移动开发经典丛书

一线Swift程序员必备

Classic Computer Science Problems in Swift

Swift 常用算法

经典计算机科学问题的Swift实现

[美] 大卫·科帕克 (David Kopec) 著

韩智文 张杰良 等译

MANNING

清华大学出版社

移动开发经典丛书

Swift 常用算法

经典计算机科学问题的 Swift 实现

[美] 大卫·科帕克(David Kopec) 著
韩智文 张杰良 等译

清华大学出版社
北 京

David Kopec
Classic Computer Science Problems in Swift
EISBN: 978-1-61729-489-1
Original English language edition published by Manning Publications, 178 South Hill Drive, Westampton, NJ 08060 USA. Copyright ©2018 by Manning Publications. Simplified Chinese-language edition copyright ©2019 by Tsinghua University Press. All rights reserved.

北京市版权局著作权合同登记号　图字：01-2018-3781

图书在版编目(CIP)数据

Swift 常用算法：经典计算机科学问题的 Swift 实现 / (美)大卫·科帕克(David Kopec) 著；韩智文 等译. —北京：清华大学出版社，2019
(移动开发经典丛书)
书名原文: Classic Computer Science Problems in Swift
ISBN 978-7-302-51709-2

Ⅰ. ①S…　Ⅱ. ①大… ②韩…　Ⅲ. ①程序语言－程序设计　Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2018)第 260086 号

责任编辑：王　军
封面设计：孔祥峰
版式设计：思创景点
责任校对：牛艳敏
责任印制：杨　艳

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　邮　　编：100084
社 总 机：010-62770175　　邮　　购：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者：三河市国英印务有限公司
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：13.25　　**字　　数**：245 千字
版　　次：2019 年 1 月第 1 版　　**印　　次**：2019 年 1 月第 1 次印刷
定　　价：98.00 元

产品编号：080320-01

献　　词

谨以此书纪念 Danny Kopec 博士，他在国际象棋、计算机科学和生活领域教导过成千上万的人。感谢 Jay Selman 博士，他是我亲爱的舅父，在我父亲过早去世后，他给了我无微不至的关怀和培养，这份爱，我将终身铭记！

致　　谢

感谢 Manning 出版团队让这本书顺利投向市场。特别要感谢策划编辑 Brian Sawyer，感谢他对我初次提议的信任，他接受了我提出的独特概念，并看到了“愿景”。也必须提到开发编辑 Jennifer Stout 在本书写作过程中给予的关照和理解。

MEAP 读者和官方评论人员的深入思考促使本书得到显著改进，这些人包括 Albert Choy、Alberto Chiesa、Arun Kumar、Becky Huett、Chad Johnston、Damian Esteban、Eric Giannini、Jeremy Gailor、Julien Pohie、Karolina Kafel、Laurence Giglio、Patrick Regan、Shawn Eion Smith 和 Tahir Akhtar。感谢所有在本书面世过程中提出建设性和具体批评意见的人！各位给出的反馈意见得到了认真倾听和采纳。

感谢我的家人、朋友和同事在本书撰写过程中给予我鼓励，特别是 Jay Selman 博士、Joshua Auerbach 博士和 Rebecca Driesen 博士，他们对一些篇章进行了勘校。感谢 Sylvia Kopec 和 Rebecca Driesen 帮助开发本书前几章中的图表。感谢我在纽约州立大学萨福克分校(SUNY Suffolk)和尚普兰学院(Champlain College)的学生，他们让我持续拥有作为一名教师的工作激情。

最后，对购买本书的读者致以最诚挚的谢意。在网络教程占据半壁江山的当今世界，仍然支持出版某个主题上的深入研究之作是很重要的。在线教程是极好的资源，但是读者朋友的购书之举使得完整的、经过审查和精心创作的书籍在计算机科学教育中仍将占有一席之地。

关于本书

编码约定和代码存储库

本书包含许多源代码示例，它们或者以独立形式存在，或者内嵌在正常文本中。在这两种情况下，源代码都采用等宽字体进行格式化，以便与普通文本区分开来。在某些地方，原始的源代码已经被重新格式化，添加了换行符并重新缩进，以适合书中有限的页面空间。代码行如果需要换行，通常使用行延续标记(➥)来表示。

本书代码的GitHub存储库的网址为https://github.com/davecom/ClassicComputerScienceProblemsInSwift。本书出版之际，包含源代码的zip文件也发布在出版商的网站上，网址为https://www.manning.com/books/classic-computer-science-problems-in-swift。

商标

本书中出现了一些商标名称，但并没有在每次出现商标名称时都标注商标符号。这些商标名称仅以编辑风格使用，以确保商标所有者的权益，并无意侵犯商标权。Apple、Xcode、Swift、Mac、iOS、iPhone和macOS都是苹果公司的注册商标。

读书论坛

购买本书后，读者能够免费访问由Manning出版社运营的一个私人网络论坛，在那里可以评论本书，提出技术问题，并从作者和其他用户那里得到帮助。要访问该论坛，请访问 https://forums.manning.com/forums/classic-computer-science-problems-in-swift。也可以在 https://forums.manning.com/forums/about 上了解更多

关于 Manning 出版社的论坛和行为准则。Manning 出版社向读者承诺提供一个空间，让读者之间以及读者与作者之间能够进行有意义的对话。本书作者不会就具体参与的程度做出承诺，他对论坛所做的贡献基于自愿(而且是无偿的)。我们建议读者尝试向作者提出具有挑战性的问题，以便激发他的兴趣！只要这本书仍在市面上，该论坛以及之前讨论的存档文件都可以在出版商的网站上找到。

关于作者

David Kopec 是佛蒙特州伯灵顿的尚普兰学院计算机科学与创新专业的副教授。他是一名经验丰富的 iOS 开发人员，也是 *Dart for Absolute Beginners*(Apress 出版社，2014 年出版)一书的作者。David 拥有达特茅斯学院的经济学学士学位和计算机科学硕士学位。

前　言

感谢购买本书。Swift 正处于一个激动人心的发展阶段。随着这门语言的日趋稳定和普及，有必要让传统的计算机科学教育内容接触到这门语言。这本中级编程书籍所涉及的问题将有助于经验丰富的程序员深入学习 Swift 语言，也能帮助新手程序员加快自己的计算机技术学习进度。本书涵盖大量用以解决问题的技术，每个人都能够从中找到自己所需。

