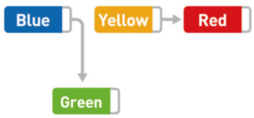
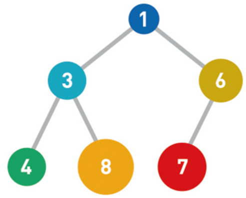
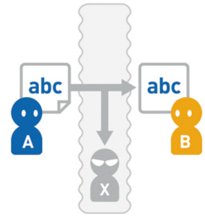
图灵程序设计丛书



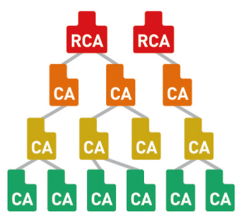


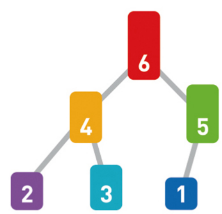


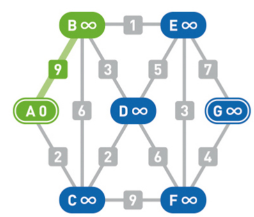
SHOEISHA

我的第一本算法书

[日] 石田保辉 宫崎修一 / 著 张贝 / 译







481张步骤图

详解26个算法
和7个数据结构的基本原理

全彩印刷

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

石田保辉

自由职业工程师，现居日本东京。2011年毕业于日本京都大学研究生院。辗转于几个创新型企业后独立，成为自由职业者。2016年，个人制作的面向工程师的学习型App“算法动画图解”上架，不到1年时间全球下载量即达到50万次，并入选了“App Store日本区2016年度最佳应用”榜单。

宫崎修一

日本京都大学学术信息媒体中心副教授。1998年从日本九州大学博士生院工学专业毕业后，开始担任日本京都大学研究生院信息学研究科助手，2002年起担任现职。主要研究算法和计算复杂性理论。近期的重点研究对象为相似算法和在线算法。主要著作有《图论入门：基本知识和算法》（日本森北出版社，2015年出版）。

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

图灵程序设计丛书

我的第一本算法书

[日] 石田保辉 宫崎修一 / 著 张贝 / 译

人民邮电出版社
北京

图书在版编目（CIP）数据

我的第一本算法书 /（日）石田保辉，（日）宫崎修一著；张贝译. -- 北京：人民邮电出版社，2018.11
（图灵程序设计丛书）
ISBN 978-7-115-49524-2

Ⅰ. ①我… Ⅱ. ①石… ②宫… ③张… Ⅲ. ①电子计算机—算法理论 Ⅳ. ①TP301.6

中国版本图书馆CIP数据核字（2018）第223107号

内 容 提 要

本书采用大量图片，通过详细的分步讲解，以直观、易懂的方式展现了 7 个数据结构和 26 个基础算法的基本原理。第 1 章介绍了链表、数组、栈等 7 个数据结构；从第 2 章到第 7 章，分别介绍了和排序、查找、图论、安全、聚类等相关的 26 个基础算法，内容涉及冒泡排序、二分查找、广度优先搜索、哈希函数、迪菲 – 赫尔曼密钥交换、*k*-means 算法等。

本书没有枯燥的理论和复杂的公式，而是通过大量的步骤图帮助读者加深对数据结构原理和算法执行过程的理解，便于学习和记忆。将本书作为算法入门的第一步，是非常不错的选择。

本书适合所有对算法感兴趣，想要从零开始学习算法的读者阅读。

◆ 著 [日] 石田保辉 宫崎修一
译 张 贝
责任编辑 高宇涵
责任印制 周昇亮
◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷
◆ 开本：880×1230 1/24
印张：8.5
字数：210千字 2018年11月第1版
印数：1–4 000册 2018年11月北京第1次印刷
著作权合同登记号 图字：01-2017-7988号

定价：69.00元
读者服务热线：(010)51095186转600 印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字20170147号

版 权 声 明

本书主页

http://www.ituring.com.cn/book/2464

阅读本书后，您可通过以上网址将您的感想、意见或问题写下来。
另外，点击页面右侧的“提交 / 查看勘误”可以提交或查看本书勘误。

前言

本书以 iOS 和 Android 平台上的应用程序“算法动画图解”为基础，以图配文，详细讲解了各种算法和数据结构的基本原理。如果本书能够帮助大家理解基本算法的操作和特征，那么我将感到十分荣幸。

使用不同的算法解决同一个问题时，就算得到的结果是一样的，算法之间的性质也有很大的差异。比如，某个算法的运行时间很短，但需要占用大量内存；而另一个算法运行时间较长，但内存资源占用较少。学习各种算法可以使我们在编程时有更多的选择。成为优秀程序员的必要条件之一，就是可以根据应用场景选择最合适的算法。

如果您对算法有兴趣，还可以挑战一下“算法理论”这门学科，试着去发现更高效的算法，或者研究目前用算法还无法解决的问题。

石田保辉

算法是解决问题的计算步骤，用于编写程序之前。即使是解决同样的问题，高效算法和低效算法所花费的时间也迥然不同。另外，要想执行高效的算法，还需要使用合适的数据结构。本书的目的就是让初学者也能轻松地理解算法和数据结构。

本书以 iOS 和 Android 平台上的应用程序“算法动画图解”为基础。该应用以动画的形式展示了算法的流程，而本书则采用了大量的图片来分步讲解，尽量保留了原应用易懂的优点。为了配合出版，本书还添加了“什么是算法”“算法的运行时间”“图的基础知识”等应用中没有的章节，相信会让读者对算法的理解更加深刻。

读完本书，不过是站在了算法世界的入口，这个世界还有很多领域等待人们去探索。如果您由此对算法产生了兴趣，请务必继续深入学习。

宫崎修一

谢辞

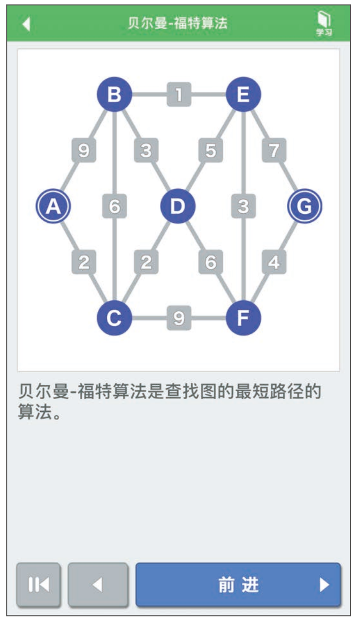
本书中大量使用了应用程序“算法动画图解”中的图片，在使用之前，我们得到了图片制作者光森裕树先生的许可。此外，从选题策划、内容编辑到出版进度管理，翔泳社的秦和宏先生在本书的整个出版流程中都付出了颇多心血。在此对二位表示由衷的感谢。

石田保辉　宫崎修一

关于应用程序“算法动画图解”的说明

本书根据 iOS 和 Android 平台上的应用程序“算法动画图解”编写而成，为配合图书出版，对内容进行了补充和修正，专门添加了基础理论方面的内容。

对于本书中出现的各个算法，应用内都采用动画交互的方式进行了讲解，部分算法还可以通过改变设置尝试不同的模式。将应用和本书结合使用，可以加深对相关知识的理解。请各位读者通过下面的步骤下载该应用并加以灵活运用。



▶ iPhone / iPad 用户

① 打开 App Store

② 点击“搜索”，输入“算法动画图解”

③ 点击“获取”

▶ Android 用户

① 打开 Google Play

② 打开页面上的“搜索”项，输入“算法动画图解”进行搜索

③ 进入应用“Algorithms: Explained and Animated”的页面，点击“安装”

※ 应用为免费下载，可以免费试用部分算法，但解锁所有项目需要在应用内购买。

序章

算法的基本知识

No.
0-1 什么是算法

算法与程序的区别

算法就是计算或者解决问题的步骤。我们可以把它想象成食谱。要想做出特定的料理，就要遵循食谱上的步骤；同理，要想用计算机解决特定的问题，就要遵循算法。这里所说的特定问题多种多样，比如“将随意排列的数字按从小到大的顺序重新排列”“寻找出发点到目的地的最短路径”，等等。

食谱和算法之间最大的区别就在于算法是严密的。食谱上经常会有描述得比较模糊的部分，而算法的步骤都是用数学方式来描述的，所以十分明确。

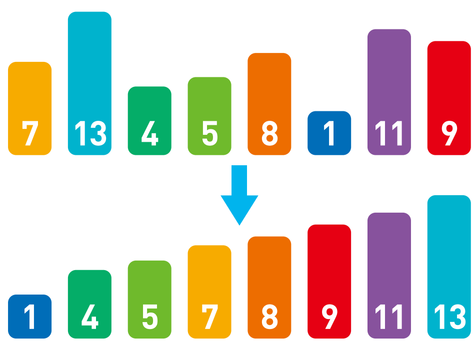
算法和程序有些相似，区别在于程序是以计算机能够理解的编程语言编写而成的，可以在计算机上运行，而算法是以人类能够理解的方式描述的，用于编写程序之前。不过，在这个过程中到哪里为止是算法、从哪里开始是程序，并没有明确的界限。

就算使用同一个算法，编程语言不同，写出来的程序也不同；即便使用相同的编程语言，写程序的人不同，那么写出来的程序也是不同的。

排列整数的算法：排序

▶ 查找最小的数字并交换：选择排序

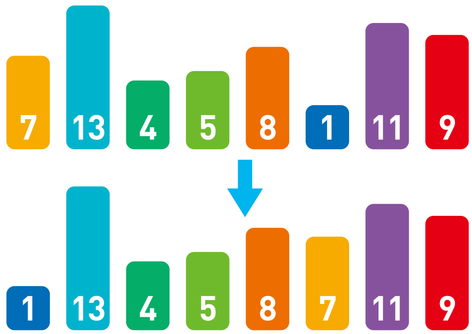
来看一个具体的算法示例吧。这是一个以随意排列的整数为输入，把它们按从小到大的顺序重新排列的问题。这类排序问题我们将在第 2 章详细讲解。



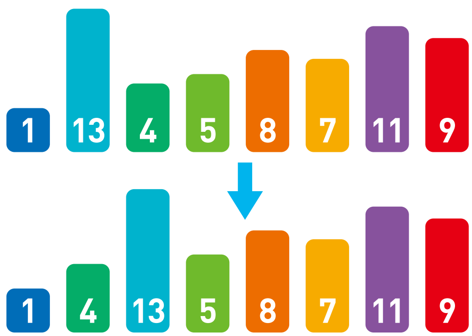
只解决这一个问题很简单，但是算法是可以应对任意输入的计算步骤，所以必须采用通用的描述。虽然在这个示例中输入的整数个数 n 为 8，然而不管 n 多大，算法都必须将问题解决。

那么，你首先想到的方法，是不是先从输入的数字中找出最小的数字，再将它和最左边的数字交

换位置呢？在这个示例中就是找到最小数字 1，然后将它和最左边的 7 交换位置。



这之后 1 的位置便固定下来，不再移动。接下来，在剩下的数字里继续寻找最小数，再将它和左边第 2 个数字交换位置。于是，4 和 13 也交换了位置。



我们将这样的一次交换称为“1 轮”。到了第 k 轮的时候，就把剩下的数字中最小的一个，与左边开始第 k 个数字进行交换。于是在结束第 k 轮后，从左数的 k 个数字便都按从小到大的顺序排列了。只要将这个步骤重复 n 次，那么所有的数字都将按从小到大的顺序排列。

这便是我们将在 2-3 节中介绍的选择排序。不管输入的数字是什么、n 有多大，都可以用这个算法解决问题。

▶ 用计算机能理解的方式构思解法：算法的设计

计算机擅长高速执行一些基本命令，但无法执行复杂的命令。此处的“基本命令”指的是“做加法”或者“在指定的内存地址上保存数据”等。

计算机是以这些基本命令的组合为基础运行的，面对复杂的操作，也是通过搭配组合这些

基本命令来应对的。上文中提到的“对 n 个数字进行排序”对计算机来说就是复杂的操作。如何设计算法来解决这个排序问题，也就等同于构思如何搭配组合计算机可以执行的那些基本命令来实现这个操作。

如何选择算法

能解决排序问题的算法不止选择排序这一个。那么，当有多个算法都可以解决同一个问题时，我们该如何选择呢？在算法的评判上，考量的标准也各有不同。

比如，简单的算法对人来说易于理解，也容易被写成程序，而在运行过程中不需要耗费太多空间资源的算法，就十分适用于内存小的计算机。

不过，一般来说我们最为重视的是算法的运行时间，即从输入数据到输出结果这个过程所花费的时间。

对 50 个数字排序所花的时间竟然比宇宙的历史还要长吗

▶ 使用全排列算法进行排序

为了让大家体会一下低效率算法的效果，这里来看看下面这个排序算法。

① 生成一个由 n 个数字构成的数列（不和前面生成的数列重复）
② 如果①中生成的数列按从小到大的顺序排列就将其输出，否则回到步骤①

我们就把这个算法称为“全排列算法”吧。全排列算法列出了所有的排列方法，所以不管输入如何，都可以得到正确的结果。

那么，需要等多久才能出结果呢？若运气好，很快就能出现正确排列的话，结果也就立马出来了。然而，实际情况往往并不如我们所愿。最差的情况，也就是直到最后才出现正确排列的情况下，计算机就不得不确认所有可能的排列了。

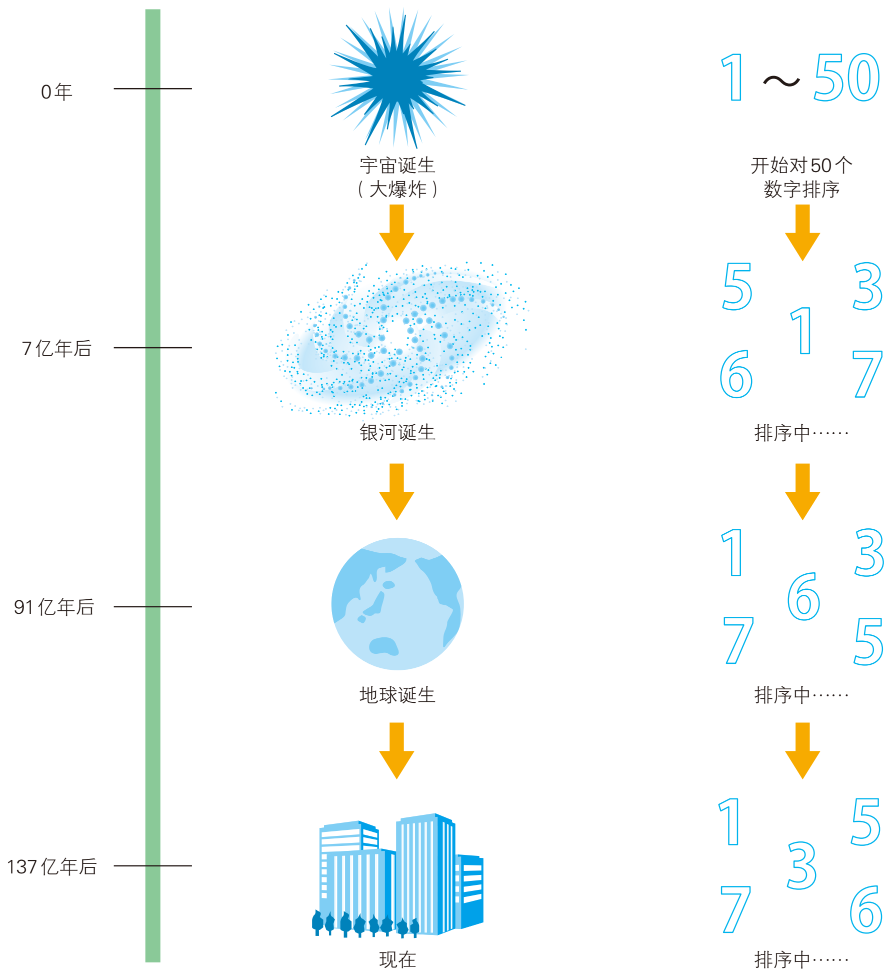
n 个数字有 $n!$ 种不同的排列方法（$n!=n(n-1)(n-2)\cdots3\cdot2\cdot1$）。现在，我们来看看 $n=50$ 时是怎样一种情况吧。

① $50! = 50\cdot49\cdot48\cdots3\cdot2\cdot1$
② $\qquad 50\cdot49\cdot48\cdots3\cdot2\cdot1 > 50\cdot49\cdot48\cdots13\cdot12\cdot11$
③ $\qquad\qquad 50\cdot49\cdot48\cdots13\cdot12\cdot11 > 10^{40}$

公式①中，50! 即为数字 1 到数字 50 的乘积。为了便于计算，我们通过公式②③将结果近似转换为 10 的 n 次方的形式。公式②右边部分去掉了 10 以下的数字，因此小于 50!。公式③左右都是 40 个数字的乘积，但左边数字都大于 10，因此大于右边的 10^{40}。接下来我们就用 10^{40} 近似代表 50 个数字的所有排列情况来进行计算。

假设 1 台高性能计算机 1 秒能检查 1 万亿（$=10^{12}$）个数列，那么检查 10^{40} 个数列将花费的时间为 $10^{40} \div 10^{12}=10^{28}$ 秒。1 年为 31 536 000 秒，不到 10^{8} 秒。因此，10^{28} 秒> 10^{20} 年。

从大爆炸开始宇宙已经经历了约 137 亿年，即便如此也少于 10^{11} 年。也就是说，仅仅是对 50 个数字进行排序，若使用全排列算法，就算花费宇宙年龄的 10^{9} 倍时间也得不出答案。



使用选择排序算法进行排序

那么，使用前文提到的选择排序算法，情况又将如何呢?

首先，为了在第 1 轮找到最小的数字，需要从左往右确认数列中的数字，只要查询 n 个数字即可。在接下来的第 2 轮中，需要从 $n-1$ 个数字中寻找最小值，所以需要查询 $n-1$ 个数字。将这个步骤进行到第 n 轮的时候，需要查询的次数如下。

$$n+(n-1)+(n-2)+\cdots 3+2+1=\frac{n\,(n+1)}{2}\leqslant n^2$$

$n=50$ 的时候 $n^2=2500$。假设 1 秒能确认 1 万亿（$=10^{12}$）个数字，那么 $2500\div 10^{12}=0.000\,000\,002\,5$ 秒便能得出结果，比全排列算法的效率高得多。

No.
0-2 运行时间的计算方法

了解输入数据的量和运行时间之间的关系

上一节在结尾说明了算法的不同会导致其运行时间产生大幅变化，本节将讲解如何求得算法的运行时间。

使用相同的算法，输入数据的量不同，运行时间也会不同。比如，对 10 个数字排序和对 1 000 000 个数字排序，大家很容易就想到后者的运行时间更长。那么，实际上运行时间会长多少呢？后者是前者的 100 倍，还是 1 000 000 倍？就像这样，我们不光要理解不同算法在运行时间上的区别，还要了解根据输入数据量的大小，算法的运行时间具体会产生多大的变化。

如何求得运行时间

那么，如何测算不同输入所导致的运行时间的变化程度呢？最为现实的方法就是在计算机上运行一下程序，测试其实际花费的时间。但是，就算使用同样的算法，花费的时间也会根据所用计算机的不同而产生偏差，十分不便。

所以在这里，我们使用“步数”来描述运行时间。“1 步”就是计算的基本单位。通过测试“计算从开始到结束总共执行了多少步”来求得算法的运行时间。

作为示例，现在我们试着从理论层面求出选择排序的运行时间。选择排序的步骤如下。

① 从数列中寻找最小值
② 将最小值和数列最左边的数字进行交换，排序结束。回到①

如果数列中有 n 个数字，那么①中“寻找最小值”的步骤只需确认 n 个数字即可。这里，将“确认 1 个数字的大小”作为操作的基本单位，需要的时间设为 T_c，那么步骤①的运行时间就是 $n \times T_c$。

接下来，把“对两个数字进行交换”也作为操作的基本单位，需要的时间设为 T_s。那么，①和②总共重复 n 次，每经过“1 轮”，需要查找的数字就减少 1 个，因此总的运行时间如下。

$$(n\times T_c+T_s)+((n-1)\times T_c+T_s)+((n-2)\times T_c+T_s)+\cdots+(2\times T_c+T_s)+(1\times T_c+T_s)$$
$$=\frac{1}{2}T_c n(n+1)+T_s n$$
$$=\frac{1}{2}T_c n^2+(\frac{1}{2}T_c+T_s)n$$

虽说只剩最后 1 个数字的时候就不需要确认了，但是方便起见还是把对它的确认和交换时间计算在内比较好。

运行时间的表示方法

虽说我们已经求得了运行时间，但其实这个结果还可以简化。T_c 和 T_s 都是基本单位，与输入无关。会根据输入变化而变化的只有数列的长度 n，所以接下来考虑 n 变大的情况。n 越大，上式中的 n^2 也就越大，其他部分就相对变小了。也就是说，对式子影响最大的是 n^2。所以，我们删掉其他部分，将结果表示成下式右边的形式。

$$\frac{1}{2}T_c n^2+(\frac{1}{2}T_c+T_s)n=O(n^2)$$

通过这种表示方法，我们就能大致了解到排序算法的运行时间与输入数据量 n 的平方成正比。

同样地，假设某个算法的运行时间如下。

$$5T_x n^3+12T_y n^2+3T_z n$$

那么，这个结果就可以用 $O(n^3)$ 来表示。如果运行时间为

$$3n\log n+2T_y n$$

这个结果就可以用 $O(n\log n)$ 来表示。

O 这个符号的意思是“忽略重要项以外的内容”，读音同 Order。$O(n^2)$ 的含义就是“算法的运行时间最长也就是 n^2 的常数倍”，准确的定义请参考相关专业书籍。重点在于，通过这种表示方法，我们可以直观地了解算法的时间复杂度[①]。

比如，当我们知道选择排序的时间复杂度为 $O(n^2)$、快速排序的时间复杂度为 $O(n\log n)$ 时，很快就能判断出快速排序的运算更为高速。二者的运行时间根据输入 n 产生的变化程度也一目了然。

关于算法的基本知识就介绍到这里了。从下一章开始，我们就来具体学习各种算法吧。

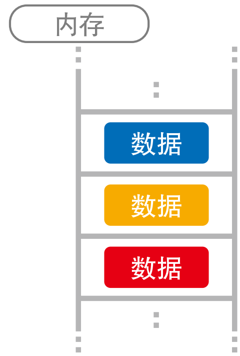
① 时间复杂度是一个可以描述算法运行时间的函数，常用大 O 符号来表述。——译者注

第 1 章

数据结构

No.
1-1 什么是数据结构

决定了数据的顺序和位置关系



数据存储于计算机的内存中。内存如右图所示，形似排成 1 列的箱子，1 个箱子里存储 1 个数据。

数据存储于内存时，决定了数据顺序和位置关系的便是“数据结构”。

电话簿的数据结构

▶ 例① 从上往下顺序添加

举个简单的例子。假设我们有 1 个电话簿——虽说现在很多人都把电话号码存在手机里，但是这里我们考虑使用纸质电话簿的情况——每当我们得到了新的电话号码，就按从上往下的顺序把它们记在电话簿上。

姓名	电话号码
武小小	010-uuuu-uuuu
田美丽	010-xxxx-xxxx
王野	010-yyyy-yyyy
小季酒店	021-zzzz-zzzz
……	……

假设此时我们想给“张伟”打电话，但是因为数据都是按获取顺序排列的，所以我们并不知道张伟的号码具体在哪里，只能从头一个个往下找（虽说也可以“从后往前找”或者“随机查找”，但是效率并不会比“从上往下找”高）。如果电话簿上号码不多的话很快就能找到，但如果存了 500 个号码，找起来就不那么容易了。

▶ 例② 按姓名的拼音顺序排列

接下来，试试以联系人姓名的拼音顺序排列吧。因为数据都是以字典顺序排列的，所以它们是有“结构”的。

姓名	电话号码
董大海	010-aaaa-aaaa
方帅	010-bbbb-bbbb
韩宏宇	020-zzzz-zzzz
李希	010-cccc-cccc
……	……

使用这种方式给联系人排序的话，想要找到目标人物就轻松多了。通过姓名的拼音首字母就能推测出该数据的大致位置。

那么，如何往这个按拼音顺序排列的电话簿里添加数据呢？假设我们认识了新朋友“柯津博”并拿到了他的电话号码，打算把号码记到电话簿中。由于数据按姓名的拼音顺序排列，所以柯津博必须写在韩宏宇和李希之间，但是上面的这张表里已经没有空位可供填写，所以需要把李希及其以下的数据往下移 1 行。

此时我们需要从下往上执行“将本行的内容写进下一行，然后清除本行内容”的操作。如果一共有 500 个数据，一次操作需要 10 秒，那么 1 个小时也完成不了这项工作。

▶ 两种方法的优缺点

总的来说，数据按获取顺序排列的话，虽然添加数据非常简单，只需要把数据加在最后就可以了，但是在查询时较为麻烦；以拼音顺序来排列的话，虽然在查询上较为简单，但是添加数据时又会比较麻烦。

虽说这两种方法各有各的优缺点，但具体选择哪种还是要取决于这个电话簿的用法。如果电话簿做好之后就不再添加新号码，那么选择后者更为合适；如果需要经常添加新号码，但不怎么需要再查询，就应该选择前者。

▶ 将获取顺序与拼音顺序结合起来怎么样

我们还可以考虑一种新的排列方法，将二者的优点结合起来。那就是分别使用不同的表存储不同的拼音首字母，比如表 L、表 M、表 N 等，然后将同一张表中的数据按获取顺序进行排列。

表 L

姓名	电话号码
李博	010-aaaa-aaaa
林广川	010-bbbb-bbbb
陆顺平	021-zzzz-zzzz
刘彻	010-ccc-cccc
……	……

表 M

姓名	电话号码
马岩	010-aaaa-aaaa
孟田	021-zzzz-zzzz
明小慧	010-zzzz-zzzz
孟舒怡	010-aaaa-aaaa
……	……

表 N

姓名	电话号码
宁川	021-aaaa-aaaa
……	……
……	……
……	……
……	……

这样一来，在添加新数据时，直接将数据加入到相应表中的末尾就可以了，而查询数据时，也只需要到其对应的表中去查找即可。

因为各个表中存储的数据依旧是没有规律的，所以查询时仍需从表头开始找起，但比查询整个电话簿来说还是要轻松多了。

选择合适的数据结构以提高内存的利用率

数据结构方面的思路也和制作电话簿时的一样。将数据存储于内存时，根据使用目的选择合适的数据结构，可以提高内存的利用率。

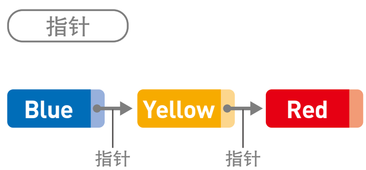
本章将会讲解 7 种数据结构。如本节开头所述，数据在内存中是呈线性排列的，但是我们也可以使用指针等道具，构造出类似“树形”的复杂结构（树形结构将在 4-2 节详细说明）。

▶参考：4-2 广度优先搜索

No.
1-2 链表

链表是数据结构之一，其中的数据呈线性排列。在链表中，数据的添加和删除都较为方便，就是访问比较耗费时间。

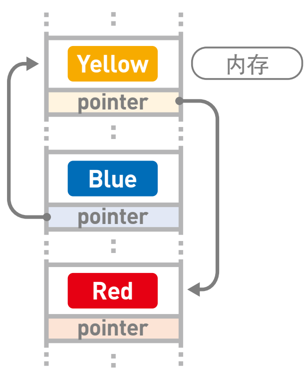
01



Red 是最后 1 个数据，所以 Red 的指针不指向任何位置。

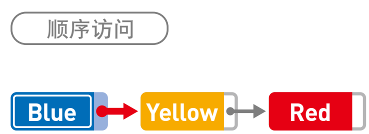
这就是链表的概念图。Blue、Yellow、Red 这 3 个字符串作为数据被存储于链表中。每个数据都有 1 个“指针”，它指向下一个数据的内存地址。

02



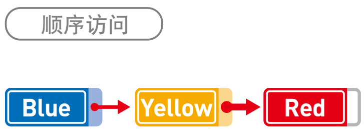
在链表中，数据一般都是分散存储于内存中的，无须存储在连续空间内。

03



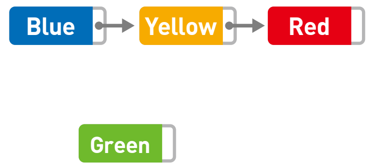
因为数据都是分散存储的，所以如果想要访问数据，只能从第 1 个数据开始，顺着指针的指向一一往下访问（这便是顺序访问）。比如，想要找到 Red 这一数据，就得从 Blue 开始访问。

04



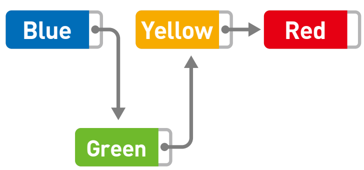
这之后，还要经过Yellow，我们才能找到Red。

05



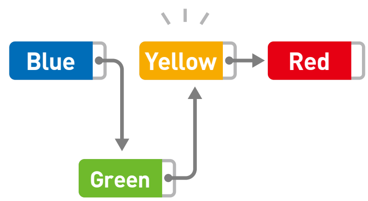
如果想要添加数据，只需要改变添加位置前后的指针指向就可以，非常简单。比如，在Blue和Yellow之间添加Green。

06



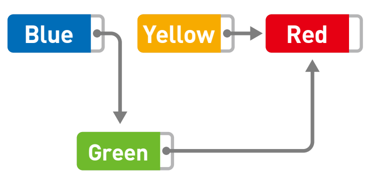
将Blue的指针指向的位置变成 Green，然后再把Green的指针指向Yellow，数据的添加就大功告成了。

07



数据的删除也一样，只要改变指针的指向就可以，比如删除Yellow。

08



这时，只需要把Green指针指向的位置从Yellow变成Red，删除就完成了。虽然Yellow本身还存储在内存中，但是不管从哪里都无法访问这个数据，所以也就没有特意去删除它的必要了。今后需要用到Yellow所在的存储空间时，只要用新数据覆盖掉就可以了。

解说

对链表的操作所需的运行时间到底是多少呢？在这里，我们把链表中的数据量记成 n。访问数据时，我们需要从链表头部开始查找（线性查找），如果目标数据在链表最后的话，需要的时间就是 $O(n)$。

另外，添加数据只需要更改两个指针的指向，所以耗费的时间与 n 无关。如果已经到达了添加数据的位置，那么添加操作只需花费 $O(1)$ 的时间。删除数据同样也只需 $O(1)$ 的时间。

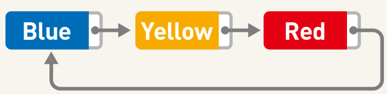
▶参考：3-1 线性查找

补充说明

上文中讲述的链表是最基本的一种链表。除此之外，还存在几种扩展方便的链表。

虽然上文中提到的链表在尾部没有指针，但我们也可以在链表尾部使用指针，并且让它指向链表头部的数据，将链表变成环形。这便是“循环链表”，也叫“环形链表”。循环链表没有头和尾的概念。想要保存数量固定的最新数据时通常会使用这种链表。

循环链表



另外，上文链表里的每个数据都只有一个指针，但我们可以把指针设定为两个，并且让它们分别指向前后数据，这就是“双向链表”。使用这种链表，不仅可以从前往后，还可以从后往前遍历数据，十分方便。

但是，双向链表存在两个缺点：一是指针数的增加会导致存储空间需求增加；二是添加和删除数据时需要改变更多指针的指向。

双向链表

No. 1-3

数组

数组也是数据呈线性排列的一种数据结构。与前一节中的链表不同，在数组中，访问数据十分简单，而添加和删除数据比较耗工夫。这和 1-1 节中讲到的姓名按拼音顺序排列的电话簿类似。

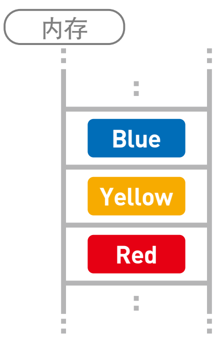
▶参考：1-1 什么是数据结构

01

a 是数组的名字，后面“[ ]”中的数字表示该数据是数组中的第几个数据（这个数字叫作“数组下标”，下标从 0 开始计数）。比如，Red 就是数组 *a* 的第 2 个数据。

这就是数组的概念图。Blue、Yellow、Red 作为数据存储在数组中。

02

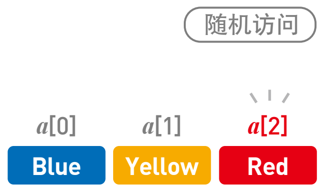


数据按顺序存储在内存的连续空间内。

03

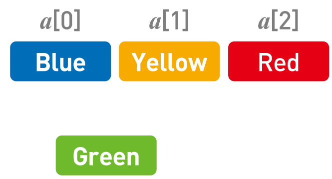
由于数据是存储在连续空间内的，所以每个数据的内存地址（在内存上的位置）都可以通过数组下标算出，我们也就可以借此直接访问目标数据（这叫作“随机访问”）。

04



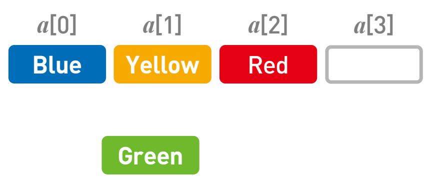
比如现在我们想要访问Red。如果使用指针就只能从头开始查找，但在数组中，只需要指定*a*[2]，便能直接访问Red。

05



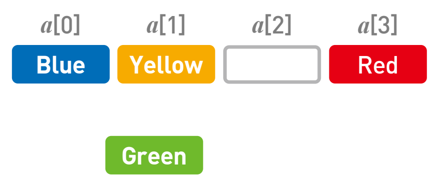
但是，如果想在任意位置上添加或者删除数据，数组的操作就要比链表复杂多了。这里我们尝试将Green添加到第2个位置上。

06



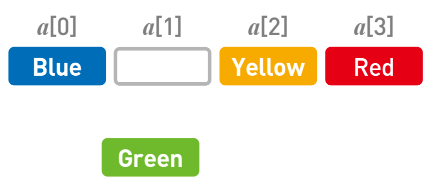
首先，在数组的末尾确保需要增加的存储空间。

07



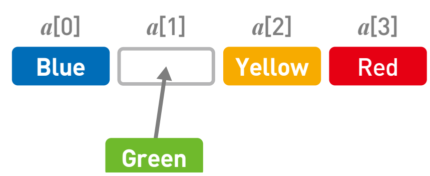
为了给新数据腾出位置，要把已有数据一个个移开。首先把Red往后移。

08



然后把Yellow往后移。

09

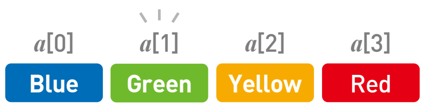


最后在空出来的位置上写入Green。

10

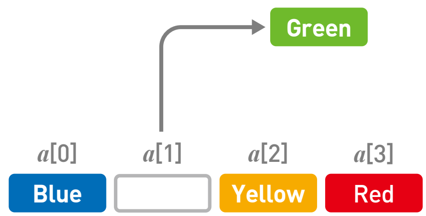
添加数据的操作就完成了。

11



反过来，如果想要删除Green……

12

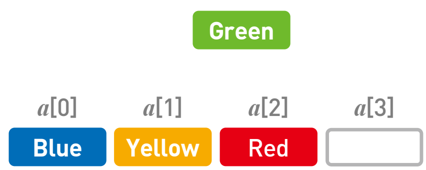


首先，删掉目标数据（在这里指Green）。

13

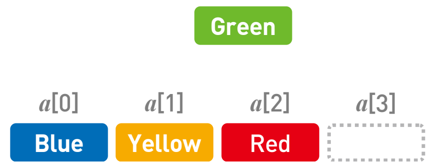
然后把后面的数据一个个往空位移。先把Yellow往前移。

14



接下来移动Red。

15



最后再删掉多余的空间。这样一来Green便被删掉了。

解说

这里讲解一下对数组操作所花费的运行时间。假设数组中有 n 个数据，由于访问数据时使用的是随机访问（通过下标可计算出内存地址），所以需要的运行时间仅为恒定的 $O(1)$。

但另一方面，想要向数组中添加新数据时，必须把目标位置后面的数据一个个移开。所以，如果在数组头部添加数据，就需要 $O(n)$ 的时间。删除操作同理。

补充说明

在链表和数组中，数据都是线性地排成一列。在链表中访问数据较为复杂，添加和删除数据较为简单；而在数组中访问数据比较简单，添加和删除数据却比较复杂。

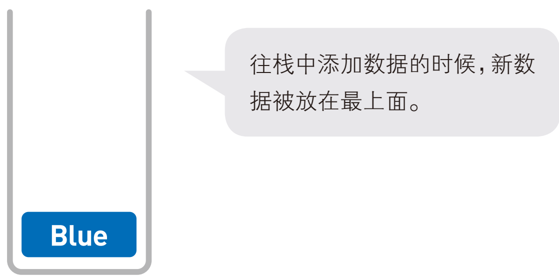
我们可以根据哪种操作较为频繁来决定使用哪种数据结构。

	访问	添加	删除
链表	慢	快	快
数组	快	慢	慢

No. 1-4 栈

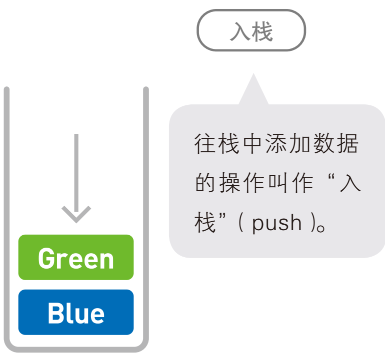
栈也是一种数据呈线性排列的数据结构，不过在这种结构中，我们只能访问最新添加的数据。栈就像是一摞书，拿到新书时我们会把它放在书堆的最上面，取书时也只能从最上面的新书开始取。

01



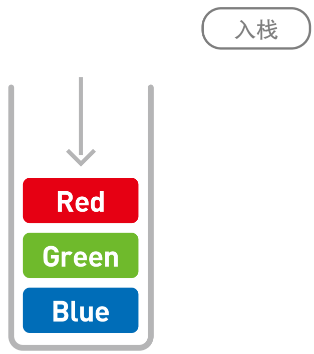
这就是栈的概念图。现在存储在栈中的只有数据Blue。

02

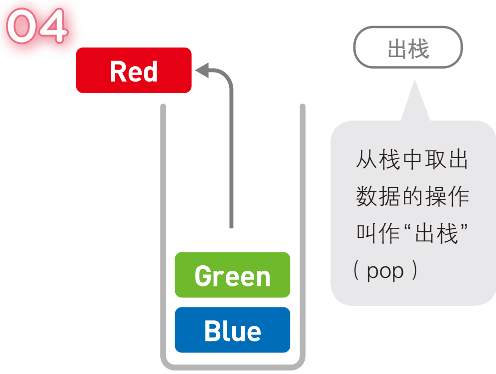


然后，栈中添加了数据Green。

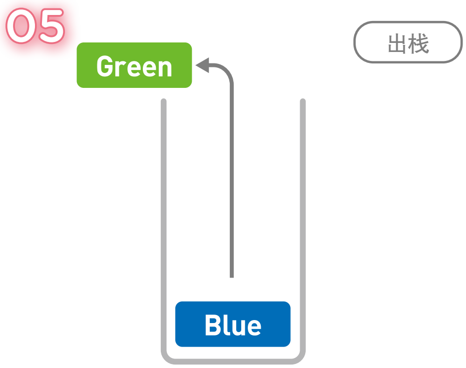
03



接下来，数据Red入栈。



从栈中取出数据时，是从最上面，也就是最新的数据开始取出的。这里取出的是Red。



如果再进行一次出栈操作，取出的就是Green了。

解说

像栈这种最后添加的数据最先被取出，即“后进先出”的结构，我们称为Last In First Out，简称LIFO。

与链表和数组一样，栈的数据也是线性排列，但在栈中，添加和删除数据的操作只能在一端进行，访问数据也只能访问到顶端的数据。想要访问中间的数据时，就必须通过出栈操作将目标数据移到栈顶才行。

