（Computer Word Length）是指计算机中用于存储和处理数据的基本单位的位数。它表示计算机能够一次性处理的二进制数据的位数。
>
> 字长的大小对计算机的性能和数据处理能力有重要影响。较大的字长通常意味着计算机能够处理更大范围的数据或执行更复杂的操作。常见的字长包括 8 位、16 位、32 位和 64 位。
>
> 较小的字长可以节省存储空间和资源，适用于简单的计算任务和资源有限的设备。较大的字长通常用于处理更大量级的数据、进行复杂的计算和支持高性能计算需求。
>
> 需要注意的是，字长并不是唯一衡量计算机性能的指标，还有其他因素如处理器速度、缓存大小、操作系统等也会影响计算机的整体性能。

实际计算机包含未在上述列表中的指令，这些指令在RAM模型中代表了一个灰色地带。例如，幂运算是否是常数时间指令？在一般情况下，并不是；当x和y是实数时，计算x^y需要多条指令。然而，在某些受限的情况下，幂运算是一个常数时间操作。许多计算机有一个“左移”指令，它可以在常数时间内将一个整数的位向左移动k个位置。在大多数计算机中，将一个整数的位向左移动一个位置等价于乘以2，因此将位向左移动k个位置就等价于乘以2^k。因此，只要k不超过计算机字的位数，这样的计算机可以通过将整数1左移k个位置来在一个常数时间内计算出2^k。我们将努力避免在RAM模型中出现这样的灰色地带，但在k是一个足够小的正整数时，我们会把2^k的计算视为一个常数时间操作。

> Real computers contain instructions not listed above, and such instructions represent a gray area in the RAM model. For example, is exponentiation a constanttime instruction? In the general case, no; it takes several instructions to compute x^y when x and y are real numbers. In restricted situations, however, exponentiation is a constant-time operation. Many computers have a “shift left” instruction, which in constant time shifts the bits of an integer by k positions to the left. In most computers, shifting the bits of an integer by one position to the left is equivalent to multiplication by 2, so that shifting the bits by k positions to the left is equivalent to multiplication by 2^k. Therefore, such computers can compute 2^k in one constant-time instruction by shifting the integer 1 by k positions to the left, as long as k is no more than the number of bits in a computer word. We will endeavor to avoid such gray areas in the RAM model, but we will treat computation of 2^k as a constant-time operation when k is a small enough positive integer.

在RAM模型中，我们并不试图模拟现代计算机中常见的内存层次结构。也就是说，我们不模拟缓存或虚拟内存。一些计算模型尝试考虑内存层次结构效应，这在实际程序运行在真实机器上时有时是非常显著的。本书中有一小部分问题会考察内存层次结构效应，但总体而言，分析不会考虑它们。包括内存层次结构的模型比RAM模型复杂得多，所以可能难于使用。此外，基于RAM模型的分析通常是实际机器性能的良好预测指标。

> In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory. Several computational models attempt to account for memory-hierarchy effects, which are sometimes significant in real programs on real machines. A handful of problems in this book examine memory-hierarchy effects, but for the most part, the analyses in this book will not consider them. Models that include the memory hierarchy are quite a bit more complex than the RAM model, and so they can be difficult to work with. Moreover, RAM-model analyses are usually excellent predictors of performance on actual machines.

即使是在RAM模型中分析一个简单的算法也可能是一项挑战。所需的数学工具可能包括组合数学、概率论、代数技巧以及识别公式中最重要项的能力。由于一个算法的行为可能对每个可能的输入都是不同的，我们需要一种方式来用简单且易于理解的公式概括这种行为。

> Analyzing even a simple algorithm in the RAM model can be a challenge. The mathematical tools required may include combinatorics, probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula. Because the behavior of an algorithm may be different for each possible input, we need a means for summarizing that behavior in simple, easily understood formulas.

尽管我们通常只选择一种机器模型来分析给定的算法，但在决定如何表达我们的分析时，我们仍然面临许多选择。我们希望有一种方法可以简单地书写和操作，能够展示算法资源需求的重要特征，并抑制繁琐的细节。

> Even though we typically select only one machine model to analyze a given algorithm, we still face many choices in deciding how to express our analysis. We would like a way that is simple to write and manipulate, shows the important characteristics of an algorithm's resource requirements, and suppresses tedious details.

1 Analysis of insertion sort

在算法分析中有两个关键概念：输入规模（input size）和运行时间（running time），并以插入排序（INSERTION-SORT）为例来说明这些概念。

> The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers. Moreover, INSERTIONSORT can take different amounts of time to sort two input sequences of the same size depending on how nearly sorted they already are. In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input" more carefully.

输入规模是衡量一个算法输入大小的标准，它取决于具体的问题。对于一些问题，比如排序或计算离散傅里叶变换，最自然的度量方式是输入中项目的数量——例如，排序时的数组长度n。对于其他问题，如整数乘法，输入规模的最佳度量可能是表示输入所需的总位数。有时，用两个数字描述输入规模比用一个更合适，例如，如果算法的输入是一个图，则可以用顶点和边的数量来描述输入规模。

> The best notion for **input size** depends on the problem being studied. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input—for example, the array size n for sorting. For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph. We shall indicate which input size measure is being used with each problem we study.

运行时间是指特定输入上算法执行的基本操作或"步骤"的数量。为了使这个概念尽可能与机器无关，通常假设伪代码中的每一行需要常量时间来执行。也就是说，每一行可能需要不同的时间，但每执行第i行所需的时间为ci，其中ci是一个常数。这种观点符合随机访问机（RAM）模型，并且反映了大多数实际计算机上如何实现伪代码。

> The **running time** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine-independent as possible. For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time ci, where ci is a constant. This viewpoint is in keeping with the **RAM** model, and it also reflects how the pseudocode would be implemented on most actual computers.

随着讨论的深入，对插入排序运行时间的表达将从使用所有语句成本ci的复杂公式演变为一种更简单、更简洁、更易处理的符号。这种简化后的符号也将使得比较不同算法的效率变得更加容易。

> In the following discussion, our expression for the running time of INSERTIONSORT will evolve from a messy formula that uses all the statement costs ci to a much simpler notation that is more concise and more easily manipulated. This simpler notation will also make it easy to determine whether one algorithm is more efficient than another.

接下来，使用插入排序过程来展示每个语句的时间"成本"以及每个语句被执行的次数。对于每一个`j = 2, 3, ..., n`（其中`n = A.length`），令t_j表示当j取该值时，第7行的while循环测试被执行的次数。当for或while循环以常规方式退出（即由于循环头中的测试）时，测试被执行的次数比循环体多一次。另外，这里假定注释不是可执行语句，因此它们不需要任何时间。

> We start by presenting the INSERTION-SORT procedure with the time "cost" of each statement and the number of times each statement is executed. For each `j = 2, 3, ... , n`, where `n = A.length`, we let t_j denote the number of times the **while** loop test in line 7 is executed for that value of j . When a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body. We assume that comments are not executable statements, and so they take no time.

Implementation of Insertion Sort Algorithm

`\sum_{j=2}^{n} t_j` 是 $\sum_{j=2}^{n} t_j$ 的LaTex表示，

`\sum_{j=2}^{n} t_{j}-1` 是 $\sum_{j=2}^{n} t_j - 1$ 的LaTex表示。

```python
def insertion_sort(arr):                                # cost  times
    for i in range(1, len(arr)):                        # c1    n
        j = i                                           # c2    n - 1

        # Insert arr[j] into the
        # sorted sequence arry[0..j-1]                  # 0     n - 1
        while arr[j - 1] > arr[j] and j > 0:            # c4    \sum_{j=2}^{n} t_j
            arr[j - 1], arr[j] = arr[j], arr[j - 1]     # c5    \sum_{j=2}^{n} t_j - 1
            j -= 1                                      # c6    \sum_{j=2}^{n} t_j - 1


arr = [2, 6, 5, 1, 3, 4]
insertion_sort(arr)
print(arr)
```

```
15
16 # [1, 2, 3, 4, 5, 6]
```

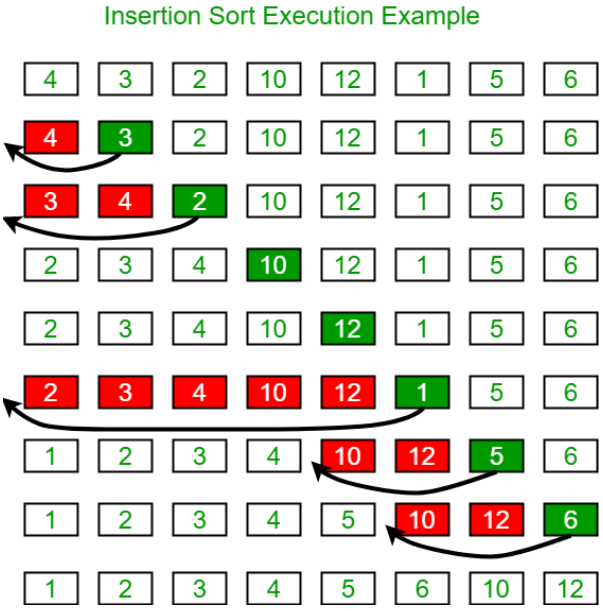
https://www.geeksforgeeks.org/insertion-sort/

插入排序是一种简单的排序算法，其工作原理类似于你在手中整理扑克牌的方式。数组被虚拟地分成已排序和未排序两部分。从未排序部分选取值，并将其放置到已排序部分的正确位置上。

Insertion sort is a simple sorting algorithm that works similarly to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed in the correct position in the sorted part.

要以升序对大小为N的数组进行排序，需要遍历数组并将当前元素（称为“关键字”）与它的前一个元素进行比较；如果关键字比它的前一个元素小，则将它与前面的元素进行比较。将较大的元素移动一个位置以腾出空间给被交换的元素。

To sort an array of size N in ascending order iterate over the array and compare the current element (key) to its predecessor, if the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



Q: Suppose you have the following list of numbers to sort: [15, 5, 4, 18, 12, 19, 14, 10, 8, 20] which list represents the partially sorted list after three complete passes of insertion sort? (C)

> A. [4, 5, 12, 15, 14, 10, 8, 18, 19, 20]
> B. [15, 5, 4, 10, 12, 8, 14, 18, 19, 20]
> **C. [4, 5, 15, 18, 12, 19, 14, 10, 8, 20]**
> D. [15, 5, 4, 18, 12, 19, 14, 8, 10, 20]

算法的运行时间是每个被执行语句的运行时间之和；一个执行一次需要c_i步骤并且总共执行n次的语句将对总运行时间贡献$c_i \times n$。为了计算T(n)，即在n个值的输入上INSERTION-SORT的运行时间，我们将成本列和次数列的乘积相加，得到

$$T(n) = \sum(c_i \times t_i)$$

这里，每个c_i代表伪代码中第i行执行一次所需的时间（常量），而t_i则表示该行被执行的次数。对于插入排序，我们需要考虑每一行代码被执行的具体情况，特别是内层循环的执行次数会依赖于数组中元素的初始排列。通过这种方式，我们可以得出一个关于n（输入大小）的函数表达式，用来描述插入排序的运行时间。

> The running time of the algorithm is the sum of running times for each statement executed; a statement that takes c_i steps to execute and executes n times will contribute $c_i n$ to the total running time. To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} t_j - 1 + c_6 \sum_{j=2}^{n} t_j - 1$$

即使对于给定大小的输入，算法的运行时间也可能取决于给出的是哪个具体输入。例如，在插入排序（INSERTION-SORT）中，最佳情况发生在数组已经是有序的时候。对于每个i = 1, 2, 3, ..., n-1，当j在其初始值i时，我们在第7行发现$arr[j-1] \leq arr[j]$。因此，对于i = 1, 2, 3, ..., n-1，有$t_j = 1$，这时最佳情况下的运行时间是

> Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given. For example, in INSERTION-SORT, the best case occurs if the array is already sorted. For each i = 1, 2, 3, ... , n-1, we then find that $arr[j-1] \leq arr[j]$ in line 7 when j has its initial value of i. Thus $t_j = 1$ for i = 1, 2, 3, ... , n-1, and the best-case running time is

$$\begin{aligned} T_{\text{best}}(n) &= c_1 n + c_2(n-1) + c_4(n-1) \\ &= (c_1 + c_2 + c_4)n - (c_2 + c_4) \end{aligned}$$

我们可以将这个运行时间表示为`an + b`，其中常量a和b取决于语句成本c_i；因此，它是n的**线性函数**。

如果数组是以逆序排序的——也就是说，以递减顺序排列——那么就会出现最坏情况。我们必须将每个元素A[j]与整个已排序子数组`A[0..j-1]`中的每个元素进行比较，因此 $t_j = j$ 对于`j = 1, 2, ..., n-1`。注意到这一点，

> We can express this running time as `an + b` for constants a and b that depend on the statement costs c_i; it is thus a **linear function** of n.

> If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element A[j] with each element in the entire sorted subarray `A[0..j-1]`, and so $t_j = j$ for `j = 1, 2, ..., n-1`. Noting that

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

$\sum_{j=2}^{n} j - 1 = \frac{n(n-1)}{2}$

we find that in the worst case, the running time of INSERTION-SORT is

$$T_{\text{worst}}(n) = c_1 n + c_2(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n-1)}{2}) + +c_6(\frac{n(n-1)}{2})$$
$$= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2})n - (c_2 + c_4)$$

我们可以将这种最坏情况下的运行时间表示为$an^2 + bn + c$，其中常量a、b和c再次取决于语句成本c_i；因此，它是n的**二次函数**。

通常情况下，就像在插入排序中一样，对于给定的输入，算法的运行时间是固定的，尽管在后续章节中我们将看到一些有趣的“随机化”算法，即使对于固定的输入，它们的行为也可能有所不同。

> We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs ci; it is thus a **quadratic function** of n.
>
> Typically, as in insertion sort, the running time of an algorithm is fixed for a given input, although in later chapters we shall see some interesting “randomized” algorithms whose behavior can vary even for a fixed input.

2 Worst-case and average-case analysis

在我们对插入排序的分析中，我们既考虑了最佳情况，即输入数组已经排序的情况，也考虑了最坏情况，即输入数组是逆序排列的情况。然而，在本书的其余部分，我们将通常专注于寻找只有**最坏情况下的运行时间**，也就是对于任何大小为n的输入最长的运行时间。我们给出关注最坏情况的三个理由：

> In our analysis of insertion sort, we looked at both the best case, in which the input array was already sorted, and the worst case, in which the input array was reverse sorted. For the remainder of this book, though, we shall usually concentrate on finding only the **worst-case running time**, that is, the longest running time for any input of size n. We give three reasons for this orientation.

- 一个算法的最坏情况下的运行时间为任何输入提供了一个运行时间的上限。了解它提供了算法永远不会超过这个时间的保证。我们不需要对运行时间做出一些有根据的猜测，并希望它不会变得更糟。
- 对于某些算法，最坏情况出现得相当频繁。例如，在数据库中搜索特定信息时，当信息不在数据库中时，搜索算法的最坏情况经常发生。在某些应用中，可能经常会进行不存在的信息搜索。

- “平均情况”通常几乎和最坏情况一样糟糕。假设我们随机选择n个数字并应用插入排序。确定元素A[j]应该插入到子数组`A[0 .. j-1]`中的哪个位置需要多长时间？平均来说，`A[0 .. j-1]`中的一半元素小于`A[j]`，另一半大于`A[j]`。因此，平均而言，我们需要检查子数组`A[0 .. j-1]`的一半，所以t_j大约是$j/2$。结果得到的平均情况下的运行时间最终是输入规模的二次函数，就像最坏情况下的运行时间一样。

> The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer. We need not make some educated guess about the running time and hope that it never gets much worse.
>
> For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm’s worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.
>
> The “average case” is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray `A[0 .. j-1]` to insert element A[j] ? On average, half the elements in `A[0 .. j-1]` are less than `A[j]`, and half the elements are greater. On average, therefore, we check half of the subarray `A[0 .. j-1]`, and so t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

> 在某些特殊情况下，我们将对算法的**平均情况**运行时间感兴趣；我们将在本书中看到**概率分析**技术应用于各种算法。平均情况分析的范围是有限的，因为可能不清楚什么构成特定问题的“平均”输入。我们经常假设所有给定大小的输入都是等可能的。实际上，这一假设可能会被违反，但有时我们可以使用一种**随机化算法**，它会做出随机选择，从而使概率分析成为可能，并得出一个**期望**的运行时间。
>
> In some particular cases, we shall be interested in the **average-case** running time of an algorithm; we shall see the technique of **probabilistic analysis** applied to various algorithms throughout this book. The scope of average-case analysis is limited, because it may not be apparent what constitutes an “average” input for a particular problem. Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a **randomized algorithm**, which makes random choices, to allow a probabilistic analysis and yield an **expected** running time.

3 Order of growth

我们使用了一些简化的抽象来简化对INSERTIONSORT过程的分析。首先，我们忽略了每个语句的实际成本，用常数c_i来表示这些成本。然后，我们注意到即使这些常数也给了我们比实际需要更多的细节：我们将最坏情况下的运行时间表达为$an^2 + bn + c$，其中a、b和c是依赖于语句成本c_i的常数。因此，我们不仅忽略了实际的语句成本，还忽略了抽象成本c_i。

> We used some simplifying abstractions to ease our analysis of the INSERTIONSORT procedure. First, we ignored the actual cost of each statement, using the constants ci to represent these costs. Then, we observed that even these constants give us more detail than we really need: we expressed the worst-case running time as $an^2 + bn + c$ for some constants a, b, and c that depend on the statement costs c_i. We thus ignored not only the actual statement costs, but also the abstract costs c_i.

现在我们将引入另一个简化的抽象：真正引起我们兴趣的是运行时间的增长率，或称为增长阶。因此，我们只考虑公式的主要项（例如，an^2），因为对于n的较大值来说，低阶项相对来说不那么重要。我们也忽略主要项的常数系数，因为在确定大输入的计算效率时，常数因子不如增长率重要。对于插入排序，当我们忽略低阶项和主要项的常数系数后，我们剩下的是来自主要项的n^2因子。我们说插入排序具有$\Theta(n^2)$（发音为“theta of n-squared”）的最坏情况运行时间。

> We shall now make one more simplifying abstraction: it is the **rate of growth**, or **order of growth**, of the running time that really interests us. We therefore consider only the leading term of a formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n. We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs. For insertion sort, when we ignore the lower-order terms and the leading term's constant coefficient, we are left with the factor of n^2 from the leading term. We write that insertion sort has a worst-case running time of $\Theta(n^2)$ (pronounced “theta of n-squared”).