本书不是对 Swift 语言的基本介绍。苹果公司出版了一本优秀的免费书籍来满足入门需求[1]。相反，本书假设读者已经掌握 Swift 语法的基本知识，但不要求精通 Swift。事实上，本书的内容基于以下假设：可以作为学习材料来帮助人们精通 Swift。另一方面，本书不适合初学者使用。

为什么选择 Swift

苹果公司的 Swift 是一门令人兴奋的新型编程语言，它在面向对象和函数范式之间建立联系。Swift 的创造者在这两个领域之间取得了惊人的平衡，许多人甚至认为达到了最佳效果。Swift 通过苹果公司的开发工具获得了广泛部署，它符合现代的语法规范，融合了其他语言的优秀特性，拥有精心设计的范式平衡，具有作为 iOS 和 Mac 应用主要开发语言的发展前景。以上种种因素，决定了现在是学习 Swift 的好时机。

苹果公司宣称 Swift 是第一门面向协议的语言，这是由于其强大的协议特性集合以及该集合在其标准库中得到广泛使用[2]。然而，许多长期从事 Objective-C 和 Java 开发的人缺乏函数式编程经验，更不用说面向协议的编程技术。与此同时，一些函数式程序员进入 Swift 社区，他们尝试按照自己在 Haskell 或 Scheme 中的编程方式来解决问题，有时缺少更优雅的、面向对象的解决方案。

---

1 苹果公司，*The Swift Programming Language*，http://mng.bz/6fKi。

2 戴夫 •亚伯拉罕，“Swift 的面向协议编程”(WWDC 2015，第 408 节，苹果公司)，http://mng.bz/zWP3。

本书的目的是通过使用Swift解决资深程序员(以及新程序员)应该熟悉的经典问题，而不是固守单一的语法范式，充当这些领域之间的桥梁。因而，读者对于各个领域的内容都会有所体验。不同领域的结合是掌握 Swift 的正确途径，而搭建桥梁则是社区前进的方式。

什么是经典的计算机科学问题

有人说计算机对于计算机科学而言，就像望远镜对于天文学一样。如果是这样的话，那么编程语言就像望远镜的镜头吗？不管怎样比拟，本书中“计算机科学问题”这个术语是指“计算机科学本科课程中的典型编程问题”。

一些提供给新程序员解决的编程问题，无论是在攻读(计算机科学、软件工程等)学士学位的课堂环境里，还是在中级编程教材(例如，关于人工智能或算法的入门教材)里涉及，这些问题司空见惯，足以被视为“经典”。本书就是这些问题的汇聚。

这包括通过数行代码即可解决的琐碎问题，乃至需要跨越多章内容构建系统的复杂问题。有些问题涉及人工智能，还有一些问题只需要常识。有些问题存在于现实世界之中，而有些问题则是凭空想象出来的。

本书涵盖的问题类型

第 1 章介绍多数读者可能都熟悉的问题解决技巧，诸如递归、记忆法/计算缓存和模拟等内容是后面各章要探讨的其他技术的基本构建模块。

第 2 章重点说明搜索问题。搜索主题涵盖广泛，甚至可以将本书中的大部分问题都置于其范畴之内。第 2 章介绍最基本的搜索算法，包括二分搜索、深度优先搜索、广度优先搜索和 A*搜索。这些算法在本书的其余部分都会重复用到。

第 3 章构建一类广泛问题的解决框架，这类问题由具有约束条件且具有有限定义域的变量加以抽象定义，包括经典的“八皇后问题”“澳大利亚地图着色问题”和密码算法“SEND+MORE=MONEY”。

第 4 章探索图算法。对于门外汉来说，这类算法的适用性极为广泛。本章将构建图的数据结构，然后用它来解决多个经典的优化问题。

第 5 章探索遗传算法，这种技术与书中介绍的大多数技术相比具有更多的不确定性，但有时可以解决传统算法无法在合理的时间内解决的问题。

第 6 章涵盖 k-均值聚类，这或许是本书中在算法方面涉及最具体的一章。这

种聚类技术实现简单，易于理解，具有广泛的适用性。

第 7 章阐述神经网络的概念，并让读者对于极简单的神经网络有所感受。本章并不寻求能够全面涵盖这个令人兴奋、不断发展的领域。

最后，第 8 章讨论不适于编入前面各章的一些值得关注的有趣问题。

本书的目标读者

本书既适合中级程序员，也适合经验丰富的资深程序员。想学习 Swift 的专业程序员会从本书中发现他们在计算机科学或编程教育中熟知的问题。编程新手将通过自己选定的 Swift 语言接触到这些经典的问题。准备接受编程面试的开发人员可能会发现本书是很有价值的备用材料。

除了专业程序员之外，对 Swift 感兴趣的计算机科学专业大学生也可能会发现本书很有帮助。它没有试图严谨地介绍数据结构和算法——它并非数据结构和算法教科书——书中没有 big-O 符号的证明或广泛运用。相反，本书被定位为易于使用、可亲手实践解决问题的技术教程，而这些技术应该是数据结构、算法和人工智能课程的最终产物。

再次重申，本书假定读者具备 Swift 语法和语义的基本知识。没有编程经验的读者很难从本书中获益。没有接触过 Swift 的程序员也肯定会大费周章。换句话说，本书可以称为又一部关于 Swift 的极好书籍。

Swift 的版本控制和工具

本书中的源代码遵循 Swift 4.1 版本规范。该版本与 Xcode 9.3 一起由苹果公司于 2018 年初发布。书中代码的 GitHub 库的网址是 https://github.com/davecom/ClassicComputerScienceProblemsInSwift。

本书中的大部分源代码能够原封不动地在 Linux(以及移植了 Swift 的其他平台)上运行，因为它们只依赖于 Foundation(而不是 AppKit/UIKit)。源代码文件作为 Xcode 的 Swift 基础组成部分进行发布，但是其中包含的原始.swift 文件可以提取出来在 Linux 上使用。跨平台兼容性是本书的目标之一，但对大多数读者来说，Mac 之上的便利性是更大的目标。

本书没有说明如何使用 Xcode、构建 Swift 项目或使用 Playground。关于这些主题，网上和纸质出版物中都有很多优质资源。本书假定读者已具备完成这些任务的能力。

没有图形，也没有用户界面代码

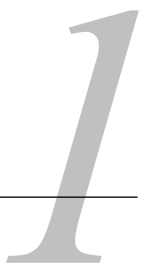
本书并不讲解 UIKit 或 AppKit，书中的例子也不需要用到它们。本书中的示例不会产生图形化结果，这是因为我们希望能够使用尽可能简明易读的解决方案来解决书中提出的问题。很多时候，处理图形会造成阻碍，或者会极大增加各种解决方案在阐释技术或算法时的复杂性。

