版权信息

书名：算法图解

作者：[美] Aditya Bhargava

译者：袁国忠

ISBN：978-7-115-44763-0

版权声明

谨以此书献给我的父母、**Sangeeta**和**Yogesh**

前言

我因为爱好而踏入了编程殿堂。*Visual Basic 6 for Dummies*教会了我基础知识，接着我不断阅读，学到的知识也越来越多，但对算法却始终没搞明白。至今我还记得购买第一本算法书后的情景：我琢磨着目录，心想终于要把这些主题搞明白了。但那本书深奥难懂，看了几周后我就放

弃了。直到遇到一位优秀的算法教授后，我才认识到这些概念是多么地简单而优雅。

几年前，我撰写了第一篇图解式博文。我是视觉型学习者，对图解式写作风格钟爱有加。从那时候起，我撰写了多篇介绍函数式编程、Git、机器学习和并发的图解式博文。顺便说一句，刚开始我的写作水平很一般。诠释技术概念很难，设计出好的示例需要时间，阐释难以理解的概念也需要时间，因此很容易对难讲的内容一带而过。我本以为自己已经做得相当好了，直到有一篇博文大受欢迎，有位同事却跑过来跟我 说：“我读了你的博文，但还是没搞懂。”看来在写作方面我要学习的还 有很多。

在撰写这些博文期间，Manning出版社找到我，问我想不想编写一本图解式图书。事实证明，Manning出版社的编辑对如何诠释技术概念很在行，他们教会了我如何做。我编写本书的目的就是要把难懂的技术主题说清楚，让这本算法书易于理解。与撰写第一篇博文时相比，我的写作水平有了长足进步，但愿你也认为本书内容丰富、易于理解。

致谢

感谢Manning出版社给我编写本书的机会，并给予我极大的创作空间。感谢出版人Marjan Bace，感谢Mike Stephens领我入门，感谢Bert Bates 教我如何写作，感谢Jennifer Stout的快速回复以及大有帮助的编辑工 作。感谢Manning出版社的制作人员，他们是Kevin Sullivan、Mary Piergies、Tiffany Taylor、Leslie Haimes以及其他幕后人员。另外，还要 感谢阅读手稿并提出建议的众人，他们是Karen Bensdon、Rob Green、 Michael Hamrah、Ozren Harlovic、Colin Hastie、

Christopher Haupt、 Chuck Henderson、Pawel Kozlowski、Amit Lamba、Jean-Francois Morin、Robert Morrison、Sankar Ramanathan、Sander Rossel、Doug Sparling和Damien White。

感谢一路上向我伸出援手的人：Flaskhit游戏专区的各位教会了我如何编写代码；很多朋友帮助审阅手稿、提出建议并让我尝试不同的诠释方式，其中包括Ben Vinegar、Karl Puzon、Alex Manning、Esther Chan 、 Anish Bhatt、Michael Glass、Nikrad Mahdi、Charles Lee、Jared Friedman、Hema Manickavasagam、Hari Raja、Murali Gudipati、Srinivas Varadan等；Gerry Brady教会了我算法。还要深深地感谢算法方面的学 1
者，如CLRS 、高德纳和Strang。我真的是站在了巨人的肩上。

1 《算法导论》四位作者的姓氏（Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest和Clifford Stein）首字母缩写。——译者注

感谢爸爸、妈妈、Priyanka和其他家庭成员，感谢你们一贯的支持。深深感谢妻子Maggie，我们的面前还有很多艰难险阻，有些可不像周五晚上待在家里修改手稿那么简单。

最后，感谢所有试读本书的读者，还有在论坛上提供反馈的读者，你们让本书的质量更上了一层楼。

关于本书

本书易于理解，没有大跨度的思维跳跃，每次引入新概念时，都立即进行诠释，或者指出将在什么地方进行诠释。核心概念都通过练习和反复诠释进行强化，以便你检验假设，跟上步伐。

书中使用示例来帮助理解。我的目标是让你轻松地理解这些概念，而不是让正文充斥各种符号。我还认为，如果能够回忆起熟悉的情形，学习效果将达到最佳，而示例有助于唤醒记忆。因此，如果你要记住数组和链表（第2章）之间的差别，只要想想在电影院找座位就坐的情形。另

外，不怕你说我啰嗦，我是视觉型学习者，因此本书包含大量的图示。

本书内容是精挑细选的。没必要在一本书中介绍所有的排序算法，不然还要维基百科和可汗学院做什么。书中介绍的所有算法都非常实用，对我从事的软件工程师的工作大有帮助，还可为阅读更复杂的主题打下坚实的基础。祝你阅读愉快！

路线图

本书前三章将帮助你打好基础。

> 第1章：你将学习第一种实用算法——二分查找；还将学习使用大O表示法分析算法的速度。本书从始至终都将使用大O表示法来分析算法的速度。
>
> 第2章：你将学习两种基本的数据结构——数组和链表。这两种数据结构贯穿本书，它们还被用来创建更高级的数据结构，如第5章介绍的散列表。
>
> 第3章：你将学习递归，一种被众多算法（如第4章介绍的快速排序）采用的实用技巧。

根据我的经验，大O表示法和递归对初学者来说颇具挑战性，因此介绍这些内容时我放慢了脚步，花费的篇幅也较长。

余下的篇幅将介绍应用广泛的算法。

问题解决技巧：将在第4、8和9章介绍。遇到问题时，如果不确定 该如何高效地解决，可尝试分而治之（第4章）或动态规划（第9 章）；如果认识到根本就没有高效的解决方案，可转而使用贪婪算 法（第8章）来得到近似答案。

散列表：将在第5章介绍。散列表是一种很有用的数据结构，由键 值对组成，如人名和电子邮件地址或者用户名和密码。散列表的用 途之大，再怎么强调都不过分。每当我需要解决问题时，首先想到 的两种方法是：可以使用散列表吗？可以使用图来建立模型吗？

图算法：将在第6、7章介绍。图是一种模拟网络的方法，这种网络 包括人际关系网、公路网、神经元网络或者任何一组连接。广度优 先搜

索（第6章）和狄克斯特拉算法（第7章）计算网络中两点之间 的最短距离，可用来计算两人之间的分隔度或前往目的地的最短路 径。

K最近邻算法（KNN）：将在第10章介绍。这是一种简单的机器学 习算法，可用于创建推荐系统、OCR引擎、预测股价或其他值 （如“我们认为Adit会给这部电影打4星”）的系统，以及对物件进行 分类（如“这个字母是Q”）。

接下来如何做：第11章概述了适合你进一步学习的10种算法。 如何

阅读本书

本书的内容和排列顺序都经过了细心编排。如果你对某个主题感兴趣，直接跳到那里阅读即可；否则就按顺序逐章阅读吧，因为它们都以之前介绍的内容为基础。

强烈建议你动手执行示例代码，这部分的重要性再怎么强调都不过分。可以原封不动地输入代码，也可从www.manning.com/books/grokking algorithms或https://github.com/egonschiele/grokking_algorithms下载，再 执行它们。这样，你记住的内容将多得多。

另外，建议你完成书中的练习。这些练习都很短，通常只需一两分钟就

能完成，但有些可能需要5~10分钟。这些练习有助于检查你的思路，以免偏离正道太远。

读者对象

本书适合任何具备编程基础并想理解算法的人阅读。你可能面临一个编程问题，需要找一种算法来实现解决方案，抑或你想知道哪些算法比较有用。下面列出了可能从本书获得很多帮助的部分读者。

业余程序员

编程培训班学员

需要重温算法的计算机专业毕业生

对编程感兴趣的物理或数学等专业毕业生

代码约定和下载

本书所有的示例代码都是使用Python 2.7编写的。书中在列出代码时使 用了等宽字体。有些代码还进行了标注，旨在突出重要的概念。

本书的示例代码可从出版社网站下载，也可从https://github.com/egonschiele/grokking_algorithms下载。

我认为，如果能享受学习过程，就能获得最好的学习效果。请尽情地享受学习过程，动手运行示例代码吧！

作者在线

购买英文版的读者可免费访问Manning出版社管理的专用网络论坛，并可以评论本书、提出技术性问题以及获得作者和其他读者的帮助。若要访问并订阅该论坛，可在浏览器的地址栏中输入www.manning.com/books/grokking-algorithms。这个网页会告诉你注册 后如何进入论坛、可获得哪些帮助以及讨论时应遵守的规则。

Manning出版社致力于为读者和作者提供能够深入交流的场所。然而，作者参与论坛讨论纯属自愿，没有任何报酬，因此Manning出版社对其参与讨论的程度不做任何承诺。建议你向作者提些有挑战性的问题，以免他失去参与讨论的兴趣！只要本书还在销售，你就能通过出版社的网站访问作者在线论坛以及存档的讨论内容。

第 1 章　算法简介

本章内容

为阅读后续内容打下基础。

编写第一种查找算法——二分查找。

学习如何谈论算法的运行时间——大O表示法。

了解一种常用的算法设计方法——递归。

1.1 引言

算法是一组完成任务的指令。任何代码片段都可视为算法，但本书只介绍比较有趣的部分。本书介绍的算法要么速度快，要么能解决有趣的问题，要么兼而有之。下面是书中一些重要内容。

第1章讨论二分查找，并演示算法如何能够提高代码的速度。在一个示例中，算法将需要执行的步骤从40亿个减少到了32个！

GPS设备使用图算法来计算前往目的地的最短路径，这将在第6、7 和8章介绍。

你可使用动态规划来编写下国际跳棋的AI算法，这将在第9章讨论。

对于每种算法，本书都将首先进行描述并提供示例，再使用大O表示法讨论其运行时间，最后探索它可以解决的其他问题。

1.1.1 性能方面

好消息是，本书介绍的每种算法都很可能有使用你喜欢的语言编写的实现，因此你无需自己动手编写每种算法的代码！但如果你不明白其优缺点，这些实现将毫无用处。在本书中，你将学习比较不同算法的优缺

点：该使用合并排序算法还是快速排序算法，或者该使用数组还是链表。仅仅改用不同的数据结构就可能让结果大不相同。

1.1.2 问题解决技巧

你将学习至今都没有掌握的问题解决技巧，例如：

如果你喜欢开发电子游戏，可使用图算法编写跟踪用户的AI系统；

你将学习使用K最近邻算法编写推荐系统；

有些问题在有限的时间内是不可解的！书中讨论NP完全问题的部分将告诉你，如何识别这样的问题以及如何设计找到近似答案的算法。

总而言之，读完本书后，你将熟悉一些使用最为广泛的算法。利用这些新学到的知识，你可学习更具体的AI算法、数据库算法等，还可在工作中迎接更严峻的挑战。

需要具备的知识

要阅读本书，需要具备基本的代数知识。具体地说，给定函数$f(x) = x \times 2$，$f(5)$的值是多少呢？如果你的答案为10，那就够了。

另外，如果你熟悉一门编程语言，本章（以及本书）将更容易理解。本书的示例都是使用Python编写的。如果你不懂任何编程语言 但想学习一门，请选择Python，它非常适合初学者；如果你熟悉其 他语言，如Ruby，对阅读本书也大有帮助。

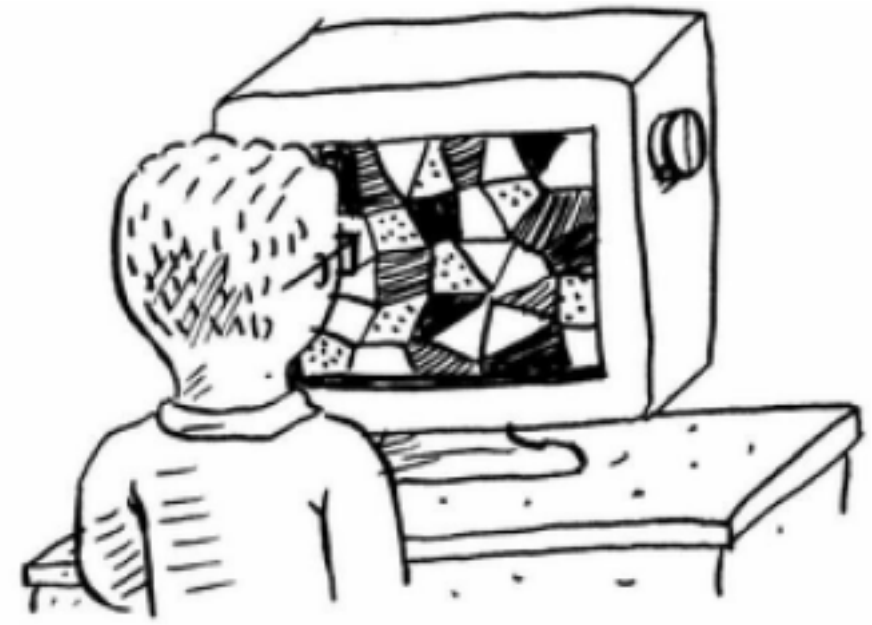
1.2 二分查找

假设要在电话簿中找一个名字以K打头的人，（现在谁还用电话簿！）可以从头开始翻页，直到进入以K打头的部分。但你很可能不这样做，而是从中间开始，因为你知道以K打头的名字在电话簿中间。

又假设要在字典中找一个以O打头的单词，你也将从中间附近开始。

现在假设你登录Facebook。当你这样做时，Facebook必须核实你是否有 其网站的账户，因此必须在其数据库中查找你的用户名。如果你的用户 名为karlmageddon，Facebook可从以A打头的部分开始查找，但更合乎 逻辑的做法是从中间开始查找。

这是一个查找问题，在前述所有情况下，都可使用同一种算法来解决问题，这种算法就是二分查找。

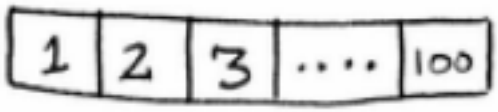


二分查找是一种算法，其输入是一个有序的元素列表（必须有序的原因稍后解释）。如果要查找的元素包含在列表中，二分查找返回其位置；否则返回null。

下图是一个例子。

使用二分查找在电话簿中查找公司

下面的示例说明了二分查找的工作原理。我随便想一个1～100的数字。



你的目标是以最少的次数猜到这个数字。你每次猜测后，我会说小了、大了或对了。

假设你从1开始依次往上猜，猜测过程会是这样。

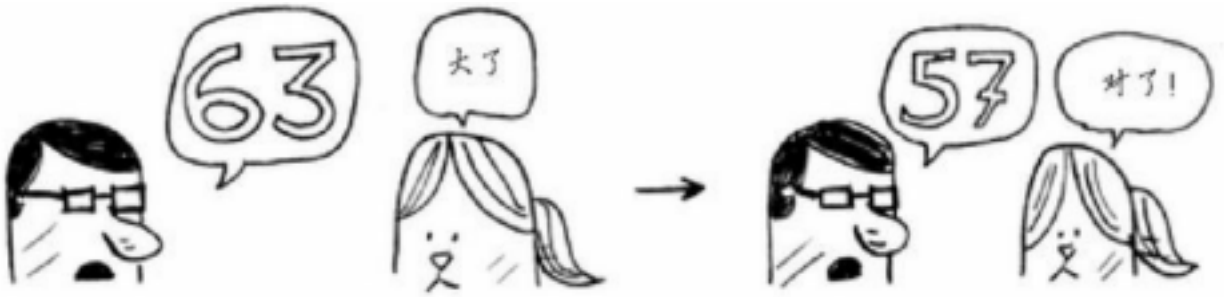
这是简单查找，更准确的说法是傻找。每次猜测都只能排除一个数字。如果我想的数字是99，你得猜99次才能猜到！

1.2.1 更佳的查找方式

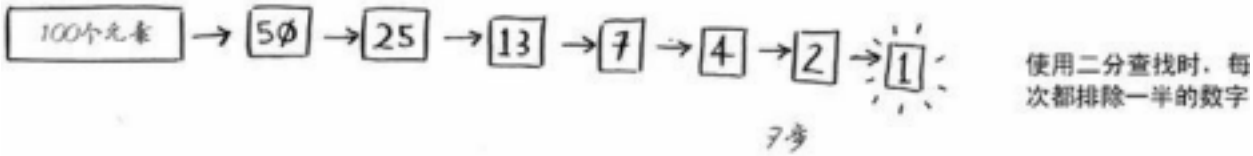
下面是一种更佳的猜法。从50开始。

小了，但排除了一半的数字！至此，你知道1～50都小了。接下来，你猜75。

大了，那余下的数字又排除了一半！使用二分查找时，你猜测的是中间的数字，从而每次都将余下的数字排除一半。接下来，你猜63（50和75 中间的数字）。

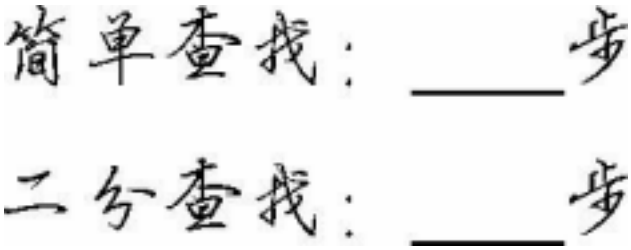


这就是二分查找，你学习了第一种算法！每次猜测排除的数字个数如下。

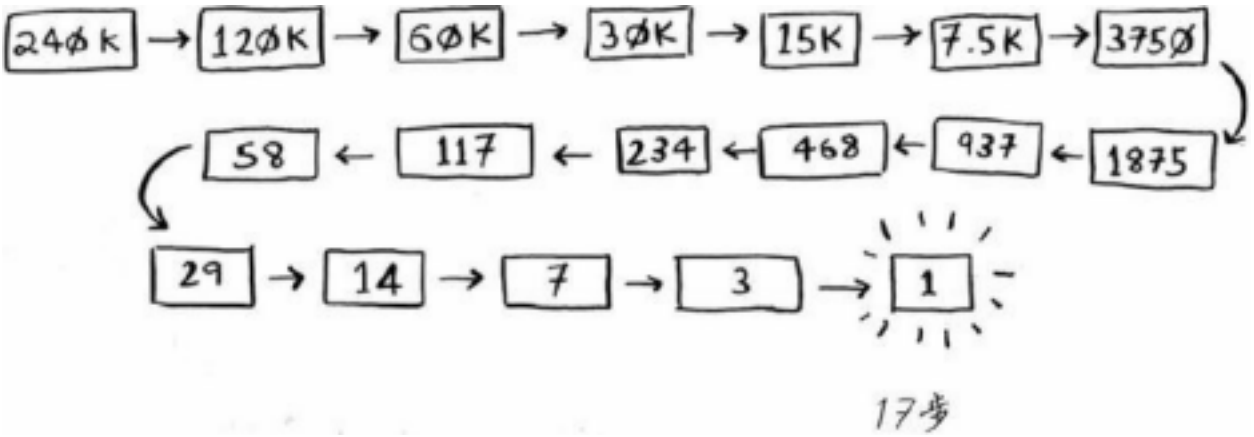


不管我心里想的是哪个数字，你在7次之内都能猜到，因为每次猜测都将排除很多数字！

假设你要在字典中查找一个单词，而该字典包含240 000个单词，你认 为每种查找最多需要多少步?



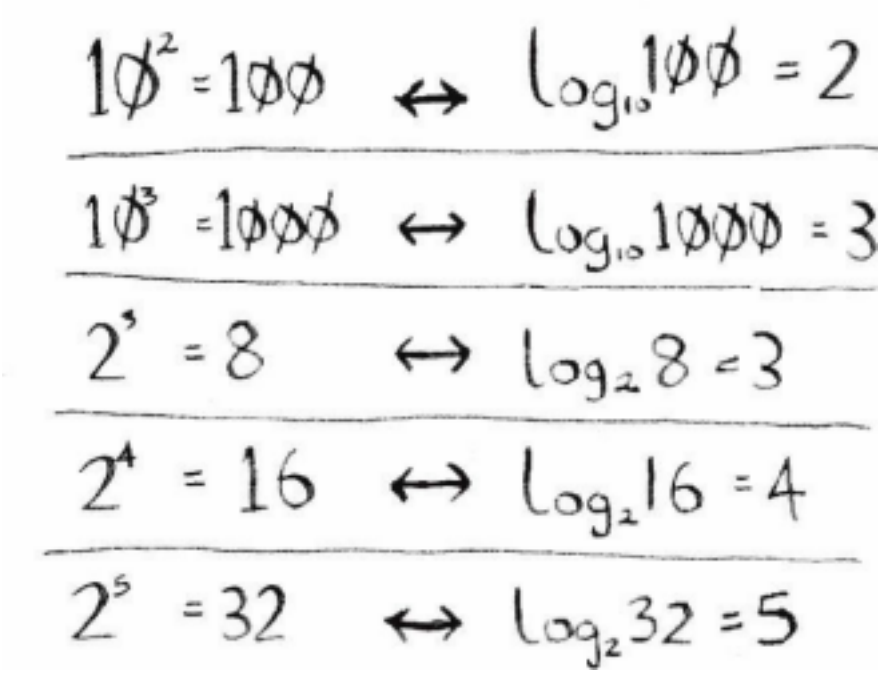
如果要查找的单词位于字典末尾，使用简单查找将需要240 000步。使 用二分查找时，每次排除一半单词，直到最后只剩下一个单词。



因此，使用二分查找只需18步——少多了！一般而言，对于包含n个元素的列表，用二分查找最多需要$\log_2 n$步，而简单查找最多需要n步。

对数

你可能不记得什么是对数了，但很可能记得什么是幂。$\log_{10}100$ 相 当于问“将多少个10相乘的结果为100”。答案是两个：10 × 10 = 100。因此，$\log_{10}100 = 2$。对数运算是幂运算的逆运算。



对数是幂运算的逆运算

本书使用大O表示法（稍后介绍）讨论运行时间时，log指的都是$\log_2$。使用简单查找法查找元素时，在最糟情况下需要查看每个元素。因此，如果列表包含8个数字，你最多需要检查8个数字。而使用二分查找时，最多需要检查log *n*个元素。如果列表包含8个元 素，你最多需要检查3个元素，因为log 8 = 3（$2^3 = 8$）。如果列表包含1024个元素，你最多需要检查10个元素，因为log 1024 = 10（$2^{10} = 1024$）。

说明

本书经常会谈到log时间，因此你必须明白对数的概念。如果你不明白，可汗学院（khanacademy.org）有一个不错的视频，把这个概 念讲得很清楚。

说明

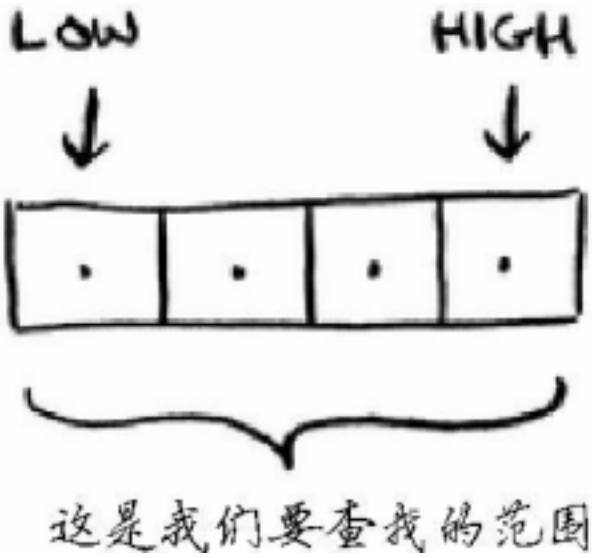
仅当列表是有序的时候，二分查找才管用。例如，电话簿中的名字是按字母顺序排列的，因此可以使用二分查找来查找名字。如果名

字不是按顺序排列的，结果将如何呢?