应用示例

栈只能在一端操作这一点看起来似乎十分不便，但在只需要访问最新数据时，使用它就比较方便了。

比如，规定（AB（C（DE）F）（G（（H）I J）K））这一串字符中括号的处理方式如下：首先从左边开始读取字符，读到左括号就将其入栈，读到右括号就将栈顶的左括号出栈。此时，出栈的左括号便与当前读取的右括号相匹配。通过这种处理方式，我们就能得知配对括号的具体位置。

另外，我们将要在4-3节中学习的深度优先搜索算法，通常会选择最新的数据作为候补顶点。在候补顶点的管理上就可以使用栈。

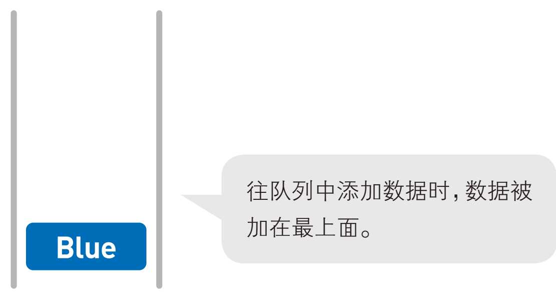
▶参考：4-3 深度优先搜索

No. 1-5

队列

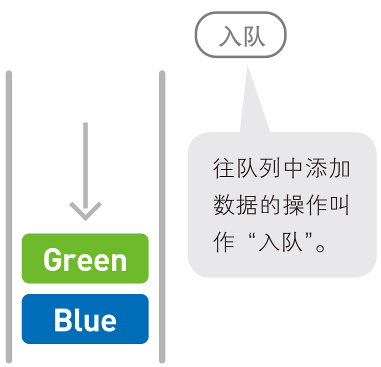
与前面提到的数据结构相同，队列中的数据也呈线性排列。虽然与栈有些相似，但队列中添加和删除数据的操作分别是在两端进行的。就和“队列”这个名字一样，把它想象成排成一队的人更容易理解。在队列中，处理总是从第一名开始往后进行，而新来的人只能排在队尾。

01



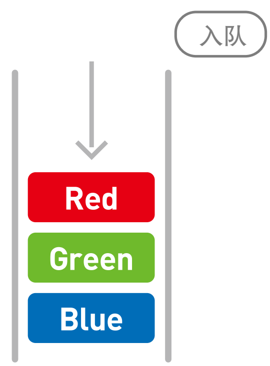
这就是队列的概念图。现在队列中只有数据Blue。

02

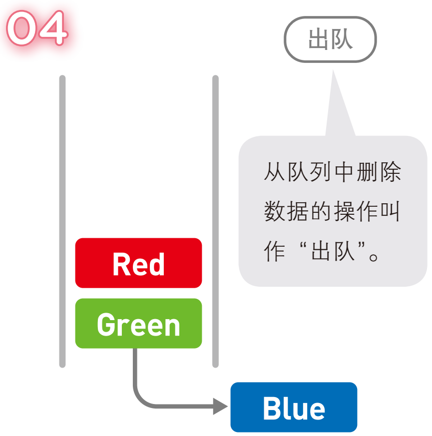


然后，队列中添加了数据Green。

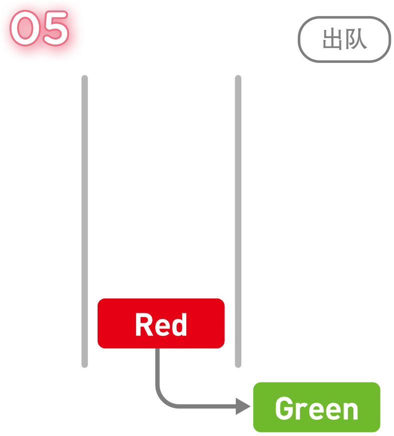
03



紧接着，数据Red也入队了。



从队列中取出（删除）数据时，是从最下面，也就是最早入队的数据开始的。这里取出的是Blue。



如果再进行一次出队操作，取出的就是Green了。

解说

像队列这种最先进去的数据最先被取来，即“先进先出”的结构，我们称为 First In First Out，简称 FIFO。

与栈类似，队列中可以操作数据的位置也有一定的限制。在栈中，数据的添加和删除都在同一端进行，而在队列中则分别是在两端进行的。队列也不能直接访问位于中间的数据，必须通过出队操作将目标数据变成首位后才能访问。

应用示例

“先来的数据先处理”是一种很常见的思路，所以队列的应用范围非常广泛。比如4-2节中将要讲解的广度优先搜索算法，通常就会从搜索候补中选择最早的数据作为下一个顶点。此时，在候补顶点的管理上就可以使用队列。

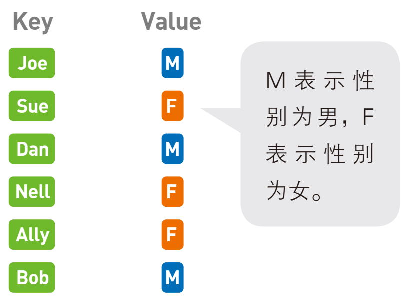
▶参考：4-2 广度优先搜索

No. 1-6 哈希表

在哈希表这种数据结构中，使用将在 5-3 节讲解的“哈希函数”，可以使数据的查询效率得到显著提升。

▶参考：5-3 哈希函数

01



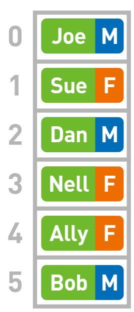
哈希表存储的是由键（key）和值（value）组成的数据。例如，我们将每个人的性别作为数据进行存储，键为人名，值为对应的性别。

02

为了和哈希表进行对比，我们先将这些数据存储在数组中（数组的详细讲解在 1-3 节）。

▶参考：1-3 数组

03



此处准备了 6 个箱子（即长度为 6 的数组）来存储数据。假设我们需要查询 Ally 的性别，由于不知道 Ally 的数据存储在哪个箱子里，所以只能从头开始查询。这个操作便叫作“线性查找”（线性查找的讲解在 3-1 节）。

▶参考：3-1 线性查找

提示 一般来说，我们可以把键当成数据的标识符，把值当成数据的内容。

04

0号箱子中存储的键是Joe而不是Ally。

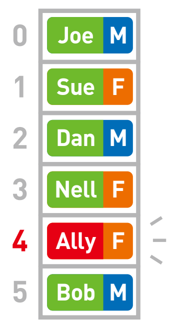
05

1号箱子中的也不是Ally。

06

同样，2号、3号箱子中的也都不是Ally。

07

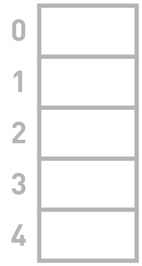


查找到4号箱子的时候，发现其中数据的键为Ally。把键对应的值取出，我们就知道Ally的性别为女（F）了。

08

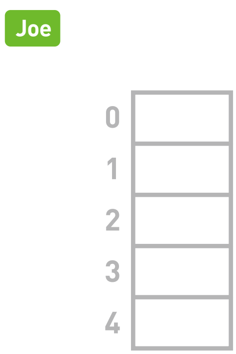
数据量越多，线性查找耗费的时间就越长。由此可知：由于数据的查询较为耗时，所以此处并不适合使用数组来存储数据。

09



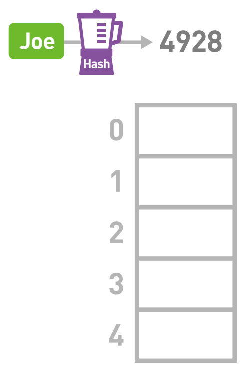
但使用哈希表便可以解决这个问题。首先准备好数组，这次我们用5个箱子的数组来存储数据。

10



尝试把Joe存进去。

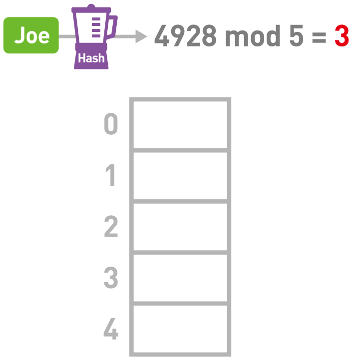
11



使用哈希函数（Hash）计算Joe的键，也就是字符串“Joe”的哈希值。得到的结果为4928（哈希函数的详细说明在5-3节）。

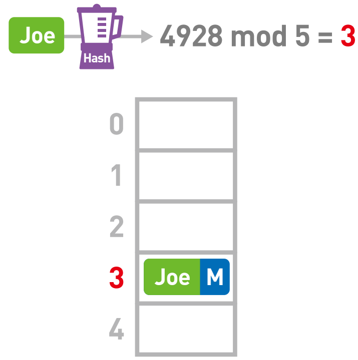
参考：5-3 哈希函数

12



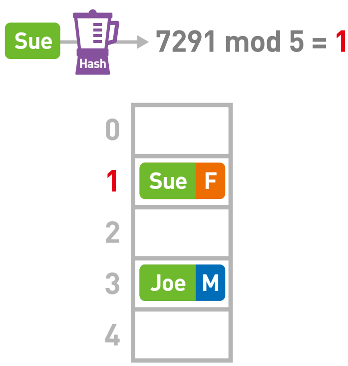
将得到的哈希值除以数组的长度5，求得其余数。这样的求余运算叫作“mod运算”。此处mod运算的结果为3。

13



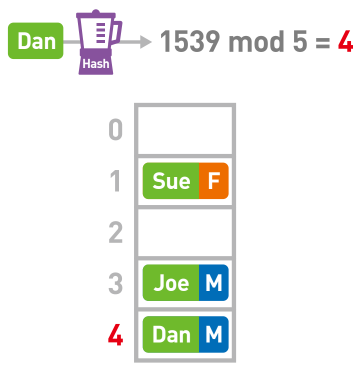
因此，我们将Joe的数据存进数组的3号箱子中。重复前面的操作，将其他数据也存进数组中。

14



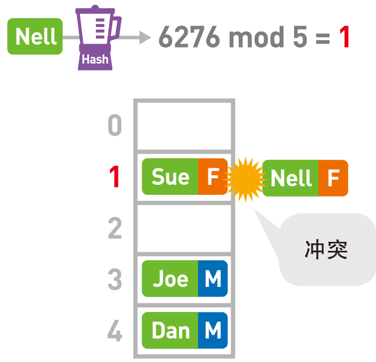
Sue键的哈希值为7291，mod 5的结果为1，将Sue的数据存进1号箱中。

15



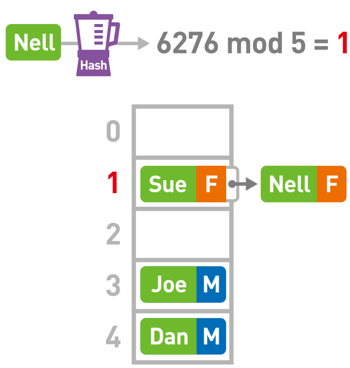
Dan键的哈希值为1539，mod 5的结果为4，将Dan的数据存进4号箱中。

16



Nell键的哈希值为6276，mod 5的结果为1。本应将其存进数组的1号箱中，但此时1号箱中已经存储了Sue的数据。这种存储位置重复了的情况便叫作“冲突”。

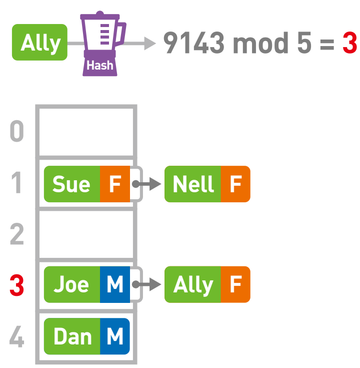
17



遇到这种情况，可使用链表在已有数据的后面继续存储新的数据。关于链表的详细说明请见1-2节。

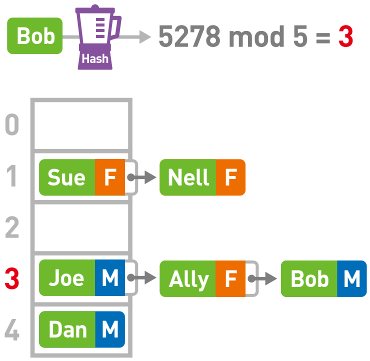
参考：1-2 链表

18



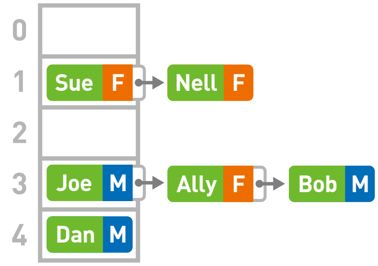
Ally键的哈希值为9143，mod 5的结果为3。本应将其存储在数组的3号箱中，但3号箱中已经有了Joe的数据，所以使用链表，在其后面存储Ally的数据。

19



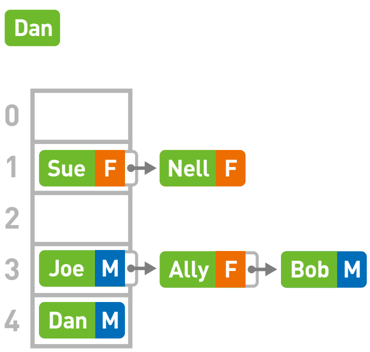
Bob键的哈希值为5278，mod 5的结果为3。本应将其存储在数组的3号箱中，但3号箱中已经有了Joe和Ally的数据，所以使用链表，在Ally的后面继续存储Bob的数据。

20



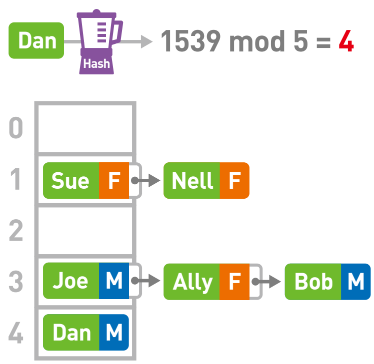
像这样存储完所有数据，哈希表也就制作完成了。

21



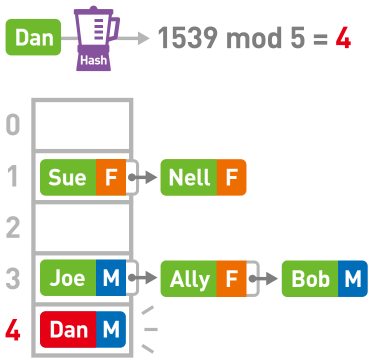
接下来讲解数据的查询方法。假设我们要查询Dan的性别。

22



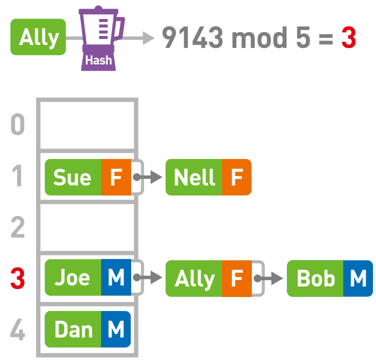
为了知道Dan存储在哪个箱子里，首先需要算出Dan键的哈希值，然后对其进行mod运算。最后得到的结果为4，于是我们知道了它存储在4号箱中。

23



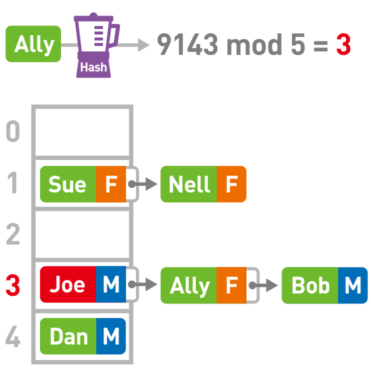
查看4号箱可知，其中的数据的键与Dan一致，于是取出对应的值。由此我们便知道了Dan的性别为男（M）。

24



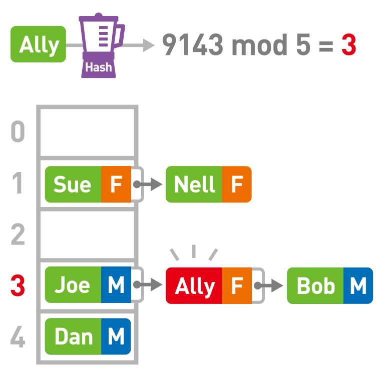
那么，想要查询Ally的性别时该怎么做呢？为了找到它的存储位置，先要算出Ally键的哈希值，再对其进行mod运算。最终得到的结果为3。

25



然而3号箱中数据的键是Joe而不是Ally。此时便需要对Joe所在的链表进行线性查找。

26



于是我们找到了键为Ally的数据。取出其对应的值，便知道了Ally的性别为女（F）。

解说

在哈希表中，我们可以利用哈希函数快速访问到数组中的目标数据。如果发生哈希冲突，就使用链表进行存储。这样一来，不管数据量为多少，我们都能够灵活应对。

如果数组的空间太小，使用哈希表的时候就容易发生冲突，线性查找的使用频率也会更高；反过来，如果数组的空间太大，就会出现很多空箱子，造成内存的浪费。因此，给数组设定合适的空间非常重要。

补充说明

在存储数据的过程中，如果发生冲突，可以利用链表在已有数据的后面插入新数据来解决冲突。这种方法被称为“链地址法”。

除了链地址法以外，还有几种解决冲突的方法。其中，应用较为广泛的是“开放地址法”。这种方法是指当冲突发生时，立刻计算出一个候补地址（数组上的位置）并将数据存进去。如果仍然有冲突，便继续计算下一个候补地址，直到有空地址为止。可以通过多次使用哈希函数或“线性探测法”等方法计算候补地址。

另外，本书在 5-3 节关于哈希函数的说明中将会提到“无法根据哈希值推算出原值”这个条件。不过，这只是在把哈希表应用于密码等安全方面时需要留意的条件，并不是使用哈希表时必须要遵守的规则。

因为哈希表在数据存储上的灵活性和数据查询上的高效性，编程语言的关联数组等也常常会使用它。

No. 1-7

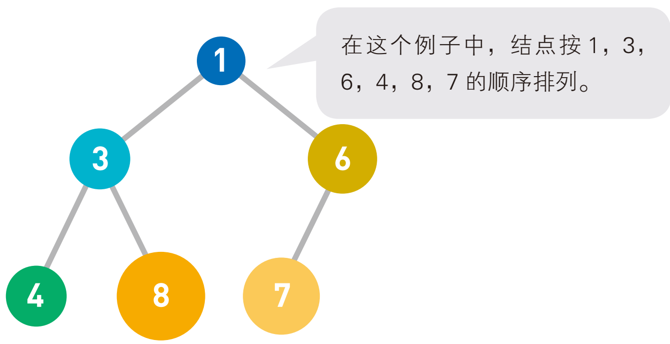
堆

堆是一种图的树形结构，被用于实现“优先队列”（priority queues）（树形结构的详细讲解在 4-2 节）。优先队列是一种数据结构，可以自由添加数据，但取出数据时要从最小值开始按顺序取出。在堆的树形结构中，各个顶点被称为“结点”（node），数据就存储在这些结点中。

▶参考：4-1 什么是图

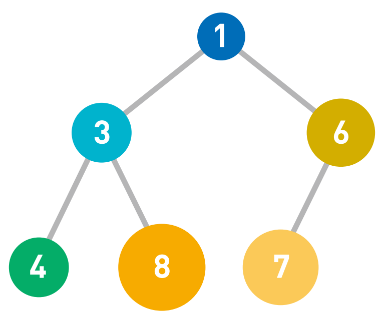
▶参考：4-2 广度优先搜索

01



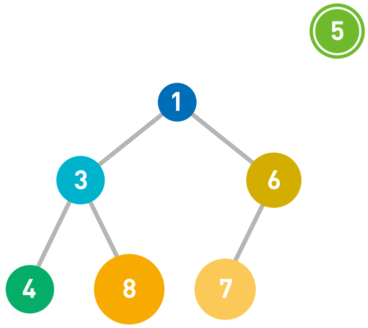
这就是堆的示例。结点内的数字就是存储的数据。堆中的每个结点最多有两个子结点。树的形状取决于数据的个数。另外，结点的排列顺序为从上到下，同一行里则为从左到右。

02



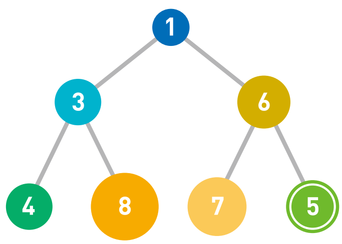
在堆中存储数据时必须遵守这样一条规则：子结点必定大于父结点。因此，最小值被存储在顶端的根结点中。往堆中添加数据时，为了遵守这条规则，一般会把新数据放在最下面一行靠左的位置。当最下面一行里没有多余空间时，就再往下另起一行，把数据加在这一行的最左端。

03



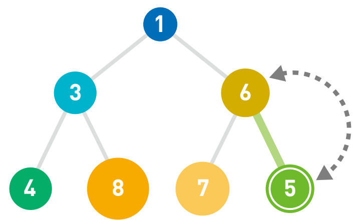
我们试试往堆里添加数字5。

04



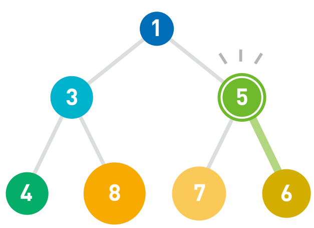
首先按照02的说明寻找新数据的位置。该图中最下面一排空着一个位置，所以将数据加在此处。

05



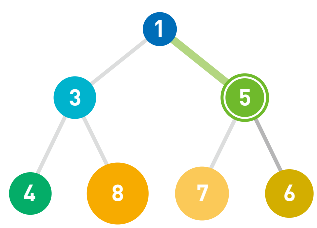
如果父结点大于子结点，则不符合上文提到的规则，因此需要交换父子结点的位置。

06



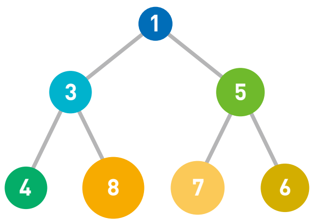
这里由于父结点的6大于子结点的5，所以交换了这两个数字。重复这样的操作直到数据都符合规则，不再需要交换为止。

07



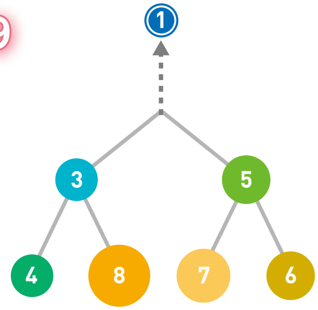
现在，父结点的1小于子结点的5，父结点的数字更小，所以不再交换。

08



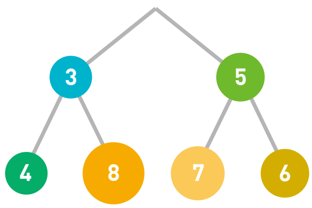
这样，往堆中添加数据的操作就完成了。

09



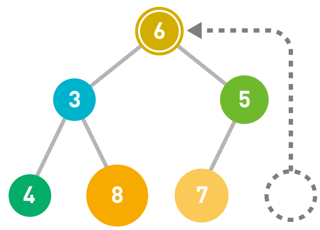
从堆中取出数据时，取出的是最上面的数据。这样，堆中就能始终保持最上面的数据最小。

10



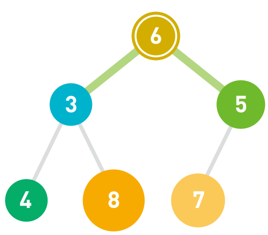
由于最上面的数据被取出，因此堆的结构也需要重新调整。

11



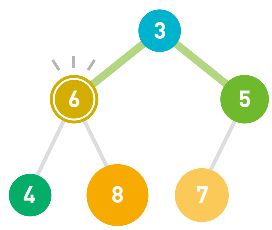
按照01中说明的排列顺序，将最后的数据（此处为6）移动到最顶端。

12



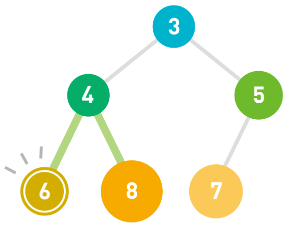
如果子结点的数字小于父结点的，就将父结点与其左右两个子结点中较小的一个进行交换。

13



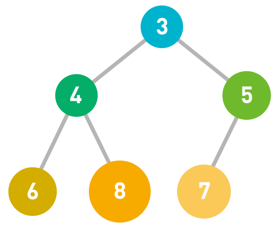
这里由于父结点的6大于子结点（右）的5大于子结点（左）的3，所以将左边的子结点与父结点进行交换。重复这个操作直到数据都符合规则，不再需要交换为止。

14



现在，子结点（右）的8大于父结点的6大于子结点（左）的4，需要将左边的子结点与父结点进行交换。

15



这样，从堆中取出数据的操作便完成了。

解说

堆中最顶端的数据始终最小，所以无论数据量有多少，取出最小值的时间复杂度都为 $O(1)$。

另外，因为取出数据后需要将最后的数据移到最顶端，然后一边比较它与子结点数据的大小，一边往下移动，所以取出数据需要的运行时间和树的高度成正比。假设数据量为 n，根据堆的形状特点可知树的高度为 $\log_2 n$，那么重构树的时间复杂度便为 $O(\log n)$。

添加数据也一样。在堆的最后添加数据后，数据会一边比较它与父结点数据的大小，一边往上移动，直到满足堆的条件为止，所以添加数据需要的运行时间与树的高度成正比，也是 $O(\log n)$。

应用示例

如果需要频繁地从管理的数据中取出最小值，那么使用堆来操作会非常方便。比如4-5节中提到的狄克斯特拉算法，每一步都需要从候补顶点中选择距离起点最近的那个顶点。此时，在顶点的选择上就可以用到堆。

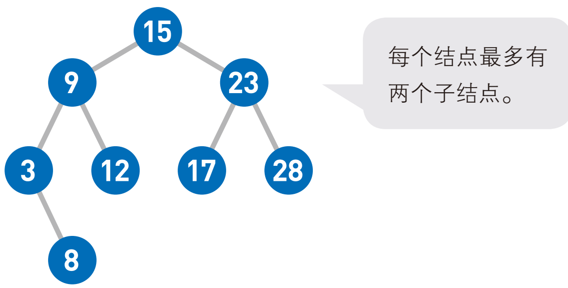
参考：4-5 狄克斯特拉算法

No. 1-8 二叉查找树

二叉查找树（又叫作二叉搜索树或二叉排序树）是一种数据结构，采用了图的树形结构（关于树形结构的详细说明请参考 4-2 节）。数据存储于二叉查找树的各个结点中。

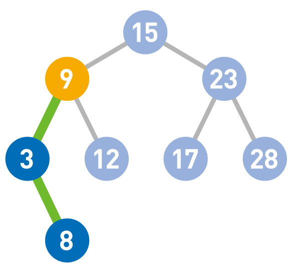
▶参考：4-1 什么是图
▶参考：4-2 广度优先搜索

01



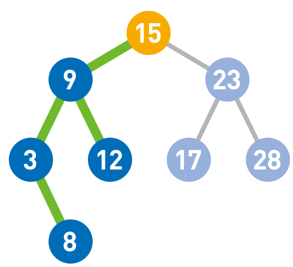
这就是二叉查找树的示例。结点中的数字便是存储的数据。此处以不存在相同数字为前提进行说明。

02



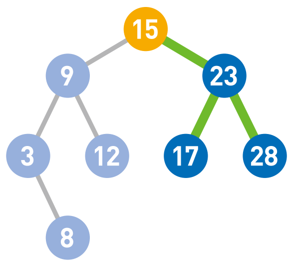
二叉查找树有两个性质。第一个是每个结点的值均大于其左子树上任意一个结点的值。比如结点9大于其左子树上的3和8。

03



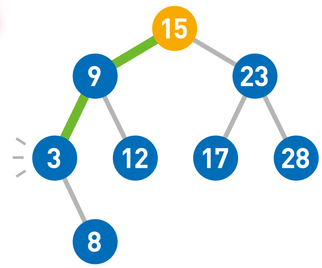
同样，结点15也大于其左子树上任意一个结点的值。

04



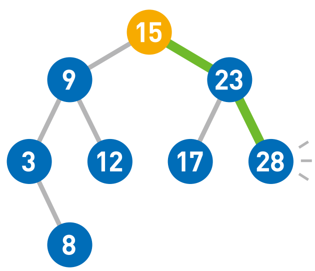
第二个是每个结点的值均小于其右子树上任意一个结点的值。比如结点15小于其右子树上的23、17和28。

05



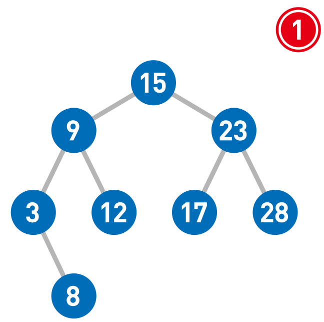
根据这两个性质可以得到以下结论。首先，二叉查找树的最小结点要从顶端开始，往其左下的末端寻找。此处最小值为3。

06



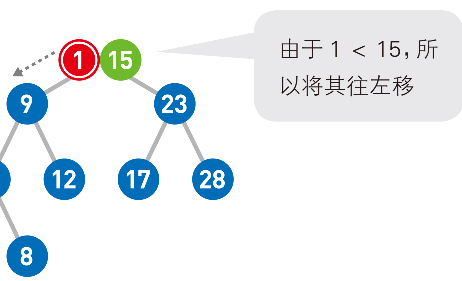
反过来，二叉查找树的最大结点要从顶端开始，往其右下的末端寻找。此处最大值为28。

07



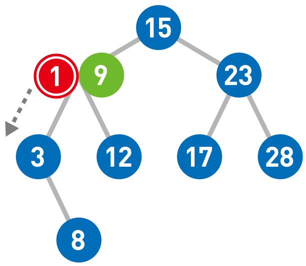
下面我们来试着往二叉查找树中添加数据。比如添加数字1。

08



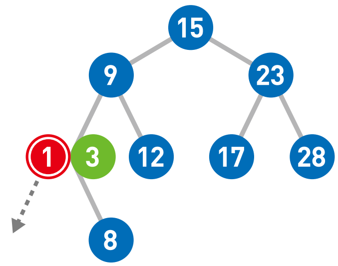
首先，从二叉查找树的顶端结点开始寻找添加数字的位置。将想要添加的1与该结点中的值进行比较，小于它则往左移，大于它则往右移。

09



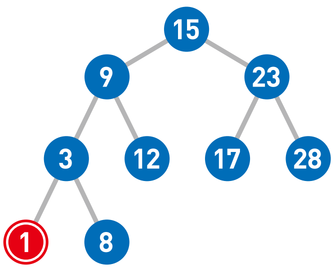
由于1 < 9，所以将1往左移。

10



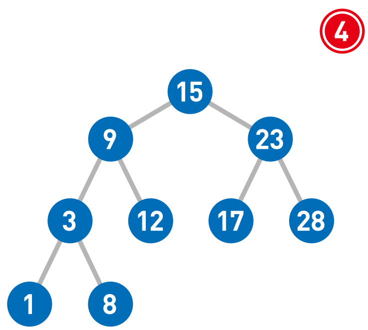
由于1 < 3，所以继续将1往左移，但前面已经没有结点了，所以把1作为新结点添加到左下方。

11



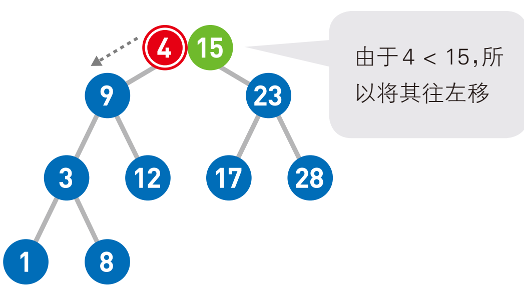
这样，1的添加操作便完成了。

12



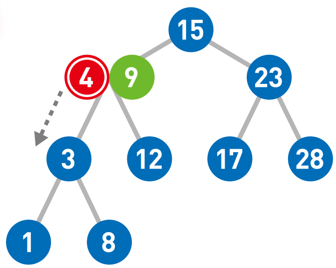
接下来，我们再试试添加数字4。

13



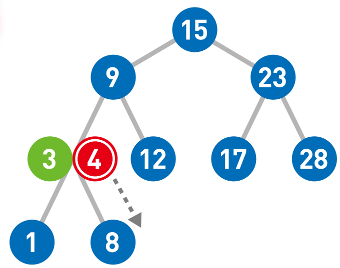
和前面的步骤一样，首先从二叉查找树的顶端结点开始寻找添加数字的位置。

14



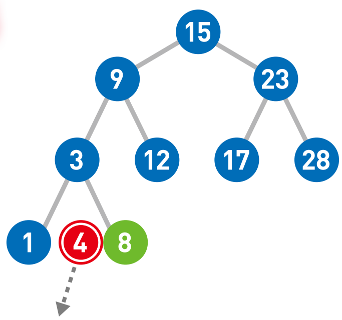
由于4 < 9，所以将其往左移。

15



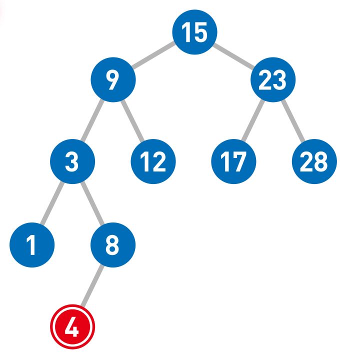
由于4 > 3，所以将其往右移。

16



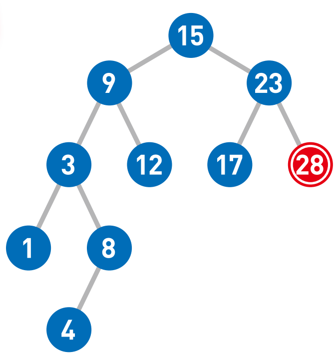
由于4 < 8，所以需要将其往左移，但前面已经没有结点了，所以把4作为新结点添加到左下方。

17



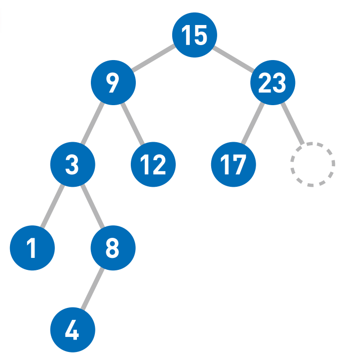
于是4的添加操作也完成了。

18



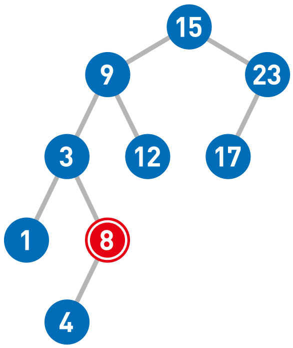
接下来看看如何在二叉查找树中删除结点。比如我们来试试删除结点28。

19



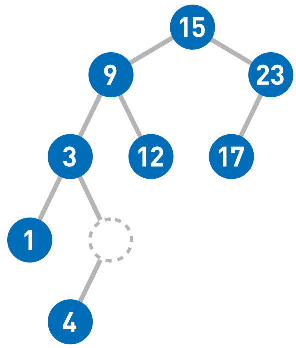
如果需要删除的结点没有子结点，直接删掉该结点即可。

20



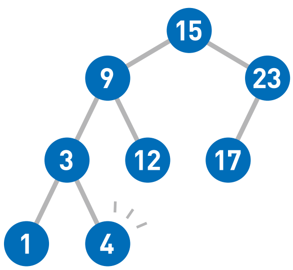
再试试删除结点8。

21



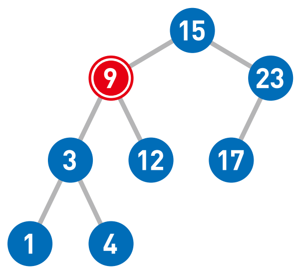
如果需要删除的结点只有一个子结点，那么先删掉目标结点……

22



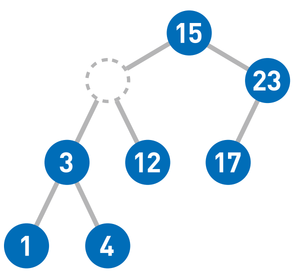
然后把子结点移到被删除结点的位置上即可。

23



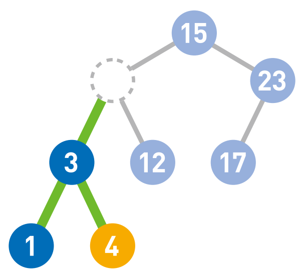
最后来试试删除结点9。

24



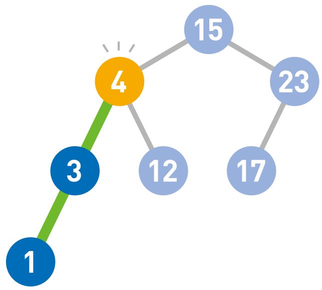
如果需要删除的结点有两个子结点，那么先删掉目标结点……

25



然后在被删除结点的左子树中寻找最大结点……

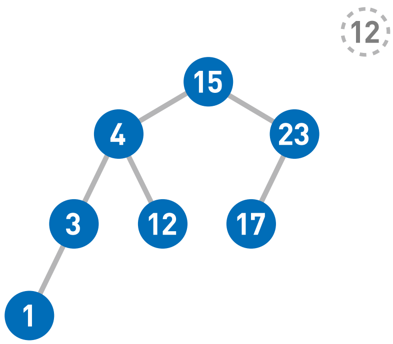
26



最后将最大结点移到被删除结点的位置上。这样一来，就能在满足二叉查找树性质的前提下删除结点了。如果需要移动的结点（此处为4）还有子结点，就递归执行前面的操作（关于递归，请参照7-4节的内容）。

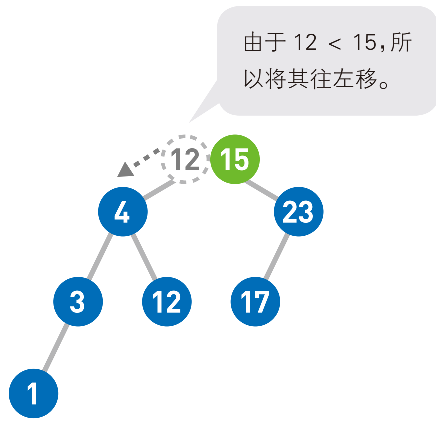
参考：7-4 汉诺塔

27



下面来看看如何在二叉查找树中查找结点。比如我们来试试查找12。

28

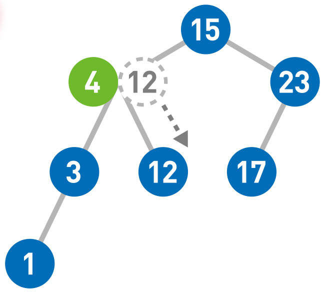


从二叉查找树的顶端结点开始往下查找。和添加数据时一样，把12和结点中的值进行比较，小于该结点的值则往左移，大于则往右移。

提示

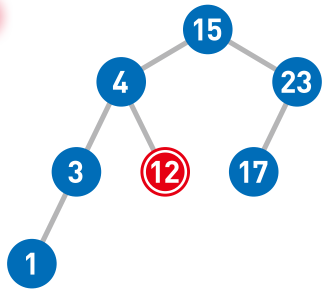
删除9的时候，我们将“左子树中的最大结点”移动到了删除结点的位置上，但是根据二叉查找树的性质可知，移动“右子树中的最小结点”也没有问题。

29



由于 12 > 4，所以往右移。

30



找到结点 12 了。

解说

我们可以把二叉查找树当作是二分查找算法思想的树形结构体现（二分查找的详细说明在 3-2 节）。因为它具有前面提到的那两个性质，所以在查找数据或寻找适合添加数据的位置时，只要将其和现有的数据比较大小，就可以根据比较结果得知该往哪边移动了。

比较的次数取决于树的高度。所以如果结点数为 n，而且树的形状又较为均衡的话，比较大小和移动的次数最多就是 $\log_2 n$。因此，时间复杂度为 $O(\log n)$。但是，如果树的形状朝单侧纵向延伸，树就会变得很高，此时时间复杂度也就变成了 $O(n)$。