通常我们认为一个算法如果其最坏情况运行时间的增长阶较低，则它比另一个算法更高效。由于常数因子和低阶项的影响，一个运行时间增长阶较高的算法在小输入的情况下可能会比一个增长阶较低的算法花费的时间更少。但对于足够大的输入，例如，在最坏情况下，一个$\Theta(n^2)$的算法将比一个$\Theta(n^3)$的算法运行得更快。

> We usually consider one algorithm to be more efficient than another if its worstcase running time has a lower order of growth. Due to constant factors and lowerorder terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, a $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\Theta(n^3)$ algorithm.
>
> 这里提到的Θ 符号是用来描述算法运行时间的增长阶的紧确界，意味着算法的运行时间在渐近情况下既不会快于也不会慢于n^2的某个常数倍。这是关于算法复杂度分析中的渐近记法的一种表述方式，用来概括地说明算法性能随输入规模变化的趋势。

4 O-notation

> 通用的记号应该是，O表示上界，Ω表示下界，Θ表示渐进阶，就是既上界又下界。

Θ-记号从渐近上界和下界两个方面约束一个函数。当我们只有渐近上界时，我们使用O-记号。对于给定的函数g(n)，我们用O(g(n)（读作“大O of g of n”或简称“O of g of n”）来表示满足以下条件的函数集合：

> The Θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n) (pronounced “big-oh of g of n” or sometimes just “oh of g of n”) the set of functions

$$O(g(n)) = \{f(n)：存在正的常数c和n_0，使得对所有n \geq n_0, 有0 \leq f(n) \leq cg(n)\}$$

我们使用O-记号来给出一个函数的上界，最多相差一个常数因子。

利用O-记号，我们通常可以通过检查算法的整体结构来简单描述算法的运行时间。例如，插入排序算法中的双重嵌套循环结构立即给出了最坏情况下运行时间为$O(n^2)$的上界。

由于O-记号描述的是上界，当我们用它来约束算法的最坏情况运行时间时，我们就得到了该算法在任何输入上的运行时间的上限。这意味着，在最坏情况下，算法不会比这个上界更慢，无论输入是什么。

> We use O-notation to give an upper bound on a function, to within a constant factor.
>
> Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example, the doubly nested loop structure of the insertion sort algorithm immediately yields an $O(n^2)$ upper bound on the worst-case running time.
>
> Since O-notation describes an upper bound, when we use it to bound the worstcase running time of an algorithm, we have a bound on the running time of the algorithm on every input.

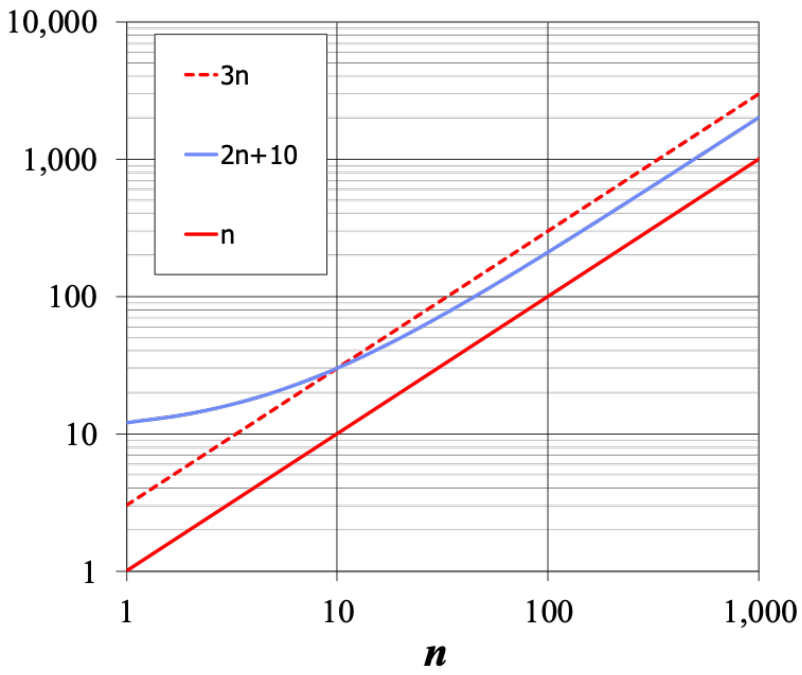
举例： $2n + 10$ is $O(n)$

$2n + 10 \leq cn$

$(c - 2)n \geq 10$

$n \geq 10/(c - 2)$

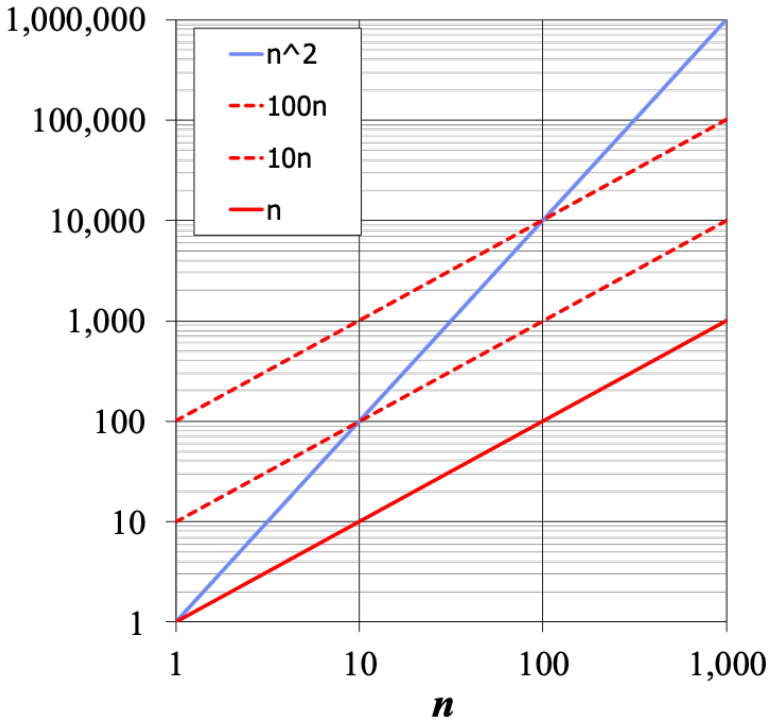
Pick $c = 3 \; and \; n_0 = 10$



举例： the function n^2 is not O(n)

$n^2 \leq cn$

$n \leq c$, the inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

$7n - 2$ is O(n)

need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$

$3n^3 + 20n^2 + 5$ is $O(n^3)$

need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

This is true for $c = 4$ and $n_0 = 21$

$3logn + 5$ is $O(logn)$

need $c > 0$ and $n_0 > 1$ such that $3logn + 5 \leq clogn$ for $n \geq n_0$

this is true for $c = 8$ and $n_0 = 2$

大O记号给出了函数增长率的上界。陈述`f(n) 是 O(g(n))`意味着`f(n)`的增长率不超过`g(n)`的增长率。

我们可以使用大O记号根据它们的增长率来对函数进行排序。

换句话说，如果一个函数`f(n)`是O(g(n))，那么对于足够大的输入n，`f(n)`的值不会超过`g(n)`的某个常数倍。这提供了一种方式来描述和比较不同算法随着输入规模增加而表现出的效率差异，通过将算法的运行时间或空间需求的增长率与一些基准函数（如线性、平方、立方、指数等）进行对比。在算法分析中，我们经常使用大O记号来简化表达，并专注于算法性能的关键趋势，忽略掉那些对于大规模输入影响较小的细节。

> The big-Oh notation gives an upper bound on the growth rate of a function. The statement `f(n) is O(g(n))` means that the growth rate of `f(n)` is no more than the growth rate of `g(n)`. •We can use the big-Oh notation to rank functions according to their growth rate.

Big-Oh Rules

If is `f(n)` a polynomial of degree `d`, then `f(n)` is $O(n^d)$, i.e.,

Drop lower-order terms 忽略低阶项

Drop constant factors 忽略常数因子

Use the smallest possible class of functions 使用尽可能小的函数类别

Say $2n$ is $O(n)$ instead of $2n$ is $O(n^2)$

Use the simplest expression of the class 使用该类别中最简单的表达方式

Say $3n + 5$ is $O(n)$ instead of $3n + 5$ is $O(3n)$

Asymptotic Algorithm Analysis 渐近算法分析

算法的渐近分析确定了以大O记号表示的运行时间。

为了执行渐近分析，找到作为输入规模函数的最坏情况下的原始操作数量，并用大O记号来表示这个函数。

例子：

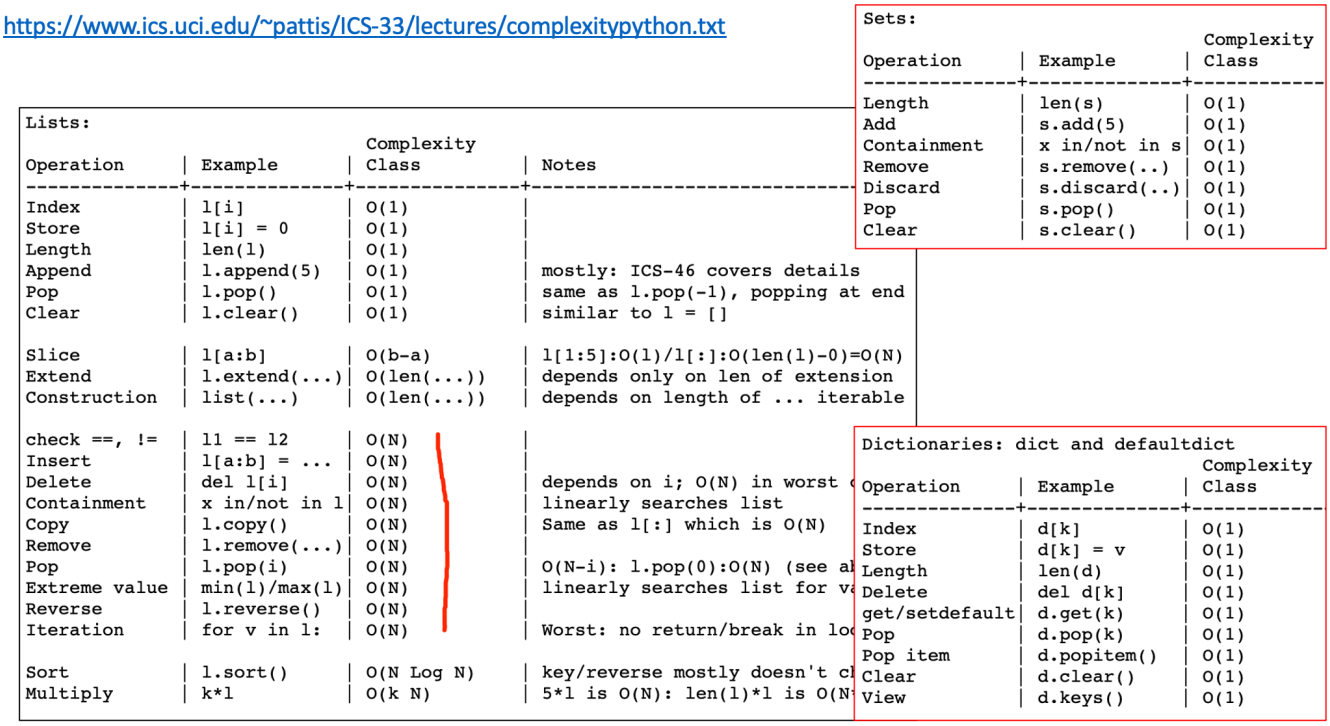
可以说算法find_max“在O(n)时间内运行”

由于最终无论如何都会忽略常数因子和低阶项，所以在计算原始操作时可以不考虑它们。

> 这意味着，在进行算法分析时，我们主要关注的是随着输入大小增加，算法性能如何变化的趋势。通过忽略那些对于大输入规模影响较小的细节（如低阶项和常数因子），我们可以简化分析，并专注于理解算法在处理大规模数据时的行为。这种分析方法允许我们比较不同算法之间的效率，而不需要深入到具体的实现细节中去。
>
> The asymptotic analysis of an algorithm determines the running time in big-Oh notation.
>
> To perform the asymptotic analysis, find the worst-case number of primitive operations executed as a function of the input size, express this function with big-Oh notation
>
> Example:
>
> say that algorithm **find_max** “runs in **O**(**n**) time”
>
> Since constant factors and lower-order terms are eventually dropped anyhow, disregard them when counting primitive operations

https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt

https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt



Lists:

Operation	Example	Complexity Class	Notes
Index	l[i]	O(1)	
Store	l[i] = 0	O(1)	
Length	len(l)	O(1)	
Append	l.append(5)	O(1)	mostly: ICS-46 covers details
Pop	l.pop()	O(1)	same as l.pop(-1), popping at end
Clear	l.clear()	O(1)	similar to l = []
Slice	l[a:b]	O(b-a)	l[1:5]:O(l)/l[:]:O(len(l)-0)=O(N)
Extend	l.extend(...)	O(len(...))	depends only on len of extension
Construction	list(...)	O(len(...))	depends on length of ... iterable
check ==, !=	l1 == l2	O(N)	
Insert	l[a:b] = ...	O(N)	
Delete	del l[i]	O(N)	depends on i; O(N) in worst c
Containment	x in/not in l	O(N)	linearly searches list
Copy	l.copy()	O(N)	Same as l[:] which is O(N)
Remove	l.remove(...)	O(N)	
Pop	l.pop(i)	O(N)	O(N-i): l.pop(0):O(N) (see al
Extreme value	min(l)/max(l)	O(N)	linearly searches list for va
Reverse	l.reverse()	O(N)	
Iteration	for v in l:	O(N)	Worst: no return/break in loc
Sort	l.sort()	O(N Log N)	key/reverse mostly doesn't cl
Multiply	k*l	O(k N)	5*l is O(N): len(l)*l is O(N

Sets:

Operation	Example	Complexity Class
Length	len(s)	O(1)
Add	s.add(5)	O(1)
Containment	x in/not in s	O(1)
Remove	s.remove(..)	O(1)
Discard	s.discard(..)	O(1)
Pop	s.pop()	O(1)
Clear	s.clear()	O(1)

Dictionaries: dict and defaultdict

Operation	Example	Complexity Class
Index	d[k]	O(1)
Store	d[k] = v	O(1)
Length	len(d)	O(1)
Delete	del d[k]	O(1)
get/setdefault	d.get(k)	O(1)
Pop	d.pop(k)	O(1)
Pop item	d.popitem()	O(1)
Clear	d.clear()	O(1)
View	d.keys()	O(1)

2.2 Sorting Algorithm

Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

There are a lot of different types of sorting algorithms. Some widely used algorithms are:

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort
- Merge Sort
- ShellSort

There are several other sorting algorithms also and they are beneficial in different cases. You can learn about them and more in our dedicated article on Sorting algorithms.

1 Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Algorithm

> In Bubble Sort algorithm,

> - traverse from left and compare adjacent elements and the higher one is placed at right side.
> - In this way, the largest element is moved to the rightmost end at first.
> - This process is then continued to find the second largest and place it and so on until the data is sorted.

```python
# Optimized Python program for implementation of Bubble Sort
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already in place
        for j in range(0, n - i - 1):

            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if (swapped == False):
            break


# Driver code to test above
if __name__ == "__main__":
    arr = [64, 34, 25, 12, 22, 11, 90]

    bubbleSort(arr)
    print(' '.join(map(str, arr)))

```

Complexity Analysis of Bubble Sort:

Time Complexity: O(N2)
Auxiliary Space: O(1)

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

Some FAQs related to Bubble Sort:

Q1. What is the Boundary Case for Bubble sort?

Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid O(N2) time complexity.

Q2. Does sorting happen in place in Bubble sort?

Yes, Bubble sort performs the swapping of adjacent pairs without the use of any major data structure. Hence Bubble sort algorithm is an in-place algorithm.

Q3. Is the Bubble sort algorithm stable?

Yes, the bubble sort algorithm is stable.

Q4. Where is the Bubble sort algorithm used?

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.

Q: Suppose you have the following list of numbers to sort: [19, 1, 9, 7, 3, 10, 13, 15, 8, 12] which list represents the partially sorted list after three complete passes of bubble sort?? （B）

A：[1, 9, 19, 7, 3, 10, 13, 15, 8, 12] B：**[1, 3, 7, 9, 10, 8, 12, 13, 15, 19]**

C：[1, 7, 3, 9, 10, 13, 8, 12, 15, 19] D：[1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

2 Selection Sort

> **Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

```
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
```

```
    min_idx = i
    for j in range(i + 1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

        # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print(' '.join(map(str, A)))

# Output: 11 12 22 25 64
```

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n−1 passes to sort *n* items, since the final item must be in place after the (n−1) st pass.

Figure 3 shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on.