下面来看看如何编写执行二分查找的Python代码。这里的代码示例使用了数组。如果你不熟悉数组，也不用担心，下一章就会介绍。你只需知道，可将一系列元素存储在一系列相邻的桶（bucket），即数组中。这些桶从0开始编号：第一个桶的位置为#0，第二个桶为#1，第三个桶为#2，以此类推。

函数binary_search接受一个有序数组和一个元素。如果指定的元素包含在数组中，这个函数将返回其位置。你将跟踪要在其中查找的数组部分——开始时为整个数组。

```
low = 0
high = len(list) - 1
```

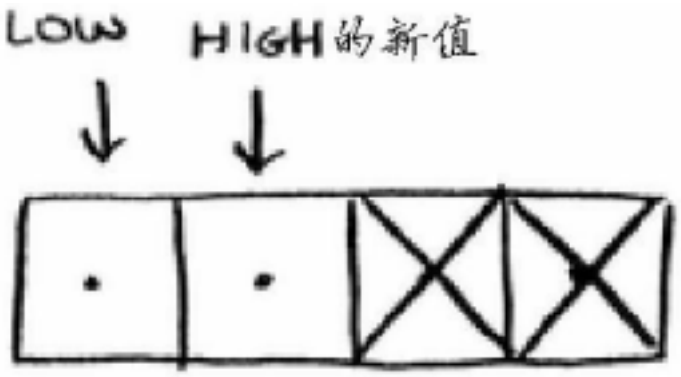


你每次都检查中间的元素。

```
mid = (low + high) / 2 ←---如果(low + high)不是偶数，Python自动将mid向下取整。guess = list[mid]
```

如果猜的数字小了，就相应地修改low。

```
if guess < item:
 low = mid + 1
```



如果猜的数字大了，就修改high。完整的代码如下。

```
def binary_search(list, item):
 low = 0  (以下2行）low和high用于跟踪要在其中查找的列表部分
high = len(list)—1

 while low <= high: ←--------------只要范围没有缩小到只包含一个元素
,  mid = (low + high) / 2 ←--------------就检查中间的元素 guess =
list[mid]
 if guess == item: ←--------------找到了元素
return mid
 if guess > item: ←--------------猜的数字大了
high = mid - 1
 else: ←----------------------------猜的数字小了
low = mid + 1
 return None ←---------------------没有指定的元素

my_list = [1, 3, 5, 7, 9] ←------------来测试一下!

print binary_search(my_list, 3) # => 1 ←---------------------别忘了索引从
0print binary_search(my_list, -1) # => None ←---------------------在
Python中,
```

开始，第二个位
None表示空，

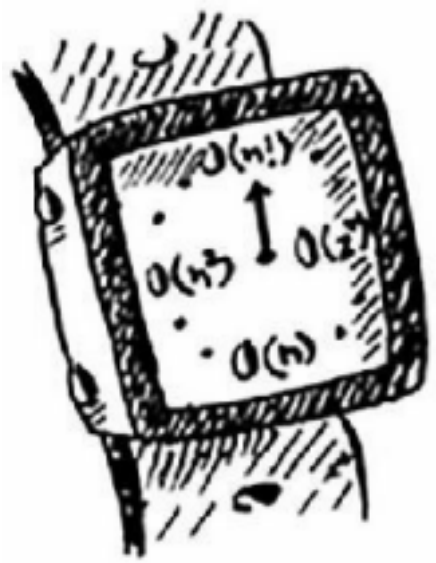
练习

1.1 假设有一个包含128个名字的有序列表，你要使用二分查找在其中 查找一个名字，请 问最多需要几步才能找到?

1.2 上面列表的长度翻倍后，最多需要几步?

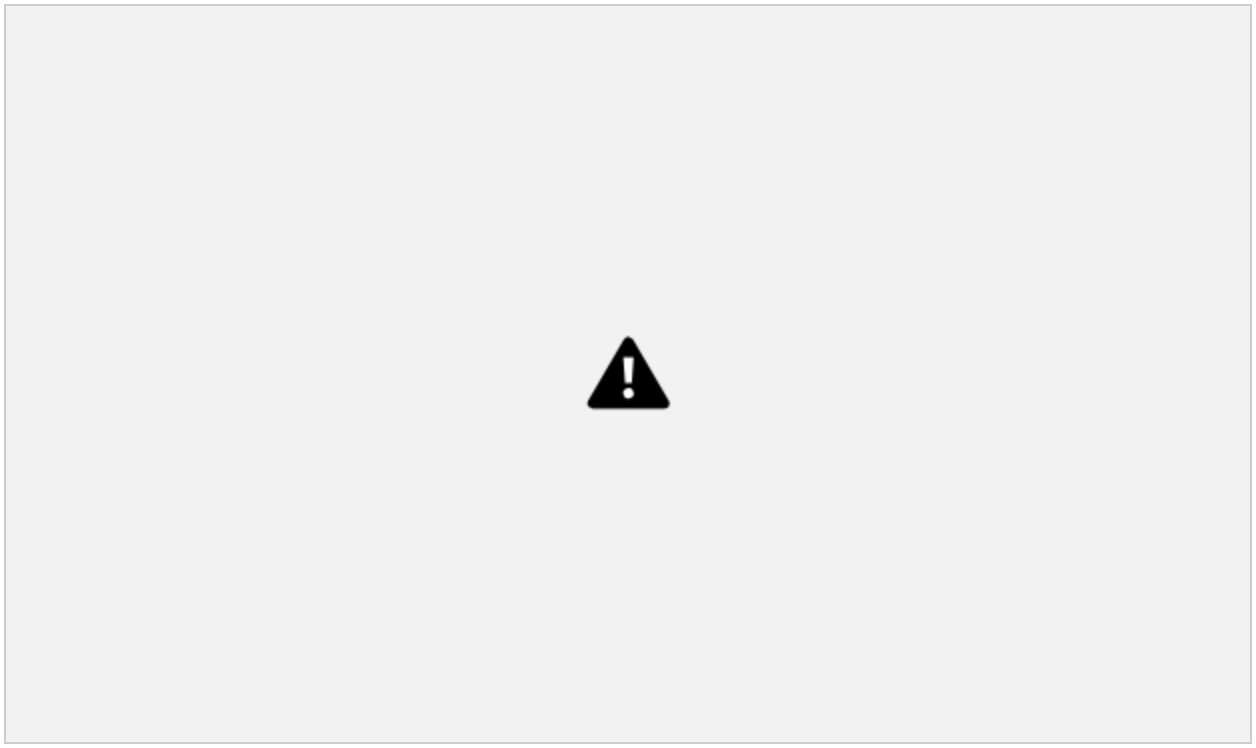
1.2.2 运行时间

每次介绍算法时，我都将讨论其运行时间。一般而言，应选择效率最高的算法，以最大限度地减少运行时间或占用空间。



回到前面的二分查找。使用它可节省多少时间呢？简单查找逐个地检查数字，如果列表包含100个数字，最多需要猜100次。如果列表包含40亿　个数字，最多需要猜40亿次。换言之，最多需要猜测的次数与列表长度 相同，这被称为线性时间（linear time）。

二分查找则不同。如果列表包含100个元素，最多要猜7次；如果列表包含40亿个数字，最多需猜32次。厉害吧？二分查找的运行时间为对数时间（或log时间）。下表总结了我们发现的情况。

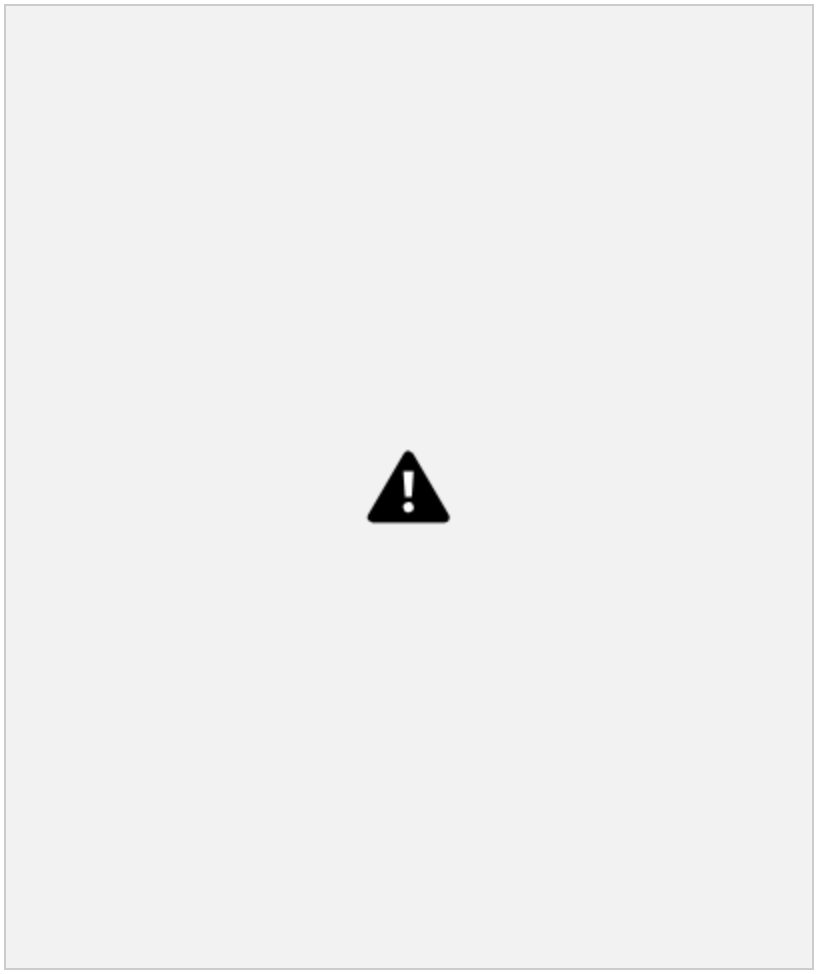


1.3 大O表示法

大**O**表示法是一种特殊的表示法，指出了算法的速度有多快。谁在乎呢？实际上，你经常要使用别人编写的算法，在这种情况下，知道这些算法的速度大有裨益。本节将介绍大O表示法是什么，并使用它列出一些最常见的算法运行时间。

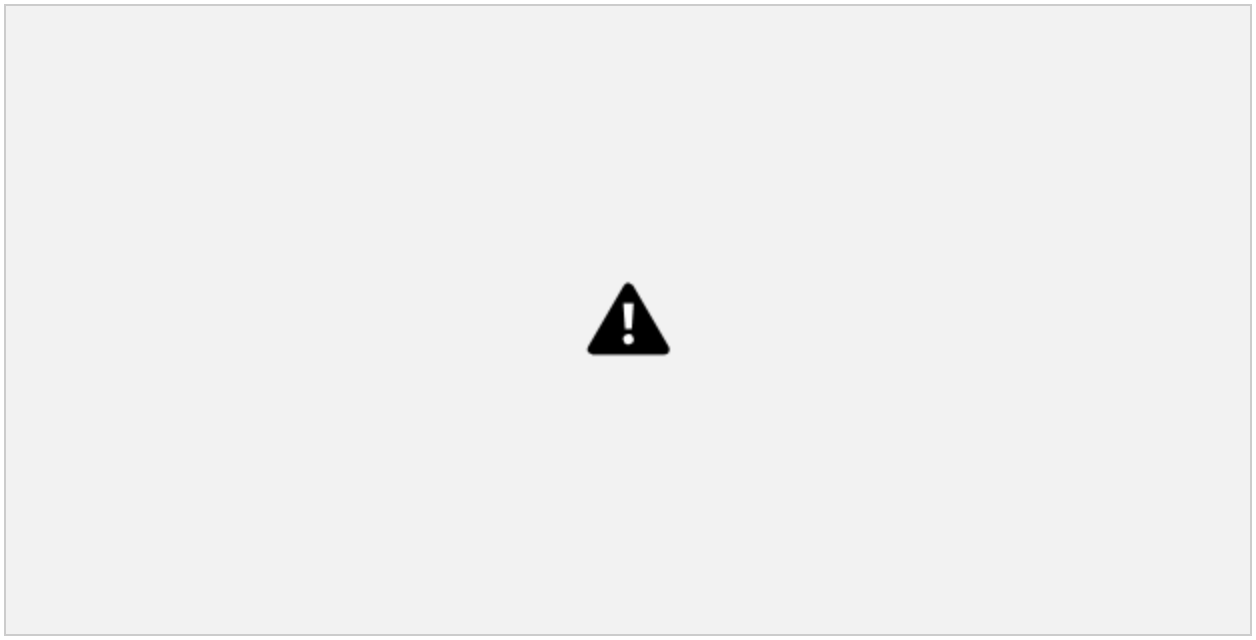
1.3.1 算法的运行时间以不同的速度增加

Bob要为NASA编写一个查找算法，这个算法在火箭即将登陆月球前开始执行，帮助计算着陆地点。



这个示例表明，两种算法的运行时间呈现不同的增速。Bob需要做出决定，是使用简单查找还是二分查找。使用的算法必须快速而准确。一方面，二分查找的速度更快。Bob必须在10秒钟内找出着陆地点，否则火箭将偏离方向。另一方面，简单查找算法编写起来更容易，因此出现bug的可能性更小。Bob可不希望引导火箭着陆的代码中有bug！为确保万无一失，Bob决定计算两种算法在列表包含100个元素的情况下需要的时间。

假设检查一个元素需要1毫秒。使用简单查找时，Bob必须检查100个元素，因此需要100毫秒才能查找完毕。而使用二分查找时，只需检查7个元素（$\log_2 100$大约为7），因此需要7毫秒就能查找完毕。然而，实际要 查找的列表可能包含10亿个元素，在这种情况下，简单查找需要多长时 间呢？二分查找又需要多长时间呢？请务必找出这两个问题的答案，再 接着往下读。

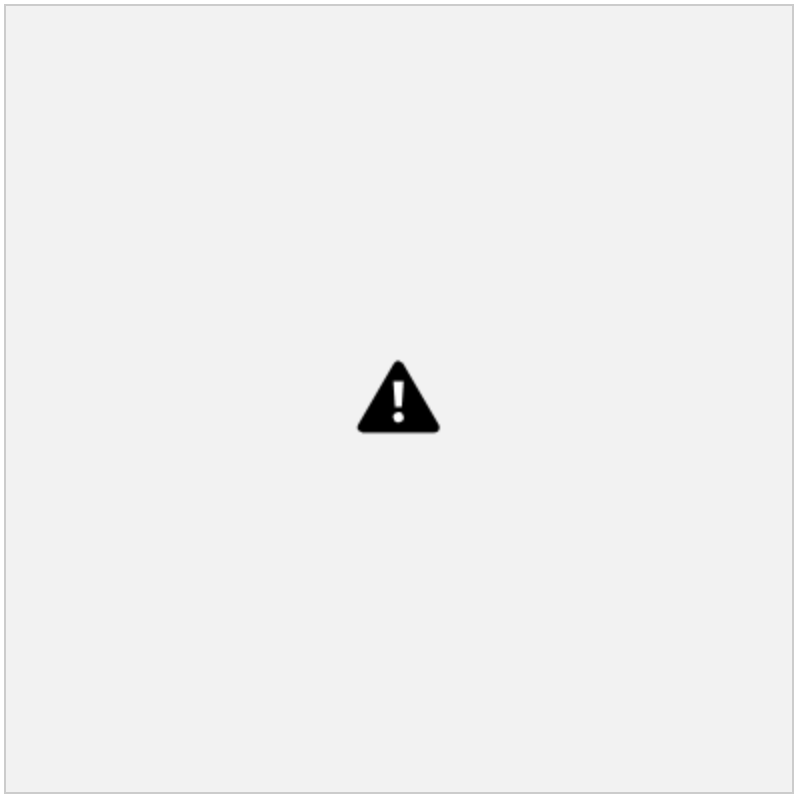


Bob使用包含10亿个元素的列表运行二分查找，运行时间为30毫秒 （$\log_2$1 000 000 000大约为30）。他心里想，二分查找的速度大约为简单查找的15倍，因为列表包含100个元素时，简单查找需要100毫秒，而 二分查找需要7毫秒。因此，列表包含10亿个元素时，简单查找需要30 × 15 = 450毫秒，完全符合在10秒内查找完毕的要求。Bob决定使用简单 查找。这是正确的选择吗?

不是。实际上，Bob错了，而且错得离谱。列表包含10亿个元素时，简单查找需要10亿毫秒，相当于11天！为什么会这样呢？因为二分查找和简单查找的运行时间的增速不同。

也就是说，随着元素数量的增加，二分查找需要的额外时间并不多，而简单查找需要的额外时间却很多。因此，随着列表的增长，二分查找的速度比简单查找快得多。Bob以为二分查找速度为简单查找的15倍，这不对：列表包含10亿个元素时，为3300万倍。有鉴于此，仅知道算法需 要多长时间才能运行完毕还不够，还需知道运行时间如何随列表增长而

增加。这正是大O表示法的用武之地。



大O表示法指出了算法有多快。例如，假设列表包含*n* 个元素。简单查找需要检查每个元素，因此需要执行*n* 次操作。使用大O表示法，这个运行时间为*O*(*n*)。单位秒呢？没有——大O表示法指的并非以秒为单位的速度。大**O**表示法让你能够比较操作数，它指出了算法运行时间的增速。

再来看一个例子。为检查长度为*n* 的列表，二分查找需要执行log *n* 次操作。使用大O表示法，这个运行时间怎么表示呢？*O*(log *n*)。一般而言，大O表示法像下面这样。

这指出了算法需要执行的操作数。之所以称为大O表示法，是因为操作数前有个大O。这听起来像笑话，但事实如此！

下面来看一些例子，看看你能否确定这些算法的运行时间。

1.3.2 理解不同的大O运行时间

下面的示例，你在家里使用纸和笔就能完成。假设你要画一个网格，它包含16个格子。

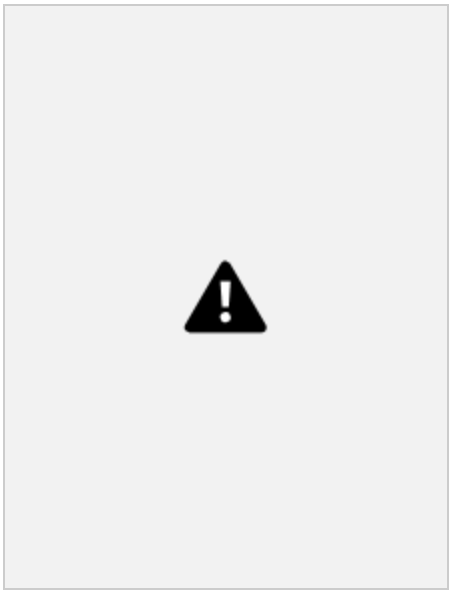
算法**1**

一种方法是以每次画一个的方式画16个格子。记住，大O表示法计算的是操作数。在这个示例中，画一个格子是一次操作，需要画16个格子。如果每次画一个格子，需要执行多少次操作呢?

画16个格子需要16步。这种算法的运行时间是多少?