补充说明

有很多以二叉查找树为基础扩展的数据结构，比如“平衡二叉查找树”。这种数据结构可以修正形状不均衡的树，让其始终保持均衡形态，以提高查找效率。

另外，虽然文中介绍的二叉查找树中一个结点最多有两个子结点，但我们可以把子结点数扩展为 m（m 为预先设定好的常数）。像这种子结点数可以自由设定，并且形状均衡的树便是 B 树。

第 2 章

排序

No. 2-1

什么是排序

将数字按从小到大的顺序排列

假设下面这张表是初中三年级学生的全国模拟考试成绩数据。

姓名	语文	数学	理科综合	社会	英语	合计
张秋	84	43	66	77	72	342
李优	87	64	88	91	65	395
赵野	49	48	71	67	78	313
……	……	……	……	……	……	……

要想知道考生的各科排名和综合排名，就需要按照各科成绩和总成绩的高低顺序对各行数据进行排列。

再比如积攒在电子邮箱里的邮件。

发件人▼	主题▼	收件时间▼
武小小	关于下次聚会	6/1/2017 10:05
田美丽	行程确认	5/30/2017 18:01
王野	昨天的谢礼	5/30/2017 9:39
小季酒店	关于订购商品	5/29/2017 11:45
董大海	Re: 有关算法的问题	5/25/2017 13:22
方帅	合作请求	5/24/2017 12:57
小季酒店	新酒已到货	5/21/2017 16:10
韩宏宇	研讨会通知	5/20/2017 15:02

点击“收件时间”，邮件就会按照收件时间的早晚排列；点击“发件人”，邮件就会按照发件人姓名的拼音顺序来排列。这是因为电子邮箱将收件时间和发件人都视为数字，点击后就会将它们按数字从小到大的顺序排好。

这种需要将数字按照大小排列的例子还有很多。在这种场景中能派上大用场的就是排序算法了。

所谓排序

排序就是将输入的数字按照从小到大的顺序进行排列。这里我们用柱形来表示数字，数字越大，柱形就越高。

假设现在有如上图所示的输入数据，那么我们的目标就是将它们像下图一样，按从小到大的顺序从左边开始依次排列。

如果只有 10 个数字，手动排序也能轻松完成，但如果有 10 000 个数据，排序就不那么容易了。这时，使用高效率的排序算法便是解决问题的关键。

各种各样的排序算法

由于排序是一个比较基础的问题，所以排序算法的种类也比较多。本章将在接下来的几节中对各种排序算法进行介绍。在接下来的说明中，输入的数字个数都设定为 n。为了便于讲解，同一个例子中不会出现相同的数字，但实际上，即使有相同的数字，算法依然可以正常运行。

No.
2-2 冒泡排序

冒泡排序就是重复“从序列右边开始比较相邻两个数字的大小，再根据结果交换两个数字的位置”这一操作的算法。在这个过程中，数字会像泡泡一样，慢慢从右往左“浮”到序列的顶端，所以这个算法才被称为“冒泡排序”。

01

在序列的最右边放置一个天平，比较天平两边的数字。如果右边的数字较小，就交换这两个数字的位置。

02

由于6 < 7，所以交换这两个数字。

03

完成后，天平往左移动一个位置，比较两个数字的大小。此处4 < 6，所以无须交换。

04

继续将天平往左移动一个位置并比较数字。重复同样的操作直到天平到达序列最左边为止。

05

不断对数字进行交换，天平最终到达了最左边。通过这一系列操作，序列中最小的数字就会移动到最左边。

06

最左边的数字已经归位。

07

将天平移回最右边，然后重复之前的操作，直到天平到达左边第2个位置为止。

08

当天平到达左边第2个位置时，序列中第2小的数字也就到达了指定位置。

09

将天平再次移回最右边，重复同样的操作直到所有数字都归位为止。

10

排序中……

11

排序中……

12

排序完成。

解说

在冒泡排序中，第 1 轮需要比较 $n-1$ 次，第 2 轮需要比较 $n-2$ 次……第 $n-1$ 轮需要比较 1 次。因此，总的比较次数为 $(n-1)+(n-2)+\cdots+1 \approx n^2/2$。这个比较次数恒定为该数值，和输入数据的排列顺序无关。

不过，交换数字的次数和输入数据的排列顺序有关。假设出现某种极端情况，如输入数据正好以从小到大的顺序排列，那么便不需要任何交换操作；反过来，输入数据要是以从大到小的顺序排列，那么每次比较数字后便都要进行交换。因此，冒泡排序的时间复杂度为 $O(n^2)$。

No. 2-3

选择排序

选择排序就是重复“从待排序的数据中寻找最小值，将其与序列最左边的数字进行交换”这一操作的算法。在序列中寻找最小值时使用的是线性查找。

▶参考：3-1 线性查找

01

对数字 1~9 进行排序。

02

使用线性查找在数据中寻找最小值，于是我们找到了最小值 1（线性查找的详细说明在 3-1 节）。

▶参考：3-1 线性查找

03

将最小值 1 与序列最左边的 6 进行交换，最小值 1 归位。不过，如果最小值已经在最左端，就不需要任何操作。

04

在余下的数据中继续寻找最小值。这次我们找到了最小值2。

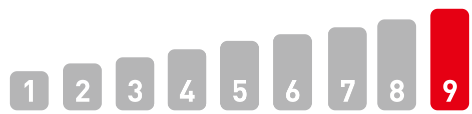
05

将数字2与左边第2个数字6进行交换，最小值2归位。

06

重复同样的操作直到所有数字都归位为止。

07



排序完成。

解说

选择排序使用了线性查找来寻找最小值，因此在第 1 轮中需要比较 $n-1$ 个数字，第 2 轮需要比较 $n-2$ 个数字……到第 $n-1$ 轮的时候就只需比较 1 个数字了。因此，总的比较次数与冒泡排序的相同，都是 $(n-1)+(n-2)+\cdots+1 \approx n^2/2$ 次。

每轮中交换数字的次数最多为 1 次。如果输入数据就是按从小到大的顺序排列的，便不需要进行任何交换。选择排序的时间复杂度也和冒泡排序的一样，都为 $O(n^2)$。

No.
2-4 插入排序

插入排序是一种从序列左端开始依次对数据进行排序的算法。在排序过程中，左侧的数据陆续归位，而右侧留下的就是还未被排序的数据。插入排序的思路就是从右侧的未排序区域内取出一个数据，然后将它插入到已排序区域内合适的位置上。

01

此处同样对数字1~9进行排序。

02

首先，我们假设最左边的数字5已经完成排序，所以此时只有5是已归位的数字。

03

接下来，从待排数字（未排序区域）中取出最左边的数字3，将它与左边已归位的数字进行比较。若左边的数字更大，就交换这两个数字。重复该操作，直到左边已归位的数字比取出的数字更小，或者取出的数字已经被移到整个序列的最左边为止。

04

由于5 > 3，所以交换这两个数字。

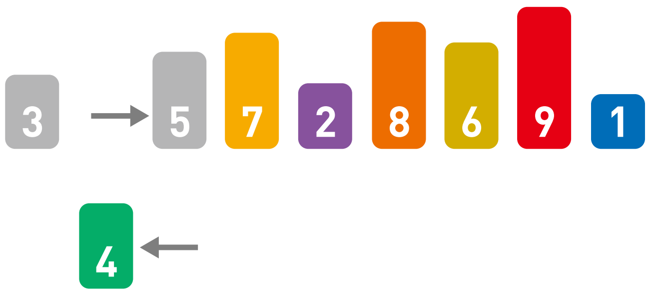
05

对数字3的操作到此结束。此时3和5已归位，还剩下右边7个数字尚未排序。

06

接下来是第3轮。和前面一样，取出未排序区域中最左边的数字4，将它与左边的数字5进行比较。

07



由于5 > 4，所以交换这两个数字。交换后再把4和左边的3进行比较，发现3 < 4，因为出现了比自己小的数字，所以操作结束。

08

于是4也归位了。此时3、4、5都已归位，已排序区域也得到了扩大。

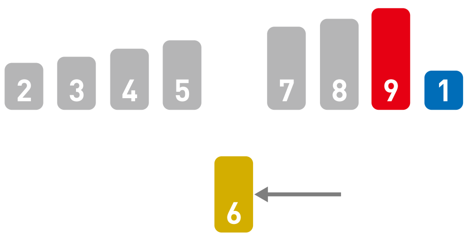
09

遇到左边的数字都比自己小的情况时……

10

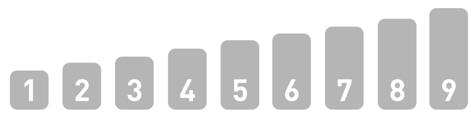
不需要任何操作即可完成排序。

11



重复上述操作，直到所有数字都归位。

12



对所有数字的操作都结束时，排序也就完成了。

解说

在插入排序中，需要将取出的数据与其左边的数字进行比较。就跟前面讲的步骤一样，如果左边的数字更小，就不需要继续比较，本轮操作到此结束，自然也不需要交换数字的位置。

然而，如果取出的数字比左边已归位的数字都要小，就必须不停地比较大小，交换数字，直到它到达整个序列的最左边为止。具体来说，就是第 k 轮需要比较 $k-1$ 次。因此，在最糟糕的情况下，第 2 轮需要操作 1 次，第 3 轮操作 2 次……第 n 轮操作 $n-1$ 次，所以时间复杂度和冒泡排序的一样，都为 $O(n^2)$。

和前面讲的排序算法一样，输入数据按从大到小的顺序排列时就是最糟糕的情况。

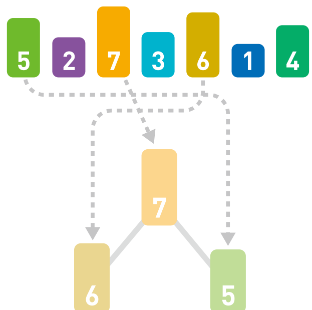
No. 2-5

堆排序

堆排序的特点是利用了数据结构中的堆。关于堆的详细说明在 1-7 节。

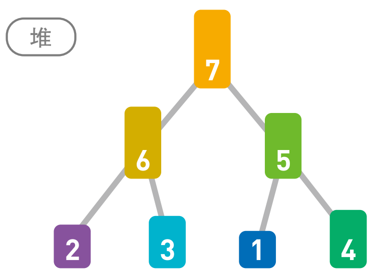
参考：1-7 堆

01



首先，在堆中存储所有的数据，并按降序来构建堆。

02

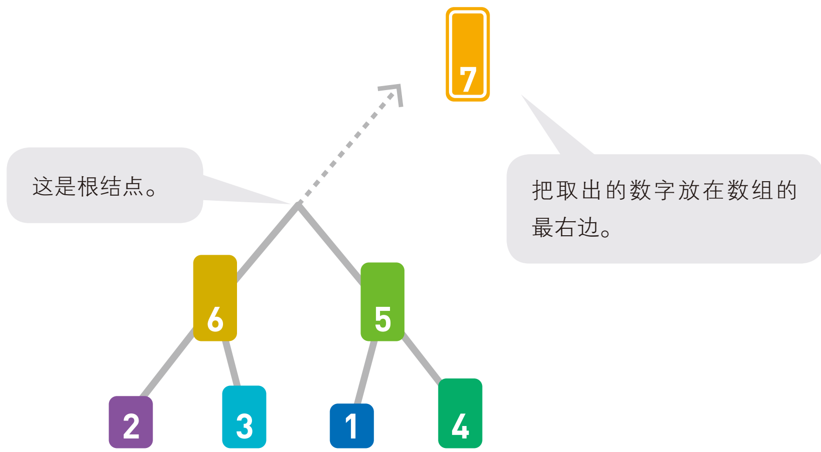


现在，所有数据都存进堆里了。为了排序，需要再从堆中把数据一个个取出来。

提示

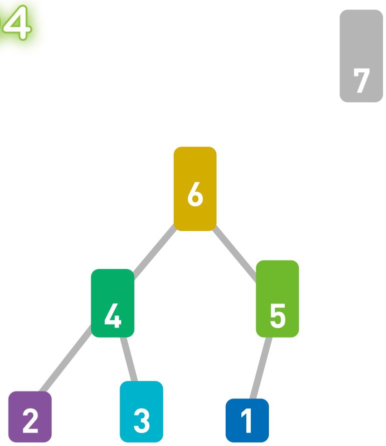
从降序排列的堆中取出数据时会从最大的数据开始取，所以将取出的数据反序输出，排序就完成了。

03



我们来试一试吧。首先取出根结点的数字7。

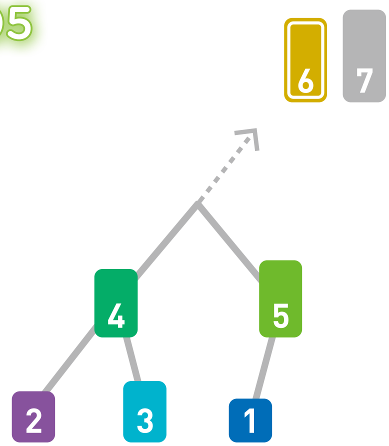
04



重新构造堆。重构的规则请参考1-7节的内容。

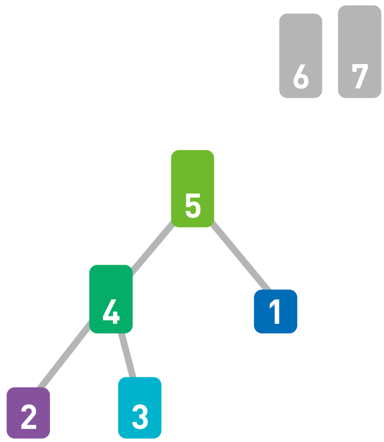
参考：1-7 堆

05



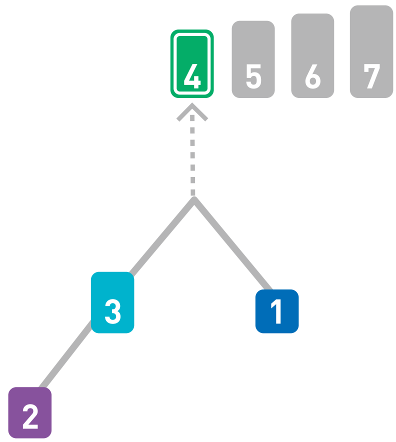
同样地，取出根结点的数字6，将它放在右数第2个位置上。

06



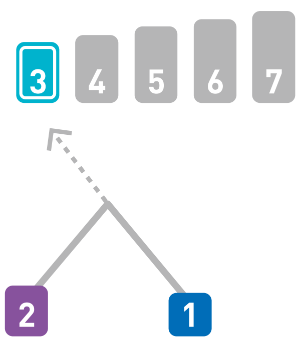
重新构造堆。

07



重复上述操作直到堆变空为止。

08



排序中……

09

从堆中取出了所有数字，排序完成。

解说

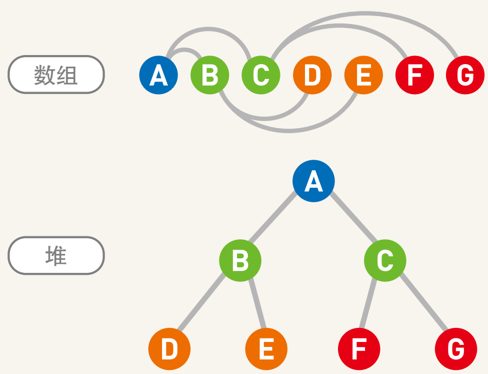
堆排序一开始需要将 n 个数据存进堆里，所需时间为 $O(n\log n)$。排序过程中，堆从空堆的状态开始，逐渐被数据填满。由于堆的高度小于 $\log_2 n$，所以插入 1 个数据所需要的时间为 $O(\log n)$。

每轮取出最大的数据并重构堆所需要的时间为 $O(\log n)$。由于总共有 n 轮，所以重构后排序的时间也是 $O(n\log n)$。因此，整体来看堆排序的时间复杂度为 $O(n\log n)$。

这样来看，堆排序的运行时间比之前讲到的冒泡排序、选择排序、插入排序的时间 $O(n^2)$ 都要短，但由于要使用堆这个相对复杂的数据结构，所以实现起来也较为困难。

补充说明

一般来说，需要排序的数据都存储在数组中。这次我们使用了堆这种数据结构，但实际上，这也相当于将堆嵌入到包含了序列的数组中，然后在数组中通过交换数据来进行排序。具体来说，就是让堆中的各结点和数组像下图这样呈对应关系。正如大家所见，这可以说是强行在数组中使用了堆结构。



No. 2-6

归并排序

归并排序算法会把序列分成长度相同的两个子序列，当无法继续往下分时（也就是每个子序列中只有一个数据时），就对子序列进行归并。归并指的是把两个排好序的子序列合并成一个有序序列。该操作会一直重复执行，直到所有子序列都归并为一个整体为止。

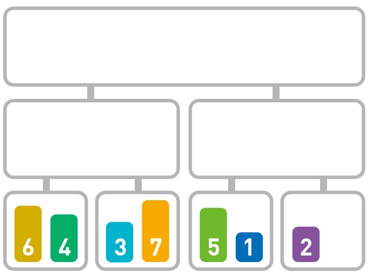
01

首先，要把序列对半分割。

02

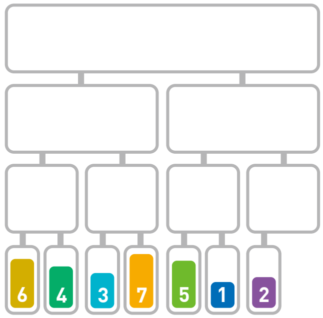
先分成两段……

03



再继续往下分……

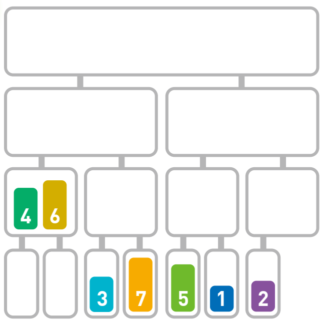
04



合并时需要将数字按从小到大的顺序排列。

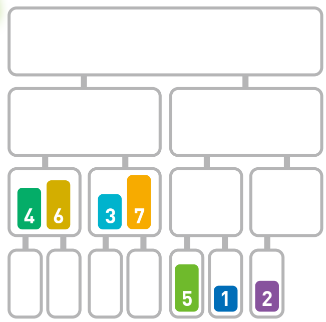
分割完毕。接下来对分割后的元素进行合并。

05



把6和4合并，合并后的顺序为[4, 6]。

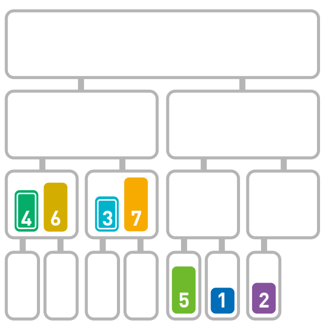
06



接下来把3和7合并，合并后的顺序为[3, 7]。

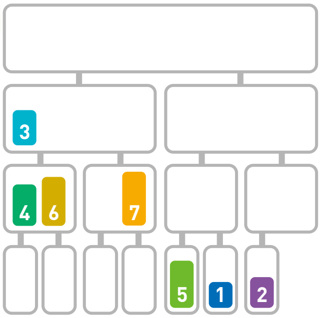
07

此时要比较两个子序列的首位数字4和3。



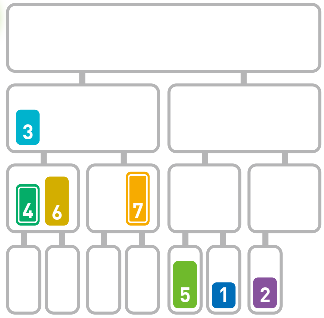
下面，我们来看看怎么合并[4, 6]和[3, 7]。合并这种含有多个数字的子序列时，要先比较首位数字，再移动较小的数字。

08



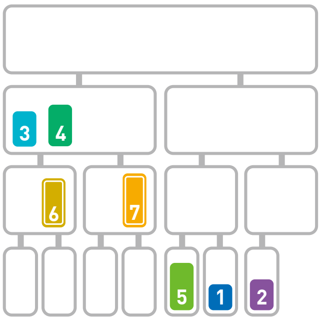
由于4 > 3，所以移动3。

09



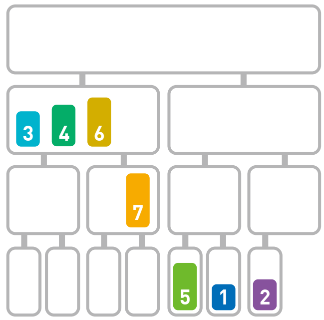
同样地，再次比较序列中剩下的首位数字。

10



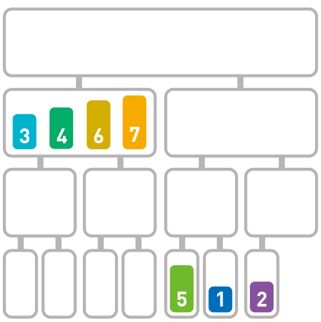
由于4 < 7，所以移动4。

11



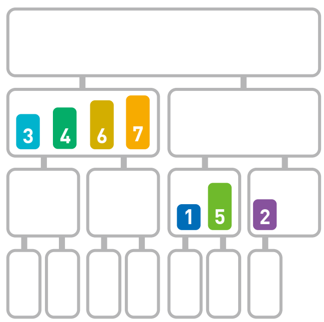
由于6 < 7，所以移动6。

12



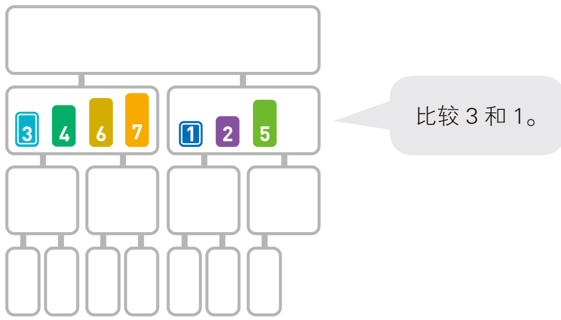
最后移动剩下的7。

13



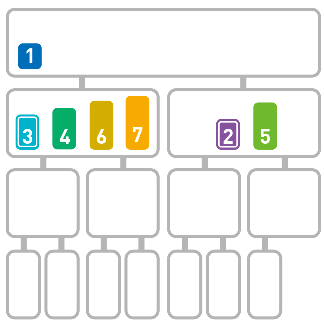
递归执行上面的操作，直到所有的数字都合为一个整体为止。

14



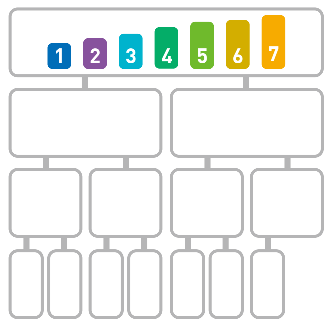
这里也要比较两个子序列中的首位数字。

15



由于3 > 1，所以移动1。继续操作……

16



合并完成，序列的排序也就完成了。

解说

归并排序中，分割序列所花费的时间不算在运行时间内（可以当作序列本来就是分割好的）。在合并两个已排好序的子序列时，只需重复比较首位数据的大小，然后移动较小的数据，因此只需花费和两个子序列的长度相应的运行时间。也就是说，完成一行归并所需的运行时间取决于这一行的数据量。

看一下上面的图便能得知，无论哪一行都是 n 个数据，所以每行的运行时间都为 $O(n)$。而将长度为 n 的序列对半分割直到只有一个数据为止时，可以分成 $\log_2 n$ 行，因此，总共有 $\log_2 n$ 行。也就是说，总的运行时间为 $O(n\log n)$，这与前面讲到的堆排序相同。

No.
2-7 快速排序

快速排序算法首先会在序列中随机选择一个基准值（pivot），然后将除了基准值以外的数分为“比基准值小的数”和“比基准值大的数”这两个类别，再将其排列成以下形式。

[比基准值小的数] 基准值 [比基准值大的数]

接着，对两个“[ ]”中的数据进行排序之后，整体的排序便完成了。对“[ ]”里面的数据进行排序时同样也会使用快速排序。

01

下面我们就来看看快速排序的步骤。

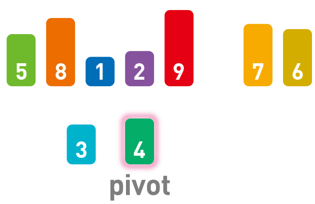
02

在序列中随机选择一个基准值。这里选择了4。

03

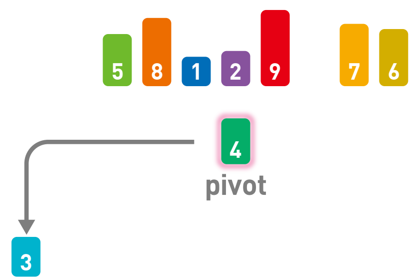
将其他数字和基准值进行比较。小于基准值的往左移，大于基准值的往右移。

04



首先，比较3和基准值4。

05

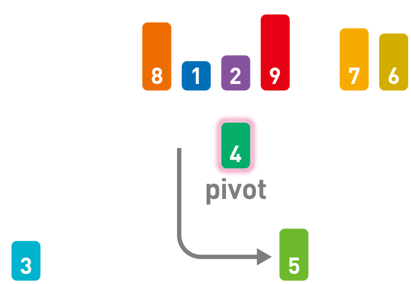


因为3 < 4，所以将3往左移。

06

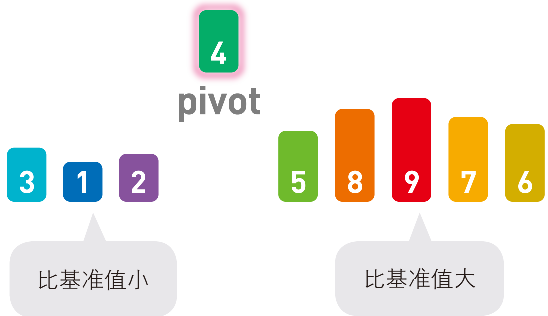
接下来，比较5和基准值4。

07



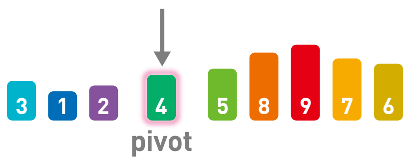
因为5 > 4，所以将5往右移。

08



对其他数字也进行同样的操作，最终结果如上图所示。

09

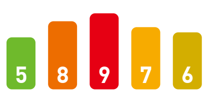


把基准值4插入序列。这样，4左边就是比它小的数字，右边就是比它大的数字。

10

分别对左边和右边的数据进行排序后，就能完成整体的排序。

11

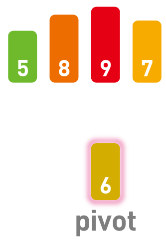


两边的排序操作也和前面的一样。首先来看看如何对右边的数据进行排序吧。

12

随机选择一个基准值。这次选择6。

13

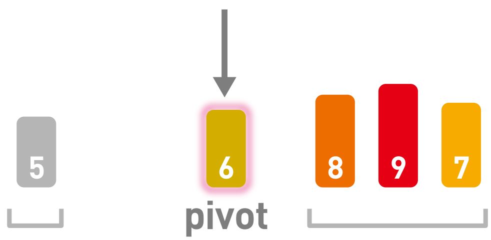


把其余数据分别和基准值6进行比较，小于基准值的就往左移，大于的就往右移。

14

完成了大小比较和位置移动。

15



和前面一样，对左右两边分别进行排序，进而完成整体排序。但是此时左边只有5，所以已经是排序完成的状态，不需要任何操作。而右边就和前面一样，先选出基准值。

16

选择8作为基准值。

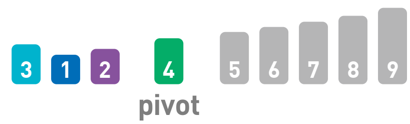
17

将9和7分别与基准值8进行比较后，两个数字的位置便分好了。8的两边都只有一个数据，因此不需要任何操作。这样7、8、9便完成排序了。

18

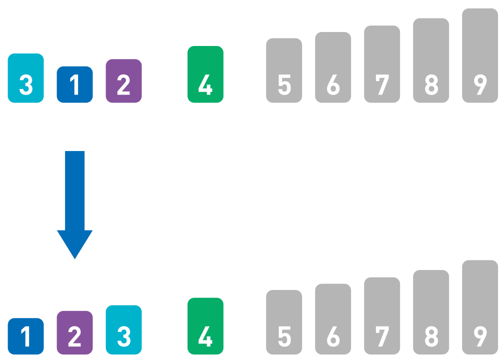
回到上一行，由于7、8、9完成了排序，所以5、6、7、8、9也完成了排序。

19



于是，最初选择的基准值4的右边排序完毕。

20



左边也以相同的操作进行排序，整体的排序工作也就完成了。

补充说明

快速排序是一种“分治法”。它将原本的问题分成两个子问题（比基准值小的数和比基准值大的数），然后再分别解决这两个问题。子问题，也就是子序列完成排序后，再像一开始说明的那样，把他们合并成一个序列，那么对原始序列的排序也就完成了。

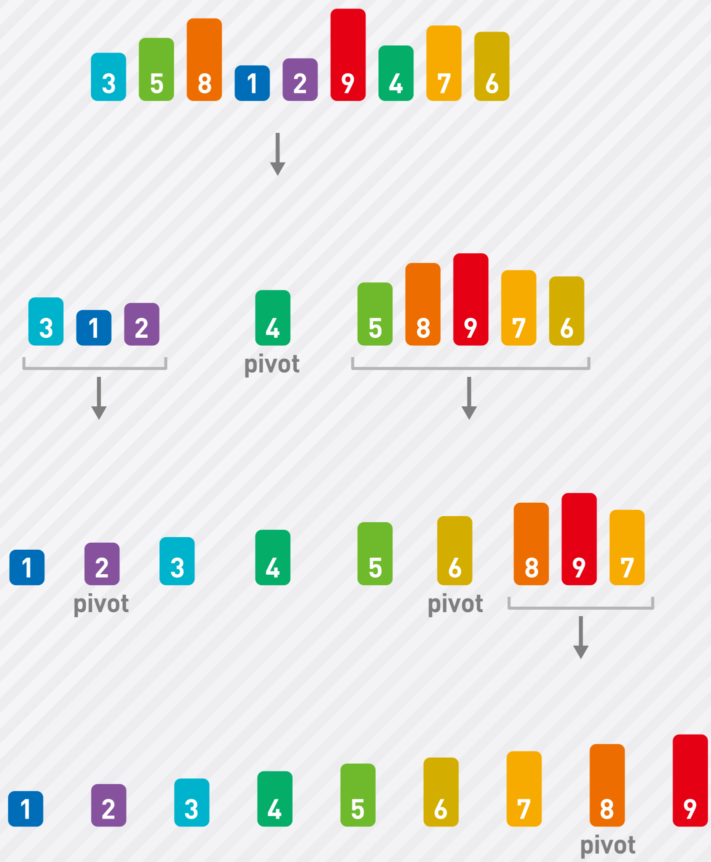
不过，解决子问题的时候会再次使用快速排序，甚至在这个快速排序里仍然要使用快速排序。只有在子问题里只剩一个数字的时候，排序才算完成。

像这样，在算法内部继续使用该算法的现象被称为“递归”。关于递归的详细说明在 7-4 节。实际上前一节中讲到的归并排序也可看作是一种递归的分治法。

参考：7-4 汉诺塔

解说

分割子序列时需要选择基准值，如果每次选择的基准值都能使得两个子序列的长度为原本的一半，那么快速排序的运行时间和归并排序的一样，都为 $O(n\log n)$。和归并排序类似，将序列对半分割 $\log_2 n$ 次之后，子序列里便只剩下一个数据，这时子序列的排序也就完成了。因此，如果像下图这样一行行地展现根据基准值分割序列的过程，那么总共会有 $\log_2 n$ 行。



每行中每个数字都需要和基准值比较大小，因此每行所需的运行时间为 $O(n)$。由此可知，整体的时间复杂度为 $O(n\log n)$。

如果运气不好，每次都选择最小值作为基准值，那么每次都需要把其他数据移到基准值的右边，递归执行 n 行，运行时间也就成了 $O(n^2)$。这就相当于每次都选出最小值并把它移到了最左边，这个操作也就和选择排序一样了。此外，如果数据中的每个数字被选为基准值的概率都相等，那么需要的平均运行时间为 $O(n\log n)$。

第 3 章

数组的查找

No. 3-1 线性查找

线性查找是一种在数组中查找数据的算法（关于数组的详细讲解在1-3节）。与将在3-2节中讲解的二分查找不同，即便数据没有按顺序存储，也可以应用线性查找。线性查找的操作很简单，只要在数组中从头开始依次往下查找即可。虽然存储的数据类型没有限制，但为了便于理解，这里我们假设存储的是整数。

▶参考：1-3 数组

01

来试试查找数字6吧。

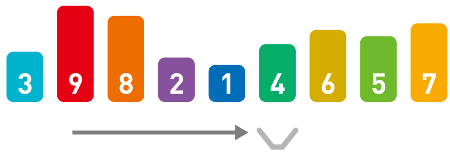
02

首先，检查数组中最左边的数字，将其与6进行比较。如果结果一致，查找便结束，不一致则向右检查下一个数字。

03

此处不一致，所以向右检查下一个数字。

04



重复上面的操作直到找到数字6为止。

05

找到6了，查找结束。

解说

线性查找需要从头开始不断地按顺序检查数据，因此在数据量大且目标数据靠后，或者目标数据不存在时，比较的次数就会更多，也更为耗时。若数据量为 n，线性查找的时间复杂度便为 $O(n)$。

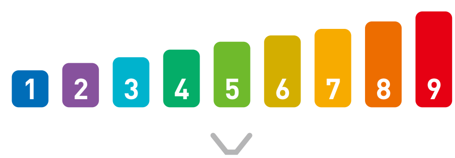
No. 3-2 二分查找

二分查找也是一种在数组中查找数据的算法。和 3-1 节讲到的线性查找不同，它只能查找已经排好序的数据。二分查找通过比较数组中间的数据与目标数据的大小，可以得知目标数据是在数组的左边还是右边。因此，比较一次就可以把查找范围缩小一半。重复执行该操作就可以找到目标数据，或得出目标数据不存在的结论。

01

还是来试试查找数字6吧。

02

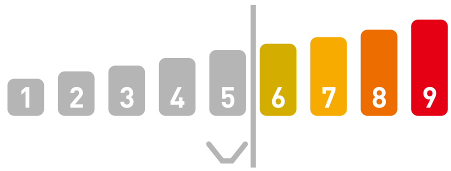


首先找到数组中间的数字，此处为5。

03

将5和要查找的数字6进行比较。

04



把不需要的数字移出查找范围。

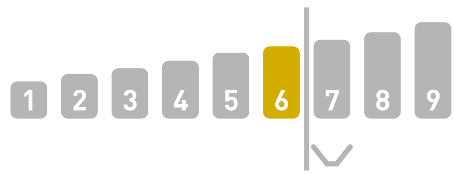
05

在剩下的数组中找到中间的数字，此处为7。

06

比较7和6。

07

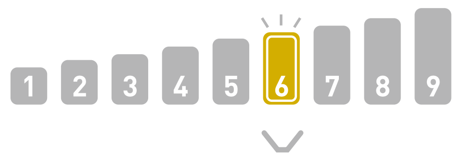


把不需要的数字移出查找范围。

08

在剩下的数组中找到中间的数字，此处为6。

09



6=6，成功找到目标数字。

解说

二分查找利用已排好序的数组，每一次查找都可以将查找范围减半。查找范围内只剩一个数据时查找结束。

数据量为 n 的数组，将其长度减半 $\log_2 n$ 次后，其中便只剩一个数据了。也就是说，在二分查找中重复执行“将目标数据和数组中间的数据进行比较后将查找范围减半”的操作 $\log_2 n$ 次后，就能找到目标数据（若没找到则可以得出数据不存在的结论），因此它的时间复杂度为 $O(\log n)$。

补充说明

二分查找的时间复杂度为 $O(\log n)$，与线性查找的 $O(n)$ 相比速度上得到了指数倍提高（$x=\log_2 n$，则 $n=2^x$）。

但是，二分查找必须建立在数据已经排好序的基础上才能使用，因此添加数据时必须加到合适的位置，这就需要额外耗费维护数组的时间。

而使用线性查找时，数组中的数据可以是无序的，因此添加数据时也无须顾虑位置，直接把它加在末尾即可，不需要耗费时间。

综上，具体使用哪种查找方法，可以根据查找和添加两个操作哪个更为频繁来决定。

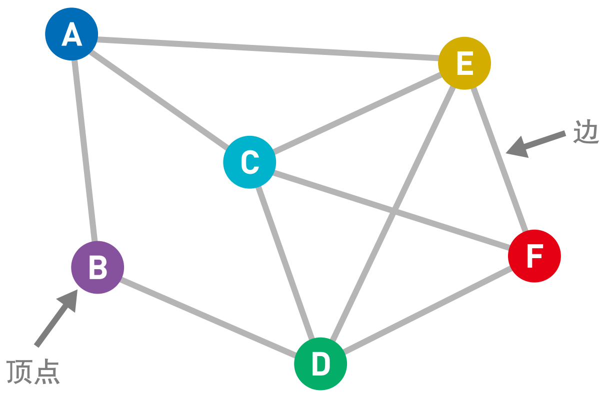
第 4 章

图的搜索

No.
4-1 什么是图

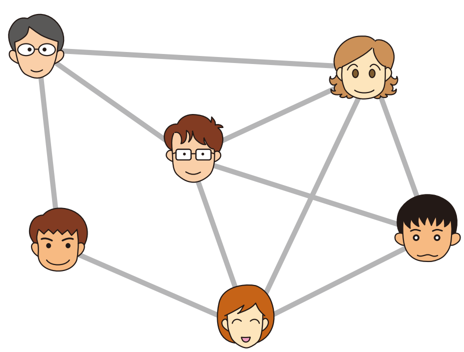
离散数学中的图

说到“图”，可能大部分人想到的是饼状图、柱状图，或者数学中 $y=f(x)$ 所呈现的图，而计算机科学或离散数学中说的“图”却是下面这样的。



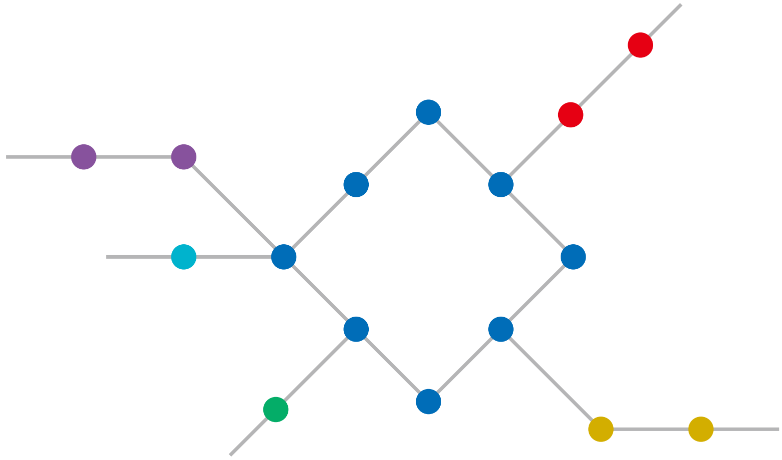
上图中的圆圈叫作“顶点”（也叫“结点”），连接顶点的线叫作“边”。也就是说，由顶点和连接每对顶点的边所构成的图形就是图。

图可以表现各种关系

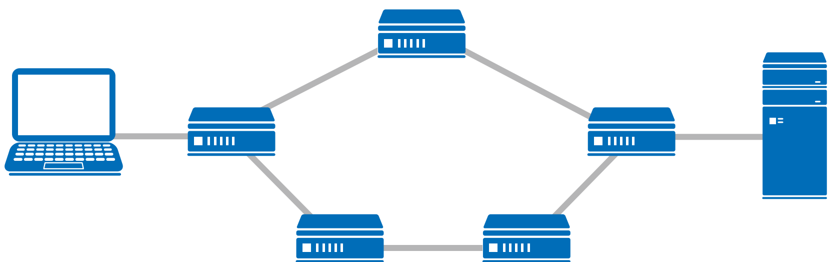


图可以表现社会中的各种关系，使用起来非常方便。假设我们要开一个派对，将参加人员作为顶点，把互相认识的人用边连接，就能用图来表现参加人员之间的人际关系了。

再举个例子，若将车站作为顶点，将相邻两站用边连接，就能用图来表现地铁的路线了。



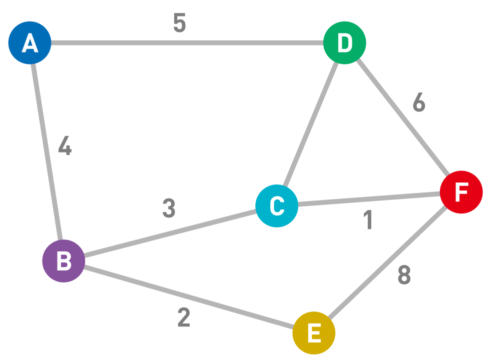
另外，还可以在计算机网络中把路由器作为顶点，将相互连接的两个路由器用边连接，这样就能用图来表现网络的连接关系了。