此外，为了在 Linux 上实现 Swift 的跨平台兼容性，也不能使用 UIKit 和 AppKit。在本书撰写之际，只有 Foundation 被移植到 Linux 上。书中给出的解决方案很大程度上只依赖于 Swift 标准库，在标准库功能薄弱的领域则将 Foundation 作为补充。

本书并不讲授如何编写完整的应用程序，但会介绍使用 Swift 进行软件开发的基础知识。书中的内容仅限于本书主题范围。

目　录

第 1 章 小型问题



首先，我们探讨一些简单问题，这些问题只需要几个较简短的函数就能够解决。虽然这些问题很小，但它们仍然能够为我们提供一些解决问题的技巧。就把这些简单的问题作为一次很好的热身吧！

1.1 斐波那契数列

斐波那契数列是一系列数字，除第一个和第二个数字外，任何数字都是前两个数字之和：

```
0、1、1、2、3、5、8、13、21、…
```

数列中第一个斐波那契数的值为 0，第四个斐波那契数的值为 2。数列中第 n 个斐波那契数的值可以通过下述公式计算：

```
fib(n) = fib(n - 1) + fib(n - 2)
```

1.1.1 尝试递归方法

上述用于计算斐波那契数列中数字的公式(如图 1-1 所示)采用的是伪代码形式，可以非常方便地转换为 Swift 递归(recursive)函数(递归函数是一种调用自身的函数)。这种机械式翻译就作为我们尝试编写的斐波那契数列函数的第一版，该函数返回指定的斐波那契数列的值：

```
func fib1(n: UInt) -> UInt {
    return fib1(n: n - 1) + fib1(n: n - 2)
}
```

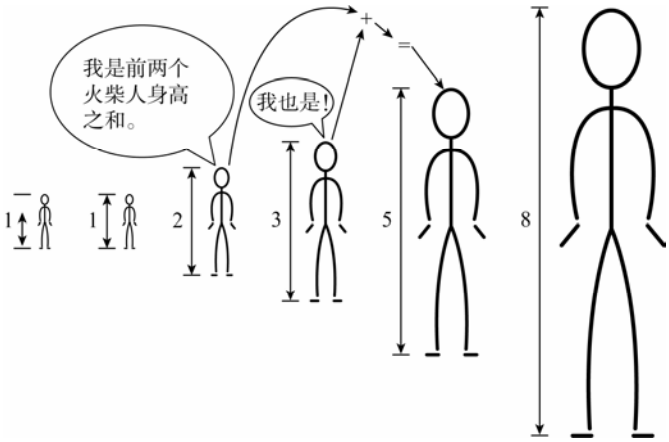


图 1-1 每个火柴人的身高是前两个火柴人身高之和

注意：将 fib1()函数的数据类型定义为 UInt 而不是 Int，这是因为斐波那契数列中没有负整数。

如果使用一个值来调用运行这个函数，则这个函数将会永远不停地运行，不会返回最终结果，如图 1-2 所示，我们称这种情况为无穷递归(infinite recursion)，无穷递归和无穷循环(infinite loop)类似。

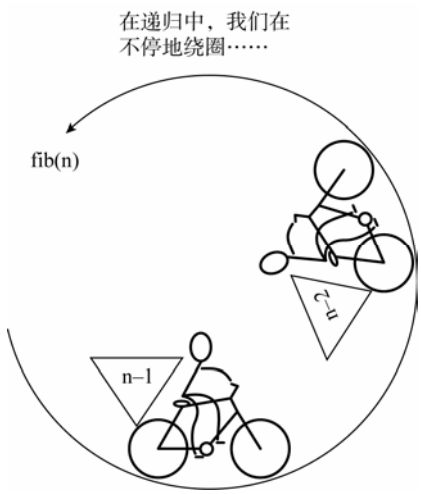


图 1-2 递归函数 fib(n)通过参数 n−2 和 n−1 调用自身

1.1.2 利用基本情形

这里需要注意的是，对于上述斐波那契函数 fib1()，Xcode 并不会报错。避免代码出现无穷递归是程序员的责任。无穷递归发生的原因是我们从未指定基本情

形(base case)。在递归函数中，基本情形用作终止点(stopping point)。

对于斐波那契函数而言，我们拥有天然的基本情形，就是指定前两个特殊的数列值为 0 和 1。0 和 1 都不是数列中前两个数字的和，相反，它们是特殊的前两个值。下面就把这两个值指定为基本情形：

```
func fib2(n: UInt) -> UInt {
    if (n < 2) { // base cases
        return n
    }
    return fib2(n: n - 2) + fib2(n: n - 1) // recursive cases
}
```

注意：fib2()版本的斐波那契函数将返回的 0 作为第零个数(fib2(n:0))，而不是第一个数，这符合我们的最初设想。在编程上下文中，这种处理是有意义的，因为数列(如 Swift 的 Array 类型)通常从零元素开始。

fib2()函数可以成功调用，并返回正确的结果。下面用一些较小的数值来尝试调用 fib2()函数：

```
fib2(n: 5)
fib2(n: 10)
```

现在还不要尝试调用 fib2(n:50)，如果尝试调用 fib2(n:50)的话，那么 fib2()函数就永远执行不完！这是什么原因呢？原因在于 fib2()需要递归调用 fib2(n:n−1)和 fib2(n:n−2)，所以每次对 fib2()的调用都会导致额外增加两次对 fib2()的调用(见图1-3)。换句话说，调用次数是呈指数级增长的。例如，调用 fib2(n:4)的整个调用集包括：

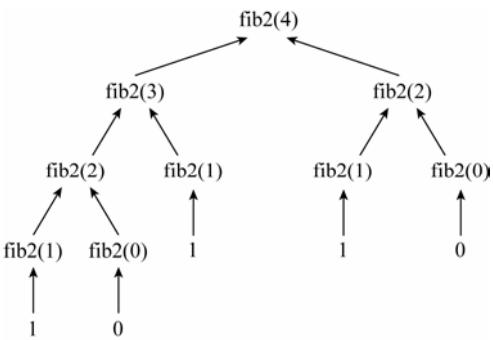


图 1-3　每次对 fib2()的非基本情形调用都会导致再多调用两次 fib2()

```
fib2(n: 4) -> fib2(n: 3), fib2(n: 2)
fib2(n: 3) -> fib2(n: 2), fib2(n: 1)
fib2(n: 2) -> fib2(n: 1), fib2(n: 0)
```

```
fib2(n: 2) -> fib2(n: 1), fib2(n: 0)
fib2(n: 1) -> 1
fib2(n: 1) -> 1
fib2(n: 1) -> 1
fib2(n: 0) -> 0
fib2(n: 0) -> 0
```