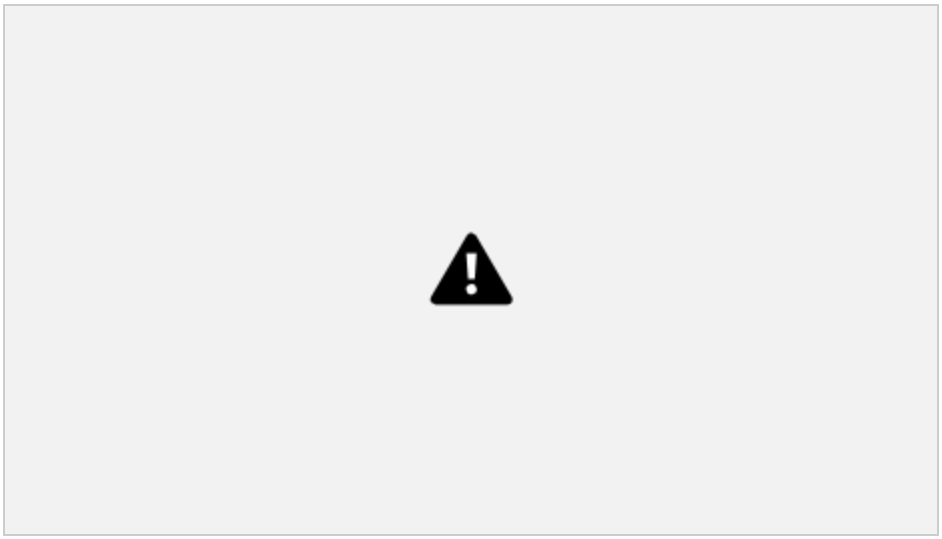
算法**2**

请尝试这种算法——将纸折起来。

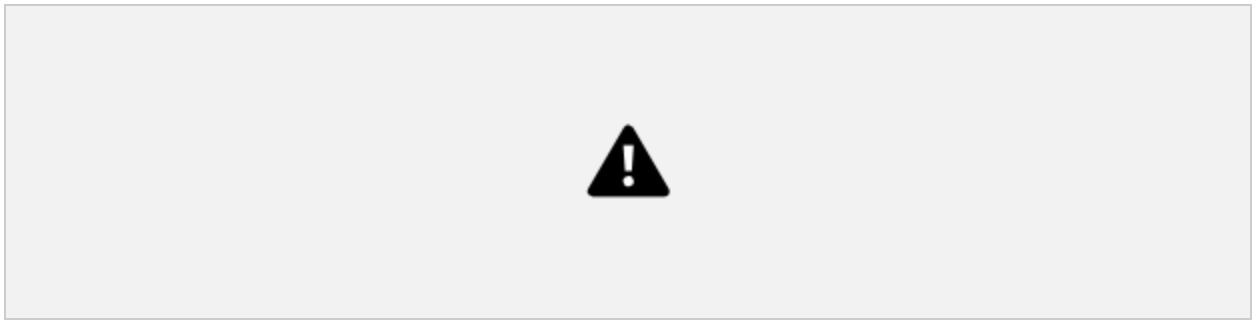


在这个示例中，将纸对折一次就是一次操作。第一次对折相当于画了两个格子！

再折，再折，再折。



折4次后再打开，便得到了漂亮的网格！每折一次，格子数就翻倍，折4 次就能得到16个格子！



你每折一次，绘制出的格子数都翻倍，因此4步就能“绘制”出16个格子。这种算法的运行时间是多少呢？请搞清楚这两种算法的运行时间之后，再接着往下读。

答案如下：算法1的运行时间为$O(n)$，算法2的运行时间为$O(\log n)$

。1.3.3 大O表示法指出了最糟情况下的运行时间

假设你使用简单查找在电话簿中找人。你知道，简单查找的运行时间为$O(n)$，这意味着在最糟情况下，必须查看电话簿中的每个条目。如果要查找的是Adit——电话簿中的第一个人，一次就能找到，无需查看每个条目。考虑到一次就找到了Adit，请问这种算法的运行时间是$O(n)$还是$O(1)$呢?

简单查找的运行时间总是为$O(n)$。查找Adit时，一次就找到了，这是最佳的情形，但大O表示法说的是最糟的情形。因此，你可以说，在最糟情况下，必须查看电话簿中的每个条目，对应的运行时间为$O(n)$。这是一个保证——你知道简单查找的运行时间不可能超过$O(n)$。

> 说明
>
> 除最糟情况下的运行时间外，还应考虑平均情况的运行时间，这很重要。最糟情况和平均情况将在第4章讨论。

1.3.4 一些常见的大O运行时间

下面按从快到慢的顺序列出了你经常会遇到的5种大O运行时间。

$O(\log n)$，也叫对数时间，这样的算法包括二分查找。

$O(n)$，也叫线性时间，这样的算法包括简单查找。

$O(n * \log n)$，这样的算法包括第4章将介绍的快速排序——一种速度较快的排序算法。

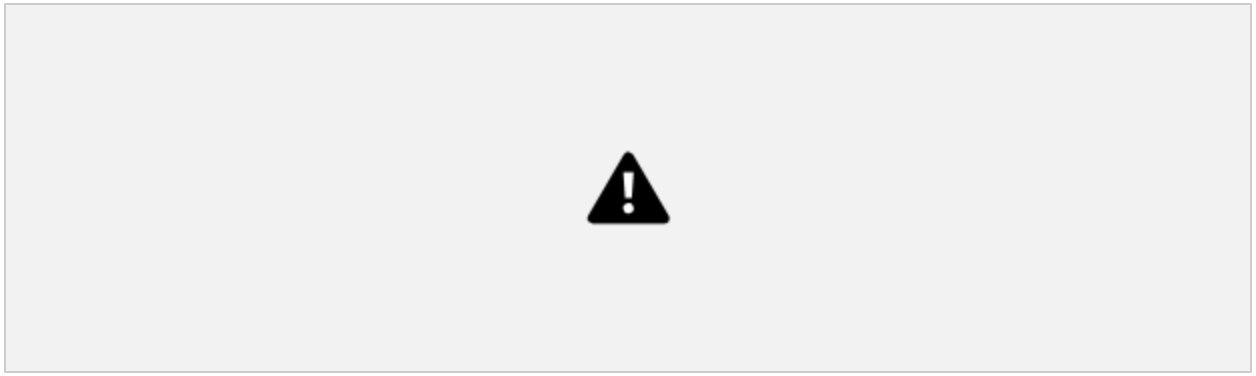
$O(n^2)$，这样的算法包括第2章将介绍的选择排序——一种速度较慢的排序算法。

$O(n!)$，这样的算法包括接下来将介绍的旅行商问题的解决方案——一种非常慢的算法。

假设你要绘制一个包含16格的网格，且有5种不同的算法可供选择，这些算法的运行时间如上所示。如果你选择第一种算法，绘制该网格所需的操作数将为4（log 16 = 4）。假设你每秒可执行10次操作，那么绘制该网格需要0.4秒。如果要绘制一个包含1024格的网格呢？这需要执行10（log 1024 = 10）次操作，换言之，绘制这样的网格需要1秒。这是使 用第一种算法的情况。

第二种算法更慢，其运行时间为$O(n)$。即要绘制16个格子，需要执行16次操作；要绘制1024个格子，需要执行1024次操作。执行这些操作需要 多少秒呢？

下面按从快到慢的顺序列出了使用这些算法绘制网格所需的时间：



还有其他的运行时间，但这5种是最常见的。

这里做了简化，实际上，并不能如此干净利索地将大O运行时间转换为操作数，但就目前而言，这种准确度足够了。等你学习其他一些算法后，第4章将回过头来再次讨论大O表示法。当前，我们获得的主要启

示如下。

算法的速度指的并非时间，而是操作数的增速。

谈论算法的速度时，我们说的是随着输入的增加，其运行时间将以

什么样的速度增加。

算法的运行时间用大O表示法表示。

O(log *n*)比*O*(*n*)快，当需要搜索的元素越多时，前者比后者快得越 多。

练习

使用大O表示法给出下述各种情形的运行时间。

1.3 在电话簿中根据名字查找电话号码。

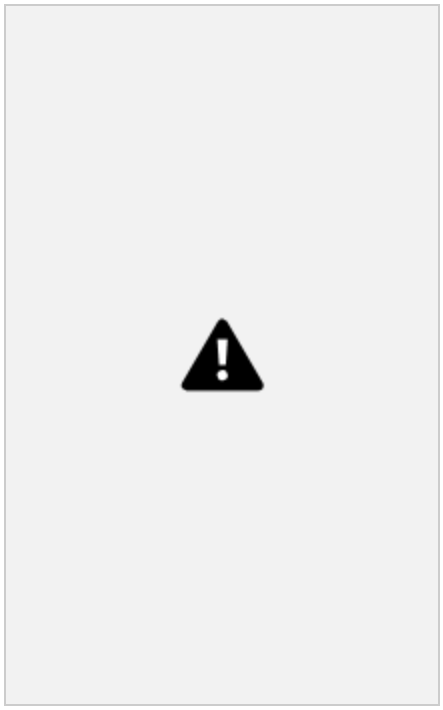
1.4 在电话簿中根据电话号码找人。（提示：你必须查找整个电话簿。）

1.5 阅读电话簿中每个人的电话号码。

1.6 阅读电话簿中姓名以A打头的人的电话号码。这个问题比较棘手，它涉及第4章的概 念。答案可能让你感到惊讶！

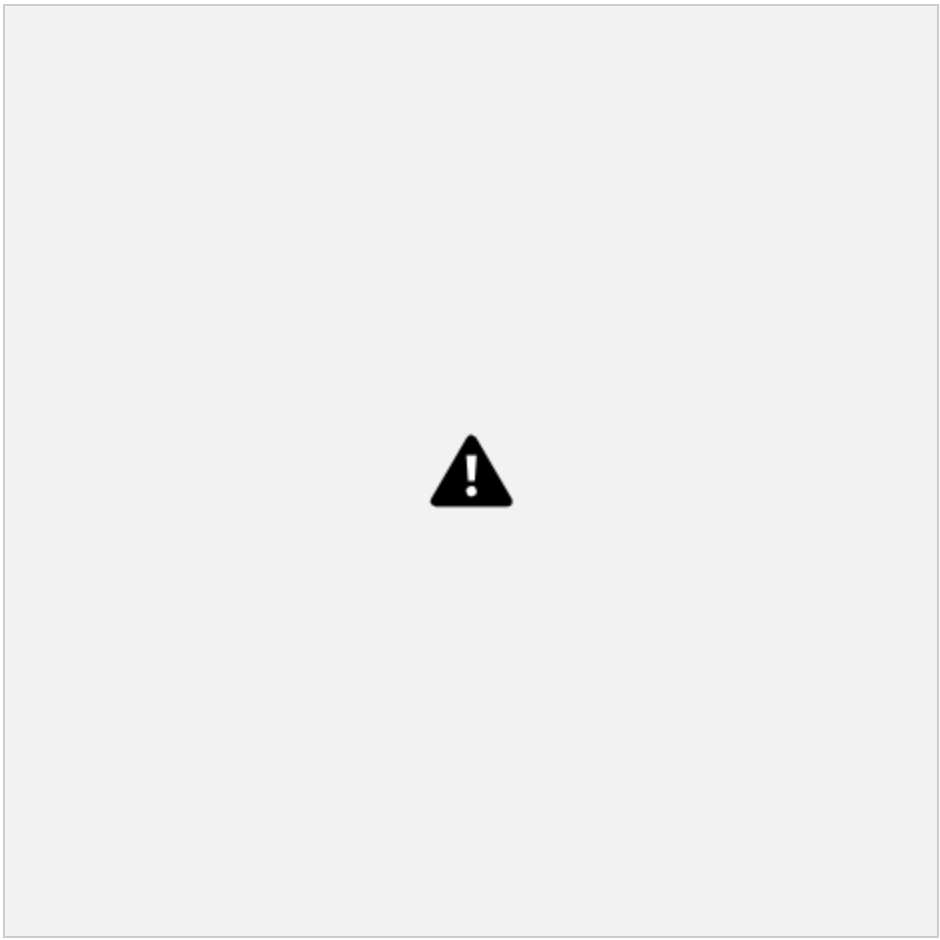
1.3.5 旅行商

阅读前一节时，你可能认为根本就没有运行时间为*O*(*n*!)的算法。让我来证明你错了！ 下面就是一个运行时间极长的算法。这个算法要解决的是计算机科学领域非常著名的旅行商问题，其计算时间增加得非常快，而有些非常聪明的人都认为没有改进空间。

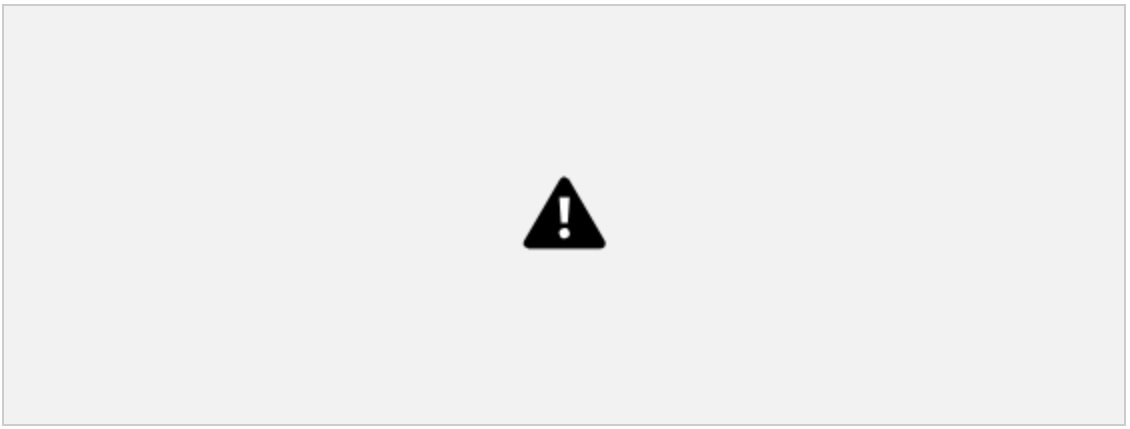


有一位旅行商。

他需要前往5个城市。

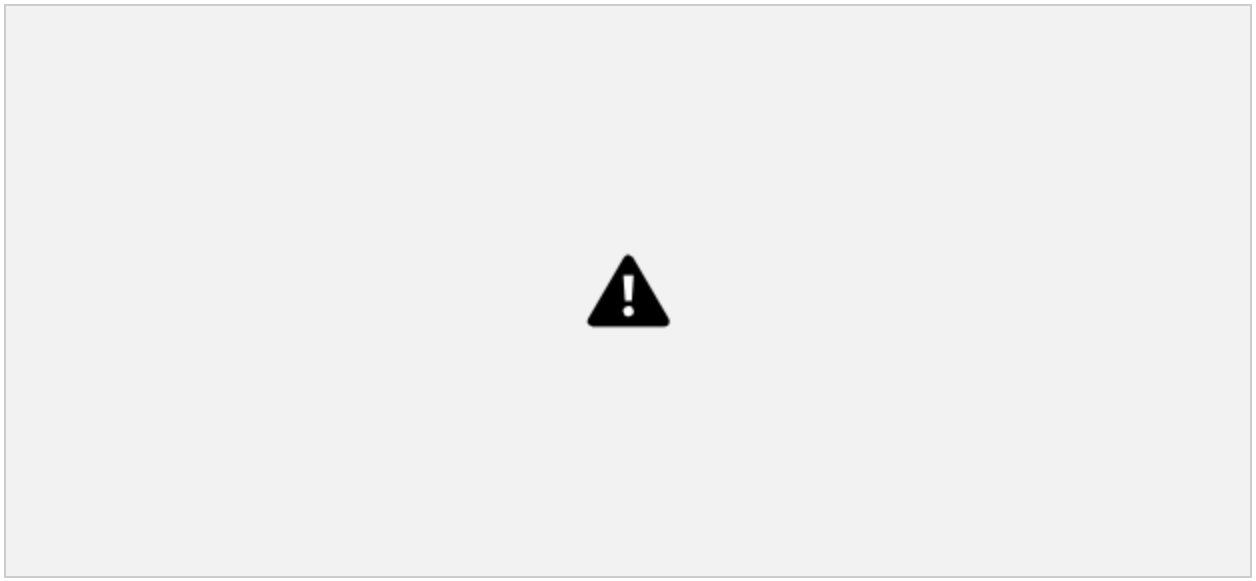


这位旅行商（姑且称之为Opus吧）要前往这5个城市，同时要确保旅程 最短。为此，可考虑前往这些城市的各种可能顺序。



对于每种顺序，他都计算总旅程，再挑选出旅程最短的路线。5个城市有120种不同的排列方式。因此，在涉及5个城市时，解决这个问题需要执行120次操作。涉及6个城市时，需要执行720次操作（有720种不同的

排列方式）。涉及7个城市时，需要执行5040次操作！



推而广之，涉及*n*个城市时，需要执行*n*!（*n*的阶乘）次操作才能计算出结果。因此运行时间为*O*(*n*!)，即阶乘时间。除非涉及的城市数很少，否则需要执行非常多的操作。如果涉及的城市数超过100，根本就不能在合理的时间内计算出结果——等你计算出结果，太阳都没了。

这种算法很糟糕！Opus应使用别的算法，可他别无选择。这是计算机科学领域待解的问题之一。对于这个问题，目前还没有找到更快的算法，有些很聪明的人认为这个问题根本就没有更巧妙的算法。面对这个问 题，我们能做的只是去找出近似答案，更详细的信息请参阅第10章。

最后需要指出的一点是，高水平的读者可研究一下二叉树，这在最后一章做了简要的介绍。

1.4 小结

二分查找的速度比简单查找快得多。

O(log *n*)比*O*(*n*)快。需要搜索的元素越多，前者比后者就快得越 多。

算法运行时间并不以秒为单位。

算法运行时间是从其增速的角度度量的。

算法运行时间用大O表示法表示。

第 2 章　选择排序

本章内容

学习两种最基本的数据结构——数组和链表，它们无处不在。

第1章使用了数组，其他各章几乎也都将用到数组。数组是个重要的主题，一定要高度重视！但在有些情况下，使用链表比使用数组更合适。本章阐述数组和链表的优缺点，让你能够根据要实现的算法选择合适的一个。

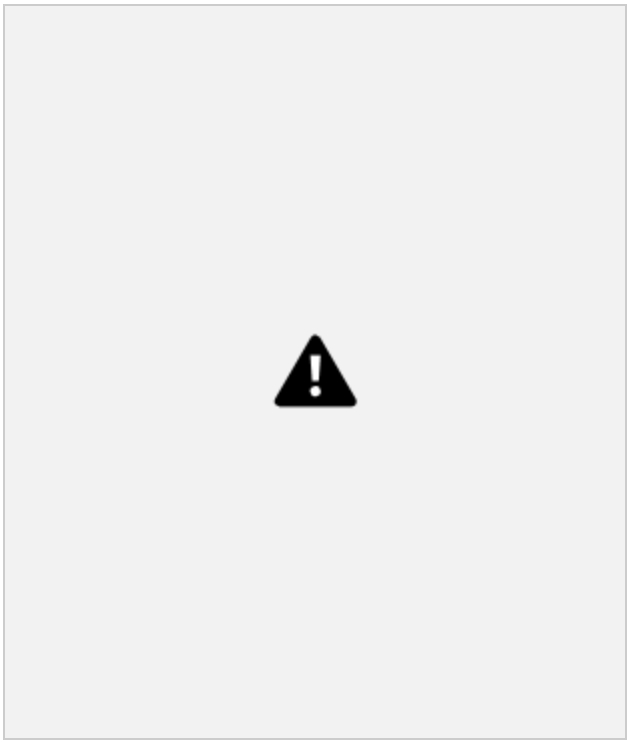
学习第一种排序算法。很多算法仅在数据经过排序后才管用。还记得二分查找吗？它只能用于有序元素列表。本章将介绍选择排序。很多语言都内置了排序算法，因此你基本上不用从头开始编写自己的版本。但选择排序是下一章将介绍的快速排序的基石。快速排序是一种重要的算法，如果你熟悉其他排序算法，理解起来将更容易。

需要具备的知识

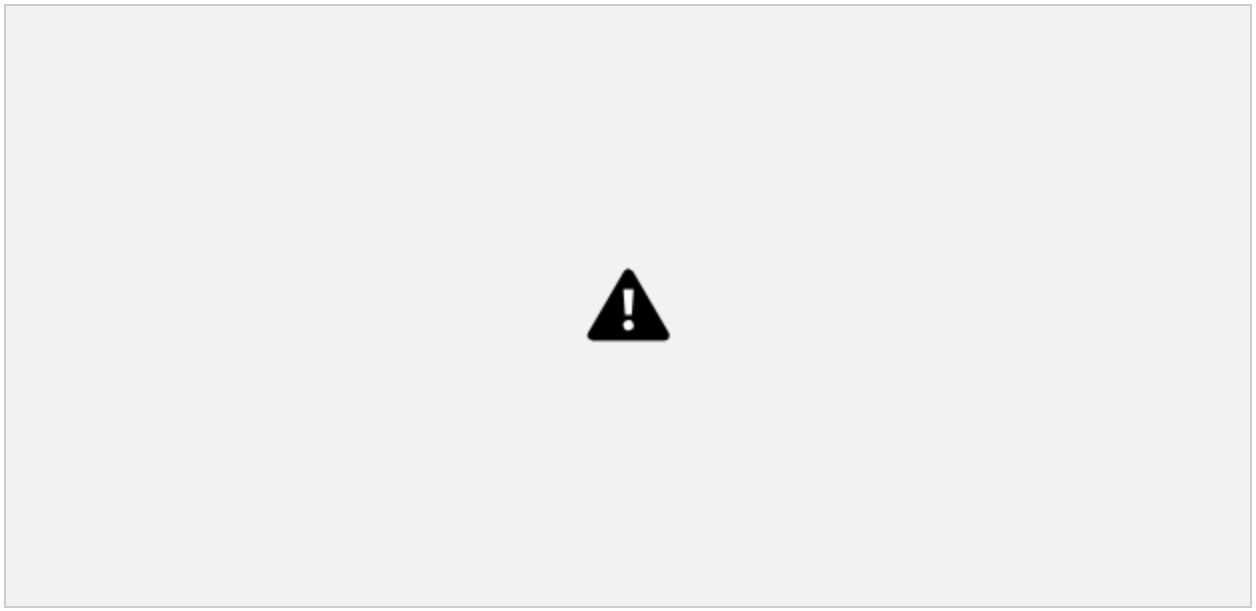
要明白本章的性能分析部分，必须知道大O表示法和对数。如果你不懂，建议回过头去阅读第1章。本书余下的篇幅都会用到大O表示法。

