计算机算法

—设计与分析导论

Sara Baase
Allen Van Gelder
译者：sttony.dark

前言

第一章 分析算法和问题：原理与例子

1.1 概述

说一个问题是算法*可解的*（非正式的），意味着可以编写一个计算机程序，如果我们允许这个程序有足够的运行时间和存储空间，这个程序可以为任何输入处理出一个正确的结果。在 20 世纪 30 年代，计算机出现之前，数学家们就已经积极的定型和研究算法的概念。算法被解释成（非正式的）一套清楚的简单指令，按照这套指令可以解决某个问题和计算某个函数。多种形式的计算模型是设计和研究出来的。早期这个领域工作的重点叫*可计算理论*，其主要是描述或总结那些可算法解决问题的特征，以及指出那些问题是不能算法解决的。由阿兰. 图灵建立的一个重要的否定结论是：证明“停机问题”（halting problem）是不可解决的。停机问题是说：能否判断一个任意给出的算法（或计算机程序）在给定输入的情况下是否最终停止（也就是说是否进入死循环）。这个问题不能用计算机程序解决。

尽管可计算理论对于计算机科学是明显和基础的本质，但是判断一个问题是否在理论上可以在计算机上解决的知识还不足以告诉在实际中是否也可以。例如，可以写出一个完美的国际象棋程序。这并不是一件很困难的事情；棋子在棋盘上的摆放方式是有限的，在一定的规则下棋局总会在有限步之后结束。程序可以考虑计算机可能走的每一步，对手对这一步所有可能的响应，再考虑如何应对对手这一步…… 一直到每一种可能的走法到达结束。既然知道了每一步的最后结果，计算机就可以选择一个最好的。考虑棋子在棋盘上合理的排列（这比步数要少的多），估计就超过 10^{50}。检查所有可能结果的程序需要运行几千年，如此程度的程序不能运行。

实际应用中大量的问题都是可解决的——就是说可以为他们写出程序——但是对一个实用的程序来说时间和存储空间的要求是十分重要的。一个实际程序明确的时间空间要求是很重要的。因此，他们变成了计算机科学中一个分支的主题，叫*计算复杂性*。本书并不包含这一分支，这一分支关心一套正式、有些抽象的关于可计算函数复杂性的理论。（解决一个问题等价于根据一套输入计算出函数的输出。）度量复杂性的公理已经公式化了；他们是如此的基础和一般以致一个程序执行指令的条数和需要存储空间的 bit 数都可以作为 复杂性度量。使用这些公理，我们可以证明任意一个复杂问题的存在，以及问题没有最好的程序。

（Using these axioms, we can prove the existence of arbitrarily complex problems and of problems for which there is no best program.）

本书中学习的计算复杂性分支只关心分析特殊问题和特殊算法。本书打算帮助读者建立一份解决通用问题的经典算法的清单，分析算法、问题的一些一般性的设计技术、工具和指导方针，以及证明正确性的方法。我们将呈现、学习和分析计算机程序中常用的解决各种问题的算法。我们将分析算法执行所需的时间，我们也分析算法执行所需要的空间。在描述各种问题的算法的时候，我们将看到几种经常被证明很有用的算法技术。因此我们暂停一下谈一谈一些一般性的技术，比如分而治之（divide-and-conquer）、贪婪算法（greedy algorithms）、深度优先搜索（depth-first search）和动态编程（dynamic programming）。我们也将学习问题本生的复杂性，就是不管用什么算法解决问题所需固有的时间和空间。我们将学习分类 NP 完全问题——目前还没有找到这类问题的有效算法——并试图用一些试探的方法找到有用的结果。我们也将说明用 DNA 代替电子计算机来向解决这类问题接近。最后，我们将介绍并行计算机算法的主题。

在下面的一节中我们将给出算法描述语言的大概，回顾一些本书用到的背景知识和工具，并展示在分析算法中涉及的主要概念。

1.2 Java 作为算法描述语言

通过平衡多种条件之后，我们选择 Java 作为算法描述语言。算法必须易读。我们想将注意力放在一个算法的策略和技术上，而不是编译器关心的申明和语法细节。语言必须支持数据抽象和问题分解，这样才能够简单清楚的表示一个算法的思想。语言必须提供一种实际的实现（即已经了编译器，原文 The language should provide a practical pathway to implementation.）。他必须在大部分平台上可用而且提供程序开发的支持。实际的实现和运行一个算法能增强学生的理解，不能陷入与编译器、调试器失败的战斗中。最后因为本书是教算法的，不是程序设计，能轻易的将算法转换成读者使用的各种语言必须是合理的要求，特殊语言特性必须减到最少。

在我们要求的许多方面 Java 表现的很好，尽管我们没有宣称他是理想的。他支持自然的数据抽象。Java 是类型安全的，这意味着一种类型的对象不能在需要另一种对象的操作中使用；不允许任意类型之间的转换（叫"cast"）。他有显式的 **boolean** 类型，所以如果程序员混淆了"="（赋值运算符）和"= ="（比较运算符），编译器可以发现这个错误。

Java 不允许指针操作，这些要求通常是含糊不清错误的源头；事实上编译器向程序员隐藏了指针，而在幕后自动处理。在运行期，Java 检查越界的数组下标，检查其他由含糊不清错误引起的矛盾。他执行垃圾收集器，这意味着自动回收不在引用的对象；这大大减轻了程序员管理空间的负担。

Java 的缺点是：他有许多和 C 一样的简洁、含糊的语法特征。对象结构可能导致时间和空间上的低效。许多 Java 构造需要比其他语言大的多的冗余，比如 C。

尽管 Java 有许多特殊的特性，本书展示的算法避免使用他们，只对语言独立的部分感兴趣。事实上，算法中许多步骤都是易读的伪代码。这一节描述了我们在这本书中使用的 Java 的一个小子集，以及用于增加算法可读性的伪代码约定。附录 A 中给出了运行 Java 程序需要的其他的实现细节，但是这些细节对理解主要内容没有关系。

1.2.1 一个可用的 Java 子集

完全熟悉 Java 对于理解本书中的算法并不重要。本节给出了本书出现的 Java 特性的一个大概，主要是为那些想亲自实现算法的读者准备的。这里，我们指出使用的 Java 的 OO 特性，但是尽量避免以使文字得到完全的语言独立性；这主要对熟悉其他 OO 语言（比如 C++）而不是完全熟悉 Java 的读者有好处。附录 A 中有一个简单的 Java 程序。许多书深入讲解 Java 语言。

有丰富 Java 经验的读者肯定会注意到许多例子中都没有使用 Java 的优秀特性。但是，算法后面的*概念*不需要任何特殊的特性，而且我们希望这些概念可以容易的被领悟，并使用各种语言，所以我们将他留给读者，一旦读者领悟了这些概念，就可以用他们喜欢的语言实现。

熟悉 C 语法的读者将会意识到 Java 的语法有许多相似的地方：块由大括号分隔，"{"和" }"；数组的索引在方括号"[" 和"]"中。和 C 和 C++一样，二维数组实际上是一个元素是一维数组的一维数组，所以存取二维数组需要两重 [ ] 就像" matrix[i][j]" 。运算符 "= =" "! =" "< =" 和"> =" 是数学关系运算符的关键字。在文中使用的伪代码中使用的是数学符号。文中使用 "++" 和"--" 运算符来增加和减少，但是决不会将他们嵌入到其他表达式中。这里也有从 C 借用的运算符 "+=" "- =" "* =" "/ =" 。例如

```
p+=q ; /*p=p+q */
y-=x ; // y=y-x
```

就像在例子中一样，注释从 "// " 到行结尾，或是从 " /* " 到" */ "，和 C++一样。

Java 的函数头也和 C 一样。函数头在函数名字后的括号中指定了*参数的类型特征*；在函数名字之前指定了返回类型。返回类型和参数类型的组合叫函数的完全类型特征也叫原型。因此

```
int getMin(PriorityQ pq)
```

告诉我们 getMin 接收一个类型（或类）为 PriorityQ 的参数，返回类型 int。

Java 只有很少的原始类型，其他所有类型都叫 classes。原始类型是逻辑（**boolean**）和数字类型（**byte**、**char**、**short**、**int**、**long**、**float** 和 **double**）类型。所有的 Java 类（非原始类型）都是引用类。在底层，在类中申明的变量都是“指针”；他们的值都是地址。类的实例叫*对象*。申明一个变量不会创建对象。通常用"**new**"运算符来创建对象，"new"返回新对象的一个引用。

对象的数据域按 OO 术语叫*实例域*。二元的点运算符用来存取对象的实例域。

例 1.1　创建和存取一个 Java 对象

在这个例子，让我们假设数据信息遵循下面的嵌套逻辑结构：

- year
  - number
  - isLeap
- month
- day

这里使用非正式的术语，year 是由布尔属性 isLeap 和整数属性 number 构成的组合属性，而 month 和 day 只是简单的整数属性。为了反映这个嵌套结构，我们必须在 Java 中定义两个类，一个表示整个数据，另一个表示 year 域。假设我们为这两个类分别取名 Date 和 Year。接下来我们要申明 number 和 isLeap 作为 Year 类的实例域，申明 year、month、day 为 Date 类的实例域。除此之外，我们最好将 Year 定义为 Date 的内部类。语法如图 1.1。

```
class Date
{
  public   Year year;
  public    int      month;
  public    int       day;
  public    static class Year
                {
                    public   int   number;
                    public   boolean   isLeap;
                }
}
```

图 1.1　带一个内部类 Year 的类 Date 的 Java 语法

不带 **public** 关键字，实例域就不能在 Date 和 Year 外面访问；为了简单，我们在这里使用 **public**。申明内部类，Year 带 static 修饰符的原因是，因为我们可以单独创建 Year 的实例而不与任何特定 Date 对象相关联。本书中所有的内部类都将是 static。

假定我们创建了一个由 dueDate 变量引用的 Date 对象。为了存取对象中的 year 实例域，使用.运算符，如"dueDate . year"。如果一个实例域在类中（与之对应的是在原始类型中），要再接一个. 来存取他的实例域，如"dueDate .year. isLeap" 。

赋值语句仅仅拷贝类对象的引用或地址；不拷贝实例域。例如，" noticDate = dueDate" 导致变量 noticDate 指向了变量 dueDate 指向的同一对象。因此下面的代码很可能是逻辑错误：

```
noticDate = dueDate;
noticDate .day = dueData .day -7 ;
```

参见 1.2.2 节，对这个问题有另外的讨论。

Java 中的控制语句 **if** , **else** , **while** , **for** 和 break 和他们在 C（和 C++）中的意义相同，本书中将用到。存在许多其他的控制语句，但不用他们。while 和 for 的语法是

while （继续条件）body

for （初始化；继续条件；增量）body

这里“初始化”和“增量”都是简单语句（不带{ }），“body”是任意语句，“继续条件”是 **boolean** 表达式。**break** 语句导致立即从最近的 **for** 或 **while** 循环中跳出（当然也包括 **switch**，但本

书不使用 **switch**）。

Java 是单根继承，其根是 **Object**。当申明一个新类时，他可以从原来的类中 **extends**，新类就成了继承树种以前定义类的派生类。文中我们将不建立这样的体系，为了尽可能的保持语言独立性；但是在附录 A 中给出了一些例子。如果新类没有申明为从任何类 extend，默认的他从 **Object** extend。学习算法不需要复杂的类体系。

在 OO 术语里对象的操作叫*方法*；但是我们将限制自己只使用*静态方法*，他们都是简单的过程和函数。在我们的术语中，*过程*是一个可以取名字、可以被调用的计算步骤序列（带参数的）；*函数*则是一个可以向调用者返回值的过程。在 Java 中不返回值的过程申明返回类型为 **void**；这一点和 C 和 C++一样。Java 术语 *static* 意味着这个方法可以被任何对象和合适类型的对象（对象类型是它的类）所调用，只要依照方法的类型特征就行（经常被称作原型）。一个静态方法与任何特定对象都无关。静态方法的行为就像其他程序设计语言中的函数和过程一样。但是，他们的名字必须跟在类名字的后面，例如"List . first(x) " 指以参数 x 调用在类 List 中定义的 first 方法。

默认情况下 Java 中的实例域是私有的，这意味着他们仅能被定义在类中的方法（函数和过程）所存取。这和抽象数据类型（ADT）的设计主题是一致的，即对象只能由定义在 ADT 中的方法所存取。实现这些 ADT 操作（或是静态方法，或是函数、过程）的代码存在与类中，知道这些私有的实例域和其类型。 默认情况下方法也是私有的，但一般指定为 "public" ，所以定义在其他类中的方法可以调用他们。但是，一些只能由类中其他方法调用的底层方法可能也是私有的。

ADT 的*客户*（client）（调用 ADT 的函数、过程）在 ADT“生存”以外的类中实现，所以他们只能存取 ADT 类的 public 部分。私有数据的维护叫做*封装*或*信息隐藏*。

实例域在对象的生存期之内保存赋给他的值，直到后来又赋给他别的值。这里我们可以看到将实例域指定为私有的优点。一个共有的实例域可以被整个程序的任意一段代码赋成任意一个值。一个私有的实例域就只能通过 ADT 类中为此目的而设计的方法所赋值。这个方法可以进行其他计算，并测试赋给他的值是否和 ADT 要求的一致，是否和其他实例域一致。

一个新对象通过语句"**new** classname( )"创建，例如：

```
Date dueDate = new Date
```

这条语句导致 Java 调用 Date 类的默认*构造函数*。构造函数保留一个新对象所占的存储空间，返回一个新对象的引用（很可能是地址）用于存取对象。新对象的实例域没有初始化。

> *Java 语言提示*：程序员可以为类定义一个额外的构造函数，构造函数可以初始化各种实例域，执行其他计算。我们感兴趣的是语言独立性，文中将不使用这样的构造函数，所以忽略其细节。

Java 中的数组申明和 C/C++中不太一样，他们的特征也明显不同。Java 语法申明一个整型数组（非常明确，申明了一个类型为“整型数组”的变量）是 " **int** [ ] x "，而 C 使用 " **int** x [ ]"。这条语句不初始化 x ，下面这条语句才初始化

```
x = new int [howMany] ;
```

这里 howMany 可以是一个常量也可以是一个变量，他的值指示了数组的长度。申明一个类的数组是一样的。申明和初始化可以，通常都是合到一条语句中：

```
int [ ] x = new int [howMany];
Date [ ] dates = new Date [howMany ];
```

当这些语句初始化 x 和 dates ，保留数组的存储空间时，只能以默认值初始化元素，这未必有用。因为在单个元素使用之前可能需要单独的值（很可能使用 **new** 运算符）。在 Date 类外初始化的语法

```
dates[0] = new    Date( );
dates[0]. month    = 1 ;
dates[0]. day =1 ;
dates[0]. year = new    Data.Year( );
dates[0]. year. number =2000;
dates[0]. year. isLeap =true ，
```

注意，域名字跟在指定数组元素的索引之后。在第二条 **new** 语句中，内部类名字，Year，被外部类 Date 所限定，因为这条语句在类 Date 的外面。就像前面提到的，Java 程序员可以写一个带参数的构造函数以初始化一个构造好的新对象，但是本文为了语言独立性不使用这样的构造函数。

一旦用 **new** 语句初始化了 x 数组，x 引用的数组长度就不能改变了。Java 提供查询长度的方法，x . length。*实例域* length 是 **new** 语句自动附加到数组对象上的，可以被 x 所存取。

有效的元素索引是从 0 到（x.length －1）。如果程序试图存取一个越界的元素，Java 将停止程序（抛出异常）。我们经常希望索引从 1 到 n，因此常常初始化数组为 " **new int** [n+1] " 。

Java 允许*重载* 和*覆盖*方法。一个重载方法是说，有多个带不同参数的定义，但是有相同的返回值。许多算术操作是*重载*的。覆盖则是在类体系中多次定义了有同样参数类型的同一方法，Java 采用在类体系中最接近的定义。（还是为了和其他语言兼容，而且这个特性对于理解算法也不是决定性的，我们避免使用这个特性，读者可以去阅读 Java 语言的书。）在不同类中可能使用相同名字的方法，但是这不是覆盖，因为在类外面方法时，必须使用类名（对象名）来限制。在后面的例子将看的更清楚 。

对于熟悉 C++的读者，必须指出 Java 不允许程序员重载*运算符*。本文在伪代码中使用这样的运算符来增加可读性（例如，x < y 这里 x 和 y 都是非数值类，比如 **String**）。但是，如果你定义一个类，开发一个实际的 Java 程序，你就必须写一个有名字的函数（比如 less（）），调用他来比较你的类。

1.2.2 组织者类

我们创造术语组织者类，这不是标准的 Java 术语，他指一种只是为了将多个实例域组织到一起的简单的类。组织者类实现类似 C 的结构、Pascal 或 Modula 记录的规则；类似的东西存在于 Lisp、ML 和其他大部分程序设计语言中。组织者类和 ADT 的目的正好相反；他们只是简单的组织数据，不限制数据的存取，也不为数据提供任何定制的方法。习惯于在其他类中定义组织者类；由于这个原因，在 Java 术语中组织者类叫内部类。

一个组织者类仅有一个方法，叫 copy。既然 Java 中的变量都是对象的引用，赋值语句仅拷贝引用，而不是对象的实例域，就像我们在例 1.1 中看到的。如果这些变量在一个叫 Date 的组织者类中定义，我们可以使用语句

```
noticDate = Date. copy( dueDate );
noticDate .day = dueDate .day -7 ;
```

来拷贝 dueDate 对象的实例域到 noticDate 所引用的新对象中，之后修改的只是 noticDate 的 day 域。

定义 1.1 组织者类的拷贝函数

组织者类将实例域赋值到一个新对象的 copy 函数（方法）的一般规则如下（例子中假设对象 d 被拷贝到新对象 d2）：

1. 如果实例域（year）是其他的组织者类，copy 方法将调用该类的 copy 方法，如 d2.year = Year. copy(d . year)。

2. 如果实例域（day）不是其他组织者类，使用一般的赋值，如同 d2 .day = d.day。

完整的例子在图 1.2 中给出。

```
class  Date
{
    public  Year  year;
    public  int  month;
    public  int  day;
    public  static  class  Year
    {
        public  int number;
        public  boolean  isLeap;
        public  static  Year copy(Year y)
            {
                Year y2 =new  Year( );
                y2. number = y. number;
                y2. isLeap = y. isLeap;
                return  y2;
            }
    }
    public  static Date  copy(Date d)
    {
        Date d2 =new Date( );
        d2. year = Year. copy ( d . year);
        d2. month = d.month;
        d2. day = d. day;
        return d2;
    }
    public static int defaultCentury;
}
```

图 1.2 带内部组织者类 Year 的组织者类 Date

程序员必须保证在定义组织者类时不能发生循环引用，否则 copy 函数将陷入递归不能结束。当然，组织者类中新对象可以由一般方法创建：

```
Date  someDate = new  Date ( )
```

Java 语言提示：Java 提供一种一层对象的机制，不需要写出每一条赋值语句，clone 方法，但是他不能自动处理像 Date 这种嵌套结构；你仍然要写出处理这种情况的代码。附录 A 给出了一个 copy1level 函数的一般性代码。

一个组织者类包含唯一一个 **public** 实例域。如果 **static** 关键字也出现在域的申明中，该域不与任何实际对象相关，关键在于他不是一个全局变量了。

例 1.2　典型组织者类

在图 1.2 中，为例 1.1 的类增加了 copy 函数，所以他们有成为组织者类的资格。就像我们看到的，copy 的定义是机械的，而且单调的。在后面的例子中将省略实现的细节。为了更加完整，我们包含了全局变量 defaultCentury，而一般情况组织者类是不包含全局变量的。

总结，我们创造了术语组织者类，他指那些只是简单将实例域组织起来，为他们定义一个拷贝函数的类

1.2.3　基于 Java 的伪代码约定

本书中的大多数算法使用了基于 Java 的更易读的伪代码，而不是严格的 Java。使用了下面的约定（除了在附录 A 中的以外）。

1. 省略了块分隔符（"{" 和"}"）。块边界用缩进指出。
2. 方法（函数或过程）申明中省略了 **static** 关键字。本文中申明的所有方法都是 **static**（偶尔会有 Java 内建的非静态方法；特别是 s. length( ) 用来获得字符串的长度。）关键字 static 会出现在需要实例域和内部类的地方。
3. 在调用方法（函数或过程）前面省略了类名字。例如 x = cons(z, x) ，用完整的 Java 语法需要写成 x= IntList. cons (z, x) （IntList 类在 2.3.2 节中描述）。无论什么时候静态方法在其定义的类之外调用都必须加上类名字前缀。
4. 省略存取控制的关键字 **public** 、 **private** 和 **protecte**。将所有与一个 Java 程序相关的文件都放在一个目录下，省去了处理可见的麻烦。
5. 经常使用使用数学关系运算符 ≤,=,≥ 来代替他们的关键字版本。关系运算符用在意义明确的地方，比如 **String** ，即使在 Java 中是非法的。
6. Java 保留的和 Java 标准部分的关键字都以黑体： **int** 、**String** 。*注释用斜体*。代码语句和程序变量字体不变。伪代码语句的字体也不变。

当我们特别指出这是 Java 语言时会偶尔离开这些约定。

1.3　数学知识

本书中我们使用各种数学概念、工具和技术。大多数你应该熟悉了，可能会有少部分是新的。本节一一列举他们，为你提供一个参考，也是一个回顾。设计较深的证明在第三章。

1.3.1 集合、元组和关系

本节提供一些非正式的定义和集合、关系概念的基础特性。集合是“一堆”不同元素，我们相将他们作为一个对象处理。通常，元素有同样的类型，有利于我们将他们看作一个对象的公有的特性。符号 $e \in S$ 读作“元素 e 是集合 S 的一个成员”，或“e 属于 S”注意这里 e 和 S 是不同的类别。例如，如果 e 是一个整数，S 是一个与整数完全不同的整数集合。

一个实际的集合有两种定义方式，列举法和描述法，两种方式都在一对大括号中。例如下面的符号

$$S_1 = \{a,b,c\},\quad S_2 = \{x \mid x\text{是2的整数次幂}\},\quad S_3 = \{1,....,n\}.$$

表达式 S_2 读作“集合中所有元素 x 都是 2 的整数次幂”。（译者：这里罗嗦的说了一大通怎么表示集合。大家对集合的表示应该比较了解吧！什么你不知道集合？去看看高中的数学！）。

如果一个集合 S_1 所有的元素都是另一个集合 S_2 的元素，就说集合 S_1 包含于集合 S_2，集合 S_2 包含集合 S_1。符号是 $S_1 \subseteq S_2$ 和 $S_2 \supseteq S_1$。为了表示 S_1 是 S_2 的子集而不等于 S_2，写为 $S_1 \subset S_2$ 和 $S_2 \supset S_1$。区别 $\in$ 和 $\subset$ 是很重要的。前者表示是一个元素，后者表示集合之间的包含。空集记做 Φ，表示没有一个元素，所以他是所有集合的子集。