如果统计调用次数(就如在 Xcode 环境中调用 fib2(n:4))，那么仅仅为了计算数列中的第 4 个元素就总共对 fib2()调用了 9 次！如果要计算更大的元素，情况会变得更糟。计算第 5 个元素需要 15 次调用，计算第 10 个元素需要 177 次调用，而计算第 20 个元素则需要 21 891 次调用。不过我们还可以对算法进行完善。

1.1.3 计算缓存技术

计算缓存/记忆法(memoization)技术指的是在计算任务完成时存储计算任务的结果，以便再次需要计算结果时可以直接查看结果，而不是再次(或者百万次)重复进行计算(见图 1-4)[1]。

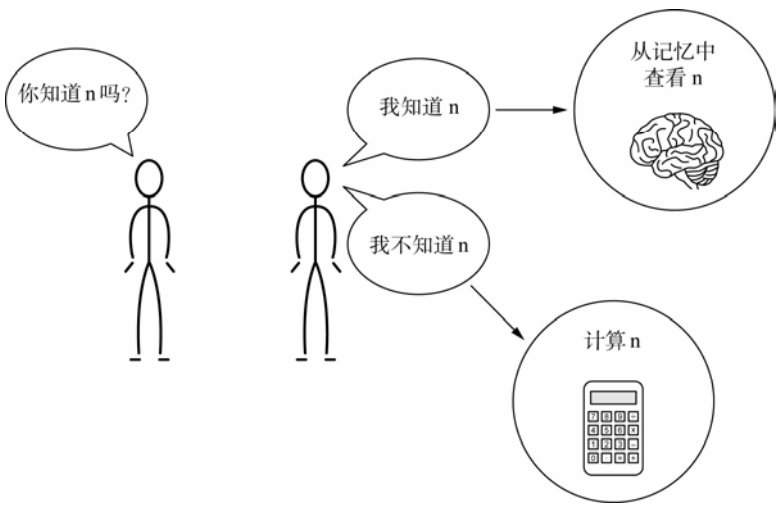


图 1-4 人类记忆机器

接下来我们创建一个新版本的斐波那契函数，该函数使用 Swift 的 Dictionary 来实现计算缓存目的。

```
var fibMemo: [UInt: UInt] = [0: 0, 1: 1] // our old base cases
func fib3(n: UInt) -> UInt {
```

1 Donald Michie(英语著名计算机科学家)创造了术语“计算缓存”，参见 Donald Michie 发表的文章 *Memo functions:a language feature with “rote-learning” properties*(Edinburgh 大学，机器智能与感知系，1967 年)。

```
    if let result = fibMemo[n] { // our new base case
        return result
    } else {
        fibMemo[n] = fib3(n: n - 1) + fib3(n: n - 2) // memoization
    }
    return fibMemo[n]!
}
```

警告：使用!来强制解包可选项的方式并不推荐，但这里使用!是为了方便，因为可以证明在调用最后的 return 语句时 fibMemo 中已经包含结果。

现在就可以安全地调用 fib3(n:50)。调用 fib3(n:20)只需要调用 39 次 fib3()，作为对比，调用 fib2(n:20)需要调用 21 891 次 fib2()。fibMemo 中预先填充了初期的基本情形 0 和 1，使 fib3()避开了另一个 if 语句的复杂性。

1.1.4　保持斐波那契简单

解决斐波那契数列问题还有性能更高的方法，这个方法就是老式的迭代方法。

```
func fib4(n: UInt) -> UInt {
    if (n == 0) { // special case
        return n
    }
    var last: UInt = 0, next: UInt = 1 // initially set to fib(0) & fib(1)
    for _ in 1..<n {
        (last, next) = (next, last + next)
    }
    return next
}
```

警告：fib4()中 for 循环的主体使用了元组(tuple)，或许这种方法有点显得过于聪明了。有些人可能会觉得这段代码为了简洁性而牺牲了可读性，有些人可能会发现代码本身的简洁性使代码更可读。这段代码的要点是，last 被设置为 next 的前一个值，而 next 被设置为 last 的前一个值加上 next 的前一个值，这样就可以避免创建一个临时变量来保存 last 更新后的 next 的旧值(但要在 next 更新前)。

通过这种方法，for 循环的主体最多只需要运行 n−1 次。换句话说，这是斐波那契数列的最高效版本。对于第 20 个斐波那契数而言，该函数只需要运行 19 次 for 循环主体；而如果采用 fib2()实现方法的话，则需要递归调用 fib2() 21 891 次。在现实世界的应用程序中这会产生明显的差异！

在递归解决方法中，我们采用倒向计算；而在迭代解决方法中，我们采用正向计算。有时递归是解决问题最直观的方法，例如 fib1()和 fib2()的实现几乎就是斐波那契原始公式的机械式翻译。然而，简单质朴的递归解决方法会带来显著的性能开销。请记住，任何可用递归解决的问题也可用迭代来解决。

1.2 简单数据压缩

不管是虚拟空间还是真实空间，节省空间通常都很重要。使用的空间越少，效率就越高，也越节省资金。在租用公寓时，如果公寓的空间超出自己和家庭的需求，就会寻找小的地方，这样价格也会便宜。如果在服务器存储数据时需要按字节付费，就可能希望压缩数据，从而降低存储成本。压缩(compression)是一种对数据进行编码、改变数据形式，从而节省数据占用空间的行为。解压缩(decompression)是压缩的逆过程，解压缩将数据还原成原始形式。

如果压缩数据的存储效率更高，那么为什么不对所有的数据都进行压缩呢？压缩数据需要在时间和空间之间进行权衡，压缩一段数据并将其解压缩回原始形式需要时间。因此，只有在数据规模优先于快速执行的情况下压缩数据才有意义。通过互联网传输大文件就是这种情况，因为传输文件所需的时间要比接收到文件后对文件进行解压缩需要的时间长，如此压缩这些文件才有意义。此外，在原始服务器上因存储空间压缩文件花费的时间只需要考虑文件压缩一次的时间。