2.1 内存的工作原理

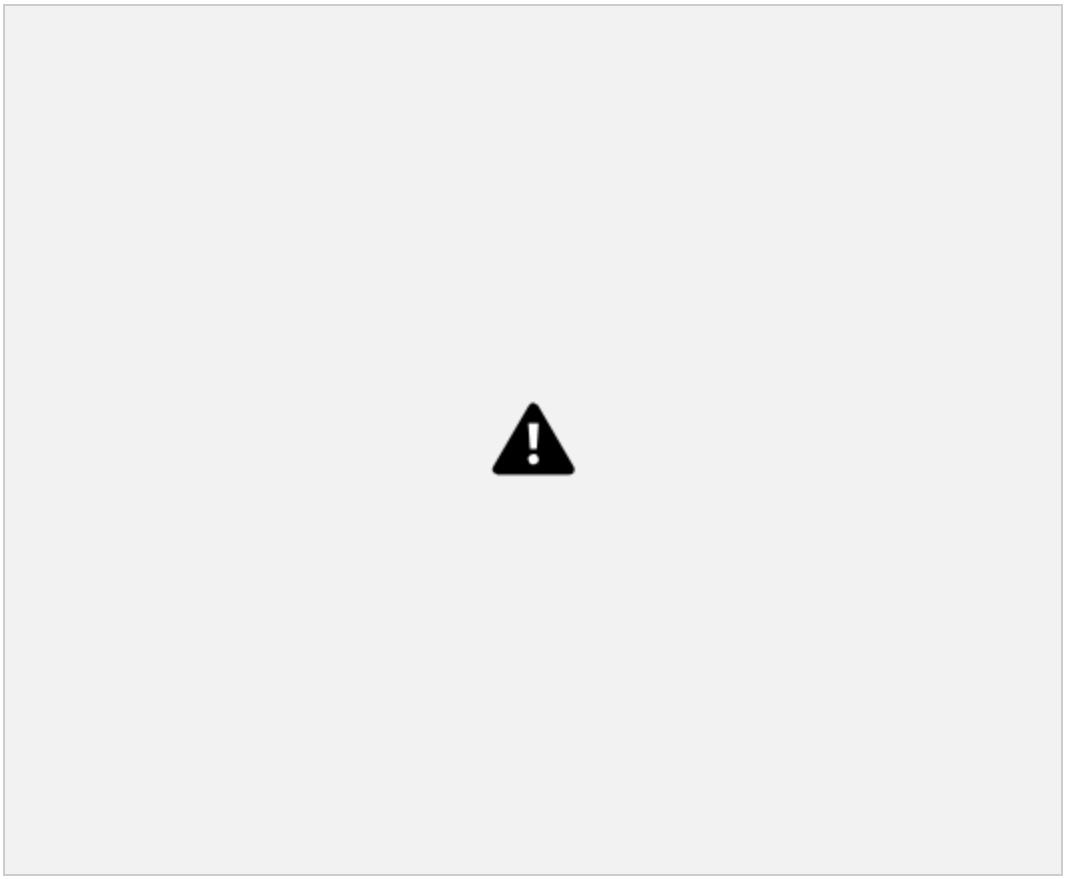
假设你去看演出，需要将东西寄存。寄存处有一个柜子，柜子有很多抽屉。



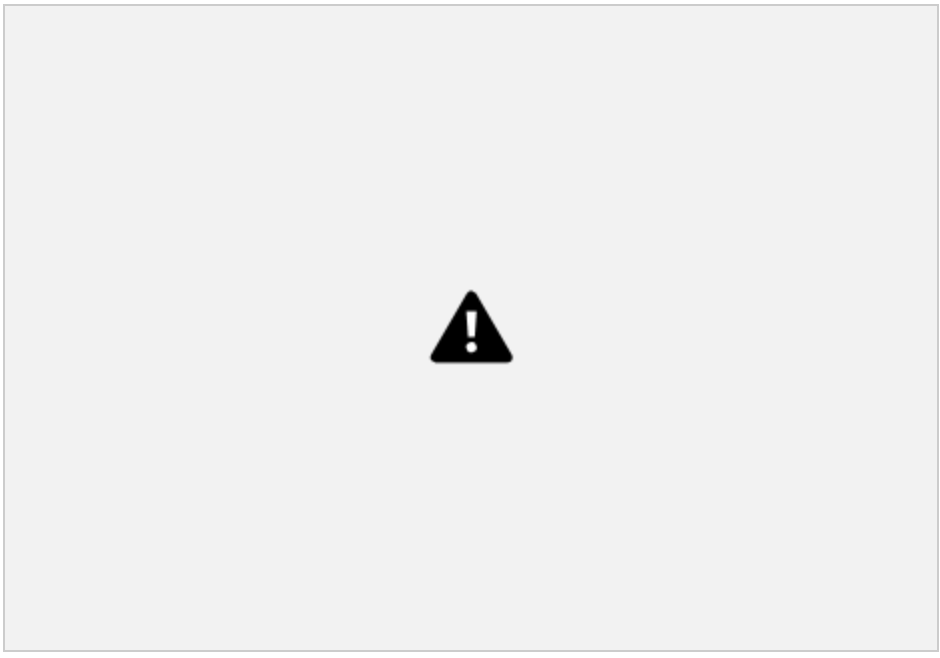
每个抽屉可放一样东西，你有两样东西要寄存，因此要了两个抽屉。



你将两样东西存放在这里。



现在你可以去看演出了！这大致就是计算机内存的工作原理。计算机就像是很多抽屉的集合体，每个抽屉都有地址。



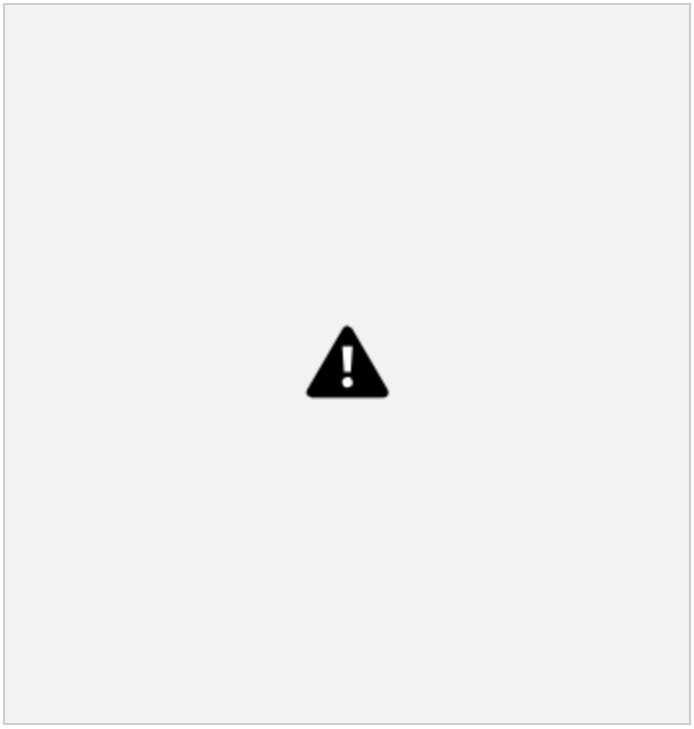
fe0ffeeb是一个内存单元的地址。

需要将数据存储到内存时，你请求计算机提供存储空间，计算机给你一个存储地址。需要存储多项数据时，有两种基本方式——数组和链表。但它们并非都适用于所有的情形，因此知道它们的差别很重要。接下来

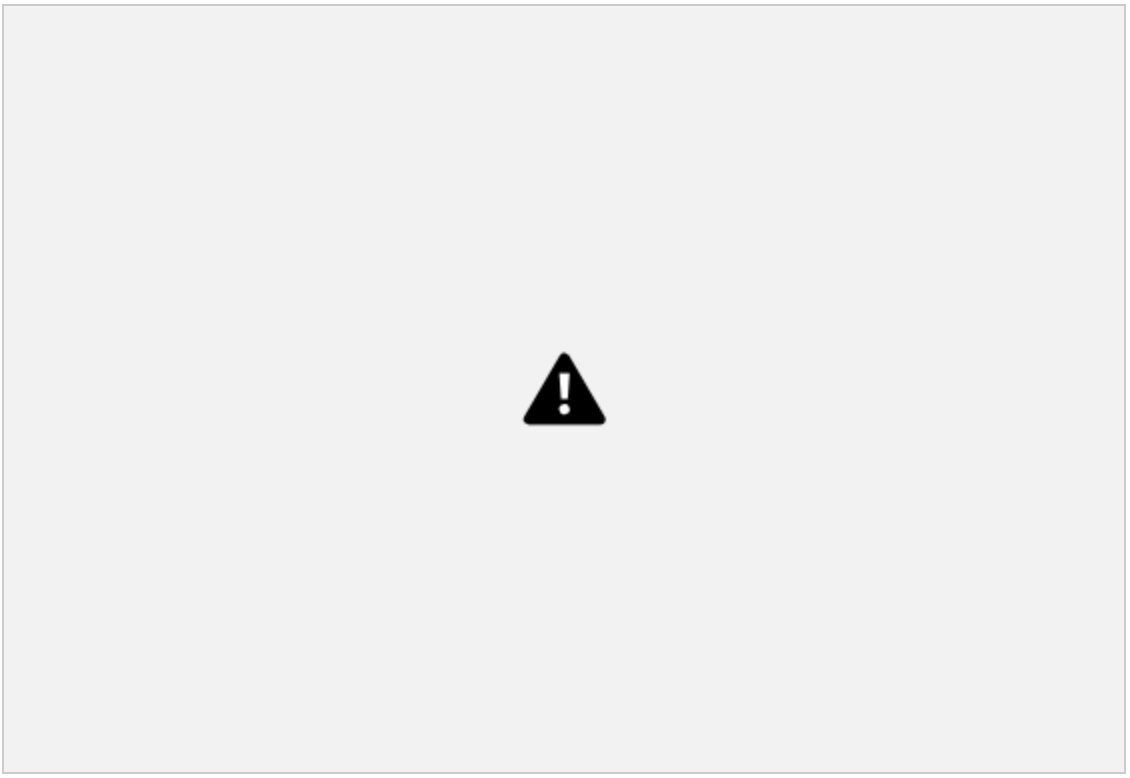
介绍数组和链表以及它们的优缺点。

2.2 数组和链表

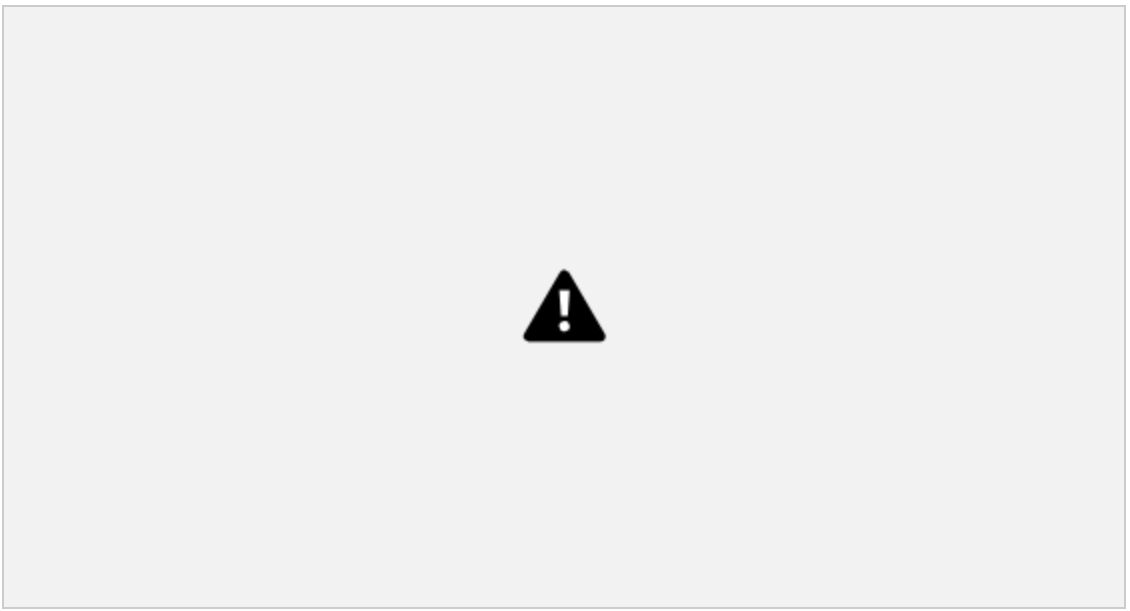
有时候，需要在内存中存储一系列元素。假设你要编写一个管理待办事项的应用程序，为此需要将这些待办事项存储在内存中。



应使用数组还是链表呢？鉴于数组更容易掌握，我们先将待办事项存储在数组中。使用数组意味着所有待办事项在内存中都是相连的（紧靠在一起的）。



现在假设你要添加第四个待办事项，但后面的那个抽屉放着别人的东西！



这就像你与朋友去看电影，找到地方就坐后又来了一位朋友，但原来坐的地方没有空位置，只得再找一个可坐下所有人的地方。在这种情况下，你需要请求计算机重新分配一块可容纳4个待办事项的内存，再将

所有待办事项都移到那里。

如果又来了一位朋友，而当前坐的地方也没有空位，你们就得再次转移！真是太麻烦了。同样，在数组中添加新元素也可能很麻烦。如果没

有了空间，就得移到内存的其他地方，因此添加新元素的速度会很慢。一种解决之道是“预留座位”：即便当前只有3个待办事项，也请计算机提供10个位置，以防需要添加待办事项。这样，只要待办事项不超过10 个，就无需转移。这是一个不错的权变措施，但你应该明白，它存在如 下两个缺点。

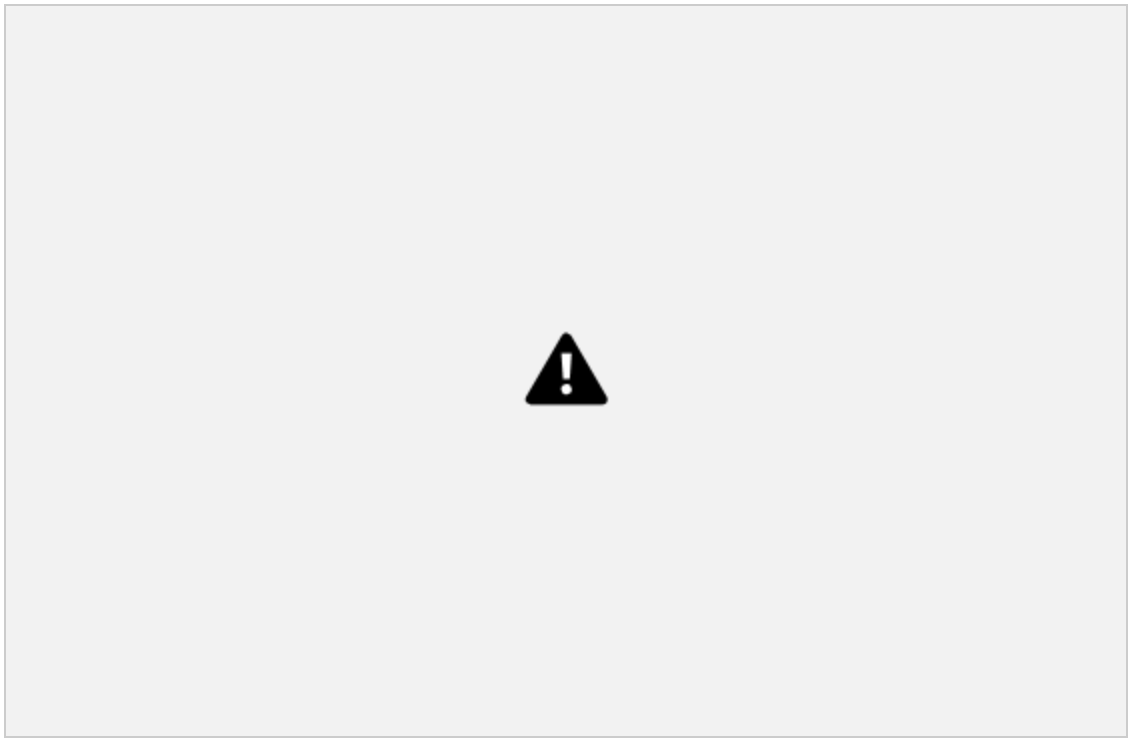
你额外请求的位置可能根本用不上，这将浪费内存。你没有使用， 别人也用不了。

待办事项超过10个后，你还得转移。

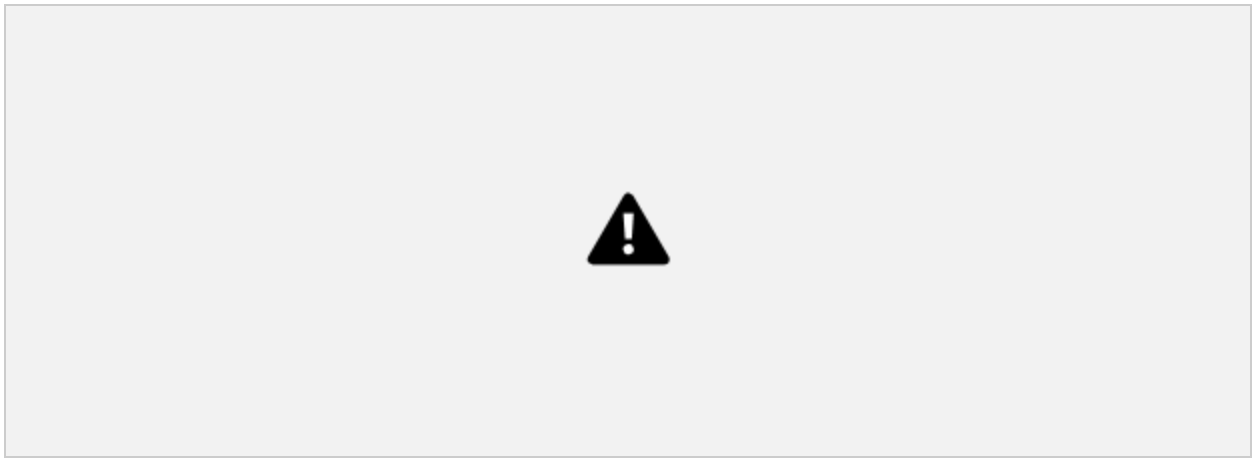
因此，这种权宜措施虽然不错，但绝非完美的解决方案。对于这种问题，可使用链表来解决。

2.2.1 链表

链表中的元素可存储在内存的任何地方。



链表的每个元素都存储了下一个元素的地址，从而使一系列随机的内存地址串在一起。



这犹如寻宝游戏。你前往第一个地址，那里有一张纸条写着“下一个元素的地址为123”。因此，你前往地址123，那里又有一张纸条，写着“下一个元素的地址为847”，以此类推。在链表中添加元素很容易：只需将其放入内存，并将其地址存储到前一个元素中。

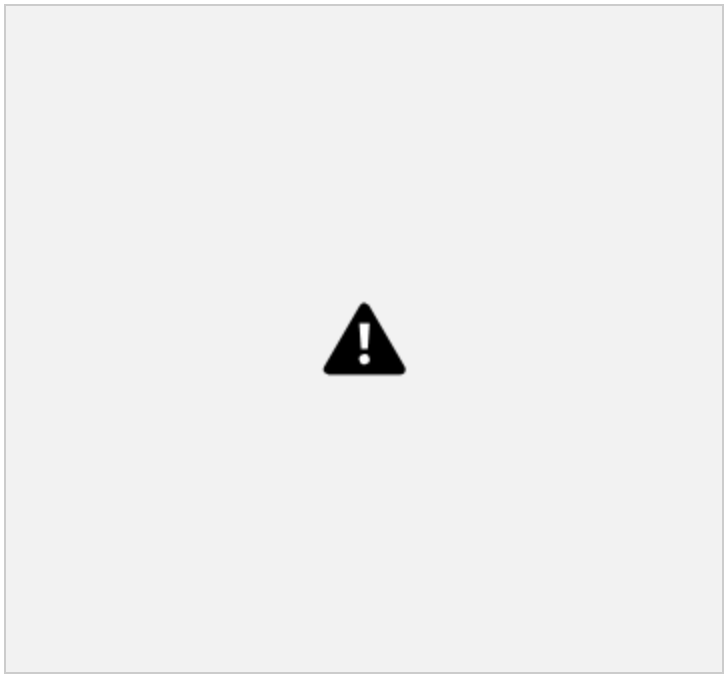
使用链表时，根本就不需要移动元素。这还可避免另一个问题。假设你与五位朋友去看一部很火的电影。你们六人想坐在一起，但看电影的人较多，没有六个在一起的座位。使用数组时有时就会遇到这样的情况。假设你要为数组分配10 000个位置，内存中有10 000个位置，但不都靠在一起。在这种情况下，你将无法为该数组分配内存！链表相当于 说“我们分开来坐”，因此，只要有足够的内存空间，就能为链表分配内存。

链表的优势在插入元素方面，那数组的优势又是什么呢?

2.2.2 数组

排行榜网站使用卑鄙的手段来增加页面浏览量。它们不在一个页面中显示整个排行榜，而将排行榜的每项内容都放在一个页面中，并让你单击Next来查看下一项内容。例如，显示十大电视反派时，不在一个页面中显示整个排行榜，而是先显示第十大反派（Newman）。你必须在每个页面中单击Next，才能看到第一大反派（Gustavo Fring）。这让网站能够在10个页面中显示广告，但用户需要单击Next 九次才能看到第一个，真的是很烦。如果整个排行榜都显示在一个页面中，将方便得多。这

样，用户可单击排行榜中的人名来获得更详细的信息。



链表存在类似的问题。在需要读取链表的最后一个元素时，你不能直接读取，因为你不知道它所处的地址，必须先访问元素#1，从中获取元素#2的地址，再访问元素#2并从中获取元素#3的地址，以此类推，直到访 问最后一个元素。需要同时读取所有元素时，链表的效率很高：你读取 第一个元素，根据其中的地址再读取第二个元素，以此类推。但如果你 需要跳跃，链表的效率真的很低。

数组与此不同：你知道其中每个元素的地址。例如，假设有一个数组，它包含五个元素，起始地址为00，那么元素#5的地址是多少呢?