集合没有固定顺序。因此，在前面的例子中，S_1 也可以定义成 $\{b,c,a\}$，S_3 也可以定义为 $\{i \mid 1 \le i \le n\}$，可以理解为 i 是一个整数。

以*特定顺序*组合的一组元素称为*序列*。除了顺序，集合和序列的另一个很重要的不同点是，序列的元素可以重复。序列被表示成元素按顺序的列表，在加上一个圆括号。因而(a,b,c)，（b,c,a）和和（a,b,c,a）是不同的序列。序列中也可以使用省略号，比如（1,.....n）。

如果存在一个整数 n，集合 S 元素的数量可以和 $\{1,....,n\}$ 一一对应，则我们称该集合为有限集；这种情况下写为 $|S|=n$。通常 $|S|$ 表示集合 S 元素的数量，也叫*集合的基*。如果存在一个整数 n，序列 S 元素的数量可以和 $\{1,....,n\}$ 一一对应，则我们称该序列是有限序列。集合和序列不是有限就是无限的。如果有限序列中所有的元素都不同，则称该序列是由同样元素构成有限集合的一个排列。这再一次强调了集合和序列的不同。一个有 n 的元素的集合有 n！个不同的排列。

一个有 n 个元素的有限集有多少不同的子集呢？记住空集和集合本身。我们有 2^n 个子集。

其中基为 k 的子集有多少呢？有一个专门的符号来表示这个量：$\binom{n}{k}$，读作“n 中选 k”，或是或者更罗嗦“n 个中一次选 k 组合的数量”。也使用符号 C(n,k)，这个量称为*二项式系数*。

计算 $\binom{n}{k}$ 或 C(n,k)的表达式（译者：这里对表达式的推倒说了半天，排列组合的知识大家都应该学过了，直接给出结论。主要是为了偷懒）

$$C(n,k) \equiv \binom{n}{k} = \frac{n(n-1)...(n-k+1)}{k!} = \frac{n!}{(n-k)!k!} \text{其中} n \ge k \ge 0 \qquad (1.1)$$

从 0 到 n 每一个数对应子集的数量都知道，我们得到

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \qquad (1.2)$$

元组和叉积

*元组*是一个有不同类别元素的有限序列。例如在一个二维平面上的点可以表示为一个有序对（x,y）。如果是几何平面，x 和 y 都是长度。如果是一个问题大小和运行时间的曲线图，则 y 可能是秒，x 可能是一个整数。短元组有特定的名字对、三元组、四元组等等（英文中有 pair、triple、quadruple 等词）。在元组的上下文里，这些词（指上面那些词，译成汉语有意义十分明确了）是有序的，在其他上下文中“对”可能表示“两个元素的集合”而不是“两个元素的序列”。一个 k-元组是有 k 个元素的元组。

两个集合 S 和 T 的叉积，是一个对的集合，该集合中对的第一个元素一定来自 S，第二个元素一定来自 T。我们将叉积写成：

$$S \times T = \{(x, y) \mid x \in S, y \in T\} \tag{1.3}$$

因此| S×T| = | S | | T |。两个集合 S 和 T 通常是一样的，但这不一样也可以。我们可以定义叉积迭代，就可以产生更长的元组。例如，S×T×U，将得到一个由三元组组成的集合，三元组的元素分别来自 S、T 和 U。

关系和函数

关系是叉积的一个简单子集。这个子集可以是有些也可以是无限的，可以是空集也可以是叉积本身。最重要的关系是*二元关系*，他是简单叉积的子集。许多二元关系的例子都是我们非常熟悉的，例如实数的“小于”关系。以 R 表示实数，“小于”关系可以定义成 $\{(x, y) \mid x \in R, y \in R, x < y\}$。就像我们看到的，他是 R×R 的子集。另一个例子，P 表示所有的人，P×P 表示所有的一对人。我们可以定义“父子关系”（x,y），或是“爷孙关系”，他们都是 P×P 的一个子集。

尽管许多二元关系的两个元素都有相同的类型，但这不是必须的。$\{(x, y) \mid x \in S, y \in T\}$ 也是二元关系。回顾我们早先的图表的例子，就是一个程序规模和程序运行时间的关系。另一个例子，我们将 F 定义为所有的女人，“母子关系”就是 F×P 的子集。

尽管关系可以是任意子集，但是两个元素都来自一个集合的关系 R，有许多令我们感兴趣的特性。因为标准的关系是一个中间符号（比如 x<y），符号 xRy 常用于表示 $(x, y) \in R$。

定义 1.2 关系的重要属性

令 $R \subseteq S \times S$，下面是一些特性以及特性的条件。

自反性　　对于所有 $x \in S$，有 $(x, x) \in R$。

对称性　　$(x, y) \in R$，则 $(y, x) \in R$。

反对称性　　$(x, y) \in R$，则 $(y, x) \notin R$。

传递性 $(x,y)\in R$ 且 $(y,z)\in R$，则 $(x,z)\in R$。

如果一个关系是*自反*、*对称* 和*可传递* 的，则称关系为全等关系，记做≡。

注意，“小于”关系是传递和反对称的，而“小于或等于”是传递和自反的，但不反对称（因为 x ≤ x）。

“等价”关系在很多问题里面都是重要的，因为这样的关系*划分*底层集合 S；也就是说，他将集合 S 划分成一组不相交子集（也叫等*价类*）S1,S2.....，S1 中所有的元素都都等价于集合中的其他元素，S2 中所有的元素也都等价于其他元素等等。例如，如果 S 是非负整数的集合，R 定义为 $\{(x,y)\mid x\in S,y\in S,(x-y)\text{能被3整除}\}$，则 R 等价于 S。因为（x-x）可以被 3 整除，如果（x-y）可以被 3 整除，（y-x）也可以，最后（x-y）和（y-x）可以被 3 整除的话，（x-z）也可以。所以 R 满足等价关系的条件。R 如何划分 S 呢？可以分为 3 组，按除以 3 的余数分。所有除以 3 余数相等地元素都等价。

既然二元关系是二元组的集合，经常将他写成一张两列的表，每一行是一个二元组。函数是一个关系，这个关系第一列中的元素不会在关系中出现。

许多涉及二元关系的问题都可以转换成一个图上的问题。图问题构成一大类很有挑战性的算法问题。例如，在一个有很多相互关联的小任务的大工程里面，我们有很多形如“任务x必须在任务 y 完成以后才可以开始”的条件。有固定的人来完成任务，如何安排来在最短时间内完成？在后面的章节中我们将遇到很多这样的问题。

1.3.2 代数和微积分工具

本节提供关于对数、概率、排列、求和公式、数列级数（在这里，级数指数列的和）的定义和一些基本特性。我们将为第三章中的递推方程介绍数学工具。你可以在本章后面的注意和参考找到这里没有的公式。

Floor 和 Ceiling 函数

对于实数 x，$\lfloor x\rfloor$（x 的 floor）是大于或等于 x 的最大整数。$\lceil x\rceil$（x 的 ceiling）是小于或等于 x 的最大整数。例如 $\lfloor 2.9\rfloor=2$，$\lceil 6.1\rceil=7$。

对数

对数函数，通常以 2 为底，是本书中经常用到的数学工具。尽管在自然科学中对数不总是使用，他们在计算机科学中很流行。

定义 1.3 对数函数和对数底

对于 b>1，x>0，$\log_b x$（读作以 b 为底 x 底对数）是一个实数 L，满足 $b^L = x$。

下列对数的属性可以从对数的定义简单的得到。

引理 1.1 令 x 和 y 是任一非负实数，令 a 是任一实数，令 b>1，c>1 的实数。

1. $\log_b$ 是单调递增函数，就是说，如果 x>y，则 $\log_b x > \log_b y$。
2. $\log_b$ 是一个一一对应函数，即如果 $\log_b x = \log_b y$，则 x=y。
3. $\log_b 1$=0
4. $\log_b b^a$=a
5. $\log_b(xy) = \log_b x + \log_b y$
6. $\log_b(x^a) = a\log_b x$
7. $x^{\log_b y} = y^{\log_b x}$
8. $\log_b x = (\log_b x)/(\log_b c)$

既然以 2 为底的对数在计算复杂性中用的最多，特别为他指定一个记号“lg”$\lg x = \log_2 x$。自然对数（以 e 为底）表示为“In"；就是说，Inx= $\log_e x$。当 log(x)步标注底时，意味着对于底为任何数的情况都适用。

有时对数函数自己进行复合。符号 lglg(x)表示着 lg(lg(x))。符号 $\lg^{(p)}(x)$ 表示复合 p 次，所以 $\lg^{(2)}(x)$ 等同于 lglg(x)。主义 $\lg^{(3)}(65536) = 2$，而不是 $(\lg(65536))^3 = 4096$。

尽管在上文中我们总是对一个整数求对数，没有用一个负数，我们经常用一个整数值作为对数的底。令 n 是一个负数。如果 n 是 2 的幂，让 $n = 2^k$，对于有些整数 k，则有 $\lg n = k$。如果 n 不是 2 的幂，则有一个整数 k 使得 $2^k < n < 2^{k+1}$。在这种情况下，$\lfloor \lg n \rfloor = k$ 且 $\lceil \lg n \rceil = k+1$。表达式 $\lfloor \lg n \rfloor$ 和 $\lceil \lg n \rceil$ 将经常使用。你必须验证下面的不等式：

$$n \le 2^{\lceil \lg n \rceil} \le 2n$$

$$\frac{n}{2} < 2^{\lfloor \lg n \rfloor} \le n$$

最后，这里有一些有用的结论：$\lg e \approx 1.443$，$\lg 10 \approx 3.32$。In(x)的导数是 1/x。使用引理的第八条，lg(x)的导数是 lg(e)/x。

排列

n 个不同对象的一个排列是包含每一个对象的序列。令 $S = \{s_1, s_2, ...s_n\}$。注意 S 的元素以他们的索引排序；就是说 s_1 是第一个元素，s_2 是第二个元素等等。S 的一个排列是从集合{1,2….,3}到自己的一个一一对应函数 π。我们将 π 看作通过将第 i 个元素 s_i 移动到 π(i)的位置，重新排列 S 得到。我们可以通过列出他的值来简单的描述 π，$(\pi(1), \pi(2)....\pi(i))$。例如，对于 n=5，π＝（4，3，1，5，2）是以 s_3, s_5, s_2, s_1, s_4 排列 S 得到的。

N 个不同对象的排列总数是 n!。第一个元素可以移动到 n 个位置中的任意一个，第二个有 n-1 种选择…，总数是 n×（n-1）×（n-2）× …×2×1＝n!。

概率

假设在给定的状况下，，一个事件，或试验有一个或 k 个结果，$s_1, s_2, s_3, ...s_k$。这些结果叫*基本事件*。所有基本事件的集合叫*事件空间*（*universe*）以 U 标记。对每一个 s_i 我们关联一个实数 $\Pr(s_i)$，叫 s_i 的概率，有如下性质：

$$0 \le \Pr(s_i) \le 1 \text{ 其中} 1 \le i \le k;$$

$$\Pr(s_1) + \Pr(s_2) + ... + \Pr(s_k) = 1$$

很自然的，令 $\Pr(s_i)$ 为 s_i 出现的次数和总试验次数的比值。(Note, However, that the definition does not require that the probabilities correspond to anything in the real world. 译者：“概率对应现实世界的任何事 anything” ？！）事件 $s_1, s_2, s_3, ...s_k$ 被称为相互*独立事件*，因为至多只有一个事件会发生。

经常使用的展示概率含义的例子有，投硬币、投骰子以及在玩扑克牌时的各种事件。事实上最开始概率理论就是从法国数学家 Blaise Pascal 对赌博游戏的研究开始的。如果“试验”

是投掷一个硬币，则硬币可能出现“正面”朝上和“背面”朝上。我们令 s_1,=“正面"，s_2=“背面”并假定 $\Pr(s_1)=1/2$， $\Pr(s_2)=1/2$。（因为硬币不可能以边立在地上，我们可以令 s_3=“边"， $\Pr(s_3)=0$。但是即使有无穷次试验，发生概率为 0 的事件也不产生影响，所以基本上不定义这样的基本事件。）如果投掷六面骰子，则有六种可能的结果：$1\le i\le 6$， s_i=“骰子以数字 i 朝上"， $\Pr(s_i)=1/6$。一般情况下，如果有 k 种可能的结果，每个结果发生的概率都相等，则对于每一个 i 令 $\Pr(s_i)=1/k$。通常没有理由认为所有结果等概率；主要是在例子种做这样的假设，或没有数据做出更好的判断之前做等概率假设。

如果试验包含许多对象，则基本事件必须对所有观测到的事件计数。例如，如果有两个骰子，A 和 B 一起投掷，则事件“A 以数字 1 朝上”就不是基本事件，因为 B 可以有多个结果。这种情况下，基本事件必须是 s_{ij}=“A 以数字 i 朝上，且 B 以数字 j 朝上”$1\le i,j\le 6$，我们简写为“A 出现 i，且 B 出现 j"。此时有 36 个基本事件，每个基本事件发生的概率是 1/36。

我们经常需要考虑几个特定结果发生的概率，或是有特定属性的结果发生的概率。令 S 是基本事件 $\{s_1,s_2...s_k\}$ 的一个子集，则 S 叫事件，且 $\Pr(S)=\sum_{s_i\in S}\Pr(s_i)$。例如，假设投一个骰子，定义事件 S 是“出现可以被 3 整除的数字”。则 S 的概率是 $\Pr(S)=\Pr(\{s_3,s_6\})=\Pr(s_3)+\Pr(s_6)=1/3$，基本事件也是事件。

两个特殊的事件是*必然事件*，$U=\{s_1,s_2...s_k\}$，他的概率是 1，和*不可能事件*Φ，他的概率是 0。（Φ也表示空集。）对于所有的事件 S，有一个补事件“非 S"，由所有不再 S 中的事件组成，即 U-S。显然 Pr(not S)=1-Pr(S)。

事件可以通过同组的其他使用逻辑连词“与”和“或”来定义。事件“ S_1和 S_2 ”是 $(S_1\bigcap S_2)$， S_1和 S_2的交集。事件“ S_1或 S_2 ”是 $(S_1\bigcup S_2)$，是 S_1和 S_2的并集。

我们经常需要分析分析一些带权的概率，权值是基于试验的知识确定的。这叫*条件概率*。

定义 1.4 条件概率

给出事件 T 的情况下事件 S 条件概率定义为

$$\Pr(S\mid T)=\frac{\Pr(SandT)}{\Pr(T)}=\frac{\sum_{s_i\in S\cap T}\Pr(s_i)}{\sum_{s_j\in T}\Pr(s_j)} \qquad (1.4)$$

这里 s_i和 s_j包括基本事件。

例 1.3 两个骰子的条件概率

假设在试验中投掷两个骰子，A 和 B。我们定义 3 个事件：

S_1：“A 是 1”

S_2：“B 是 6”

S_3：“A 和 B 数字的和小于等于 4”

为了得到条件概率的概念，让我们考虑所有基本事件都等概率的简单情况。对于我们的例子，36 个基本事件是“A 是 i，且 B 是 j”，$1 \le i, j \le 6$。则条件概率 $\Pr(S_1 \mid S_3)$ 可以解释成这样一个问题，“S_3 中所有的基本事件，也在 S_1 中的比例？”

让我们列出所有在 S_3 中基本事件：

“A 是 1，且 B 是 1”，“A 是 2，且 B 是 1”，

“A 是 1，且 B 是 2”，“A 是 2，且 B 是 2”，

“A 是 1，且 B 是 3”，“A 是 3，且 B 是 1”。

事件 S_1 由 6 个基本事件组成，即 A 是 1，而 B 是所有可能的 6 个数字。三个 S_3 中的基本事件也在 S_1 中，所以问题的答案是 3/6，1/2。根据等式（1.4）可以计算出给出 S_3 时 S_1 发生的概率

$$\Pr(S_1 \mid S_3) = \frac{3/36}{6/36} = 1/2 \text{。}$$

注意，给出 S_3 时 S_2 发生的概率是 0，即 $\Pr(S_2 \mid S_3) = 0$。

一般情况，计算条件概率的过程是消去指定事件之外的所有事件，之后通过通用的因子来调整剩下的事件，使得调整之后概率和为 1。（原文：In general, the procedure for calculating conditional probabilities given some specified event S is to eliminate all the elementary events by the same factor so that the rescale the probabilities sum to 1.）要求的因子是 1/Pr(S)。

事件的条件概率可能比事件的非条件概率要大或是小。在例 1.3 中 S_1 的非条件概率是 1/6，在给出 S_3 的情况下条件概率是 1/2。另一方面，“A 的数字能被 3 整除”的非条件概率

是 1/3。但是在例 1.3 中，我们看到给出 S_3 的情况下“A 的数字能被 3 整除”的条件概率是 1/6。

定义 1.5 相互独立

给出两个事件 S 和 T，如果

$$\Pr(S and T) = \Pr(S)\Pr(T)$$

则 S 和 T 是*相互独立事件*，或者简称*相互独立*。

如果 S 和 T 是相互独立事件，则 $\Pr(S \mid T) = \Pr(S)$（参见练习 1.8）。就是说，事件 T 的发生不以任何方式影响事件 S 发生的概率。当独立属性存在时他非常有用，因为他使得可以分开分析两个不同事件的概率。但是，当错误的假设了独立属性时，将产生许多错误的分析结果。

例 1.4 相互独立事件

继续我们在例 1.3 中的事件定义，事件 S_1 和 S_2 是相互独立的，因为每一个的概率都是 1/6，而且（S_1 and S_2）组成一个基本事件，他的概率是 1/36。注意，$\Pr(S_1 \mid S_2) = (1/36)(6/36) = 1/6 = \Pr(S_1)$。

从例 1.3 的讨论，我们可以看出 S_1 和 S_3 不是相互独立事件，S_2 和 S_3 也不是相互独立的。

随机变量和他们的期望值在涉及概率的许多情况下都是很重要的。一个随机变量是一个与事件是否发生相关的实数函数；就是说，他是一个为事件定义的函数。例如，如果一个算法执行操作的数目倚赖于输入，每一个可能的输入都是一个事件，而操作的数目是随机变量。

定义 1.6 期望和条件期望

令 f(e)是一个随机变量定义在一套基本事件 $e \in U$ 上。f 的期望，标记为 E(f)，定义为

$$E(f) = \sum_{e \in U} f(e)\Pr(e)$$

这常被称为 f 的平均值。给出事件 S 后 f 的条件期望标记为 E(f | S)，定义为

$$E(f \mid S) = \sum_{e \in U} f(e)\Pr(e \mid S) = \sum_{e \in S} f(e)\Pr(e \mid S)$$

因而任何不再 S 中的事件的条件概率是 0。

期望常常比随机变量本身容易处理，特别是当涉及几个相互联系的随机变量时。得出下面的一条重要规则，他可以从定义上简单的得到证明。

引理 1.2 （期望法则）对于定义在一套事件 $e \in U$ 上的随机变量 f(e) 和 g(e)，以及任意事件 S：

$$E(f+g) = E(f) + E(g)$$

$$E(f) = \Pr(S)E(f \mid S) + \Pr(notS)E(f \mid notS)$$

例 1.5 条件概率和顺序

在第四章我们将把通过比较得到的顺序信息和概率连起来考虑。让我们看一个这种类型的例子，这个例子涉及 4 个元素 A、B、C、D，都是互不相同的数值，但是开始我们不知道他们值和关系的任何信息。我们将以字母的顺序来标记基本事件，基本事件是他们的关系顺序；就是说 CBDA 表示 C<B<D<A。有 24 种可能的结果：

ABCD ACBD CABD ACDB CADB CDAB

ABDC ADBC DABC ADCB DACB DCAB

BACD BCAD CBAD BCDA CBDA CDBA

BACD BDAC DBAC BDCA DBCA DCBA

我们从假设所有输入结果都是等概率开始，则每个事件的概率都是 1/24。A<B 的概率是多少？换句话说，将 A<B 定义为事件，其概率是多少。凭直觉，我们期望他是 1/2，可以通过统计 A 出现在 B 之前的结果数量来验证。同样的，对于每一对元素，其概率至少是 1/2，例如，事件 B<D 的概率是 1/2。

现在假设程序*比较* A 和 B，发现 A<B。这对概率的影响如何？为了使这个问题更严格，我们将他表述成“在事件 A<B 下的条件概率是多少？”。通过检查我们看到组成 A<B 的基本事件都在表的前两行。因此在给出条件 A<B 的情况下，基本事件的条件概率是原来的两倍，2/24=1/12，但在给出 A<B 的情况下，后两行的基本事件的条件概率是 0。

回到先前的比较，事件 B<D 的概率是 1/2。我们没有比较 B 和 D。给出 A<B 的情况下 B<D 的条件概率是否还是 1/2？为了回答这个问题，我们检查在前两行中 B 在 D 前面的结果有几个。事实上，只有四个。所以 Pr(B<D|A<B)=1/3。

现在考虑事件 C<D。他的条件概率是不是也不是 1/2？再一次检查表的前两行，我们检查到 C 出现在 D 之前共有 6 次，所以 Pr(C<D|A<B)=1/2。因此事件 A<B 和 C<D 是相互独立的。这正是我们期望的：A 和 B 的关联顺序“不对”C 和 D 的顺序产生任何影响。

最后，假设程序作了另一个比较，发现 D<C（已经发现了 A<B）。让我们检查在同时给出这两个事件情况下的条件概率（就是说给出单独一个事件“A<B 且 D<C”）。通过检查我们看到事件“A<B 且 D<C”由表第二行的元素组成。为了使得条件概率的和为 1，所有基本事件都必须有 1/6 的条件概率。程序不比较 A 或 B 也不比较 C 或 D。这是否意味着事件 A<C、A<D、B<C 和 B<D 的条件概率不变都是 1/2？答案在练习 1.10 给出。

例 1.6 逆序数的期望数值

考虑象例 1.5 那样的概率空间。让我们定义随机变量 I(e)为一组元素中关联顺序与他们的字母顺序相反的字母的个数。这个叫做结果的*逆序数*。例如，I(ABCD)=0，I(ABDC)=1 因为 D<C 但 C 的字母顺序大于 D，I(DCBA)=6 等等。通过检查我们看到 E(I)=3。现在考虑 E(I|A<B)和 E(I|B<A)。通过直接计数我们发现他们分别是 2.5 和 3.5。既然 Pr(A<B)=Pr(B<A)=1/2。引理 1.2 告诉我们 E(I)=1/2(2.5+3.5)，这是真命题。

总结，条件概率反映了在我们知道部分知识的情况下的不确定性。他们的计算步骤是：消去确认在当前情况下不可能在的基本事件来计算，然后调整剩下的基本事件的概率使他们的和为 1。与给出事件相互独立的任何事件，其概率都*不会*由于这个计算而改变。相互独立事件经常涉及相互补影响的对象（比如，多次投硬币，多次投骰子）。

级数与求和

在分析算法的时候，有几个求和公式经常用到。在本小节和下一小节中列出他们，在这里简单的描述他们有助你记住他们，注意这个术语：*级数*就是一个*数列*的和。

算术级数：连续整数的和。

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad (1.5)$$

如何记住他：写出 1 到 n 的整数。首尾两两相加，1+n，2 +n-1，3+n-2 等等。每一对都是 n+1，共有 n/2 对。（如果 n 是奇数，中心的元素记为“半对”）这个技巧不仅限于 1 到 n。