压缩数据最简单的方法就是要意识到这一现实情况，即数据存储类型所使用的位数要比其内容严格要求的位数多。例如，如果将一个永远不可能超过 65 535 的无符号整数存储为 UInt 类型(在大多数 Swift 平台上为 64 位无符号整数)，那么存储效率就不可能高效。相反，可以把这个无符号整数存储为 UInt16 类型(16 位无符号整数)，这样就可以把实际数字消耗的空间减少 75%(用 16 位取代 64 位)。如果有数百万个这样的数字被低效存储，那么浪费的空间总计可能达到数百万字节。

注意：如果对二进制还不太了解的话，请记住二进制中的一位就表示一个值，不是 1 就是 0。一连串的 1 和 0 以二进制形式表示一个数字。就本节而言，并不需要进行二进制计算，只需要了解一种类型存储的位数决定这种类型可以表示多少个不同的数值。例如，1 位可以表示 2 个值(0 或 1)，2 位可以表示 4 个值(00、01、10、11)，3 位可以表示 8 个值，依此类推。

如果某种类型打算表示的不同可能值的数量，小于用于存储它的位数所能

表示的值的数量，则存储效率有可能得以提高。下面我们以 DNA 中形成基因的核苷酸的存储为例进行说明[1]，每个核苷酸只能有 4 个可能的取值：A、C、G 或 T(第 2 章将深入介绍)。然而，如果将基因存储为 String(可认为是各种字符的集合)，那么每个核苷酸可由一个字符表示，一个字符通常需要 8 位存储。在二进制中，只需要 2 位就可以存储具有 4 个可能值的类型：00、01、10 和 11。2 位就可以表示这 4 个不同的值。如果为 A 赋值 00、为 C 赋值 01、为 G 赋值 10 以及为 T 赋值 11，那么一串核苷酸信息所需的存储空间就可以减少 75%(每个核苷酸由 8 位减少到 2 位)。

接下来我们将核苷酸存储为位串(bit string)，而不再存储为 String(见图 1-5)。顾名思义，位串就是一个由 1 和 0 组成的任意长度的序列。然而，Swift 标准库没有包含任何现成的构造用于处理任意长度的位串，但 Swift 可以利用底层的 C 库 Core Foundation，Core Foundation 包含 CFMutableBitVector。下述代码实现了由 A、C、G 和 T 组成的 String 和 CFMutableBitVector 之间的相互转换。

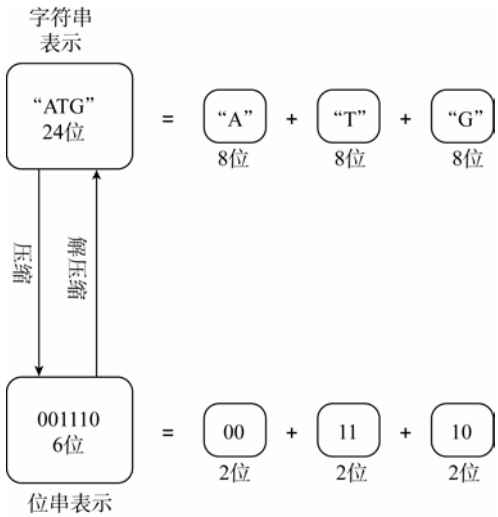


图 1-5　将表示基因的 String 压缩成每核苷酸 2 位的位串

```
struct CompressedGene {
    let length: Int
    private let bitVector: CFMutableBitVector

    init(original: String) {
        length = original.count
        // default allocator, need 2 * length number of bits
```

1 本示例的灵感源于 Robert Sedgewick 和 Kevin Wayne 合著的 *Algorithms 4th Edition* 一书(Addison-Wesley Professional 出版社，2011 年)，第 819 页。

```
            bitVector = CFBitVectorCreateMutable(kCFAllocatorDefault, length * 2)
            CFBitVectorSetCount(bitVector, length * 2) // fills the bit vector
            ➥ with 0s
            compress(gene: original)
        }
    }
```

CompressedGene 内部将核苷酸序列存储为位串。init()方法的主要职责就是初始化位串结构 CFMutableBitVector，并调用 compress()来完成将所提供的核苷酸 String 实际转换为位串的繁重工作。CFBitVectorCreateMutable()需要分配器和容量，容量等于 length*2，这是因为每个核苷酸需要 2 位。CFMutableBitVector 的大小和容量容易令人迷惑，实际上 CFMutableBitVector 的大小是指其中到底有多少位，而 CFMutableBitVector 的容量是指其中可以有多少位，两者是不同的。CFBitVectorSetCount()设置位向量的大小并将所有位初始化为 0。

接下来，我们演示如何实际执行压缩。

提示：诸如 CFMutableBitVector 的 Core Foundation 构造都是通过采用可移植的 C 语言实现的，在 Linux 平台上的 Swift 中可以使用。在 Linux 平台上，需要使用 import CoreFoundation 包含此类构造；而在 macOS 平台上，需要使用 import Foundation 隐式包含此类构造。

```
private func compress(gene: String) {
    for (index, nucleotide) in gene.uppercased().enumerated() {
        let nStart = index * 2 // start of each new nucleotide
        switch nucleotide {
        case "A": // 00
            CFBitVectorSetBitAtIndex(bitVector, nStart, 0)
            CFBitVectorSetBitAtIndex(bitVector, nStart + 1, 0)
        case "C": // 01
            CFBitVectorSetBitAtIndex(bitVector, nStart, 0)
            CFBitVectorSetBitAtIndex(bitVector, nStart + 1, 1)
        case "G": // 10
            CFBitVectorSetBitAtIndex(bitVector, nStart, 1)
            CFBitVectorSetBitAtIndex(bitVector, nStart + 1, 0)
        case "T": // 11
            CFBitVectorSetBitAtIndex(bitVector, nStart, 1)
            CFBitVectorSetBitAtIndex(bitVector, nStart + 1, 1)
        default:
            print("Unexpected character \(nucleotide) at \(index)")
        }
    }
}
```

compress()方法依次查看核苷酸 String 中的每个 Character，当 compress()方法查看到 A 时，它就在位串中添加 00；当 compress()方法查看到 C 时，它就在位串中添加 01，依此类推。记住每个核苷酸需要 2 位空间，因此初始 String 中每个 Character 的索引都被乘以 2，从而能够查找到位串中每个核苷酸的起始位置。

接下来，我们实现解压缩功能。

```
func decompress() -> String {
    var gene: String = ""
    for index in 0..<length {
      let nStart = index * 2 // start of each nucleotide
      let firstBit = CFBitVectorGetBitAtIndex(bitVector, nStart)
      let secondBit = CFBitVectorGetBitAtIndex(bitVector, nStart + 1)
      switch (firstBit, secondBit) {
      case (0, 0): // 00 A
          gene += "A"
      case (0, 1): // 01 C
          gene += "C"
      case (1, 0): // 10 G
          gene += "G"
      case (1, 1): // 11 T
          gene += "T"
      default:
          break // unreachable, but need default
      }
    }
    return gene
}
```