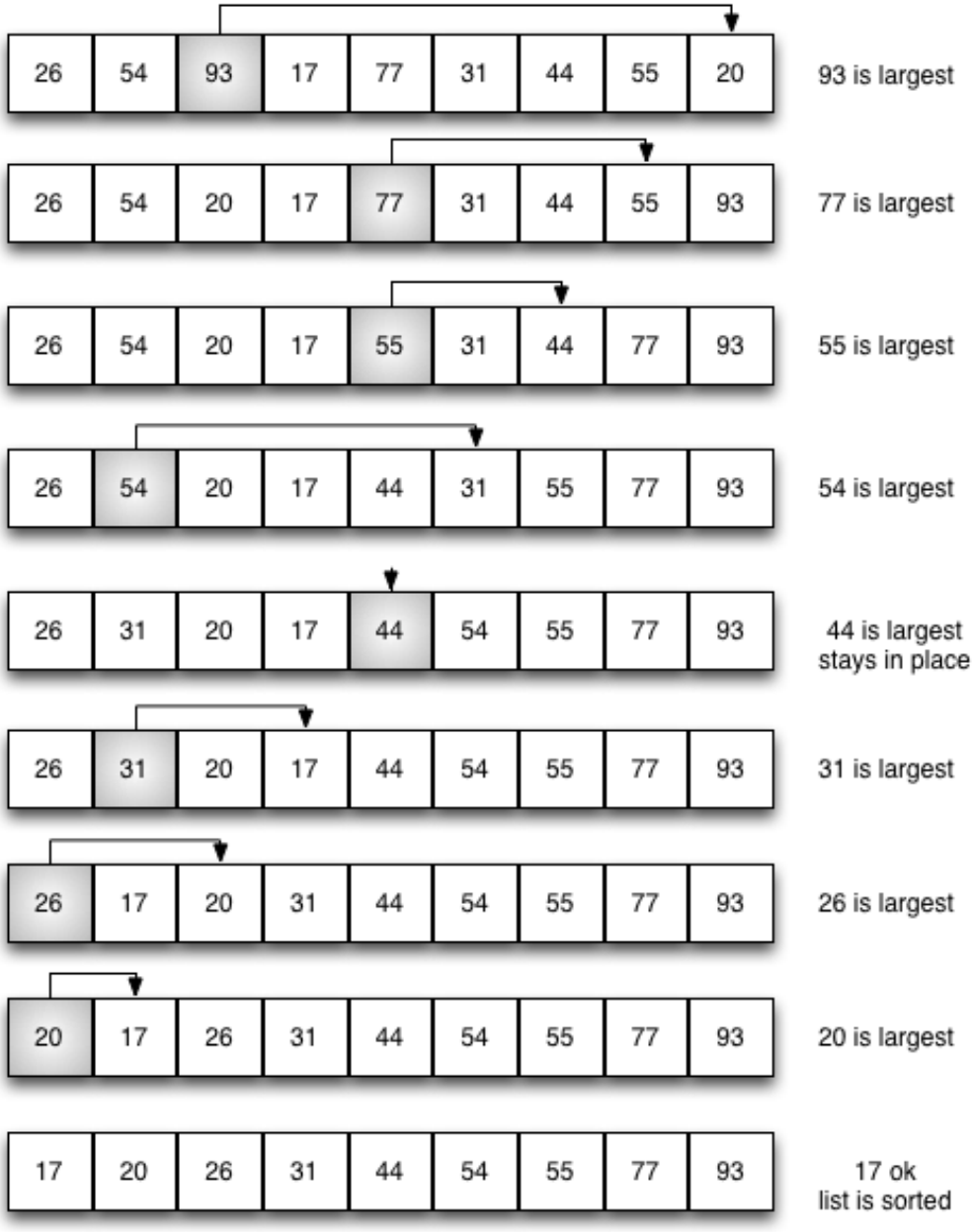


Figure 3: `selectionSort`

```
def selectionSort(alist):
    for fillslot in range(len(alist)-1, 0, -1):
        positionOfMax = 0
        for location in range(1, fillslot+1):
            if alist[location] > alist[positionOfMax]:
                positionOfMax = location

        if positionOfMax != fillslot:
            alist[fillslot], alist[positionOfMax] = alist[positionOfMax],
alist[fillslot]

alist = [54, 26, 93, 17, 77, 31, 44, 55, 20]
selectionSort(alist)
```

```
print(alist)

# [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

You may see that the selection sort makes the same number of comparisons as the bubble sort and is therefore also (O^2). However, due to the reduction in the number of exchanges, the selection sort typically executes faster in benchmark studies. In fact, for our list, the bubble sort makes 20 exchanges, while the selection sort makes only 8.

Complexity Analysis of Selection Sort

Time Complexity: The time complexity of Selection Sort is **O(N2)** as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)
- Therefore overall complexity = O(N) * O(N) = O(N*N) = O(N2)

Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

Disadvantages of the Selection Sort Algorithm

- Selection sort has a time complexity of O(n^2) in the worst and average case.
- Does not work well on large datasets.
- Does not preserve the relative order of items with equal keys which means it is not stable.

Frequently Asked Questions on Selection Sort

Q1. Is Selection Sort Algorithm stable?

The default implementation of the Selection Sort Algorithm is **not stable**. However, it can be made stable. Please see the stable Selection Sort for details.

Q2. Is Selection Sort Algorithm in-place?

Yes, Selection Sort Algorithm is an in-place algorithm, as it does not require extra space.

Q: Suppose you have the following list of numbers to sort: [11, 7, 12, 14, 19, 1, 6, 18, 8, 20] which list represents the partially sorted list after three complete passes of selection sort? (D)

A. [7, 11, 12, 1, 6, 14, 8, 18, 19, 20]
B. [7, 11, 12, 14, 19, 1, 6, 18, 8, 20]
C. [11, 7, 12, 14, 1, 6, 8, 18, 19, 20]
D. **[11, 7, 12, 14, 8, 1, 6, 18, 19, 20]**

3 Quick Sort

> quickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

How does QuickSort work?

> The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.
>
> Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

> https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm-data-structure-and-algorithm-tutorials/
>
> **Divide And Conquer**
> This technique can be divided into the following three parts:
>
> 1. **Divide:** This involves dividing the problem into smaller sub-problems.
> 2. **Conquer:** Solve sub-problems by calling recursively until solved.
> 3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.
>
> The following are some standard algorithms that follow Divide and Conquer algorithm.
>
> 1. **Quicksort** is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
> 2. **Merge Sort** is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.
>
> What does not qualifies as Divide and Conquer:
> Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in the array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for the left side

> of the middle element, else recurs for the right side of the middle element. Contrary to popular belief, this is not an example of Divide and Conquer because there is only one sub-problem in each step (Divide and conquer requires that there must be two or more sub-problems) and hence this is a case of Decrease and Conquer.

在partition函数中两个指针 `i` 和 `j` 的方式实现。快排中的partition可以用双指针或者单指针实现，前者更容易理解，也是机考中喜欢出的题目类型。

```python
def quicksort(arr, left, right):
    if left < right:
        partition_pos = partition(arr, left, right)
        quicksort(arr, left, partition_pos - 1)
        quicksort(arr, partition_pos + 1, right)


def partition(arr, left, right):
    i = left
    j = right - 1
    pivot = arr[right]
    while i <= j:
        while i <= right and arr[i] < pivot:
            i += 1
        while j >= left and arr[j] >= pivot:
            j -= 1
        if i < j:
            arr[i], arr[j] = arr[j], arr[i]
    if arr[i] > pivot:
        arr[i], arr[right] = arr[right], arr[i]
    return i


arr = [22, 11, 88, 66, 55, 77, 33, 44]
quicksort(arr, 0, len(arr) - 1)
print(arr)

# [11, 22, 33, 44, 55, 66, 77, 88]
```

To analyze the `quickSort` function, note that for a list of length *n*, if the partition always occurs in the middle of the list, there will again be $logn$ divisions. In order to find the split point, each of the *n* items needs to be checked against the pivot value. The result is $nlogn$. In addition, there is no need for additional memory as in the merge sort process.

Unfortunately, in the worst case, the split points may not be in the middle and can be very skewed to the left or the right, leaving a very uneven division. In this case, sorting a list of *n* items divides into sorting a list of 0 items and a list of n−1 items. Then sorting a list of n−1 divides into a list of size 0 and a list of size n−2, and so on. The result is an $O(n^2)$ sort with all of the overhead that recursion requires.

We mentioned earlier that there are different ways to choose the pivot value. In particular, we can attempt to alleviate some of the potential for an uneven division by using a technique called **median of three**. To choose the pivot value, we will consider the first, the middle, and the last element in the list. In our example, those are 54, 77, and 20. Now pick the median value, in our case 54, and use it for the pivot value (of course, that was the pivot value we used originally). The idea is that in the case where the first item in the list does not belong toward the middle of the list, the median of three will choose a better "middle" value. This will be particularly useful when the original list is somewhat sorted to begin with.

Complexity Analysis of Quick Sort:

Time Complexity:

- Best Case: $\Omega(Nlog(N))$
  The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
- Average Case: $\Theta(Nlog(N))$
  Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
- Worst Case: $O(N^2)$
  The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

Auxiliary Space: O(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Q. Choose the leftmost element as pivot, given the following list of numbers [14, 17, 13, 15, 19, 10, 3, 16, 9, 12] which answer shows the contents of the list after the second partitioning according to the quicksort algorithm? (D)

A. [9, 3, 10, 13, 12]
B. [9, 3, 10, 13, 12, 14]
C. [9, 3, 10, 13, 12, 14, 17, 16, 15, 19]
D. [9, 3, 10, 13, 12, 14, 19, 16, 15, 17]

The first partitioning works on the entire list, and the second partitioning works on the left partition not the right. It's important to remember that quicksort works on the entire list and sorts it in place.

Q: Given the following list of numbers [1, 20, 11, 5, 2, 9, 16, 14, 13, 19] what would be the first pivot value using the median of 3 method? (B)

A. 1
B. 9
C. 16
D. 19

although 16 would be the median of 1, 16, 19 the middle is at len(list) // 2.

Q: Which of the following sort algorithms are guaranteed to be O(n log n) even in the worst case? (C)

A. Shell Sort
B. Quick Sort
C. Merge Sort
D. Insertion Sort

Merge Sort is the only guaranteed O(n log n) even in the worst case. The cost is that merge sort uses more memory.

4 Merge Sort

> Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

How does Merge Sort work?

> Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

```python
def mergeSort(arr):
  if len(arr) > 1:
    mid = len(arr)//2

    L = arr[:mid] # Dividing the array elements
    R = arr[mid:] # Into 2 halves

    mergeSort(L) # Sorting the first half
    mergeSort(R) # Sorting the second half

    i = j = k = 0
    # Copy data to temp arrays L[] and R[]
    while i < len(L) and j < len(R):
      if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
      else:
        arr[k] = R[j]
        j += 1
      k += 1

    # Checking if any element was left
    while i < len(L):
      arr[k] = L[i]
      i += 1
      k += 1

    while j < len(R):
      arr[k] = R[j]
      j += 1
      k += 1


if __name__ == '__main__':
  arr = [12, 11, 13, 5, 6, 7]
  mergeSort(arr)
  print(' '.join(map(str, arr)))
# Output: 5 6 7 11 12 13
```

Complexity Analysis of Merge Sort

Time Complexity: O(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

> T(n) = 2T(n/2) + θ(n)

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

- Sorting large datasets: Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
- External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
- Custom sorting: Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.
- Inversion Count Problem: Inversion Count for an array indicates – how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

Advantages of Merge Sort:

- Stability: Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- Guaranteed worst-case performance: Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
- Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

- Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Not always optimal for small datasets: For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Q: 给定排序列表 [21,1,26,45,29,28,2,9,16,49,39,27,43,34,46,40]，在归并排序的第 3 次递归调用时，排序的是哪个子表？（B）

A： [16, 49,39,27,43,34,46,40] B： **[21,1]** C： [21,1,26,45] D：[21]

Remember mergesort doesn't work on the right half of the list until the left half is completely sorted.

Q: 排序数据同上，归并排序中，哪两个子表是最先归并的？（C）

A： [21,1] and [26,45] B：[1, 2, 9,21,26,28,29,45] and [16,27,34,39,40,43, 46,49]

C： **[21] and [1]** D：[9] and [16]

The lists [21] and [1] are the first two base cases encountered by mergesort and will therefore be the first two lists merged.

5 Shell Sort

Shell sort is mainly a variation of **Insertion Sort**. In insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved. The idea of ShellSort is to allow the exchange of far items. In Shell sort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h'th element are sorted.

Algorithm:

Step 1 − Start
Step 2 − Initialize the value of gap size. Example: h
Step 3 − Divide the list into smaller sub-part. Each must have equal intervals to h
Step 4 − Sort these sub-lists using insertion sort
Step 5 – Repeat this step 2 until the list is sorted.
Step 6 – Print a sorted list.
Step 7 – Stop.

```python
def shellSort(arr, n):
    # code here
    gap = n // 2

    while gap > 0:
        j = gap
        # Check the array in from left to right
        # Till the last possible index of j
        while j < n:
            i = j - gap  # This will keep help in maintain gap value

            while i >= 0:
                # If value on right side is already greater than left side value
                # We don't do swap else we swap
                if arr[i + gap] > arr[i]:
                    break
                else:
                    arr[i + gap], arr[i] = arr[i], arr[i + gap]

                i = i - gap  # To check left side also
            # If the element present is greater than current element
            j += 1
```

```
        gap = gap // 2


# driver to check the code
arr2 = [12, 34, 54, 2, 3]

shellSort(arr2, len(arr2))
print(' '.join(map(str, arr2)))

# Output: 2 3 12 34 54
```

Time Complexity: Time complexity of the above implementation of Shell sort is $O(n^2)$. In the above implementation, the gap is reduced by half in every iteration. There are many other ways to reduce gaps which leads to better time complexity. See this for more details.

Worst Case Complexity
The worst-case complexity for shell sort is $O(n^2)$

Shell Sort Applications

1. Replacement for insertion sort, where it takes a long time to complete a given task.
2. To call stack overhead we use shell sort.
3. when recursion exceeds a particular limit we use shell sort.
4. For medium to large-sized datasets.
5. In insertion sort to reduce the number of operations.

https://en.wikipedia.org/wiki/Shellsort

The running time of Shellsort is heavily dependent on the gap sequence it uses. For many practical variants, determining their time complexity remains an open problem.

Unlike **insertion sort**, Shellsort is not a **stable sort** since gapped insertions transport equal elements past one another and thus lose their original order. It is an **adaptive sorting algorithm** in that it executes faster when the input is partially sorted.

Stable sort algorithms sort equal elements in the same order that they appear in the input.

Q: Given the following list of numbers: [5, 16, 20, 12, 3, 8, 9, 17, 19, 7] Which answer illustrates the contents of the list after all swapping is complete for a gap size of 3? (A)

A. [5, 3, 8, 7, 16, 19, 9, 17, 20, 12]
B. [3, 7, 5, 8, 9, 12, 19, 16, 20, 17]
C. [3, 5, 7, 8, 9, 12, 16, 17, 19, 20]
D. [5, 16, 20, 3, 8, 12, 9, 17, 20, 7]

Each group of numbers represented by index positions 3 apart are sorted correctly.

6 Comparison sorts

> 在排序算法中，稳定性是指相等元素的相对顺序是否在排序后保持不变。换句话说，如果排序算法在排序过程中保持了相等元素的相对顺序，则称该算法是稳定的，否则是不稳定的。
>
> 对于判断一个排序算法是否稳定，一种常见的方法是观察交换操作。挨着交换（相邻元素交换）是稳定的，而隔着交换（跳跃式交换）可能会导致不稳定性。

Below is a table of comparison sorts. A comparison sort cannot perform better than *O*(*n* log *n*) on average.

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$nlog^2n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.
Heapsort	$nlogn$	$nlogn$	$nlogn$	1	No	Selection	
Merge sort	$nlogn$	$nlogn$	$nlogn$	*n*	Yes	Merging	Highly parallelizable (up to *O*(log *n*) using the Three Hungarian's Algorithm)
Timsort	*n*	$nlogn$	$nlogn$	*n*	Yes	Insertion & Merging	Makes *n-1* comparisons when the data is already sorted or reverse sorted.
Quicksort	$nlogn$	$nlogn$	n^2	$logn$	No	Partitioning	Quicksort is usually done in-place with *O*(log *n*) stack space.
Shellsort	$nlogn$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Insertion sort	*n*	n^2	n^2	1	Yes	Insertion	*O*(n + d), in the worst case over sequences that have *d* inversions.
Bubble sort	*n*	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.

Highly tuned implementations use more sophisticated variants, such as Timsort (merge sort, insertion sort, and additional logic), used in Android, Java, and Python, and introsort (quicksort and heapsort), used (in variant forms) in some C++ sort implementations and in .NET.

3 基本数据结构

What Are Linear Structures?

We will begin our study of data structures by considering four simple but very powerful concepts. Stacks, queues, deques, and lists are examples of data collections whose items are ordered depending on how they are added or removed. Once an item is added, it stays in that position relative to the other elements that came before and came after it. Collections such as these are often referred to as **linear data structures**.

Linear structures can be thought of as having two ends. Sometimes these ends are referred to as the “left” and the “right” or in some cases the “front” and the “rear.” You could also call them the “top” and the “bottom.” The names given to the ends are not significant. What distinguishes one linear structure from another is the way in which items are added and removed, in particular the location where these additions and removals occur. For example, a structure might allow new items to be added at only one end. Some structures might allow items to be removed from either end.

These variations give rise to some of the most useful data structures in computer science. They appear in many algorithms and can be used to solve a variety of important problems.

What is a Stack?

A **stack** (sometimes called a “push-down stack”) is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the “top.” The end opposite the top is known as the “base.”

The base of the stack is significant since items stored in the stack that are closer to the base represent those that have been in the stack the longest. The most recently added item is the one that is in position to be removed first. This ordering principle is sometimes called **LIFO**, **last-in first-out**. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base.

Many examples of stacks occur in everyday situations. Almost any cafeteria has a stack of trays or plates where you take the one at the top, uncovering a new tray or plate for the next customer in line.

What Is a Queue?

A queue is an ordered collection of items where the addition of new items happens at one end, called the “rear,” and the removal of existing items occurs at the other end, commonly called the “front.” As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO**, **first-in first-out**. It is also known as “first-come first-served.”

线性表是一种逻辑结构，描述了元素按线性顺序排列的规则。常见的线性表存储方式有**数组**和**链表**，它们在不同场景下具有各自的优势和劣势。

数组是一种连续存储结构，它将线性表的元素按照一定的顺序依次存储在内存中的连续地址空间上。数组需要预先分配一定的内存空间，每个元素占用相同大小的内存空间，并可以通过索引来进行快速访问和操作元素。访问元素的时间复杂度为O(1)，因为可以直接计算元素的内存地址。然而，插入和删除元素的时间复杂度较高，平均为O(n)，因为需要移动其他元素来保持连续存储的特性。

链表是一种存储结构，它是线性表的链式存储方式。链表通过节点的相互链接来实现元素的存储。每个节点包含元素本身以及指向下一个节点的指针。链表的插入和删除操作非常高效，时间复杂度为O(1)，因为只需要调整节点的指针。然而，访问元素的时间复杂度较高，平均为O(n)，因为必须从头节点开始遍历链表直到找到目标元素。

选择使用数组还是链表作为存储方式取决于具体问题的需求和限制。如果需要频繁进行随机访问操作，数组是更好的选择。如果需要频繁进行插入和删除操作，链表更适合。通过了解它们的特点和性能，可以根据实际情况做出选择。

在Python中，list 更接近于数组的存储结构。

https://www.ics.uci.edu/~pattis/ICS-33/lectures/complexitypython.txt

Lists:

Operation	Example	Complexity Class	Notes
Index	l[i]	O(1)	
Store	l[i] = 0	O(1)	
Length	len(l)	O(1)	
Append	l.append(5)	O(1)	mostly: ICS-46 covers details
Pop	l.pop()	O(1)	same as l.pop(-1), popping at end
Clear	l.clear()	O(1)	similar to l = []
Slice	l[a:b]	O(b-a)	l[1:5]:O(l)/l[:]:O(len(l)-0)=O(N)
Extend	l.extend(...)	O(len(...))	depends only on len of extension
Construction	list(...)	O(len(...))	depends on length of ... iterable
check ==, !=	l1 == l2	O(N)	
Insert	l[a:b] = ...	O(N)	
Delete	del l[i]	O(N)	depends on i; O(N) in worst
Containment	x in/not in l	O(N)	linearly searches list
Copy	l.copy()	O(N)	Same as l[:] which is O(N)
Remove	l.remove(...)	O(N)	
Pop	l.pop(i)	O(N)	O(N-i): l.pop(0):O(N) (see al
Extreme value	min(l)/max(l)	O(N)	linearly searches list for va
Reverse	l.reverse()	O(N)	
Iteration	for v in l:	O(N)	Worst: no return/break in loo
Sort	l.sort()	O(N Log N)	key/reverse mostly doesn't cl
Multiply	k*l	O(k N)	5*l is O(N): len(l)*l is O(N

Sets:

Operation	Example	Complexity Class
Length	len(s)	O(1)
Add	s.add(5)	O(1)
Containment	x in/not in s	O(1)
Remove	s.remove(..)	O(1)
Discard	s.discard(..)	O(1)
Pop	s.pop()	O(1)
Clear	s.clear()	O(1)

Dictionaries: dict and defaultdict

Operation	Example	Complexity Class
Index	d[k]	O(1)
Store	d[k] = v	O(1)
Length	len(d)	O(1)
Delete	del d[k]	O(1)
get/setdefault	d.get(k)	O(1)
Pop	d.pop(k)	O(1)
Pop item	d.popitem()	O(1)
Clear	d.clear()	O(1)
View	d.keys()	O(1)

4 线性表之顺序表

线性表（$List$）的定义：零个或多个数据元素的**有限**序列。

线性表的数据集合为$\{a_1,a_2.....a_n\}$，该序列有唯一的头元素和尾元素，除了头元素外，每个元素都有唯一的前驱元素，除了尾元素外，每个元素都有唯一的后继元素。

线性表中的元素属于相同的数据类型，即每个元素所占的空间相同。

框架：

$$\text{线性表}\begin{cases}\text{顺序存储}\text{——}\text{顺序表}\\ \text{链式存储}\begin{cases}\text{单链表}\\ \text{双链表}\\ \text{循环链表}\end{cases}\end{cases}\tag{1}$$

4.1 顺序表

python中的顺序表就是列表，元素在内存中连续存放，每个元素都有唯一序号(下标），且根据序号访问（包括读取和修改）元素的时间复杂度是$O(1)$的（**随机访问**）。

代码使用Python的内置列表来实现