多项式级数：首先我们考虑下面平方和。

$$\sum_{i=1}^{n} i^2 = \frac{2n^3 + 3n^2 + n}{6} \qquad (1.6)$$

可以通过归纳法证明之。关键是要记住第一项是 $n^3/3$。文中不用等式(1.6)，但是你可能在一些练习需要用到。

更一般的是

$$\sum_{i=1}^{n} i^k \approx \frac{1}{k+1} n^{k+1} \qquad (1.7)$$

这是一个合理的近似，将在后面的章节讨论。（对于给定的 k 可以通过归纳法证明。）仔细比较这个和下面的“几何级数”。

2 的幂：这是最常见的几何级数。

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \qquad (1.8)$$

如何记住他：把每一个 2^i 当成二进制数的一位；则

$$\sum_{i=0}^{k} 2^i = 11....1 \text{。}$$

共有 k+1 位。如果给这个数加 1 则是 100…0= 2^{k+1} 。（结果也可以通过下面的几何级数求和公式得到。）

几何级数：

$$\sum_{i=0}^{k} ar^i = a\left(\frac{r^{k+1}-1}{r-1}\right) \qquad (1.9)$$

为了验证他，变化右边。作为一种特殊情况，令 r=1/2，我们得到：

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k} \qquad (1.10)$$

一个几何级数的特征是一个常数底和变量幂。多项式级数是一个变量底，常数幂。两者的行为差异很大。

调和级数：

$$\sum_{i=1}^{n} \frac{1}{i} \approx In(n) + \gamma \text{，其中} \gamma \approx 0.577 \qquad (1.11)$$

和叫调和数。常数 γ 叫*欧拉常量*。参见练习 1.7。

算术—几何级数：在下一个和中，i 将是一个算术级数，而 2^i 则是一个几何级数，因此名字。

$$\sum_{i=1}^{n} i2^i = (k-1)2^{k+1} + 2 \qquad (1.12)$$

他的推导是一个“部分求和”的例子，他和“部分整数求和”类似。将和重新安排成两个不同的部分，消去中间部分，只留前后两项。

$$\sum_{i=1}^{k} i2^i = \sum_{i=1}^{k} i(2^{i+1} - 2^i)$$

$$= \sum_{i=1}^{k} i2^{i+1} - \sum_{i=0}^{k-1} (i+i)2^{i+1}$$

$$= \sum_{i=1}^{k} i2^{i+1} - \sum_{i=0}^{k-1} i2^{i+1} - \sum_{i=0}^{k-1} 2^{i+1}$$

$$= k2^{k+1} - 0 - (2^{k+1} - 2) = (k-1)2^{k+1} + 2$$

Fibonacci 数列：Fibonacci 数列以递归方式定义：

$$F_n = F_{n-1} + F_{n-2} \quad n \geq 2 \qquad (1.13)$$

$$F_0 = 0, F_1 = 1$$

尽管这不是一个和，但是这个在算法分析时经常出现。

单调递增函数和凸函数

有时非常普通的属性就足以让我们得出关于函数行为的有用的结论。这样的两个属性是*单调递增*和*凸*。贯穿整个对单调递增和凸的讨论，我们假定都在区间 $a \le x < \infty$，其中 a 一般是 0，但涉及 log 时为 1。所有点都在这个区间之内，f 定义为在这个区间上。域可以是实数也可以是整数。

定义 1.7 单调函数和非单调函数

一个函数 f(x)，如果对于任意 $x \le y$ 都有 $f(x) \le f(y)$，则说函数 f 是单调递增或非递减的。如果函数-f(x)是单调递增的的，则函数 f(x)是单调递减，或非递增的。

最熟悉的单调递增函数是 x，$x^2 (x \ge 0)$，$\log(x)(x > 0)$ 和 e^x。不太熟悉的单调函数是 $\lfloor x \rfloor$ 和 $\lceil x \rceil$，这也显示出单调递增函数不一定是连续的。一个单调递减函数的例子 1/x，x>0。

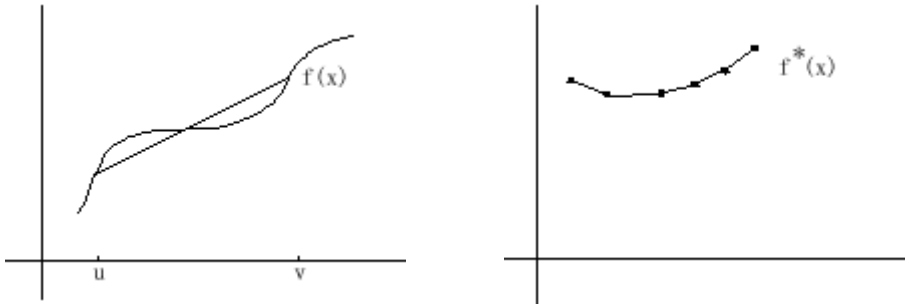
定义 1.8 线性插值函数

给定函数 f(x)，他两点 u 和 v 之间间的*线性插值*定义为如下函数

$$L_{f,u,v}(x) = \frac{(v-x)f(u)+(x-u)f(v)}{(v\_u)}$$
$$= f(u) + (x-u)\frac{f(v)-f(u)}{v-u} = f(v) - (v-x)\frac{f(v)-f(u)}{v-u}$$

(1.14)

是在 f(u)和 f(v)之间的直线（参见图 1.3a）。



(a)线性插值　　（b）将 f(n)扩展到 $f^*(x)$

插图 1.3 凸函数的讨论：（a）和（b）中的函数 f 是不同的。在（b）中，$f^*(x)$ 凸函数。

定义 1.9 凸函数

一个函数 f(x)如果是凸的，则对于所有的 u<v，区间（u,v）中所有的 $f(x) \le L_{f,u,v}(x)$。非正式的，如果函数 f(x)从不向下弯，则他是凸的。

因而象 x，x^2，1/x，e^x 这样的函数是凸的。图（b）中的函数是凸的（但不是单调的），不管他是在实数上还是在整数上；图（a）中的函数是单调的，但不是凸的。$\log(x)$ 和 $\sqrt{x}$ 业不是凸的。那么 xlog(x)呢？下面的引理开发出了一种测试凸函数的实用方法。简单的看（可以证明的）不连续（discontinuous）函数不能是凸的（译者：那 b？）。引理 1.3 介绍了一种只需比较几个空间点就能判断凸的简单方法。证明在练习 1.16 中。

引理 1.3

1. 令 f(x)是一个定义在实数上的连续函数。则当且仅当对于任意两点 x、y 都有 $f(\frac{1}{2}(x+y)) \le \frac{1}{2}(f(X)+f(y))$ 时，f(x)是凸的。换句话说，f 在 x、y 之间所有的点都比 f 在 x、y 之间的线性插值函数中间点要小。注意，f 在 x、y 之间的线性插值函数中间点只是 f(x)和 f(y)的平均值。
2. 函数 f(n)是定义在整数上，当且仅当对于任意 n、n+1、n+2 都有 $f(n+1) \le \frac{1}{2}(f(n)+f(n+2))$ 时 f(n)是凸的。就是说 f(n+1)小于等于 f(n)和 f(n+2)的平均值。

引理 1.4 总结了单调和凸的许多有用的性质。还说明了将仅定义在实数上的函数通过线性插值扩展到实数上，而且保留了单调和凸的性质。也说明了一些涉及到导数的性质。证明在练习 1.17 到 1.19。

引理 1.4

1. f(n)仅定义在整数上。令 $f^*(x)$ 是 f 通过线性插值扩展到实数上的函数（见图 1.3b）。

   a) f(n)是单调当且仅当 $f^*(x)$ 是单调的。

   b) f(n)是凸的当且仅当 $f^*(x)$ 是凸的

2. 如果 f(x)的一阶导数存在且非负，则 f(x)是单调的。
3. 如果 f(x)的一阶导数存在且单调，则 f(x)是凸的。
4. 如果 f(x)的二阶导数存在且非负，则 f(x)是凸的。（从 2，3 得到）。

利用积分求和

许多求和公式在算法分析时经常用到，可以用积分来近似（上界和下界）他们。首先让我们回顾一些有用的积分公式：

$$\int_0^n x^k dx = \frac{1}{k+1} n^{k+1} \quad , \quad \int_0^n e^{ax} dx = \frac{1}{a}(e^{an} - 1)$$

$$\int_1^n x^k In(x) dx = \frac{1}{k+1} n^{k+1} In(n) - \frac{1}{(k+1)^2} n^{k+1} \qquad (1.15)$$

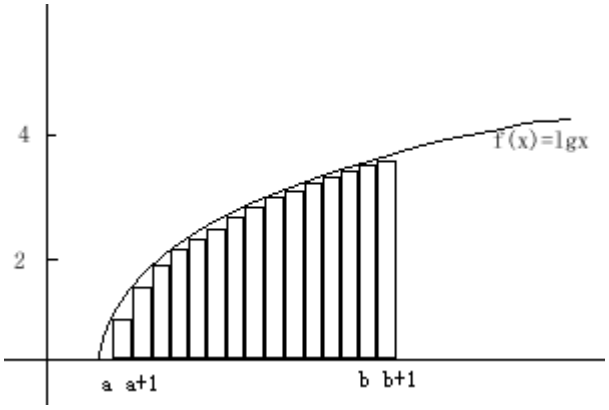
如果 f(x)是单调递增，则

$$\int_{a-1}^{b} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a}^{b+1} f(x)dx \qquad (1.16)$$

同样的，如果 f(x)是单调递减，则

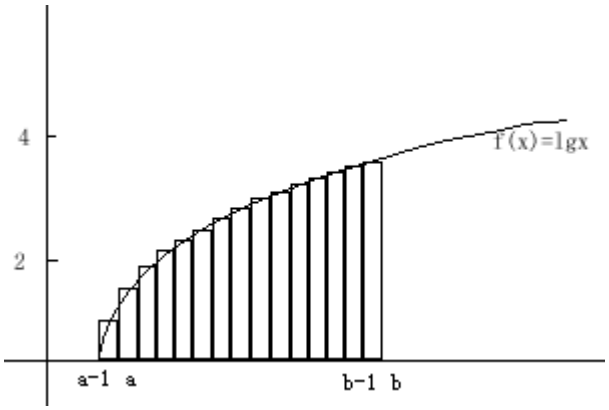
$$\int_{a}^{b+1} f(x)dx \le \sum_{i=a}^{b} f(i) \le \int_{a-1}^{b} f(x)dx \qquad （1.17）$$

图 1.4 中展示了这两种情况。后面有两个使用他的两个例子。



$$\sum_{i=a}^{b} f(i) \le \int_{a}^{b+!} f(x)dx$$

（a）Over approximation



$$\int_{a-1}^{b} f(x)dx \le \sum_{i=a}^{b} f(i)$$

(b) Under approximation

图 1.4 单调递增函数的和近似

例 1.7 估计 $\sum_{i=1}^{n}\frac{1}{i}$

$$\sum_{i=1}^{n}\frac{1}{i}\le 1+\int_1^n\frac{dx}{x}=1+Inx|_1^n=1+Inn-In1=In(n)+1$$

使用等式(1.17)得到。注意我们分开了和的第一项，对剩下的部分进行了微分近似。避免了积分下限的除 0 运算。类似的

$$\sum_{i=1}^{n}\frac{1}{i}\ge In(n+1)$$

参见等式(1.11)取得的更接近的近似。

例 1.8 $\sum_{i=1}^{n}\lg i$的下界

$$\sum_{i=1}^{n}\lg i=0+\sum_{i=2}^{n}\lg i\ge\int_1^n\lg xdx$$

使用等式(1.16)（参见图 1.4b）。

$$\int_1^n\lg xdx=\int_1^n(\lg e)Inxdx=(\lg e)\int_1^n Inxdx$$
$$=(\lg e)(xInx-x)|_1^n=(\lg e)(nInn-n+1)$$
$$=n\lg n-n\lg e+\lg e\ge n\lg n-n\lg e$$

既然 $\lg e<1.443$

$$\sum_{i=1}^{n}\lg i\ge n\lg n-1.443n \qquad (1.18)$$

使用前面例子的思想，但是使用更精确的数学方法，就可以推导出 *Stirling's formula* 给出的 n!的边界：

$$\left(\frac{n}{e}\right)^n\sqrt{2\pi n}<n!<\left(\frac{n}{e}\right)^n\sqrt{2\pi n}(1+\frac{1}{11n}) \qquad \text{其中 } n\ge 1 \qquad (1.19)$$

不等式运算

下列连接不等式的规则经常用到。

传递性	加法	正数缩放
如果 $A \le B$	如果 $A \le B$	如果 $A \le B$
且 $B \le C$	且 $C \le D$	且 $\alpha > 0$
则 $A \le C$	则 $A + C \le B + D$	则 $\alpha A \le \alpha B$

（1．20）

1.3.3 逻辑元素

逻辑是一个形式化自然语言语句的系统，他使得我们可以更精确的推理问题。最简单的命题叫*原子*公式。更复杂的命题可以使用*逻辑连词*来组成。*原子*公式例如“4<3”，“4.2 是一个整数”，“x+1>x”。注意一个逻辑命题不一定是真。一个正确的证明才能显示一个命题是真。

最常用的逻辑连词是∧（与），∨（或）和¬（非），也叫做布尔运算符。复杂命题的真值，通过连词的规则从原子命题中推出。令 A 和 B 是逻辑命题，则

1. A∨B 为真，当且仅当 A 和 B 都为真时。
2. A∧B 为真，当且仅当 A 和 B 其中一个为真，或都为真时。
3. ¬A 是真，当且仅当 A 为假时。

另一个重要的推理连词叫“推导出（implies）”，我们用符号“$\Rightarrow$”来表示他。（也有时用“$\rightarrow$”。）命题 $A \Rightarrow B$ 读作“A 推导出 B”，或是“如果 A 则 B”。（注意这个命题没有“else”子句。）“推导出”运算符可以表示成其他运算符的组合，考虑下面的等价：

$$A \Rightarrow B \text{ 在逻辑上等价于 } \neg A \vee B \qquad (1.21)$$

这可以通过真值表来检验。

一个常用的等价变换叫 *DeMorgan* 法则：

$$\neg(A \wedge B) \text{ 在逻辑上的等价于 } \neg A \vee \neg B \qquad (1.22)$$

$$\neg(A \vee B) \text{ 在逻辑上的等价于 } \neg A \wedge \neg B \qquad (1.23)$$

量词

还有一种重要的逻辑连词，*量词*。符号 $\forall x$ 叫*普遍量词*，读作“对于所有 x”，而符号 $\exists x$ 叫*存在量词*，读作“存在 x”。这两个量词可以适用于所有包含变量 x 的命题。命题 $\forall x P(x)$ 为真，当且仅当对于所有 x，P（x）都为真时。命题 $\exists x P(x)$ 为真，当且仅当有 x 使得 P（x）为真。最经常的，一个普通的量词命题是条件：$\forall x(A(x) \Rightarrow B(x))$。读作“对于所有 x，A（x）成立，则 B（x）也成立。”

量词命题服从变相的 *DeMorgan* 法则：

$$\forall x A(x) \text{ 在逻辑上的等价于 } \neg\exists x(\neg A(x)) \qquad (1.24)$$

$$\exists x A(x) \text{ 在逻辑上等价于 } \neg\forall x(\neg A(x)) \qquad (1.25)$$

有时将自然语言转换成量词命题是很麻烦的。人们说得话并不总是很有逻辑。我们需要分清“for any x ”常常意味着“for all x"，尽管“any”和“some”在正常语言中是可交换的（译者：汉语中也有类似的问题）。最好的方针是，尝试重新以逻辑语言的形式阐述一句自然语言，然后问自己是否和自然语言有同样的含义。例如，“任何（any）一个活人都必须呼吸"，可以阐述为“对所有人 x，x 要活着就必须呼吸"，和“对于有些人 x，x 要活着就必须呼吸"。那一个和原来的句子意义一样？

否定量词命题、反例

证明一个一般命题，说 $\forall x(A(x) \Rightarrow B(x))$ 是错的必要条件是什么？我们可以使用前面提到的等价条件来阐明目标。首先要认识到没有必要证明 $\forall x(A(x) \Rightarrow \neg B(x))$。这是一个过于严格的命题。否定的 $\forall x(A(x) \Rightarrow B(x))$ 是 $\neg(\forall x(A(x) \Rightarrow B(x)))$，他还可以进行一系列逻辑变换：

$$\begin{aligned} &\neg(\forall x(A(x) \Rightarrow B(x))) \text{ 在逻辑上等价于 } \exists x \neg(A(x) \Rightarrow B(x)) \\ &\qquad \text{在逻辑上等价于 } \exists x \neg(\neg A(x) \vee B(x)) \\ &\qquad \text{在逻辑上等价于 } \exists x(A(x) \wedge \neg B(x)) \end{aligned} \tag{1.26}$$

换句话说如果我们可以找到一个 x 使得 A(x) 为真而 B(x) 为假，则我们就证明了 $\forall x(A(x) \Rightarrow B(x))$ 为假。象这样的 x 称为*反例*。

逆否命题（contrapositives）

当试图证明一个命题时，经常将其转换成逻辑上的等价形式。其中的一个形式叫做*逆否命题*。$A \Rightarrow B$ 的逆否命题是 $(\neg B) \Rightarrow (\neg A)$。方程（1.21）允许我们验证当一个命题为真时他的逆否命题也为真：

$$A \Rightarrow B \text{ 在逻辑上等价于 } (\neg B) \Rightarrow (\neg A) \tag{1.27}$$

有时，证明命题的逆否命题，叫“反证法"，但是“反证法”有更精确的描述。“反证法”的严谨定义在下一小节中。

反证法

假定要证明的命题的形式 $A \Rightarrow B$。严谨的反证法先做一个附加假设 B 为假即 $\neg B$，在证明 B 自己。这样要证明的全命题就是 $(A \wedge \neg B) \Rightarrow B$。下面的恒等式证明了这个方法：

$$A \Rightarrow B \text{ 在逻辑上等价于 } (A \wedge \neg B) \Rightarrow B \tag{1.28}$$

在算法分析里很难遇到真真的反证法。但是练习 1.21 遇到了一个。大多数叫反证法的实际上是证明的逆否命题。

推理规则

迄今为止我们看到了大量的*逻辑等价*命题，或是*逻辑恒等式*：一个命题是真，当且仅当第二个命题也是真。恒等式是“可逆的”。但是很多证明使用的不可逆命题组合。要证明的完全命题有“如果*前提*，则*结论*”的形式。可逆的“如果结论，则前提”经常不是真的。逻辑恒等式在证明诸如“如果*前提*，则*结论*”的命题时不够灵活。在这种情况下，我们需要*推理规则*。

一个推理规则是一个通用的允许我们从一套给定命题中得到新结论的模式。他可以被阐述为“如果我们知道 $B_1.....B_k$，我们可以总结出 C”，这里 $B_1.....B_k$ 和 C 都是逻辑命题。这里有几个熟悉的规则：

如果我们知道			则我们可以总结出	
B	和	$B \Rightarrow C$	C	(1.29)
$A \Rightarrow B$	和	$B \Rightarrow C$	$A \Rightarrow C$	(1.30)
$B \Rightarrow C$	和	$\neg B \Rightarrow C$	C	(1.31)

有些规则通过他们的希腊和拉丁名字为人所知。等式(1.29)是 *modus ponens*, 等式(1.30)是 *syllogism*， 等式(1.31)是 *rule of cases*。这些规则不是相互独立的；在练习 1.21 中我们可以证明 rule of cases 使用了其他规则和逻辑恒等式。

1.4 分析算法和问题

我们分析一个算法的目的是要改进他，如果可能，为在解决一个问题重到底选那一个方法提供依据。我们使用下面的标准

1. 正确性
2. 所做工作的量
3. 使用空间的量
4. 简单，清晰
5. 最优性

我们将讨论每一个标准的尺度，并给出应用他们的几个例子。当考虑算法的最优性时，我们将介绍建立问题最低边界复杂性的技术。

1.4.1 正确性

建立算法正确性主要有三步。第一，在任何决定一个算法是否正确的企图之前，我们必须对“正确”有一个清晰的概念。我们需要一个清晰的命题来描述期望有效输入的特性（叫前提），和处理每一个输入的结果（叫后置条件）。在输入前提条件满足，算法结束之后后置条件也正确的话，我们尝试证明输入输出关系的命题。

算法有两个方面：要解决的问题和完成所需要的指令序列，也就是他的实现。Establishing

the correctness of the method and/or formulas used may be easy or may require a long sequence of lemmas and theorems about th objects on which th algorithm works (e.g. gtaphs,permutations, matrices).（译者：这句话不太明白，好像是说：“要建立方法的正确性，阐明使用的关于算法处理对象（例如图、排列、矩阵）的引理和定理，以及如何使用这些理论。”）例如，应用高斯消去法解系统的线性方程的有效性，依赖于一大堆线性代数的定理。在本书的算法中使用的方法不是显然正确的；他们必须被定理证明。

一旦建立了方法，我们用程序实现他。如果一个算法短小，简洁很有美感，我们一般只使用非正式的含义来使自己相信，其他的输入也会象我们期望的一样得到正确的结果。我们可能会仔细的检查一些细节（例如循环计数器的初值和终值），以及用一个简单的例子手工演算算法。这些都不能证明算法的正确性，但是非正式方法对于一些小的程序就足够了。更加正式的技术，比如循环不变式（loop invariants），可能会用来验证部分程序。3.3 节扩展了这个话题。

大多数写在类外面程序非常大而复杂的。为了证明大程序的正确性，我们可以尝试将程序分解比较小的模块；这样的话，如果所有的小模块都正常的工作，那么整个程序是正确的；剩下的工作就是要证明每一个小模块的正确性。这样只要算法和程序是以模块形式写成的，而且模块相互独立，可以分开验证，任务就简单多了（更精确的说法是“只有这样的时候，任务才是可能的”）。这是许多强壮的算法是结构化、模块化的原因之一。本书中的大多数算法都是小片断组成的，这样我们就不必处理困难的巨型算法的证明。

本书中，我们不总是正式的证明，有时候给出主题，或是算法中困难或复杂部分的解释。正确性是可以证明的，尽管对于长、复杂的程序而言这是一个可怕的任务。在第三章我们将介绍一些技术来帮助进行证明。

1.4.2 所做工作的量

如何度量一个算法的工作量？这个我们选择的度量应该能帮助我们比较解决同一个问题的两个算法，这样我们就能决定到底那一个方法更有效。如果我们比较两个算法的实际执行时间来度量两者所作工作的量将很简单，但是有很多原因使我们不能用执行时间来作为工作的度量。首先当然是，他随使用的计算机而变化，而我们不想为一种计算机开发一种理论。我们可以用程序执行的指令数或语句的数量来代替执行时间。他严重依赖于使用的程序设计语言和程序员的编码风格。他还需要我们花费时间和精力为每一个要研究的算法编写调试程序。我们想要一种工作量的度量方式，他可以告诉我们算法使用方法的效率而不依赖计算机、程序设计语言和程序员，但也不依赖于许多具体的实现细节，比如增加循环计数器的开销、计算数组索引以及设置数据结构指针的开销。我们的工作量度量必须足够的精确、足够的普通，一致我们可以开发出一种可以用作许多算法和应用程序的丰富的理论。