最后，decompress()每次从位串中读取 2 位，并将这些位组装到元组中，元组使用 Swift 内置的 switch 模式匹配语句进行求值，原始 String 得以重新组装并返回，直至循环完成。下面我们测试一下结果。

```
print(CompressedGene(original: "ATGAATGCC").decompress())
```

在完成压缩/解压缩循环后，控制台上应该出现原始 String。

1.3　牢不可破的加密

一次性密码本(one-time pad)是一种加密数据块的方式，该方式将数据块与无意义的随机虚拟数据(dummy data)组合在一起，在没有同时得到加密数据和虚拟数据的情况下就不能重构原始数据。实质上，加密器拥有一对密钥(其中一个密钥是加密数据，另一个密钥是随机虚拟数据)。只得到一个密钥没有任何作用——只

有组合使用两个密钥才能解锁原始数据。正确执行时，一次性密码本是一种牢不可破的加密形式，图 1-6 演示了这个过程。

```
Original Data + Dummy Data -Encryption> Key-Pair (Dummy Data, Product)
➥ -Decryption> Original Data
```

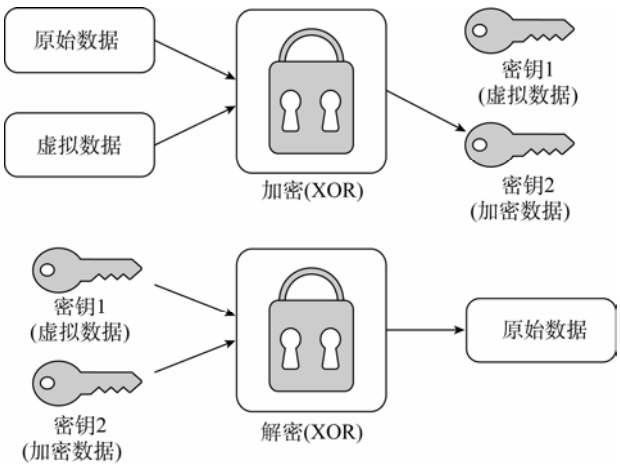


图 1-6 一次性密码本产生两个单独的密钥，将密钥组合在一起可以恢复原始数据

1.3.1 数据排序

在下面的例子中，我们将使用一次性密码本来加密 String。Swift 语言的 String 可以看作一系列的 UTF-8 字节(使用 UTF-8 作为 Unicode 字符编码)。String 通过 utf8 实例变量定义自己为一串 UTF-8 字节的视图。这个“视图”实际上是一串 UInt8，每个 UTF-8 字节由一个 UInt8 表示。因此，我们可以为一次性密码本和密钥对定义一种类型。

```
typealias OTPKey = [UInt8]
typealias OTPKeyPair = (key1: OTPKey, key2: OTPKey)
```

如果得到的加密数据牢不可破的话，一次性密码本加密操作中使用的虚拟数据必须满足三个条件，即虚拟数据必须与原始数据的长度相同、真正随机并且完全保密。第一个和第三个条件符合常理。如果虚拟数据太短，那么虚拟数据就会重复，就有可能观察出加密模式。如果其中一个密钥不是真正保密(也许在别处被重新使用或部分泄露)，那么攻击者就可以在其中找到线索。第二个条件本身就提出了一个问题——我们能否产生真正的随机数据？对于大多数计算机而言，答案是否定的。

在本例中，我们使用伪随机数生成函数 arc4random_uniform()，因此我们的数

据并不是真正的随机数(但对于我们的目的而言已经足够接近随机数)。下面我们生成一个随机数 OTPKey，用作虚拟数据。

```
func randomOTPKey(length: Int) -> OTPKey {
    var randomKey: OTPKey = OTPKey()
    for _ in 0..<length {
        let randomKeyPoint = UInt8(arc4random_uniform(UInt32(UInt8.max)))
        randomKey.append(randomKeyPoint)
    }
    return randomKey
}
```

函数 randomOTPKey()创建一个 OTPKey(UInt8 数组)，并用 length 个随机数进行填充，随机数使用 UInt8 最大值的上限来生成。由于 arc4random_uniform()的要求和输出以及我们对 UInt8 的需求，在各种整数类型之间有一些令人讨厌的转换。换句话说，randomOTPKey()的最终输出是一系列 UInt8 随机数。

1.3.2　加密和解密

虚拟数据如何与我们希望加密的原始数据结合在一起呢？答案是采用 XOR(异或)运算。XOR 运算是一种按位逻辑运算(在位级运算)，当其操作数中只有一个为真时返回真，但当两个操作数都为真或都不为真时返回假。正如大家所猜测的，XOR 代表异或运算。

在 Swift 语言中，XOR 运算符为^。在二进制数字的位上下文中，对于 0^1 和 1^0 运算，XOR 返回 1；而对于 0^0 和 1^1 运算，XOR 返回 0。对于 XOR 运算而言，有一个非常有价值的属性，就是如果两个数字的各位都使用 XOR 运算组合，那么得到的加密数据可以使用其中任何一个操作数进行重新组合，生成另一个操作数。

```
A ^ B = C
C ^ B = A
C ^ A = B
```

这就是一次性密码本加密基础的关键所在。为了形成加密数据，我们简单地将原始 String 中的每个 UInt8 与虚拟数据中的每个 UInt8 进行 XOR 运算，返回的密钥对为虚拟数据和加密数据。

```
func encryptOTP(original: String) -> OTPKeyPair {
    let dummy = randomOTPKey(length: original.utf8.count)
    let encrypted: OTPKey = dummy.enumerated().map { i, e in
        return e ^ original.utf8[original.utf8.index(original.utf8
```

```
            ➥ .startIndex, offsetBy: i)]
        }
        return (dummy, encrypted)
    }
```

注意：这里在 map()之前使用 enumerated()是为了获取每个被映射元素(e)的索引(i)，索引用来从原始 String 中提取与虚拟数据中字节相对应的具体字节。这里必须使用 startIndex 和 offsetBy，而不是原始的整数，这是因为索引类型必须与下标的预期类型匹配。