```python
class SequentialList:
    def __init__(self, n=0):
        """
        初始化顺序表，可以指定初始元素的数量n，默认为0。
        如果n大于0，则初始化一个包含从0到n-1整数的顺序表。
        """
        self.data = list(range(n)) if n > 0 else []

    def is_empty(self):
        """检查顺序表是否为空"""
        return len(self.data) == 0

    def length(self):
        """返回顺序表中元素的数量"""
        return len(self.data)

    def append(self, item):
        """在顺序表末尾添加一个新元素"""
        self.data.append(item)

    def insert(self, index, item):
        """在指定位置插入一个新元素"""
        if not (0 <= index <= len(self.data)):
            raise IndexError('Index out of range')
        self.data.insert(index, item)

    def delete(self, index):
        """删除指定位置的元素"""
        if not (0 <= index < len(self.data)):
            raise IndexError('Index out of range')
        del self.data[index]

    def get(self, index):
        """获取指定位置的元素"""
        if not (0 <= index < len(self.data)):
            raise IndexError('Index out of range')
        return self.data[index]

    def set(self, index, target):
        """设置指定位置的元素值"""
        if not (0 <= index < len(self.data)):
            raise IndexError('Index out of range')
        self.data[index] = target

    def display(self):
        """打印顺序表中的所有元素"""
        print(self.data)

# 示例用法
if __name__ == "__main__":
```

```
    # 创建一个空的顺序表
    lst = SequentialList()
    print("Initial empty list:")
    lst.display()  # 应该输出: []

    # 添加一些元素
    lst.append(1)
    lst.append(2)
    lst.append(3)
    print("After appending 1, 2, 3:")
    lst.display()  # 应该输出: [1, 2, 3]

    # 在特定位置插入元素
    lst.insert(1, 5)
    print("After inserting 5 at index 1:")
    lst.display()  # 应该输出: [1, 5, 2, 3]

    # 获取和设置元素
    print(f"Element at index 2: {lst.get(2)}")  # 应该输出: Element at index 2: 2
    lst.set(2, 7)
    print("After setting index 2 to 7:")
    lst.display()  # 应该输出: [1, 5, 7, 3]

    # 删除元素
    lst.delete(1)
    print("After deleting element at index 1:")
    lst.display()  # 应该输出: [1, 7, 3]

    # 检查长度和是否为空
    print(f"Length of the list: {lst.length()}")  # 应该输出: Length of the list: 3
    print(f"Is the list empty? {lst.is_empty()}")  # 应该输出: Is the list empty?
False

    # 尝试创建一个带有初始元素的顺序表
    lst_with_initial_elements = SequentialList(5)
    print("List with initial elements (0 to 4):")
    lst_with_initial_elements.display()  # 应该输出: [0, 1, 2, 3, 4]

```

关于线性表的时间复杂度：

生成、求表中元素个数、表尾添加/删除元素、返回/修改对应下标元素，均为$O(1)$；

而查找、删除、插入元素，均为$O(n)$。

线性表的优缺点：

优点：1、无须为表中元素之间的逻辑关系而增加额外的存储空间；

2、可以快速的存取表中任一位置的元素。

缺点：1、插入和删除操作需要移动大量元素；

2、当线性表长度较大时，难以确定存储空间的容量；

3、造成存储空间的“碎片”。

5 线性表之链表

链表（Linked List）是一种常见的数据结构，用于存储和组织数据。它由一系列节点组成，每个节点包含一个数据元素和一个指向下一个节点（或前一个节点）的指针。

在链表中，每个节点都包含两部分：

1. 数据元素（或数据项）：这是节点存储的实际数据。可以是任何数据类型，例如整数、字符串、对象等。
2. 指针（或引用）：该指针指向链表中的下一个节点（或前一个节点）。它们用于建立节点之间的连接关系，从而形成链表的结构。

根据指针的类型和连接方式，链表可以分为不同类型，包括：

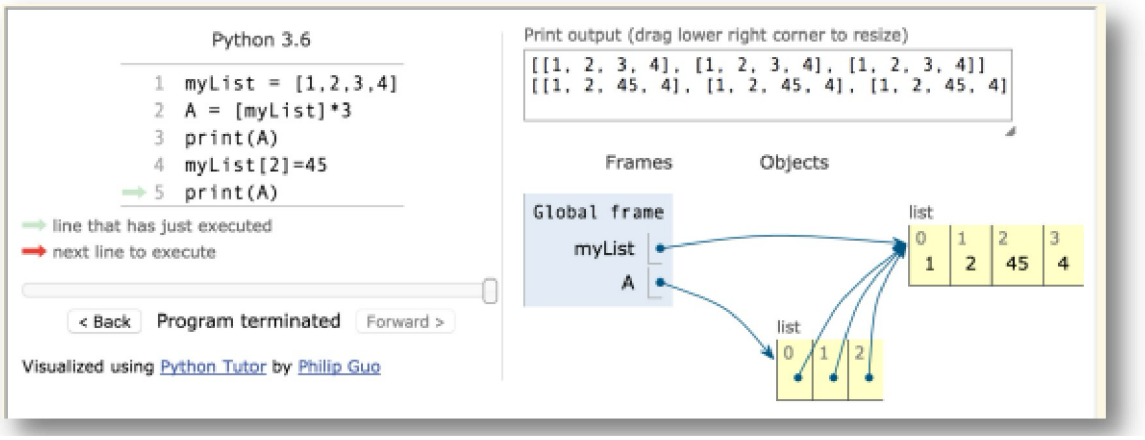
1. 单向链表：每个节点只有一个指针，指向下一个节点。链表的头部指针指向第一个节点，而最后一个节点的指针为空（指向 `None`）。
2. 双向链表：每个节点有两个指针，一个指向前一个节点，一个指向后一个节点。双向链表可以从头部或尾部开始遍历，并且可以在任意位置插入或删除节点。
3. 循环链表：最后一个节点的指针指向链表的头部，形成一个环形结构。循环链表可以从任意节点开始遍历，并且可以无限地循环下去。

链表相对于数组的一个重要特点是，链表的大小可以动态地增长或缩小，而不需要预先定义固定的大小。这使得链表在需要频繁插入和删除元素的场景中更加灵活。

然而，链表的访问和搜索操作相对较慢，因为需要遍历整个链表才能找到目标节点。与数组相比，链表的优势在于插入和删除操作的效率较高，尤其是在操作头部或尾部节点时。因此，链表在需要频繁插入和删除元素而不关心随机访问的情况下，是一种常用的数据结构。

可变类型的变量引用情况（回头看）

- 列表里面没有直接存对象，而是对象的指针（地址/引用）
- 不管是什么类型，它的指针大小都是一样的
- Python 把这些内部细节都隐藏起来了
  - alist $= [1, 2, 3, 4]$
  - blist $=$ alist
  - blist$[0] ='$ abc$'$



在 Python 中，`list` 是使用动态数组（Dynamic Array）实现的，而不是链表。动态数组是一种连续的、固定大小的内存块，可以在需要时自动调整大小。这使得 `list` 支持快速的随机访问和高效的尾部操作，例如附加（append）和弹出（pop）。

与链表不同，动态数组中的元素在内存中是连续存储的。这允许通过索引在 `list` 中的任何位置进行常数时间（O(1)）的访问。此外，动态数组还具有较小的内存开销，因为它们不需要为每个元素存储额外的指针。

当需要在 `list` 的中间进行插入或删除操作时，动态数组需要进行元素的移动，因此这些操作的时间复杂度是线性的（O(n)）。如果频繁地插入或删除元素，而不仅仅是在尾部进行操作，那么链表可能更适合，因为链表的插入和删除操作在平均情况下具有常数时间复杂度。

总结起来，Python 中的 `list` 是使用动态数组实现的，具有支持快速随机访问和高效尾部操作的优点。但是，如果需要频繁进行插入和删除操作，可能需要考虑使用链表或其他数据结构。

Python 中的 list 和 C++ 中的 STL（Standard Template Library）中的 vector 具有相似的实现和用法。vector 也是使用动态数组实现的，提供了类似于 list 的功能，包括随机访问、尾部插入和删除等操作。

链表在某种意义上可以给树打基础。

5.1 单向链表

基本概念

单向链表（Singly Linked List）是由一系列节点（Node）构成的线性数据结构，每个节点包含两个部分：

- 数据部分：存储节点的数据。
- 指针部分：存储指向下一个节点的指针（或引用）。

单链表的特点是每个节点只有一个指针，指向下一个节点。因此，它是单向的，只能从头到尾遍历。

单向链表结构图

```
Head -> Node1 -> Node2 -> Node3 -> NULL
```

- `Head`：指向链表的第一个节点。
- 每个 `Node`：包含数据和指向下一个节点的指针（`next`）。
- `NULL`：表示链表的结束，最后一个节点的 `next` 指向 `NULL`。

常见操作

- 插入操作：可以在链表的头部、尾部或中间插入新节点。
- 删除操作：可以删除链表中的某个节点。
- 遍历操作：从头部开始，逐一访问链表中的每个节点。
- 查找操作：根据节点数据查找对应的节点。

单向链表实现1：尾插法

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        new_node = Node(value)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

    def delete(self, value):
        if self.head is None:
```

```
            return

        if self.head.value == value:
            self.head = self.head.next
        else:
            current = self.head
            while current.next:
                if current.next.value == value:
                    current.next = current.next.next
                    break
                current = current.next

    def display(self):
        current = self.head
        while current:
            print(current.value, end=" ")
            current = current.next
        print()

# 使用示例
linked_list = LinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()  # 输出：1 2 3
linked_list.delete(2)
linked_list.display()  # 输出：1 3
```

单向链表实现2，保存了链表的长度

```
class LinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data = data  # Store data
            self.next = next  # Point to the next node

    def __init__(self):
        self.head = None  # Initialize head as None
        self.tail = None  # Initialize tail as None
        self.size = 0  # Initialize size to 0

    def print(self):
        ptr = self.head
        while ptr is not None:
            if ptr != self.head:  # Avoid printing a comma before the first element
                print(',', end='')
            print(ptr.data, end='')
            ptr = ptr.next
        print()  # Move to the next line after printing all elements
```

```
20
21      def insert_after(self, p, data):
22          nd = LinkList.Node(data)
23          if p is None:  # If p is None, insert at the beginning
24              self.pushFront(data)
25          else:
26              nd.next = p.next
27              p.next = nd
28              if p == self.tail:  # Update tail if necessary
29                  self.tail = nd
30              self.size += 1
31
32      def delete_after(self, p):
33          if p is None or p.next is None:
34              return  # Nothing to delete
35          if self.tail is p.next:  # Update tail if necessary
36              self.tail = p
37          p.next = p.next.next
38          self.size -= 1
39
40      def popFront(self):
41          if self.head is None:
42              raise Exception("Popping front from empty link list.")
43          else:
44              data = self.head.data
45              self.head = self.head.next
46              self.size -= 1
47              if self.size == 0:
48                  self.tail = None
49              return data
50
51      def pushFront(self, data):
52          nd = LinkList.Node(data, self.head)
53          self.head = nd
54          if self.size == 0:
55              self.tail = nd
56          self.size += 1
57
58      def pushBack(self, data):
59          if self.size == 0:
60              self.pushFront(data)
61          else:
62              self.insert_after(self.tail, data)
63
64      def clear(self):
65          self.head = None
66          self.tail = None
67          self.size = 0
68
69      def __iter__(self):
70          self.ptr = self.head
71          return self
```

```
72
73      def __next__(self):
74          if self.ptr is None:
75              raise StopIteration()
76          else:
77              data = self.ptr.data
78              self.ptr = self.ptr.next
79              return data
80
81  # 示例用法
82  if __name__ == "__main__":
83      ll = LinkList()
84      ll.pushFront(1)
85      ll.pushFront(2)
86      ll.pushBack(3)
87      ll.print()  # 应该输出：2,1,3
88      ll.delete_after(ll.head)  # 删除第二个元素 (1)
89      ll.print()  # 应该输出：2,3
90      print(f"Pop Front: {ll.popFront()}")  # 应该输出：Pop Front: 2
91      ll.print()  # 应该输出：3
```

单链表的应用

- 动态内存管理：链表可以灵活地分配内存空间，特别适用于内存空间不固定的场景。
- 实现队列和栈：链表能够有效地支持栈（LIFO）和队列（FIFO）的实现，因为其在插入和删除操作上有优势。
- 动态集合管理：对于集合操作（如动态插入和删除元素）非常高效。

5.2 双向链表

基本概念

双向链表（Doubly Linked List）是一种数据结构，其中每个节点不仅包含指向下一个节点的指针（`next`），还包含指向前一个节点的指针（`prev`）。这样，双向链表能够在两端进行遍历：从头到尾和从尾到头。

双链表的结构图

```
1  NULL <- Node1 <-> Node2 <-> Node3 -> NULL
```

- 每个节点有两个指针：
  - `next`：指向下一个节点。
  - `prev`：指向前一个节点。
- `NULL`：表示链表的头和尾，头节点head/Node1的 `prev` 指向 `NULL`，尾节点tail/Node3的 `next` 指向 `NULL`。

常见操作

- 插入操作：可以在链表的头部、尾部或中间插入新节点，插入操作需要同时调整 `next` 和 `prev` 指针。
- 删除操作：可以删除链表中的某个节点，删除操作需要更新前后节点的指针。
- 遍历操作：可以从头到尾或从尾到头进行遍历。

双向链表代码实现

```python
class Node:
    def __init__(self, data):
        self.data = data  # 节点数据
        self.next = None  # 指向下一个节点
        self.prev = None  # 指向前一个节点

class DoublyLinkedList:
    def __init__(self):
        self.head = None  # 链表头部
        self.tail = None  # 链表尾部

    # 在链表尾部添加节点
    def append(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node

    # 在链表头部添加节点
    def prepend(self, data):
        new_node = Node(data)
        if not self.head:  # 如果链表为空
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    # 删除链表中的指定节点
    def delete(self, node):
        if not self.head:  # 链表为空
            return

        if node == self.head:  # 删除头部节点
            self.head = node.next
            if self.head:  # 如果链表非空
                self.head.prev = None
        elif node == self.tail:  # 删除尾部节点
```

```
            self.tail = node.prev
            if self.tail:  # 如果链表非空
                self.tail.next = None
        else:  # 删除中间节点
            node.prev.next = node.next
            node.next.prev = node.prev

        node = None  # 删除节点

    # 打印链表中的所有元素，从头到尾
    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")

    # 打印链表中的所有元素，从尾到头
    def print_reverse(self):
        current = self.tail
        while current:
            print(current.data, end=" <-> ")
            current = current.prev
        print("None")

# 创建双向链表对象
dll = DoublyLinkedList()

# 添加节点
dll.append(10)
dll.append(20)
dll.append(30)

# 在头部添加节点
dll.prepend(5)

# 打印链表
print("从头到尾打印：")
dll.print_list()    # 5 <-> 10 <-> 20 <-> 30 <-> None

# 打印链表（逆序）
print("从尾到头打印：")
dll.print_reverse() # 30 <-> 20 <-> 10 <-> 5 <-> None

# 删除节点
dll.delete(dll.head.next)  # 删除第二个节点（数据为10）

# 打印链表
print("删除一个节点后，链表为：")
dll.print_list()    # 5 <-> 20 <-> 30 <-> None

```

> - **append**：将新节点添加到链表的尾部。
> - **prepend**：将新节点添加到链表的头部。
> - **delete**：删除链表中的指定节点（无论是头节点、尾节点还是中间节点）。
> - **print_list**：从头到尾打印链表中的所有节点。
> - **print_reverse**：从尾到头打印链表中的所有节点。
>
> 双向链表相对于单向链表的优势在于它能实现双向遍历，使得在某些操作上（例如反向遍历、删除特定节点等）更加高效。

双链表的应用

- 双向遍历：由于双链表可以从头到尾或从尾到头遍历，因此在某些需要双向遍历的数据结构（如浏览器历史记录、操作系统任务调度等）中非常有用。
- 实现双端队列（Deque）：双链表非常适合用于双端队列的实现，可以在队头和队尾都进行快速的插入和删除。
- 内存管理和垃圾回收：双链表用于管理动态内存块，常见于操作系统的内存管理和垃圾回收机制中

5.3 单链表与双链表的对比

特性	单链表	双链表
指针数量	每个节点一个指针，指向下一个节点	每个节点两个指针，分别指向前后节点
访问方向	只能从头到尾访问	可以从头到尾或从尾到头访问
内存开销	较低，仅需存储一个指针	较高，需要存储两个指针
插入/删除效率	在头部插入删除高效，但中间插入删除较慢	在任意位置插入删除较高效
操作复杂度	操作简单，适合轻量级应用	操作复杂，适用于双向操作场景
应用场景	动态内存管理，队列、栈实现	双端队列实现，双向遍历等

- 单链表 适用于动态内存管理和需要简单数据操作的场景，其操作效率相对较低，特别是在中间插入和删除时。
- 双链表 通过提供双向指针，增强了操作的灵活性，适用于需要双向遍历和高效插入删除的场景，如双端队列、浏览器历史记录等。
- 两者的选择应根据具体应用场景而定。如果需要简单的线性遍历和动态插入，单链表即可满足需求；而如果涉及到双向操作和复杂的内存管理，双链表则更加合适。

链表结构是基础数据结构之一，理解其操作和算法对于深入学习更复杂的算法和数据结构具有重要意义。

5.4 循环链表

将单链表中终端节点的指针端由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相接的单链表称为单循环链表，简称循环链表。

然而这样会导致访问最后一个结点时需要$O(n)$的时间，所以我们可以写出**仅设尾指针的循环链表**。

```python
class CircleLinkList:
    class Node:
        def __init__(self, data, next=None):
            self.data = data
            self.next = next

    def __init__(self):
        self.tail = None  # 尾指针，指向最后一个节点
        self.size = 0  # 链表大小

    def is_empty(self):
        """检查链表是否为空"""
        return self.size == 0

    def pushFront(self, data):
        """在链表头部插入元素"""
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail  # 自己指向自己形成环
        else:
            nd.next = self.tail.next  # 新节点指向当前头节点
            self.tail.next = nd  # 当前尾节点指向新节点
        self.size += 1

    def pushBack(self, data):
        """在链表尾部插入元素"""
        nd = CircleLinkList.Node(data)
        if self.is_empty():
            self.tail = nd
            nd.next = self.tail  # 自己指向自己形成环
        else:
            nd.next = self.tail.next  # 新节点指向当前头节点
            self.tail.next = nd  # 当前尾节点指向新节点
            self.tail = nd  # 更新尾指针
        self.size += 1

    def popFront(self):
        """移除并返回链表头部元素"""
        if self.is_empty():
            return None
        else:
            old_head = self.tail.next
```

```
            if self.size == 1:
                self.tail = None  # 如果只有一个元素，更新尾指针为None
            else:
                self.tail.next = old_head.next  # 跳过旧头节点
            self.size -= 1
            return old_head.data

    def popBack(self):
        """移除并返回链表尾部元素"""
        if self.is_empty():
            return None
        elif self.size == 1:
            data = self.tail.data
            self.tail = None
            self.size -= 1
            return data
        else:
            prev = self.tail
            while prev.next != self.tail:  # 找到倒数第二个节点
                prev = prev.next
            data = self.tail.data
            prev.next = self.tail.next  # 跳过尾节点
            self.tail = prev  # 更新尾指针
            self.size -= 1
            return data

    def printList(self):
        """打印链表中的所有元素"""
        if self.is_empty():
            print('Empty!')
        else:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=', ' if ptr != self.tail else '\n')
                if ptr == self.tail:
                    break
                ptr = ptr.next

# 示例用法
if __name__ == "__main__":
    clist = CircleLinkList()

    print("Pushing elements to front:")
    for i in range(3):
        clist.pushFront(i)
        clist.printList()  # 应该依次输出: 0, 1,0, 2,1,0,

    print("Pushing elements to back:")
    for i in range(3, 6):
        clist.pushBack(i)
        clist.printList()  # 应该依次输出: 2,1,0,3, 2,1,0,3,4, 2,1,0,3,4,5,