只需执行简单的数学运算就知道：04。需要随机地读取元素时，数组的效率很高，因为可迅速找到数组的任何元素。在链表中，元素并非靠在一起的，你无法迅速计算出第五个元素的内存地址，而必须先访问第一个元素以获取第二个元素的地址，再访问第二个元素以获取第三个元素的地址，以此类推，直到访问第五个元素。

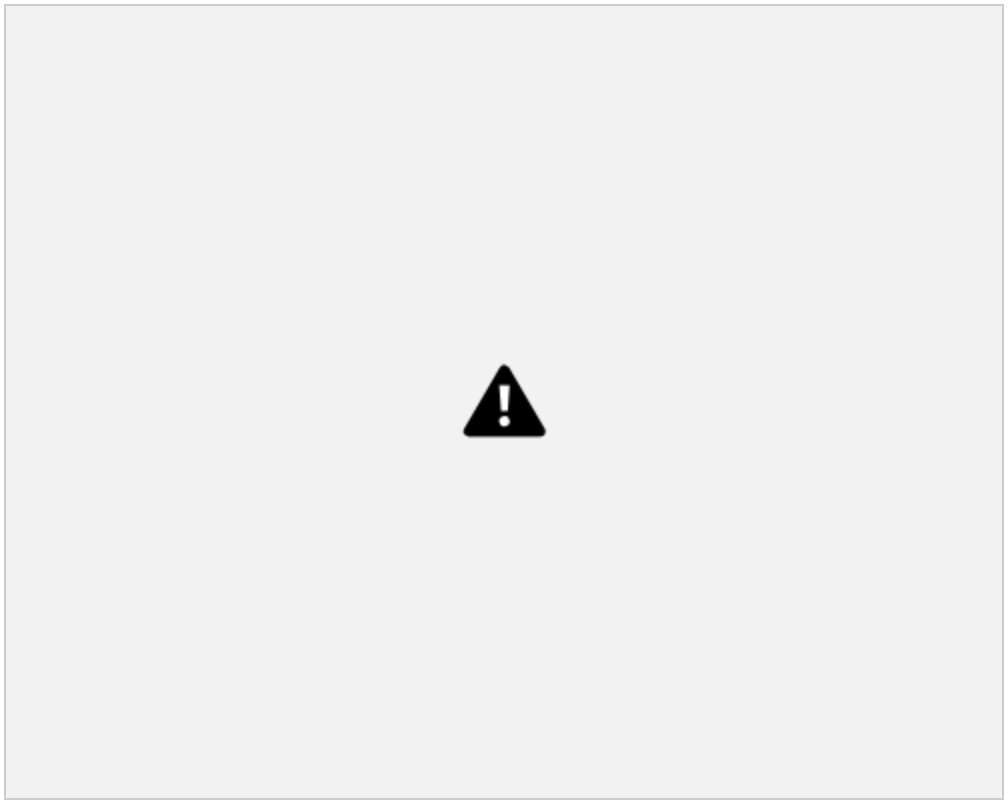
2.2.3 术语

数组的元素带编号，编号从0而不是1开始。例如，在下面的数组中，元素20的位置为1。

而元素10的位置为0。这通常会让新手晕头转向。从0开始让基于数组的代码编写起来更容易，因此程序员始终坚持这样做。几乎所有的编程语言都从0开始对数组元素进行编号。你很快就会习惯这种做法。

元素的位置称为索引。因此，不说“元素20的位置为1”，而说“元素20位于索引1处”。本书将使用索引来表示位置。

下面列出了常见的数组和链表操作的运行时间。



问题：在数组中插入元素时，为何运行时间为$O(n)$呢？假设要在数组开头插入一个元素，你将如何做？这需要多长时间？请阅读下一节，找出这些问题的答案！

练习

2.1 假设你要编写一个记账的应用程序。

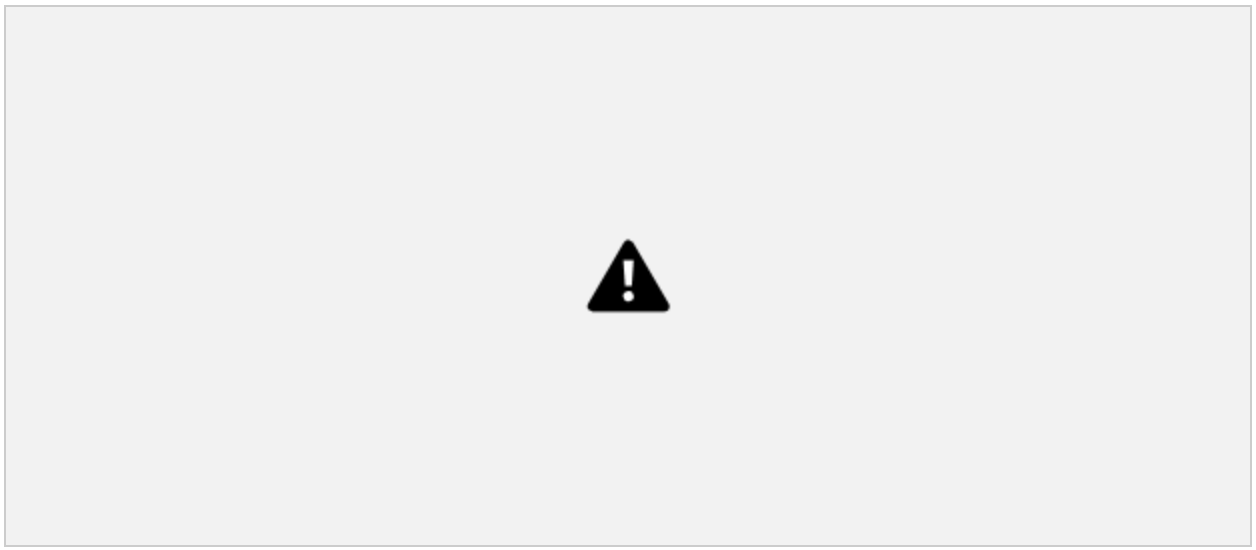
你每天都将所有的支出记录下来，并在月底统计支出，算算当月花了多少钱。因此，你执行的插入操作很多，但读取操作很少。该使用数组还是链表呢？

2.2.4 在中间插入

假设你要让待办事项按日期排列。之前，你在清单末尾添加了待办事

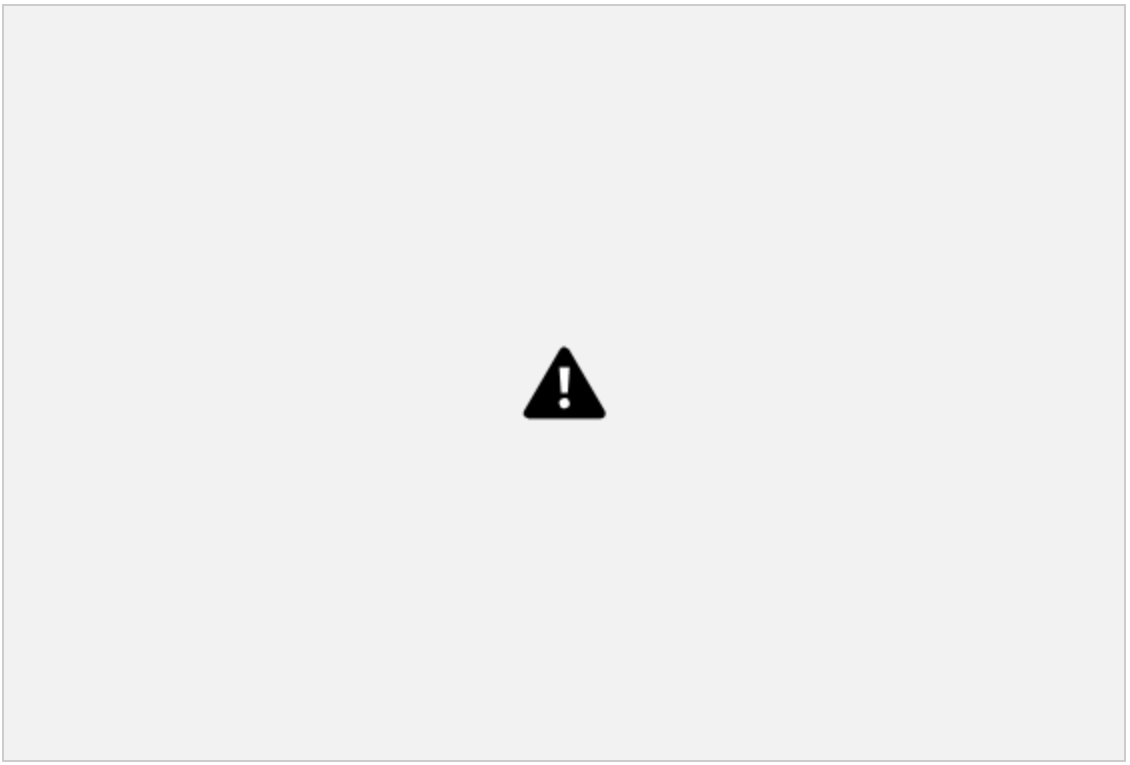
项。

但现在你要根据新增待办事项的日期将其插入到正确的位置。



需要在中间插入元素时，数组和链表哪个更好呢？使用链表时，插入元素很简单，只需修改它前面的那个元素指向的地址。

而使用数组时，则必须将后面的元素都向后移。



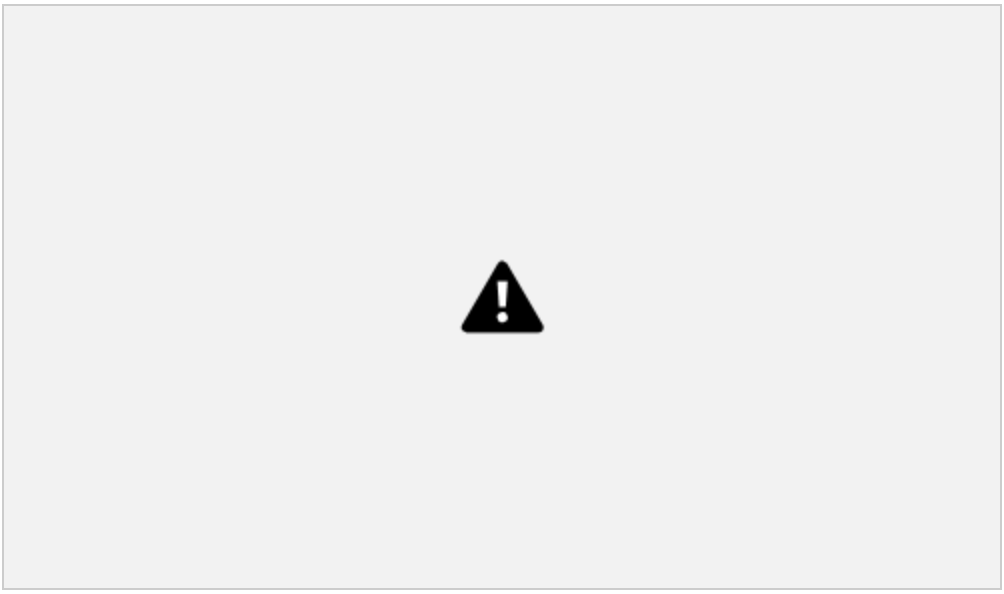
如果没有足够的空间，可能还得将整个数组复制到其他地方！因此，当需要在中间插入元素时，链表是更好的选择。

2.2.5 删除

如果你要删除元素呢？链表也是更好的选择，因为只需修改前一个元素指向的地址即可。而使用数组时，删除元素后，必须将后面的元素都向前移。

不同于插入，删除元素总能成功。如果内存中没有足够的空间，插入操作可能失败，但在任何情况下都能够将元素删除。

下面是常见数组和链表操作的运行时间。

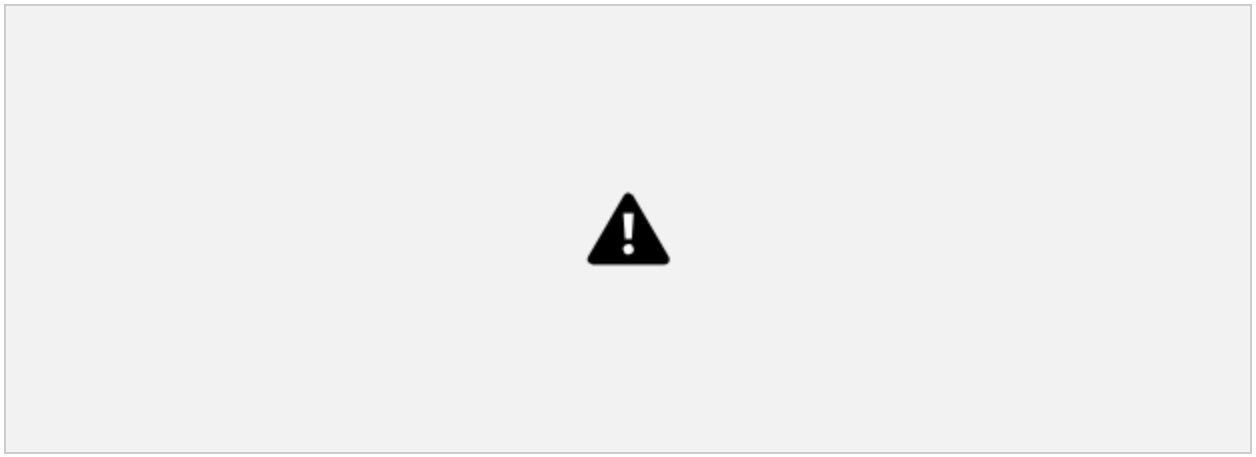


需要指出的是，仅当能够立即访问要删除的元素时，删除操作的运行时间才为*O*(1)。通常我们都记录了链表的第一个元素和最后一个元素，因此删除这些元素时运行时间为*O*(1)。

数组和链表哪个用得更多呢？显然要看情况。但数组用得很多，因为它支持随机访问。有两种访问方式：随机访问和顺序访问。顺序访问意味着从第一个元素开始逐个地读取元素。链表只能顺序访问：要读取链表的第十个元素，得先读取前九个元素，并沿链接找到第十个元素。随机访问意味着可直接跳到第十个元素。本书经常说数组的读取速度更快，这是因为它们支持随机访问。很多情况都要求能够随机访问，因此数组用得很多。数组和链表还被用来实现其他数据结构，这将在本书后面介绍。

练习

2.2　假设你要为饭店创建一个接受顾客点菜单的应用程序。这个应用程序存储一系列点菜单。服务员添加点菜单，而厨师取出点菜单并制作菜肴。这是一个点菜单队列：服务员在队尾添加点菜单，厨师取出队列开头的点菜单并制作菜肴。



你使用数组还是链表来实现这个队列呢？（提示：链表擅长插入和删 除，而数组擅长随机访问。在这个应用程序中，你要执行的是哪些操作呢？）

2.3 我们来做一个思考实验。假设Facebook记录一系列用户名，每当有 用户试图登录Facebook时，都查找其用户名，如果找到就允许用户登 录。由于经常有用户登录Facebook，因此需要执行大量的用户名查找操 作。假设Facebook使用二分查找算法，而这种算法要求能够随机访问 ——立即获取中间的用户名。考虑到这一点，应使用数组还是链表来存 储用户名呢?

2.4 经常有用户在Facebook注册。假设你已决定使用数组来存储用户名，在插入方面数组有何缺点呢？具体地说，在数组中添加新用户将出现什么情况?

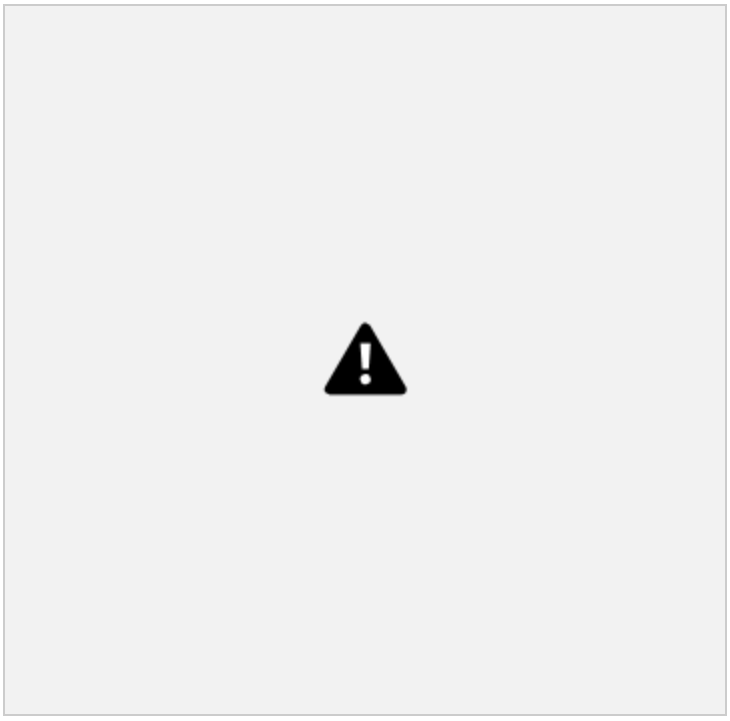
2.5 实际上，Facebook存储用户信息时使用的既不是数组也不是链表。 假设Facebook使用的是一种混合数据：链表数组。这个数组包含26个元 素，每个元素都指向一个链表。例如，该数组的第一个元素指向的链表
包含所有以A打头的用户名，第二个元素指向的链表包含所有以B打头的用户名，以此类推。

假设Adit B在Facebook注册，而你需要将其加入前述数据结构中。因此，你访问数组的第一个元素，再访问该元素指向的链表，并将Adit B添加到这个链表末尾。现在假设你要查找Zakhir H。因此你访问第26个元素，再在它指向的链表（该链表包含所有以z打头的用户名）中查找Zakhir H。

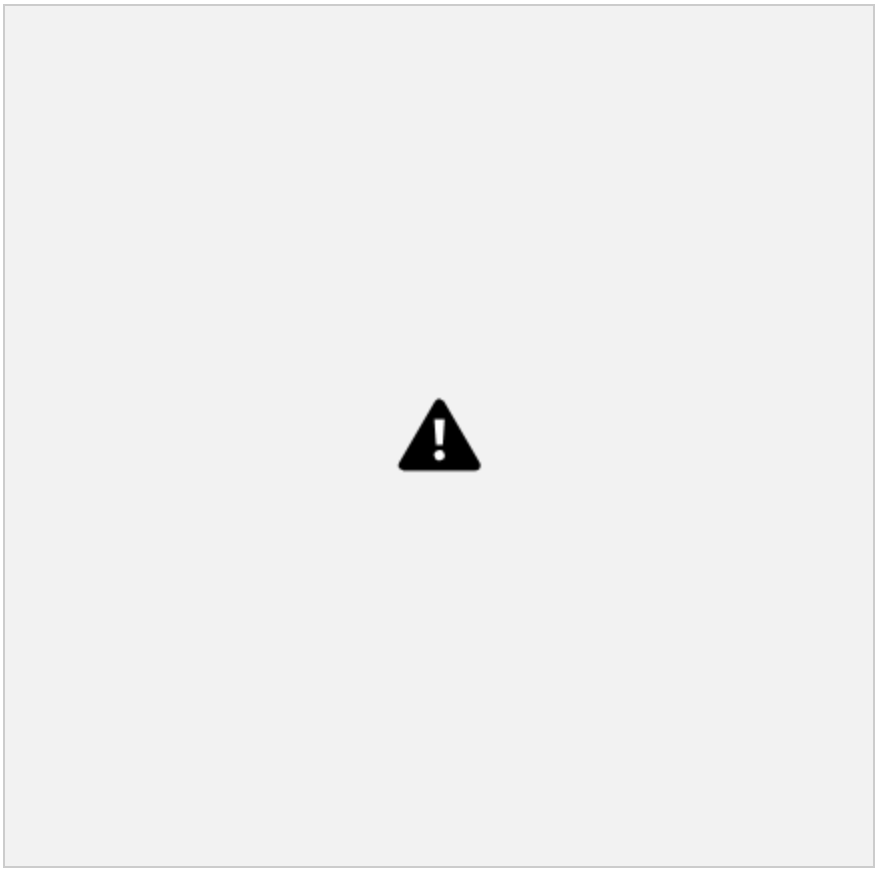
请问，相比于数组和链表，这种混合数据结构的查找和插入速度更慢还是更快？你不必给出大O运行时间，只需指出这种新数据结构的查找和插入速度更快还是更慢。