加权图

上面讲到的都是由顶点和边构成的图，而我们还可以给边加上一个值。

这个值叫作边的“权重”或者“权”，加了权的图被称为“加权图”。没有权的边只能表示两个顶点的连接状态，而有权的边就可以表示顶点之间的“连接程度”。



这个“程度”是什么意思呢？根据图的内容不同，“程度”表示的意思也不同。比如在计算机网络中，给两台路由器之间的边加上传输数据所需要的时间，这张图就能表示网络之间的通信时间了。

而在路线图中，如果把地铁在两个车站间行驶的时间加在边上，这张图就能表现整个路线的移动时间；如果把两个车站间的票价加在边上，就能表现乘车费了。虽然在一些情况下顶点也可以有权重，但本书中并不涉及这类情况，故此处忽略。

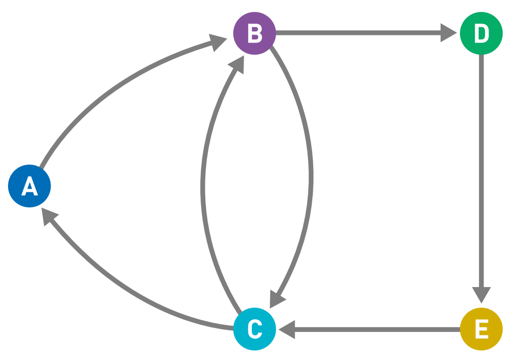
有向图

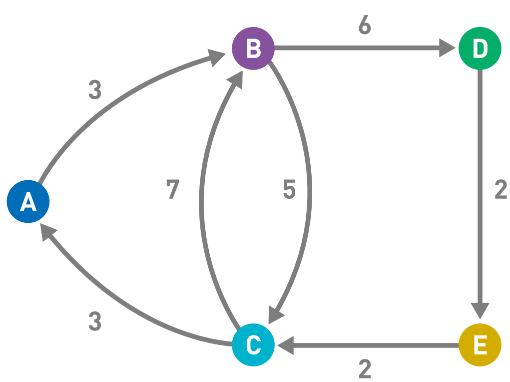
当我们想在路线图中表示该路线只能单向行驶时，就可以给边加上箭头，而这样的图就叫作“有向图”。比如网页里的链接也是有方向性的，用有向图来表示就会很方便。

与此相对，边上没有箭头的图便是“无向图”。

右图中我们可以从顶点 A 到顶点 B，但不能直接从 B 到 A，而 B 和 C 之间有两条边分别指向两个方向，因此可以双向移动。

和无向图一样，有向图的边也可以加上权重。





在上图中，从顶点 B 到顶点 C 的权重为 5，而从 C 到 B 的权重为 7。如果做的是一个表示移动时间的图，而从 B 到 C 是下坡路，就有可能出现这样的情况。就像这样，使用有向图还可以设置非对称的权重。

图能给我们带来哪些便利

想一想图能给我们带来的好处吧。假设图中有两个顶点 s 和 t，而我们设计出了一种算法，可以找到“从 s 到 t 的权重之和最小”的那条路径。

那么，这种算法就可以应用到这些问题上：寻找计算机网络中通信时间最短的路径，寻找路线图中耗时最短的路径，寻找路线图中最省乘车费的路径等[①]。

就像这样，只要能用图来表示这些关系，我们就可以用解决图问题的算法来解决这些看似不一样的问题。

本章的知识点

本章将要学习的是图的搜索算法，和可以解决图的基本问题——最短路径问题的算法。

图的搜索指的就是从图的某一顶点开始，通过边到达不同的顶点，最终找到目标顶点的过程。根据搜索的顺序不同，图的搜索算法可分为“广度优先搜索”和“深度优先搜索”这两种。

最短路径问题和前文提到的一样，就是要在从 s 到 t 的路径中，找到一条所经过的边的权重总和最小的路径。

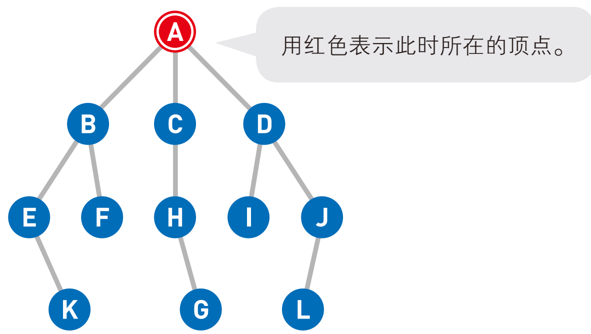
---

① 现实中的情况会稍有不同，因为换乘地铁也需要一定的时间，而且乘车费也不是按各站之间票价的总和来计算的。

No.
4-2 广度优先搜索

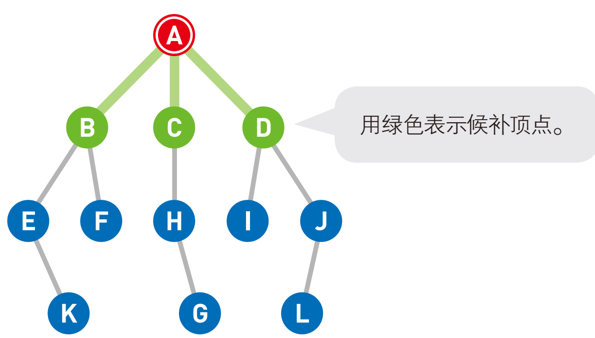
广度优先搜索是一种对图进行搜索的算法。假设我们一开始位于某个顶点（即起点），此时并不知道图的整体结构，而我们的目的是从起点开始顺着边搜索，直到到达指定顶点（即终点）。在此过程中每走到一个顶点，就会判断一次它是否为终点。广度优先搜索会优先从离起点近的顶点开始搜索。

01



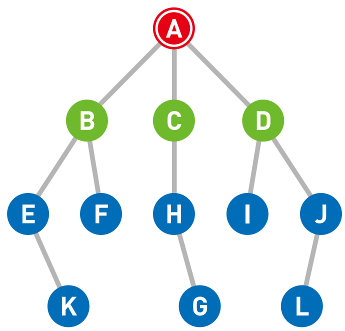
A为起点，G为终点。一开始我们在起点A上，此时并不知道G在哪里。

02



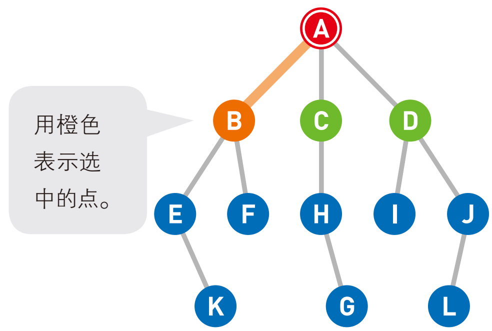
将可以从A直达的三个顶点B、C、D设为下一步的候补顶点。

03



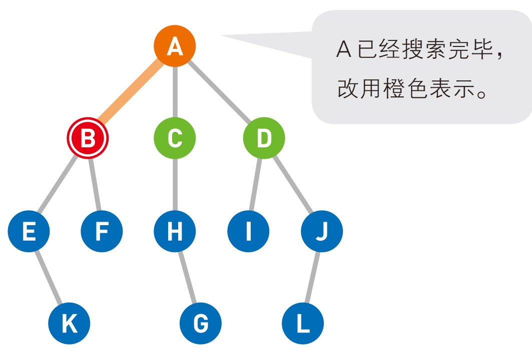
从候补顶点中选出一个顶点。优先选择最早成为候补的那个顶点，如果多个顶点同时成为候补，那么可以随意选择其中一个。

04



此处B、C、D同时成为候补，所以我们随机选择了最左边的顶点B。

05



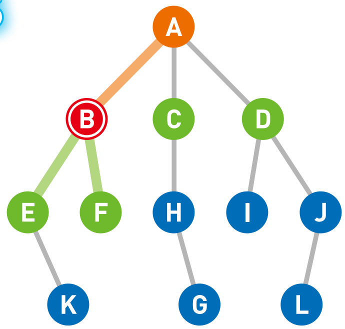
移动到选中的顶点B上。此时我们在B上，所以B变为红色，同时将已经搜索过的顶点变为橙色。

提示

此处，候补顶点是用“先入先出”（FIFO）的方式来管理的，因此可以使用“队列”这个数据结构。

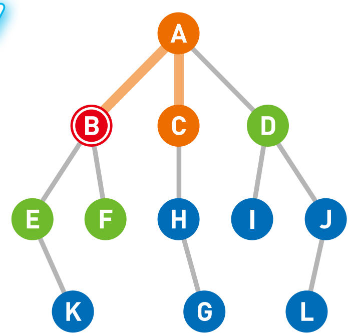
▶参考：1-5 队列

06



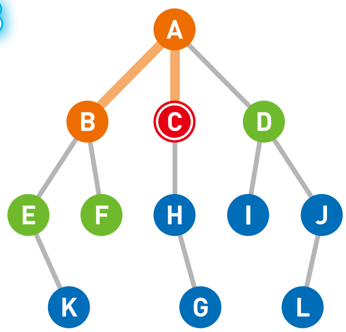
将可以从B直达的两个顶点E和F设为候补顶点。

07



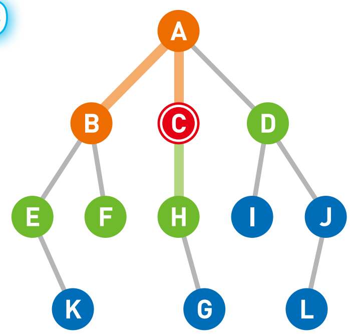
此时，最早成为候补顶点的是C和D，我们选择了左边的顶点C。

08



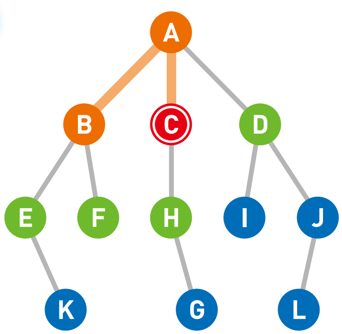
移动到选中的顶点C上。

09



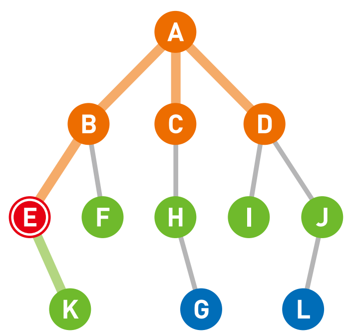
将可以从C直达的顶点H设为候补顶点。

10



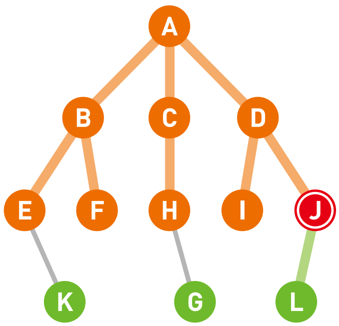
重复上述操作直到到达终点，或者所有的顶点都被遍历为止。

11



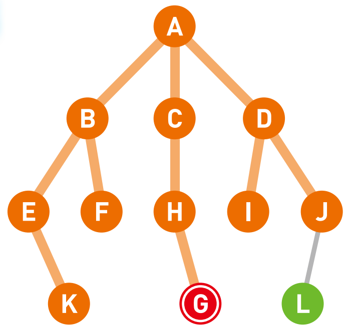
这个示例中的搜索顺序为A、B、C、D、E、F、H、I、J、K。

12



完成了从A到I的搜索，现在在顶点J处。

13



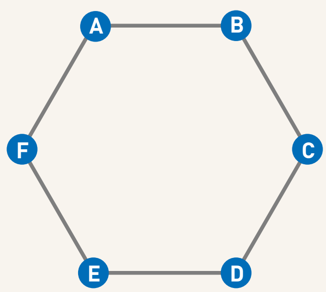
到达终点G，搜索结束。

解说

广度优先搜索的特征为从起点开始，由近及远进行广泛的搜索。因此，目标顶点离起点越近，搜索结束得就越快。

补充说明

为了方便说明，这次讲解用的是没有闭环[1]的图。不过，如果图中有闭环，其搜索步骤也是一样的。像示例那样的没有闭环的图叫作“树”。



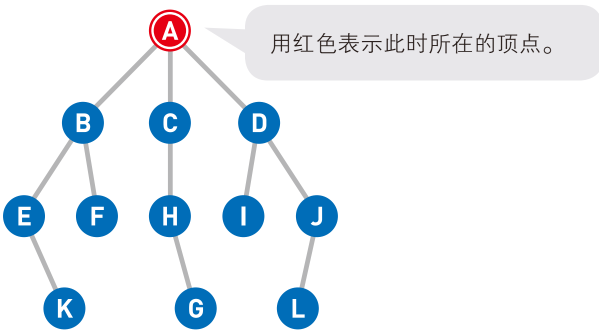
① 闭环如下图所示，起点和终点是同一个顶点。

No. 4-3

深度优先搜索

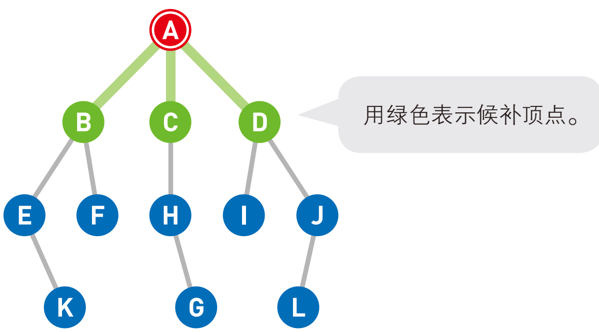
深度优先搜索和广度优先搜索一样，都是对图进行搜索的算法，目的也都是从起点开始搜索直到到达指定顶点（终点）。深度优先搜索会沿着一条路径不断往下搜索直到不能再继续为止，然后再折返，开始搜索下一条候补路径。

01



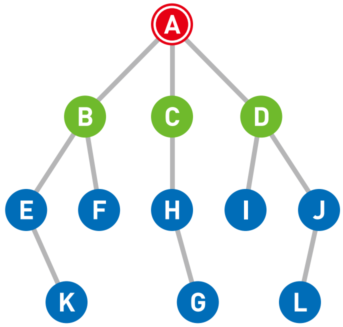
A为起点，G为终点。一开始我们在起点A上。

02



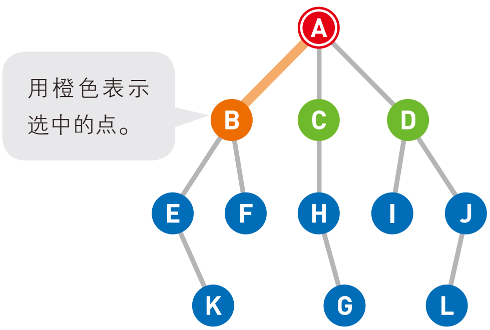
将可以从A直达的三个顶点B、C、D设为下一步的候补顶点。

03



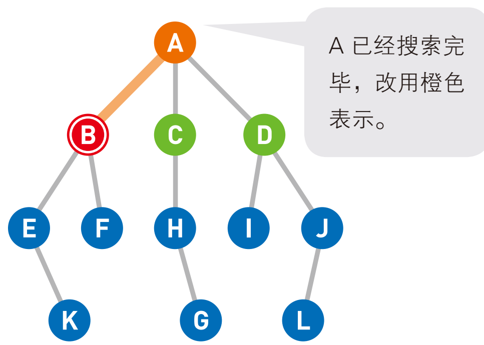
从候补顶点中选出一个顶点。优先选择最新成为候补的点，如果几个顶点同时成为候补，那么可以从中随意选择一个。

04



此处B、C、D同时成为候补，所以我们随机选择了最左边的顶点。

05



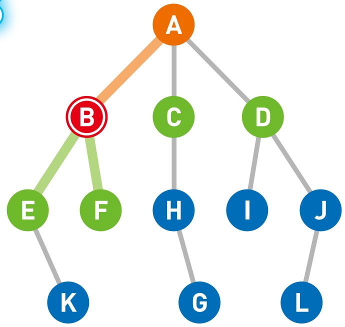
移动到选中的顶点B。此时我们在B上，所以B变为红色，同时将已经搜索过的顶点变为橙色。

提示

此处，候补顶点是用“后入先出”（LIFO）的方式来管理的，因此可以使用“栈”这个数据结构。

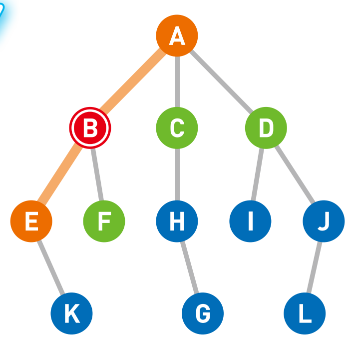
参考：1-4 栈

06



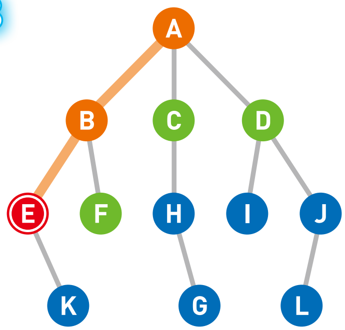
将可以从B直达的两个顶点E和F设为候补顶点。

07



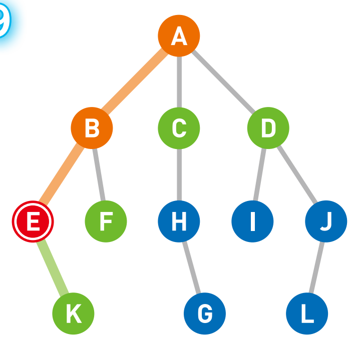
此时，最新成为候补顶点的是E和F，我们选择了左边的顶点E。

08



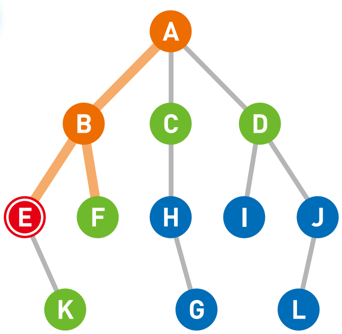
移动到选中的顶点E上。

09



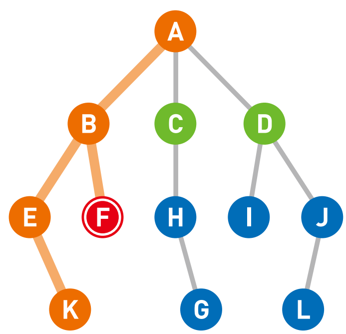
将可以从E直达的顶点K设为候补顶点。

10



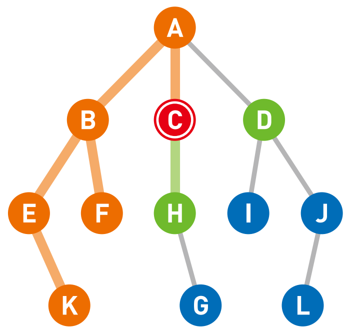
重复上述操作直到到达终点，或者所有顶点都被遍历为止。

11



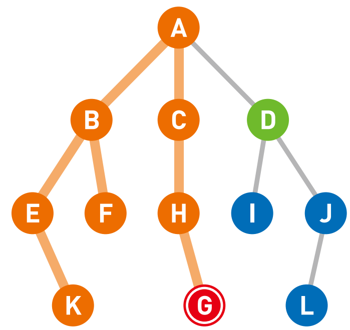
这个示例的搜索顺序为A、B、E、K、F、C、H。

12



现在我们搜索到了顶点C。

13



到达终点G，搜索结束。

解说

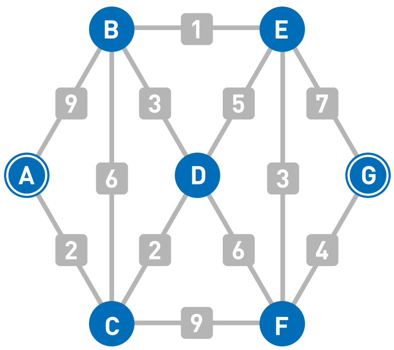
深度优先搜索的特征为沿着一条路径不断往下，进行深度搜索。虽然广度优先搜索和深度优先搜索在搜索顺序上有很大的差异，但是在操作步骤上却只有一点不同，那就是选择哪一个候补顶点作为下一个顶点的基准不同。

广度优先搜索选择的是最早成为候补的顶点，因为顶点离起点越近就越早成为候补，所以会从离起点近的地方开始按顺序搜索；而深度优先搜索选择的则是最新成为候补的顶点，所以会一路往下，沿着新发现的路径不断深入搜索。

No.
4-4 贝尔曼 – 福特算法

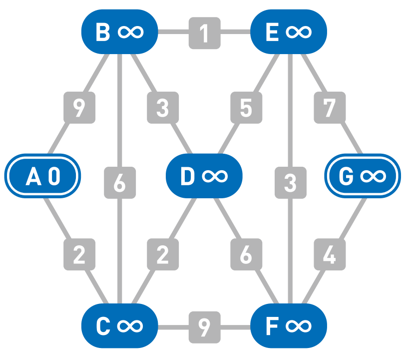
贝尔曼 – 福特（Bellman-Ford）算法是一种在图中求解最短路径问题的算法。最短路径问题就是在加权图指定了起点和终点的前提下，寻找从起点到终点的路径中权重总和最小的那条路径。

01



这里我们设A为起点、G为终点，来讲解贝尔曼 – 福特算法。

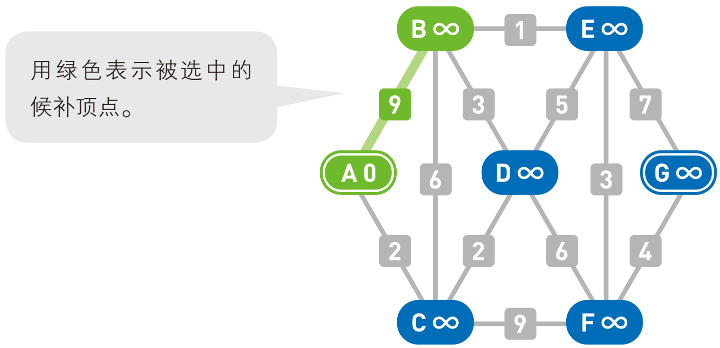
02



最初并不知道要走多远才能到达其他顶点（甚至不知道能否到达），因此将起点以外的顶点权重设为无穷大。

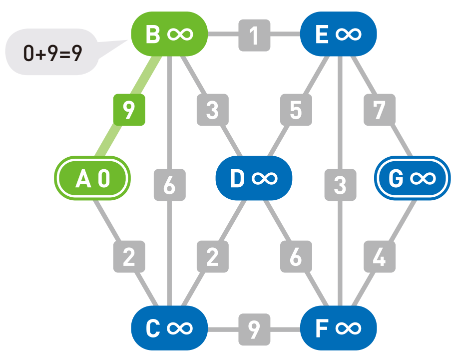
首先设置各个顶点的初始权重：起点为0，其他顶点为无穷大（∞）。这个权重表示的是从A到该顶点的最短路径的暂定距离。随着计算往下进行，这个值会变得越来越小，最终收敛到正确的数值。

03



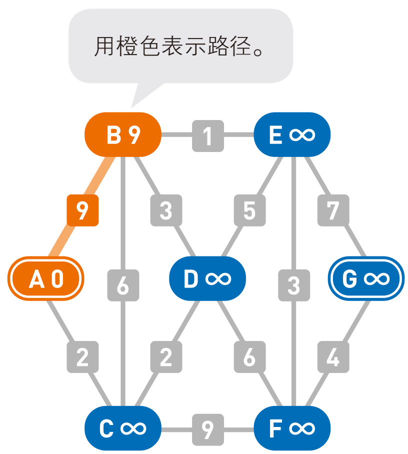
从所有的边中选出一条边，此处选择了连接A-B的边。然后，分别计算这条边从一端到另一端的权重，计算方法是“顶点原本的权重＋边的权重”。只要按顺序分别计算两个方向的权重即可，从哪一端开始都没有问题。此处我们选择按顶点权重从小到大的方向开始计算。

04



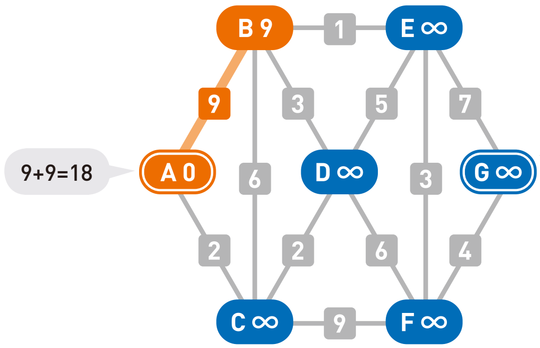
A 的权重小于 B，因此先计算从 A 到 B 的权重。A 的权重是 0，边 A–B 的权重是 9，因此 A 到 B 的权重是 0+9=9。

05



如果计算结果小于顶点的值，就更新这个值。顶点B的权重是无穷大，比9大，所以把它更新为9。更新时需要记录计算的是从哪个顶点到该顶点的路径。

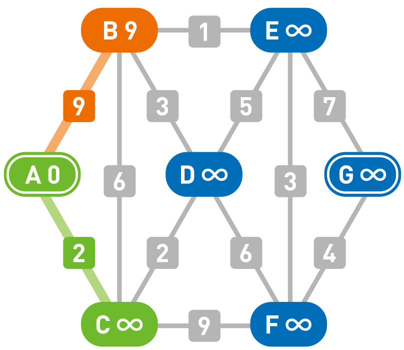
06



从权重较大的顶点到较小的顶点时，只要边的权重不为负，就不会更新权重。

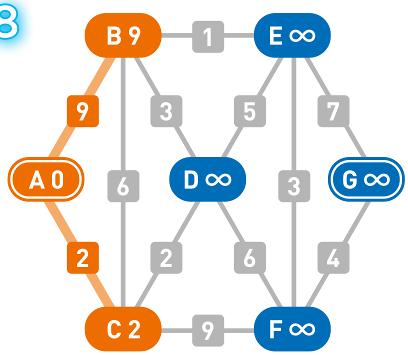
接下来计算从 B 到 A 的权重。B 的权重为 9，从 B 到 A 的权重便为 9+9=18。与顶点 A 现在的值 0 进行比较，因为现在的值更小，所以不更新。

07



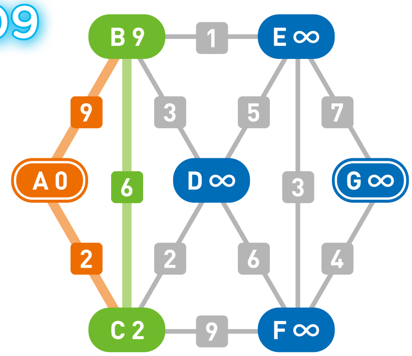
对所有的边都执行同样的操作。在执行顺序上没有特定要求，此处我们选择从靠近左侧的边开始计算。先选出一条边……

08



数值更新了，顶点 C 的权重变成了 2。

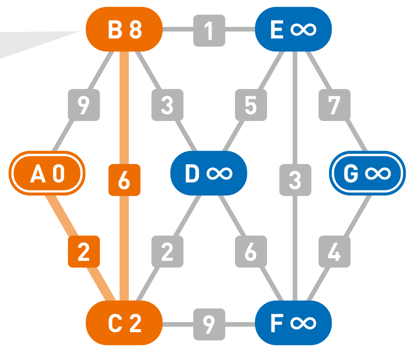
09



同样地，再选出一条边……

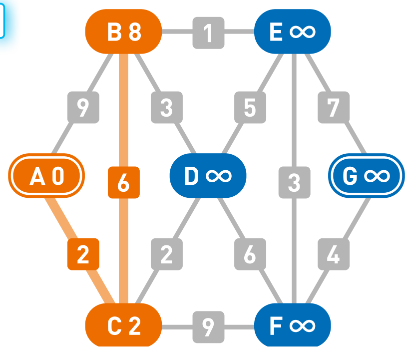
10

此处顶点 B 因为边 B-C 而更新了权重，所以路径从之前的 A-B 变为了现在的 B-C。因此 A-B 不再以橙色表示，而 B-C 变为橙色。



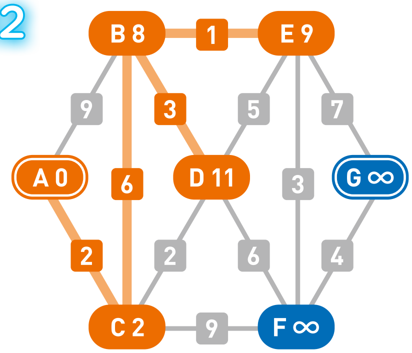
权重又更新了。此时就能看出，从顶点 A 前往顶点 B 时，比起从 A 直达 B，在 C 中转一次的权重更小。

11



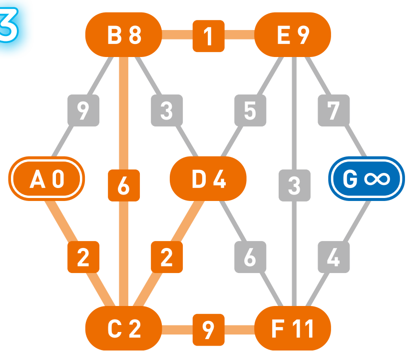
接着对所有的边进行更新操作。

12

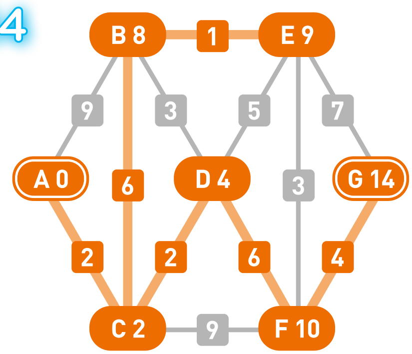


更新边 B-D 和边 B-E。

13

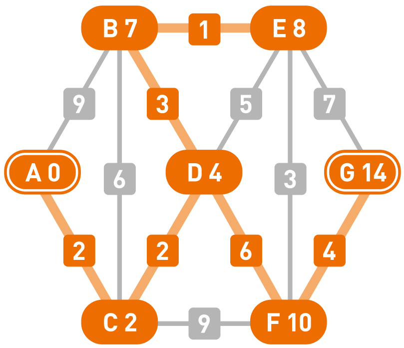


14



更新边 C–D 和边 C–F。

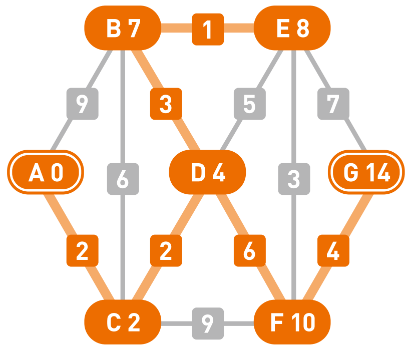
15



第2轮更新也结束了。顶点B的权重从8变成了7，顶点E的权重从9变成了8。接着，再执行一次更新操作。

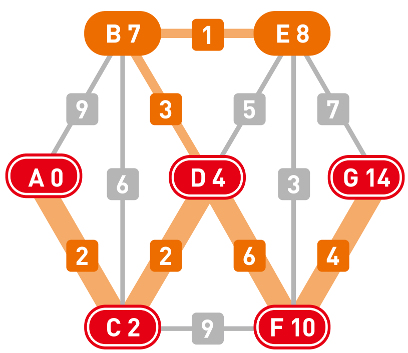
更新完所有的边后，第1轮更新就结束了。接着，重复对所有边的更新操作，直到权重不能被更新为止。

16



第3轮更新结束，所有顶点的权重都不再更新，操作到此为止。算法的搜索流程也就此结束，我们找到了从起点到其余各个顶点的最短路径。

17



根据搜索结果可知，从起点 A 到终点 G 的最短路径是 A–C–D–F–G，权重为 14。

解说

将图的顶点数设为 n、边数设为 m，我们来思考一下贝尔曼－福特算法的时间复杂度是多少。该算法经过 n 轮更新操作后就会停止，而在每轮更新操作中都需要对各个边进行 1 次确认，因此 1 轮更新所花费的时间就是 $O(m)$，整体的时间复杂度就是 $O(nm)$。

为了便于说明，前面的讲解以无向图为例，但在有向图中同样可以求解最短路径问题。选出一条边并计算顶点的权重时，无向图中的计算如前文步骤 03~06 所示，两个方向都要计算，而在有向图中只按照边所指向的那个方向来计算就可以了。

补充说明

计算最短路径时，边的权重代表的通常都是时间、距离或者路费等，因此基本都是非负数①。不过，即便权重为负，贝尔曼 - 福特算法也可以正常运行。

但是，如果在一个闭环中边的权重总和是负数，那么只要不断遍历这个闭环，路径的权重就能不断减小，也就是说根本不存在最短路径。遇到这种对顶点进行 n 次更新操作后仍能继续更新的情况，就可以直接认定它“不存在最短路径”。

另外，如果使用 4-5 节将会介绍的狄克斯特拉算法，那么当输入的权重为负时还有可能无法得出正确的答案。

▶参考：4-5 狄克斯特拉算法

小知识

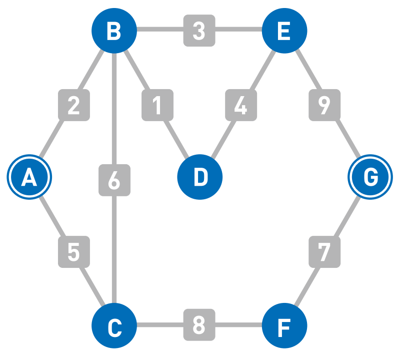
贝尔曼－福特算法的名称取自其创始人理查德 · 贝尔曼和莱斯特 · 福特的名字。贝尔曼也因为提出了该算法中的一个重要分类“动态规划”而被世人所熟知。

① 指不是负数的实数，即 0 和正数。

No.
4-5 狄克斯特拉算法

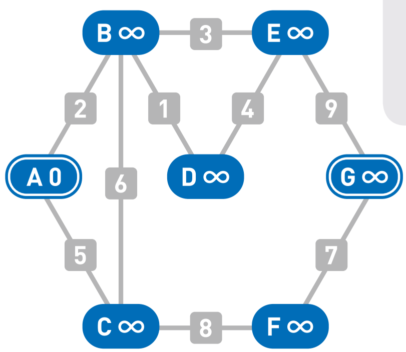
与前面提到的贝尔曼－福特算法类似，狄克斯特拉（Dijkstra）算法也是求解最短路径问题的算法，使用它可以求得从起点到终点的路径中权重总和最小的那条路径路径。

01



这里我们设A为起点、G为终点，来讲解狄克斯特拉算法。

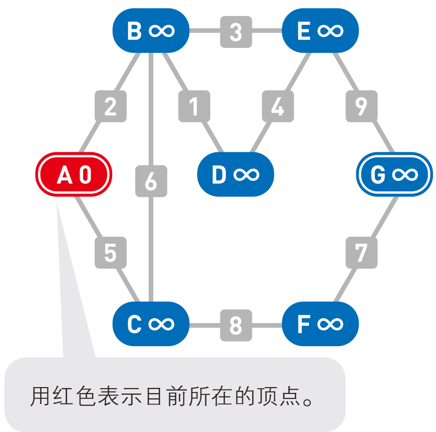
02



这个权重和贝尔曼－福特算法中的权重所代表的意思一样，都是最短路径的暂定距离。

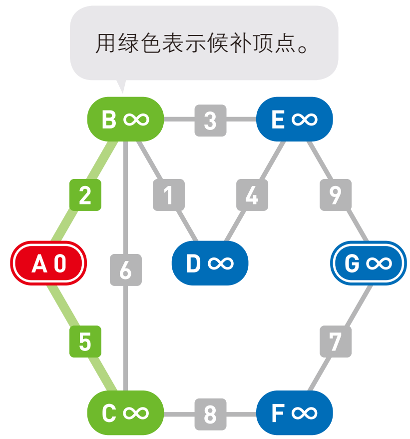
首先设置各个顶点的初始权重：起点为 0，其他顶点为无穷大（∞）。

03



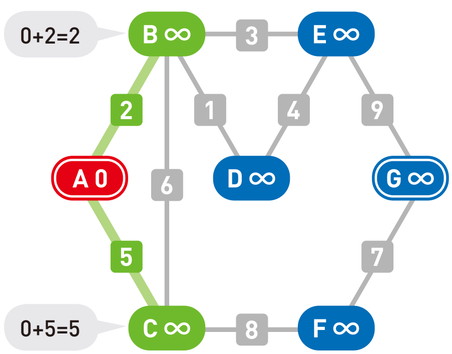
从起点出发。

04



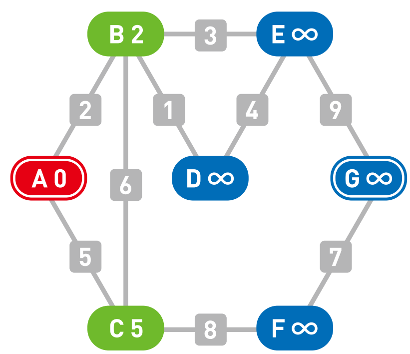
寻找可以从目前所在的顶点直达且尚未被搜索过的顶点，此处为顶点B和顶点C，将它们设为下一步的候补顶点。

05



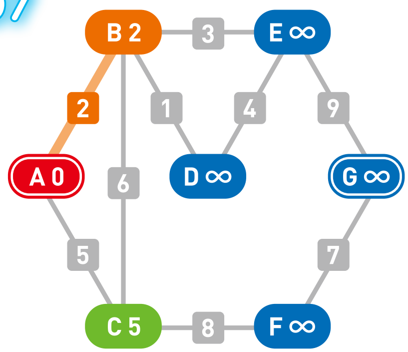
计算各个候补顶点的权重。计算方法是“目前所在顶点的权重＋目前所在顶点到候补顶点的权重”。比如起点A的权重是0，那么顶点B的权重就是0+2=2。用同样的方法计算顶点C，结果就是0+5=5。

06



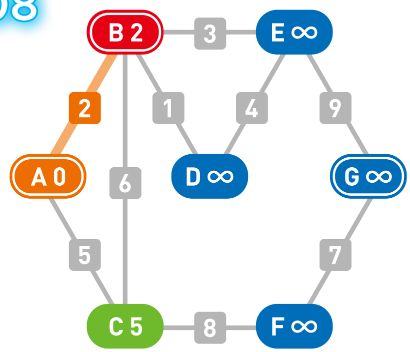
如果计算结果小于候补顶点的值，就更新这个值。顶点B和顶点C现在的权重都是无穷大，大于计算结果，所以更新这两个顶点的值。

07



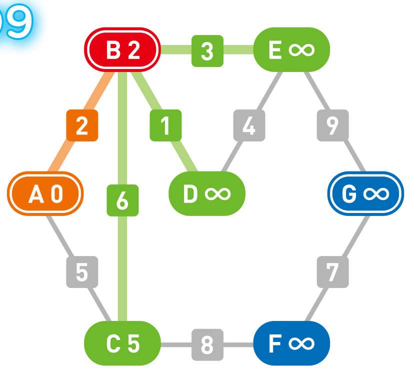
从候补顶点中选出权重最小的顶点。此处 B 的权重最小，那么路径 A–B 就是从起点 A 到顶点 B 的最短路径。因为如果要走别的路径，那么必定会经过顶点 C，其权重也就必定会高于 A–B 这条路径。