```

```
    print("Popping from front:")
    for _ in range(3):
        print(f"Popped: {clist.popFront()}")
        clist.printList()  # 应该依次输出：2,1,0,3,4,5, 1,0,3,4,5, 0,3,4,5,

    print("Popping from back:")
    for _ in range(3):
        print(f"Popped: {clist.popBack()}")
        clist.printList()  # 应该依次输出：5, 3,4, 5, 4, 3, Empty!
```

5.5 常见算法：链表的操作

1 链表反转（Reverse Linked List）

链表反转是一个经典的算法，它将链表中的节点顺序反转，使得原本指向下一个节点的指针指向前一个节点。该操作在处理栈或队列时非常有用。

单链表反转算法

```
def reverse_linked_list(head):
    prev = None
    current = head
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node
    return prev
```

2 合并两个排序的链表

合并两个已经排序的链表是一种常见的操作，特别是在归并排序中。

合并两个排序链表

```
def merge_sorted_lists(l1, l2):
    dummy = Node(0)
    tail = dummy
    while l1 and l2:
        if l1.data < l2.data:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    if l1:
```

```
            tail.next = l1
        else:
            tail.next = l2
        return dummy.next
```

3 查找链表的中间节点

通过快慢指针的方法，可以在 O(n) 的时间复杂度内找到链表的中间节点。

查找中间节点

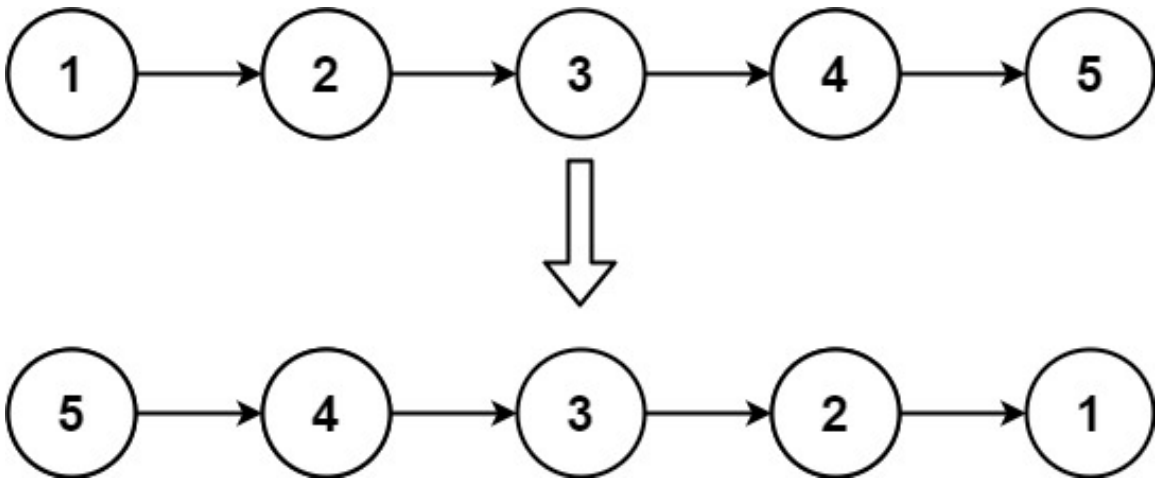
```
def find_middle_node(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

示例206.反转链表

linked-list, https://leetcode.cn/problems/reverse-linked-list/

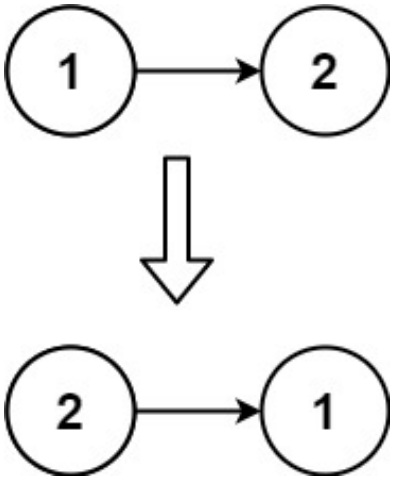
给你单链表的头节点 `head` ，请你反转链表，并返回反转后的链表。

示例 1：




```
输入：head = [1,2,3,4,5]
输出：[5,4,3,2,1]
```

示例 2：




```
输入：head = [1,2]
输出：[2,1]
```

示例 3：

```
输入：head = []
输出：[]
```

提示：

- 链表中节点的数目范围是 `[0, 5000]`
- `-5000 <= Node.val <= 5000`

```python
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        pre = None
        current = head
        while current:
            next_node = current.next
            current.next = pre
            pre = current
            current = next_node

        return pre

```

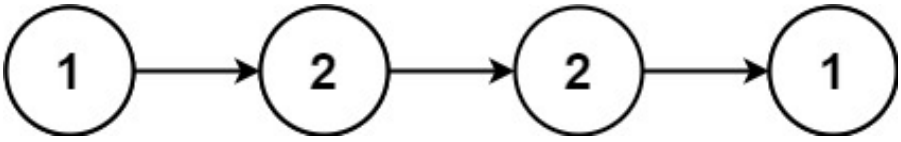
示例234.回文链表

linked-list, https://leetcode.cn/problems/palindrome-linked-list/

给你一个单链表的头节点 `head` ，请你判断该链表是否为

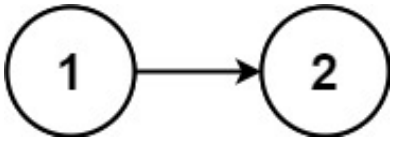
回文链表（**回文** 序列是向前和向后读都相同的序列。如果是，返回 `true` ；否则，返回 `false` 。

示例 1：




```
输入：head = [1,2,2,1]
输出：true
```

示例 2：




```
输入：head = [1,2]
输出：false
```

提示：

- 链表中节点数目在范围 `[1, 105]` 内
- `0 <= Node.val <= 9`

进阶：你能否用 `O(n)` 时间复杂度和 `O(1)` 空间复杂度解决此题?

快慢指针查找链表的中间节点

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
```

```python
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        if not head or not head.next:
            return True

        # 1. 使用快慢指针找到链表的中点
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # 2. 反转链表的后半部分
        prev = None
        while slow:
            next_node = slow.next
            slow.next = prev
            prev = slow
            slow = next_node

        # 3. 对比前半部分和反转后的后半部分
        left, right = head, prev
        while right:  # right 是反转后的链表的头
            if left.val != right.val:
                return False
            left = left.next
            right = right.next

        return True

```

示例20.删除链表元素

http://dsbpython.openjudge.cn/dspythonbook/P0020/

程序填空，删除链表元素

```python
class Node:
  def __init__(self, data, next=None):
    self.data, self.next = data, next
class LinkList:  #循环链表
  def __init__(self):
    self.tail = None
    self.size = 0
  def isEmpty(self):
    return self.size == 0
  def pushFront(self,data):
    nd = Node(data)
    if self.tail == None:
      self.tail = nd
      nd.next = self.tail
    else:
```

```
      nd.next = self.tail.next
      self.tail.next = nd
    self.size += 1
  def pushBack(self,data):
    self.pushFront(data)
    self.tail = self.tail.next
  def popFront(self):
    if self.size == 0:
      return None
    else:
      nd = self.tail.next
      self.size -= 1
      if self.size == 0:
        self.tail = None
      else:
        self.tail.next = nd.next
    return nd.data
  def printList(self):
    if self.size > 0:
      ptr = self.tail.next
      while True:
        print(ptr.data,end = " ")
        if ptr == self.tail:
          break
        ptr = ptr.next
      print("")

  def remove(self,data):
// 在此处补充你的代码
t = int(input())
for i in range(t):
  lst = list(map(int,input().split()))
  lkList = LinkList()
  for x in lst:
    lkList.pushBack(x)
  lst = list(map(int,input().split()))
  for a in lst:
    result = lkList.remove(a)
    if result == True:
      lkList.printList()
    elif result == False:
      print("NOT FOUND")
    else:
      print("EMPTY")
  print("----------------")
```

输入

第一行为整数t，表示有t组数据。
每组数据2行
第一行是若干个整数，构成了一张链表
第二行是若干整数，是要从链表中删除的数。

输出

对每组数据第二行中的每个整数x:

1. 如果链表已经为空，则输出 "EMPTY"
2. 如果x在链表中，则将其删除，并且输出删除后的链表。如果删除后链表为空，则没输出。如果有重复元素，则删前面的。

3）如果链表不为空且x不在链表中，则输出"NOT FOUND"

样例输入

```
2
1 2 3
3 2 2 9 5 1 1 4
1
9 88 1 23
```

样例输出

```
1 2
1
NOT FOUND
NOT FOUND
NOT FOUND
EMPTY
EMPTY
----------------
NOT FOUND
NOT FOUND
EMPTY
----------------
```

来源

郭炜

程序填空题目，需要掌握“补充代码”题型，例如写出某个函数的实现代码，如 def remove(self,data):

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


class LinkList:  # 循环链表
    def __init__(self):
        self.tail = None
        self.size = 0

    def isEmpty(self):
```

```
        return self.size == 0

    def pushFront(self, data):
        nd = Node(data)
        if self.tail == None:
            self.tail = nd
            nd.next = self.tail
        else:
            nd.next = self.tail.next
            self.tail.next = nd
        self.size += 1

    def pushBack(self, data):
        self.pushFront(data)
        self.tail = self.tail.next

    def popFront(self):
        if self.size == 0:
            return None
        else:
            nd = self.tail.next
            self.size -= 1
            if self.size == 0:
                self.tail = None
            else:
                self.tail.next = nd.next
        return nd.data

    def printList(self):
        if self.size > 0:
            ptr = self.tail.next
            while True:
                print(ptr.data, end=" ")
                if ptr == self.tail:
                    break
                ptr = ptr.next
            print("")

    def remove(self, data):  # 填空：实现函数
        if self.size == 0:
            return None
        else:
            ptr = self.tail
            while ptr.next.data != data:
                ptr = ptr.next
                if ptr == self.tail:
                    return False
            self.size -= 1
            if ptr.next == self.tail:
                self.tail = ptr
            ptr.next = ptr.next.next
            return True
```

```python


t = int(input())
for i in range(t):
    lst = list(map(int, input().split()))
    lkList = LinkList()
    for x in lst:
        lkList.pushBack(x)
    lst = list(map(int, input().split()))
    for a in lst:
        result = lkList.remove(a)
        if result == True:
            lkList.printList()
        elif result == False:
            print("NOT FOUND")
        else:
            print("EMPTY")
    print("----------------")

"""
样例输入
2
1 2 3
3 2 2 9 5 1 1 4
1
9 88 1 23

样例输出
1 2
1
NOT FOUND
NOT FOUND
NOT FOUND
EMPTY
EMPTY
----------------
NOT FOUND
NOT FOUND
EMPTY
----------------
"""
```

示例4.插入链表元素

http://dsbpython.openjudge.cn/2024allhw/004/

很遗憾，一意孤行的Y君没有理会你告诉他的饮食计划并很快吃完了他的粮食储备。
但好在他捡到了一张校园卡，凭这个他可以偷偷混入领取物资的队伍。
为了不被志愿者察觉自己是只猫，他想要插到队伍的最中央。（插入后若有偶数个元素则选取靠后的位置）
于是他又找到了你，希望你能帮他修改志愿者写好的代码，在发放顺序的中间加上他的学号6。

你虽然不理解志愿者为什么要用链表来写这份代码，但为了不被发现只得在此基础上进行修改：

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

class LinkList:
  def __init__(self):
    self.head = None

  def initList(self, data):
    self.head = Node(data[0])
    p = self.head
    for i in data[1:]:
      node = Node(i)
      p.next = node
      p = p.next

  def insertCat(self):
// 在此处补充你的代码
########
  def printLk(self):
    p = self.head
    while p:
      print(p.data, end=" ")
      p = p.next
    print()

lst = list(map(int,input().split()))
lkList = LinkList()
lkList.initList(lst)
lkList.insertCat()
lkList.printLk()
```

输入

一行，若干个整数，组成一个链表。

输出

一行，在链表中间位置插入数字6后得到的新链表

样例输入

```
### 样例输入1
8 1 0 9 7 5
### 样例输入2
1 2 3
```

样例输出

```
### 样例输出1
8 1 0 6 9 7 5
### 样例输出2
1 2 6 3
```

来源

Lou Yuke

程序填空题目，需要掌握“补充代码”题型，例如写出某个函数的实现代码，如 def insertCat(self):

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

class LinkList:
    def __init__(self):
        self.head = None

    def initList(self, data):
        self.head = Node(data[0])
        p = self.head
        for i in data[1:]:
            node = Node(i)
            p.next = node
            p = p.next

    def insertCat(self):
        # 计算链表的长度
        length = 0
        p = self.head
        while p:
            length += 1
            p = p.next

        # 找到插入位置
        position = length // 2 if length % 2 == 0 else (length // 2) + 1
        p = self.head
        for _ in range(position - 1):
            p = p.next

        # 在插入位置处插入数字6
        node = Node(6)
        node.next = p.next
        p.next = node

    def printLk(self):
        p = self.head
        while p:
            print(p.data, end=" ")
```

```
            p = p.next
        print()

lst = list(map(int, input().split()))
lkList = LinkList()
lkList.initList(lst)
lkList.insertCat()
lkList.printLk()

"""
### 样例输入1
8 1 0 9 7 5
### 样例输入2
1 2 3

### 样例输出1
8 1 0 6 9 7 5
### 样例输出2
1 2 6 3
"""
```

6 The Stack Abstract Data Type

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the “top.” Stacks are ordered LIFO. The stack operations are given below.

- `Stack()` creates a new stack that is empty. It needs no parameters and returns an empty stack.
- `push(item)` adds a new item to the top of the stack. It needs the item and returns nothing.
- `pop()` removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
- `peek()` returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
- `isEmpty()` tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items on the stack. It needs no parameters and returns an integer.

For example, if `s` is a stack that has been created and starts out empty, then Table 1 shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

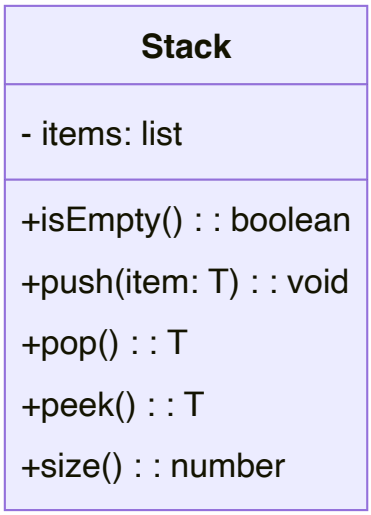
Table 1: Sample Stack Operations

Stack Operation	Stack Contents	Return Value
`s.isEmpty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.isEmpty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

6.1 Implementing a Stack in Python

Now that we have clearly defined the stack as an abstract data type we will turn our attention to using Python to implement the stack. Recall that when we give an abstract data type a physical implementation we refer to the implementation as a data structure.

As we described in Chapter 1, in Python, as in any object-oriented programming language, the implementation of choice for an abstract data type such as a stack is the creation of a new class. The stack operations are implemented as methods. Further, to implement a stack, which is a collection of elements, it makes sense to utilize the power and simplicity of the primitive collections provided by Python. We will use a list.




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

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

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

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

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

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

s = Stack()

print(s.is_empty())
s.push(4)
s.push('dog')

print(s.peek())
s.push(True)
print(s.size())
print(s.is_empty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())

```

```
"""
True
dog
3
False
8.4
True
2
"""
```

要求自己会用类实现Stack，但是实际编程时候，直接使用系统的list更好。

```
#function rev_string(my_str) that uses a stack to reverse the characters in a string.
def rev_string(my_str):
    s = [] # Stack()
    rev = []
    for c in my_str:
        s.append(c) # push(c)

    #while not s.is_empty():
    while s:
        rev.append(s.pop())
    return "".join(rev)

test_string = "cutie"

print(rev_string(test_string))

# output: eituc

```

6.2 匹配括号

We now turn our attention to using stacks to solve real computer science problems. You have no doubt written arithmetic expressions such as

(5+6)∗(7+8)/(4+3)

where parentheses are used to order the performance of operations. You may also have some experience programming in a language such as Lisp with constructs like

```
(defun square(n)
     (* n n))
```

This defines a function called `square` that will return the square of its argument `n`. Lisp is notorious for using lots and lots of parentheses.

In both of these examples, parentheses must appear in a balanced fashion. **Balanced parentheses** means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Consider the following correctly balanced strings of parentheses:

```
(()()()())

(((())))

(()((())()))
```

Compare those with the following, which are not balanced:

```
(((((((())

())) 

(()()(()
```

The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

The challenge then is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. To solve this problem we need to make an important observation. As you process symbols from left to right, the most recent opening parenthesis must match the next closing symbol (see Figure 4). Also, the first opening symbol processed may have to wait until the very last symbol for its match. Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out. This is a clue that stacks can be used to solve the problem.

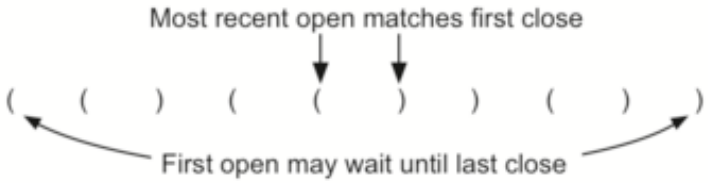


Figure 4: Matching Parentheses

```python
#returns a boolean result as to whether the string of parentheses is balanced
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol == "(":
            s.append(symbol) # push(symbol)
        else:
```

```
            #if s.is_empty():
            if not s:
                balanced = False
            else:
                s.pop()
        index = index + 1

    #if balanced and s.is_empty():
    if balanced and not s:
        return True
    else:
        return False

print(par_checker('((()))'))
print(par_checker('(()'))

# True
# False
```

1 Balanced Symbols (A General Case)

The balanced parentheses problem shown above is a specific case of a more general situation that arises in many programming languages. The general problem of balancing and nesting different kinds of opening and closing symbols occurs frequently. For example, in Python square brackets, `[` and `]`, are used for lists; curly braces, `{` and `}`, are used for dictionaries; and parentheses, `(` and `)`, are used for tuples and arithmetic expressions. It is possible to mix symbols as long as each maintains its own open and close relationship. Strings of symbols such as

```
{ { ( [ ] [ ] ) } ( ) }

[ [ { { ( ( ) ) } } ] ]

[ ] [ ] [ ] ( ) { }
```

are properly balanced in that not only does each opening symbol have a corresponding closing symbol, but the types of symbols match as well.

Compare those with the following strings that are not balanced:

```
( [ ) ]

( ( ( ) ] ) )

[ { ( ) ]
```

The simple parentheses checker from the previous section can easily be extended to handle these new types of symbols. Recall that each opening symbol is simply pushed on the stack to wait for the matching closing symbol to appear later in the sequence. When a closing symbol does appear, the only difference is that we must check to be sure that it correctly matches the type of the opening symbol on top of the stack. If the two symbols do not match, the string is not balanced. Once again, if the entire string is processed and nothing is left on the stack, the string is correctly balanced.

```python
def par_checker(symbol_string):
    s = [] # Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol in "([{":
            s.append(symbol) # push(symbol)
        else:
            top = s.pop()
            if not matches(top, symbol):
                balanced = False
        index += 1
        #if balanced and s.is_empty():
        if balanced and not s:
            return True
        else:
            return False

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

print(par_checker('{{}}[]]'))

# output: False
```

练习OJ03704: 括号匹配问题

stack, http://cs101.openjudge.cn/practice/03704

在某个字符串（长度不超过100）中有左括号、右括号和大小写字母；规定（与常见的算数式子一样）任何一个左括号都从内到外与在它右边且距离最近的右括号匹配。写一个程序，找到无法匹配的左括号和右括号，输出原来字符串，并在下一行标出不能匹配的括号。不能匹配的左括号用"$"标注，不能匹配的右括号用"?"标注.