2.3 选择排序

有了前面的知识，你就可以学习第二种算法——选择排序了。要理解本节的内容，你必须熟悉数组、链表和大O表示法。



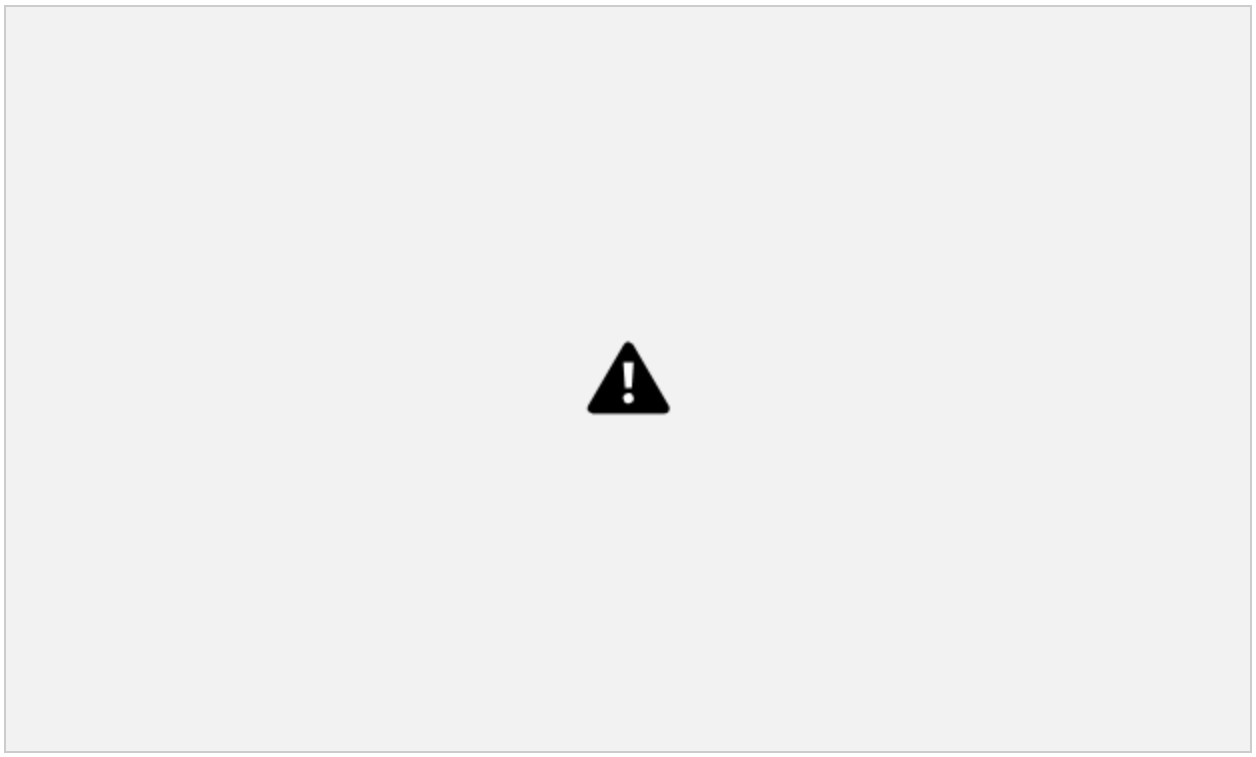
假设你的计算机存储了很多乐曲。对于每个乐队，你都记录了其作品被播放的次数。



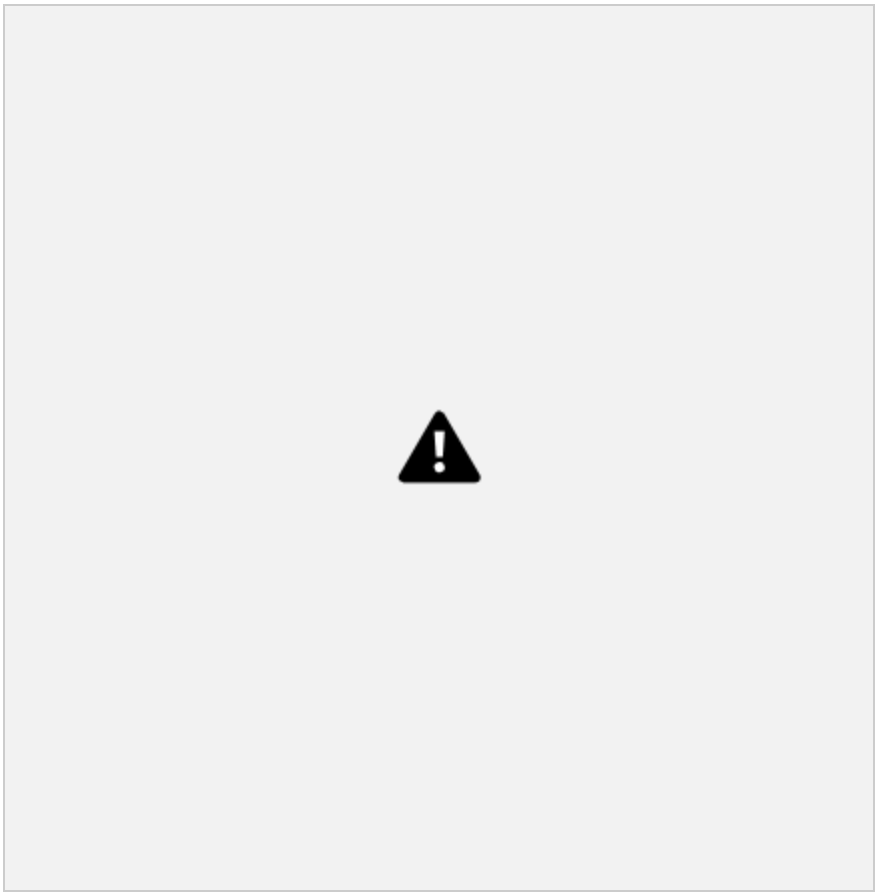
你要将这个列表按播放次数从多到少的顺序排列，从而将你喜欢的乐队排序。该如何做呢?

一种办法是遍历这个列表，找出作品播放次数最多的乐队，并将该乐队添加到一个新列表中。

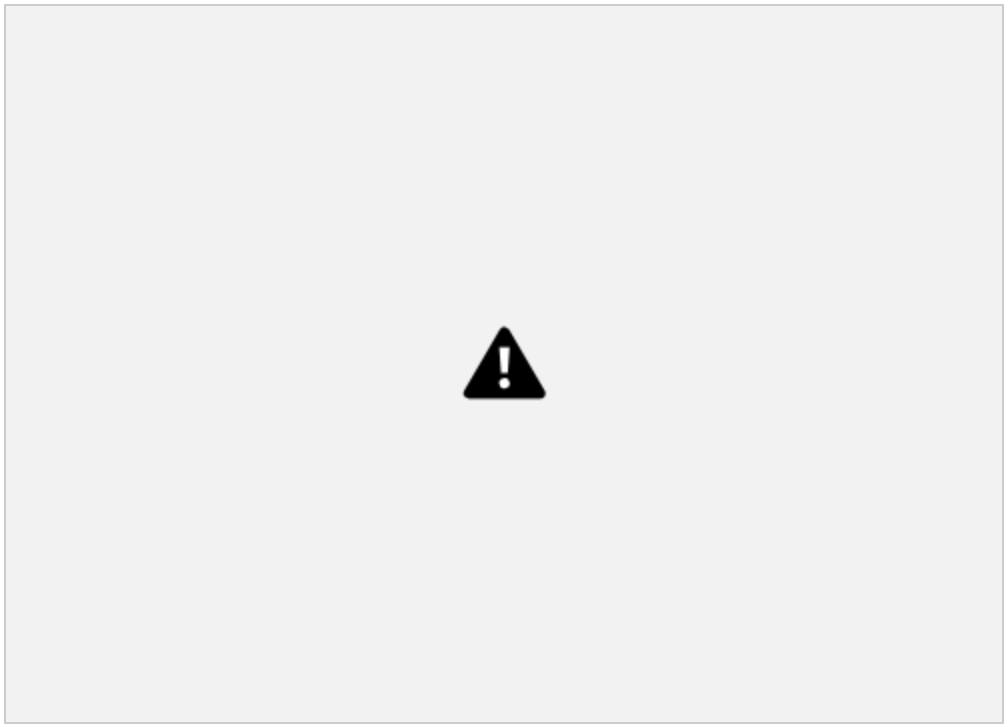
再次这样做，找出播放次数第二多的乐队。



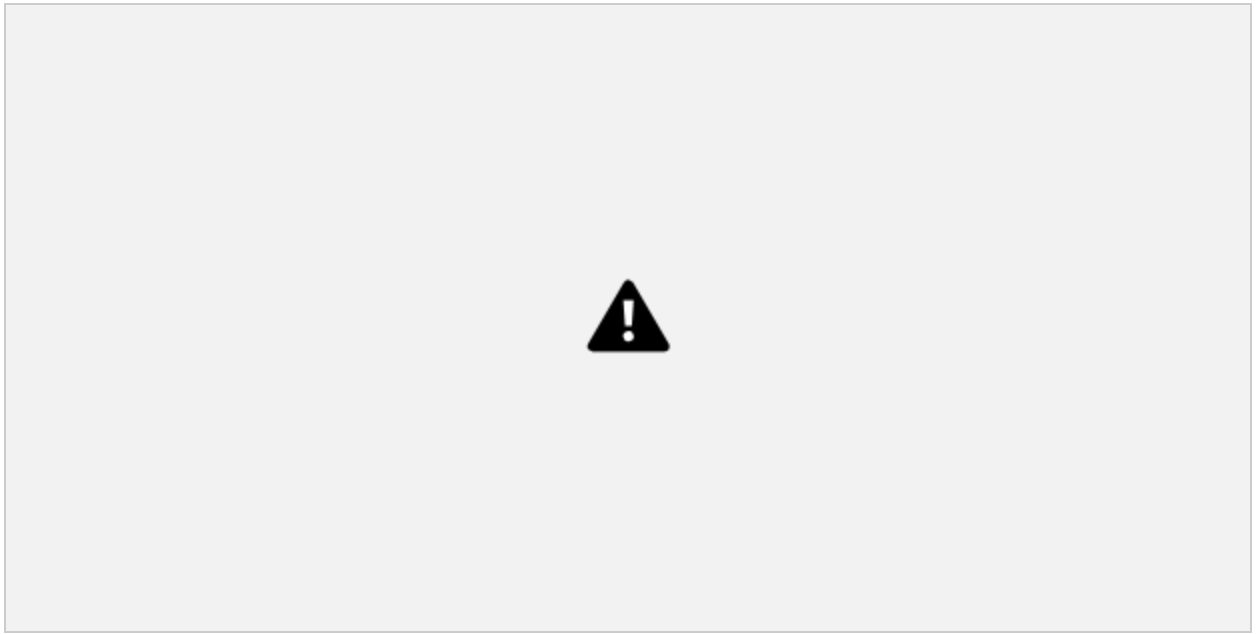
继续这样做，你将得到一个有序列表。



下面从计算机科学的角度出发，看看这需要多长时间。别忘了，*O*(*n*)时间意味着查看列表中的每个元素一次。例如，对乐队列表进行简单查找时，意味着每个乐队都要查看一次。



要找出播放次数最多的乐队，必须检查列表中的每个元素。正如你刚才看到的，这需要的时间为$O(n)$。因此对于这种时间为$O(n)$的操作，你需要执行n次。



需要的总时间为 $O(n \times n)$，即$O(n^2)$。

排序算法很有用。你现在可以对如下内容进行排序：

电话簿中的人名

旅行日期

电子邮件（从新到旧）

需要检查的元素数越来越少

随着排序的进行，每次需要检查的元素数在逐渐减少，最后一次需要检查的元素都只有一个。既然如此，运行时间怎么还是$O(n^2)$呢？这个问题问得好，这与大O表示法中的常数相关。第4章将详细解释，这里只简单地说一说。

你说得没错，并非每次都需要检查n个元素。第一次需要检查n个元素，但随后检查的元素数依次为n - 1, n – 2, …, 2和1。平均每次检查的元素数为1/2 × n，因此运行时间为$O(n \times 1/2 \times n)$。但大O表示法省略诸如1/2这样的常数（有关这方面的完整讨论，请参阅第4章），因此简单地写作$O(n \times n)$或$O(n^2)$。

选择排序是一种灵巧的算法，但其速度不是很快。快速排序是一种更快的排序算法，其运行时间为$O(n \log n)$，这将在下一章介绍。

示例代码

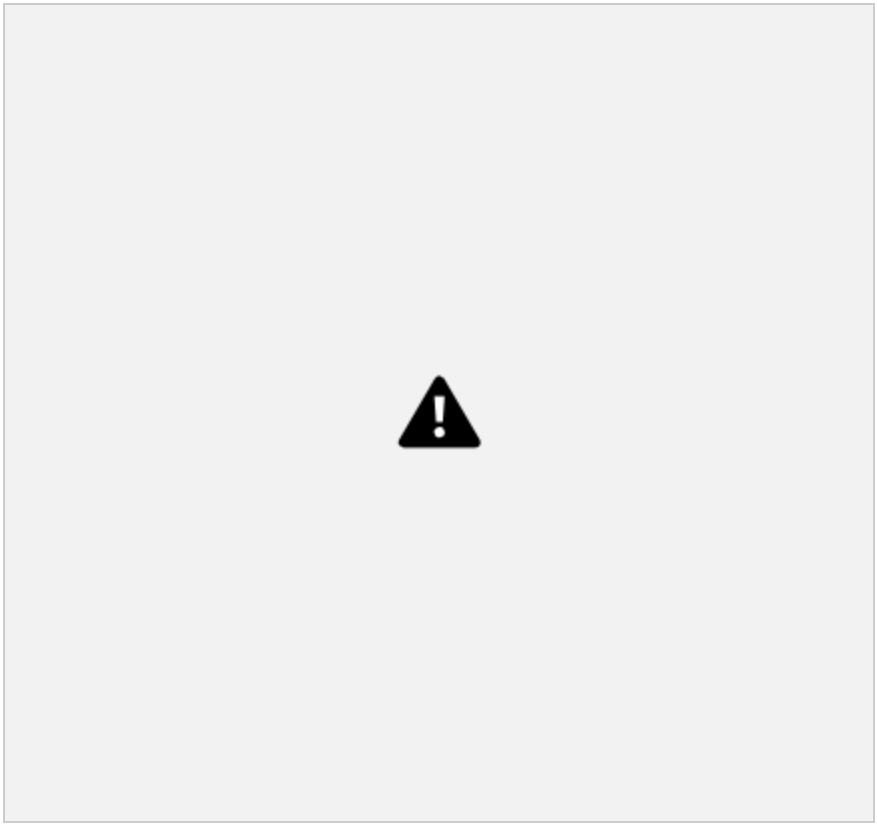
前面没有列出对乐队进行排序的代码，但下述代码提供了类似的功能：将数组元素按从小到大的顺序排列。先编写一个用于找出数组中最小元素的函数。

```
def findSmallest(arr):
 smallest = arr[0] ←------存储最小的值
smallest_index = 0 ←------存储最小元素的索引
for i in range(1, len(arr)):
 if arr[i] < smallest:
 smallest = arr[i]
 smallest_index = i
 return smallest_index
```


现在可以使用这个函数来编写选择排序算法了。

```
def selectionSort(arr): ←------对数组进行排序
 newArr = []
 for i in range(len(arr)):
 smallest = findSmallest(arr) ←------找出数组中最小的元素，并将其加入到新数组
newArr.append(arr.pop(smallest))
 return newArr

print selectionSort([5, 3, 6, 2, 10])
```



2.4 小结

计算机内存犹如一大堆抽屉。

需要存储多个元素时，可使用数组或链表。

数组的元素都在一起。

链表的元素是分开的，其中每个元素都存储了下一个元素的地址。

数组的读取速度很快。

链表的插入和删除速度很快。

在同一个数组中，所有元素的类型都必须相同（都为int、double 等）。

第 3 章　递归

本章内容

学习递归。递归是很多算法都使用的一种编程方法，是理解本 书后续内容的关键。

学习如何将问题分成基线条件和递归条件。第4章将介绍的分 而治之策略使用这种简单的概念来解决棘手的问题。

我怀着激动的心情编写本章，因为它介绍的是递归——一种优雅的问题解决方法。递归是我最喜欢的主题之一，它将人分成三个截然不同的阵营：恨它的、爱它的以及恨了几年后又爱上它的。我本人属于第三个阵营。为帮助你理解，现有以下建议。

本章包含很多示例代码，请运行它们，以便搞清楚其中的工作原 理。

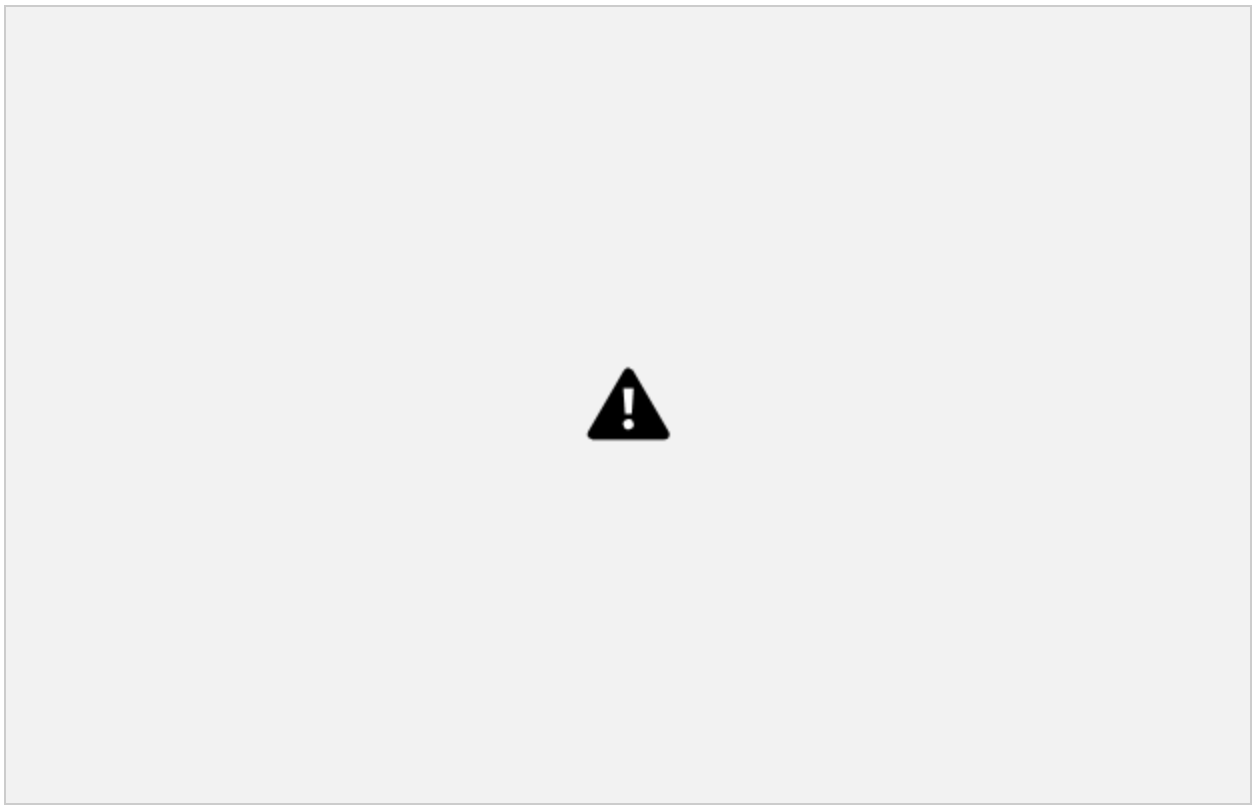
请用纸和笔逐步执行至少一个递归函数，就像这样：我使用5来调 用`factorial`，这将使用4调用`factorial`，并将返回结果乘以5， 以此类推。这样逐步执行递归函数可搞明白递归函数的工作原理。

本章还包含大量伪代码。伪代码是对手头问题的简要描述，看着像代码，但其实更接近自然语言。

3.1 递归

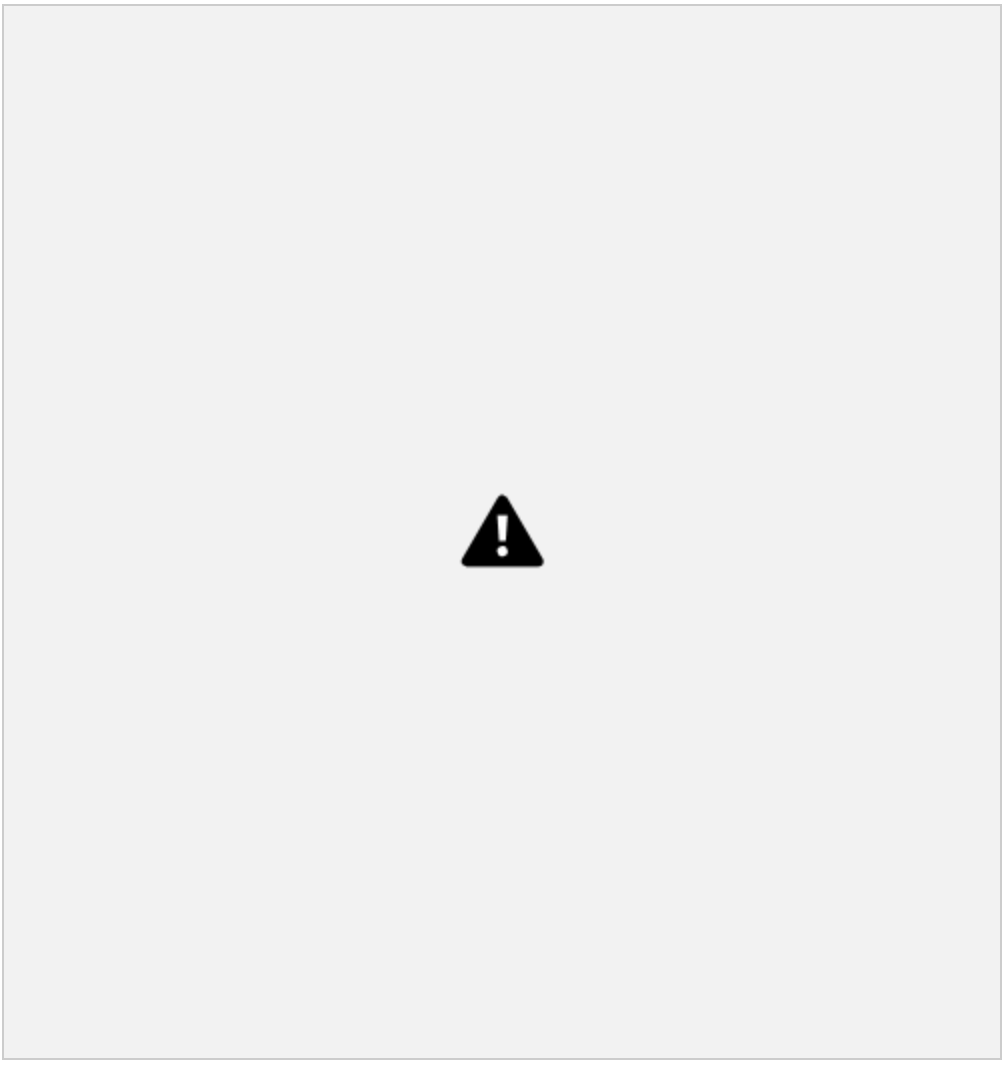
假设你在祖母的阁楼中翻箱倒柜，发现了一个上锁的神秘手提箱。 pdf

祖母告诉你，钥匙很可能在下面这个盒子里。



这个盒子里有盒子，而盒子里的盒子又有盒子。钥匙就在某个盒子中。为找到钥匙，你将使用什么算法？先想想这个问题，再接着往下看。

下面是一种方法。



(1) 创建一个要查找的盒子堆。

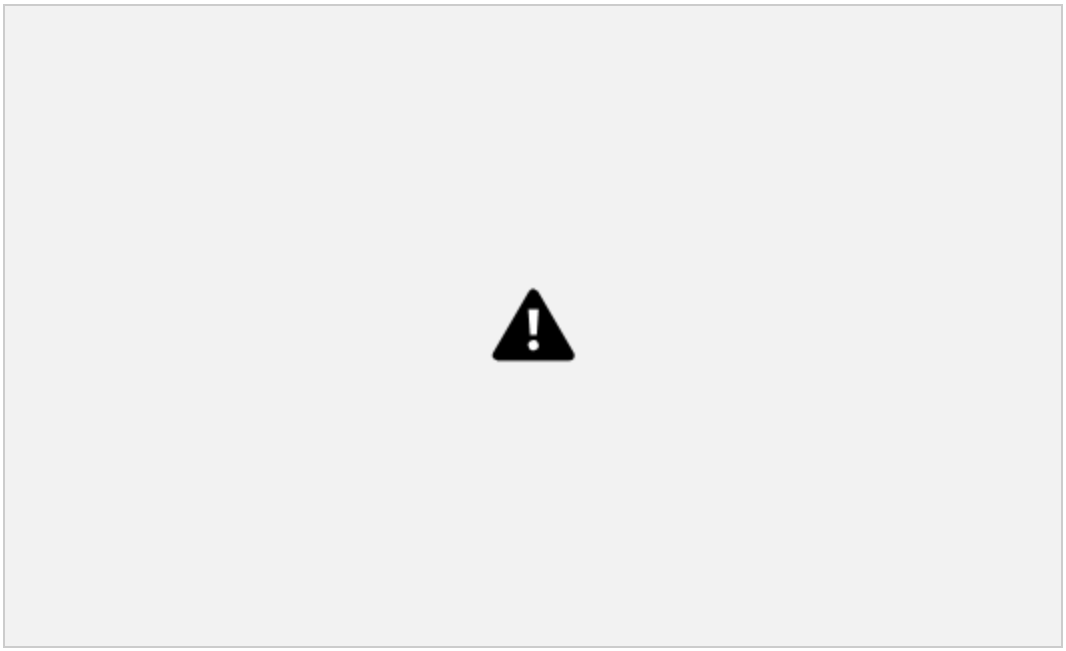
(2) 从盒子堆取出一个盒子，在里面找。

(3) 如果找到的是盒子，就将其加入盒子堆中，以便以后再查找。

(4) 如果找到钥匙，则大功告成！

(5) 回到第二步。

下面是另一种方法。



(1) 检查盒子中的每样东西。

(2) 如果是盒子，就回到第一步。

(3) 如果是钥匙，就大功告成！

在你看来，哪种方法更容易呢？第一种方法使用的是while循环：只要盒子堆不空，就从中取一个盒子，并在其中仔细查找。

```
def look_for_key(main_box):
 pile = main_box.make_a_pile_to_look_through()
 while pile is not empty:
 box = pile.grab_a_box()
 for item in box:
 if item.is_a_box():
 pile.append(item)
 elif item.is_a_key():
 print "found the key!"
```