08



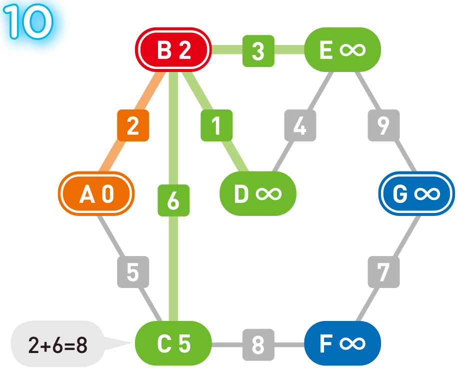
确定了最短路径，移动到顶点 B。

09



将可以从顶点B直达的顶点设为新的候补顶点，于是顶点D和顶点E也成为了候补。目前有三个候补顶点C、D、E。

10

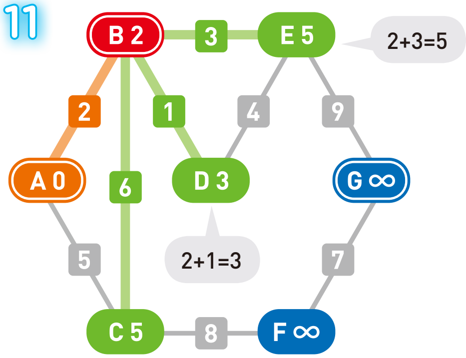


用相同的方法计算各个候补顶点的权重。从B到C的权重为2+6=8，比C当前的权重5大，因此不更新这个值。

小知识

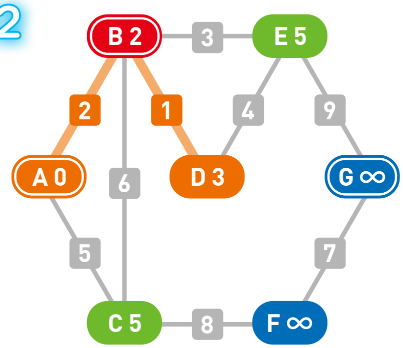
狄克斯特拉算法的名称取自该算法的提出者埃德斯加 · 狄克斯特拉，他在 1972 年获得了图灵奖。

11



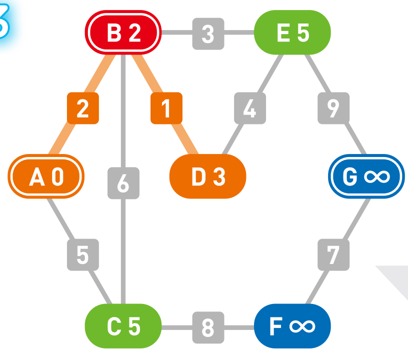
更新了剩下的顶点 D 和 E。

12



从候补顶点中选出权重最小的顶点。此处 D 的权重最小，那么路径 A–B–D 就是从起点 A 到顶点 D 的最短路径。

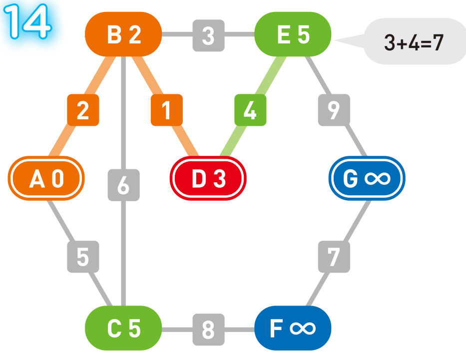
13



就像这样，狄克斯特拉算法一边逐一确定起点到各个顶点的最短路径，一边对图进行搜索。

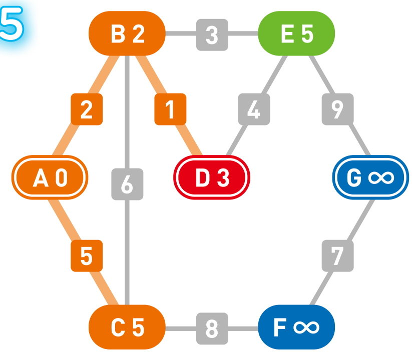
A–B–D 这条路径是通过逐步从候补顶点中选择权重最小的顶点来得到的，所以 如果经过其他顶点到达 D，其权重必定会大于这条路径。

14



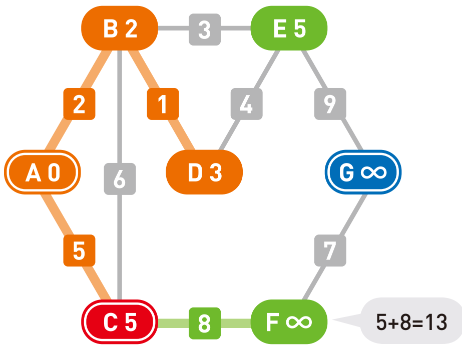
要重复执行同样的操作直到到达终点 G 为止。移动到顶点 D 后算出了 E 的权重，然而并不需要更新它（因为 3+4=7）。现在，两个候补顶点 C 和 E 的权重都为 5，所以选择哪一个都可以。

15



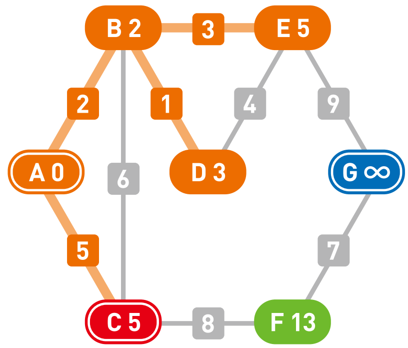
此处我们选择了 C，于是起点 A 到顶点 C 的最短路径便确定了。

16



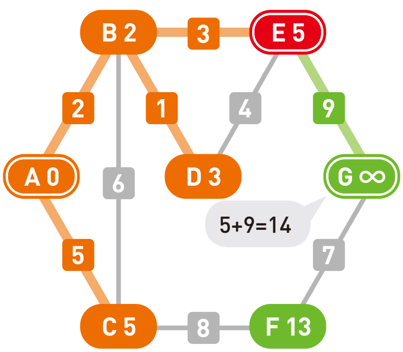
移动到C后，顶点F成为了新的候补顶点，且F的权重被更新为13。此时的候补顶点中，E为5、F为13，所以……

17



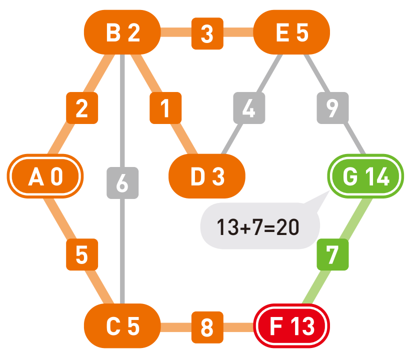
我们选择了权重更小的E，起点A到顶点E的最短路径也就确定了下来。

18



移动到E。G成了新的候补顶点，其权重也被更新为14。此时的候补顶点中，F为13、G为14，所以选择了F。由此，起点A到顶点F的最短路径也就确定了下来。

19

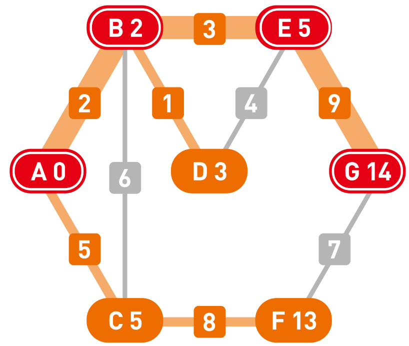


移动到F。顶点G的权重计算结果为13+7=20，比现在的值14要大，因此不更新它。由于候补顶点只剩G了，所以选择G，并确定了起点A到顶点G的最短路径。

20

到达终点G，搜索结束。

21



用粗线条标注的就是从起点A到终点G的最短路径。

解说

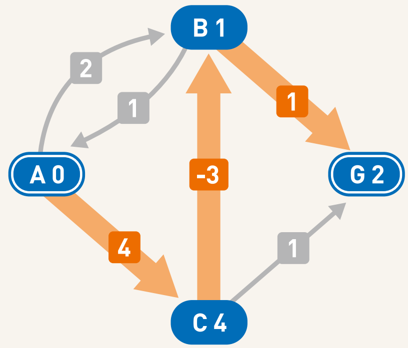
比起需要对所有的边都重复计算权重和更新权重的贝尔曼－福特算法，狄克斯特拉算法多了一步选择顶点的操作，这使得它在求最短路径上更为高效。

将图的顶点数设为 n、边数设为 m，那么如果事先不进行任何处理，该算法的时间复杂度就是 $O(n^2)$。不过，如果对数据结构进行优化，那么时间复杂度就会变为 $O(m+n\log n)$。

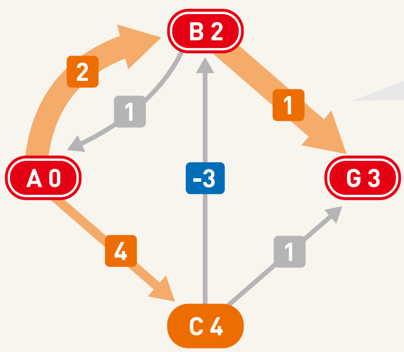
补充说明

狄克斯特拉算法和贝尔曼－福特算法一样，也能在有向图中求解最短路径问题。

但是如果图中含有负数权重，狄克斯特拉算法可能会无法得出正确答案，这一点和贝尔曼－福特算法有所不同。比如右边这个图中，A–C–B–G 为正确的最短路径，权重为 4+(–3)+1=2。



然而，如果用狄克斯特拉算法来求解，得到的却是下面这样的最短路径树。从起点 A 到终点 G 的最短路径为 A–B–G，权重为 3。这个答案显然是错误的。



最初 B 和 C 成为候补顶点时，B 的权重更小，因此确定了 A–B 路径。但是，因为有负数权重，所以实际上 A–C–B 路径的整体权重更小。只是狄克斯特拉算法一开始还不知道 C–B 路径的存在，导致出现了错误。

4-4 节中曾讲过，如果闭环中有负数权重，就不存在最短路径。贝尔曼－福特算法可以直接认定不存在最短路径，但在狄克斯特拉算法中，即便不存在最短路径，它也会算出一个错误的最短路径出来。因此，有负数权重时不能使用狄克斯特拉算法。

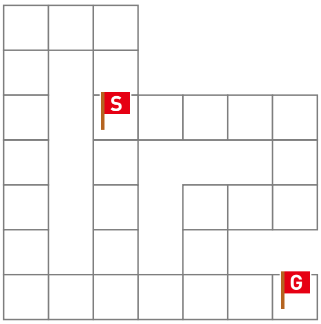
总的来说，就是不存在负数权重时，更适合使用效率较高的狄克斯特拉算法，而存在负数权重时，即便较为耗时，也应该使用可以得到正确答案的贝尔曼－福特算法。

▶参考：4-4 贝尔曼－福特算法

No.
4-6 A* 算法

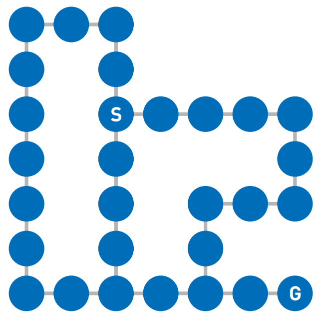
A*（A-Star）算法也是一种在图中求解最短路径问题的算法，由狄克斯特拉算法发展而来。狄克斯特拉算法会从离起点近的顶点开始，按顺序求出起点到各个顶点的最短路径。也就是说，一些离终点较远的顶点的最短路径也会被计算出来，但这部分其实是无用的。与之不同，A* 就会预先估算一个值，并利用这个值来省去一些无用的计算。

01



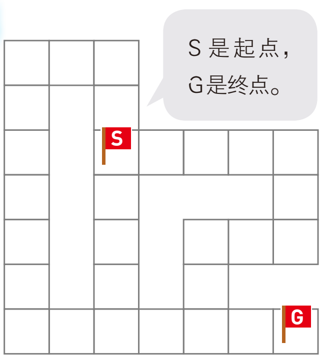
首先，试着用狄克斯特拉算法来求该迷宫中的最短路径吧。

02



将迷宫看作是一个图，其中每个方块都是一个顶点，各顶点间的距离（权重）都为1。

03

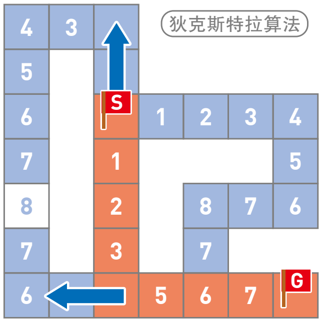


以这个设定为前提，用狄克斯特拉算法来求最短路径。

04

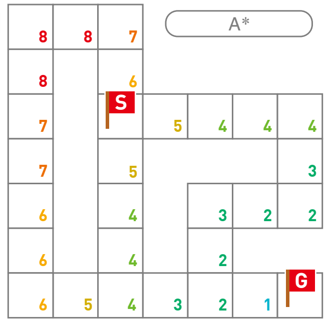
用狄克斯特拉算法求最短路径的结果会如上图所示，方块中的数字表示从起点到该顶点的距离（权重），蓝色和橙色的方块表示搜索过的区域，橙色方块同时还表示从S到G的最短路径。

05



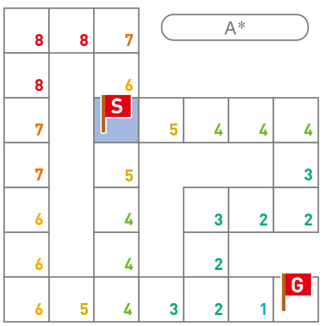
狄克斯特拉算法只根据起点到候补顶点的距离来决定下一个顶点。因此，它无法发现蓝色箭头所指的这两条路径其实离终点越来越远，同样会继续搜索。

06



而A*算法不仅会考虑从起点到候补顶点的距离，还会考虑从当前所在顶点到终点的估算距离。这个估算距离可以自由设定，此处我们用的是将顶点到终点的直线距离四舍五入后的值。

07



接下来，就试着用A*算法来求解吧。首先把起点设为搜索完毕状态。搜索完的点都用蓝色表示。

08

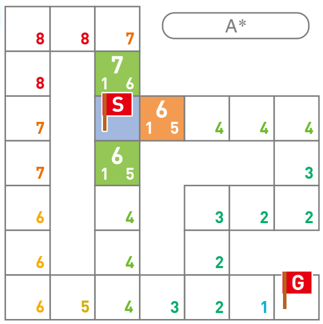
分别计算起点周围每个顶点的权重。计算方法是“从起点到该顶点的距离”（方块左下）加上“距离估算值”（方块右下）。

提示

如步骤06中方块右下角的数字所示，由人工预先设定的估算距离被称为“距离估算值”。如果事先根据已知信息设定合适的距离估算值，再将它作为启发信息辅助计算，搜索就会变得更加高效①。这里我们只知道终点所在位置，不知道该如何通往终点，所以使用了直线距离。

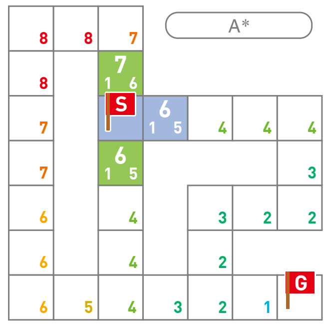
① 因此，这样的算法也被称为启发式算法。——编者注

09



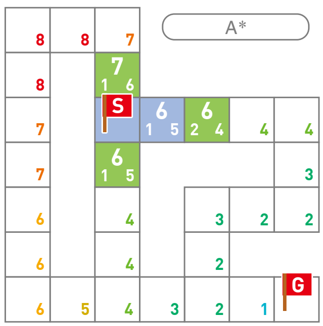
选择一个权重最小的顶点，用橙色表示。

10



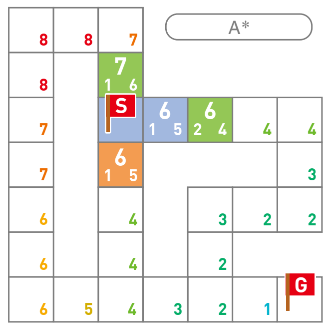
将选择的顶点设为搜索完毕状态。

11



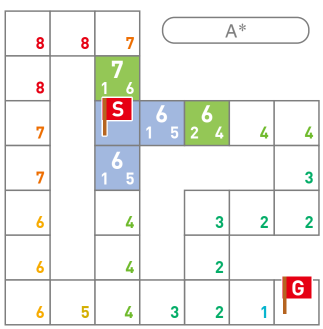
计算搜索完毕的顶点到下一个顶点的权重。

12



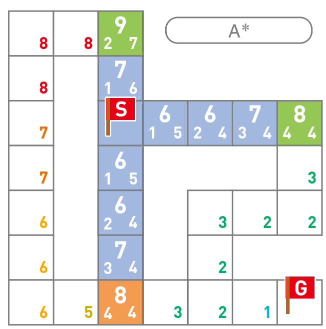
选择距离最短的一个顶点。

13



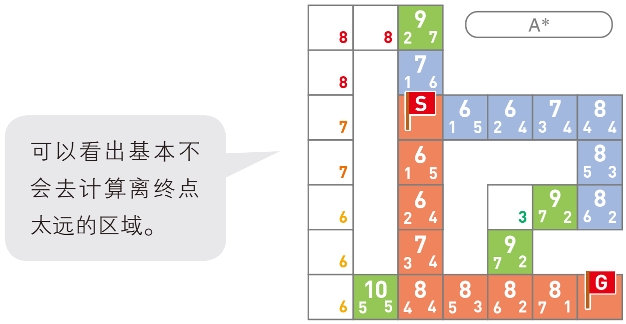
将选好的顶点设为搜索完毕状态。之后重复上述操作，直到到达终点为止。

14



搜索中……

15



搜索完毕。效率比狄克斯特拉算法的高了很多。

解说

如果我们能得到一些启发信息，即各个顶点到终点的大致距离（这个距离不需是准确的值）我们就能使用 A* 算法。当然，有时这类信息是完全无法估算的，这时就不能使用 A* 算法。

距离估算值越接近当前顶点到终点的实际值，A* 算法的搜索效率也就越高；反过来，如果距离估算值与实际值相差较大，那么该算法的效率可能会比狄克斯特拉算法的还要低。如果差距再大一些，甚至可能无法得到正确答案。

不过，当距离估算值小于实际距离时，是一定可以得到正确答案的（只是如果没有设定合适的距离估算值，效率会变差）。

应用示例

A* 算法在游戏编程中经常被用于计算敌人追赶玩家时的行动路线等，但由于该算法的计算量较大，所以可能会使游戏整体的运行速度变慢。因此在实际编程时，需要考虑结合其他算法，或者根据具体的应用场景做出相应调整。

第 5 章

安全算法

No. 5-1

安全和算法

互联网中不可或缺的安全技术

通过互联网交换数据时，数据要经过各种各样的网络和设备才能传到对方那里。数据在传输过程中有可能会经过某些恶意用户的设备，从而导致内容被盗取。

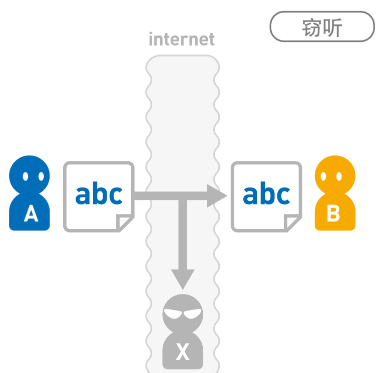
因此，要想安全地使用互联网，安全技术是不可或缺的。本章将要学习的就是保障安全的各种算法和利用了这些算法的机制。

传输数据时的四个问题

首先，介绍一下用互联网传输数据时可能会发生的四个主要问题。

▶ 窃听

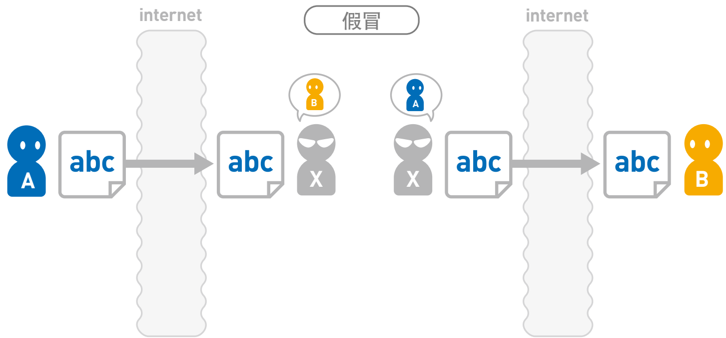
A 向 B 发送的消息可能会在传输途中被 X 偷看（如右图）。这就是“窃听”。



▶ 假冒

A 以为向 B 发送了消息，然而 B 有可能是 X 冒充的（如下页上图）；反过来，B 以为从 A 那里收到了消息，然而 A 也有可能是 X 冒充的。

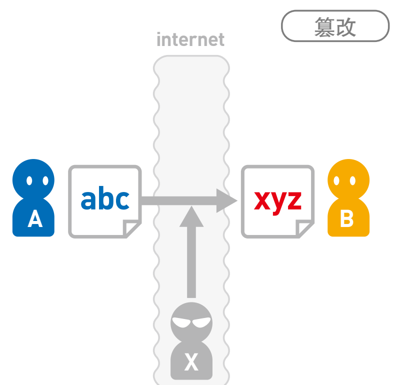
这种问题就叫作“假冒”。



▶ 篡改

即便 B 确实收到了 A 发送的消息，但也有可能像右图这样，该消息的内容在途中就被 X 更改了。

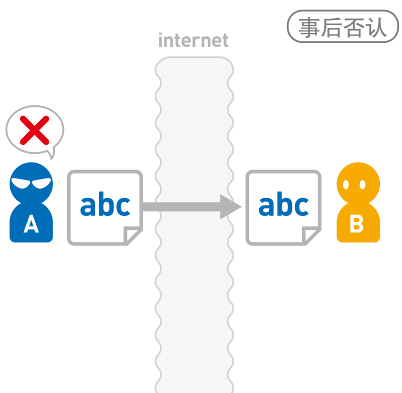
这种行为就叫作“篡改”。除了被第三者篡改外，通信故障导致的数据损坏也可能会使消息内容发生变化。



▶ 事后否认

B 从 A 那里收到了消息，但作为消息发送者的 A 可能对 B 抱有恶意，并在事后声称“这不是我发送的消息”（参考下图）。

这种情况会导致互联网上的商业交易或合同签署无法成立。这种行为便是“事后否认”。



四个主要问题到这里就介绍完毕了。这些问题不光发生在用户之间交流的时候，也有可能发生在用户浏览网页的时候。

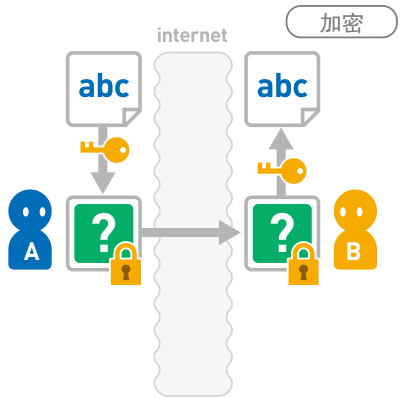
解决这些问题的安全技术

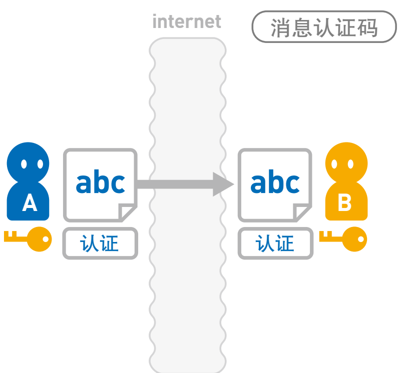
为了解决这些问题，我们需要使用哪些安全技术呢？来简单了解一下每个问题的应对方法吧。

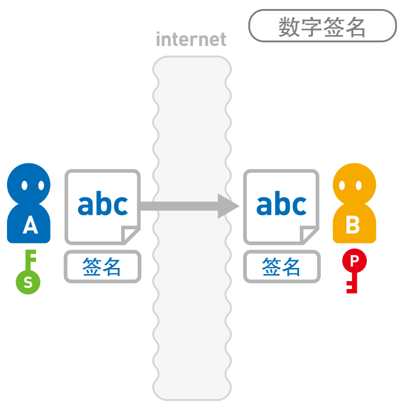
为了应对第一个问题“窃听”，我们会使用“加密”技术。

为了应对第二个问题“假冒”，我们会使用“消息认证码”（下图左）或“数字签名”（下图右）技术。

为了应对第三个问题“篡改”，我们同样会使用“消息认证码”或“数字签名”技术。其中“数字签名”技术还可以用于预防第四个问题“事后否认”。







第5章的知识点

问题和相应的解决方法可总结成如下页表格。

问题	解决方法
① 窃听	加密
② 假冒	消息认证码 or 数字签名
③ 篡改	
④ 事后否认	数字签名

“数字签名”技术存在“无法确认公开密钥的制作者”这一问题。要想解决这个问题，可以使用“数字证书”技术。

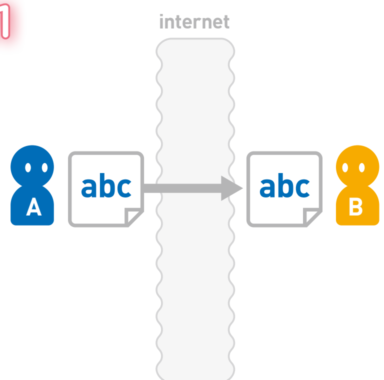
本章就将详细讲解这些安全技术。

No. 5-2

加密的基础知识

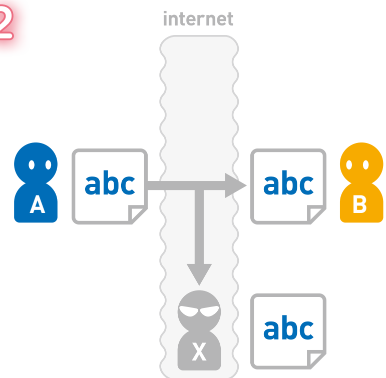
在现代互联网社会中，加密技术是不可或缺的。那么对数据进行加密和解密时，计算机会进行哪些处理呢？这一节我们将讲解加密技术的必要性和基本原理。

01



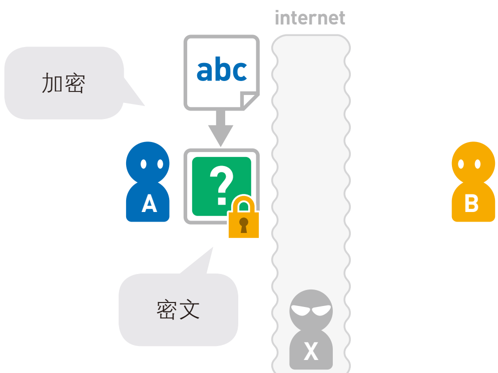
假设A想通过互联网向B发送消息。数据要经过互联网上各种各样的网络和设备才能到达B那里。如果像上图这样直接发送数据的话……

02



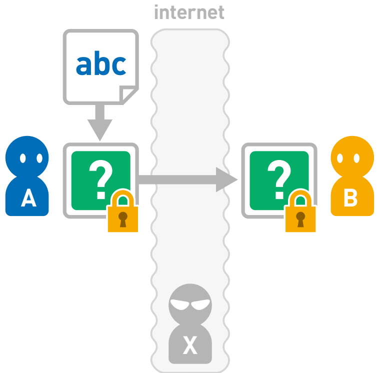
数据可能会被第三者恶意窃听。

03



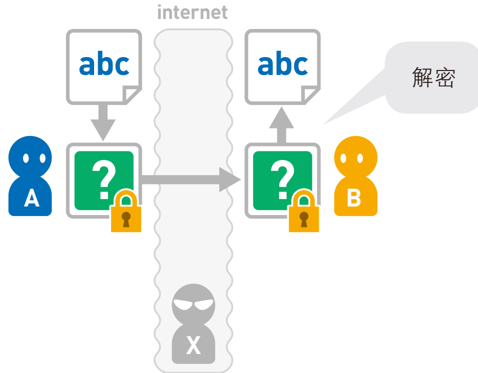
因此，我们需要给想要保密的数据加密。加密后的数据被称为“密文”。

04



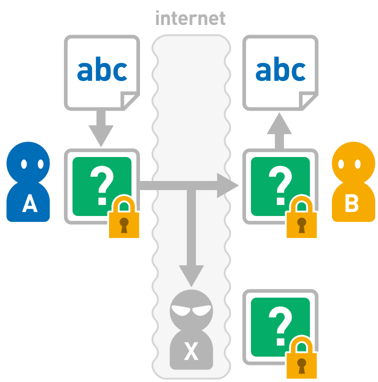
把密文发送给B。

05



B收到密文后，需要解除加密才能得到原本的数据。把密文恢复为原本数据的操作就叫作“解密”。

06

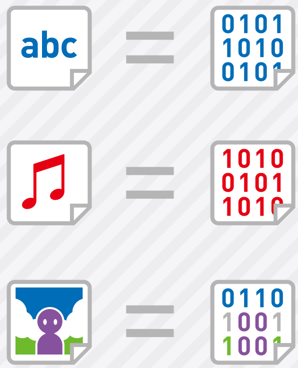


像这样对数据进行加密，就不用担心会被人窃听了。

解说

在现代互联网社会中，加密技术变得十分重要。这里，我们再来说明一下加密的具体操作。

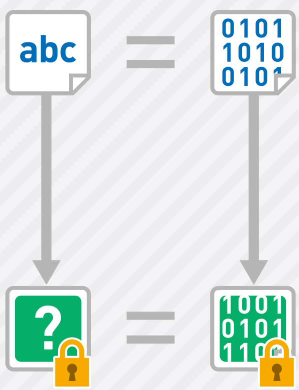
首先，计算机会用由 0 和 1 这两个数字表示的二进制来管理所有数据。如下图所示，数据虽然有文本、音频、图像等不同的形式，但是在计算机中都是用二进制来表示的。



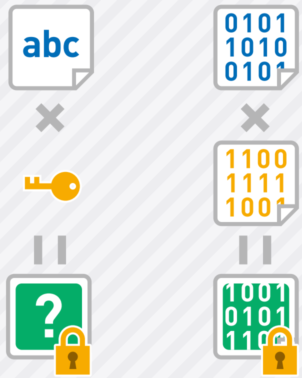
在此基础上，我们来思考如何加密数据。

对计算机来说，数据就是一串有意义的数字罗列。密文也是数字罗列，只不过它是计算机无法理解的无规律的数字罗列。

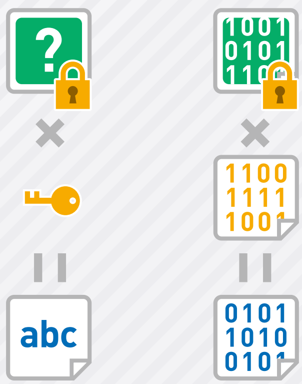
也就是说，加密就是数据经过某种运算后，变成计算机无法理解的数的过程（请参考下图）。



在加密运算上会用到“密钥”。所以加密就是用密钥对数据进行数值运算，把数据变成第三者无法理解的形式的过程（请参考下图）。



反过来，解密就是像下图这样，通过密钥进行数值计算，把密文恢复成原本数据的过程。



像这样，将数据变成第三者的计算机无法理解的形式，然后再将其恢复成原本数据的一系列操作就是加密技术。

No.
5-3 哈希函数

哈希函数可以把给定的数据转换成固定长度的无规律数值。转换后的无规律数值可以作为数据摘要应用于各种各样的场景。

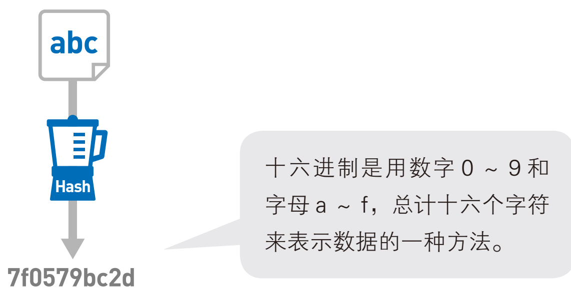
01

为了便于理解，我们可以把哈希函数想像成搅拌机。

02

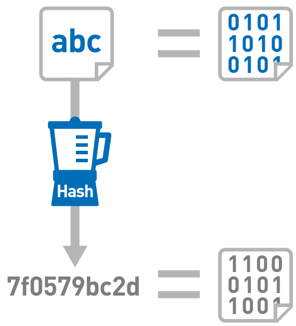
将数据输入到哈希函数后……

03



输出固定长度的无规律数值。把哈希函数想像成搅拌数据的搅拌机就很容易理解了。输出的无规律数值就是“哈希值”。哈希值虽然是数字，但多用十六进制来表示。

04

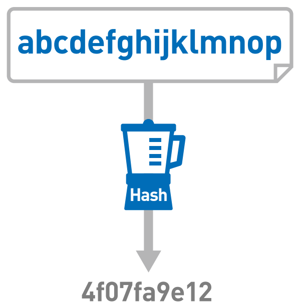


计算机会用由0和1这两个数字表示的二进制来管理所有的数据。虽然哈希值是用十六进制表示的，但它也是数据，在计算机内部同样要用二进制来进行管理。也就是说，哈希函数实际上是在计算机内部进行着某种运算的。

05

以此为前提，我们再来看看哈希函数的特征。第一个特征是输出的哈希值数据长度不变。

06

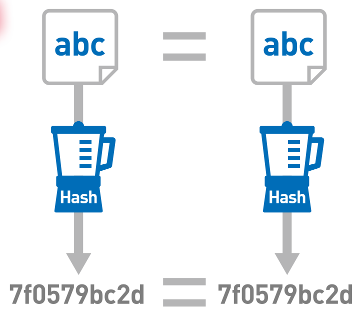


即使输入了相当大的数据，输出的哈希值的长度也保持不变。

07

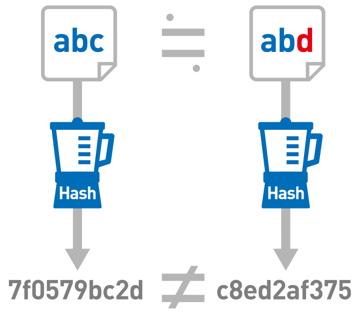
同样地，不管输入的数据多小，哈希值的长度仍然相同。

08



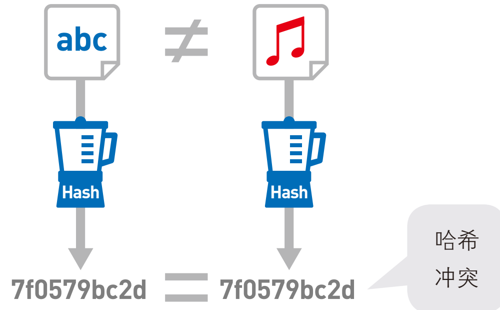
第二个特征是如果输入的数据相同，那么输出的哈希值也必定相同。

09



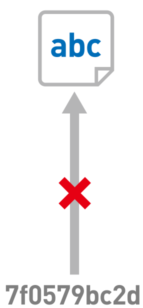
第三个特征是即使输入的数据相似，但哪怕它们只有一比特的差别，那么输出的哈希值也会有很大的差异。输入相似的数据并不会导致输出的哈希值也相似。

10



第四个特征是即使输入的两个数据完全不同，输出的哈希值也有可能是相同的，虽然出现这种情况的概率比较低。这种情况叫作“哈希冲突”。

11



第五个特征是不可能从哈希值反向推算出原本的数据。输入和输出不可逆这一点和加密有很大不同。

12

最后一个特征是求哈希值的计算相对容易。

提示

哈希函数可以应用于各种各样的场景。本书详细讲解了哈希函数在哈希表和消息认证码中的应用。

▶参考：1-6 哈希表
▶参考：5-8 消息认证码

解说

哈希函数的算法中具有代表性的是 MD5[①]、SHA-1[②]和 SHA-2 等。其中 SHA-2 是现在应用较为广泛的一个，而 MD5 和 SHA-1 存在安全隐患，不推荐使用。

不同算法的计算方式也会有所不同，比如 SHA-1 需要经过数百次的加法和移位运算才能生成哈希值。

虽然本节中讲过如果输入的数据相同，那么输出的哈希值也必定相同，但这是在使用同一个算法的前提下得出的结论。若使用的算法不同，那么就算输入的数据相同，得到的哈希值也是不同的。

应用示例

将用户输入的密码保存到服务器时也需要用到哈希函数。

如果把密码直接保存到服务器，可能会被第三者窃听，因此需要算出密码的哈希值，并只存储哈希值。当用户输入密码时，先算出该输入密码的哈希值，再把它和服务器中的哈希值进行比对。这样一来，就算保存的哈希值暴露了，鉴于上文中提到的哈希函数的第五个特征（输入输出不可逆），第三者也无法得知原本的密码。

就像这样，使用哈希函数可以更安全地实现基于密码的用户认证。

① Message Digest Algorithm 5 的缩写。

② SHA 是 Secure Hash Algorithm 的缩写。

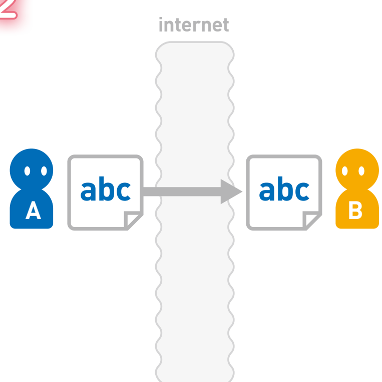
No. 5-4 共享密钥加密

加密数据的方法可以分为两种：加密和解密都使用相同密钥的“共享密钥加密”和分别使用不同密钥的“公开密钥加密”。本节将讲解共享密钥加密的机制及其相关问题。

01

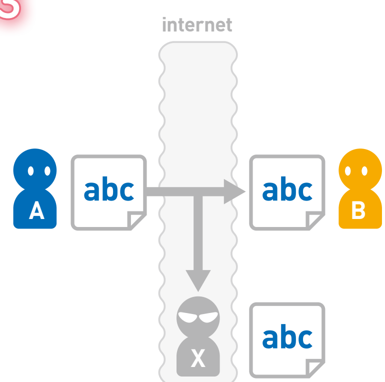
共享密钥加密是加密和解密都使用相同密钥的一种加密方式。由于使用的密钥相同，所以这种算法也被称为“对称加密”。

02



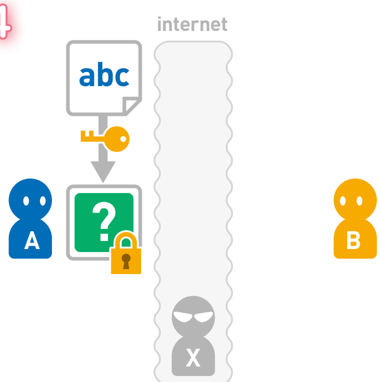
我们先从整体上来了解一下共享密钥加密的处理流程。假设A准备通过互联网向B发送数据。