一个简单的算法可能由许多初始化构造和循环组成。对于这样一个算法来说，循环体执行的次数是一个公平指示工作量的好的度量。（译者：原文“ The number of passes made through the body of the loop is a fairly good indication of the work done by such an algorithm.”）当然，一次循环内的工作量和其他循环次数中可能不一样，一个算法可能有比其他算法大的循环体，但是我们只限于良好的工作量的度量。尽管有些循环有 5 步而有些是 9 步，但循环次数巨大时，一次循环的步数就显得不重要了。因此统计一个算法循环的次数是一个好主意。

在许多时候，为了分析一个算法，我们可以隔离出一种解决所研究问题的基本操作（或者是算法的组成类型），忽略初始化、循环控制和其他额外开销，仅仅统计算法执行的选择

的基本操作的量。对于许多算法，每一次循环执行一次这样的操作，所以这种度量与前一段描述的比较类似。

这里的例子是对于几个问题合理基本操作的选择：

问题	操作
在一个数组中查找 x	比较 x 和数组中的元素
两个实数矩阵相乘	两个实数作乘法（或者是乘法和加法）
数组排序	比较数组的两个元素
遍历一二叉树（参见 2.3.3 节）	遍历边
任一迭代过程，包括递归	过程调用

只要选定了基本操作，就可以粗略的估计出执行基本操作的数目，我们有了一个度量算法工作量的好标准，以及一个比较算法的好标准。这是我们在本章和其他章节中使用的方法。你可能还是不认同这是一个好的选择；在下一节中我们将列出他合理的另一个例子。现在，我们简单的摆出这个观点。

首先，有些情况下，我们本来就对基本操作感兴趣：可能是一个花费昂贵的比较操作，或者可能因为一些理论上的兴趣。

其次，我们经常对随输入的增大算法所消耗时间增大的变化率感兴趣。只要得到总操作数量和基本操作数量粗略的比例，合理的计算就可以让我们对算法对于大输入要计算多久又一个相当清晰的概念。

最后，这种工作量的度量带了很大的灵活性。尽管我们经常选择一个或最多两个操作，我们可以包括一些开销巨大的操作，极端的，我们可以选择特定计算机的一套指令作为基本操作。另一种极端情况，我们可以将“一次循环”作为基本操作。因此，通过多样的基本操作的选择方法，我们可以在分析中改变精确和抽象的程度来适应我们的需要。

如果我们为一个问题选择一个基本操作，然后发现算法指向操作的总数不于基本操作的数目成比例，怎么办？如果基本操作的开销很大怎么办？在极端情况下，我们可能为一个特定的问题选择了一种基本操作，然后我们发现解决这个问题的有些算法使用完全不同的操作，而根本不使用我们要统计的操作。这时我们有两个选择。我们可以放弃关注特定的操作，转向统计循环的次数。或者，如果我们对选择的操作特别感兴趣，我们可以将我们的研究范围限定在适用该操作的*一类算法*内。对于使用与选择基本操作不同技术的其他算法，应该个别的处理。对于一个问题的一类算法，经常通过要处理的数据的基本操作来定义（译者：原文 A class of algorithm for a problem is usually defined by specifying the operations that may be performed on the data.）。（选择基本操作正式的程度是多样的；本书中经常使用非正式的描述。）

在这一节中，我们经常使用短语“算法所作工作的度量"。他可以用术语“一个算法的复杂性”来代替。*复杂性*意味着做了多少工作，通过一些特殊的*复杂性度量方式*来度量，在我们大多数的例子中是用的特定基本操作执行的数量。注意，这里复杂性与算法有多复杂多棘手无关；一个非常复杂的算法可能有很低的复杂性。我们将使用术语“复杂性"，“工作量"和“所作基本操作的数量"，在本书中这几个术语都是可替换的。

1.4.3 平均和最坏分析

现在我们有了分析一个算法工作量的一般方法，我们需要可以简明展示分析结果的途径。工作量不能用一个简单数值来描述，因为对于不同的输入执行的步数不一样。我们首先观测到工作量依赖于输入量的大小。例如，使用同样的算法，按字母排序 1000 个名字需要

的操作肯定比按字母排序 100 个名字要多。接一个 12 阶 12 元线性方程肯定解 2 阶 2 元线性方程做的工作要多。其次，我们观察到，即使我们仅考虑一个固定的输入规模，操作执行的数目还是可能依赖于你到底输入的是什么。如果在数组中只有少数名字不按顺序、在前面的例子中的算法就只需要做很少的工作，但是如果数组十分混乱，就不得不做很多的工作。如果大多数系数都是 0，解一个 12 阶线性方程也不需要很多工作。

第一个观察指出，我们需要一种度量输入规模的标准。选一个合理的输入规模的度量通常是很容易的。这里有一些例子：

问题	输入的规模
在名字数组中查找 x	数组中名字的个数
两个矩阵乘法	矩阵的维数
遍历二叉树	树节点的数量
解系统的线性方程	等式的数量，或未知数的数量，或者两者
一个关于图的问题	图节点的数量，或边的数量，或者两者

即使输入规模固定，比如 n，操作执行的数量还是依赖于到底输入了什么。那么到底如何表示算法分析的结果呢？我们经常用*最坏情况复杂性*来描述算法的行为。

定义 1.10 最坏情况复杂性

令 D_n 是所考虑问题的规模为 n 的输入集合，I 是 D_n 的一个元素。令 t(I)，是对于输入 I 要执行的基本操作的数量。我们定义函数 W

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

函数 W(n)叫做算法的最坏复杂性。W(n)是算法对于规模为 n 的输入所要做基本操作的最大数量。

通常计算 W(n)是很复杂的。1.5 节介绍了一种当精确计算十分困难时常用的技术。最坏复杂性是很有价值得，他给出了算法所作工作量的上限。最坏分析可以用于帮助估计一个算法特定实现的时间限制。对于本书中出现的大多数算法，我们都将进行最坏分析。除了特别指出，当我们说一个算法的工作量时，都是指最坏情况时的工作量。

似乎更有用、更自然的描述算法行为的方法是指出他的平均工作量；就是说，计算规模为 n 的所有输入的基本操作执行的数量，然后取平均。实际中一些输入出现的频率要高于其他输入，所以带权平均更有用。

定义 1.11 平均复杂性

令是输入 I 出现的概率。则算法的平均行为定义为

$$A(n) = \sum_{I \in D_n} P_r(I)t(I)$$

我们通过分析算法来决定 t(I)，但是 $P_r(I)$ 不能通过分析计算。函数 $P_r(I)$ 取决于经验以及具体使用算法的应用，或者只能通过简单的假设（例如，所有情况等概率）。如果 $P_r(I)$ 很复杂，计算平均行为将很困难。当然，如果 $P_r(I)$ 依赖于使用算法的应用，函数 A

描述的算法的平均行为仅对该应用适用。

下面的例子展示了最坏和平均分析。

例 1.9 在无序数组中查找

问题：令 E 是包含 n 个元素的数组（关键字），E[0],E[1]… E[n-1]，其中元素是无序的。查找关键字为 K 的元素，如果 K 在数组中，返回索引号；如果 K 不再数组中返回－1。（将在 1.6 节中研究元素是有序时的情况。）

策略：依次将数组的元素与 K 比较，直到找到一个匹配的，或者到达数组尾。如果 K 不再数组中返回－1，否则算法返回索引号。

有一大类过程和这个类似，我们叫这些过程为普通查找例程。通常他们的子程序要复杂的多。

定义 1.12 普通查找例程

*普通查找例程*是一个处理不确定数量数据的过程，一直到数据处理完或是得到目标。下面是大概的算法：

如果没有数据了：

失败。

否则

处理一笔数据。

如果得到我们想要的：

成功。

否则

*继续处理*剩余的数据。

之所以叫普通查找例程，是因为例程经常执行诸如查找、移动元素、增加删除元素之类的简单操作。

算法 1.1 顺序查找，无序的

输入：E、n、K，这里 E 是一个有 n 个元素的数组（索引号从 0，…n-1），K 是要查找的条目 。为了简化问题，我们假定 K 和 E 的元素都是整数，和 n 一样。

输出：返回 ans，指示 K 在 E 中位置（如果 K 没有找到，返回－1）。

```
int seqSearch(int[] E, int n, int K)
{
1    int ans, index;
```

```
2    ans= -1;
3    for( index=0; index<n; ++index)
     {
4        if(K= =E[index])
         {
5            ans=index;
6            break;
         }
     }
7    return ans;
}（译者：实在不习惯不加{},都给加上了。）
```

基本操作：比较 x 和数组的元素。

最坏分析：显然 W(n)=n，最坏情况是在数组的最后一个元素才与 K 匹配的，或者根本没有一个元素符合条件。这两种情况都将导致要比较 K 和所有 n 个元素。

平均行为分析：我们将做一些简单的假设来分析第一个例子，稍后再以不同的假设分析比较复杂的第二个例子。我们就假定数组中所有的元素都不相同，则如果 K 在数组中他将只和其中一个元素相等。

对于我们的第一种情况，我们假设 K 在数组中，我们用“succ”来表示这一事件，和 1.3.2 节中概率的术语保持一致。输入可以根据 K 到底再那里出现来分类，所以有 n 个要考虑的输入。对于 $0 \le i < n$，令 I_i 表示 K 出现在数组第 i 个位置的事件。令 t(I)为输入 I 时算法进行比较的次数（即程序中 K= =E[index]的次数）。显然对于 $0 \le i < n, t(I_i) = i+1$。因此

$$A_{succ}(n) = \sum_{i=0}^{n-1} P_r(I_i \mid succ)t(I_i) = \sum_{i=0}^{n-1} (\frac{1}{n})(i+1) = (\frac{1}{n})\frac{n(n+1)}{n} = \frac{n+1}{2}$$

脚标“succ”表示，我们假定在本次计算中完成了查找。结果符合我们平均的直觉，大约数组的一半将被查找。

现在让我们考虑事件 K 不在数组中的情况，我们称为“fail”。对于这种情况只有一种输入，我们称为 I_{fail}。比较的次数 $t(I_{fail})=n$，所以 $A_{fail}=n$。

最后，我们结合 K 在数组中和 K 不在数组中两种情况。令 q 是 K 在数组中的概率。根据条件期望定律（Lemma 1.2）：

$$A(n)=P_r(succ)A_{succ}(n)+P_r(fail)A_{fail}(n)=q(\frac{1}{2}(n+1))+(1-q)n=n(1-\frac{1}{2}q)+\frac{1}{2}q$$

如果 q=1，即 K 总在数组中，则 A(n)=(n+1)/2，和前面一样。如果 q=1/2，即 50%的机会 K 不在数组中，则 A(n)=3n/4+1/4；大概要检查 3/4 的数组元素。这是例 1.9 的结论。

例 1.9 展示了我们如何解释规模为 n 的输入集合 D_n。我们仅考虑了影响算法行为的输入，即 K 在不在数组中以及 K 出现的位置，而不是所有可能的名字和数组大小。D_n 中的一个元素 I 可以认为是一个 K 在指定位置出现的数组的集合（或等价类）。则 t(I)是对于输入 I 的基本操作执行的次数。

显然，造成算法有最坏行为的输入依赖算法本身，而不是问题。对于算法 1.1 当 K 在数组的最后一个位置时，最坏情况发生。对于按逆序查找的算法（例如，从 index==n-1 开始），当 K 出现在数组最前面时发生最坏的情况。（另一种最坏情况发生的条件是 K 不在数组中。）

最后，例 1.9 展示了在进行查找和排序的平均分析时经常用到的假设；所有的元素都是不同的。对不同元素的平均分析给出了对重复元素较少情况的的比较接近的平均行为。如果这里有很多重复元素，就很难对 K 第一次出现在数组的什么地方做出合理的假设。

例 1.10 矩阵乘法

问题：令 A=(a_{ij})是一个 m×n 矩阵，B=(b_{ij})是一个 n×p 矩阵，两者都是实数矩阵。计算矩阵 C=AB。（这个问题在第十二章中大量讨论。在许多情况下，我们假设矩阵是方阵，即 m=n，p=n。）

策略：根据矩阵积的定义使用下面的算法：

$$c_{ij}=\sum_{k=0}^{n-1}a_{ij}b_{kj}\qquad 0\le i<m,0\le j<p$$

算法 1.2 矩阵乘法

输入：矩阵 A 和 B，整数 m,n,p，其中 A 是一个 m×n 矩阵，而 B 是一个 n×P 矩阵。

输出：矩阵 C，m×p 矩阵。C 被传递进来，算法填充之。

matMulit(A,B,C, m, n, p)

```
{
        for(i=0; i<m;++i)
        {
            for(j=0;j<p;++j)
            {
                cij=0;
                for(k=0;k;k<n;k++)
                    cij+=aik*bkj
            }
        }
}
```

基本操作：矩阵元素的乘法

分析：为了计算 C 的每一个元素，要做 n 次乘法。C 有 m*p 歌元素，所以

$$A(m,n,p)=W(m,n,p)=mnp$$

对于 m=n=p 时，$A(n)=W(n)=n^3$。这是例 1.10 的结论。

例 1.10 展示了有些算法只是执行指令，因此工作量不依赖于具体的输入细节；仅依赖于输入的规模。这种情况下，最花和平均都是一样的。对于其他算法，就不一定了。

最坏和平均分析行为分析的概念是很有用的，即使我们选择了一种很复杂的工作量的度量（比如，执行时间）。显然，工作量通常依赖于输入的规模和输入的特性，这使得不管选择了什么度量的方式我们都要研究最坏和平均行为。

1.4.4 空间使用

一个程序使用的内存单元数量和执行程序所需要的时间一样，依赖于具体的实现。然而，通过检查算法可以对空间的使用情况作出一些结论。一个程序需要存储空间来放置他的代码、常量和程序使用的变量，以及输入的数据。还可能需要空间来操作数据，存储需要输出的结果信息。输入数据本身可能也需要一些 form，通常也需要不少空间。

如果输入数据有一个自然的形式（比如，一个数组或矩阵），则我们撇开程序和输入，分析还需要多少*额外*的空间。如果额外空间的大小是一个于输入规模有关的常量，则这样的算法称为 *in place*。这个属于特指排序算法。（*in place* 的一个不严格定义常用于，额外空间不为为常数的但是仅和输入规模成对数关系，原因是对数函数增长缓慢；我们会在使用不严格定义是特别指明。）

如果输入可以表示为变量的形式，作为我们认为输入本身所需要的空间也是额外空间的一部分。一般情况下，我们用“单元”的数量，而不对“单元”做严格的定义。你可以认为单元足够大可以容纳一个数值或对象。如果使用的空间依赖于特定的输入，可以做最坏和平均分析。

1.4.5 简明性

通常，但不总是，最简洁最直接解决问题的方法不是最有效的。可是算法的简明性还是一个需要的特性。这可以更简单的验证算法的正确性。当选择一个算法时，调试一个算法的时间也是需要考虑的，但是程序经常被使用时，他的效率将是主要的选择依据。

1.4.6 最优性

不论我们多么聪明，我们都不能将一类问题的算法性能提高的超过一个邻接点。每一类问题都有固有的复杂性，就是说要解决问题需要做的工作量有一个最小值。为了分析一个问题的复杂性，与之对立的是我们前面讲到的分析算法的复杂性，我们选择一类算法（通常通过算法允许执行的操作来决定）和一种复杂性的度量，例如基本操作执行的次数。之后我们会问解决问题究竟要执行多少基本操作。如果其中有一个算法执行基本操作的次数比别的算法都少（最坏情况），我们说这个算法是最优的（最坏情况）。注意当我们说这一类算法时，并不意味着目前人们想到的这些算法。我们指所有可能的算法，包括海没有被发现的。“最优”不意味着“已知最好的”，他意味着“所有可能的最好的”。

1.4.7 低限和问题复杂度

然而我们怎么才能知道一个算法是最优的呢？我们是否必须分析所有可能的算法？（包括一些我们目前还没有想出来的？）幸运的是，不；我们可以从理论上证明要解决一个问题执行基本操作数的低限。然后任何算法执行的基本操作的数量都可以朝这个目标优化。因此，为了找到一个好的算法，或换个角度说为了解决一个问题：“解决一个问题必须做的工作有多大？”， 我们必须要完成两个任务。

1. 设计一个看上去很有效率的算法 **A**。分析 **A**，找到在输入规模为 n 时，**A** 在最坏情况下执行基本操作的数量的函数 $W_A(n)$。
2. 找到一个函数 F，证明对于考虑的一类算法，在输入规模是 n 时，算法至少要执行 F(n) 步基本操作。

如果函数 $W_A(n)$与 F(n)相等，则算法 **A** 是最优的(对于最坏情况)。如果不等，可能有更好的算法或是有更好的低限。考虑，分析一个特定算法解决一个问题执行基本操作的*上限*数，以及在条款 2 中讲到的给出解决问题基本操作数*低限*（最坏情况）的理论。在本书中，我们对于那些最优算法已知的问题，以及那些在所知的最低限和所知的最好算法之间仍有差距得问题。下面是这两者简单的例子。

在最坏情况下算法*低限*的概念对于计算复杂性是十分重要的。例 1.11 和 1.6 节研究的问题以及第四章和第五章将帮助你理解低限的概念，并展示一些建立低限的技术。你必须紧记“F 是一类算法的低限”的定义意味着属于这一类的*所有*算法，对于规模为 n 的*所有*输入，算法都*至少*执行 F(n)步基本操作。

例 1.11 查找数组中最大的元素

问题：在 n 个元素的数组中找到最大的元素。（假设类型为 **float**；当然也可以是其他类型。）

该类算法：算法可以比较合拷贝类型为 float 的数值，但是不能做其他操作。

基本操作：比较数组的元素和其他 **float** 类型的数据。可以是存储在其他数组中的或是存储变量中。

上限：假设数组 E 中存在要找到值。下面的算法找出最大值。

算法 1.3 查找最大值

输入：E，数组，索引号从 0,… n-1; $n \geq 1$，n 是数组元素的数量。

输出：返回 max，E 中最大的元素。

```
int findMa(E, n)
{
    max=E[0];
    for( index=1; index<n; ++index)
    {
        if(max<E[index])
            max=E[index];
    }
    return max;
}
```

在 **if**(max<E[index])这一句比较数组的元素，他执行了 n-1 次。因此 n-1 是最坏情况下所作比较操作的上限值。有算法可以更少吗？

低限：为了建立低限，我们假定数组中的元素都不相同。这个假设是可行的，因为如果我们可以为这种输入（数组的元素都不同）在最坏情况下建立低限，他也是考虑所有输入情况下最坏行为下的低限。

在 n 个元素都不同的数组中，n-1 个元素不是最大的。我们可以得出结论，如果一个元素不是最大，当且仅当他必至少一个元素小。因而，n-1 个元素都必须在算法中通过比较来筛选掉。每一次比较去掉一个元素，所以至少要做 n-1 次比较。也就是说，如果算法结束时还有两个或多个没有被筛选掉的元素存在，就不能肯定他是不是最大。因此 F(n)=n-1 是要做比较次数的低限。

结论：算法 1.3 是最优的，这是例 1.11 的结论。

在建立例 1.11 的低限中我们可能需要一些稍微不同的观点。如果我们给出一个算法和 n 个元素的数组，于是算法终止并且在做少于 n-1 次比较之后给出了一个结果，则我们可以证明对于某些形式的输入算法给出了*错误*的结果。如果不做多于 n-2 次的比较，就会有两个元

素永远不会被筛选掉；也就是说无法知道其中一个是否比另一个小。算法可以指定其中任意一个为最大元素。我们可以简单的将另一个替换成一个比较大的数（如果必须）。既然所有比较的结果都和前面的类似，算法给出和前面相同的结果，他将是错误的。（译者：此处没有搞明白，原文是“The algorithm can specify at most one of them as maximum. We can simply replace the other with a larger number(if necessary). Since the results of all comparisons done will be the same as before, the algorithm will give the same answer as before and it will wrong.”似乎说的是，将 n-2 替换成一个较小的数，会得到和 n-1 一样的结论。）

这个观点是逆否命题的一个证据（参见 1.3.3 节）。我们证明“如果 A 在任何情况下都做少于 n-1 次比较，则 A 是不正确的。”通过逆否命题，我们可以得到结论“如果 A 是正确的，则 A 在任何情况下都要至少要做 n-1 次比较。”他展示了一种对建立低限十分有效的技术，即，如果算法不做足够的工作，就可以找到一种输入使算法得到错误的结果。

例 1.12 矩阵乘法

问题：令 $A=(a_{ij})$， $B=(b_{ij})$ 是两个 n×n 实数矩阵。计算矩阵 C＝AB；

该类算法：算法可以执行实数的乘法，除法，加法和减法。

基本操作：乘法

上限：常用的算法（参见例 1.10）做 n^3 次乘法，因此至多需要 n^3。

低限：有文献证明至少需要 n^2 次乘法。

结论：没有办法根据这些信息确定常用的算法是不是最优的。许多学者在努力改善低限，证明至少需要 n^2 乘法，其他的正在寻找更好的算法。

到此为止，常用的算法*不是*最优的；有算法可以达到近似 $n^{2.376}$ 次乘法。

这个算法是最优的吗？目前低限还没有进一步提高，所以我们不知道是否还有算法可以更低。

到现在为止，我们都在讨论低限和最坏情况的行为。平均行为如何呢？我们可以使用和最坏行为一样的方法。选择一个看上去比较好的算法，位算法找到一个函数 A(n)，即在输入规模为 n 时在平均行为下，算法要做 A(n)次基本操作。然后再证明这一类算法对于规模 n 的输入至少执行 G(n)次操作。如果 A=G，我们可以所算法的平均行为是最优的。如果不是，继续找更好的算法或更好的低限（或者两者）。

对于许多问题分析执行操作的数量是很困难的。通常，如果一个算法执行操作的次数在已知的最优执行次数范围之内（有时这个算法就是所知的最好的算法。），我们就认为这个算法是最优的。在 1.5 节，我们将得出一种分析许多问题是否在已知的最优执行次数范围之内，即使我们不能执行精确的分析。

我们可以使用与分析时间相同的方法来研究空间的使用。分析一个特定算法使用空间数量的上限，在证明一个理论上的低限。我们可以认为对一个问题的算法的时间和空间都是最优的吗？这个问题的答案是，有时是。对于许多问题存在时间和空间的平衡。