第二种方法使用递归——函数调用自己，这种方法的伪代码如下。

```
def look_for_key(box):
 for item in box:
 if item.is_a_box():
```

```
 look_for_key(item) ←------递归!
elif item.is_a_key():
 print "found the key!"
```

这两种方法的作用相同，但在我看来，第二种方法更清晰。递归只是让解决方案更清晰，并没有性能上的优势。实际上，在有些情况下，使用循环的性能更好。我很喜欢Leigh Caldwell在Stack Overflow上说的一句话：“如果使用循环，程序的性能可能更高；如果使用递归，程序可能[1]更容易理解。如何选择要看什么对你来说更重要。”

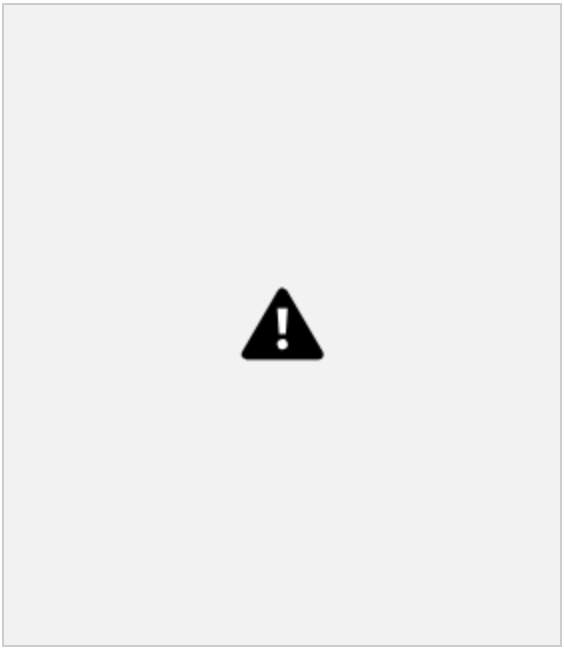
[1] 参见http://stackoverflow.com/a/72694/139117。

很多算法都使用了递归，因此理解这种概念很重要。

3.2 基线条件和递归条件

由于递归函数调用自己，因此编写这样的函数时很容易出错，进而导致无限循环。例如，假设你要编写一个像下面这样倒计时的函数。

```
> 3...2...1
```



为此，你可以用递归的方式编写，如下所示。

```
def countdown(i):
print i
countdown(i-1)
```

如果你运行上述代码，将发现一个问题：这个函数运行起来没完没了！

```
> 3...2...1...0...-1...-2...
```

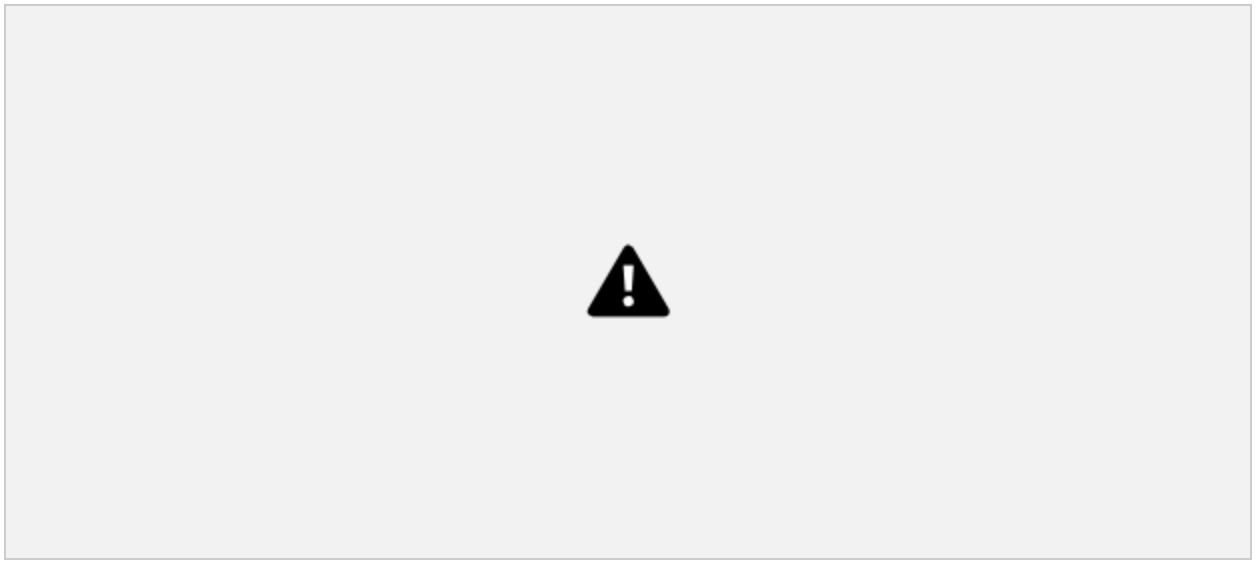
（要让脚本停止运行，可按Ctrl+C。）

编写递归函数时，必须告诉它何时停止递归。正因为如此，每个递归函数都有两部分：基线条件（base case）和递归条件（recursive case）。 递归条件指的是函数调用自己，而基线条件则指的是函数不再调用自 己，从而避免形成无限循环。

我们来给函数countdown添加基线条件。

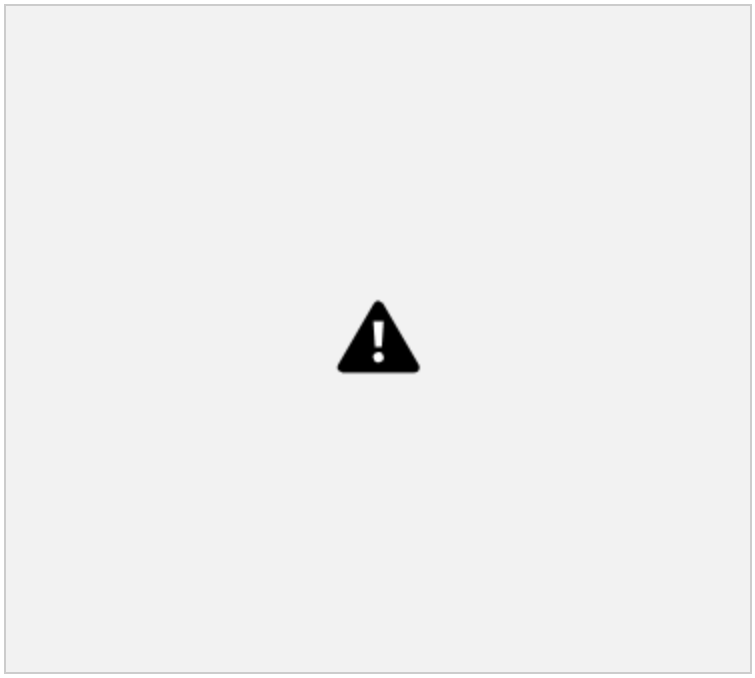
```
def countdown(i):
 print i
 if i <= 1: ←------基线条件
return
 else: ←------递归条件
countdown(i-1)
```

现在，这个函数将像预期的那样运行，如下所示。

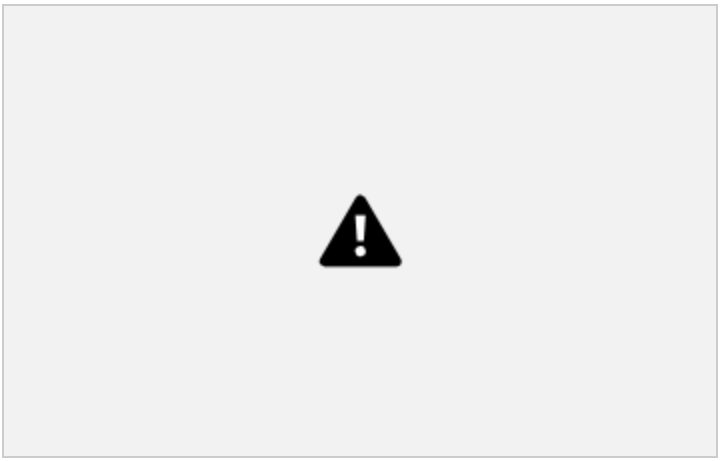


3.3 栈

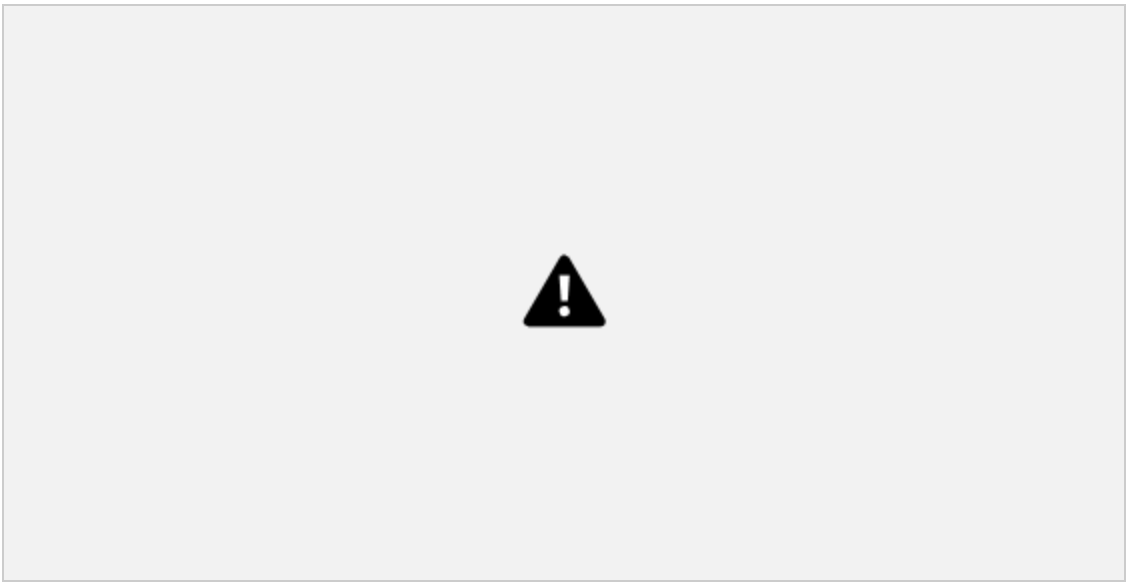
本节将介绍一个重要的编程概念——调用栈（call stack）。调用栈不仅对编程来说很重要，使用递归时也必须理解这个概念。



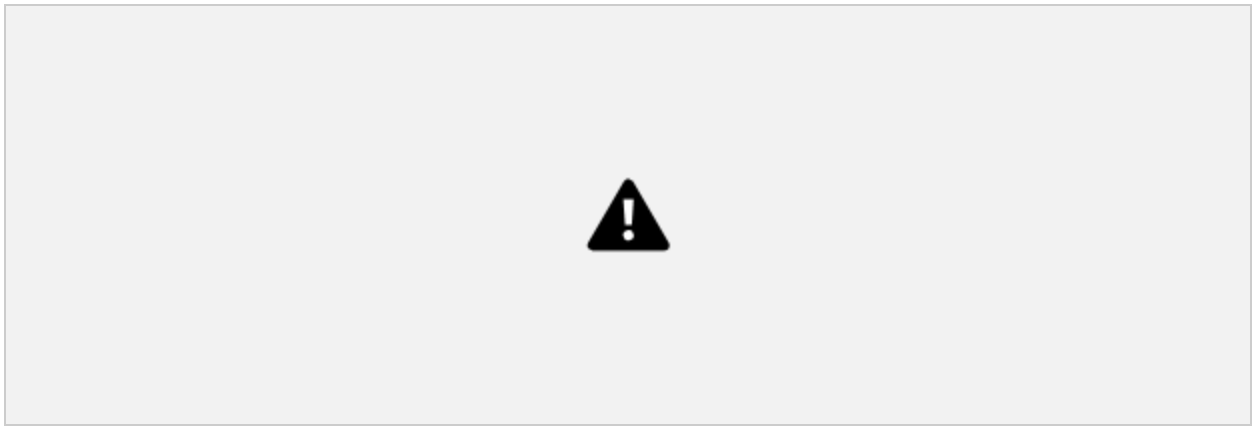
假设你去野外烧烤，并为此创建了一个待办事项清单——一叠便条。



本书之前讨论数组和链表时，也有一个待办事项清单。你可将待办事项添加到该清单的任何地方，还可删除任何一个待办事项。一叠便条要简单得多：插入的待办事项放在清单的最前面；读取待办事项时，你只读取最上面的那个，并将其删除。因此这个待办事项清单只有两种操作：压入（插入）和弹出（删除并读取）。



下面来看看如何使用这个待办事项清单。



这种数据结构称为栈。栈是一种简单的数据结构，刚才我们一直在使用它，却没有意识到！

3.3.1 调用栈

计算机在内部使用被称为调用栈的栈。我们来看看计算机是如何使用调用栈的。下面是一个简单的函数。

```
def greet(name):
 print "hello, " + name + "!"
 greet2(name)
 print "getting ready to say bye..."
 bye()
```

这个函数问候用户，再调用另外两个函数。这两个函数的代码如下。

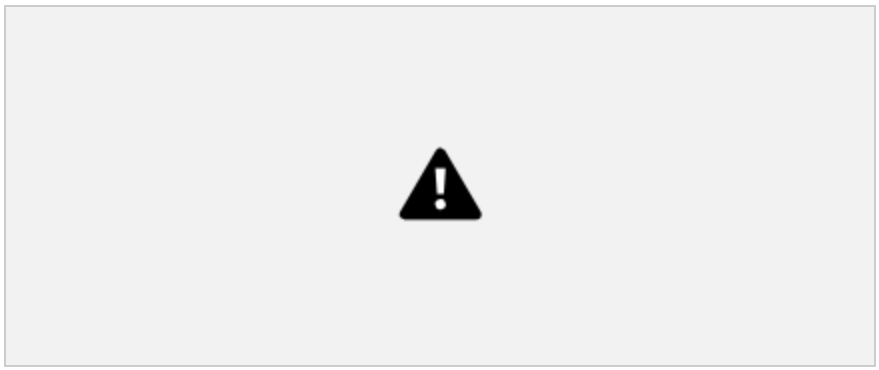
```
def greet2(name):
 print "how are you, " + name + "?"
def bye():
 print "ok bye!"
```

下面详细介绍调用函数时发生的情况。

> 说明
>
> 在Python中，print是一个函数，但出于简化考虑，这里假设它不 是函数。你也这样假设就行了。

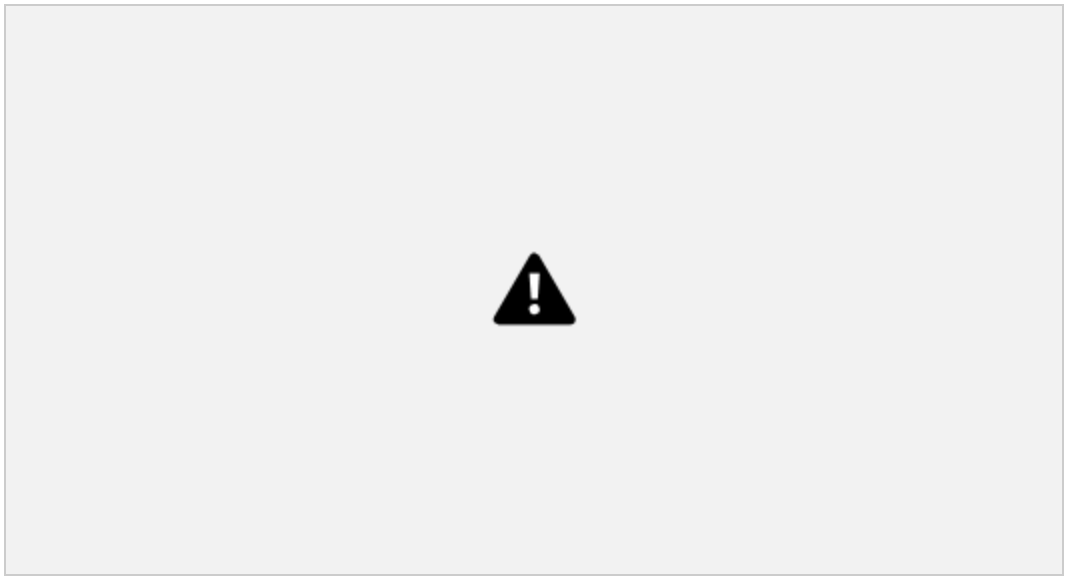
假设你调用greet("maggie")，计算机将首先为该函数调用分配一块内存。

我们来使用这些内存。变量name被设置为maggie，这需要存储到内存中。

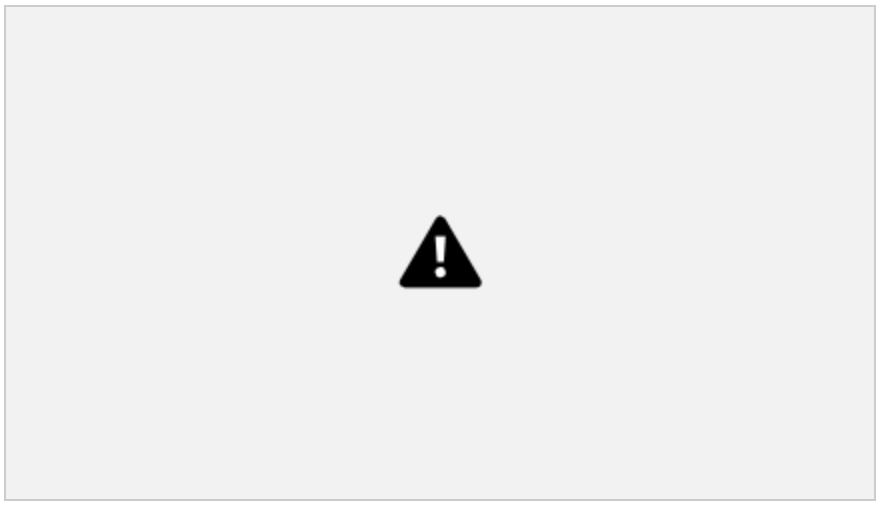


每当你调用函数时，计算机都像这样将函数调用涉及的所有变量的值存储到内存中。接下来，你打印hello，maggie!，再调 用greet2("maggie")。同样，计算机也为这个函数调用分配一块内 存。

计算机使用一个栈来表示这些内存块，其中第二个内存块位于第一个内存块上面。你打印how are you，maggie?，然后从函数调用返回。此时，栈顶的内存块被弹出。

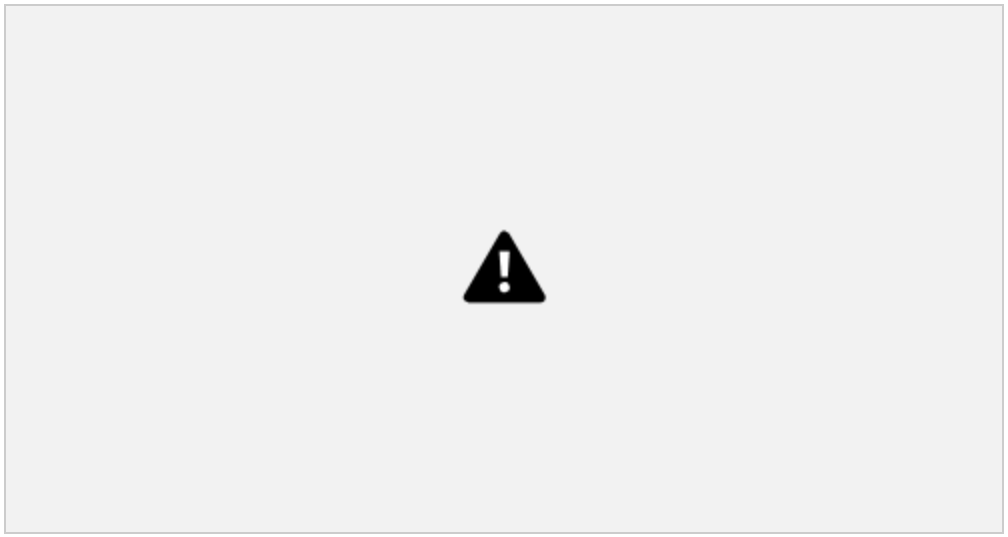


现在，栈顶的内存块是函数greet的，这意味着你返回到了函 数greet。当你调用函数greet2时，函数greet只执行了一部分。这是 本节的一个重要概念：调用另一个函数时，当前函数暂停并处于未完 成状态。该函数的所有变量的值都还在内存中。执行完函数greet2 后，你回到函数greet，并从离开的地方开始接着往下执行：首先打印getting ready to say bye...，再调用函数bye。



在栈顶添加了函数bye的内存块。然后，你打印ok bye!，并从这个函

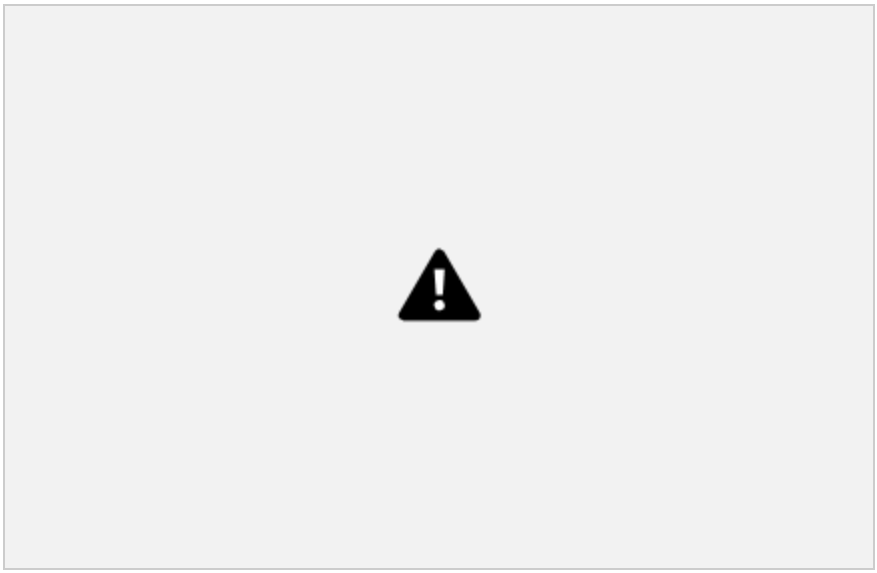
数返回。



现在你又回到了函数greet。由于没有别的事情要做，你就从函 数greet返回。这个栈用于存储多个函数的变量，被称为调用栈。

练习

3.1 根据下面的调用栈，你可获得哪些信息?