输入

输入包括多组数据，每组数据一行，包含一个字符串，只包含左右括号和大小写字母，**字符串长度不超过100**
注意：cin.getline(str,100)最多只能输入99个字符！

输出

对每组输出数据，输出两行，第一行包含原始输入字符，第二行由"$","?"和空格组成，"$"和"?"表示与之对应的左括号和右括号不能匹配。

样例输入

```
((ABCD(x)
)(rttyy())sss)(
```

样例输出

```
((ABCD(x)
$$
)(rttyy())sss)(
?            ?$
```

```python
# https://www.cnblogs.com/huashanqingzhu/p/6546598.html

lines = []
while True:
    try:
        lines.append(input())
    except EOFError:
        break

ans = []
for s in lines:
    stack = []
    Mark = []
    for i in range(len(s)):
        if s[i] == '(':
            stack.append(i)
            Mark += ' '
        elif s[i] == ')':
            if len(stack) == 0:
                Mark += '?'
            else:
                Mark += ' '
                stack.pop()
        else:
            Mark += ' '

    while(len(stack)):
        Mark[stack[-1]] = '$'
        stack.pop()

    print(s)
    print(''.join(map(str, Mark)))
```

6.3 进制转换

1 将十进制数转换成二进制数

In your study of computer science, you have probably been exposed in one way or another to the idea of a binary number. Binary representation is important in computer science since all values stored within a computer exist as a string of binary digits, a string of 0s and 1s. Without the ability to convert back and forth between common representations and binary numbers, we would need to interact with computers in very awkward ways.

Integer values are common data items. They are used in computer programs and computation all the time. We learn about them in math class and of course represent them using the decimal number system, or base 10. The decimal number 233_{10} and its corresponding binary equivalent 11101001_2 are interpreted respectively as

$$2\times10^2 + 3\times10^1 + 3\times10^0$$

and

$$1\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 0\times2^1 + 1\times2^0$$

But how can we easily convert integer values into binary numbers? The answer is an algorithm called “Divide by 2” that uses a stack to keep track of the digits for the binary result.

The Divide by 2 algorithm assumes that we start with an integer greater than 0. A simple iteration then continually divides the decimal number by 2 and keeps track of the remainder. The first division by 2 gives information as to whether the value is even or odd. An even value will have a remainder of 0. It will have the digit 0 in the ones place. An odd value will have a remainder of 1 and will have the digit 1 in the ones place. We think about building our binary number as a sequence of digits; the first remainder we compute will actually be the last digit in the sequence. As shown in Figure 5, we again see the reversal property that signals that a stack is likely to be the appropriate data structure for solving the problem.

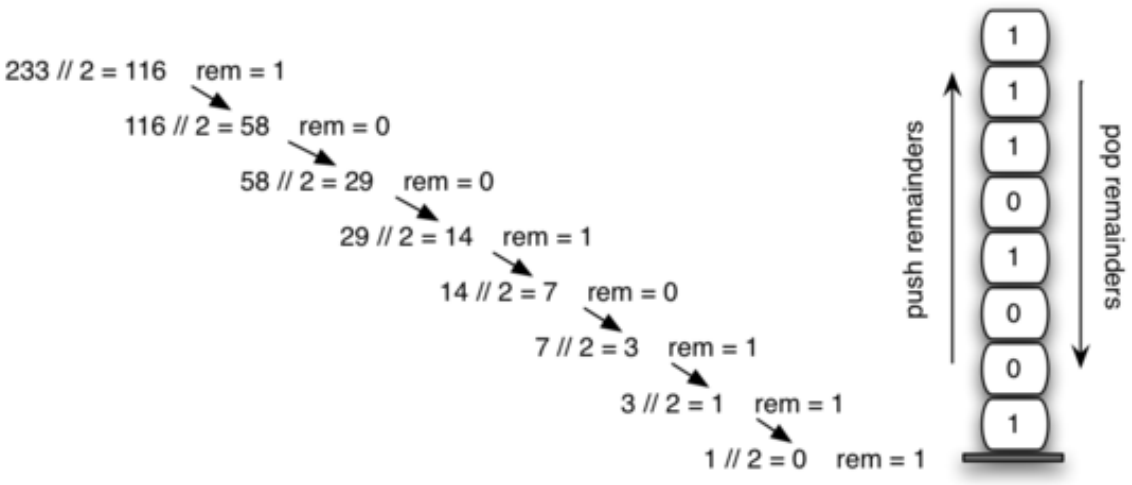


Figure 5: Decimal-to-Binary Conversion

```python
def divide_by_2(dec_num):
    rem_stack = [] # Stack()

    while dec_num > 0:
        rem  = dec_num % 2
        rem_stack.append(rem) # push(rem)
        dec_num = dec_num // 2

    bin_string = ""
    #while not rem_stack.is_empty():
    while rem_stack:
        bin_string = bin_string + str(rem_stack.pop())

    return bin_string

print(divide_by_2(233))

# output: 11101001
```

```python
def base_converter(dec_num, base):
    digits = "0123456789ABCDEF"

    rem_stack = [] # Stack()

    while dec_num > 0:
        rem = dec_num % base
        #rem_stack.push(rem)
        rem_stack.append(rem)
        dec_num = dec_num // base

    new_string = ""
    #while not rem_stack.is_empty():
    while rem_stack:
        new_string = new_string + digits[rem_stack.pop()]

    return new_string

print(base_converter(25, 2))
print(base_converter(2555, 16))

# 11001
# 9FB
```

练习OJ02734: 十进制到八进制

http://cs101.openjudge.cn/practice/02734/

把一个十进制正整数转化成八进制。

输入

一行，仅含一个十进制表示的整数a(0 < a < 65536)。

输出

一行，a的八进制表示。

样例输入

```
9
```

样例输出

```
11
```

使用栈来实现十进制到八进制的转换可以通过不断除以8并将余数压入栈中的方式来实现。然后，将栈中的元素依次出栈，构成八进制数的各个位。

```python
decimal = int(input())  # 读取十进制数

# 创建一个空栈
stack = []

# 特殊情况：如果输入的数为0，直接输出0
if decimal == 0:
    print(0)
else:
    # 不断除以8，并将余数压入栈中
    while decimal > 0:
        remainder = decimal % 8
        stack.append(remainder)
        decimal = decimal // 8

    # 依次出栈，构成八进制数的各个位
    octal = ""
    while stack:
        octal += str(stack.pop())

    print(octal)
```

6.4 中序、前序和后序表达式

When you write an arithmetic expression such as B * C, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator * appears between them in the expression. This type of notation is referred to as **infix** since the operator is *in between* the two operands that it is working on.

Consider another infix example, A + B * C. The operators + and * still appear between the operands, but there is a problem. Which operands do they work on? Does the + work on A and B or does the * take B and C? The expression seems ambiguous.

In fact, you have been reading and writing these types of expressions for a long time and they do not cause you any problem. The reason for this is that you know something about the operators + and *. Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Let's interpret the troublesome expression A + B * C using operator precedence. B and C are multiplied first, and A is then added to that result. (A + B) * C would force the addition of A and B to be done first before the multiplication. In expression A + B + C, by precedence (via associativity), the leftmost + would be done first.

Although all this may be obvious to you, remember that computers need to know exactly what operators to perform and in what order. One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a **fully parenthesized** expression. This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.

The expression A + B * C + D can be rewritten as ((A + (B * C)) + D) to show that the multiplication happens first, followed by the leftmost addition. A + B + C + D can be written as (((A + B) + C) + D) since the addition operations associate from left to right.

There are two other very important expression formats that may not seem obvious to you at first. Consider the infix expression A + B. What would happen if we moved the operator before the two operands? The resulting expression would be + A B. Likewise, we could move the operator to the end. We would get A B +. These look a bit strange.

These changes to the position of the operator with respect to the operands create two new expression formats, **prefix** and **postfix**. Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands. A few more examples should help to make this a bit clearer (see Table 2).

A + B * C would be written as + A * B C in prefix. The multiplication operator comes immediately before the operands B and C, denoting that * has precedence over +. The addition operator then appears before the A and the result of the multiplication.

In postfix, the expression would be A B C * +. Again, the order of operations is preserved since the * appears immediately after the B and the C, denoting that * has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

Table 2: Exmples of Infix, Prefix, and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Now consider the infix expression (A + B) * C. Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when A + B was written in prefix, the addition operator was simply moved before the operands, + A B. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us * + A B C. Likewise, in postfix A B + forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then A B + C *.

Consider these three expressions again (see Table 3). Something very important has happened. Where did the parentheses go? Why don't we need them in prefix and postfix? The answer is that the operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

Table 3: An Expression with Parentheses

Infix Expression	Prefix Expression	Postfix Expression
(A + B) * C	* + A B C	A B + C *

Table 4 shows some additional examples of infix expressions and the equivalent prefix and postfix expressions. Be sure that you understand how they are equivalent in terms of the order of the operations being performed.

Table 4: Additional Examples of Infix, Prefix and Postfix

Infix Expression	Prefix Expression	Postfix Expression
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

1 Conversion of Infix Expressions to Prefix and Postfix

So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.

The first technique that we will consider uses the notion of a fully parenthesized expression that was discussed earlier. Recall that A + B * C can be written as (A + (B * C)) to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

Look at the right parenthesis in the subexpression (B * C) above. If we were to move the multiplication symbol to that position and remove the matching left parenthesis, giving us B C *, we would in effect have converted the subexpression to postfix notation. If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (see Figure 6).

Figure 6: Moving Operators to the Right for Postfix Notation

If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see Figure 7). The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.

Figure 7: Moving Operators to the Left for Prefix Notation

So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.

Here is a more complex expression: (A + B) * C - (D - E) * (F + G). Figure 8 shows the conversion to postfix and prefix notations.

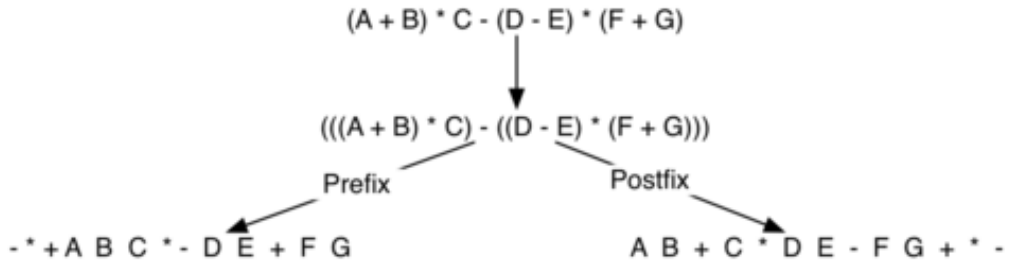


Figure 8: Converting a Complex Expression to Prefix and Postfix Notations

2 通用的中缀转后缀算法

We need to develop an algorithm to convert any infix expression to a postfix expression. To do this we will look closer at the conversion process.

Consider once again the expression A + B * C. As shown above, A B C * + is the postfix equivalent. We have already noted that the operands A, B, and C stay in their relative positions. It is only the operators that change position. Let's look again at the operators in the infix expression. The first operator that appears from left to right is +. However, in the postfix expression, + is at the end since the next operator, *, has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence. This is the case with the addition and the multiplication in this example. Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used. Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

What about (A + B) * C? Recall that A B + C * is the postfix equivalent. Again, processing this infix expression from left to right, we see + first. In this case, when we see *, + has already been placed in the result expression because it has precedence over * by virtue of the parentheses. We can now start to see how the conversion algorithm will work. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique). When that right parenthesis does appear, the operator can be popped from the stack.

As we scan the infix expression from left to right, we will use a stack to keep the operators. This will provide the reversal that we noted in the first example. The top of the stack will always be the most recently saved operator. Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and -, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on. The following steps will produce a string of tokens in postfix order.

1. Create an empty stack called `opstack` for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method `split`.
3. Scan the token list from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the `opstack`.
    - If the token is a right parenthesis, pop the `opstack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.

- If the token is an operator, *, /, +, or -, push it on the `opstack`. However, first remove any operators already on the `opstack` that have higher or equal precedence and append them to the output list.

4. When the input expression has been completely processed, check the `opstack`. Any operators still on the stack can be removed and appended to the end of the output list.

```
def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = [] # Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            #opStack.push(token)
            opStack.append(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            #while (not opStack.is_empty()) and (prec[opStack.peek()] >=
prec[token]):
            while opStack and (prec[opStack[-1]] >= prec[token]):
                postfixList.append(opStack.pop())
            #opStack.push(token)
            opStack.append(token)

    #while not opStack.is_empty():
    while opStack:
        postfixList.append(opStack.pop())
    return " ".join(postfixList)

print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))

print(infixToPostfix("( A + B ) * ( C + D )"))
print(infixToPostfix("( A + B ) * C"))
print(infixToPostfix("A + B * C"))

"""
```

```
A B * C D * +
A B + C * D E - F G + * -
A B + C D + *
A B + C *
A B C * +
"""
```

练习OJ24591:中序表达式转后序表达式

http://cs101.openjudge.cn/practice/24591/

中序表达式是运算符放在两个数中间的表达式。乘、除运算优先级高于加减。可以用"()"来提升优先级 --- 就是小学生写的四则算术运算表达式。中序表达式可用如下方式递归定义：

1）一个数是一个中序表达式。该表达式的值就是数的值。

2. 若a是中序表达式，则"(a)"也是中序表达式(引号不算)，值为a的值。
3. 若a,b是中序表达式，c是运算符，则"acb"是中序表达式。"acb"的值是对a和b做c运算的结果，且a是左操作数，b是右操作数。

输入一个中序表达式，要求转换成一个后序表达式输出。

输入

第一行是整数n(n<100)。接下来n行，每行一个中序表达式，数和运算符之间没有空格，长度不超过700。

输出

对每个中序表达式，输出转成后序表达式后的结果。后序表达式的数之间、数和运算符之间用一个空格分开。

样例输入

```
3
7+8.3
3+4.5*(7+2)
(3)*((3+4)*(2+3.5)/(4+5))
```

样例输出

```
7 8.3 +
3 4.5 7 2 + * +
3 3 4 + 2 3.5 + * 4 5 + / *
```

来源: Guo wei

Shunting yard algorightm（调度场算法）是一种用于将中缀表达式转换为后缀表达式的算法。它由荷兰计算机科学家 Edsger Dijkstra 在1960年代提出，用于解析和计算数学表达式。

Edsger W. Dijkstra

60 languages

Article Talk Read Edit View history Tools

From Wikipedia, the free encyclopedia

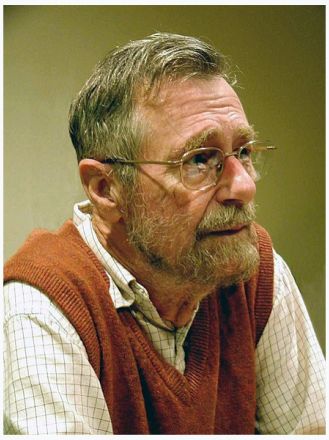
Edsger Wybe Dijkstra (/ˈdaɪkstrə/ *DYKE-strə*; Dutch: [ˈɛtsxər ˈʊibə ˈdɛikstra] ⓘ; 11 May 1930 – 6 August 2002) was a Dutch computer scientist, programmer, software engineer, and science essayist.[1][2]

Born in Rotterdam, the Netherlands, Dijkstra studied mathematics and physics and then theoretical physics at the University of Leiden. Adriaan van Wijngaarden offered him a job as the first computer programmer in the Netherlands at the Mathematical Center in Amsterdam, where he worked from 1952 until 1962. He formulated and solved the shortest path problem in 1956, and in 1960 developed the first compiler for the programming language ALGOL 60 in conjunction with colleague Jaap A. Zonneveld. In 1962 he moved to Eindhoven, and later to Nuenen, where he became a professor in the Mathematics Department at the Technische Hogeschool Eindhoven. In the late 1960s he built the THE multiprogramming system, which influenced the designs of subsequent systems through its use of software-based paged virtual memory. Dijkstra joined Burroughs Corporation as its sole research fellow in August 1973. The Burroughs years saw him at his most prolific in output of research articles. He wrote nearly 500 documents in the "EWD" series, most of them technical reports, for private circulation within a select group.

Dijkstra accepted the Schlumberger Centennial Chair in the Computer Science Department at the University of Texas at Austin in 1984, working in Austin, Texas until his retirement in November 1999. He and his wife returned from Austin to his original house in Nuenen, where he died on 6 August 2002 after a long struggle with cancer.[3]

He received the 1972 Turing Award for fundamental contributions to developing structured programming languages. Shortly before his death, he received the ACM PODC Influential Paper Award in distributed computing for his work on self-stabilization of program computation. This annual award was renamed the Dijkstra Prize the following year, in his honor.