03



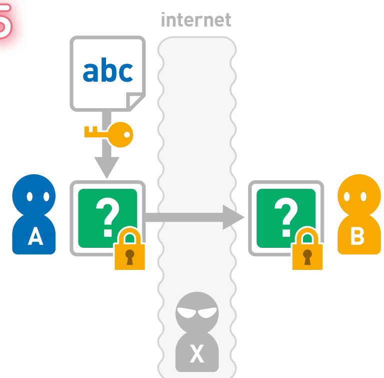
由于有被窃听的风险，所以需要把想要保密的数据加密后再发送。

04



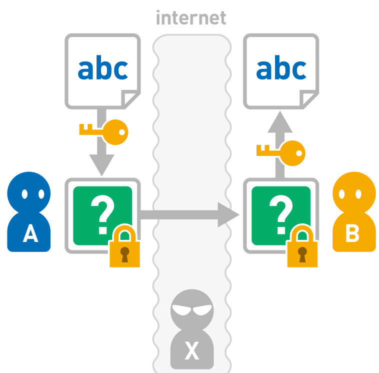
A 使用密钥加密数据。

05



A 将密文发送给 B。

06



B 收到密文后，使用相同的密钥对其进行解密。这样，B 就取得了原本的数据。只要是加密好的数据，就算被第三者恶意窃听也无须担心。

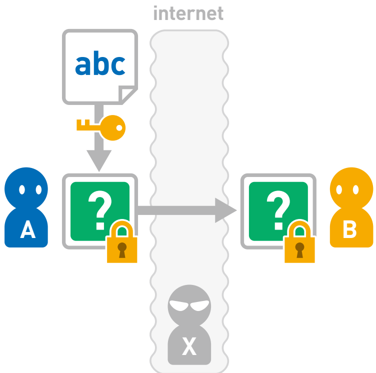
小知识

实现共享密钥加密的算法有凯撒密码、AES[1]、DES[2]、动态口令等，其中 AES 的应用最为广泛。

① Advanced Encryption Standard 的缩写。

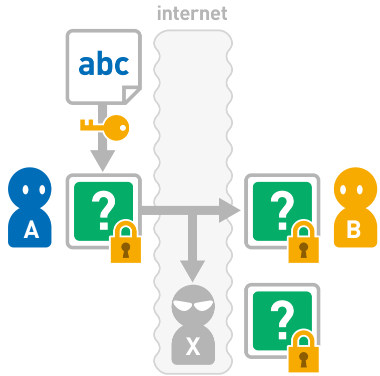
② Data Encryption Standard 的缩写。

07



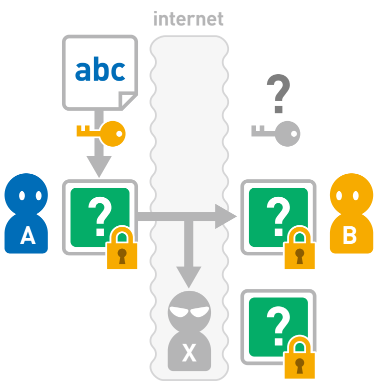
接下来想一想共享密钥加密中的问题。让我们回到B收到A发送的密文的时候。

08



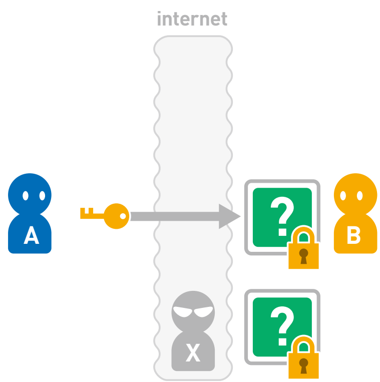
密文可能已经被X窃听了。

09



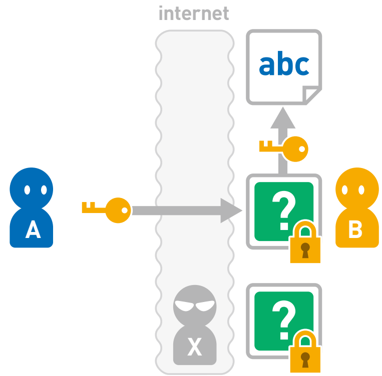
这里假设A和B无法直接沟通，B不知道加密时使用的是什么密钥。

10



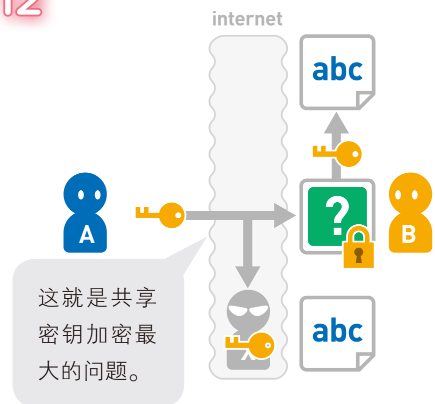
A需要通过某种手段将密钥交给B。和密文一样，A又在互联网上向B发送了密钥。

11



B使用收到的密钥对密文进行解密。

12



但是，该密钥也有可能会被X窃听。这样一来，X也可以使用密钥对密文进行解密了。

解说

既然密钥有被第三者窃听的风险，那是不是也可以先加密密钥再发送呢？使用这种方式，又会产生如何把加密密钥的密钥发送给对方的问题，还是回到了一开始的问题。

因此需要找到可以把密钥安全送出的方法，这就是“密钥分配问题”。

要想解决这个问题，可以使用“密钥交换协议”和“公开密钥加密”两种方法。后面本书将会对这两种方法进行详细说明。

▶参考：5-5 公开密钥加密
▶参考：5-7 迪菲－赫尔曼密钥交换

小知识

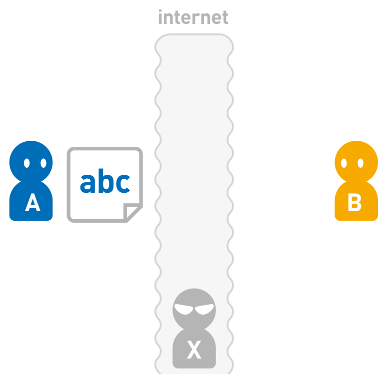
在第二次世界大战中，德军所用的“恩尼格玛密码机”（Enigma）使用的加密方式就是共享密钥加密。德军将一个月的密钥记录成表格进行交接，因此密钥分配并不是该密码机的弱点。然而，使用该密码机加密后的密文会周期性地出现相同的文字，英国的数学家艾伦・图灵就是利用这一点破译了密文，给盟军的胜利带来了极大的帮助。

现在被普遍使用的加密算法即便连续发送相似的文字，也难以被破解。

No.
5-5 公开密钥加密

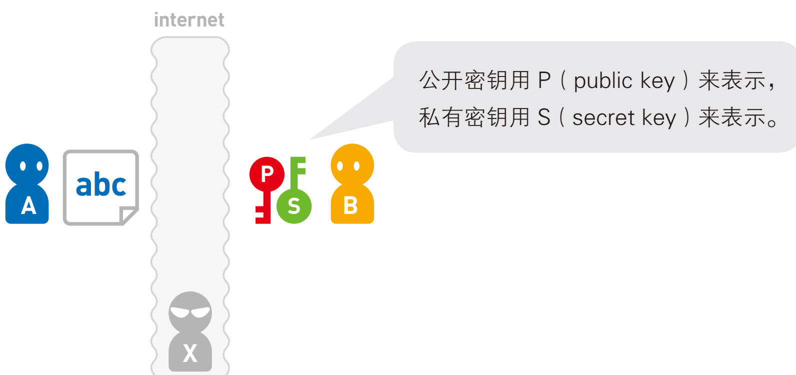
公开密钥加密是加密和解密使用不同密钥的一种加密方法。由于使用的密钥不同，所以这种算法也被称为“非对称加密”。加密用的密钥叫作“公开密钥”，解密用的叫作“私有密钥”。

01



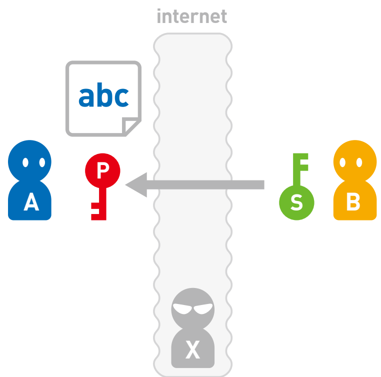
我们先从整体上来了解一下公开密钥加密的处理流程。假设A准备通过互联网向B发送数据。

02



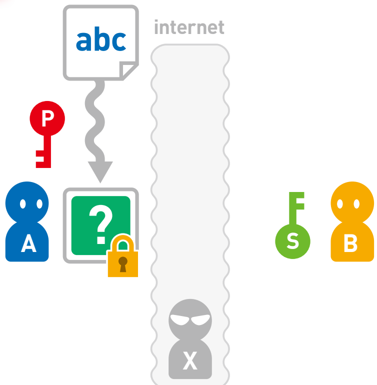
首先，需要由接收方B来生成公开密钥和私有密钥。

03



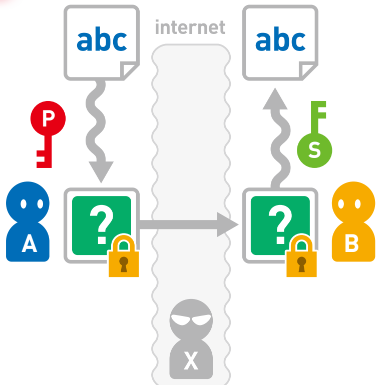
然后把公开密钥发送给A。

04



A使用B发来的公开密钥加密数据。

05

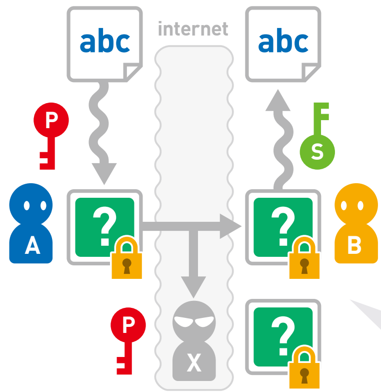


A将密文发送给B，B再使用私有密钥对密文进行解密。这样，B就得到了原本的数据。

提示

实现公开密钥加密的算法有RAS算法、椭圆曲线加密算法等，其中使用最为广泛的是RSA算法。RSA算法由其开发者Rivest、Shamir、Adleman的首字母命名而来，三人在2002年获得了图灵奖。

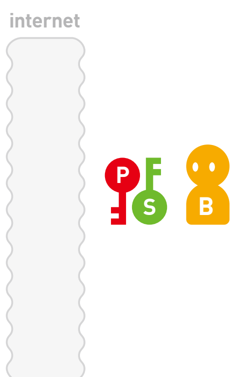
06



与共享密钥加密不同的是，公开密钥加密不会出现密钥分配问题。

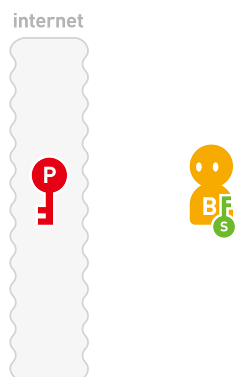
公开密钥和密文都是通过互联网传输的，因此可能会被X窃听。但是，使用公开密钥无法解密密文，因此X也无法得到原本的数据。

07



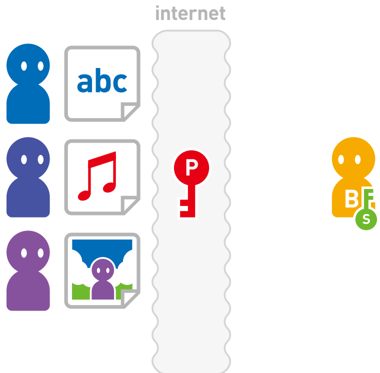
此外，在和多人传输数据时，使用公开密钥加密十分方便。来看一个具体的例子吧。假设B预先准备好了公开密钥和私有密钥。

08



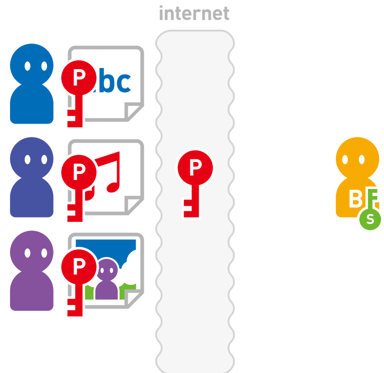
公开密钥是不怕被人知道的，所以B可以把公开密钥发布在网上。与此相反，私有密钥不能被人知道，必须严密保管。

09



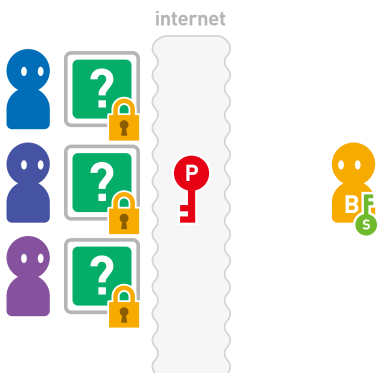
假设有许多人都想向 B 发送数据。

10



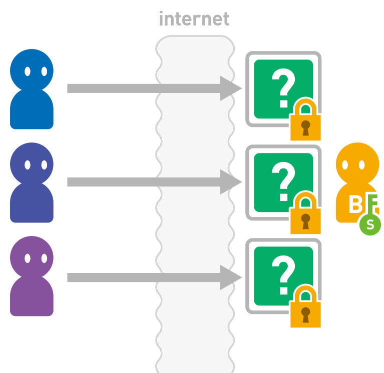
想发送数据的人首先在网上取得B发布的公开密钥。

11



然后用它加密要发送的数据。

12

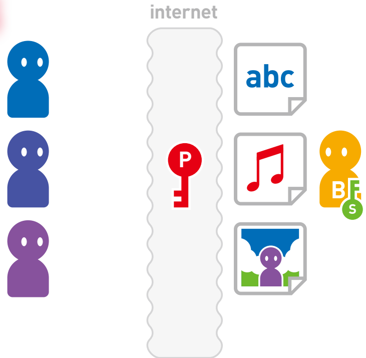


最后把密文发给 B。

提示

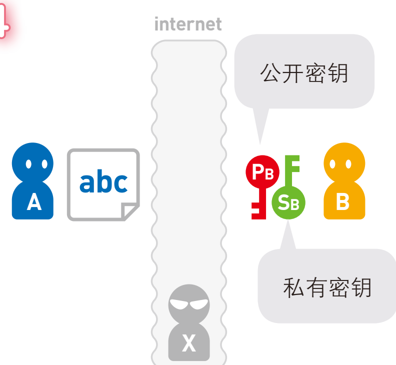
如果使用共享密钥加密，密钥的需求数量会随着发送人数的增多而急剧增多。上一节的例子中只有2个人，因此只需要2个密钥，但5人就需要10个，100人就需要4950个（n=人数，需要的数量便是$\frac{n(n-1)}{2}$）。

13



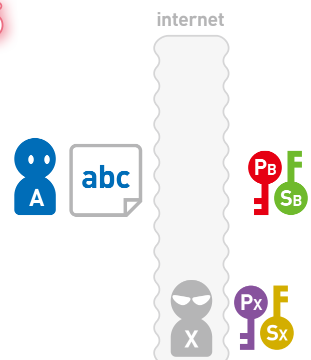
B 用私有密钥对收到的密文进行解密，取得原本的数据。这种情况就不需要为每个发送对象都准备相对应的密钥了。需要保密的私有密钥仅由接收方保管，所以安全性也更高。

14



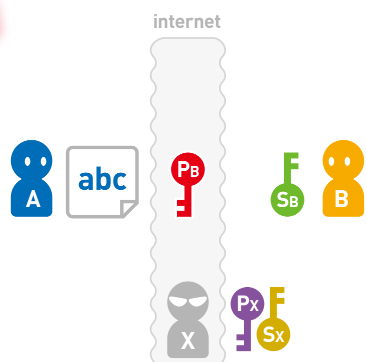
不过，公开密钥加密存在公开密钥可靠性的问题。让我们回到 B 生成公开密钥和私有密钥的时候。在接下来的说明中，B 生成的公开密钥用 P_B 来表示、私有密钥用 S_B 来表示。

15



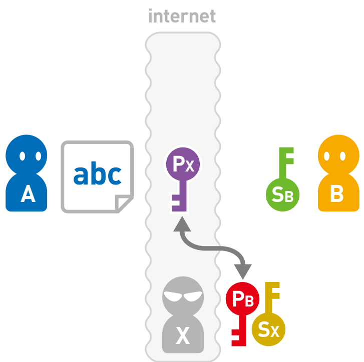
X 想要窃听 A 发给 B 的数据，于是他也准备了公开密钥 P_X 和私有密钥 S_X 。

16



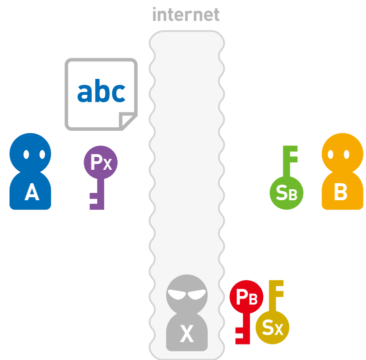
在 B 把公开密钥 P_B 发给 A 的时候……

17



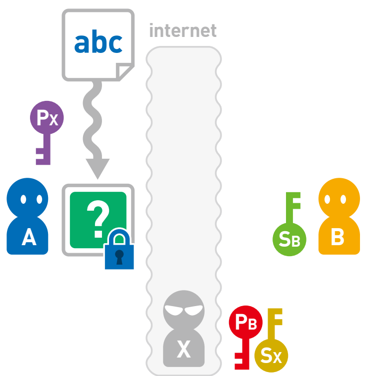
X把公开密钥P_B替换成自己的公开密钥P_X……

18



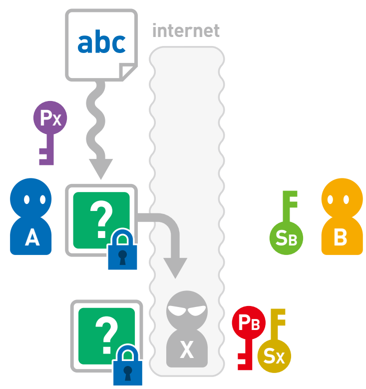
于是公开密钥P_X传到了A那里。由于公开密钥无法显示自己是由谁生成的，所以A不会发现自己收到的公开密钥已经被人替换。

19



A使用公开密钥P_X对数据加密。

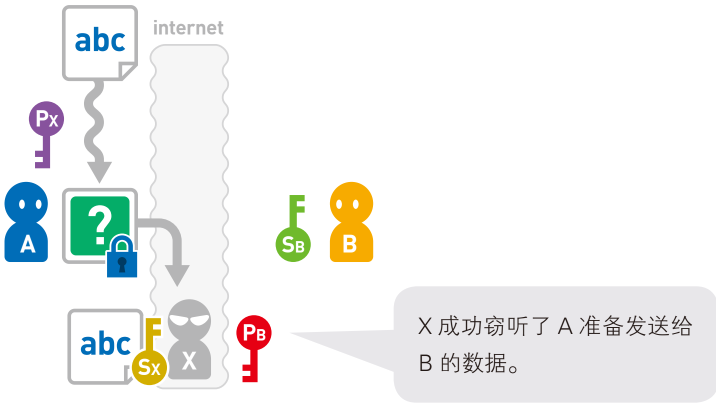
20



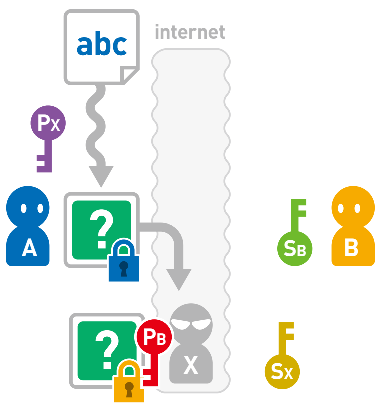
当A把想要给B的密文发送出去后，X接收了这个密文。

21

这个密文由X生成的公开密钥P_X加密而成，所以X可以用自己的私有密钥S_X对密文进行解密。

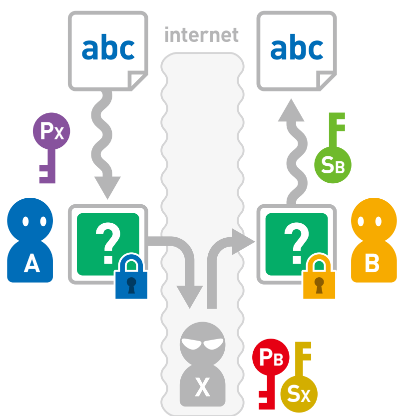


22



接下来，X用B生成的公开密钥P_B加密数据。

23



X把密文发送给B，这个密文由B发出的公开密钥P_B加密而成，所以B可以用自己的私有密钥S_B来解密。从收到密文到解密密文都没发生任何问题，因此B也意识不到数据已经被窃听。这种通过中途替换公开密钥来窃听数据的攻击方法叫作“中间人攻击”（man-in-the-middle attack）。

补充说明

公开密钥的可靠性会出现问题，就是因为 A 无法判断收到的公开密钥是否来自 B。要想解决这个问题，就要用到之后会讲到的“数字证书”。

公开密钥加密还有一个问题，那就是加密和解密都比较耗时，所以这种方法不适用于持续发送零碎数据的情况。要想解决这个问题，就要用到“混合加密”。

▶参考：5-6 混合加密
▶参考：5-10 数字证书

解说

要想找到实现公开密钥加密的算法并不容易。考虑到加密所需的计算流程，算法必须满足如下条件。

① 可以使用某个数值对数据进行加密（计算）。
② 使用另一个数值对加密数据进行计算就可以让数据恢复原样。
③ 无法从一种密钥推算出另一种密钥。

稍微思考一下便知道，想要找到满足以上条件的算法难度有多大。所以，RSA 等可以实现公开密钥加密的算法的提出，对当今互联网社会的安全有着重要的意义。

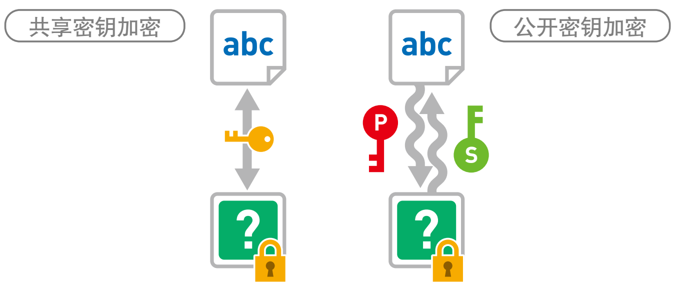
No. 5-6 混合加密

共享密钥加密存在无法安全传输密钥的密钥分配问题，公开密钥加密又存在加密解密速度较慢的问题。结合这两种方法以实现互补的一种加密方法就是混合加密。

▶参考：5-4 共享密钥加密

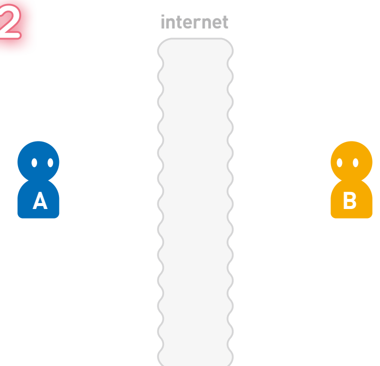
▶参考：5-5 公开密钥加密

01



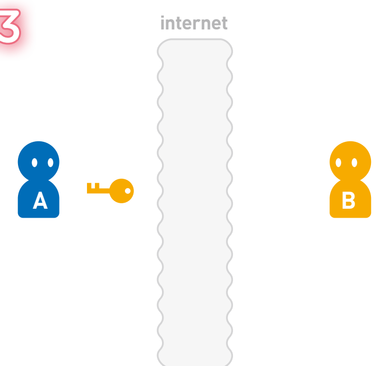
在混合加密中，要用处理速度较快的共享密钥加密对数据进行加密。不过，加密时使用的密钥，则需要用没有密钥分配问题的公开密钥加密进行处理。

02



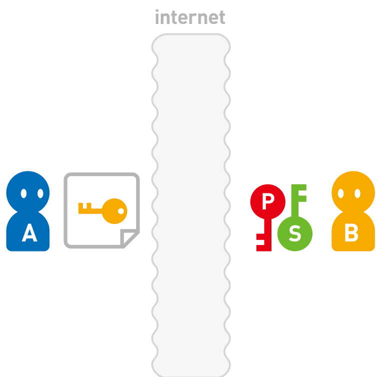
我们来看看混合加密具体的处理流程。假设A准备通过互联网向B发送数据。

03



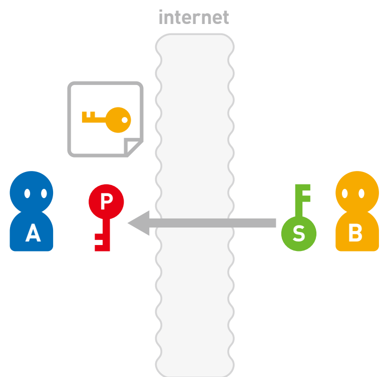
使用处理速度较快的共享密钥加密对数据进行加密。加密时所用的密钥在解密时也要用到，因此A需要把密钥发送给B。

04



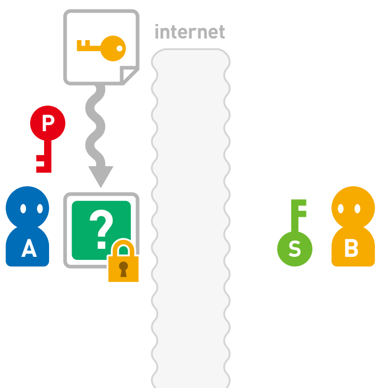
将密钥通过公开密钥加密进行加密后，A就可以将其安全地发送给B了。因此，作为接收方，B需要事先生成公开密钥和私有密钥。

05



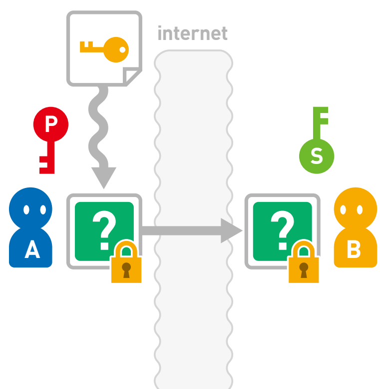
B将公开密钥发送给A。

06



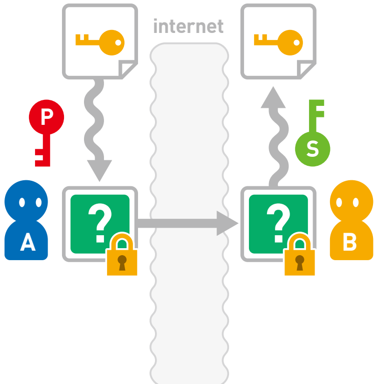
A使用收到的公开密钥，对共享密钥加密中需要使用的密钥进行加密。

07



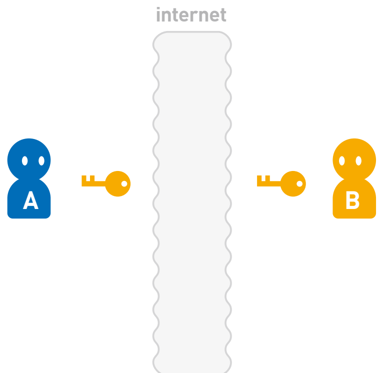
A将加密后的密钥发送给B。

08



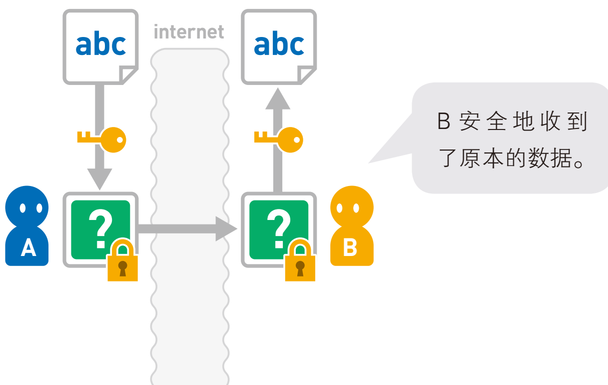
B使用私有密钥对密钥进行解密。

09



这样，A就把共享密钥加密中使用的密钥安全地发送给了B。

10



接下来，A只要将使用这个密钥加密好的数据发送给B即可。加密数据时使用的是处理速度较快的共享密钥加密。

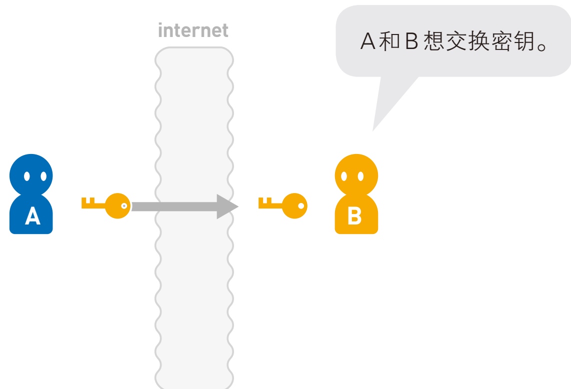
解说

像这样，混合加密在安全性和处理速度上都有优势。能够为网络提供通信安全的 SSL 协议也应用了混合加密方法。SSL 是 Secure Sockets Layer（安全套接层）的简写，该协议经过版本升级后，现在已正式命名为 TLS（Transport Layer Security，传输层安全）。但是，SSL 这个名字在人们心中已经根深蒂固，因此该协议现在也常被称为 SSL 协议或者 SSL / TLS 协议。

No. 5-7 迪菲 – 赫尔曼密钥交换

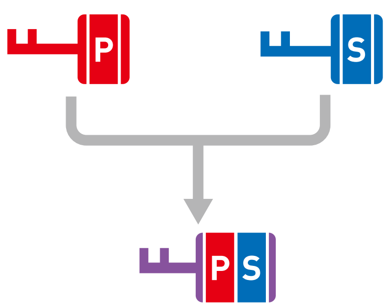
迪菲 – 赫尔曼（Diffie-Hellman）密钥交换是一种可以在通信双方之间安全交换密钥的方法。这种方法通过将双方共有的秘密数值隐藏在公开数值相关的运算中，来实现双方之间密钥的安全交换。

01



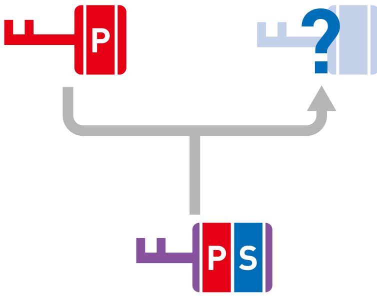
在使用公式进行讲解之前，我们先通过图片来理解一下这个算法的概念。

02



假设有一种方法可以合成两个密钥。使用这种方法来合成密钥P和密钥S，就会得到由这两个密钥的成分所构成的密钥P-S。

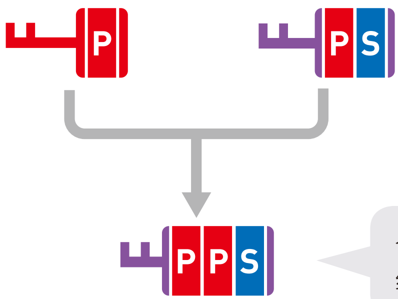
03



密钥之间可以合成，但不能分解。

这种合成方法有三个特征。第一，即使持有密钥P和合成的密钥P-S，也无法把密钥S单独取出来。

04



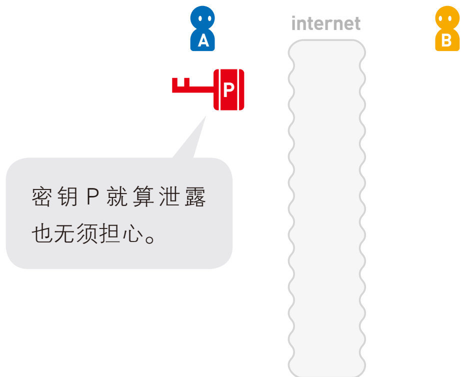
合成后的密钥还可以继续合成。

第二，不管是怎样合成而来的密钥，都可以把它作为新的元素，继续与别的密钥进行合成。比如上图中的这个例子，使用密钥P和密钥P-S，还能合成出新的密钥P-P-S。

05

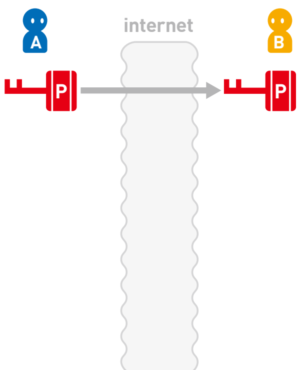
第三，密钥的合成结果与合成顺序无关，只与用了哪些密钥有关。比如合成密钥B和密钥C后，得到的是密钥B-C，再将其与密钥A合成，得到的就是密钥A-B-C。而合成密钥A和密钥C后，得到的是密钥A-C，再将其与密钥B合成，得到的就是密钥B-A-C。此处的密钥A-B-C和密钥B-A-C是一样的。

06



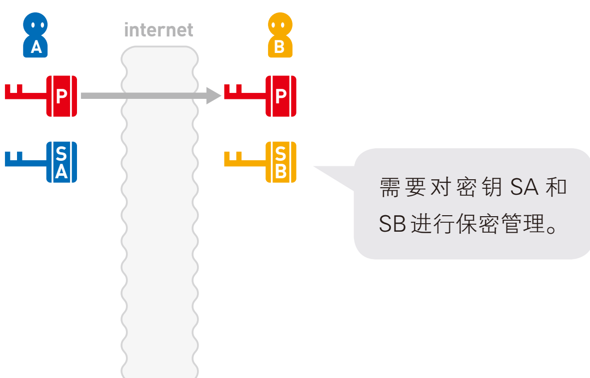
我们试一试用这种方法，在A和B这两人之间安全地交换密钥吧。首先由A生成密钥P。

07



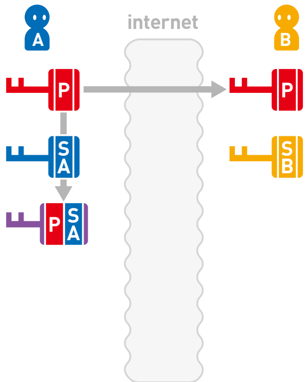
然后A把密钥P发送给B。

08



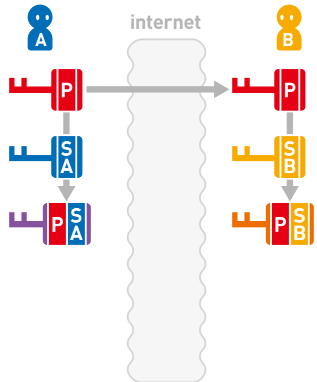
接下来，A和B各自准备自己的私有密钥SA和SB。

09



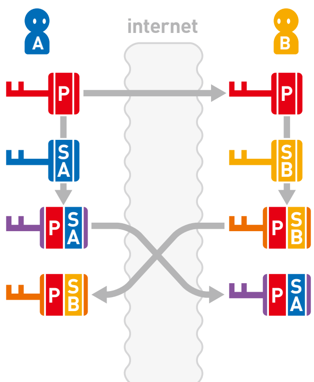
A利用密钥P和私有密钥SA合成新的密钥P-SA。

10



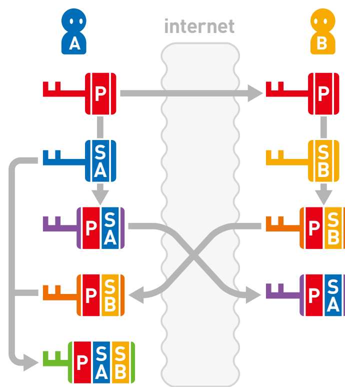
B也利用密钥P和私有密钥SB合成新的密钥P-SB。

11



A将密钥P-SA发送给B，B也将密钥P-SB发送给A。

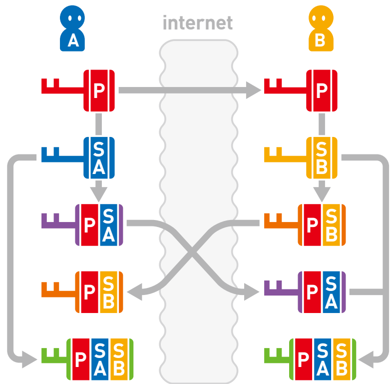
12



合成的结果与合成顺序无关，所以 SA-P-SB 和 P-SA-SB 相同。

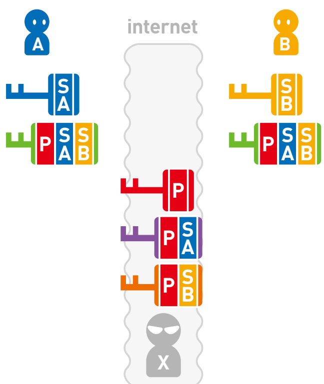
A将私有密钥SA和收到的密钥P-SB合成为新的密钥SA-P-SB。

13



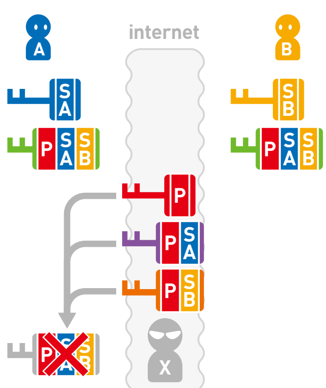
同样地，B也将私有密钥SB和收到的密钥P-SA合成为新的密钥P-SA-SB。于是A和B都得到了密钥P-SA-SB。这个密钥将作为“加密密钥”和“解密密钥”来使用。

14



下面我们来验证该密钥交换的安全性。因为密钥P、密钥P-SA和密钥P-SB需要在互联网上进行传输，所以有可能会被X窃听。

15



由于密钥无法被分解，所以X无法取得私有密钥SA和SB。

但是，X无法用自己窃听到的密钥合成出P-SA-SB，因此这种交换方式是安全的。

16

接下来用公式来表示这种密钥交换法。用 P、G 两个整数来表示一开始生成的公开密钥P。其中 P 是一个非常大的素数，而 G 是素数 P 所对应的生成元（或者“原根”）中的一个。

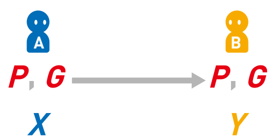
17

首先，由A来准备素数 P 和生成元 G。这两个数公开也没有关系。

18

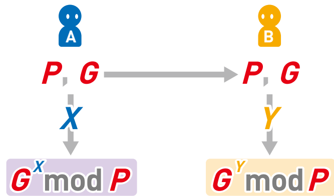
A将素数 P 和生成元 G 发送给B。

19



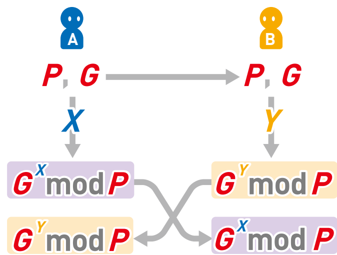
接下来，A和B分别准备了各自的秘密数字 X 和 Y。X 和 Y 都必须小于 P–2。

20



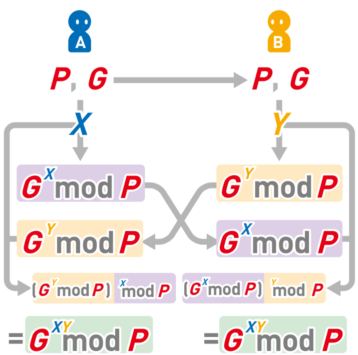
A和B分别计算“(G 的秘密数字次方) mod P”。mod运算就是取余运算。“G mod P” 就是计算G除以P后的余数。此处的运算等同于概念意义上的“合成”。

21



A和B将自己的计算结果发送给对方。

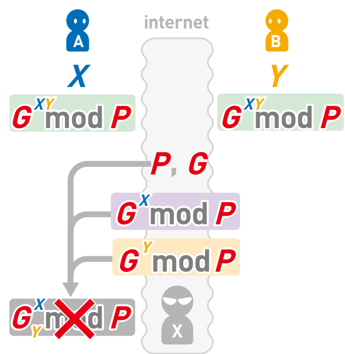
22



这样，A和B就共同拥有了相当于加密密钥的数字。

A和B收到对方的计算结果后，先计算这个值的秘密数字次方，然后再mod P。最后A和B会得到相同的结果。

23



下面来验证这种密钥交换法的安全性。即便X窃听了整个通信过程，也无法用窃听到的数字计算出A和B共有的数字。而且，X也无法计算出保密数字X和Y。因此，此处使用迪菲–赫尔曼密钥交换是安全的。

解说

迪菲–赫尔曼密钥交换由惠特菲尔德·迪菲（Whitfield Diffie）和马丁·赫尔曼（Martin Hellman）提出，两人在2015年获得了图灵奖。

根据素数P、生成元G和“$G^X \bmod P$”求出X的问题就是“离散对数问题”，人们至今还未找到这个问题的解法，而迪菲–赫尔曼密钥交换正是利用了这个数学难题。