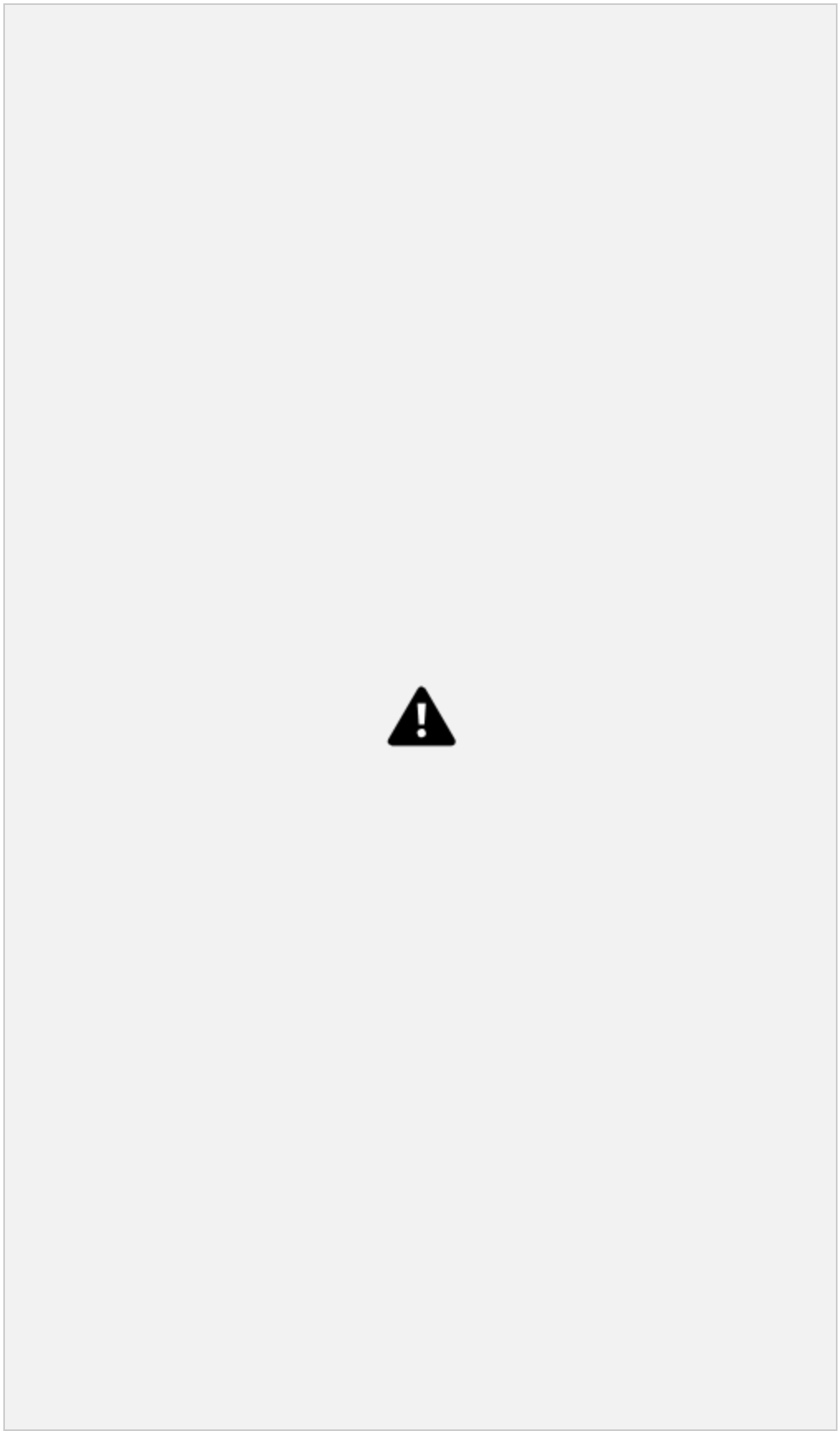
下面来看看递归函数的调用栈。

3.3.2 递归调用栈

递归函数也使用调用栈！来看看递归函数factorial的调用 栈。factorial(5)写作5!，其定义如下：5! = 5 * 4 * 3 * 2 * 1。同 理，factorial(3)为3 * 2 * 1。下面是计算阶乘的递归函数。

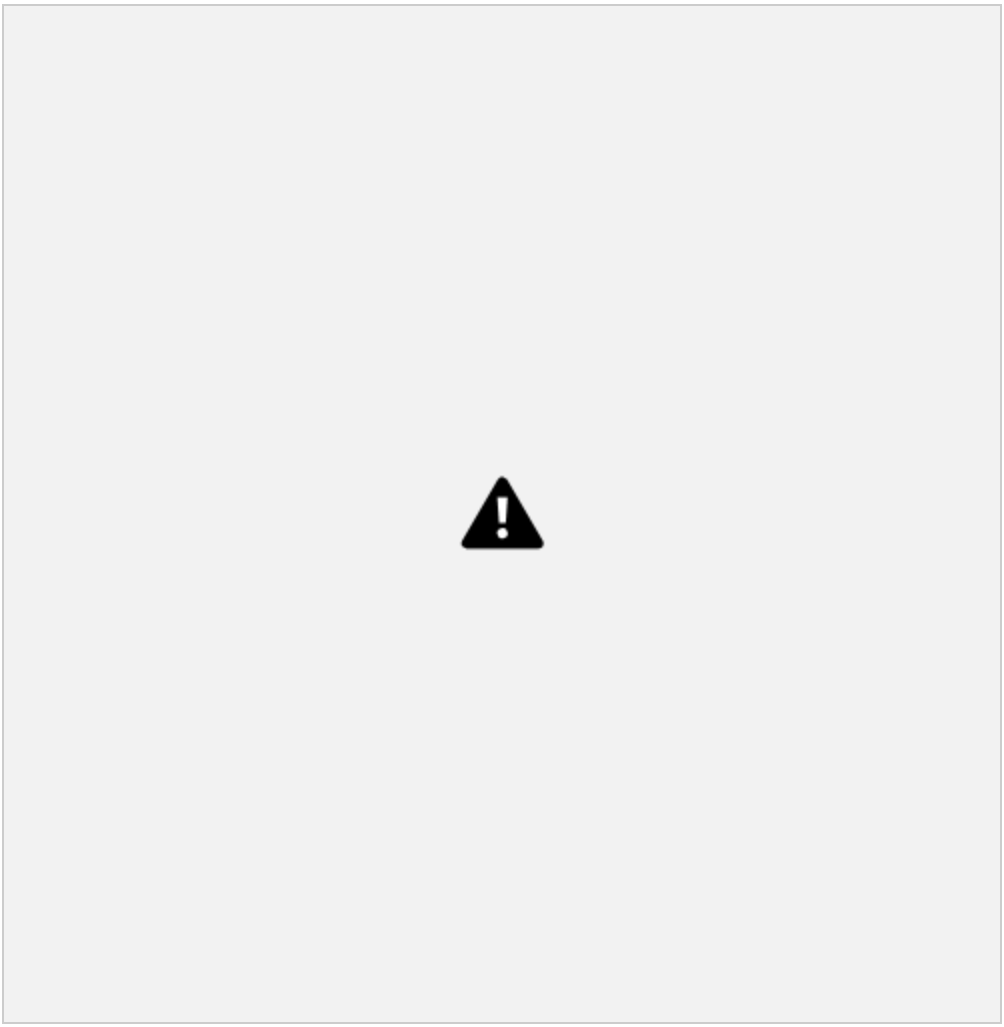
```
def fact(x):
 if x == 1:
 return 1
 else:
 return x * fact(x-1)
```

下面来详细分析调用fact(3)时调用栈是如何变化的。别忘了，栈顶的方框指出了当前执行到了什么地方。

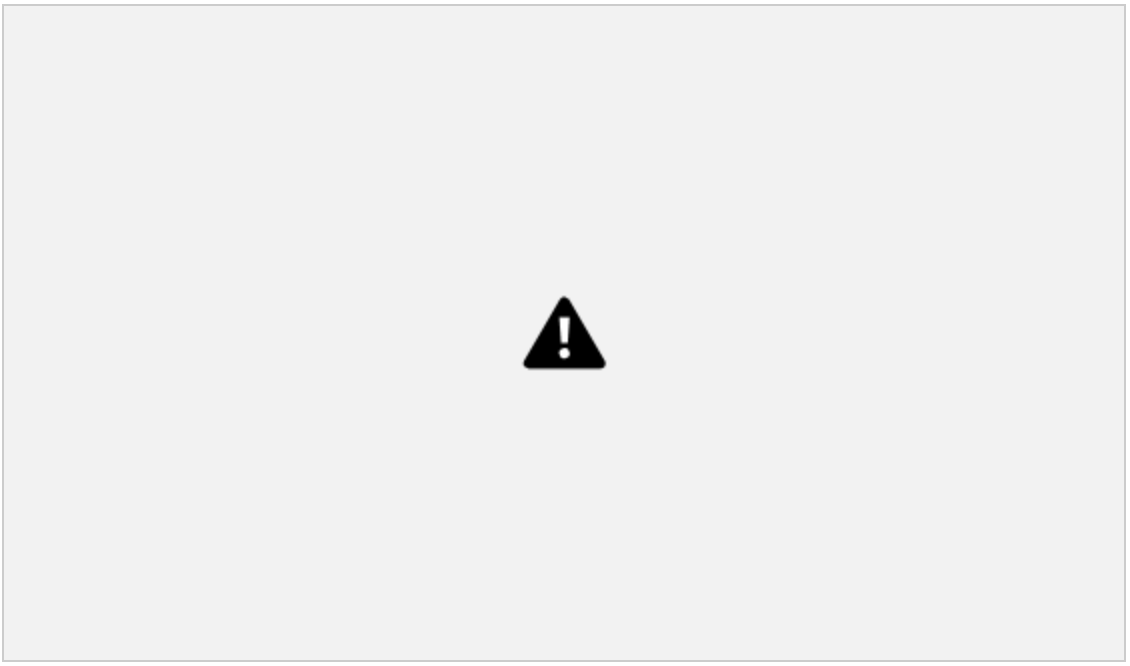


注意，每个fact调用都有自己的x变量。在一个函数调用中不能访问另一个的x变量。

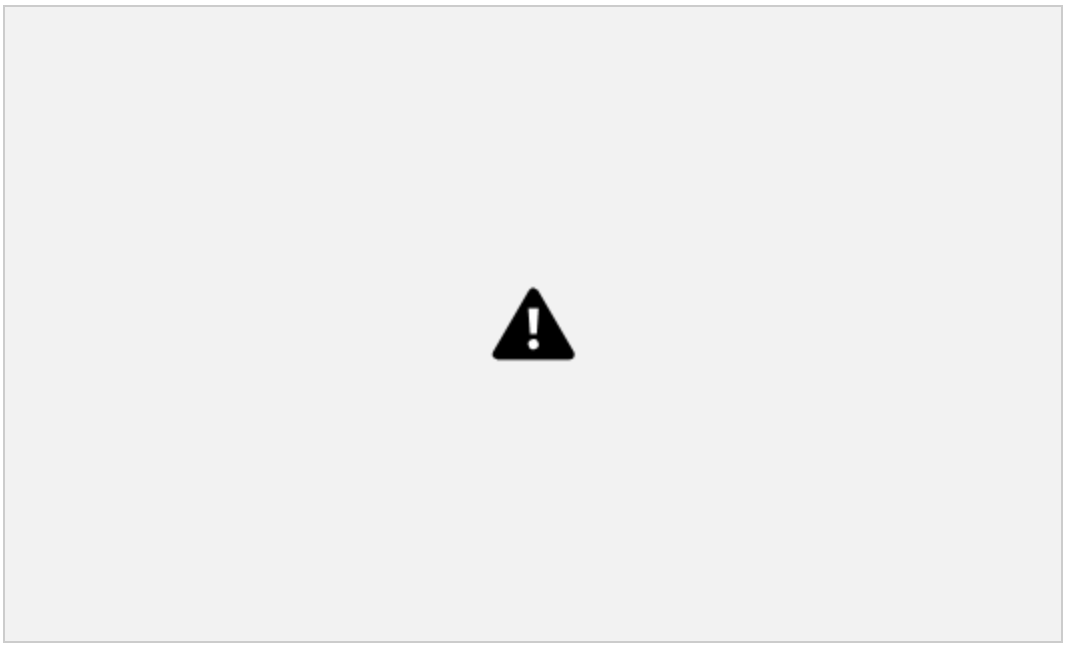
栈在递归中扮演着重要角色。在本章开头的示例中，有两种寻找钥匙的方法。下面再次列出了第一种方法。



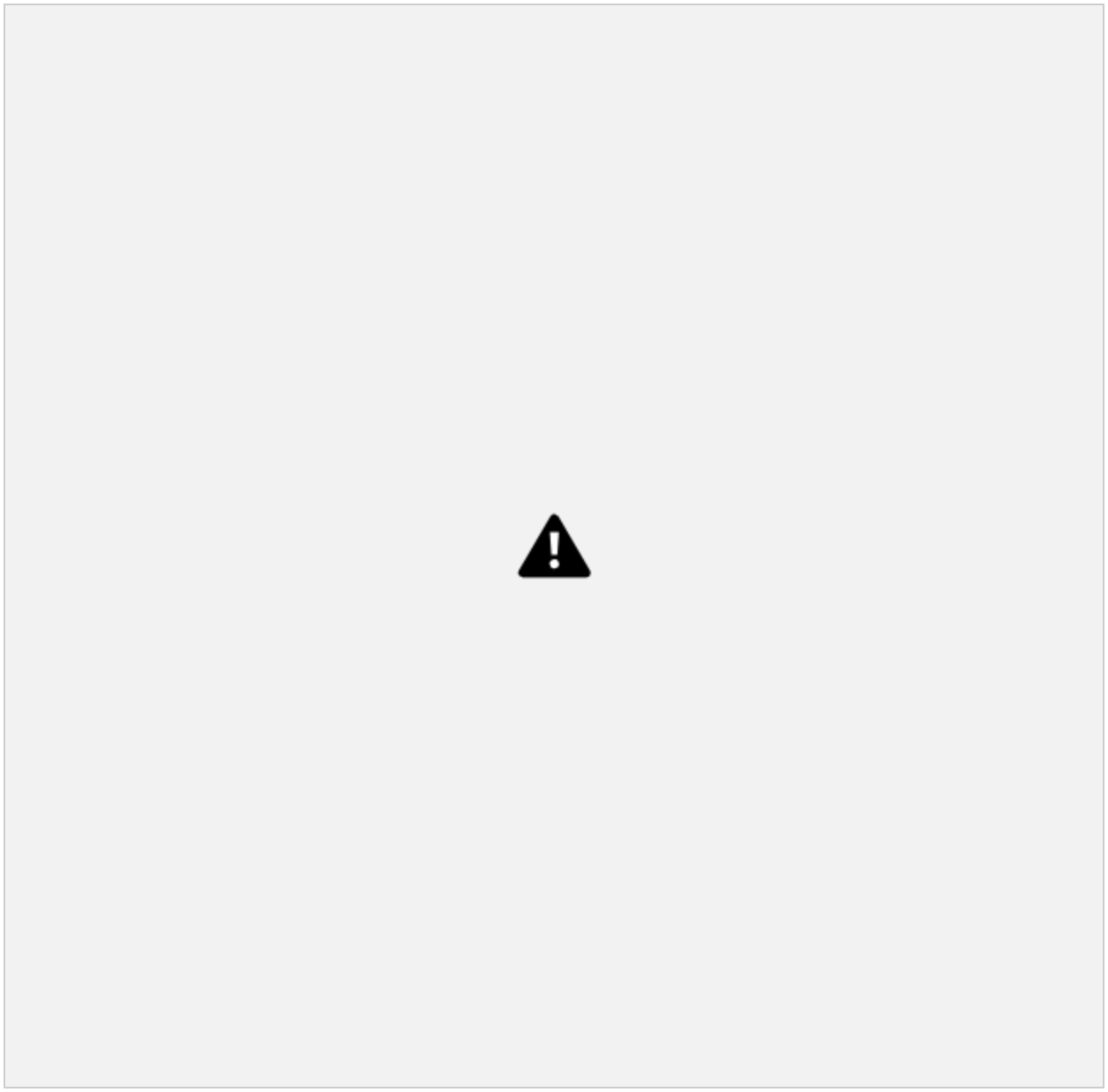
使用这种方法时，你创建一个待查找的盒子堆，因此你始终知道还有哪些盒子需要查找。



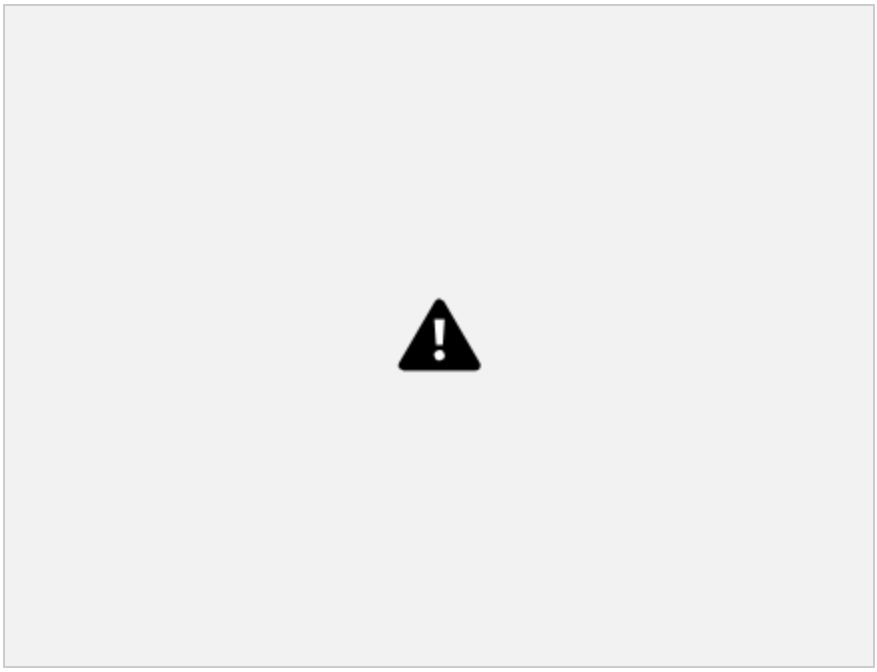
但使用递归方法时，没有盒子堆。



既然没有盒子堆，那算法怎么知道还有哪些盒子需要查找呢？下面是一个例子。



此时，调用栈类似于下面这样。



原来“盒子堆”存储在了栈中！这个栈包含未完成的函数调用，每个函数调用都包含还未检查完的盒子。使用栈很方便，因为你无需自己跟踪盒子堆——栈替你这样做了。

使用栈虽然很方便，但是也要付出代价：存储详尽的信息可能占用大量的内存。每个函数调用都要占用一定的内存，如果栈很高，就意味着计算机存储了大量函数调用的信息。在这种情况下，你有两种选择。

重新编写代码，转而使用循环。

使用尾递归。这是一个高级递归主题，不在本书的讨论范围内。另外，并非所有的语言都支持尾递归。

练习

3.2 假设你编写了一个递归函数，但不小心导致它没完没了地运行。正如你看到的，对于每次函数调用，计算机都将为其在栈中分配内存。递归函数没完没了地运行时，将给栈带来什么影响？

3.4 小结

递归指的是调用自己的函数。