Edsger W. Dijkstra



Dijkstra in 2002

Born	11 May 1930 Rotterdam, Netherlands
Died	6 August 2002 (aged 72) Nuenen, Netherlands
Citizenship	Netherlands
Education	Leiden University (B.S., M.S.) University of Amsterdam (Ph.D.)
Spouse	Maria (Ria) C. Debets
Awards	Turing Award (1972)

Shunting Yard 算法的主要思想是使用两个栈（运算符栈和输出栈）来处理表达式的符号。算法按照运算符的优先级和结合性，将符号逐个处理并放置到正确的位置。最终，输出栈中的元素就是转换后的后缀表达式。

以下是 Shunting Yard 算法的基本步骤：

1. 初始化运算符栈和输出栈为空。
2. 从左到右遍历中缀表达式的每个符号。
    - 如果是操作数（数字），则将其添加到输出栈。
    - 如果是左括号，则将其推入运算符栈。
    - 如果是运算符：
        - 如果运算符的优先级大于运算符栈顶的运算符，或者运算符栈顶是左括号，则将当前运算符推入运算符栈。
        - 否则，将运算符栈顶的运算符弹出并添加到输出栈中，直到满足上述条件（或者运算符栈为空）。
        - 将当前运算符推入运算符栈。
    - 如果是右括号，则将运算符栈顶的运算符弹出并添加到输出栈中，直到遇到左括号。将左括号弹出但不添加到输出栈中。
3. 如果还有剩余的运算符在运算符栈中，将它们依次弹出并添加到输出栈中。
4. 输出栈中的元素就是转换后的后缀表达式。

接收浮点数，是number buffer技巧。

```python
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2}
    stack = []
    postfix = []
    number = ''

    for char in expression:
        if char.isnumeric() or char == '.':
            number += char
        else:
            if number:
                num = float(number)
                postfix.append(int(num) if num.is_integer() else num)
                number = ''
            if char in '+-*/':
                while stack and stack[-1] in '+-*/' and precedence[char] <= precedence[stack[-1]]:
                    postfix.append(stack.pop())
                stack.append(char)
            elif char == '(':
                stack.append(char)
            elif char == ')':
                while stack and stack[-1] != '(':
                    postfix.append(stack.pop())
                stack.pop()

    if number:
        num = float(number)
        postfix.append(int(num) if num.is_integer() else num)

    while stack:
        postfix.append(stack.pop())

    return ' '.join(str(x) for x in postfix)

n = int(input())
for _ in range(n):
    expression = input()
    print(infix_to_postfix(expression))
```

接收数据，还可以用re处理。

```python
# 24591:中序表达式转后序表达式
# http://cs101.openjudge.cn/practice/24591/

def inp(s):
    #s=input().strip()
    import re
    s=re.split(r'([\(\)\+\-\*\/])',s)
    s=[item for item in s if item.strip()]
    return s

exp = "(3)*((3+4)*(2+3.5)/(4+5)) "
print(inp(exp))
```

3 Postfix Evaluation

As a final stack example, we will consider the evaluation of an expression that is already in postfix notation. In this case, a stack is again the data structure of choice. However, as you scan the postfix expression, it is the operands that must wait, not the operators as in the conversion algorithm above. Another way to think about the solution is that whenever an operator is seen on the input, the two most recent operands will be used in the evaluation.

```python
def postfixEval(postfixExpr):
    operandStack = []
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.append(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.append(result)
    return operandStack.pop()

def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 / op2
    elif op == "+":
        return op1 + op2
    else:
        return op1 - op2

print(postfixEval('7 8 + 3 2 + /'))

# output: 3.0
```

练习OJ24588: 后序表达式求值

http://cs101.openjudge.cn/practice/24588/

后序表达式由操作数和运算符构成。操作数是整数或小数，运算符有 + - * / 四种，其中 * / 优先级高于 + -。后序表达式可用如下方式递归定义：

1. 一个操作数是一个后序表达式。该表达式的值就是操作数的值。
2. 若a,b是后序表达式，c是运算符，则"a b c"是后序表达式。“a b c”的值是 (a) c (b),即对a和b做c运算，且a是第一个操作数，b是第二个操作数。下面是一些后序表达式及其值的例子(操作数、运算符之间用空格分隔)：

3.4　　值为：3.4
5　　值为：5
5 3.4 +　　值为：5 + 3.4
5 3.4 + 6 /　值为：(5+3.4)/6
5 3.4 + 6 * 3 + 值为：(5+3.4)*6+3

输入

第一行是整数n(n<100)，接下来有n行，每行是一个后序表达式，长度不超过1000个字符

输出

对每个后序表达式，输出其值，保留小数点后面2位

样例输入

```
3
5 3.4 +
5 3.4 + 6 /
5 3.4 + 6 * 3 +
```

样例输出

```
8.40
1.40
53.40
```

来源: Guo wei

要解决这个问题，需要理解如何计算后序表达式。后序表达式的计算可以通过使用一个栈来完成，按照以下步骤：

1. 从左到右扫描后序表达式。
2. 遇到数字时，将其压入栈中。

3. 遇到运算符时，从栈中弹出两个数字，先弹出的是右操作数，后弹出的是左操作数。将这两个数字进行相应的运算，然后将结果压入栈中。
4. 当表达式扫描完毕时，栈顶的数字就是表达式的结果。

```python
def evaluate_postfix(expression):
    stack = []
    tokens = expression.split()

    for token in tokens:
        if token in '+-*/':
            # 弹出栈顶的两个元素
            right_operand = stack.pop()
            left_operand = stack.pop()
            # 执行运算
            if token == '+':
                stack.append(left_operand + right_operand)
            elif token == '-':
                stack.append(left_operand - right_operand)
            elif token == '*':
                stack.append(left_operand * right_operand)
            elif token == '/':
                stack.append(left_operand / right_operand)
        else:
            # 将操作数转换为浮点数后入栈
            stack.append(float(token))

    # 栈顶元素就是表达式的结果
    return stack[0]

# 读取输入行数
n = int(input())

# 对每个后序表达式求值
for _ in range(n):
    expression = input()
    result = evaluate_postfix(expression)
    # 输出结果，保留两位小数
    print(f"{result:.2f}")
```

这个程序将读取输入行数，然后对每行输入的后序表达式求值，并按要求保留两位小数输出结果。

6.5 经典八皇后用递归或者栈实现

示例OJ02754: 八皇后

dfs and similar, http://cs101.openjudge.cn/practice/02754

会下国际象棋的人都很清楚：皇后可以在横、竖、斜线上不限步数地吃掉其他棋子。如何将8个皇后放在棋盘上（有8 * 8个方格），使它们谁也不能被吃掉！这就是著名的八皇后问题。
对于某个满足要求的8皇后的摆放方法，定义一个皇后串a与之对应，即$a = b_1b_2...b_8$,其中b_i为相应摆法中第i行皇后所处的列数。已经知道8皇后问题一共有92组解（即92个不同的皇后串）。
给出一个数b，要求输出第b个串。串的比较是这样的：皇后串x置于皇后串y之前，当且仅当将x视为整数时比y小。

八皇后是一个古老的经典问题：**如何在一张国际象棋的棋盘上，摆放8个皇后，使其任意两个皇后互相不受攻击。**该问题由一位德国**国际象棋排局家 Max Bezzel** 于 1848年提出。严格来说，那个年代，还没有“德国”这个国家，彼时称作“普鲁士”。1850年，**Franz Nauck** 给出了第一个解，并将其扩展成了“ **n皇后** ”问题，即**在一张 n** x **n 的棋盘上，如何摆放 n 个皇后，使其两两互不攻击**。历史上，八皇后问题曾惊动过“数学王子”高斯(Gauss)，而且正是 Franz Nauck 写信找高斯请教的。

输入

第1行是测试数据的组数n，后面跟着n行输入。每组测试数据占1行，包括一个正整数b(1 ≤ b ≤ 92)

输出

输出有n行，每行输出对应一个输入。输出应是一个正整数，是对应于b的皇后串。

样例输入

```
2
1
92
```

样例输出

```
15863724
84136275
```

先给出两个dfs回溯实现的八皇后，接着给出两个stack迭代实现的八皇后。

八皇后思路：回溯算法通过尝试不同的选择，逐步构建解决方案，并在达到某个条件时进行回溯，以找到所有的解决方案。从第一行第一列开始放置皇后，然后在每一行的不同列都放置，如果与前面不冲突就继续，有冲突则回到上一行继续下一个可能性。

```python
def solve_n_queens(n):
    solutions = []  # 存储所有解决方案的列表
    queens = [-1] * n  # 存储每一行皇后所在的列数

    def backtrack(row):
        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col):  # 检查当前位置是否合法
                    queens[row] = col  # 在当前行放置皇后
                    backtrack(row + 1)  # 递归处理下一行
```

```
                    queens[row] = -1  # 回溯，撤销当前行的选择

    def is_valid(row, col):
        for r in range(row):
            if queens[r] == col or abs(row - r) == abs(col - queens[r]):
                return False
        return True

    backtrack(0)  # 从第一行开始回溯

    return solutions


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
        return None
    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string


test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

```
def is_safe(board, row, col):
    # 检查当前位置是否安全
    # 检查同一列是否有皇后
    for i in range(row):
        if board[i] == col:
            return False
    # 检查左上方是否有皇后
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
        if board[i] == j:
            return False
        i -= 1
        j -= 1
    # 检查右上方是否有皇后
    i = row - 1
    j = col + 1
    while i >= 0 and j < 8:
        if board[i] == j:
            return False
        i -= 1
```

```
        j += 1
    return True

def queen_dfs(board, row):
    if row == 8:
        # 找到第b个解，将解存储到result列表中
        ans.append(''.join([str(x+1) for x in board]))
        return
    for col in range(8):
        if is_safe(board, row, col):
            # 当前位置安全，放置皇后
            board[row] = col
            # 继续递归放置下一行的皇后
            queen_dfs(board, row + 1)
            # 回溯，撤销当前位置的皇后
            board[row] = 0

ans = []
queen_dfs([None]*8, 0)
#print(ans)
for _ in range(int(input())):
    print(ans[int(input()) - 1])
```

如果要使用栈来实现八皇后问题，可以采用迭代的方式，模拟递归的过程。在每一步迭代中，使用栈来保存状态，并根据规则进行推进和回溯。

```
def queen_stack(n):
    stack = []  # 用于保存状态的栈
    solutions = [] # 存储所有解决方案的列表

    stack.append((0, []))  # 初始状态为第一行，所有列都未放置皇后,栈中的元素是 (row, queens) 的元组

    while stack:
        row, cols = stack.pop() # 从栈中取出当前处理的行数和已放置的皇后位置
        if row == n:    # 找到一个合法解决方案
            solutions.append(cols)
        else:
            for col in range(n):
                if is_valid(row, col, cols): # 检查当前位置是否合法
                    stack.append((row + 1, cols + [col]))

    return solutions

def is_valid(row, col, queens):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True

```

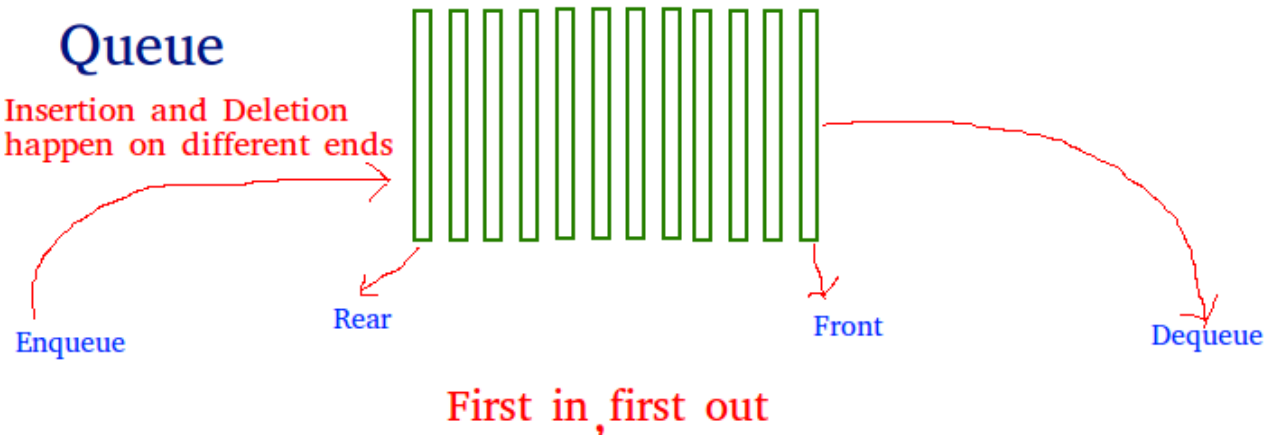
```python

# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = queen_stack(8)
    if b > len(solutions):
        return None
    b = len(solutions) + 1 - b

    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string

test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)
```

```python
def solve_n_queens(n):
    stack = []  # 用于保存状态的栈
    solutions = []  # 存储所有解决方案的列表

    stack.append((0, [-1] * n))  # 初始状态为第一行，所有列都未放置皇后

    while stack:
        row, queens = stack.pop()

        if row == n:  # 找到一个合法解决方案
            solutions.append(queens.copy())
        else:
            for col in range(n):
                if is_valid(row, col, queens):  # 检查当前位置是否合法
                    new_queens = queens.copy()
                    new_queens[row] = col  # 在当前行放置皇后
                    stack.append((row + 1, new_queens))  # 推进到下一行

    return solutions


def is_valid(row, col, queens):
    for r in range(row):
        if queens[r] == col or abs(row - r) == abs(col - queens[r]):
            return False
    return True


# 获取第 b 个皇后串
def get_queen_string(b):
    solutions = solve_n_queens(8)
    if b > len(solutions):
```

```
        return None
    b = len(solutions) + 1 - b

    queen_string = ''.join(str(col + 1) for col in solutions[b - 1])
    return queen_string


test_cases = int(input())  # 输入的测试数据组数
for _ in range(test_cases):
    b = int(input())  # 输入的 b 值
    queen_string = get_queen_string(b)
    print(queen_string)

```

7 The Queue Abstract Data Type

Like a stack, the queue is a linear data structure that stores items in a First In First Out (FIFO) manner. With a queue, the least recently added item is removed first. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.



Operations associated with queue are:

- Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition – Time Complexity : O(1)
- Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition – Time Complexity : O(1)
- Front: Get the front item from queue – Time Complexity : O(1)
- Rear: Get the last item from queue – Time Complexity : O(1)

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- `Queue()` creates a new queue that is empty. It needs no parameters and returns an empty queue.
- `enqueue(item)` adds a new item to the rear of the queue. It needs the item and returns nothing.
- `dequeue()` removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- `isEmpty()` tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the queue. It needs no parameters and returns an integer.

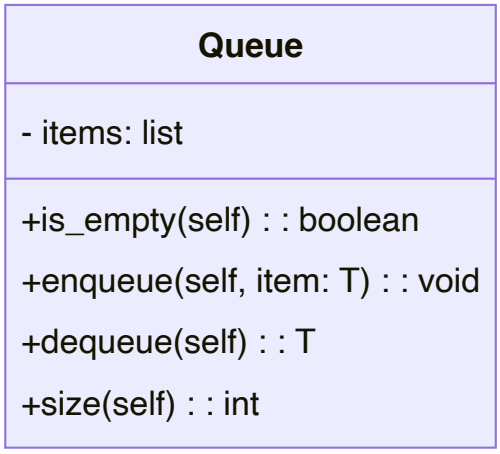
As an example, if we assume that `q` is a queue that has been created and is currently empty, then Table 1 shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by dequeue.

Queue Operation	Queue Contents	Return Value
`q.isEmpty()`	`[]`	`True`
`q.enqueue(4)`	`[4]`	
`q.enqueue('dog')`	`['dog',4]`	
`q.enqueue(True)`	`[True,'dog',4]`	
`q.size()`	`[True,'dog',4]`	`3`
`q.isEmpty()`	`[True,'dog',4]`	`False`
`q.enqueue(8.4)`	`[8.4,True,'dog',4]`	
`q.dequeue()`	`[8.4,True,'dog']`	`4`
`q.dequeue()`	`[8.4,True]`	`'dog'`
`q.size()`	`[8.4,True]`	`2`

7.1 Implementing a Queue in Python

It is again appropriate to create a new class for the implementation of the abstract data type queue. As before, we will use the power and simplicity of the list collection to build the internal representation of the queue.

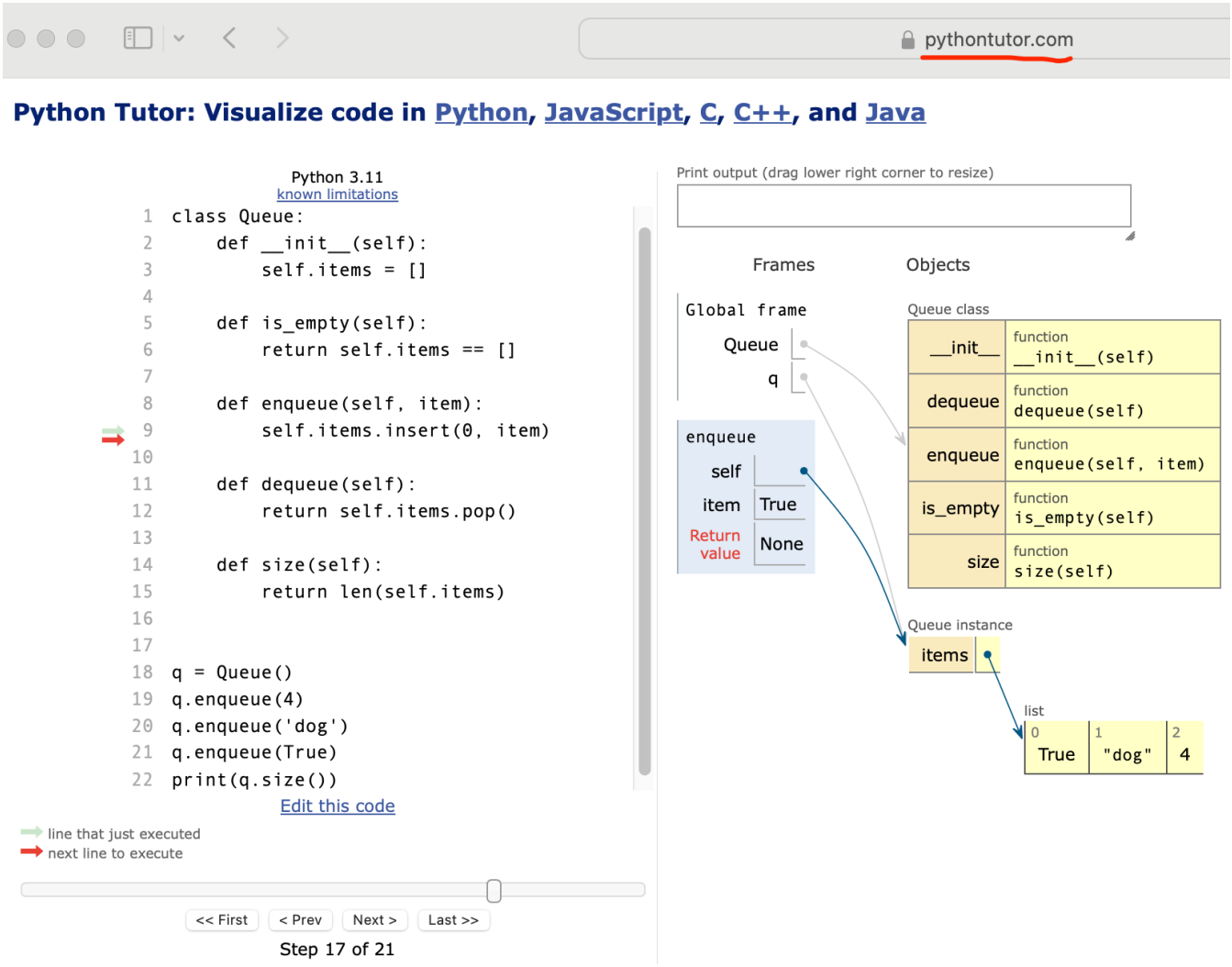
We need to decide which end of the list to use as the rear and which to use as the front. The implementation shown in Listing 1 assumes that the rear is at position 0 in the list. This allows us to use the `insert` function on lists to add new elements to the rear of the queue. The `pop` operation can be used to remove the front element (the last element of the list). Recall that this also means that enqueue will be O(n) and dequeue will be O(1).



Listing 1

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

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

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

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

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


q = Queue()

q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
print(q.items)

q.dequeue()
print(q.items)
# output:
# [3, 'dog', 'hello']
```

```
# [3, 'dog']
```



Q: Suppose you have the following series of queue operations.