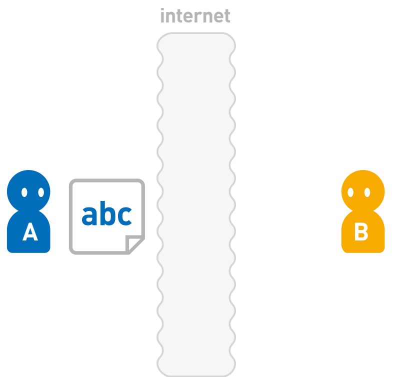
补充说明

使用迪菲–赫尔曼密钥交换，通信双方仅通过交换一些公开信息就可以实现密钥交换。但实际上，双方并没有交换密钥，而是生成了密钥。因此，该方法又被叫作“迪菲–赫尔曼密钥协议”。

No. 5-8 消息认证码

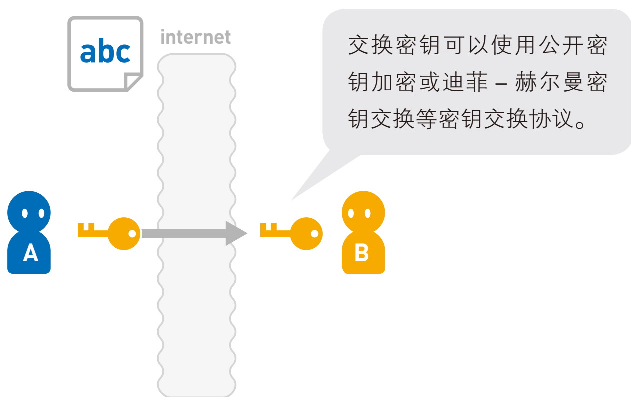
消息认证码可以实现“认证”和“检测篡改”这两个功能。密文的内容在传输过程中可能会被篡改，这会导致解密后的内容发生变化，从而产生误会。消息认证码就是可以预防这种情况发生的机制。

01



首先，我们来看看什么情况下需要使用消息认证码。假设A在B处购买商品，需要将商品编号abc告诉B。

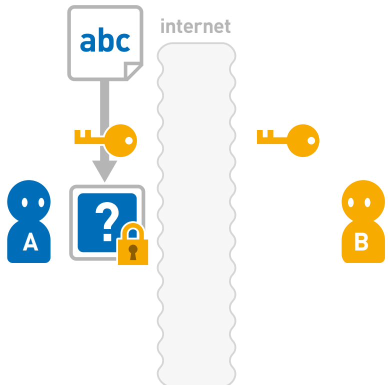
02



此处，假设A使用共享密钥加密对消息进行加密。A通过安全的方法将密钥发送给了B。

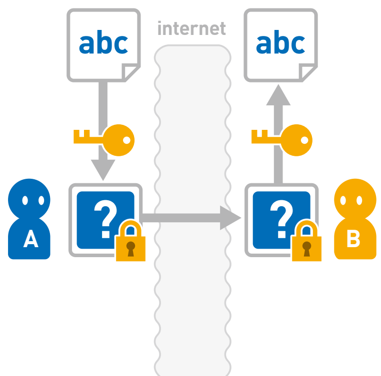
参考：5-4 共享密钥加密
参考：5-5 公开密钥加密
参考：5-7 迪菲 – 赫尔曼密钥交换

03



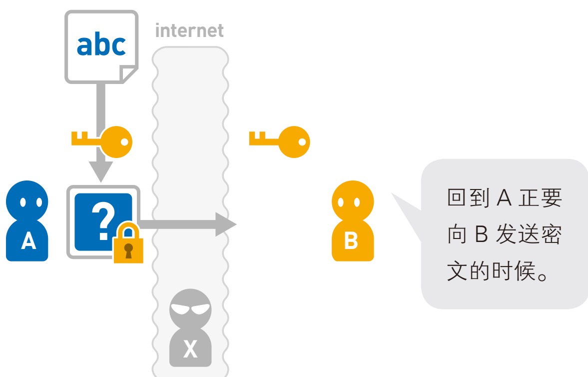
A使用双方共有的密钥对消息进行加密。

04



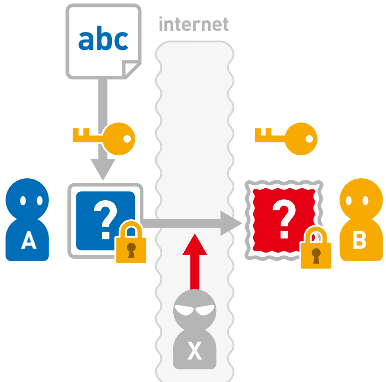
A把密文发送给B，B收到后对密文进行解密，最终得到了原本的商品编号abc。

05



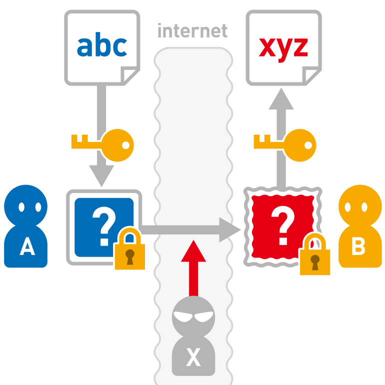
以上是没有出现问题时的流程，然而在这个过程中可能会发生下面的情况。

06



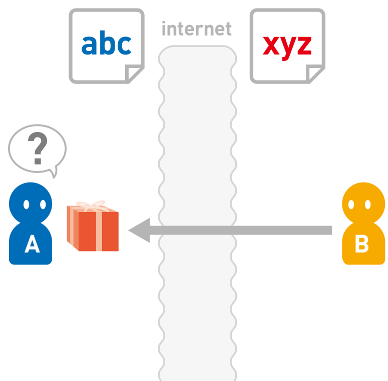
假设A发送给B的密文在通信过程中被X恶意篡改了，而B收到密文后没有意识到这个问题。

07



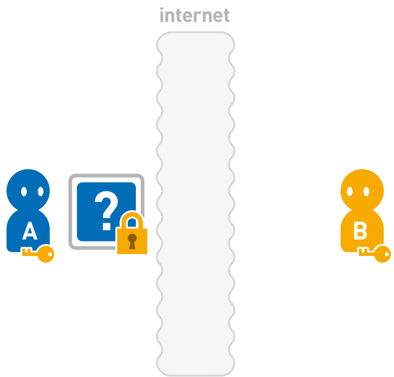
B对被篡改的密文进行解密，得到消息xyz。

08



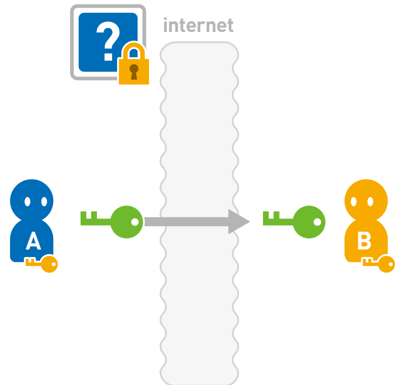
B以为A订购的是编号为xyz的商品，于是将错误的商品发送给了A。

09



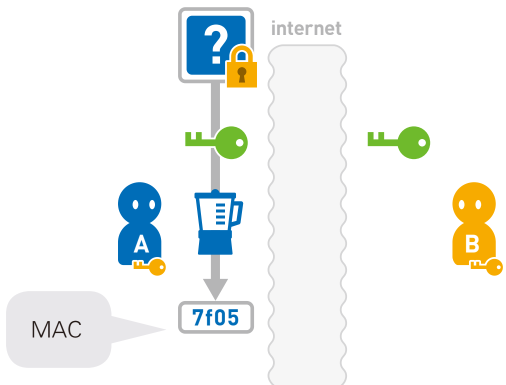
如果使用消息认证码，就能检测出消息已被篡改。为了让大家了解实际的处理流程，我们再一次回到A正要向B发送密文的时候。

10



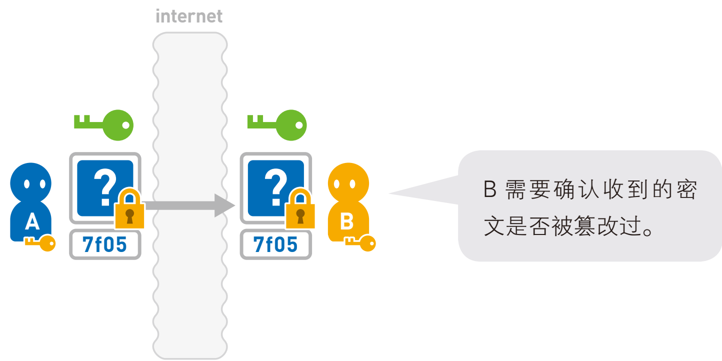
A生成了一个用于制作消息认证码的密钥，然后使用安全的方法将密钥发送给了B。

11



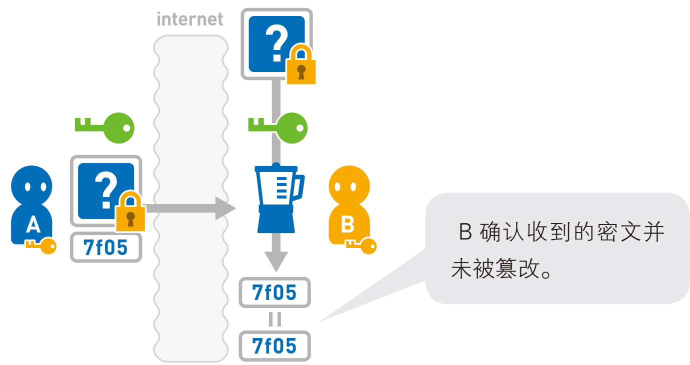
接下来，A使用密文和密钥生成一个值。此处生成的是7f05。这个由密钥和密文生成的值就是消息认证码，以下简称为MAC（Message Authentication Code）。

12



A 将 MAC（7f05）和密文发送给 B。

13



和 A 一样，B 也需要使用密文和密钥来生成 MAC。经过对比，B 可以确认自己计算出来的 7f05 和 A 发来的 7f05 一致。

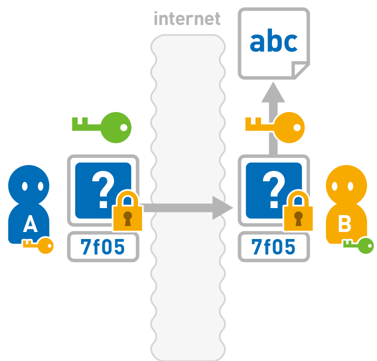
提示

我们可以把 MAC 想象成是由密钥和密文组成的字符串的“哈希值”。计算 MAC 的算法有 HMAC[①]、OMAC[②]、CMAC[③]等。目前，HMAC 的应用最为广泛。

▶参考：5-3 哈希函数

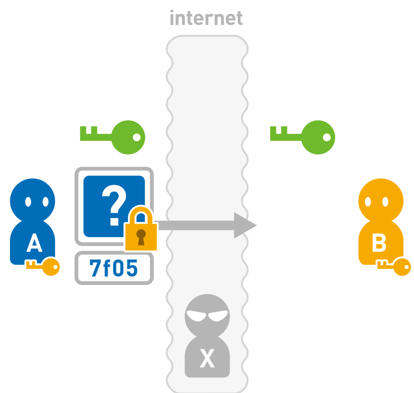
① Hash-based MAC 的缩写。
② One-key MAC 的缩写。
③ Cipher-based MAC 的缩写。

14



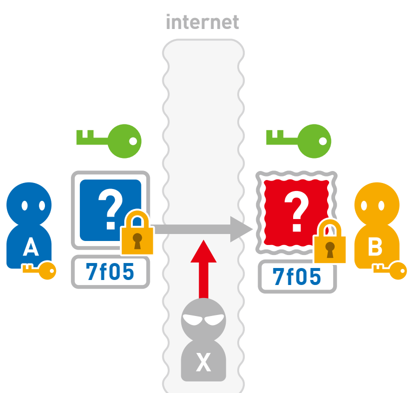
接下来，B只需使用密钥对密文进行解密即可。最终B成功取得了A发送过来的商品编号abc。

15



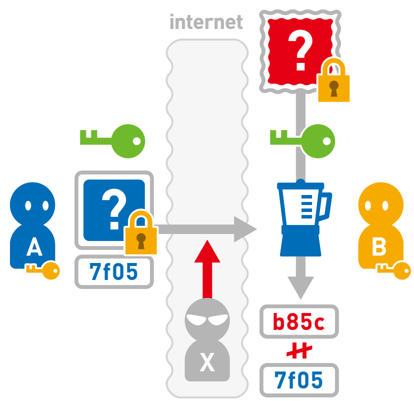
X在通信过程中对密文进行了篡改是怎样一种情况呢？让我们回到A正要向B发送密文的时候。

16



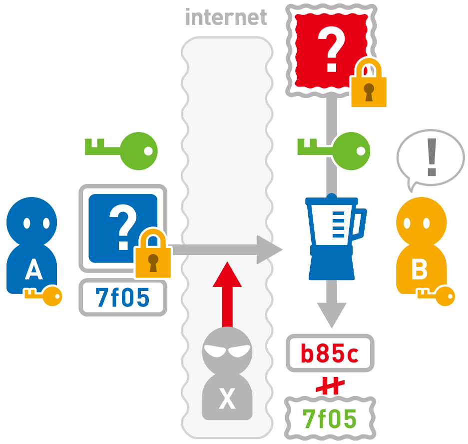
假设在A向B发送密文和MAC时，X对密文进行了篡改。

17



B使用该密文计算MAC，得到的值是b85c，发现和收到的MAC不一致。

18



由此，B意识到密文或者MAC，甚至两者都可能遭到了篡改。于是B废弃了收到的密文和MAC，向A提出再次发送的请求。

解说

加密仅仅是一个数值计算和处理的过程，所以即使密文被篡改了，也能够进行解密相关的计算。

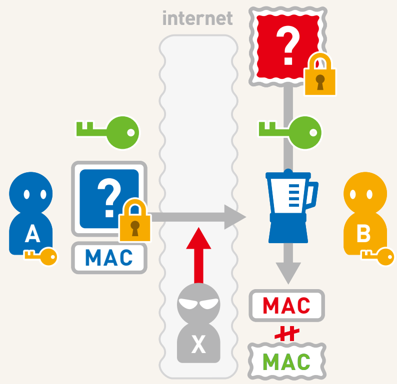
如果原本的消息是很长的句子，那么它被篡改后意思会变得很奇怪，所以接收者有可能会发现它是被篡改过的。

但是，如果原本的消息就是商品编号等无法被人们直接理解的内容，那么解密后接收者便很难判断它是否被篡改。由于密码本身无法告诉人们消息是否被篡改，所以就需要使用消息验证码来检测。

补充说明

本节以图配文讲解了使用 MAC 检测密文是否被篡改的方法，接着我们来进一步思考：X 会不会为了让密文的篡改变得合理，而对 MAC 也进行篡改呢？

X 没有计算 MAC 的密钥，所以即便他可以篡改 MAC，也无法让篡改后的密文变得合理。所以，只要 B 计算出 MAC，发现密文对应的 MAC 与自己算出的不同，就能确认通信过程中发生了篡改（请参考下图）。



就像这样，只要使用消息认证码 MAC，我们就能预防通信过程中的篡改行为。

然而，这种方法也有缺点。在使用消息认证码的过程中，AB 双方都可以对消息进行加密并且算出 MAC。也就是说，我们无法证明原本的消息是 A 生成的还是 B 生成的。

因此，假如 A 是坏人，他就可以在自己发出消息后声称“这条消息是 B 捏造的”，而否认自己的行为。如果 B 是坏人，他也可以自己准备一条消息，然后声称“这是 A 发给我的消息”。

使用 MAC 时，生成的一方和检测的一方持有同样的密钥，所以不能确定 MAC 由哪方生成。这个问题可以用下一节将会讲到的“数字签名”来解决。

▶参考：5-9 数字签名

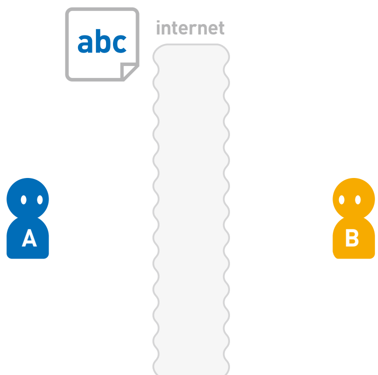
No. 5-9

数字签名

数字签名不仅可以实现消息认证码的认证和检测篡改功能，还可以预防事后否认问题的发生。由于在消息认证码中使用的是共享密钥加密，所以持有密钥的收信人也有可能是消息的发送者，这样是无法预防事后否认行为的。而数字签名是只有发信人才能生成的，因此使用它就可以确定谁是消息的发送者了。

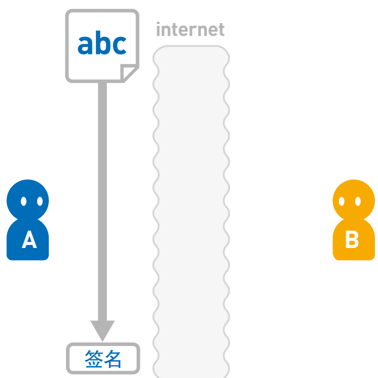
▶参考：5-8 消息认证码

01



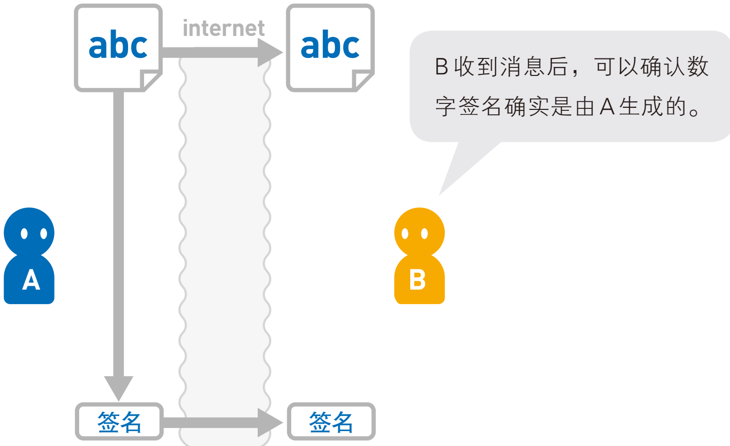
首先，我们来看一看数字签名的特征。假设A要向B发送消息。

02



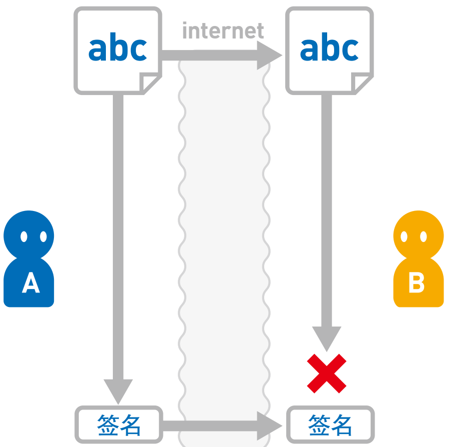
在发送前A给消息加上数字签名。数字签名只能由A生成。

03



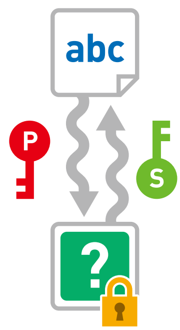
只要发送的消息上有 A 的数字签名，就能确定消息的发送者就是 A。

04



B 可以验证数字签名的正确性，但无法生成数字签名。

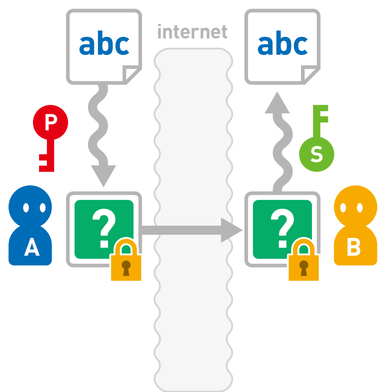
05



接下来看一看数字签名具体是怎样生成的吧。数字签名的生成使用的是公开密钥加密。

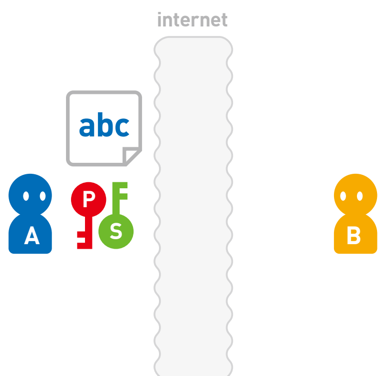
▶参考：5-5 公开密钥加密

06



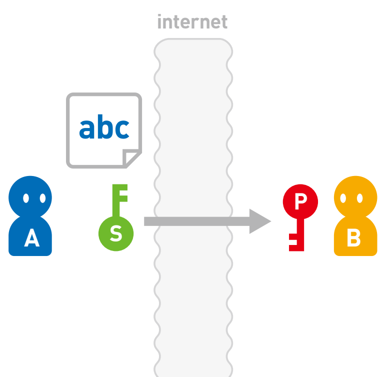
我们先来复习一下前面的知识。公开密钥加密中，加密使用的是公开密钥，解密使用的是私有密钥。任何人都可以使用公开密钥对数据进行加密，但只有持有私有密钥的人才能解密数据。然而，数字签名却是恰恰相反的。

07



下面，我们就来看一看使用了数字签名的消息交换流程。首先由A准备好需要发送的消息、私有密钥和公开密钥。由消息的发送者来准备这两个密钥，这一点与公开密钥加密有所不同。

08

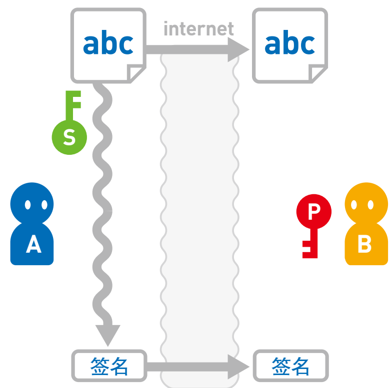


A将公开密钥发送给B。

09

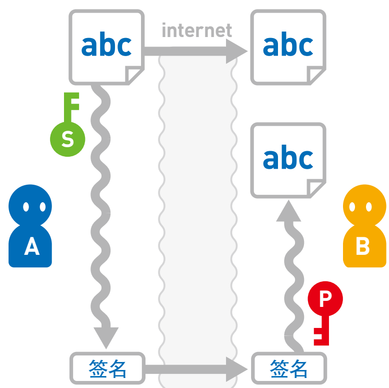
A使用私有密钥加密消息。加密后的消息就是数字签名。

10



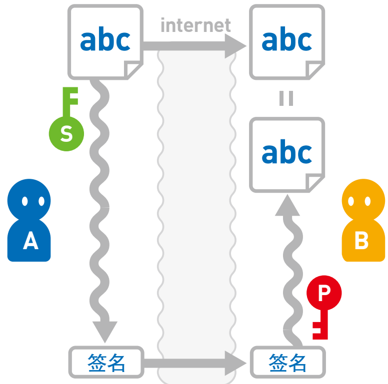
A将消息和签名都发送给了B。

11



B使用公开密钥对密文（签名）进行解密。

12



B 对解密后的消息进行确认，看它是否和收到的消息一致。流程到此结束。

解说

在步骤 07~ 步骤 12 中，生成的是“只能由持有私有密钥的 A 来加密，但只要有公开密钥，谁都可以进行解密的密文”。这个密文作为密码似乎没有任何意义。但是换一个角度来看就会发现，它可以保证这个密文的制作者只能是持有私有密钥的 A。

在数字签名中，是将“只能由 A 来加密的密文”作为“签名”来使用的。严格来说，也有使用加密运算以外的计算方法来生成签名的情况。但是，用私有密钥生成签名、用公开密钥验证签名这一机制是相同的，所以为了方便我们就以上文的方式进行了说明。

在公开密钥加密中，用公开密钥加密的数据都可以用私有密钥还原。而本节讲解的数字签名利用的是用私有密钥加密的数据，用公开密钥解密后就能还原这一性质。也就是说，即使密钥的使用顺序不同，运行结果也都是一样的。并不是所有的公开密钥加密都具有这个性质，不过 RSA 加密算法是可以的。

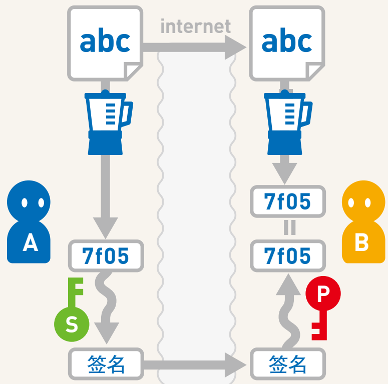
能够用 A 的公开密钥解密的密文，必定是由 A 生成的。因此，我们可以利用这个结论来确认消息的发送者是否为 A，消息是否被人篡改。

由于 B 只有公开密钥，无法生成 A 的签名，所以也预防了“事后否认”这一问题的发生。

补充说明

公开密钥加密的加密和解密都比较耗时。为了节约运算时间，实际上不会对消息直接进行加密，而是先求得消息的哈希值，再对哈希值进行加密，然后将其作为签名来使用（请参考下图）

▶参考：5-3 哈希函数



虽然数字签名可以实现“认证”“检测篡改”“预防事后否认”三个功能，但它也有一个缺陷。

那就是，虽然使用数字签名后 B 会相信消息的发送者就是 A，但实际上也有可能是 X 冒充了 A。

其根本原因在于使用公开密钥加密无法确定公开密钥的制作者是谁。收到的公开密钥上也没有任何制作者的信息。因此，公开密钥有可能是由某个冒充 A 的人生成的。

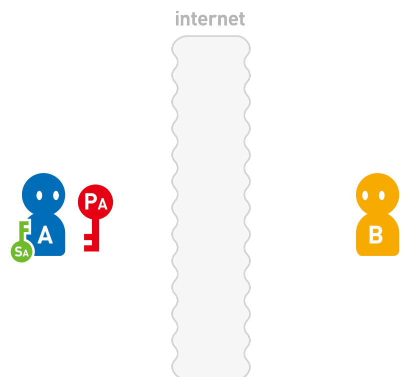
使用下一节将要讲到的“数字证书”就能解决这个问题。

▶参考：5-10 数字证书

No. 5-10 数字证书

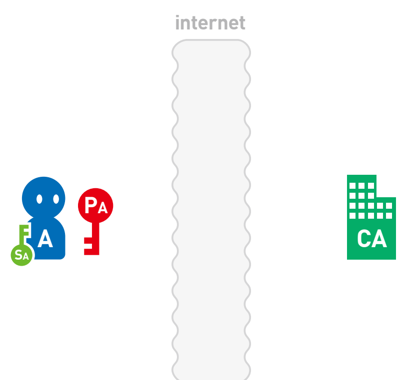
“公开密钥加密”和“数字签名”无法保证公开密钥确实来自信息的发送者。因此，就算公开密钥被第三者恶意替换，接收方也不会注意到。不过，如果使用本节讲解的数字证书，就能保证公开密钥的正确性。

01



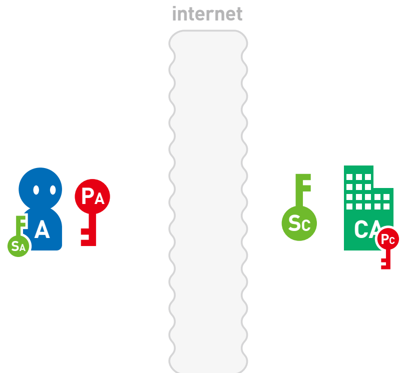
A持有公开密钥 P_A 和私有密钥 S_A，现在想要将公开密钥 P_A 发送给B。

02



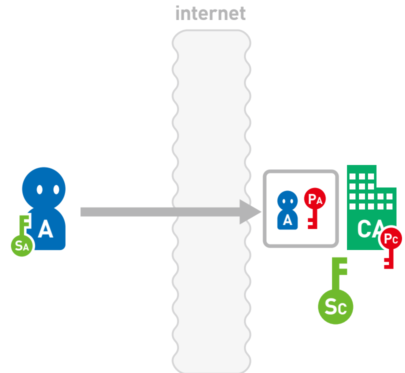
A首先需要向认证中心（Certification Authority，CA）申请发行证书，证明公开密钥 P_A 确实由自己生成。

03



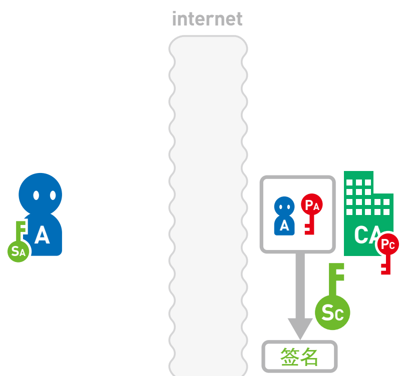
认证中心里保管着他们自己准备的公开密钥 P_C 和私有密钥 S_C 。

04



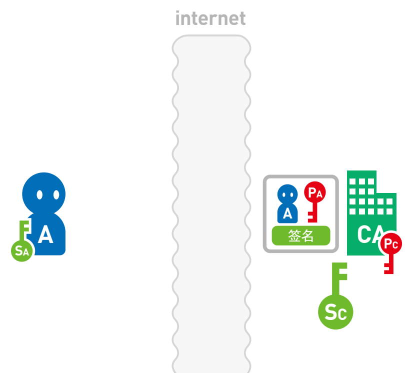
A 将公开密钥 P_A 和包含邮箱信息的个人资料发送给认证中心。

05



认证中心对收到的资料进行确认，判断其是否为 A 本人的资料。确认完毕后，认证中心使用自己的私有密钥 S_C，根据 A 的资料生成数字签名。

06

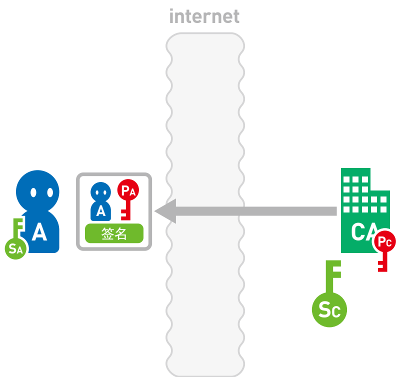


认证中心将生成的数字签名和资料放进同一个文件中。

提示

认证中心是管理数字证书的组织机构。原则上谁都可以成为认证中心，所以认证中心的数量也比较多，但建议在经过政府审查的大型企业机构进行申请，这些机构更令人放心。

07

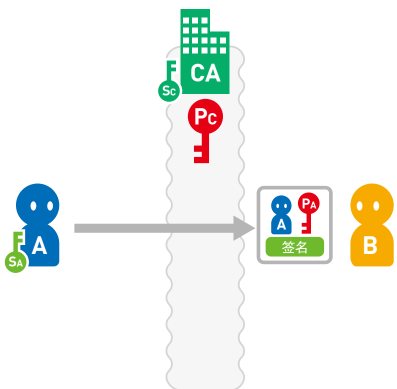


然后，把这个文件发送给A。

08

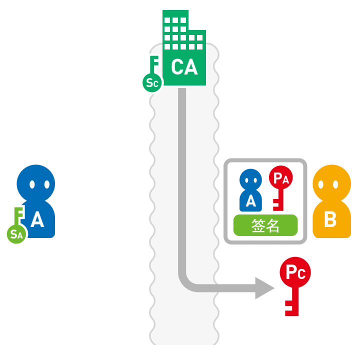
这个文件就是A的数字证书。

09



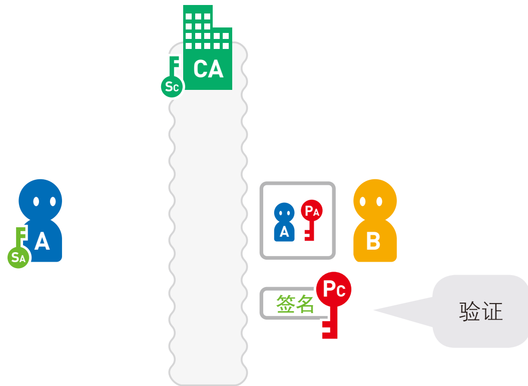
A将作为公开密钥的数字证书发送给了B。

10



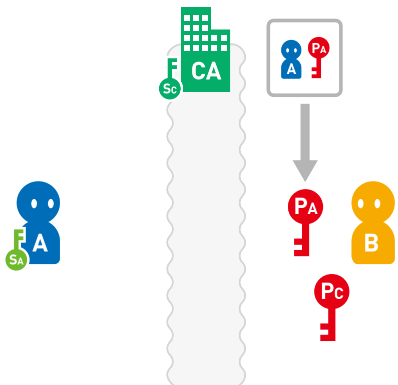
B收到数字证书后，确认证书里的邮件地址确实是A的地址。接着，B获取了认证中心的公开密钥。

11



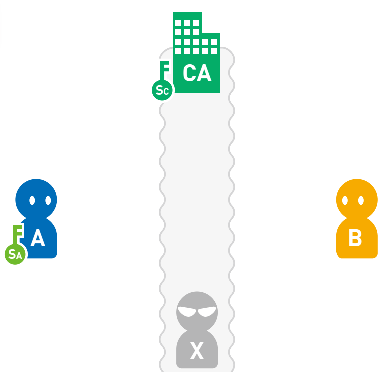
B对证书内的签名进行验证，判断它是否为认证中心给出的签名。证书中的签名只能用认证中心的公开密钥P_C进行验证。如果验证结果没有异常，就能说明这份证书的确由认证中心发行。

12



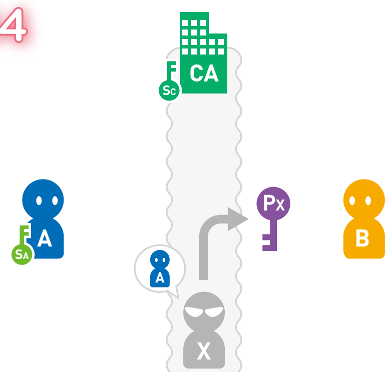
确认了证书是由认证中心发行的，且邮件地址就是A的之后，B从证书中取出A的公开密钥P_A。这样，公开密钥便从A传到了B。

13



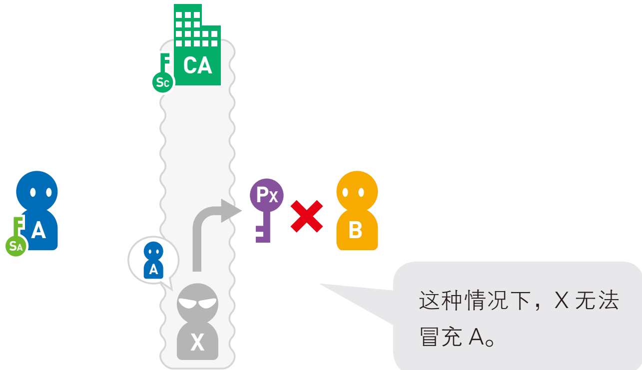
我们来看看公开密钥的交付过程有没有什么问题。

14



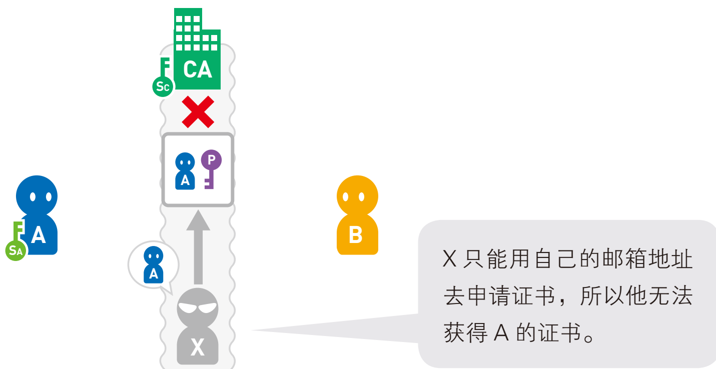
假设X冒充A，准备向B发送公开密钥P_X。

15



但是，B 没有必要信任以非证书形式收到的公开密钥。

16



假设 X 为了假冒 A，准备在认证中心登记自己的公开密钥。然而 X 无法使用 A 的邮箱地址，因此无法获得 A 的证书。

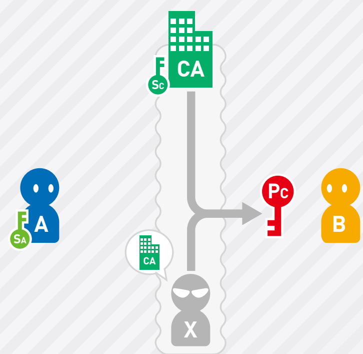
解说

通过数字证书，信息的接收者可以确认公开密钥的制作者。

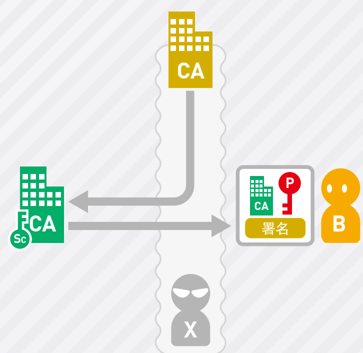
在步骤 10 中，B 得到了认证中心的公开密钥，但此处仍有一个疑问。

那就是，B 得到的公开密钥 P_C 真的来自认证中心吗？

由于公开密钥自身不能表示其制作者，所以有可能是冒充认证中心的 X 所生成的。也就是说，这里同样存在公开密钥问题（请参考下图）。



实际上，认证中心的公开密钥 P_C 是以数字证书的形式交付的，会有更高级别的认证中心对这个认证中心署名（请参考下图）。



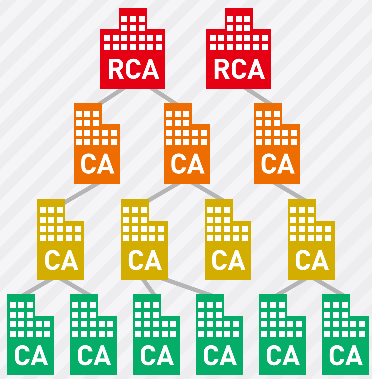
就像下页图中的树结构一样，由上面的认证中心为下面的认证中心发行证书。

那么，我们来看看这个树结构是怎么形成的吧。假设存在一个被社会广泛认可的认

证中心 A。此时出现了一个刚成立的公司 B，虽然 B 想要开展认证中心的业务，但它无法得到社会的认可。

于是 B 向 A 申请发行数字证书。当然 A 会对 B 能否开展认证中心业务进行适当的检测。只要 A 发行了证书，公司 B 就可以向社会表示自己获得了公司 A 的信任。于是，通过大型组织对小组织的信赖担保，树结构就建立了起来。

最顶端的认证中心被称为“根认证中心”（root CA），其自身的正当性由自己证明。对根认证中心自身进行证明的证书为“根证书”。如果根认证中心不被信任，整个组织就无法运转。因此根认证中心多为大型企业，或者与政府关联且已经取得了社会信赖的组织。



补充说明

到目前为止，我们了解的都是个人之间交付公开密钥的例子，其实在网站之间的通信中同样也要用到数字证书。只要能收到来自网站的含有公开密钥的证书，就能确认该网站未被第三者冒充。

此处的证书叫作“服务器证书”，同样由认证中心发行。个人的证书会与他的邮箱信息相对应，而服务器证书与域名信息相对应。因此，我们还可以确认网站域名和存储网站本身内容的服务器是由同一个组织来管理的。

数字证书就是像这样通过认证中心来担保公开密钥的制作者。这一系列技术规范被统称为“公钥基础设施”(Public Key Infrastructure，PKI)。

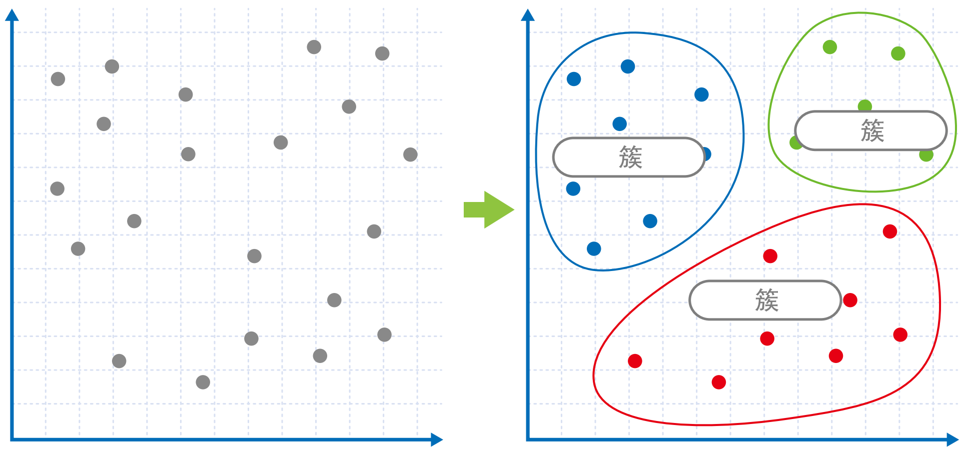
第 6 章

聚类

No. 6-1 什么是聚类

将相似的对象分为一组

聚类就是在输入为多个数据时，将“相似”的数据分为一组的操作。1 个组就叫作 1 个“簇”。下面的示例中每个点都代表 1 个数据，在平面上位置较为相近、被圈起来的点就代表一类相似的数据。也就是说，这些数据被分为了 3 个簇。