1.4.8 实现和编程

*实现*就是将一个算法转换成计算机程序。算法可以通过与计算机程序设计语言类似的操作变量和数据结构的构造来描述，或者用非常抽象的、高级的用自然语言解释解决抽象问题的方法，不提到关于计算机的任何事物。因此实现可能是灵巧、直接的翻译工作，也可能是非常困难的，需要程序员在选择数据结构上做很多判断。以后在适当的地方，我们将讨论在一般情况下，实现如何选择数据结构，如何将以英语描述的算法变成程序。进行这样的的讨论有两个原因。第一，这是生产一个好程序很自然而且重要的部分。第二，考虑实现细节对于分析算法很必要的；在抽象对象（比如图，集合）上执行不同操作的时间依赖于对象是如何表示的。例如，如果集合以链表表示，则连接两个集合只需要一次或两次操作。但是如果集合以数组表示，则需要大量的操作——将一个集合的元素拷贝到另一个集合中。

狭义上讲，实现或简单的编程，意味着将一个算法描述细节和算法使用的数据结构转换成一种特定计算机的程序。此时我们的分析是实现无关的（implementation-independent）；换句话说他们将独立与计算机和使用的程序设计语言，独立于算法或程序的小细节。

程序员也可以在分析算法时考虑特定计算机的信息。例如，如果不止统计一个操作，操作可以根据执行时间建立权值；或者估计一个程序到底要执行多少秒（在最坏情况和平均情况下）。有时使用一些具体计算机的知识将得到一个新的分析结果。例如，如果计算机有一些不寻常的、强力的可以被高效执行的指令，则可以研究使用这些指令的一类算法，将这些指令作为操作统计。如果计算机只有一套很有限的指令集，使得实现基本操作很低效，则要考虑不同的一类算法。然而一般的，如果实现无关的分析做的很好。则程序依赖分析将作为一种补充。

当然考虑一种特定的实现时，也可以对算法的空间使用情况做细节分析。

任何关于问题输入的知识，都可以提高算法分析的精度。例如，如果输入仅限于所有可能输入的一个子集，就可以只对这个子集作最坏分析。就像我们注意到的，一个好的平均行为分析依赖于对不同输入发生概率的认识程度。

1.5 简单函数的渐进增长率

我们的度量算法工作量的方法到底如何？我们对两个算法的比较有多精确？因为我们不能计算算法每一个步骤的执行情况，所以我们的分析必然存在不精确的因素。

假设一个问题的算法作了 2n 次基本操作，因而大概共有 2cn 次，c 是某个常量，，另一个算法做 4.5n 次基本操作，或者共有 4.5c`n.。那个运行的更快？我们确实不知道。第一个算法可能做了许多额外的工作；就是说他的比例系数可能很高。因而如果一个函数描述的两个算法有不同的常量因子，他可能是区别两个算法无意义的尝试（除非我们提高分析的精度）。我们考虑在同一复杂性类中的算法。

假设一个问题的算法做 $n^3/2$ 次乘法而另一个算法做 $5n^2$。那一个算法更快呢？对于 n 比较小的情况，第一个较快，但是对于 n 很大的情况，第二个更快——即使他做更多开销的操作。立方函数的增长率远大于平方函数，常数因子在 n 变的很大时影响不大。

就像在例子中暗示的，我们希望找到一种比较简单函数的方法，能忽略常数因子和小规模输入的。这正是我们将要研究的*渐进增长率*、*渐进阶*、或简单的称为函数的*阶*。

他能合理的忽略常数因子和小规模输入吗？没有完全非级数、非数学的类推来帮助你理解我们为什么使用渐进阶。假设你要选择了一个城市来居住，你的主要标准是那要有非常炎

热的气候。选择是 El Paso、Texas、和 Yuma、Arizona。他们之间温度差别不大，不是吗？但是要你在三个城市间选择呢，El Paso、Yuma 和 Anchorage，Alaska。你肯定会马上会划去 Anchorage。这类似于两个函数有相同的阶，第三个有不同的阶。知道阶，使我们能更容易的选择；我们可以消去离我们目标远的那些选择。

现在，在 El Paso 和 Yuma 如何选择呢（或者在两个阶一样的算法中如何选择呢）？我们可以查看温度记录，然后找出那一个城市的平均温度比另一个城市高一点。这可以类推到比较有相同阶的两个函数；对于算法，这表示统计所有操作，包括统计他们的精确运行时间 。另一个目的可能导致另一个标准，可能合适的工作和文化氛围是选择城市的标准，或者使用了多少额外空间是选择算法的标准。

有一天 Anchorage 会比 El Paso 暖和吗？这是肯定的；当寒流传到 El Paso 时，Anchorage 很可能有一个温暖的春天。这并不是说“一般情况下，Anchorage 比 El Paso 要冷”是错误的。在定义里我们将给出对于 O， Θ和其他“阶集合”函数行为的比较，将忽略 n 的小值。忽略小参数（对于算法是输入规模）类似于忽略 Anchorage 比 El Paso 或 Yuma 要冷的少数几天

1.5.1 渐近阶的定义和符号

我们将使用一些自然数和实数的常用记号。

定义 1.13 自然数和实数的符号

1. *自然数*集合表示为 $N = \{0,1,2,3....\}$ 。
2. 正整数表示为 $N^+ = \{1,2,3....\}$ 。
3. 实数集合表示为 R。
4. 正实数集合为 R^+
5. 非负实数的集合表示为 R^* 。

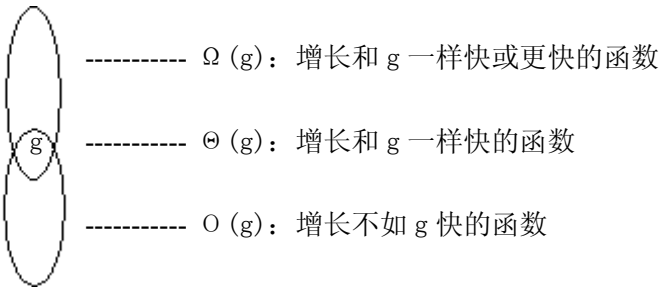


图 1.5 Ω Θ O

令 f 和 g 是从 N 到 R^* 的函数。图 1.5 非正式的描述了我们使用集合和函数阶之间的关系。记住这副图，非正式的定义将帮助验证下面的正式定义和属性。

定义 1.14 集合 O(g)

令 g 是一个从非负整数到正实数的函数。则 O(g)是函数 f 的集合，也是从非负整数到正实数的，存在实数常量 c>0 和非负整数常量 n0，对于 $n \geq n0$,有 $f(n) \leq cg(n)$ 。

将 g 作为某些*给出*函数，将 f 当作我们要分析的函数是十分有用的。注意函数 f 可能在 O(g)内，即使对于所有 f(n)>g(n)。重要的一点是 f 是所有 g *乘一个常数*的上限。也就是说当 n 比较小的时候，不考虑 f 和 g 之间的关系。图 1.6 展示了一些函数的阶关系。（注意图 1.6 中的函数都是定义在 R^+ 或 R^* 上连续的。描述我们将要研究算法大多数行为的函数有许多自然的扩展。）

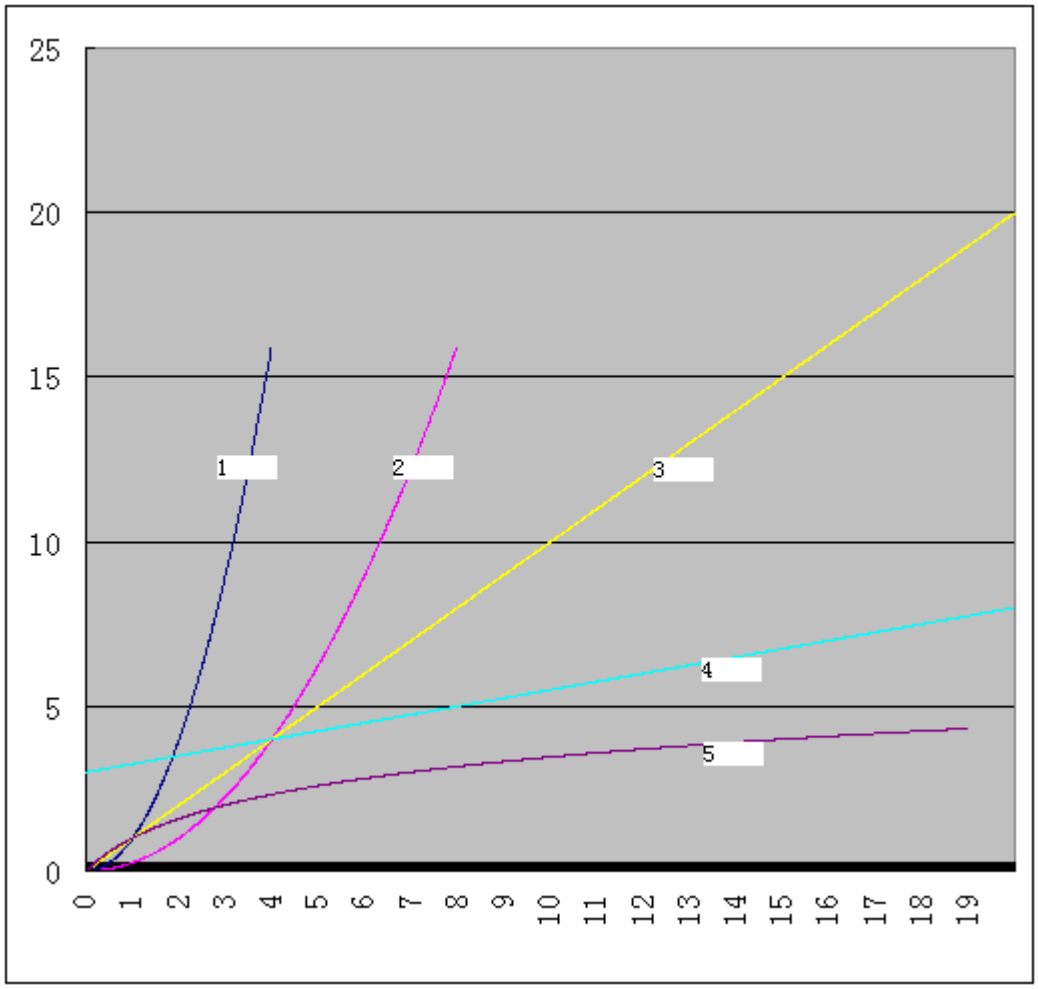


图 1.6 五条曲线分别是 $f_1(X) = x^2$ ， $f_2(x) = \frac{x^2}{4}$ ， $f_3(x) = x$ ， $f_4(x) = \frac{x}{4} + 3$ ， $f_5(x) = \lg x$ 。函数的阶是： $f_3 \in O(f_4)$ ，即使是在 x>4 的时候 $f_3(x) > f_4(x)$ ，因为这两者都是线性的。 f_1 和 f_2 有相同的阶。他们增长的比其他三个函数都要快。 f_5 是展示函数

中增长最慢的函数。

集合 O(g)常读作“g 的大 O”，或“g 的 O”，尽管“oh”实际上是希腊字母 omicron。另外，尽管我们定义 O(g)为一个集合，但实际中常说“f 是 g 的 O”，而不是“f 是 g 的 O 的一个元素”。

另外一个可选择的展示 f 在 O(g)中的技术：

引理 1.5 如果 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c<\infty$，则 $f\in O(g)$，包含极限为 0 的情况。

就是说，如果 f/g 的极限存在且不是无穷，则 f 增长的不如 g 快。如果极限是无穷，f 比 g 增长的快。

例 1.13 有不同渐进阶的函数

令 $f(n)=n^3/2$，$g(n)=37n^2+120n+17$。我们将看到 $g\in O(f)$，但是 $f\notin O(g)$。

既然对于 $n\geq 78$，g(n)<1f(n)，这遵从 $g\in O(f)$。我们也可以这样得到同样的结论：

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\lim_{n\to\infty}\frac{37n^2+120n+17}{n^3/2}=\lim_{n\to\infty}(74/n+240/n^2+34/n^3)=0。$$

我们可以通过计算 $\lim f/g=\infty$，得到 $f\notin O(g)$。这里有另一个可选择的方法。我们根据一个条件假定 $f\in O(g)$。如果 $f\in O(g)$，则存在常数 c 和 n_0 使得对于所有的 $n\geq n_0$，

$$\frac{n^3}{2}\leq 37cn^2+120cn+17c.$$

$$\frac{n}{2}\leq 37c+\frac{120c}{n}+\frac{17c}{n^2}\leq 174c.$$

既然 c 是一个常数，n 可以任意的大，所以不可能对于所有的 $n\geq n_0$，都有 $n/2\leq 174c$。

当 f 和 g 在实数上是连续的不相同函数，下面的定理对于求极限十分有用。

定理 1.6（L'Hôpital 法则）令 f 和 g 是不同的函数，分别有导数 f'和g'，且

$$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$$

则

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}.$$

例 1.14 使用 L'Hôpital 规则

令 $f(n) = n^2$， $g(n) = n\lg n$。我们将得出 $f \notin O(g)$ 但 $g \in O(f)$。首先易得

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{n\lg n} = \lim_{n\to\infty} \frac{n}{\lg n}$$

现在我们注意到（参见引理 1.1） $\lg n = \ln(n)/\ln(2)$ 符合使用 L'Hôpital 的条件：

$$\lim_{n\to\infty} \frac{n\ln(2)}{\ln n} = \lim_{n\to\infty} \frac{\ln(2)}{1/n} = \lim_{n\to\infty} n\ln(2) = \infty$$

因此 $f \notin O(g)$。但 $g \in O(f)$，因为无穷的倒数为 0。

Ω (g) 的定义：至少增长的和 g 一样快的函数，这个定义是 O (g) (原注：计划考虑其他书和文章的读者将会注意到 Ω 的定义有一些小变化：短语“对于所有的”可能弱化成“对于无限多的”。Θ 的定义也有类似的变化。) 定义的对偶。

定义 1.15 Ω(g)集合

令 g 是从非负整数到正实数的函数。则Ω(g)是函数 f 的集合，也是从非负整数到正实数的函数，存在常数 c>0，和非负整数常量 n0，对于所有 $n \geq n0$ 都有 $f(n) \geq cg(n)$

另一个可选择的判断 f 在Ω(g)方法是：

引理 1.7 函数 $f \notin \Omega(g)$，如果 $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$，包含极限为 ∞。

定义 1.16 Θ(g)，g 的渐进阶

令 g 是从非负整数到正实数的函数。则 $\Theta(g)=\mathrm{O}(g)\cap\Omega(g)$，就是说，函数集合即属于O(g)又属于Ω(g)。“$f\in\Theta(g)$”最普通的读法是“f 是 g 的阶（f is order g）”。我们经常使用短语“渐进阶”，也使用术语“渐近阶复杂度”。

我们也有：

引理 1.8 如果对于 $0<c<\infty$ 的常数 c 有 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c$，则函数 $f\in\Theta(g)$。

例 1.15 某些算法的渐进阶

算法 1.1（顺序查找，无序的）和算法 1.3（查找最多的元素）的最坏情况复杂度都是 Θ(n)。算法 1.2(最坏和平均)矩阵乘法的复杂度在 m=n=p 时是 $\Theta(n^3)$。

讨论阶集合的术语通常是不精确的。例如：“这是一个阶 n^2 算法”实际上意味着描述这个算法行为的函数在 $\Theta(n^2)$ 中。练习中列举了常用的几个阶集合，以及他们之间的关系，比如 $n(n-1)/2\in\Theta(n^2)$。

有时我们希望指出一个函数的渐进阶精确的比一个函数小或大。我们可以使用下面的定义。

定义 1.17 集合o(g)和ω(g)

令 g 是一个从非负整数到正实数的函数。

1. o(g)是函数 f 的集合，也是从非负整数到正实数的函数，$\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$。

2. ω(g)是函数 f 的集合，也是从非负整数到正实数的函数，$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$。

一般情况，“o(g)”和“ω(g)”读作“g 的小 oh”和“g 的小 omega”。这常常记忆为在函数o(g)中的函数“小于”在 O (g) 中的函数。但是，ω(g)就不是很自然了，因为在ω(g)中的函数可能大于在 Ω (g) 中的函数！o(g)更多的属性，参见练习 1.33 和 1.34。

算法	1	2	3	4	

时间函数	$33n$	$46n\lg n$	$13n^2$	$3.4n^3$	2^n
输入规模（n）	解决时间				
10	0.00033s	0.0015s	0.0013s	0.0034s	0.001s
100	0.003s	0.03s	0.13s	3.4s	$4\cdot10^{16}$ yr.
1,000	0.033s	0.45s	13s	0.94s	
10,000	0.33s	6.1s	22min	39days	
100,000	3.3s	1.3min	1.5days	108yr	
允许的时间	最大解决的输入规模（大概）				
1s	30,000	2,000	280	67	20
1m	1,800,000	82,000	2,200	260	26

表 1.1 函数增长的速度

1.5.2 渐进阶的重要性

表1.1（原注：这张表（除了最后一列）和表 1.2 根据 Jon Bentley 的《Programming Pearls》（Addison-Wesley, Reading,Mass.,1986）改编，得到授权。）展示对于同一个问题几种实际算法的运行时间。（最后一列不对应任何问题的算法；在这里用他来证明指数函数增长的多么快，以及指数算法是如何的糟糕。）纵观整个表，可以看出对于复杂性很高的算法，在输入规模增长时所消耗的时间急剧增加。表中的一个重要的经验即使是在输入很小的时候，Θ(n)和Θ(nlogn)的大常数因子也不会使得他们比其他算法要慢。

表的第二部分显示出了不同的增长率，在输入规模增大情况下所消耗的计算时间（当然你可以使用更快的计算机来求计算时间）的不同。一般情况下，并不是我们多计算 60 倍的时间（或者把速度提高 60 倍），就可以多处理 60 倍的输入；这种情况仅当算法的复杂性为 O(n)时才成立。例如对于 $\Theta(n^2)$ 的算法，仅能多处理 $\sqrt{60}$ 倍的输入。

为了进一步显示（To further drive home the point that…）当输入规模很大时，渐近线对时间造成的影响要比常数因子更重要，参考表 1.2。表 1.1 中的一个立方算法在 Cray-1 超级计算机上实现；对于规模为 n 的输入要运行 $3n^3$ 纳秒。线性算法在 TRS-80（80 年代便宜的 PC）；要运行 19.5n 微秒(19,500,000n 纳秒)。尽管线性算法的常数因子比立方算法的常数大 650 万倍，当输入 $n \geq 2500$ 时线性算法还是要快。（到底是大输入还是小输入倚赖于具体的问题）。

如果我们关注函数的渐进阶（因而说 n 和 1,000,000n 是同一个类型），则当我们说两个函数有*不同*阶时，我们是在强调描述这两个算法的函数的不同。如果两个函数有*相同*的阶，他们可能在大常数因子上有差别。但是，现在的问题是在给定的时间内在更快速的计算机上一个算法能处理最大的输入规模，常数的值是不相干的。就是说，在对于表 1.1 的最后两行，常数的值和增长不相干。让我们更仔细的观察这些数字的含义。

假设，我们固定一定量的时间（一秒，一分钟——具体是多少并不重要）。令 s 为特定算法在给定时间之内的能处理的最大输入规模。现在假设我们允许 t 倍的时间（或者我们计算机的速度增加 t 倍，或者是因为技术的进步，或者只是因为我们过时了于是买一个更贵的机器）。表 1.3 展示了几种不同复杂性对加速的影响。

	Cray-1 Fortran	TRS-80 Basic
n	$3n^3$ 纳秒	19,500,000n 纳秒
10	3 微妙	0.2 秒
100	3 毫秒	2.0 秒
1,000	3 秒	20.0 秒
2,500	50 秒	50.0 秒
10,000	49 分钟	3.2 分钟
1,000,000	95 年	5.4 小时

Cray-1 是 Cray Research 公司的商标，TRS-80 是 Tandy 公司的商标。

表 1.2 渐进阶胜出

对于输入规模 n 执行步数 $f(n)$	最大可能的输入规模 s	最大可能输入规模增加 t 倍时相应的时间 s_{new}
$\lg n$	s_1	s_1^t
n	s_2	ts_2
n^2	s_3	$\sqrt{t}s_3$
2^n	s_4	$s_4 + \lg t$

表 1.3 在最大输入规模确定时增加计算机速度得到的效果

在第 3 列的值根据下列公式计算

$$f(s_{new}) = \text{加速之后的步数} = t \bullet (\text{加速前的步数}) = tf(s)$$

和 s_{new} 的关系

$$f(s_{new}) = tf(s)$$

现在，如果我们将第一列的函数乘以常数 c，在第三列的值不会改变！这就是我们说常数的值与输入为算法能处理的最大输入规模时增加计算时间（或速度）产生的效果无关。

1.5.3 O Ω 和 Θ 的属性

渐进阶由许多有用的属性。大多数的证明都留到练习中；他们都可以从定义从轻松得到 。

对于所有的属性，假定 f、g、h 符合 $N \to R^*$ 。就是说，函数都是从非负整数到非负实数的 。

引理 1.9 如果 $f \in O(g)$ 以及 $g \in O(h)$，则 $f \in O(h)$；就是说，O 是传递的。同时，Ω、Θ、 o 和 ω 都是传递的。

证明 令 c_1 和 n_1 对于所有的 $n \geq n_1$，都有 $f(n) \leq c_1 g(n)$。令 c_2 和 n_2 对于所有的 $n \geq n_2$，都有 $g(n) \leq c_2 h(n)$。则对于所有的 $n \geq \max(n_1, n_2)$，$f(n) \leq c_1 c_2 h(n)$。所以 $f \in O(h)$。对于Ω、Θ、o和ω的证明类似

引理 1.10

1. 当且仅当 $g \in \Omega(f)$ 时，$f \in O(g)$。

2. 如果 $f \in \Theta(g)$，则 $g \in \Theta(f)$。

3. Θ 定义了一种函数的等价关系（参见 1.3.1 小节）。每一个 $\Theta(f)$ 都是可以称为复杂类的等价类。

4. $O(f+g) = O(\max(f,g))$。对于 Ω 和 Θ 也有同样的等式。（在分析复杂算法的时候他们是很有用的，这时 f 和 g 可能描述的算法的不同部分。）

既然 Θ 定义了一种等价关系，我们可以通过类中的任意函数来表示算法的复杂性类。经常选择最简单的表示。因此如果一个通过函数 $f(n) = n^3/6 + n^2 + 2\lg n + 12$ 描述的算法，我们简单的说：算法的复杂性在 $\Theta(n^3)$。如果 $f \in \Theta(n)$，我们说 f 是线性的；如果 $f \in \Theta(n^2)$，我们说 f 是平方的；如果 $f \in (n^3)$，我们说 f 是立方的（原注：注意术语线性、平方和立方比他们在数学上的用法要宽松）。 $O(1)$ 表示函数以常量为边界（对于大的 n）。

这里有两个有用的定例。他们的证明用到了在 1.5.1 小节中提到的技术，特别是 L'Hôpital 法则；证明流到练习中了。

定例 1.11 lgn 对于任意α>0 都在 $o(n^\alpha)$ 内。就是说对数函数比正指数的幂函数都要慢（包括指数为分数的情况）。

定例 1.12 n^k 对于任意 k>0 都在 $o(2^n)$ 内。就是说，幂函数比指数函数 2^n 要慢。（事实上，当指数函数的底大于 1 时，幂函数比指数函数要慢）。