解密只是使用 encryptOTP()将我们生成的密钥对重新组合的问题，同样是通过对两个密钥的每个 UInt8 执行 XOR 运算来实现的。最终的输出必须使用 Foundation 初始化方法(String 的扩展)将 OTPKey([UInt8])转换回 String。这个初始化方法返回一个可选项，因此我们的方法也返回一个可选项(假设提供了一个无效的密钥对，因为可能无法生成一个有效的 UTF-8 字符串，所以这样做才有意义)。

```
func decryptOTP(keyPair: OTPKeyPair) -> String? {
    let decrypted: OTPKey = keyPair.key1.enumerated().map { i, e in
        e ^ keyPair.key2[i]
    }
    return String(bytes: decrypted, encoding:String.Encoding.utf8)
}
```

如果一次性密码本加密能够真正起作用，则应该能够加密和解密相同的 Unicode 字符串，而不会出现问题。

```
decryptOTP(keyPair: encryptOTP(original: "¡Vamos Swift!"))
```

为了进一步探讨，可以尝试检查 encryptOTP()生成的密钥对的两个密钥。

1.4 π 的计算

数学上具有重大意义的数字 π(或 3.14159…)可以使用许多公式推导出来。其中最简单的公式就是莱布尼茨(Leibniz)公式，莱布尼茨公式假设以下无穷级数的收敛等于 π：

```
π= 4/1 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11…
```

式中无穷级数的分子一直保持为 4，而分母增加了 2，并且运算符号在加法和减法之间交替。

下面我们采用一种直接的方式对无穷序列进行建模，即将公式中的片段转换

为函数中的变量。分子是常数 4，分母是一个从 1 开始并且每次递增 2 的变量，运算根据是加法还是减法表示为−1 或 1。最后，答案用一个变量收集，无穷序列的每一项都与这个变量相加。

```
func calculatePi(nTerms: UInt) -> Double {
    let numerator: Double = 4
    var denominator: Double = 1
    var operation: Double = -1
    var pi: Double = 0
    for _ in 0..<nTerms {
        pi += operation * (numerator / denominator)
        denominator += 2
        operation *= -1
    }
    return abs(pi)
}
```

提示：函数 abs()返回数字的绝对值(总是正值)。

这个函数是这样一个例子，即公式和程序代码之间的机械转换如何在建模或模拟一个有趣的概念时既简单又有效。机械转换是一个非常有用的工具，但必须记住它不一定是最高效的解决方案。当然，π 的莱布尼茨公式可以用更高效或更紧凑的代码来实现。

注意：无穷序列中的项越多(调用 calculatePi()时 nTerms 的值越大)，π 的最终计算结果就越准确。

1.5　汉诺塔

三个垂直桩(从此称为“塔”)竖直站立，分别标记为 A、B 和 C。甜甜圈形状的盘子套在塔 A 上，其中最大的盘子位于底部，命名为盘子 1。盘子 1 上方的其他盘子用递增的数字标记，并且盘子逐渐变小。例如，如果有三个盘子，则最大的盘子(即底部的盘子)为盘子 1；下一个最大的盘子为盘子 2，位于盘子 1 的上方……最后，最小的盘子为盘子 3，位于盘子 2 的上方。我们的目标就是在下述约束下，将所有盘子从塔 A 移动到塔 C：

- 一次只能移动一个盘子。
- 只能移动每个塔上最顶部的盘子。
- 较大的盘子永远不能在较小的盘子上方。

图 1-7 概括示范了这个问题。

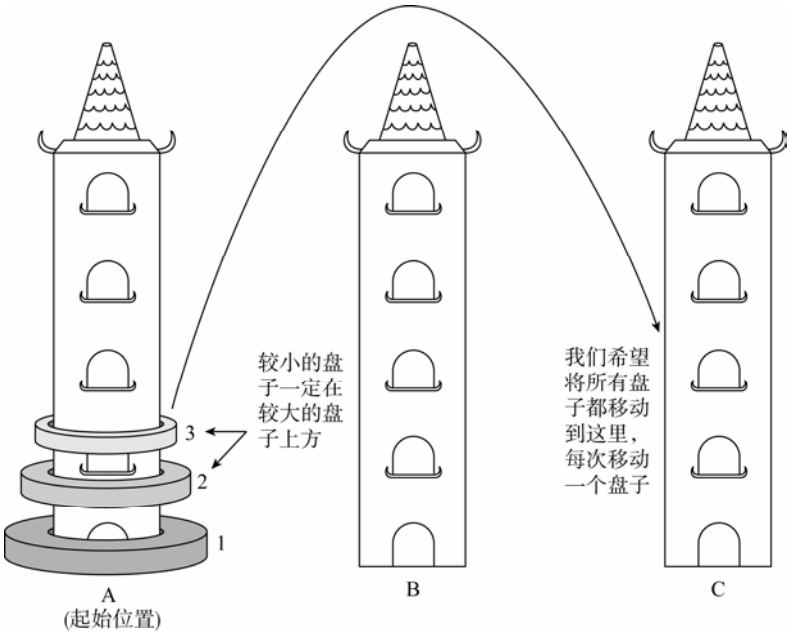


图 1-7 挑战是将三个盘子从塔 A 移动到塔 C，每次移动一个盘子，较大的盘子永远不能在较小盘子的上方

1.5.1 汉诺塔的建模

栈是一种数据结构，基于后进先出(Last-In-First-Out，LIFO)的概念建模，最后一个压入栈里的元素是从栈里取出的第一个元素。栈中两个最基本的操作是压入(push)和弹出(pop)。压入指的是将一个新项放入栈中，而弹出指的是从栈中删除并返回最后一个压入的项。在 Swift 语言中，可使用 Array 作为栈的后台存储空间来轻松地为栈建立模型。

```
public class Stack<T>: CustomStringConvertible {
    private var container: [T] = [T]()
    public func push(_ thing: T) { container.append(thing) }
    public func pop() -> T { return container.removeLast() }
    public var description: String { return container.description }
}
```

注意：本例的 Stack 类实现了 CustomStringConvertible，从而能够轻松地探索塔的内容。当对 Stack 应用 print()时，将输出 description。

栈是汉诺塔问题中塔的完美替身。当把盘子放到塔上时，就相当于把盘子压入栈中；当希望把盘子从一个塔移动到另一个塔时，可以把盘子从第一个栈中弹

出，然后压入第二个栈中。

下面我们将塔定义为 Stack，并用盘子填充第一个塔。

```
var numDiscs = 3
var towerA = Stack<Int>()
var towerB = Stack<Int>()
var towerC = Stack<Int>()
for i in 1...numDiscs { // initialize the first tower
   towerA.push(i)
}
```