如何定义“相似”

▶ 定义数据间的差距

根据数据类型不同，定义该数据是否“相似”的标准也不同。具体来说，就是要对两个数据之间的“差距”进行定义。首先来看下面的示例。

假设某所高中的某个年级中共有 400 名学生，现在我们想要将这些学生在考试中取得的语文、数学、英语成绩数据化，并将他们按照“擅长或不擅长的科目相似”进行聚类。

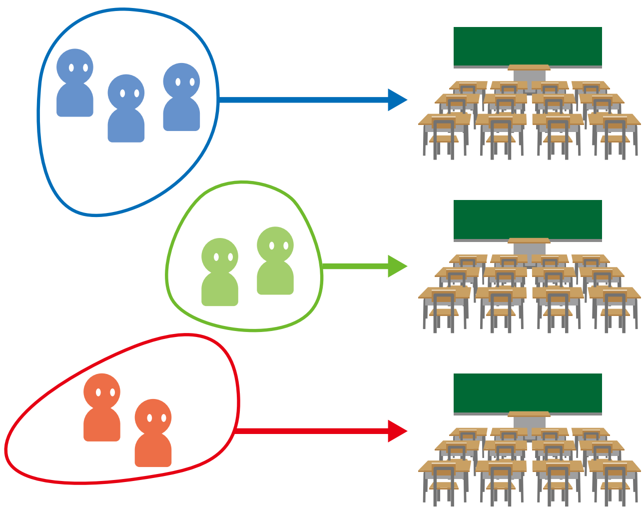
把每个学生都转换成“(语文成绩，数学成绩，英语成绩)”形式的数据后，就可以将两个数据（c_1, m_1, e_1）和（c_2, m_2, e_2）之间的差距定义为 $(c_1-c_2)^2+(m_1-m_2)^2+(e_1-e_2)^2$，其中差距小的数据就互为“相似的数据”。

▶ 符合条件的算法

即使定义好了数据间的差距，聚类的方法也会有很多种。我们可以设定各种各样的条件，比如想把数据分为 10 个簇，或者想把 1 个簇内的数据定在 30~50 人之间，再或者想把簇内数据间的最大距离设为 10，等等。而设定什么样的条件取决于进行聚类的目的。

假如是为了开办暑期补习班而对学生进行分班，那么就要根据老师和教室的数量来确定“簇的数量”，并根据教室的面积确定“每个簇内的数据量”。现在有很多种可以满足各类条件的聚类算法供我们选择。

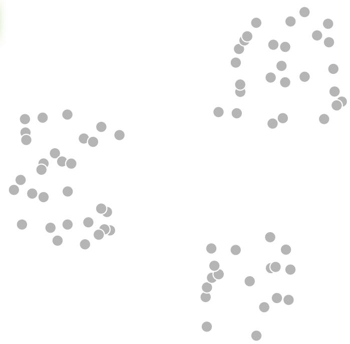
下一节就将介绍其中最基本，也是最有代表性的聚类算法“k-means 算法”。该算法可以把数据按要求分为 k 个簇。



No. 6-2 *k*-means 算法

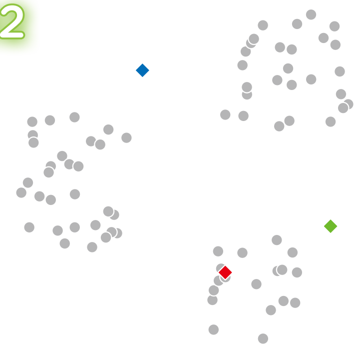
k-means 算法是聚类算法中的一种，它可以根据事先给定的簇的数量进行聚类。

01



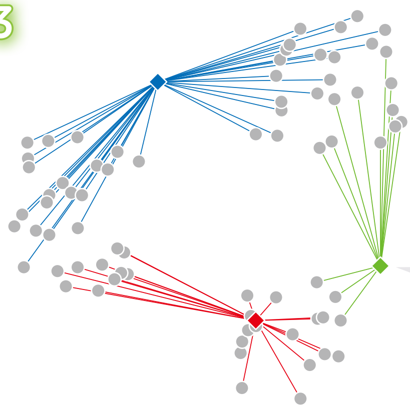
首先准备好需要聚类的数据，然后决定簇的数量。本例中我们将簇的数量定为3。此处用点表示数据，用两点间的直线距离表示数据间的差距。

02



随机选择3个点作为簇的中心点。

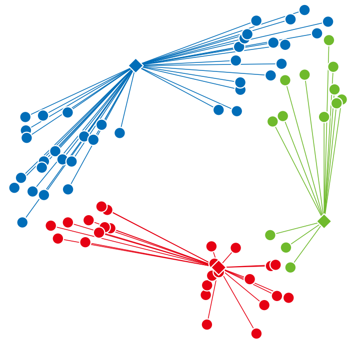
03



此处用不同颜色的线连接各个数据和距离该数据最近的中心点。

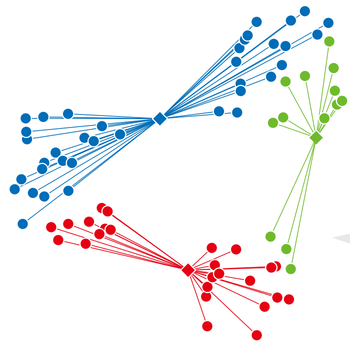
计算各个数据分别和3个中心点中的哪一个点距离最近。

04



将数据分到相应的簇中。这样，3个簇的聚类就完成了。

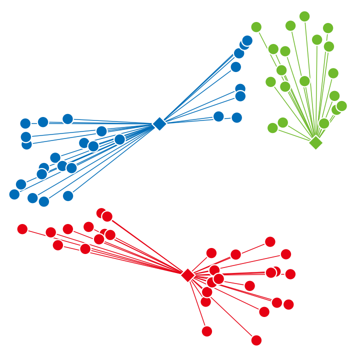
05



随着中心点的移动，部分数据的“距离自己最近的中心点”也会改变。

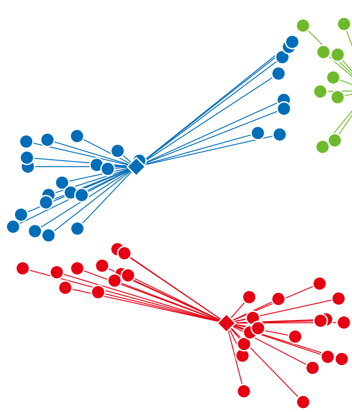
计算各个簇中数据的重心，然后将簇的中心点移动到这个位置。

06



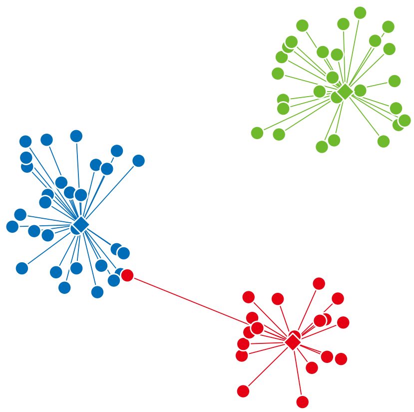
重新计算距离最近的簇的中心点，并将数据分到相应的簇中。

07



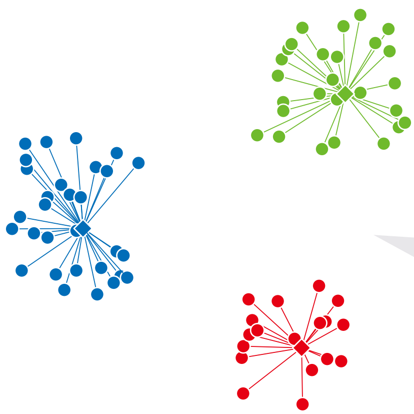
重复执行“将数据分到相应的簇中”和“将中心点移到重心的位置”这两个操作，直到中心点不再发生变化为止。

08



3轮操作结束后，结果如上图所示。

09



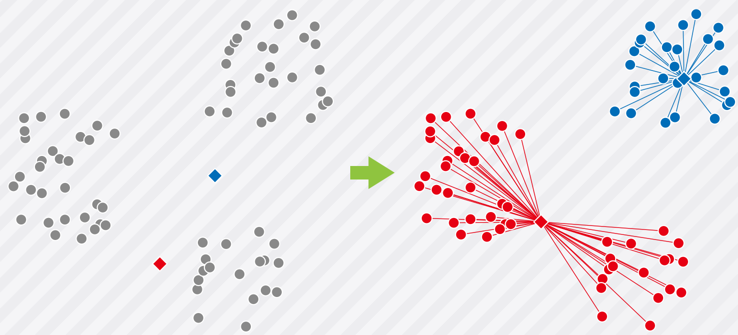
此时我们可以确定，相似的数据已经被恰当地分为一组了。

4轮操作结束后，结果如上图所示。即使继续重复操作，中心点也不会再发生变化，操作到此结束，聚类也就完成了。

解说

k-means 算法中，随着操作的不断重复，中心点的位置必定会在某处收敛，这一点已经在数学层面上得到证明。

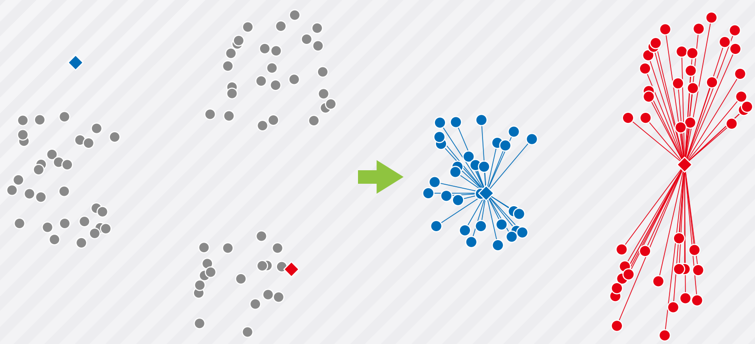
前面的例子中我们将簇的数量定为 3，若现在使用同样的数据，将簇的数量定为 2，那么聚类将如下图所示。



位于左边和下边的两个数据块被分到了一个簇中。就像这样，由于 *k*-means 算法需要事先确定好簇的数量，所以设定的数量如果不合理，运行的结果就可能会不符合我们的需求。

如果对簇的数量没有明确要求，那么我们可以事先对数据进行分析，推算出一个合适的数量，或者不断改变簇的数量来试验 *k*-means 算法。

另外，如果簇的数量同样为 2，但中心点最初的位置不同，那么也可能会出现下图这样的聚类结果。



与之前的情况不同，这次右上和右下的两个数据块被分到了一个簇中。也就是说，即使簇的数量相同，只要随机设置的中心点最初的位置不同，聚类的结果也会产生变化。因此，我们可以通过改变随机设定的中心点位置来不断尝试 *k*-means 算法，再从中选择最合适的聚类结果。

补充说明

除了 *k*-means 算法以外，聚类算法还有很多，其中“层次聚类算法”较为有名。与 *k*-means 算法不同，层次聚类算法不需要事先设定簇的数量。

在层次聚类算法中，一开始每个数据都自成一类。也就是说，有 n 个数据就会形成 n 个簇。然后重复执行“将距离最近的两个簇合并为一个”的操作 $n-1$ 次。每执行 1 次，簇就会减少 1 个。执行 $n-1$ 次后，所有数据就都被分到了一个簇中。在这个过程中，每个阶段的簇的数量都不同，对应的聚类结果也不同。只要选择其中最为合理的 1 个结果就好。

合并簇的时候，为了找出“距离最近的两个簇”，需要先对簇之间的距离进行定义。根据定义方法不同，会有“最短距离法”“最长距离法”“中间距离法”等多种算法。

第 7 章

其他算法

No. 7-1

欧几里得算法

欧几里得算法（又称辗转相除法）用于计算两个数的最大公约数，被称为世界上最古老的算法。现在人们已无法确定该算法具体的提出时间，但其最早被发现记载于公元前 300 年欧几里得的著作中，因此得以命名。

01

1112　　695

在学习欧几里得算法之前，我们先来看一看数字 1112 和 695 的最大公约数是多少吧。

02

$$1112 = \boxed{139} \times 2 \times 2 \times 2$$

$$695 = \boxed{139} \times 5$$

$$\boxed{139} \cdots \text{GCD}$$

通常的做法是先对两个数字因式分解，找出共同的素数，然后求出最大公约数（GCD）。这样就能求出 1112 和 695 的最大公约数为 139。然而两个数字越大，因式分解就越难。此时，使用欧几里得算法就能更高效地求解最大公约数。

03

1112　　695

那么，我们就来看一看欧几里得算法的具体操作流程吧。

04

1112 mod 695 =

首先用较小的数字去除较大的数字，求出余数。也就是对两个数字进行mod运算。我们在第5章也讲过mod运算即取余运算，A mod B 就是算出 A 除以 B 后的余数 C。

05

1112 mod 695 = 417

除完后的余数为417。

06

1112 mod 695 = 417
695 mod 417 = 278

接下来再用除数695和余数417进行mod运算。结果为278。

07

1112 mod 695 = 417
695 mod 417 = 278
417 mod 278 = 139

继续重复同样的操作，对417和278进行mod运算，结果为139。

08

1112 mod 695 = 417
695 mod 417 = 278
417 mod 278 = 139
278 mod 139 = 0

对278和139进行mod运算，结果为0。也就是说，278可以被139整除。

09

1112 mod 695 = 417

695 mod 417 = 278

417 mod 278 = 139

278 mod 139 = 0

139 … GCD

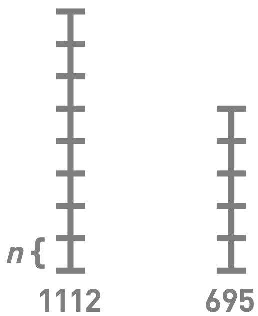
余数为0时，最后一次运算中的除数139就是1112和695的最大公约数。

10

分别用相应长度的直线来表示 1112 和 695。

为什么用欧几里得算法可以求得最大公约数呢？我们结合图片来想一想。

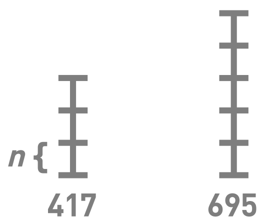
11



实际上我们并不知道两条直线可以被划分为多少个刻度，但是可以确定1112和695一定是最大公约数n的整数倍。

将最大公约数设为n，然后在直线上画出相应刻度。由于我们已知最大公约数为139，所以为了方便理解，在1112上画出8个刻度，在695上画出5个刻度。

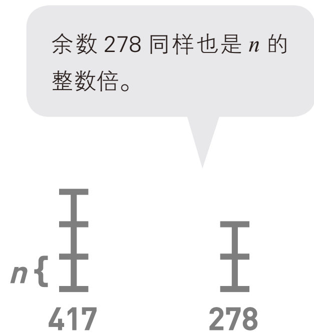
12



从这张图就能看出，417正好也可以用长度为 n 的刻度划分。

这里和前面的运算一样，用小的数去除大的数，得到的余数为417。

13



继续重复mod运算。用417去除695，得到余数278。

14

继续做除法。由于278可以被 139整除，所以……

15

余数为0。此时便能求得最大公约数n为139。

解说

使用欧几里得算法，只需重复做除法便能求得最大公约数。这个算法最大的优势就在于即使两个数字再大，只要按照步骤进行操作就能高效地求得两者的最大公约数。

No. 7-2 素性测试

素性测试是判断一个自然数是否为素数的测试。素数（prime number）就是只能被 1 和其自身整除，且大于 1 的自然数。素数从小到大有 2、3、5、7、11、13……目前在加密技术中被广泛应用的 RSA 算法就会用到大素数，因此“素性测试”在该算法中起到了重要的作用。

参考：5-5 公开密钥加密

01

3599

我们来试着判断3599是否为素数吧。简单的方法便是将3599按顺序除以比2大的数字，看是否能被整除。“整除”就是指mod运算的结果为0。由于3599的平方根为59.99…，所以只需要除以从2到59的数字。

02

3599 ≠ 素数

3599 mod 2 = 1

3599 mod 3 = 2

⋮

3599 mod 58 = 3

3599 mod 59 = 0

根据mod运算的结果可知，3599能被59整除。也就是说，3599并不是素数。但假如需要判断的数非常大，这种方法就十分耗费时间。这种时候就可以用“费马测试”来解决这个问题。

03

费马测试

5

费马测试被称为概率性素性测试，它判断的是“某个数是素数的概率大不大”。在讲解费马测试之前，我们先来学习基础知识——素数的性质。首先来看一看素数5有什么样的性质。

04

5 = 素数

$4^5 = 1024$

$3^5 = 243$

$2^5 = 32$

$1^5 = 1$

对于比5小的数，分别计算它们的5次方，结果如上图所示。

05

5 = 素数

$4^5\ (= 1024)\quad \bmod 5 = 4$

$3^5\ (= 243)\quad \bmod 5 = 3$

$2^5\ (= 32)\quad \bmod 5 = 2$

$1^5\ (= 1)\quad \bmod 5 = 1$

接下来，再对结果分别进行mod运算，求得它们除以5后的余数，结果如上图所示。

06

5 = 素数

$4^5\ (= 1024)\quad \bmod 5 = 4$

$3^5\ (= 243)\quad \bmod 5 = 3$

$2^5\ (= 32)\quad \bmod 5 = 2$

$1^5\ (= 1)\quad \bmod 5 = 1$

观察原本的数和余数，发现两者一致。

07

5 = 素数

$n < 5$

$n^5 \bmod 5 = n$

由此，可以推导出关于素数5，以上公式成立。

08

$$p = \text{素数}$$

$$n < p$$

$$n^p \bmod p = n$$

实际上不只是5，对于任意素数p，上面的公式都是成立的。这就是“费马小定理”。根据是否满足费马小定理来判断一个数是否为素数的方法就是“费马测试”。

09

113

那么，我们就用费马测试来判断一下113是否为素数吧。

10

113 = 素数（?）

$$64^{113} \bmod 113 = 64$$ ✔

$$29^{113} \bmod 113 = 29$$ ✔

$$15^{113} \bmod 113 = 15$$ ✔

随机选择3个比113小的数作为n，计算这些数的113次方，再用113去除得到的结果，求出余数。3个数最后得到的余数都和原本的数相同，因此可以判断113是素数。

解说

确认n和余数一致的次数越多，需要判断的数确实为素数的可能性就越大。但是，如果每一个小于p的数都要去计算，就会非常耗费时间。实际上，如果确认了几组n和余数之后就能判断该数是素数的可能性非常高，那么大致就可以判定该数是素数了。

比如在RSA算法中，用于素性测试的是根据费马测试改进而来的“米勒–拉宾（Miller-Rabin）素性测试”。用这个方法重复进行测试后，当数不是素数的概率小于0.5^{80}时，就可以大致判断该数为素数。

补充说明

如果 p 是素数，那么所有比 p 小的数 n 都满足 $n^p \bmod p = n$ 这个条件。但反过来，即使所有 n 都满足条件，p 也有可能不是素数。因为在极低概率下会出现所有 n 都满足条件的合数（非素数的自然数）。

比如数字 561 可以表示为 $3 \times 11 \times 17$，所以 561 是合数，不是素数。然而比 561 小的所有数字都满足上述条件。

这样的合数就叫作“卡迈克尔数”（Carmichael numbers），也叫“绝对伪素数”。下图按从小到大的顺序列举了一些卡迈克尔数，可以看出这种数字非常少。

561	1105	1729
2465	2821	6601
8911	10585	15841
29341	41041	46657
52633	62745	63973

小知识

在过去很长一段时间内，人们都无法确定是否可以应用一个算法在输入规模的多项式时间内进行（非概率意义，而是决定性的）素性测试。然而在2002年，印度的三位数学家证明了它的可行性。他们提出的就是“AKS算法”，该名称也是以他们名字的首字母来命名的。不过，虽说是“在多项式时间内”，但该算法的计算次数仍然较多，所以费马测试这样的高速算法更为实用，应用范围也更加广泛。

No. 7-3

网页排名

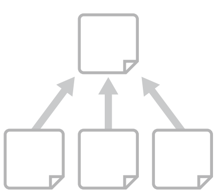
网页排名（PageRank，也叫作佩奇排名）是一种在搜索网页时对搜索结果进行排序的算法。Google 因在搜索引擎中使用了这个算法而成为了世界知名的大企业是众所周知的事情。

01

在搜索结果中越是排在前面的网页，其价值也就越高。

网页排名就是利用网页之间的链接结构计算出网页价值的算法。我们来看看具体的计算流程吧。

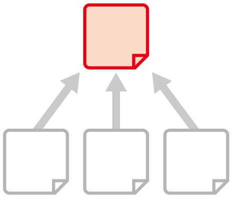
02



在这张图中，下面的 3 个网页都链向了上面的网页。

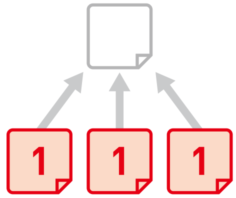
这里假设方块表示网页，箭头表示网页间的链接关系。在网页排名中，链入页面越多的网页，它的重要性也就越高。

03



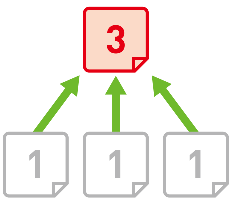
从这张图中我们可以看出，上面网页的重要性最高。实际上，各个网页的重要性会通过计算来数值化。下面我们就来了解计算方法的基本思路。

04



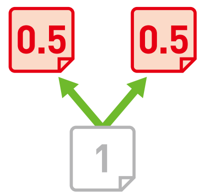
假设没有链入页面的网页权重为1。

05



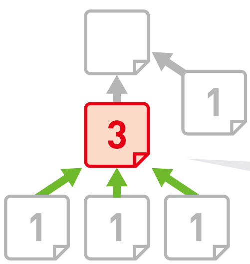
有链入页面的网页权重是其链入页面的权重之和。

06



如果一个网页链向多个页面，那么其链向的所有页面将平分它的权重。

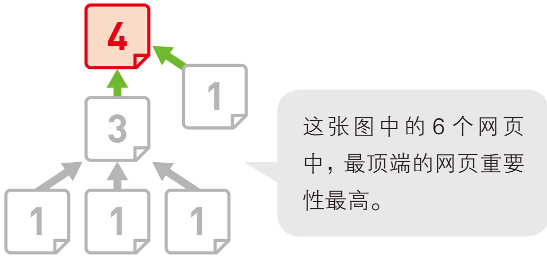
07



在这张图中，中间的网页被3个独立的页面链入，所以权重为3。

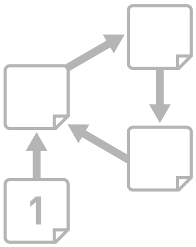
在网页排名中，链入的页面越多，该网页所发出的链接的价值就越高。

08



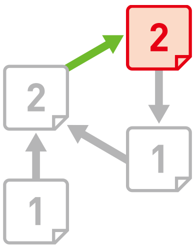
最顶端的网页被权重为3的页面链接，所以权重较高。以上就是网页排名的基本思路。

09



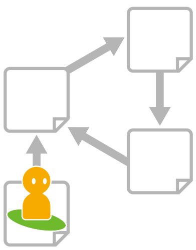
但是，如果在链接结构为环状的情况下使用这种方法，就会出现问题。

10



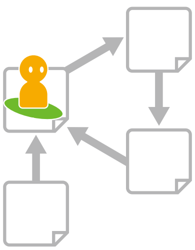
计算各个网页权重的操作会无限循环下去，从而导致环内网页的权重不断增长。

11



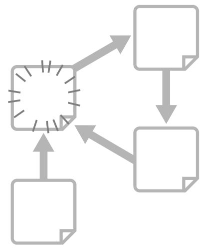
此时可以使用“随机游走模型”（random walk model）来解决这个问题。要想了解这个模型，先得思考一下人们是怎样浏览网页的。

12



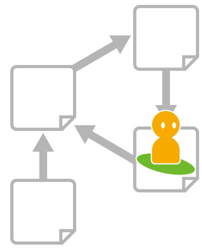
假设有一天，某位互联网用户想浏览从杂志上看到的一个有趣的网页。他从左下角的网页开始浏览，然后通过链接到达其他网页。

13



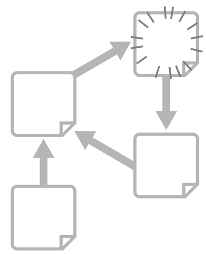
浏览了一些网页后他不想再看了，于是结束浏览。

14



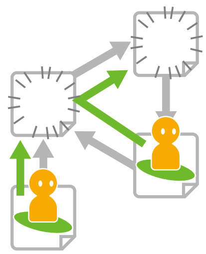
过了几天，他又开始浏览朋友推荐的一个与之前完全不同的网页。

15



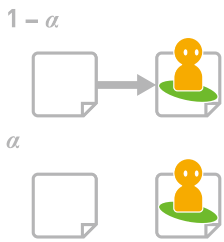
这次也通过链接浏览了别的网页，并在一段时间后结束了浏览。就像这样，他重复着从某个网页开始，通过链接浏览了几个网页后结束的流程。

16



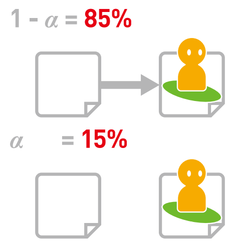
如果我们站在互联网空间的视角来观察上述流程，就会觉得浏览网页的人只是在不断重复着“在有链接指向的页面之间移动几次之后，远程跳转到了完全不相关的网页”这一过程。

17



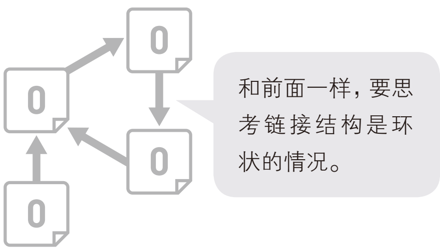
那么，用户浏览网页的操作就可以这样来定义：用户等概率跳转到当前网页所链向的一个网页的概率为 $1-\alpha$；等概率远程跳转到其他网页中的一个网页的概率为 α。

18



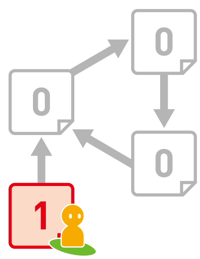
此处我们将远程跳转概率 α 设为15%，并根据前面的定义来模拟一下网页间的跳转过程。

19



网页上的数字代表用户访问该网页的次数。现在模拟还未开始，所有的数字都是0。

20



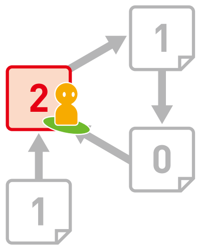
开始根据定义模拟用户浏览网页的过程，各个网页的访问次数也开始出现差距。

小知识

在Google创业初期还没有广告收入的时候，作家兼编辑凯文·凯利向Google创始人之一的拉里·佩奇提过一个问题：“你们为什么要做免费的网络搜索服务？”而对方的回答是“我们实际上是在做AI（人工智能）”。[①]

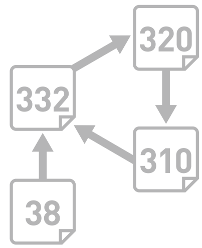
① 出自《必然》（凯文·凯利著，周峰、董理、金阳译，电子工业出版社，2016年）。

21



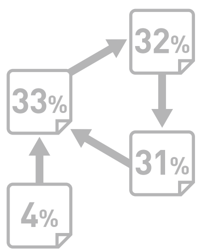
模拟中……

22



模拟该过程直到访问的总次数达到1000次为止，最终结果如上图所示。

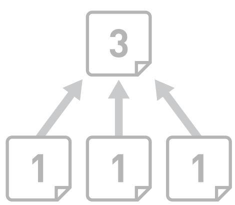
23



使用这个方法，就算网页链接结构为环状，我们也可以得出每个网页的权重。

换算成百分比后，结果如上图所示。这些值代表的是“某一刻正在浏览这个网页的概率”，随机游走模型会直接将其作为网页的权重来使用。[①]

24

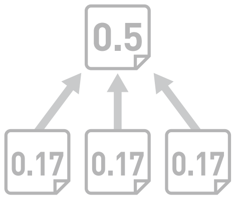


使用上述方法计算该链接构造中的网页权重。

最后，我们来确认网页排名的值和最初介绍的用链接加权计算的结果是否一致。

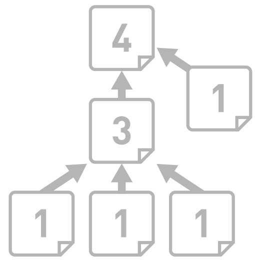
① 实际上一般不会用模拟，而是用更高效的方法来计算。但无论用哪种方法，计算结果几乎都是相同的。

25



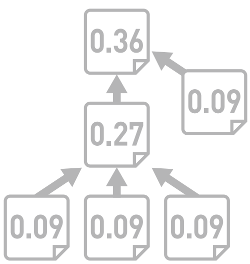
由于各个值都是四舍五入后的结果，所以所有值加起来并不等于1，但分值比例与之前的结果是非常接近的。

26



这是前面讲到的链接结构，试着计算它的权重。

27



网页排名的机制就是像这样，把之前的权重替换为访问页面的概率来进行计算。

可以看出，此处的分值比例也和之前的很相近。

解说

以前的搜索引擎都是以关键词和网页内容的关联性来决定搜索结果的排列顺序，但这种方法没有考虑网页内是否含有有效内容，因此搜索精度较低。

Google 公司提供了使用网页排名算法的搜索引擎，然后凭借其强大的性能成为了世界知名企业。当然，如今决定 Google 搜索结果排序的已不仅仅是网页排名这一个算法了。

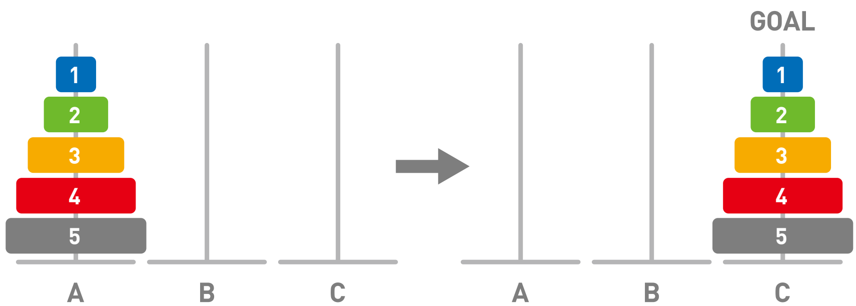
但不管是从利用网页链接结构计算出网页价值这种思路来看，还是从链接形成环状时也能进行计算这点来看，网页排名都是一个划时代的算法。

No. 7-4

汉诺塔

汉诺塔是一种移动圆盘的游戏，同时也是一个简单易懂的递归算法应用示例。

游戏规则



游戏开始时如左图所示，有 3 根柱子 A、B、C，柱子 A 上有 5 个圆盘。把这 5 个圆盘按照原本的顺序移动到柱子 C 上之后游戏就会结束。

移动圆盘时需要遵守以下两个条件。

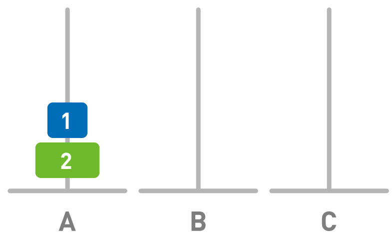
▶ 移动条件

① 1 次只能移动 1 个圆盘。

② 不能把大的圆盘放在小的圆盘上。

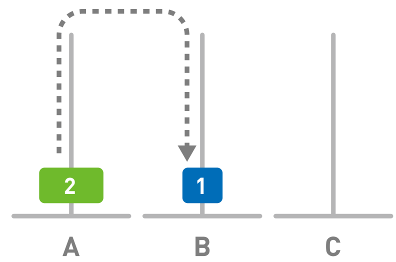
我们的目标就是在这两个条件下，通过把圆盘往 B 或 C 移动来完成游戏。

01



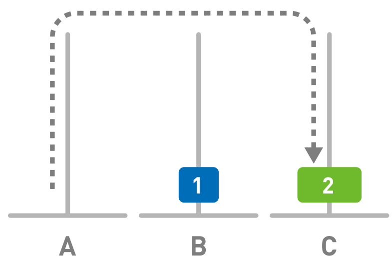
先来看看只有2个圆盘的情况吧。

02



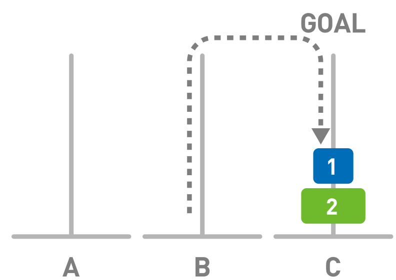
小的圆盘在最顶端，所以可以把它移动到B上。

03



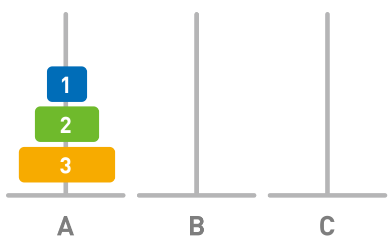
把大的圆盘移动到C上。

04



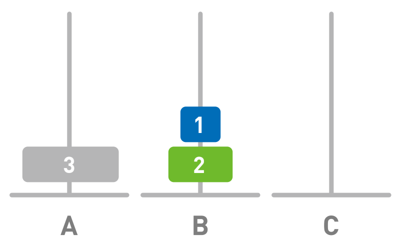
再把小圆盘往C移动，操作完毕。由于只有2个圆盘，此时便可以确认游戏结束。

05



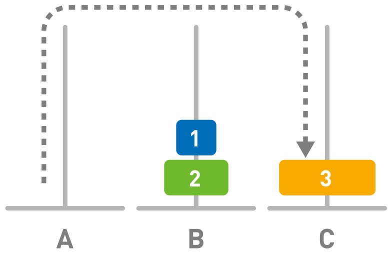
3个圆盘时又会怎样呢？我们可以忽略最大的圆盘，先将其他的圆盘往B移动。

06



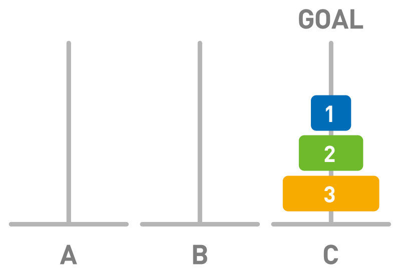
将其他的2个圆盘按照之前移动只有2个圆盘时的操作移动到B。

07



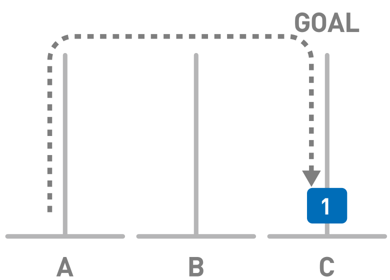
然后把最大的圆盘移动到C。

08



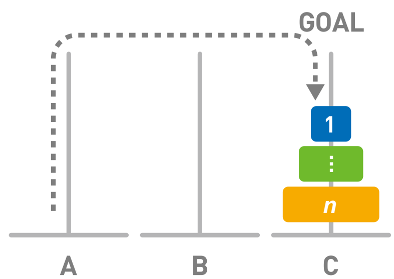
再次按照之前的操作，把B上的2个圆盘移动到C。于是，3个圆盘的移动也完成了。实际上，不管需要移动多少圆盘，这个游戏最终都能达成目标。我们试着用数学归纳法来证明这个结论吧。

09



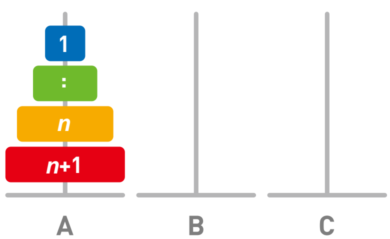
只有1个圆盘时，可以轻松达成目标。

10



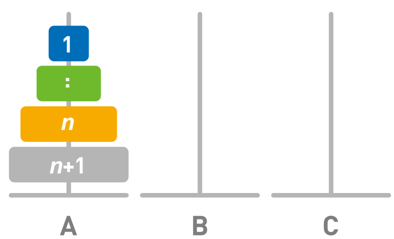
假设圆盘数量为n时同样可以达成目标。

11



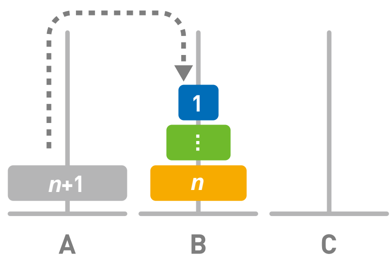
思考一下圆盘数为n+1的情况。

12



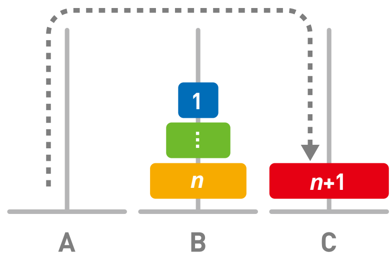
忽略最大的一个圆盘。

13



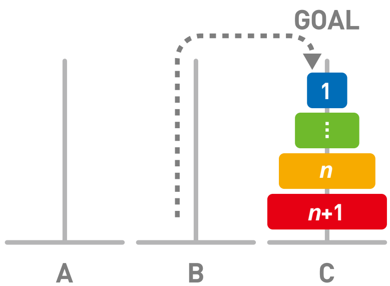
根据之前的假设，圆盘数量为 n 时可以达成目标，所以先把 n 个圆盘移动到B。

14



再把最大的圆盘移动到C。

15



最后再把B上的 n 个圆盘移动到C，游戏结束。

使用数学归纳法，证明了不管圆盘数量为多少，最终都可以达成目标。

解说

大家可能会觉得这太简单了，但是要想移动 n 个圆盘，只需要按照移动 $n-1$ 个圆盘时的方法移动即可。而要移动 $n-1$ 个圆盘，就需要按照移动 $n-2$ 个圆盘时的方法移动。这样追溯下去，最终就会回到移动 1 个圆盘时的操作方法上。

像这样，在算法描述中调用算法自身的方法就叫作“递归”。递归被运用到各种各样的算法中，这些算法统称为“递归算法”。2-6 节中讲到的归并排序和 2-7 节中讲到的快速排序便是递归算法的示例。

▶参考：2-6 归并排序
▶参考：2-7 快速排序

补充说明

我们再来看一看递归算法需要的运行时间。

假设解决有 n 个圆盘的汉诺塔问题时需要的时间为 $T(n)$。只有 1 个圆盘时只需 1 步就能完成，所以 $T(1)=1$。圆盘数为 n 时，把上面的 $n-1$ 个圆盘从 A 移到 B 上需要 $T(n-1)$，再把最大的圆盘移到 C 上需要 1 步，最后把 B 上的 $n-1$ 个圆盘移到 C 上需要 $T(n-1)$，因此 $T(n)=2T(n-1)+1$。

上面的公式即为 $T(n)=2^n-1$，这便是解决这个问题所需要的最短时间。

人人都能看懂的算法入门书

[本书特点]

- 没有枯燥的理论和复杂的代码，易于理解
- 采用大量彩色图片，清晰直观，便于记忆
- 零基础也能轻松掌握，自学算法的好搭档

ISBN 978-7-115-49524-2
9 787115 495242 >

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机科学/算法
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-49524-2
定价：69.00元

看完了

如果您对本书内容有疑问，可访问图灵社区www.ituring.com.cn，联系负责编辑或作译者，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱: ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵新知：图灵教育的科普小组
微信 图灵教育：turingbooks