1.5.4 渐进阶的和

阶符号使我们可以容易的导出和记住在分析算法中遇到的各种渐进阶的和。有些求和结论在 1.3.2 中定义。

定例 1.13 令 d 非负常量且令 r 是一个不对于 1 的正常量。

1. *多项式序列*的和将指数增加 1。度为 d 的*多项式序列*的和有 $\sum_{i=1}^{n} i^d$ 的形式，应用规则得这种形式的和在 $\Theta(n^{d+1})$ 内。

2. *几何序列*的和在最大一项的 Θ 之内。*几何序列*的和有 $\sum_{i=a}^{b} r^i$ 的形式。

   规则适用于 0<r<1 或 r>1 的情况，但是在 r=1 是是不一定的。界限 a 和 b 不能都是常量；典型的，上界 b 是 n 的某个函数，下界 a 是一个常量。

3. 对数序列的和在 Θ（最大项）。对数序列的和有 $\sum_{i=1}^{n} \log(i)$ 的形式。使用规则得到和在 $\Theta(n\log(n))$ 内。试中对数的底是多少关系不大。

4. 多项式－对数序列的和，有 $\sum_{i=1}^{n} i^d \log(i)$ 的形式，在 $\Theta(n^{d+1}\log(n))$。

证明 看图 1.7。既然定例中所有的序列都有 $\sum_{i=1}^{n} f(i)$ 的形式，这里 $f(i)$ 是单调的。很显然，高为 $f(n)$，宽为 n 的大矩形是和的上界。同时，如同在图 1.4(b)中看到的，i=0 到 i=n 之间的在 $f(i)$ 之下的图的面积

是和的下界。对于多项式和对数序列的情况，面积可以简单的由下面比较小的灰矩形界定。在左边的图中大矩形的面积是 n^{d+1}，而小矩形的面积是 $n^{d+1}/2^{d+1}$。既然两个矩形的面积有同样的渐进阶，则和也应该是这个阶。在右边的图中，两个是 nlogn 和(n/2)(logn-log2)。多项式-对数序列是类似的，但是这个技术不能用于几何序列。几何序列的规则可以直接由 1.3.2 节的等式（1.9）得出。

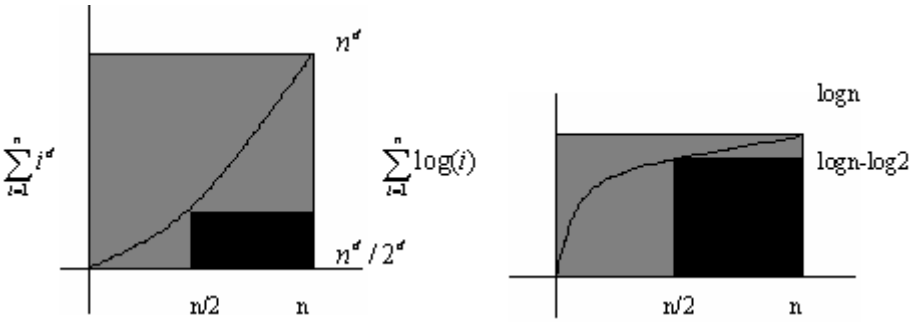


图 1.7 矩形提供了许多和的上界和下界。当两个矩形的区域都有同样的渐进阶时，就是和的渐进阶。

1.6 在有序数组中查找

为了阐明前面章节中展现的观点，我们将学习一个熟悉例子。

问题 1.1 有序数组查找

给出包含 n 个有序元素的数组 E，给出值 K，作出索引 index，K=E[index]，或者 K 不在数组中则返回-1。

实际中，K 常常是条目的*关键字*，条目是一个除了*关键字*域还带其他实例域的类，所以更精确的表述应该是 K=E[index].key。为了方便讨论，我们假定条目就是关键字，而且他们是一些数字。

让我们先假装不知道二分查找算法（Binary Search），就像我们第一次可看到这个问题一样。我们将考虑不同的算法，分析最坏和平均行为，最后考虑二分查找，并且通过建立需要进行关键字比较的低限来展示二分查找是最优的。

1.6.1 一些解决方法

显然顺序查找算法（算法 1.1）可以解决问题，但是它没有使用数组元素是有序这一事实。我们能否修改算法使之使用增加的额外信息而减少工作量呢？

第一项改进通过观察可以迅速得到，既然数组是递增的，则一旦找大于 K 的元素，算法就可以结束并返回-1 了。（必须如何修改算法的第 4 行的测试以避免在一次循环中进行两次比较？）这个改变对分析有何影响呢？无疑，修改过的算法对于有些情况更好；对于有些输入迅速的终止了。但是，在最坏情况下结果不变。如果 K 是数组的最后一个元素，或者 K 是最大的元素，则算法还是要进行 n 次比较。

对于平均情况分析，我们必须知道 K 在两个数组元素之*间*的可能性由多大。假设我们定义一个间断（gap）g_i，是 y 的集合，$E[i-1] < y < E[i]$ (i= 1,…..n)。令 g_0 是所有比 E[0]小的元素的集合，g_n 是所有比 E[n-1]大的元素的集合。我们假定，就像我们在例 1.9 中做的，K 在数组中的概率是 q。如果 K 在 E 中，我们假定 K 在所有位置的概率都是一样的（也就是有条件概率 1/n）。如果 K 不在数组中，我们假定所有的间断都是相等的（也就是有条件概率 1/(n+1)）。对于 $0 \le i < n$，需要进行 i+1 次比较来决定 K=E[i]或是 K 在 g_i 中，而且将进行 n 次比较来确定 K 在 g_n 中。我们计算比较的平均次数，有条件的成功 (A_{succ}) 和有条件的失败 (A_{fail})，如下：

$$A_{succ}(n) = \sum_{i=0}^{n-1}\left(\frac{1}{n}\right)(i+1) = \frac{n+1}{2}$$

$$A_{fail}(n) = \sum_{i=0}^{n-1}\left(\frac{1}{n+1}\right)(i+1) + \left(\frac{1}{n+1}\right)n$$

第一个等式展示了 K 在数组中的情况，与例 1.9 中的相同。第二个等式展示了 K 不在数组中的情况。估计和很简单，留到了练习中。就像在例 1.9 中一样，最后的结果通过等式 $A(n) = qA_{succ}(n) + (1-q)A_{fail}(n)$ 计算。A(n)粗略的结果是 n/2，不管 q 是多少。当 q=1/2 时算法 1.1 平均要比较 3n/4 的元素，所以修改后的算法有改进，尽管他的平均行为还是线性的。

Let's try again.我们能否找到一种在最坏情况下也小于 n 次比较的算法呢？假设我们将 K 与数组中每 4 个元素比较。如果匹配，就成功了。如果 K 比进行比较的元素大，即 E[i]，则三个在 E[i]前面的元素就不用比较了。如果 K<E[i]，则 K 在最后比较的两个元素之间。再有几次比较（是多少？）就可以决定 K 的位置了如果 K 在数组中，或者决定 K 不在数组中。算法的细节和分析留给读者去完成，但是很容易看到只有 1/4 的数组元素北检查到了。即使在最坏情况下也只做大约 n/4 次比较。

我们可以使用同样的方法，选择一个大值 j 并且设计一种算法，只需要将 K 与 1/j 的元素进行比较，因此在我们处理数组时允许每次比较消除 j-1 个元素。因而我们在 E 中确定一个可能包含 K 的小段大概需要进行 n/j 次比较。则我们继续用进行 j 次比较就能在小段中找到 K。对于一个固定的 K 算法仍然是线性的，但是如果我们选择 j 为（n/j+j）的最小值，这个值应该是 $\sqrt{n}$。此时总的比较次数仅为 $2\sqrt{n}$。我们突破了线性的屏障。

但是我们还可以做的更好吗？注意我们在定位了小段之后继续使用线性策略。段有大约 j 个元素，然后我们花了 j 次比较来搜索他，这个过程是线性的。但是现在我们知道线性消耗太高了。这建议我们在小段上递归的使用我们的“主策略”，代替线性策略。

著名的二分查找算法的想法就是将“每 1/j 的元素”应用到每一步中，在每次跳过一半的元素。不用选择一个特定的整数 j，不用将 K 与每 1/j 元素比较，我们首先将 K 与数组正中间的元素相比较。这样在一次比较中就排除了一半的元素。

一旦我们决定了那一半可能包含 K，我们递归的应用策略。我们继续比较 K 和剩下段正中间元素比较，直到可能包含 K 的段缩小到 0 或者找到 K。每一次比较之后可能包含 K 的段会减少一半。注意，这是一个普通搜索例程的另一个例子（定义 1.12）。当可能包含 K 的段收缩到 0 时，过程*失败*；如果找到 K 则*成功*；两者都没有发生时*继续*搜索。

这个过程是*分而制之*策略的主要例子，我们将在第 3 和第 4 章讨论这个策略。通过将 K 与正中间元素的比较，将在 n 个排序元素中查找 K 的问题分成了两个子问题（假定正中间的元素不是 K）。通过分析我们将看到分开解决两个子问题比解决原来没有分开的问题要简单（在最坏情况和平均情况下）。事实上，其中一个子问题不需要解决，因为我们已经知道 K 不在数组的那部分中。

算法 1.4 二分查找

输入：E，first，last 和 K，这里 E 是一个在范围 first,…last 之间顺序的数组，K 是待查找的关键字。为了简化问题，我们假定 K 和 E 的元素都是整数，和 first 和 last 一样。

输出：index，如果 K 在 E 中，E[index]=K，如果 K 不在 E 中 index=-1。

```
int binarySeach(iny[] E, int first, int last, int K)
{
1    if(last<first)
2        index=-1;
3    else
     {
4        int mid= (first+last)/2;
5        if(K==E[mid])
6            index=mid;
7        else
         {
7            if(k<E[mid])   // 译注，原文中这一句和 else 在一行
8                index=binarySearch(E,first,mid-1,K);
9            else
10               index=binarySearch(E,mid+1,last,K);
         }
     }
11   return index;
}
```

算法 1.4 正确性的详细证明在介绍了一些必须的材料之后，在 3.5.7 节中作为一个正式例子给出。这种经常使用的不正式的原因在前面讨论过了。

1.6.2 二分查找的最坏情况分析

让我们定义二分查找的大小 n=last-first+1，E 中要查找元素的数量。对二分查找的基本操作的合理原则是 K 与数组元素的一次比较。（这里讨论的“比较”总是意味着一次 K 与数组元素的比较，而不是第一行中 index 的比较。）令 W(n)为查找 n 个元素数组的时要做的比

较的数目。

通常假设比较都是 3 元比较，就像在第 5 和第 7 行进行的。（即使不进行 3 元比较，用二元比较也可以达到相同的边界，见练习 1.42。）因此，W(n)也是调用 binarySearch 的次数，除了在第 2 行没有比较就结束的情况。

Java 语言提示：许多 Java 的许多类，包括 **String**，通过 **Comparable** 接口支持三元比较；用户定义类也可以实现这个接口；参见附录 A。

假设 n>0。算法的任务是从 first 到 last 的 n 个元素中找出 K。他在第 5 行将 K 与 E[mid]比较，$mid = \lfloor (first + last)/2 \rfloor$。在最坏情况下，这些关键字都不相等，到达第 8 行或是第 10 行，取决于到底是左边还是右边的段包含 K。有多少元素在剩下的段中呢？如果 n 是偶数，则有 n/2 个元素在右边的段中，有(n/2)-1 个元素在左边的段中。如果 n 是奇数，两边的段中都有(n-1)/2 个元素。因此，在每次递归之前还有$\lfloor n/2 \rfloor$个元素在剩下的段中。因此保守估计在每次递归调用之后范围北缩小一半。

我们要将 n 除 2 多少次才能使结果刚好大于 1？换句话说，对于$n/2^d \geq 1$成立的 d 的最大值？求解 d：$2^d \leq n$和$d \leq \lg(n)$。因此在递归调用之后我们能做$\lfloor \lg(n) \rfloor$次比较，在递归调用之前还有一次，总共需要做$W(n) = \lfloor \lg(n) \rfloor + 1$次比较。练习 1.5 给出这个表达式更简明的表达式，而且对于 n=0 的情况也能很好的适应，$\lceil \lg(n+1) \rceil$。我们有：

定例 1.14 在最坏情况下二分查找算法需要将 K 与数组的元素进行$W(n) = \lceil \lg(n+1) \rceil$次比较（这里$n \geq 0$，是数组元素的数目）。由于每个函数调用做一次比较，运行时间在Θ(logn)。

在最坏情况下，二分查找平均要比顺序查找做的比较少。

1.6.3 平均情况分析

为了简化分析，我们假定 K 可能出现在数组的任何地方。就像我们在本节开始时看到的，有 2n+1 个位置 K 可能出现：E 中的 n 个位置称为*成功位置*，n+1 个间断成为*失败位置*。对于$0 \leq i < n$，令I_i表示在 K=E[i]时的所有输入。对于$1 \leq i < n$，令I_{n+i}表示 E[i-1]<K<E[i]时的所有输入。I_n和I_{2n}分别表示 K<E[0]和 K>E[n-1]时的输入。令$t(I_i)$为在输入为I_i时算法进行 K 与数组元素比较的数量。表 1.4 展示了 n=25 时 t 的值。显然大多数成功的和所有间断都在最坏情况内（Observe that most successes and all gaps are within one of the worst case, 译者：？难道比最坏情况还要多？）；大多数时候只要进行 4 或者 5 次比较。（对于 n=31 我们将发现大多数成功的和所有的间断都刚好等于最坏情况。）所以我们假设所有成功的位置都是相等的，肯定可以期望比较次数的平均值接近与 lgn。计算平均值时假定每一个位置有

1/51 的概率时比较次数是 223/51，或者大约是 4.37，$\lg 25 \approx 4.65$。

i	$t(I_i)$
0	4
1	5
2	3
3	4
4	5
5	2
6	4
7	5
8	3
9	5
10	4
11	5
12	1

i	$t(I_i)$
13	4
14	5
15	3
16	4
17	5
18	2
19	4
20	5
21	3
22	5
23	4
24	5
间断 25,28,31,38,41,44	4
其他间断	5

表 1.4 n=25 时，二分查找进行比较的数量与 K 位置的关系

既然比较的平均次数依赖于查找成功的概率，我们用 q 表示概率，定义 $A_q(n)$ 为当成功的概率是 q 时比较的平均次数。由条件概率法则我们有

$$A_q(n) = qA_1(n) + (1-q)A_0(n)$$

因此我们可以在特定情况的分别解出 $A_1(n)$（正中间成功）和 $A_0(n)$（正中间失败），合并他们就可以得到一个适用于所有 q 的结果。注意 A_1 等同 A_{succ}， A_0 等同于 A_{fail}，后者是在顺序查中使用的术语。

我们将推导 $A_0(n)$ 和 $A_1(n)$ 的近似公式，给出下列假设：

1. 所有成功位置都近似相等：$\Pr(I_i \mid succ) = 1/n$ 对于所有的 $1 \le i \le n$

2. $n = 2^k - 1$ 对于所有的整数 $k \ge 0$

后一个假设是为了简化分析。所有 n 的结果和我们将得到的结果非常接近。

至于让 $n = 2^k - 1$ 是想让每次失败的查找都刚好使用 k 次比较，不管 K 在那个间断中。因此 $A_0(n) = \lg(n+1)$。

分析成功查找平均行为的关键是做一个转换，即将“对于一个特定的输入 I_i 要进行多

少次比较？”转换成“多少输入需要做 t 次比较？”。对于$1 \le t \le k$，令s_t是使算法要做 t 次比较的输入数量。

例如对于 n=25，$s_3 = 4$，因为有 4 个输入会做 3 次比较I_2，I_8，I_{15}和I_{21}。

易的$s_1 = 1 = 2^0$，$s_2 = 2 = 2^1$，$s_3 = 4 = 2^2$，以及$s_t = 2^{t-1}$。既然每一个输入都有 1/n 的概率，算法做 t 次比较的概率是s_t / n，由等式（1.12）平均是

$$A_1(n) = \sum_{t=1}^{k} t\left(\frac{s_t}{n}\right) = \frac{1}{n}\sum_{n=1}^{k} t2^{k-1} = \frac{(k-1)2^k + 1}{n}。$$

（如果我们没有假定$n = 2^k - 1$，s_k的值将不是这种样式，有些失败情况将仅有 k-1 次比较，就像在表 1.4 中看到的。）现在，既然$n+1 = 2^k$，

$$A_1(n) = \frac{(k-1)(n+1)+1}{n} = \lg(n+1) - 1 + \mathrm{O}\left(\frac{\log n}{n}\right)$$

就像提及的，$A_0(n) = \lg(n+1)$是 K 不在数组中的情况。因而我们总结出下列的定例：

定例 1.5 二分查找（算法 1.4）对于有 n 个元素的数组要做大约 lg(n+1)-q 次比较，这里 q 是查找成功的概率，所有的成功位置都是相等的。

1.6.4 优化

在前一节里，我们将一个Θ(n)算法改进成了一个$\Theta(\sqrt{n})$算法，进而到了Θ(logn)。还可以进一步改进吗？即使我们不能在提高渐进阶，我们能够改进常数因子吗？底限分析的规则告诉我们这两个问题可能都是否定的。一个严格的底限，其中一个匹配我们算法的上限，保证我们不能在找到改进的方法。

我们将展示二分查找算法是这类只能对数组元素做比较的算法中最优的。我们将通过检查这类查找算法的*决策树*，建立一个需要比较次数的底限（原注：我们假设读者知晓二叉树的术语，包括*根*，*叶子*和*路径*；如果没有在继续之前请提前看 2.3.3 节）。令 **A** 是这样一个算法。**A** 的决策树和一个给定规模为 n 的输入是一个二叉树，二叉树的节点用 0 到 n-1 之间的数标记，遵循下面的规则：

1. 树的根由算法 A 比较的第一个数组元素的索引标记。
2. 假设一个特定的节点标记为 i。当算法比较 K 之后作左子树是 K<E[i]的情况，，右子树是 K>E[i]的情况。如果算法比较之后终止了而且发现 K<E[i]（或者 K>E[i]），则这个节点没有左（或右）子树。对于 K=E[i]的情况没有分支。一个合理的算法在这种情况下不再做比较了。

可以用这种决策树模型表示的算法是很广泛的；包括顺序查找和本节开始讨论的变长查

找。（注意允许算法在数组中两个关键字，但是这不提供任何信息，因为数组已经是排序的了，所以我们不为决策树增加这种节点。）图 1.8 展示了 n=10 时二分查找算法的的决策树。

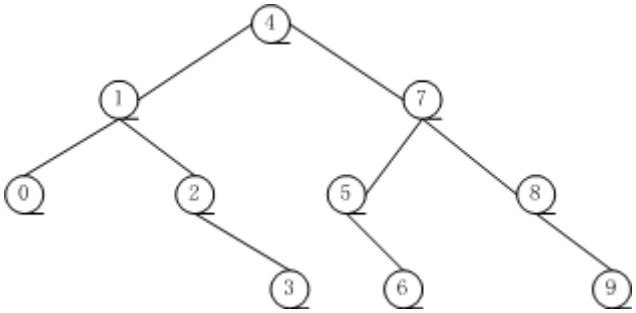


图 1.8 n=10 时二分查找算法的决策树

给出特定的输入，算法 **A** 将执行决策树中从根开始的一条路径上的所有比较。执行关键字比较的次数就是路径上节点的个数。最坏情况下执行比较的次数就是最长的一条路径的节点数；称这个数为 p。假定决策树有 N 个节点。每一个节点最多有两个孩子，则在给定深度的（counting each edge as one）节点的数目大约是前一级深度的节点数目的两倍。既然对于所有节点到根的长度是 p-1，我们有

$$N \le 1+2+4+\ldots+2^{p-1}$$

由等式 1.8 的右边的部分是 2^p-1，所以我们有 $2^p \ge (N+1)$。

我们有了 p 和 N 的关系，但是我们想要得到 p 到 n 的关系，n 是数组中要查找的元素的数目。关键的问题是如果算法 **A** 在所有情况下都正确的话，$N \ge n$。我们需要将决策树中一些节点用 i 标记，每一个 i 从 0 到 n-1。

用反证法，假设有没有被标记 i 的节点，有些 i 在 0 到 n-1 的范围内。我们可以安排两个输入数组 E1 和 E2，E1[i]=K，但 E2[i]=K'>K。对于所有小于 i 的索引 j，我们令 E1[j]=E2[j]，使用一些小于 K 的有序键值；对于所有比 i 大的索引 j，我们令 E1[j]=E2[j]，使用一些大于 K'的有序键值。既然决策树中有没有标记为 i 的节点，算法 **A** 永远不会将 K 与 E1[i]或 E2[i]相比较。既然两个输入其他的元素都是一样的，算法对于两个输入的行为也应该是一样的，应该给出两个一样的输出。如果 A 对于其中至少一个输入给出错误的结果，她就不是一个正确的算法。我们得出结论决策树至少有 n 个节点。

因为 $2^p \ge (N+1) \ge (n+1)$，这里 p 是决策树树上最长路径的比较次数。现在我们取对数，得到 $p \ge \lg(n+1)$。既然 A 是这一类算法的一个抽象算法，我们证明下列定例。

定例 1.16 任何在 n 个元素数组中查找 K（通过将 K 与数组元素比较）对于任何输入至少做 $\lceil \lg(n+1) \rceil$ 次比较

推论 1.17 既然算法 1.4 在最坏情况下做 $\lceil \lg(n+1) \rceil$ 次比较，他是最优的。

第二章 数据抽象和基本数据结构

2.1 概述

数据抽象是允许我们将注意力放在数据结构的重要属性上的技术，而不管那些不重要的未指明的概念。一个*抽象数据类型*（ADT）由申明的一系列数据结构和加在上面的一套操作数据的操作组成。ADT 的 c*lient*，或者叫用户，调用这些操作来创建、销毁、处理、询问（interrogate）抽象数据类型的*对象*（或叫*实例*）。对于这一点，一个 *client* 是一些在 ADT 外面定义的函数或过程。

本章描述了一种可以用来指明所需的抽象数据类型行为的技术，展示了如何将这种技术使用到广泛使用的一些数据结构上，也回顾了在后面的算法开发中一些标准数据结构的重要属性将其什么作用。

规范技术基于 David Parnas 的先驱工作（参见本章后面的注意和参考）。关键的思想叫*信息隐藏*，或叫*数据封装*。ADT 模块含有私有数据，外部模块只能用定义好的一些方法来存取这些私有数据。Parnas 的目标是提供一种软件技术，使得大型工程各部分都相互独立，但仍然正确的一起工作。

在算法设计和分析里面，ADT 还有其他的重要规则。主要设计可以使用 ADT 操作，而不管操作是怎么实现的。我们可以在这个层次的算法设计完之后，再着手分析 ADT 操作在算法中使用了多少次。有了这些信息，我们就可以优化 ADT 的实现，让执行最频繁的操作开销最小。