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

What items are left on the queue? (B)

A. 'hello', 'dog'
B. 'dog', 3
C. 'hello', 3
D. 'hello', 'dog', 3

7.2 练习OJ02746: 约瑟夫问题

implementation, http://cs101.openjudge.cn/practice/02746

约瑟夫问题：有 n 只猴子，按顺时针方向围成一圈选大王（编号从 1 到 n ），从第 1 号开始报数，一直数到 m，数到m的猴子退出圈外，剩下的猴子再接着从1开始报数。就这样，直到圈内只剩下一只猴子时，这个猴子就是猴王，编程求输入 n ， m后，输出最后猴王的编号。

输入

每行是用空格分开的两个整数，第一个是 n, 第二个是 m (0 < m,n <=300)。最后一行是：

0 0

输出

对于每行输入数据（最后一行除外)，输出数据也是一行，即最后猴王的编号

样例输入

```
6 2
12 4
8 3
0 0
```

样例输出

```
5
1
7
```

说明：使用 队列queue 这种数据结构会方便。它有三种实现方式，我们最常用的 list 就支持，说明，https://www.geeksforgeeks.org/queue-in-python/

用list实现队列，O(n)

```
# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = []
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.pop(0))  # O(N)
        queue.pop(0)                    # O(N)
    return queue.pop(0)                 # O(N)


while True:
    n, m = map(int, input().split())
```

```
    if {n,m} == {0}:
        break
    monkey = [i for i in range(1, n+1)]
    print(hot_potato(monkey, m-1))

```

用内置deque，O(1)

```
from collections import deque

# 先使用pop从列表中取出，如果不符合要求再append回列表，相当于构成了一个圈
def hot_potato(name_list, num):
    queue = deque()
    for name in name_list:
        queue.append(name)

    while len(queue) > 1:
        for i in range(num):
            queue.append(queue.popleft()) # O(1)
        queue.popleft()
    return queue.popleft()


while True:
    n, m = map(int, input().split())
    if {n,m} == {0}:
        break
    monkey = [i for i in range(1, n+1)]
    print(hot_potato(monkey, m-1))
```

7.3 模拟器打印机

一个更有趣的例子是模拟打印任务队列。学生向共享打印机发送打印请求，这些打印任务被存在一个队列中，并且按照先到先得的顺序执行。这样的设定可能导致很多问题。其中最重要的是，打印机能否处理一定量的工作。如果不能，学生可能会由于等待过长时间而错过
要上的课。
考虑计算机科学实验室里的这样一个场景：在任何给定的一小时内，实验室里都有约 10 个学生。他们在这一小时内最多打印 2 次，并且打印的页数从 1 到 20 不等。实验室的打印机比较老旧，每分钟只能以低质量打印 10 页。可以将打印质量调高，但是这样做会导致打印机每分钟只能打印 5 页。降低打印速度可能导致学生等待过长时间。那么，应该如何设置打印速度呢?
可以通过构建一个实验室模型来解决该问题。我们需要为学生、打印任务和打印机构建对象，如图 3-15 所示。当学生提交打印任务时，我们需要将它们加入等待列表中，该列表是打印机上的打印任务队列。当打印机执行完一个任务后，它会检查该队列，看看其中是否还有需要处理的任务。我们感兴趣的是学生平均需要等待多久才能拿到打印好的文章。这个时间等于打印任务在队列中的平均等待时间。

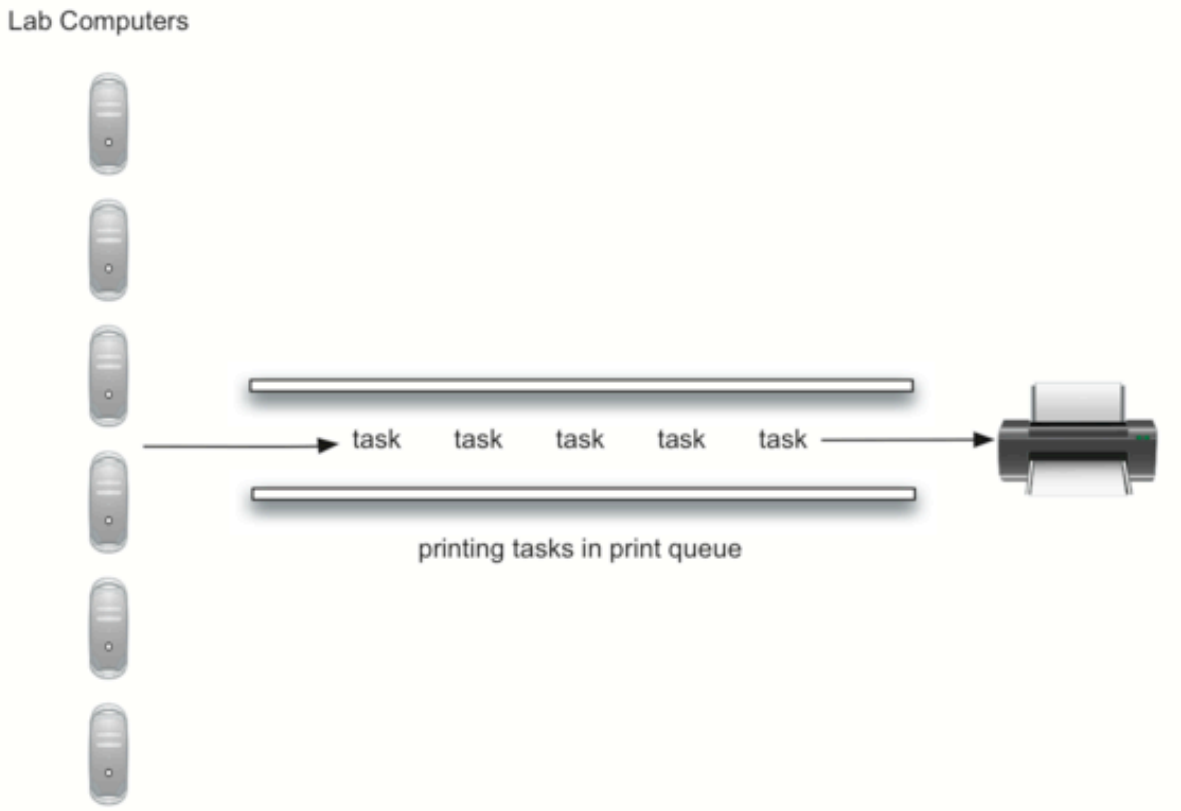


Figure 4: Computer Science Laboratory Printing Queue

在模拟时，需要应用一些概率学知识。举例来说，学生打印的文章可能有 1~20 页。如果各页数出现的概率相等，那么打印任务的实际时长可以通过 1~20 的一个随机数来模拟。
如果实验室里有 10 个学生，并且在一小时内每个人都打印两次，那么每小时平均就有 20 个打印任务。在任意一秒，创建一个打印任务的概率是多少？回答这个问题需要考虑任务与时间的比值。每小时 20 个任务相当于每 180 秒 1 个任务。

$$\frac{20\ tasks}{1\ hour} \times \frac{1\ hour}{60\ minutes} \times \frac{1\ minute}{60\ seconds} = \frac{1\ task}{180\ seconds}$$

1．主要模拟步骤
下面是主要的模拟步骤。
(1) 创建一个打印任务队列。每一个任务到来时都会有一个时间戳。一开始，队列是空的。

(2) 针对每一秒（currentSecond），执行以下操作。
❏ 是否有新创建的打印任务？如果是，以currentSecond作为其时间戳并将该任务加入到队列中。
❏ 如果打印机空闲，并且有正在等待执行的任务，执行以下操作：
■ 从队列中取出第一个任务并提交给打印机；
■ 用currentSecond减去该任务的时间戳，以此计算其等待时间；
■ 将该任务的等待时间存入一个列表，以备后用；
■ 根据该任务的页数，计算执行时间。
❏ 打印机进行一秒的打印，同时从该任务的执行时间中减去一秒。
❏ 如果打印任务执行完毕，或者说任务需要的时间减为0，则说明打印机回到空闲状态。

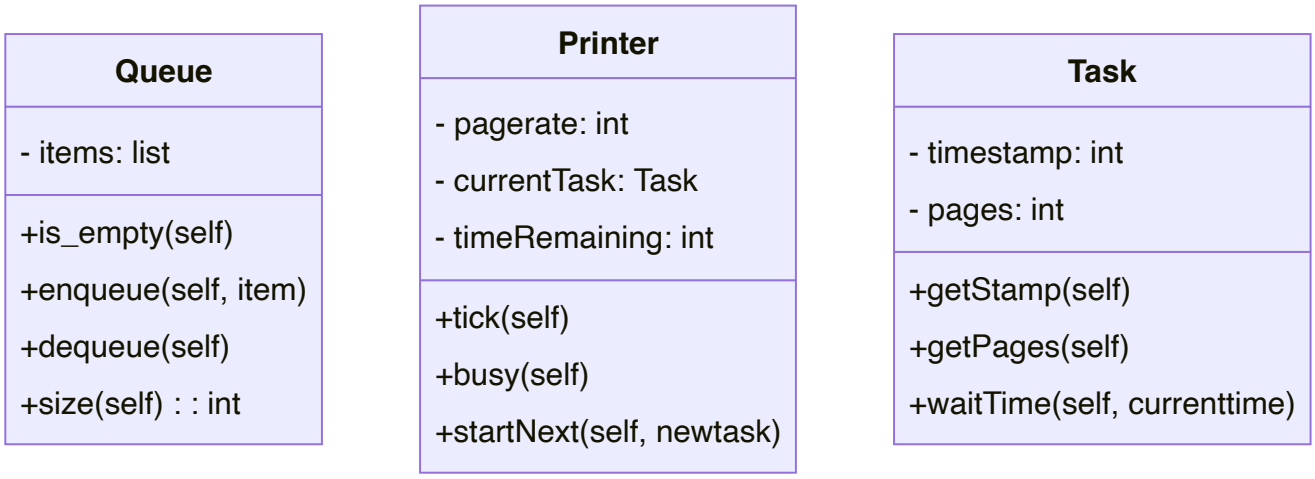
(3) 当模拟完成之后，根据等待时间列表中的值计算平均等待时间。

2. Python实现

我们创建3个类：Printer、Task和PrintQueue。它们分别模拟打印机、打印任务和队列。

Printer类需要检查当前是否有待完成的任务。如果有，那么打印机就处于工作状态（busy方法），并且其工作所需的时间可以通过要打印的页数来计算。其构造方法会初始化打印速度，即每分钟打印多少页。tick方法会减量计时，并且在执行完任务之后将打印机设置成空闲状态None。

Task类代表单个打印任务。当任务被创建时，随机数生成器会随机提供页数，取值范围是1～20。我们使用random模块中的randrange函数来生成随机数。



代码清单3-11 Printer类

```
import random

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

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

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

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

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


class Printer:
    def __init__(self, ppm):
        self.pagerate = ppm
```

```
        self.currentTask = None
        self.timeRemaining = 0

    def tick(self):
        if self.currentTask != None:
            self.timeRemaining = self.timeRemaining - 1
            if self.timeRemaining <= 0:
                self.currentTask = None

    def busy(self):
        if self.currentTask != None:
            return True
        else:
            return False

    def startNext(self, newtask):
        self.currentTask = newtask
        self.timeRemaining = newtask.getPages() * 60 / self.pagerate


class Task:
    def __init__(self, time):
        self.timestamp = time
        self.pages = random.randrange(1, 21)

    def getStamp(self):
        return self.timestamp

    def getPages(self):
        return self.pages

    def waitTime(self, currenttime):
        return currenttime - self.timestamp


def simulation(numSeconds, pagesPerMinute):
    labprinter = Printer(pagesPerMinute)
    printQueue = Queue()
    waitingtimes = []

    for currentSecond in range(numSeconds):

        if newPrintTask():
            task = Task(currentSecond)
            printQueue.enqueue(task)

        if (not labprinter.busy()) and (not printQueue.is_empty()):
            nexttask = printQueue.dequeue()
            waitingtimes.append(nexttask.waitTime(currentSecond))
            labprinter.startNext(nexttask)

        labprinter.tick()
```

```
75
76      averageWait = sum(waitingtimes) / len(waitingtimes)
77      print("Average Wait %6.2f secs %3d tasks remaining." % (averageWait,
    printQueue.size()))
78
79
80  def newPrintTask():
81      num = random.randrange(1, 181)
82      if num == 180:
83          return True
84      else:
85          return False
86
87
88  for i in range(10):
89      simulation(3600, 10) # 设置总时间和打印机每分钟打印多少页
90
91  """
92  Average Wait  20.05 secs   0 tasks remaining.
93  Average Wait  20.12 secs   0 tasks remaining.
94  Average Wait  28.32 secs   0 tasks remaining.
95  Average Wait   7.65 secs   0 tasks remaining.
96  Average Wait  13.17 secs   1 tasks remaining.
97  Average Wait  45.97 secs   0 tasks remaining.
98  Average Wait  14.94 secs   0 tasks remaining.
99  Average Wait   1.81 secs   0 tasks remaining.
100 Average Wait   0.00 secs   0 tasks remaining.
101 Average Wait   6.71 secs   0 tasks remaining.
102 """
```

每一个任务都需要保存一个时间戳，用于计算等待时间。这个时间戳代表任务被创建并放入打印任务队列的时间。 waitTime 方法可以获得任务在队列中等待的时间。

主模拟程序simulation实现了之前描述的算法。 printQueue 对象是队列抽象数据类型的实例。布尔辅助函数newPrintTask判断是否有新创建的打印任务。我们再一次使用random模块中的 randrange 函数来生成随机数，不过这一次的取值范围是 1~180。平均每 180 秒有一个打印任务。通过从随机数中选取 180，可以模拟这个随机事件。

每次模拟的结果不一定相同。对此，我们不需要在意。这是由于随机数的本质导致的。我们感兴趣的是当参数改变时结果出现的趋势。

首先，模拟 60 分钟（ 3600 秒）内打印速度为每分钟 5 页。并且，我们进行 10 次这样的模拟。由于模拟中使用了随机数，因此每次返回的结果都不同。
在模拟 10 次之后，可以看到平均等待时间是 122.092 秒，并且等待时间的差异较大，从最短的 17.27 秒到最长的 376.05 秒。此外，只有 2 次在给定时间内完成了所有任务。
现在把打印速度改成每分钟 10 页，然后再模拟 10 次。由于加快了打印速度，因此我们希望一小时内能完成更多打印任务。

3. 讨论
   在之前的内容中，我们试图解答这样一个问题：如果提高打印质量并降低打印速度，打印机能否及时完成所有任务？我们编写了一个程序来模拟随机提交的打印任务，待打印的页数也是随机的。

上面的输出结果显示，按每分钟5页的打印速度，任务的等待时间在17.27秒和376.05秒之间，相差约6分钟。提高打印速度之后，等待时间在1.29秒和28.96秒之间。此外，在每分钟5页的速度下，10次模拟中有8次没有按时完成所有任务。

可见，降低打印速度以提高打印质量，并不是明智的做法。学生不能等待太长时间，当他们要赶去上课时尤其如此。6分钟的等待时间实在是太长了。

这种模拟分析能帮助我们回答很多"如果"问题。只需改变参数，就可以模拟感兴趣的任意行为。以下是几个例子。
❏ 如果实验室里的学生增加到20个，会怎么样?
❏ 如果是周六，学生不需要上课，他们是否愿意等待?
❏ 如果每个任务的页数变少了，会怎么样?

这些问题都能通过修改本例中的模拟程序来解答。但是，模拟的准确度取决于它所基于的假设和参数。真实的打印任务数量和学生数目是准确构建模拟程序必不可缺的数据。

8. 双端队列

与栈和队列不同的是，双端队列的限制很少。双端队列是与队列类似的有序集合。它有一前、一后两端，元素在其中保持自己的位置。与队列不同的是，双端队列对在哪一端添加和移除元素没有任何限制。新元素既可以被添加到前端，也可以被添加到后端。同理，已有的元素也能从任意一端移除。

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- `Deque()` creates a new deque that is empty. It needs no parameters and returns an empty deque.
- `addFront(item)` adds a new item to the front of the deque. It needs the item and returns nothing.
- `addRear(item)` adds a new item to the rear of the deque. It needs the item and returns nothing.
- `removeFront()` removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- `removeRear()` removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- `isEmpty()` tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- `size()` returns the number of items in the deque. It needs no parameters and returns an integer.

As an example, if we assume that `d` is a deque that has been created and is currently empty, then Table {dequeoperations} shows the results of a sequence of deque operations. Note that the contents in front are listed on the right. It is very important to keep track of the front and the rear as you move items in and out of the collection as things can get a bit confusing.

Deque Operation	Deque Contents	Return Value
`d.isEmpty()`	`[]`	`True`
`d.addRear(4)`	`[4]`	
`d.addRear('dog')`	`['dog',4,]`	
`d.addFront('cat')`	`['dog',4,'cat']`	
`d.addFront(True)`	`['dog',4,'cat',True]`	
`d.size()`	`['dog',4,'cat',True]`	`4`
`d.isEmpty()`	`['dog',4,'cat',True]`	`False`
`d.addRear(8.4)`	`[8.4,'dog',4,'cat',True]`	
`d.removeRear()`	`['dog',4,'cat',True]`	`8.4`
`d.removeFront()`	`['dog',4,'cat']`	`True`

双端队列实现

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

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

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

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

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

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

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


d = Deque()
print(d.isEmpty())
d.addRear(4)
```

```
d.addRear('dog')
d.addFront('cat')
d.addFront(True)
print(d.size())
print(d.isEmpty())
d.addRear(8.4)
print(d.removeRear())
print(d.removeFront())
"""
True
4
False
8.4
True
"""
```

在双端队列的Python实现中，在前端进行的添加操作和移除操作的时间复杂度是O(1)，在后端的则是O( )n。

练习05902: 双端队列

http://cs101.openjudge.cn/practice/05902/

定义一个双端队列，进队操作与普通队列一样，从队尾进入。出队操作既可以从队头，也可以从队尾。编程实现这个数据结构。

输入
第一行输入一个整数t，代表测试数据的组数。
每组数据的第一行输入一个整数n，表示操作的次数。
接着输入n行，每行对应一个操作，首先输入一个整数type。
当type=1，进队操作，接着输入一个整数x，表示进入队列的元素。
当type=2，出队操作，接着输入一个整数c，c=0代表从队头出队，c=1代表从队尾出队。
n <= 1000

输出
对于每组测试数据，输出执行完所有的操作后队列中剩余的元素,元素之间用空格隔开，按队头到队尾的顺序输出，占一行。如果队列中已经没有任何的元素，输出NULL。

样例输入

```
2
5
1 2
1 3
1 4
2 0
2 1
6
1 1
1 2
1 3
```

```
2 0
2 1
2 0
```

样例输出

```
3
NULL
```

```
from collections import deque

for _ in range(int(input())):
    n=int(input())
    q=deque([])
    for i in range(n):
        a,b=map(int,input().split())
        if a==1:
            q.append(b)
        else:
            if b==0:
                q.popleft()
            else:
                q.pop()
    if q:
        print(*q)
    else:
        print('NULL')
```

练习04067: 回文数字（Palindrome Number）

http://cs101.openjudge.cn/practice/04067/

给出一系列非负整数，判断是否是一个回文数。回文数指的是正着写和倒着写相等的数。

输入

若干行，每行是一个非负整数（不超过99999999）

输出

对每行输入，如果其是一个回文数，输出YES。否则输出NO。

样例输入

```
11
123
0
14277241
67945497
```

样例输出

```
YES
NO
YES
YES
NO
```

Use the deque from the collections module. The is_palindrome function checks if a number is a palindrome by converting it to a string, storing it in a deque, and then comparing the first and last elements until the deque is empty or only contains one element.

```python
from collections import deque

def is_palindrome(num):
    num_str = str(num)
    num_deque = deque(num_str)
    while len(num_deque) > 1:
        if num_deque.popleft() != num_deque.pop():
            return "NO"
    return "YES"

while True:
    try:
        num = int(input())
        print(is_palindrome(num))
    except EOFError:
        break
```

练习04099: 队列和栈

http://cs101.openjudge.cn/practice/04099/

```python
from collections import deque
for _ in range(int(input())):
    queue = deque()
    stack = deque()
    stop = False
```

```
    for _ in range(int(input())):
        s = input()
        if s=='pop':
            try:
                queue.popleft()
                stack.pop()
            except IndexError:
                stop = True
        else:
            a = int(s.split()[1])
            queue.append(a)
            stack.append(a)
    if not stop:
        print(' '.join(list(map(str,queue))))
        print(' '.join(list(map(str,stack))))
    elif stop:
        print('error')
        print('error')
```

附录

cs101计概（计算机基础1/2）每日选做

https://github.com/GMyhf/2024fall-cs101/blob/main/problem_list_at_2024fall.md

Python数据结构与算法分析（第3版），https://runestone.academy/ns/books/published/pythonds3/index.html

算法导论 第三版（Thmos.H.Cormen ,Charles E. Leiserson etc.)