1.5.2　解决汉诺塔问题

汉诺塔问题如何解决呢？假设我们只尝试着移动一个盘子，我们知道该怎么做，对吧？事实上，移动一个盘子是汉诺塔递归解决方法的基本情形，递归情形是移动多个盘子。因此，解决汉诺塔问题的关键在于我们实质上需要处理两种情形：移动一个盘子(基本情形)和移动多个盘子(递归情形)。

下面我们以一个具体的例子来理解递归情形。假设塔 A 上有 3 个盘子(分别位于顶部、中间和底部)，我们希望将这些盘子移动到塔 C(下面的思路有助于勾画问题)。首先将顶部的盘子移动到塔 C，接着将中间的盘子移动到塔 B，然后将顶部的盘子从塔 C 移动到塔 B。现在底部的盘子仍在塔 A 上，上面的两个盘子已经移到了塔 B 上。实质上，我们现在已经成功地将两个盘子从一个塔(A)移动到另一个塔(B)。将底部的盘子从塔 A 移动到塔 C 是基本情形(移动一个盘子)。接下来我们可以将上面两个盘子从塔 B 移动到塔 C，步骤与将盘子从塔 A 移到塔 B 相同：首先将顶部的盘子移动到塔 A，然后将中间的盘子移动到塔 C，最后将顶部的盘子从塔 A 移动到塔 C。

提示：在“计算机科学”课堂上，使用销钉和塑料甜圈制作塔的缩微模型并不罕见。不过大家也可以使用 3 支铅笔和 3 张纸来搭建自己的模型，模型可以帮助我们形象化地演示解决方法。

在上述 3 个盘子的情景中，基本情形就是移动一个盘子，递归情形就是移动所有其他盘子(本例中为两个盘子)，第三个塔作为临时塔使用。我们可以将汉诺塔的递归解决方法分解为 3 个步骤[1]：

(1) 以塔 C 为中介，将上面的 n−1 个盘子从塔 A 移动到塔 B(临时塔)。

(2) 将底部的盘子从塔 A 移动到塔 C。

1 “About the Towers of Hanoi”，选自 *Surveying the Field of Computing*(Carl Burch，1999 年)，网址为 http://mng.bz/c1i2。

(3) 将 n−1 个盘子从塔 B 移动到塔 C。

令人惊奇的是，这种递归算法不仅适用于 3 个盘子，而且还适用于任意数量的盘子。下面我们将这个递归算法编码为一个名为 hanoi()的函数，该函数负责将盘子从一个塔移动到另一个塔(假设有第三个临时塔)。

```
func hanoi(from: Stack<Int>, to: Stack<Int>, temp: Stack<Int>, n: Int) {
    if n == 1 { // base case
        to.push(from.pop()) // move 1 disk
    } else { // recursive case
        hanoi(from: from, to: temp, temp: to, n: n-1)
        hanoi(from: from, to: to, temp: temp, n: 1)
        hanoi(from: temp, to: to, temp: from, n: n-1)
    }
}
```

在调用 hanoi()之后，应该检查一下塔 A、B 和 C，验证盘子移动成功。

```
hanoi(from: towerA, to: towerC, temp: towerB, n: numDiscs)
print(towerA)
print(towerB)
print(towerC)
```

结果证实盘子移动成功。在编写汉诺塔解决方法时，我们不必理解将多个盘子从塔 A 移动到塔 C 所需的每一步。但是，我们需要理解移动任意数量盘子的通用递归算法，我们可以编写这样的代码，剩下的就交给计算机来完成。这就是采用递归方法解决问题的魅力所在——只需要以抽象的方式思考解决方案，而不需要在脑海中严密思考每个单独的动作。

顺便提一句，hanoi()函数的执行次数以指数级数量增长，如果要移动 64 个盘子，则致使汉诺塔问题的解决变得岌岌可危。有关汉诺塔的传说，可以阅读更多资源。如果希望了解递归解决方法的数学原理，可参见 Carl Burch 在“About the Towers of Hanoi”一文中的解释(网址为 http://mng.bz/c1i2)。

1.6 实际应用

本章介绍的各种技术(递归、计算缓存、压缩和位级运算)在现代软件开发中非常普遍，以至于无法想象如果计算领域没有这些技术将会是什么样。尽管没有这些技术的话，问题仍然可以得到解决，但使用这些技术解决问题往往更符合逻辑或更有成效。

尤其是递归技术，它不仅仅是许多算法的核心，甚至是整个编程语言的核心。

在一些函数式编程语言(例如 Scheme 和 Haskell)中，递归取代了过程语言中的循环。然而，值得记住的是，任何可以用递归技术完成的事情也可以用迭代技术来完成。

计算缓存技术已成功应用于解析器(解释语言的程序)工作的加速。在最近计算结果可能会再次使用的所有领域，计算缓存技术非常有用。计算缓存技术的另一个应用就是语言运行时，某些语言运行时(例如各版本的 Prolog)会自动存储函数调用的结果(自动计算缓存，auto-memoization)，因此在下次调用相同的函数时不需要再次执行该函数。

互联网连接的世界受到带宽的限制，压缩技术使互联网变得可以接受。1.2 节中介绍的位串技术适用于现实世界的简单数据类型，这些数据类型的可能值数量有限，即使使用一个字节的空间存储它们也显得多余。然而，大多数压缩算法通过查找数据集内的模式或结构以消除重复的信息，它们相比 1.2 节中讨论的技术要复杂得多。

对于一般加密，一次性密码本并不实用，这是因为一次性密码本要求加密器和解密器都拥有相同的密钥(在本例中即虚拟数据)，以便重构原始数据，这种方式很麻烦，并且破坏了大多数加密方案(保证密钥保密)的目的。但大家可能有兴趣知道，“一次性密码本”的名称来自于冷战时期的间谍，他们使用记录有虚拟数据的真实纸垫来建立加密通信。

这些技术都是编程时使用的基本构件，其他算法构建在这些构件之上。在后面的章节中还将广泛应用到这些技术。

1.7　练习

(1) 使用自己的设计技术，编写另一个计算斐波那契数列元素 n 的函数。编写单元测试，以测试自己所编写函数的正确性，以及与本章中介绍的其他版本的函数在性能上的差异。

(2) 1.2 节中使用的 Core Foundation 构造 CFMutableBitVector 有一个 C API。按照人体工程学为这个 API 编写一个 Swift 包装器，然后使用包装器重新实现 CompressedGene。

(3) 编写一个汉诺塔问题解决程序，可以适用于任意数量的塔。

(4) 使用一次性密码本加密和解密图像。