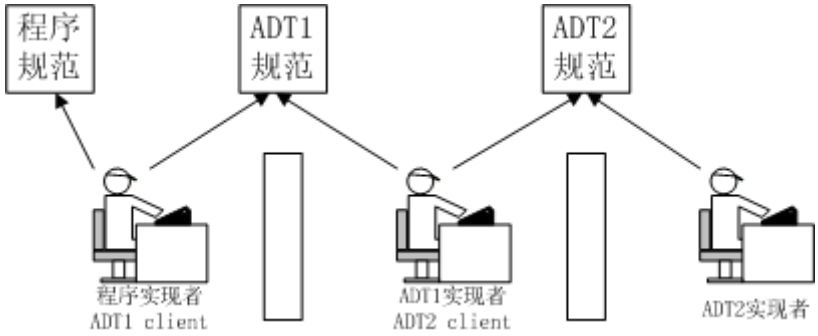


图 2.1 ADT 规范提供了 client 和实现者之间的界面。在这个例子中实现 ADT1 时使用了 ADT2

换句话说，我们可以只使用 ADT 的逻辑属性就能考察一个算法的*正确性*，完全独立于实现。但是*性能分析*倚赖于实现。用 ADT 涉及使得我们可以分开这两个概念。

一种程序设计语言*支持*数据抽象的程度在于他是否允许程序员限制 client 对抽象数据类型的存取；存取只能通过定义的方法，以及 ADT 类的 public 部分。有 private 数据叫*数据封装*或叫*信息隐藏*。这提供给程序员一个工具，程序员可以确保 ADT 对象保持某种*恒定*的状态。就是说，如果 client 唯一的存取 ADT 实例的方法是通过一套比较小的由 ADT 的接口定义的操作，则实现操作的程序员就可以（至少在理论上）确保数据结构各个部分之间的关系符合 ADT 的规范。图 2.1 展示了这种情况。这些考虑解释了为什么在软件工程中 ADT 的重要。

我们选择 Java 作为实现算法的语言，主要是因为他简单，自然的支持数据抽象。在 Java 中 ADT 作为一个*类*（但是，不是每一个类都是 ADT；例如 1.2.2 小节的例子）程序可以创建*类*的*对象*；这些对象是一个抽象数据类型的简单元素。

2.2 ADT 规格和设计技术

ADT 的*规范*描述了操作有什么行为，这个信息对 ADT 的 client 十分重要。就是说，规范应该避免引用私有实例域，因为 client 不知道私有实例域的存在。规范描述了 ADT 公有部分——通常是操作和常量——之间的*逻辑关系*。（本章后面小节的例子中，将验证这个普遍性规律。）ADT 操作（函数或过程）在 Java 术语叫“方法”。

这样设计 ADT 的一个显著的优点是，client 可以在仅仅知道 ADT 规范的情况下开发一个逻辑概念算法，client 可以不管 ADT 的具体实现（甚至不管实现的语言）。这也是本书使用 ADT 方法的主要动机。

2.2.1 ADT 设计规范

规范可以分为*先决条件*和*事后条件*。一个特定操作的先决条件是当操作被调用时假设为真的命题。（译者：即如果该命题不为真，则操作产生不可预知的结果，比如假设传递进来的指针是一个合法的。）如果操作有参数，先决条件一个重要的功能是验证参数（原文：If the operation has parameters it is important that the preconditions be stated in terms of these parameters name for clarity.）。在调用 ADT 的任何方法（静态方法、函数、过程）之前满足先决条件是 client 的责任。一个特定方法的*事后条件*是 client 在调用完方法返回之后认为已经为真的命题。(If the operation has parameters it is important that the preconditions be stated in terms of these parameters name)。另外，事后条件也叫做方法的*目标*（objectives）。

Java 提供了一种特殊的用于 class 文档的注释格式，包含方法的先决条件和事后条件。以“/**”开始的注释开始了一个 **javadoc** 注释。我们在文本中使用 **javadoc** 注释约定来作为一个信号，表示这块注释表示过程或一块代码的*规范*，否则注释是实现方式（即程序代码）的说明。

What Goes into an ADT

对于我们使用 ADT 的目的来说，ADT 是一组相关的过程和函数，他们的规范相互作用提供了一种特定的功能。我们采用最简单的观点，即只在 ADT 中包含必要的操作；这些操作“需要知道”对象是如何实现的（原文：We adopt a minimalist view, by including only the necessary operations in the ADT itself;there are the operations that “need to know “ how the object are implemented.）。因而 ADT 不是一个方便的函数库；如果需要的话，一个库可以作为一个附加的类提供。

必要的操作客分为三个部分，*构造函数*，*存取函数*和*处理函数*。销毁函数，销毁对象并释放对象占用存储空间的函数，不是必须得因为 Java 自动进行“垃圾收集”。垃圾收集定位不引用的对象，回收他们的空间。

定义 2.1 ADT 操作的类型

ADT 操作的三种类型如下：

构造函数	创建新对象并返回对象的引用
存取函数	返回对象的信息，但不改变他。

处理函数　　修改对象，但是不返回信息。

因而，在对象创建之后，一个操作可以修改对象，或者可以返回对象的信息，但是不能同时进行两项操作。

注意 ADT 的构造函数不是 Java 中的构造函数，和其他类型的 ADT 操作一样，他是独立于程序设计语言的。在 Java 中构造函数通过 **new** 关键字来调用，他使用的语法和其他函数一样（非静态）。

因为前面提到的规则，存取函数不能修改对象的任何状态，ADT 规范可能经常以一种特殊的方式组织。对于存取函数，一般不需要给出他的事后条件。此外，标注处理函数和构造函数的规范时，他们的效果必须尽量使用 ADT 存取函数的术语来描述。有时规范需要标记许多操作的组合效果。乍一看，从 ADT 构造函数和处理函数的事后条件找出一个存取函数的行为，似乎是不符合逻辑的。然而，如果我们从总体上将存取函数看作一种对象的普通值，则可以得到一个比较好的印象：无论什么时候一个操作初始化或改变一个对象的状态，操作的事后条件必须告诉我们（那一种都是相关的）对象新的一般值。（译者：这段话理解不了。 Because of our rule that access functions do not modify the state of any objects, ADT specifications can usually be organized in special way. It is normally unnecessary to give postconditions for access function. Moreover, when stating the specifications of manipulation procedures of the ADT constructors, their effects should be described in terms of the access functions of the ADT, as far as possible. Sometimes the specification needs to state the combined effect of several operations. It may seem illogical at first to look at the postconditions of the ADT constructors or manipulation procedure to find out what an access function “does.” However, if we view the access functions collectively as a sort of generalized “value” of an object, then it makes good sense: Whenever an operation initializes or changes the state of an object, the postcondition of that operation should tell us (whatever is relevant) about the new generalized “value”of the object. ）

在选择的一套 ADT 的操作里，比较重要的一点是保证有充足的存取函数来检查所有操作的先决条件。这提供给 client 一套检查操作的先决条件是否满足的能力。

对于实际的软件开发，有一个包含经常使用的 ADT 操作的库是十分方便的。库与 ADT 的不同在于库里面的操作可以用 ADT 操作来实现；他们不需要知道对象是如何实现的“look under the hood”。（当然，在有些时候，“look under the hood”可以提供一个更快版本的库函数。）

2.2.2 ADT 设计技术

一些我们在开发算法中用到的重要 ADT 的定义，将在本章后面的小节中给出。读者也可以从例子总看到如何使用 Java 来定义和实现这些 ADT。也可以轻易的领会到如何用读者熟悉的其他程序设计语言来实现他们。

对于简单，非常标准的 ADT，像链表，树，堆栈，和 FIFO 队列，设计中使用的 ADT 可以最后才给出实现。有时其他 ADT 是标准 ADT 的 client，直接将标准 ADT 作为模块使用。

对于许多复杂非标准的 ADT，比如字典，优先级队列，和联合查找（Union-Find），ADT 可以在设计使用他们地逻辑优点（比如分析正确性简单），但是在最后实现的时候他们应该可以被方便的“展开”，实现算法使用到的特定情况。

本章剩下的部分展示了几种标准的数据结构和他们那相关联的抽象数据类型，从简单到

复杂。各种要给出的规范技术以他们出现的顺序编址。（Various issues concerning specification techniques are addressed as they arise along the way.）在本章中，除了链表，其他 ADT 实现的方法都在练习中讨论。我们包括了一些简单 Java 代码的链表的例子，作为引出其他解决方案的砖头。一般情况，实现都在 ADT 的算法中讨论，所以实现可以简明的是对算法模式的使用（In general, implementations are discussed in the algorithms that the ADTs, so the implementation can be tailored to the usage pattern of that algorithm.）。
（译者：前面几段中好多都不能理解，小弟我实在比较笨）

2.3 基本 ADT－列表和树

抽象数据类型列表和数是简单的，但是用途广泛的，他们的操作都能很容易的在常数时间内实现。我们将为这些 ADT 指定一个构造器和一些存取函数，但是没有处理函数。缺少处理函数使得规范非常简单。省略处理函数的另一个原因在 2.3.2 节中解释。列表和树是最适合用递归来定义的。

2.3.1 递归 ADT

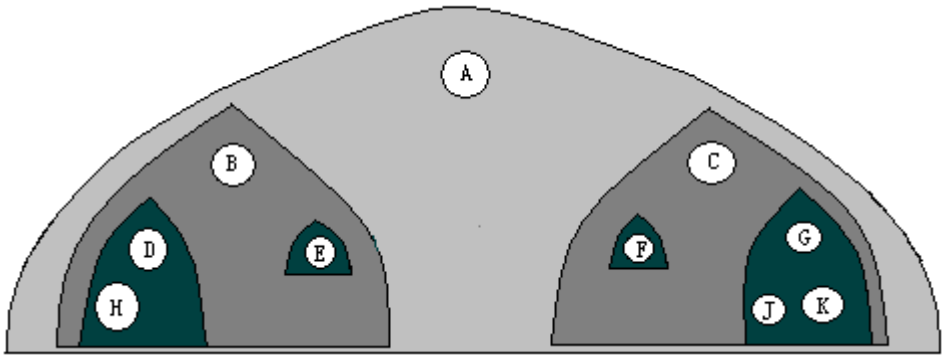


图 2.2 递归 ADT 中的对象必须是结构可以到达的所有部分，而不是只能直接存取得那一部分元素。

如果 ADT 的任何一个存取函数返回的都是自己的同类，则这个 ADT 是递归的。换句话说，对象的有些部分（即存取函数返回的部分）和对象本身是同一类型。此时，ADT 常有一个拷贝构造函数，即构造函数的参数只有一个且类型为 ADT 的类型。这样的 ADT 必须还有一个非递归的构造函数。但是，“非递归构造函数”常常简化成一个常量（可以认为是一个*没有*参数的函数）。列表和树是最适合用递归来定义的公用数据结构。我们将在 2.3.2 节到 2.3.5 节看到他们的规范非常的简单和明了。

最好的理解递归数据结构类型对象的方法是将他看作一个结构，这个结构不仅包括立即可存取得域，还包括只能通过 ADT 存取函数间接存取的域，存取函数返回一个和 ADT 类型一样的对象。例如，在图 2.2 中，最好的理解根为 A 的二叉树的方式是将他看作整个阴影的，尽管可以直接存取元素的只有根 A。

2.3.2 表 ADT

表是计算机科学中的基本数据结构，在理论上和实践上都有重要的意义。许多算法以他开发，尽管可能叫做数组，列表是主要或者唯一数据结构中的高效率版本。程序设计语言 Lisp 是原有的将表作为唯一数据结构的程序设计语言。Lisp 是“list processing”（表处理）的缩写。许多其他程序设计语言，ML 和 Prolog 将表合并成了一种内建特性。这里说得表 ADT 类似于这些程序设计语言提供的 list，操作的名字采用 Common Lisp 的名字。

本文中的术语*表*总是指数据结构内容中叫做*链表 linked list* 的东西。（对于一般的不指定数据结构的有序集合，我们使用术语*序列*。）短的术语*表*是更合适的 ADT 的名字，因为没有术语“link”出现在 ADT 的规范中；如果实现中使用了“link”， List 的 clients 不关心这个事实。

算法，特别是基于图的算法最常用到的表的类型就是整数表。因此，本节中使用这种表做说明。

> *Java 语法提示*：有经验的 Java 用户将尝试定义 IntList 以及其他特定元素的 list，作为一个非常通用 List 的子类。我们不需要这个做法，因为当元素是原生类型时它带来很大的复杂性，而且它使得在继续后面的章节前必须清楚的了解继承。这个主题于学习算法的关系不大。有很多关于 Java 语言的文章深入讨论这个问题。

IntListADT 的规范在插图 2.3 中展示。就像标题中指出的，将表转换成其他类型是简单明了的。这些注释适用于代码，就像规范语句一样。在多个类中使用 cons、first、rest 和 nil 不会引起混乱，因为语言要求用 IntList.cons 来访问在 IntList 类中的版本。

过程头，套在阴影中的部分，展示了 Java 或 C 语法中函数或过程的类型签名（申明）。每一个参数都以其类型开始。因此 cons 的第一个参数是 int，第二个参数是 IntList。在过程名字之前出现的类型或类的名字是返回类型。

规范其他的部分就是先决条件和事后条件。不需要将参数的类型必须符合规定放到先决条件中去，因为这已经在原型中给出。于 2.2.1 节中的方法学一致，存取函数 first 和 rest 的行为，在 cons 的事后条件下描述。

回想一下 List ADT 的简单性是有价值的。当你停止考虑这个问题，这甚至是有趣的，每一个可计算的函数都可以使用表作为唯一的数据结构。（译者：图灵机的模型）有一个常量表示空的表，有一个函数（它的标准名字叫 cons，所以我们采用这个名字）通过将一个新元素放到原有表（这个表可能是空表）的前面来构造表。其他函数简单的返回表（非空的）的信息。第一个元素是什么？表中其他元素是什么？很显然表的所有操作都能在常数时间内完成。（我们假定为新对象分配内存只需要常数时间，这是一个公共假设。）

IntList cons(**int** newElement, IntList oldList)

先决条件：无

事后条件：如果 x= cons(newElement, oldList)，则：

1. x 引用一个新创建的对象；
2. $x \neq nil$
3. first(x)= newElement;
4. rest(x)=oldList

int first(IntList aList)

先决条件：$aList \neq nil$

IntList rest(IntList aList)

先决条件：*aList* ≠ *nil*

IntList nil

表示空表的常量。

图 2.3 IntList ADT 的规范。函数 cons 是构造函数；first 和 rest 是存取函数。List ADT 是类似的，只是所有的 int 替换成 Object，所有的 IntList 替换成 List。其他类型元素的转换类似。

不需要改变 ADT *client* 的代码，IntList 的规范可以以许多方式实现。插图 2.4 展示了一种典型的（也是最小的）实现。注意这个实现没有对 first 和 rest 的先决条件做安全的测试。检查任何要调用函数先决条件的安全性是调用者的责任。

为了软件工程的需要我们可能想创建一个叫 IntListLib 的类（这个类没有指定构造函数。）这个库中可能需要包括 length、copy、equals、reverse、sum、max 和 min。

> *Java 语言提示*：从调试的角度来看，包含 Java *error* 和 *exception* 特性是很有帮助的，但是这使得写一整套 ADT 和 ADT 的 clients 的规范变得复杂。我们一直采用的文字接近于解决问题的精练算法代码。我们提醒读者，将经常建议嵌入软件工程方面的考虑。

部分重建和非破坏性操作（Nondestructive Operations）

认真的读者可能想知道在 IntList ADT 的制度下我们如何*修改*一个表。答案很简单：我们不修改！这里没有*处理过程*。这个 ADT 被称为非破坏性的原因之一就是：一旦一个对象被创建，它就不能被修改。（术语*不可修改的（immutable）* 也是这个含义。）为了更新一个 list 这里有三个选择：

1. 在一个面向对象语言中，比如 Java，定义一个 IntList 类的派生类，在其中增加修改的功能（使得派生类是一个破坏性的或可修改）或者
2. 修改 IntList 本身，增加修改的功能（使其是一个*破坏性的*或*可修改*）或者
3. 离开 IntList 本来的定义。为了完成修改，局部重建原来的表，产生一个新表，将表变量重新*引用*新的表而代替原来的。

局部重建的思想在图 2.5 中表示。目标是在图表中顶部标记为 *w* 的对象开始的表中，在已存在的元素 13 和 44 之间插入一个新元素 22。（按前面递归 ADT 的讨论，我们将 *w* 看作整个表，而不仅是第一个元素。）在插入点之前，重建了包含元素 10 和 13 的的部分表，就像在图表的下部展示的一样。当然，为新元素 22 创建一个新的对象。但是新对象 *x′*，包含一个新的 13 的拷贝，*w′* 包含一个新的 10 的拷贝也被创建了。对象 *x* 和 *w* 被完整的保留了

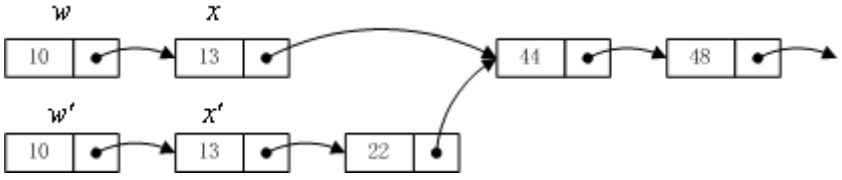


图 2.5 部分重建技术将 22 插入到顺序表 10、13、44、48 中

import java.lang.*;

public class IntList

```
{
    int element;
    IntList next;

    /** 常量 nil 表示空的表.*/
    public static final IntList nil= null;

    /** 先决条件：L 非 nil.
        返回值：L 的第一个元素.*/
    public static int first(IntList L)
    {   return L.element;}

    /** 先决条件：L 非 nil.
        返回值：除了第一个元素以外，L 其他元素构成的表.*/
    public static IntList rest(IntList L)
    {   return L.next;}

    /** 先决条件：无
        *事后条件：令 newL 作为 cons 的返回值.
        *则：newL 引用一个新的对象,newL 非 nil,
        *first(newL)= newElement, rest(newL)= oldList. */
    public static IntList cons(int newElement, Intlist oldList)
    {
        List newL = new IntList();
        newL.elemnet =newElement;
        newL.next =oldList;
        return newL;
    }
```

图 2.4 IntList ADT 的一个 Java 类的典型实现。每一个有一个私有实例域 element 和 next；公共域 nil 由于关键字 **final** 成为了常量；类剩下的部分是方法。**Javadoc** 工具将“/**…*/” 形式的注释和*跟在注释后面*的程序元素关联起来，生成 Web 浏览器格式的文档。

一般情况下，部分重建意味着对于任何对象 x，有一个域是我们要修改的，我们创建一个新的对象 x'，其他域都是一样的值，给需要修改的域一个新的值。x x' 所引用的对象都不需要改变，这就是为什么重建是局部的。但是现在，如果也不能修改的对象 w 引用 x，我们想修改的效果也影响 w，我们需要从 w 创建 w'（除了 w' 引用 x' 而不是 x 以外都是一样的）来进行递归重建。就像我们在下一个例子中看到的，定位 w 没有麻烦，因为我们仍然使用 w 来定位 x，使用 w 来调用函数的地方仍是活的。因此在创建 x' 的函数调用返回之后，我们我们回到了 w 是一个已知量的环境中。

```
/** 先决条件：oldList 是升序的。
*   返回：  升序的新表，由 newElement 和 oldList 其他元素组成
*/
public static IntList insertl(int newElement, IntList oldList)
```

```
{
    IntList newList;
    if(oldList == IntList.nil )
        newList = IntList.cons( newElement, oldList);
    else
    {
        int oldFirst= IntList.first(oldList);
        if(newElement <= oldFirst)
            // newElement 属于 oldList 的前面
            newList= IntList.cons(newElement, oldList);
        else
        {
            IntList oldRest= IntList.rest(oldList);
            IntList newRest= insertl(newElement, oldRest);

            // 局部重建
            newList =IntList.cons(oldFirst, newRest);
        }
    }
    return newList;
}
```

插图 2.6 在有序表中插入整数的函数（或 Java 方法），采用局部重建技术。注意，使用 IntList 类的成员时使用了完全名字。这是必须的因为 insertl 不在类中。

例 2.1 通过局部重建在一个有序表中插入

插图 2.6 展示的 Java 代码通过局部重建的方法在一个已存在的有序表中插入整数。在所有的递归开始之前，我们首先对一个基本事实进行检查：**oldList** 是空的吗？注意，空表是有序的！之后要检查的是另一个基本事实：一个新元素可以简单的插入到 oldList 的前面吗？这两种情况下，oldList 都不需要修改，我们直接用 **cons** 在它前面插入一个新元素。

如果不是上面这两种情况，就要做一个递归函数，其返回一个重建的表（存储在 newRest），新元素在表适当的位置。现在我们的任务是在 newRest 的“前面”包含 oldList。既然我们不能修改对象 oldList，我们通过调用 cons 创建新对象（存储在 newList）来“重建”。注意在这个递归过程中 newList 和 oldList 有相同的第一个元素。但是 rest(newList)与 rest(oldList)不同，rest(newList)在合适的地方包含 newElement。

这个过程是*普通搜索例程*的一个例子（参见定义 1.12）。我们在包含元素的表的前面搜索一个较大的关键字的元素。“失败”事件是遇到空表，因为显然没有更大的元素了。“成功”事件是在测试的表中第一个元素更大。如果这个两个事件都没有发生，我们“继续搜索”剩下的表。每一个没有成功的搜索步骤都会发生一次重建操作。

在调试和证明正确性的方面，局部变量的频繁使用。注意局部变量可以定义在“inner blocks”中而不一定非要在函数的开始。另外注意，整个例子代码中。局部变量在每个函数调用中只赋值一次；赋值一次是习惯，之后又用其他值改变会使得正确性问题变的复杂。这个主题在 3.3 节中做更深的讨论。

考虑插图 2.5 的例子，将 22 插入到包含 10，13，44，48 的表。初始表是 w，10 是第一个元素，x 是剩下的表。既然 22>10，22 必须插入到 x 之中，创建新表 x'。第一次递归调用 IntList.insert。22>13，第二次递归调用开始，这次调用创建并返回一个新对象的引用，这个对象以 22 作为第一个元素。第一个元素是 44 和 48 的对象不需要重建。

回到第一次递归调用，新对象 x' 被创建了，它的 rest 是刚返回的表，这个表的第一个元素是 22 而且 first 从 x 拷贝。这个新对象 x' 被返回到第一次调用（当然是它的引用被返回了），它将 w 当作初始的表。初始调用创建 w'，它的 rest 是 x'，它的 first 从 w 拷贝，返回一个引用 w' 的引用，按顺序插入的操作就完成了。因此当我们回到递归开始位置的时候，对象 x' 和 w' 是从 x 和 w 重建的。

现在插入完成了，整个程序还需要 w 吗？显然，这是一个 List ADT 无法回答的问题。如果回答是“no”，整个程序将（most likely）包含任何 w 的引用，应为所有指向 w 的域或变量都指向了 w'。如果回答是“yes”，仍然有一个重要的 w 的引用在程序的某个地方。众所周知实践中，程序员决定何时释放和回收存储单元是许多隐晦错误的根源，但是一个自动的垃圾收集机制将把程序员解放出来。

Java 语言提示：再次请使用 C++的读者注意，Java 不允许程序员定义一个新版本的“<=”，所以在代码转换到非数值类之前，“<=”运算符必须由一个方法调用代替。Java（从 1.2 版本）开始提供一个 Comparable 接口来用于在一般意义上排序类，具体参见附录 A。

许多其他的 ADT 都可以以 List ADT 为基础实现。多数例子是一般树（2.3.4 节）和堆栈（2.4.1 节）。其他将需要一个 List ADT 作为可更新的列表，比如 in-trees（2.3.5 节）和队列（2.4.2 节）。

2.3.3 二叉树 ADT

我们可以认为二叉树是一个最简单的非线性列表的一般形式；与线性表一个元素只有一个后继元素不同，二叉树中每一个元素有两个分支到不同的元素。在算法中，二叉树有非常多的应用。

二叉树的定义和基本属性

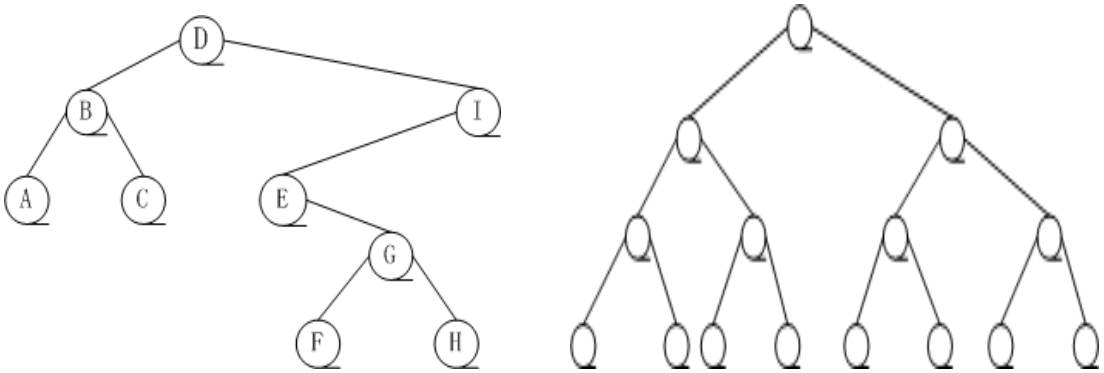
数学上，二叉*树* T 是元素的集合，元素称为节点，节点可以是空或满足下列性质：

1. 有一个独一无二的节点 r 称为*根*。
2. 其余的节点分支为两个子集，L 和 R，两个都是一个二叉树。L 称为 T 的*左子树*，R 称

为 *T*的*右子树*。

二叉树在纸上通常以图2.7那样的图例来表示。如果节点 *v* 是二叉*树 T*的*根*,且节点 *w* 是 *T*的*左（右）*子树的根，则 *w* 叫做 *v* 的*左（右）孩子*而 *v* 叫做 *w* 的*双亲*；在图中从 *v* 到 *w* 右一条直接的边。（边虽然没有箭头，但约定是向下的。）

树节点的度是它的非空子树的个数。度为 0 的节点是叶子。有正数度的节点称为*内节点*（*internal nodes*）。



（a）带节点标签　　（b）完全

图 2.7 二叉树

BinTree buildTree(Object newRoot, BinTree oldT, BinTree oldRT)

事先条件: none.

事后条件:If x=buildTree(newRoot, oldLT,oldRT), then:

1. x 指向新创建的对象；
2. $x \neq nil$；
3. $root(x) = newRoot$;
4. $leftSubtree(x) = oldLT$;
5. $rightSubtree(x) = oldRT$;

Object root(BinTree t)

事先条件: $t \neq nil$;

BinTree leftSubtree(BinTree t)

事先条件: $t \neq nil$

BinTree rightSubtree(BinTree t)

事先条件: $t \neq nil$

BinTree nil

表示空树的常量。

图 2.8 BinTree ADT 的规范。函数 buildTree 是构造函数；root，leftSubtree 和 rightSubtree 是存取函数。类中比 **Object** 一般的节点的规范以类似的方法定义。

根的深度是 0，其他节点的深度是其双亲节点深度+1（原注：小心，有的著作将根的*深度*定义为 1）。一颗*完全二叉树*的所有内节点的度是 2，所有的叶子节点都在同一深度。图 2.7 中右边的树就是完全二叉树。

二叉树的*高度*（有时候也叫做*深度*）是它的叶子最深的深度。任意二叉树节点的*高度*是以它为根的子树的深度。图 2.7（a）中，I 的深度是 1，I 的高度是 3；D 的深度是 0，高度

是 4。

下面的事实在后面经常用到。证明他们很容易，在这里省略证明。

引理 2.1 深度为 d 的二叉树至多有 2^d 个节点。

引理 2.2 高度为 h 的二叉树至多有 $2^{h+1}-1$ 个节点。

引理 2.3 有 n 个节点的二叉树其深度至少是 $\lceil \lg(n+1) \rceil - 1$

图 2.8 给出了 BinTree ADT 的规范。这显然是 2.3.2 节表 ADT 的类推。存取 root 的函数是 List.first 的类推；它存取立即有效的数据。当然，List.rest 在这里分成了两个存取函数，leftSubtree 和 rightSubtree，使得 client 可以存取树剩下的部分。（原注：这里的命名不是标准的，在其他地方使用"leftChild"和"rightChild"。但是讲述 ADT 的章节中，最好是将整个树当成一个对象看待，而不是只考虑根节点。在我们的术语中，左右孩子分别表示左子树和右子树的*根*。）

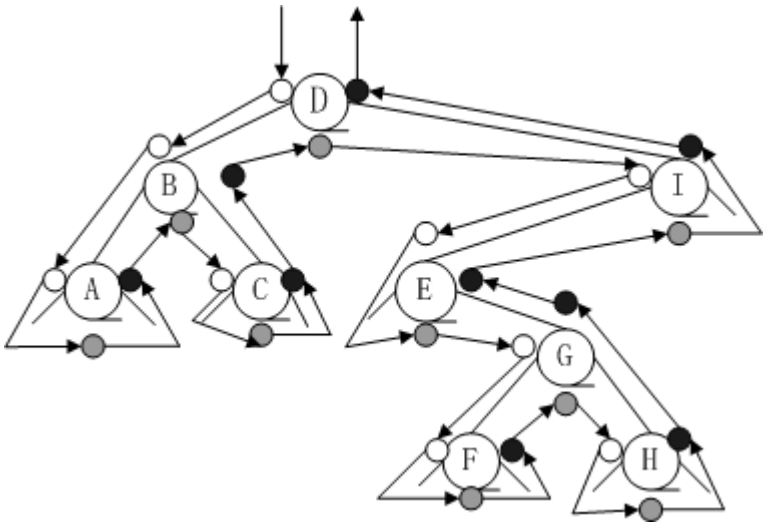


图 2.9 二叉树遍历如同围绕树的旅行

二叉树的遍历

我们可以用图的方式表示二叉树的标准遍历，如同在树上进行从根开始的乘船旅行一样，如图 2.9 所示。我们将每一个节点想像成岛，每一个边是桥，而桥太低以至船无法通过。（为了使我们的想像总是正确，假设在没有子树的地方有桥墩伸出。）小船从根节点开始沿着边航行，访问沿线的节点。节点第一次被访问（白色点），叫做它的 *preorder* time，它第二次被访问到（灰色点，从左孩子返回的那一次）叫做 *inorder* time，最后一次访问到（黑色的点，从右孩子返回的那一次）叫做 *postorder* time。三次遍历可以用下面的递归过程优雅的表示：

```
void traverse(BinTree T)
{    if(T is not empty)
     {    Preorder-process root(T);
          traverse(leftSubtree(T));
          Inorder-process root(T);
```

```
            traverse(rightSubtree(T));
            Postorder-process root(T);
        }
        return;
    }
```

traverse 的返回类型将随应用的不同而变化，它也可能需要额外的参数。前面的过程展示了一个框架。

对于图 2.9 中的二叉树，遍历树的顺序如下：

Preorder(白点): D B A C I E G F H

Inorder(灰点): A B C D E F G H I

Postorder(黑点): A C B F H G E I D

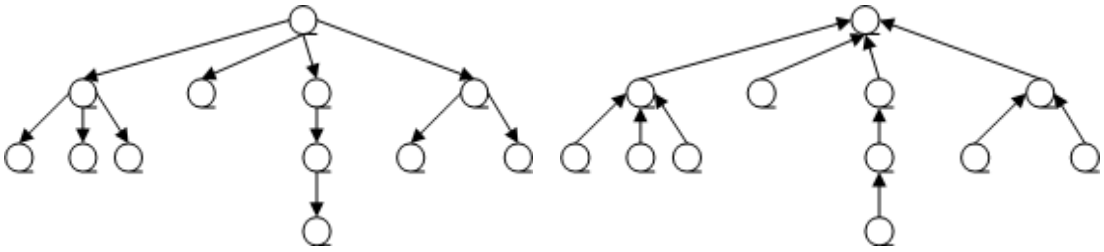
（译注：讲的有点让人发晕，其实前序、中序、后序遍历很容易的。）

2.3.4 树 ADT

*一般*的树（更精确的说，*一般的 out-tree*）是一种带节点和有向边的非空结构，有一个没有入边的节点称为*根节点*，它的其他节点都有且仅有一个入边。更进一步说，就是根到每一个节点都有路径。每一个节点出边的数量没有限制。*森林*是独立树的集合。

一个节点的*子树*由它能访问到的所有节点和它自己组成，节点是它*子树*的根。默认的每一个边都是从双亲到孩子的。如果节点 v 是节点 w 的双亲，则以 w 为根的树称为以 v 为根的树的 *principal（主要）子树*。树的每一个首要子树的节点都比树自己要少。为每一个首要子树命名是不可行的，所以树 ADT 有时比二叉树 ADT 要复杂。

对于一般的树，子树没有必要有顺序，而在二叉树中他们分为“左”和“右”。（如果一般树的子树被认为是有顺序的，这种结构称为顺序树“ordered tree”。）另一个与二叉树的区别是没有空的一般树。



(a) out-tree (b) in-tree

图 2.10 一般的 out-tree 和对应的 in-tree。

如果所有的边都是朝向根而不是离开根，这种结构称为 *in-tree*，所有的边都是从孩子到双亲（图 2.10）。对于这种不同的树有不同数据结构和操作，将在 2.3.5 小节讨论。

树 ADT（最小的操作集合）由图 2.11 的规范描述。与二叉树类似的就省略了。与二叉树不同的是，二叉树的两个子树都有名字，在树中，我们没有定义主要子树的数量，所以 List 是描述这些子树的自然选择。除非树是顺序树，否则是不关心列表的顺序的，子树可以看成是一个集合而不是序列。

第一个主要子树，叫做 t_0，*最左子树*；t_0 的 root 称为*最左孩子*。对于任意主要子树，t_i，序列中的下一个主要子树是 t_{i+1}，如果 t_{i+1} 存在，称为 t_i 的*右表兄子树*。参见图 2.12 例子。尽管这里使用这样的术语，我们重申表中子树的相关顺序对于抽象的树是没有意义的，

仅仅因为我们使用 List 来存放子树。ADT 的构造函数 buildTree 合并一个根节点和树的 list，来创建更大的树。

Tree buildTree(Object newRoot, TreeList oldTrees)

Precondition: 无

Postconditions: If *x*=buildTree(newRoot, oldTrees), then:

1. *x* 引用新传见的对象；
2. root(*x*)=newRoot;
3. subtrees(*x*)=oldTrees;

Object *root(Tree t)*

Precondition: 无

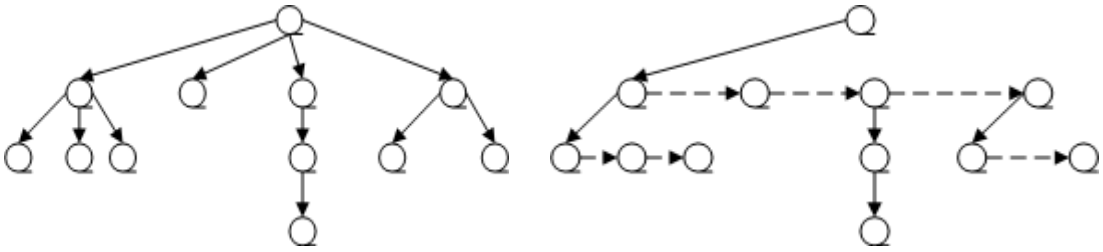
TreeList subtrees(Tree t)

Precondition: 无

TreeList ADT 是一个类似 IntList 的表，基本元素是 Tree。原型如下：

TreeList cons(Tree t, TreeList rSiblings)
Tree first(TreeList siblings)
TreeList rest(TreeList siblings)
TreeList nil

图 2.11 树 ADT 的规范。类中比 **Object** 一般的节点的规范以类似的方法定义。



（a）逻辑结构　　（b）基于表的结构

图 2.12 (a)是一般 out-tree 的逻辑或概念上的结构，（b）则是相应的以 list 存储子树的结构：朝下的实线箭头到最左边的子树，虚线箭头指向右表兄子树。

```
void traverse(Tree T)
{   TreeList remainSubs;
    Preorder-process Tree.root(T);
    remainSubTrees= Tree.subtrees(T);
    while(remainSubtrees  ≠ TreeList.nil)
    {   Tree subtree= TreeList.first(remainSubtrees);
        traverse(subtree);
        Inorder-process Tree.root(T) and subtree;
        remainSubtrees= TreeList.rest(remainSubtrees);
    }
    Postorder-process Tree.root(T);
    return;
}
```

图 2.13 树的遍历框架

Java 语言提示：为了在 Java 中定义这样的内部相关 ADTs 而且具有理想的看见性控制，必须使用 Java 的 *package*。两个文件必须在同一目录中，必须命名为 Tree.java 和 TreeList.java。细节不是困难，但是他们属于本书的范围。将 client 和 ADTs 放在相同的目录下，以避免需要处理包的细节。

树的遍历可以通过对二叉树遍历（参见 2.3.3 小节）进行逻辑扩展来表示，图 2.13 展示了一个框架。子树在一个 **while** 循环中遍历，因为子树的数量是不定的，而且有不能确定数量的中序次数。（这里都使用了类全名，因为牵涉到两个类。）遍历的返回类型将是和具体应用相关的，它也可能需要额外的参数。图 2.13 展示了一个通用的框架。

2.3.5 In-tree ADT

It is a wise father that knows his own child
- Shakespeare, *The Merchant of Venice*

InTreeNode makeNode(**int** d)
Precondition: none.
Postconditions: if x=makeNode(d), then:
1. x 引用一个新创建的对象；
2. nodeData=d;
3. isRoot(x)=**true**;

boolean isRoot(InTreeNode v)
Precondition: none.

InTreeNode parent(InTreeNode v)
Precondition: isRoot(v)=**false**.

int nodeData(InTreeNode v)
Precondition: none.

void setParent(InTreeNode v, InTreeNode p)
Precondition: Node v 不是 p 的祖先。
Postconditions:
1. nodeData(v)保持不变；
2. parent(v)=p;
3. isRoot(v)=**false**;

void setNodeData(InTreeNode v, **int** d)
Precondition: none;
Postconditions:
1. nodeData(v)=d;
2. parent(v)保持不变；
3. isRoot(v)保持不变；

图 2.14 InTreeNode ADT 的规范。函数 makeNode 是构造函数；isRoot，parent，nodeData 是存取函数；setParent 和 setNodeData 是处理函数。节点数据是不同于 **int** 的其他类时，规范类似。

通常，对于树的存取模式都是从根到叶子，一般描述为向下的方向。但是有时需要从叶子向根访问（朝上，比如图 2.10b），而且不需要向下访问。一个 *in-tree* 是只提供这种访问方式的树：节点不“知道”它的孩子。

in-tree 树的一个重要概念是*祖先*（*ancestor*），可以由下面的递归形式定义。

定义 2.2

节点 v 是它自己的祖先。如果 p 是 v 的双亲，则每一个 p 的祖先都是 v 的祖先。祖先的反转是*后代*（*descendant*）。

在 in-tree 中，一个节点可以访问它的祖先，但是不能访问它的后代。不像普通的 *Tree* ADT，在 Tree ADT 中对象是整个树；在 in-tree 的一个对象是一个节点和他的祖先，所以类的名字是 **InTreeNode**，ADT 的构造函数是 **makeNode**。

令 v 是类 InTreeNode 的一个对象。我们遍历树需要那些存取函数呢？首先需要的存取函数是 isRoot(v)，一个布尔函数，当 v 没有双亲时返回 **true**。其次需要的存取函数时 parent(v)，它的一个 Precondition 是 isRoot(v)返回 **false**。换句话说，只要 isRoot(v)是 **true**，则调用 parent(v)是错误的。

图 2.14 包含了 InTreeNode ADT 的规范。当一个节点是由 makeNode 构造的，他是树的唯一节点，所以 isRoot 是 **true**。显然我们需要构造更大树的方法。不像其他简单 ADT，这里使用了一个*处理函数*来获得这个功能。处理函数的返回类型总是 **void**；他们不返回值。这个处理函数是 setParent(v,p)，它将 p 设置成 v 的双亲。有一个 Precondition 是 v 必须不是p 的祖先（否则会创建一个环）。Postconditions 是 isRoot（v）是 **false**，以及 parent(v)返回 p。

实际应用，通常需要在节点保留一种类型数据。既然 in-tree 结构没有其他的分支，我们能定义简单的一对操作，setNodeData 和 nodeData，来允许 client 存储获得这样的数据。尽管节点数据依赖于具体的应用可能有不同的类型，我们按 **int** 定义，因为 **int** 最常见。例如，尽管一个节点不知道它的后代，但是可以保存一个每个节点到底有多少后代的记录（练习 2.12）。

In-tree 常嵌在其他的数据结构中，而不是作为一个独立的 ADT。这几乎是必须的，因为从任何一个 in-tree 的节点出发都不能访问整个树。我们将在图的最小生成树，和最短路径算法中遇到 in-tree，实现联合查找 ADT 时也要用到。联合查找 ADT 在各种算法中用到，包括一种最小生成森林算法。

2.4 堆栈和队列

堆栈和队列说明了抽象数据类型规范下一个级别的复杂性。他们的 ADT 包括操作过程，所以在一类对象可以改变他们的“状态”。现在规范需要描述什么样的状态改变可以发生。然而，这些通用 ADT 上的所有的操作都可以在常数时间内完成，没有太多的难度。有时我们需要保存任务的轨迹，而一个任务可能产生不可预知数量的其他任务，堆栈和队列是一个好的选择。

2.4.1 堆栈 ADT

*堆栈*是一种线性结构，它的删除和插入都必须在被称为*栈顶*的一端。这种更新策略叫做*后进先出*（*LIFO*）。顶部元素是最近插入的，也是唯一能被获得的。Push 一个元素到堆栈意

味着插入这个元素到堆栈。Pop 一个元素意味着删除顶部元素。一个非空堆栈的顶部元素可以通过 top 来访问。现代的做法不再将 top 和 pop 合为一个操作。图 2.15 给出了 Stack ADT 的规范。

Stack create()

Precondition: none.

Postconditions: 如果 s=create()则；

1. s 引用新创建的对象；
2. isEmpty(s)=**true**;

boolean isEmpty(Stack s)

Precondition: none.

Object top(Stack s)

Precondition: isEmpty(s) =**false**.

void push(Stack s, Object e)

Precondition: none.

Postconditions:

1. top(s) =e;
2. isEmpty(s)= **false**;

void pop(Stack s)

Precondition: isEmpty(s) =**false**.

Postconditions: 参见下面的解释。

注解：在 create 之后，任何合法的 push 和 pop 操作（例如累计起来，pop 的数量绝没有 push 多）的序列会产生一个有同样状态的堆栈，这个堆栈也可以由一系列 push 操作产生。为了获得这个 push 序列，重复下列过程：查找紧接着 push 操作的 pop，然后从序列中删除这一对操作。这里使用了*堆栈公理*：紧跟一个 pop 操作的 push 操作对堆栈没有影响。

图 2.15 堆栈 ADT 的规范。构造函数是 create；isEmpty 和 top 是存取函数；push 和 pop 是处理函数。类中比 **Object** 一般的节点的规范以类似的方法定义。

与以前的 ADT 规范不同，不可能显式的指出存取函数 isEmpty 和 top 的返回值。因此，为了提供 push 和 pop 序列的信息我们给出了一段注解。练习 2.13 给出了一个例子。注解间接描述了 pop 的 postcondition。描述操作序列属性（properties）或不变式（invariants）的技术允许 ADT 在逻辑上规范，不用引用 client 不能访问到的实现。复杂的 ADT 常需要这种技术。

大多数需要显式使用堆栈的地方都可以用递归过程来避免使用堆栈，应为”run-time “系统为每一个函数调用的局部变量实现了堆栈。堆栈可以以数组实现或是在 List ADT 的基础上实现。两种方法实现中，堆栈的每一个操作都可以在 $\Theta(1)$ 时间内实现。如果一个可能增长的堆栈的最大容量未知，其增长常用数组成倍增长技术（参见 6.2 节）。这些实现细节都可以由堆栈 ADT 来向 client 隐藏。

2.4.2 队列 ADT

队列是一种所有插入都必须在一端完成的线性结构，这一端称为 *rear* 或 *back*，而所有的删除操作都必须在另一端进行，叫 *front*。只有 front 端的元素可以被访问。这种更新策略叫先进先出(*FIFO*)。插入和删除的操作分别称为 *enqueue* 和 *dequeue*。如果队列为空我们用

存取函数 isEmpty 来测试，如果队列不为空，我们用存取函数 front 来存取它的一个元素。图 2.16 给出队列 ADT 的规范。

和堆栈 ADT 一样不可能给出在调用存取函数 dequeue 之后返回值的显式状态。因此，需要一段注解来描述 enqueue 和 dequeue 的*序列*。参见联系 2.13 给出的例子。

Queue create()

Precondition: 无

Postconditions: If q=create(), then;

1. q 引用一个新创建的对象；
2. isEmpty(q)=**true**;

boolean isEmpty(Queue q)

Precondition: none.

Object front(Queue q)

Precondition: isEmpty(q)=false;

void enqueue(Queue q, Object e)

Precondition: 无

Postconditions: 令|*q*|表示操作前 *q* 的状态。

1. 如果 isEmpty(|*q*|)=**true**; front(q)=e,
2. 如果 isEmpty(|*q*|)=**false**; front(q)= front(|*q*|),
3. isEmpty(q)=false;

void dequeue(Queue q)

Precondition: isEmpty(q)= false;

Postconditions: 参见下面的注解。

注解：在 create 之后，任何合法的 enqueue 和 dequeue 操作（例如累计起来，dequeue 的数量绝没有 enqueue 多）的序列会产生一个有同样状态的堆栈，这个堆栈也可以由一系列 enqueue 操作产生。为了获得这个 enqueue 序列，重复下列过程：找到第一个 enqueue 和第一个 dequeue 删除这对操作。存取函数 front(q)和 isEmpty(q)作用于原队列得到的值，和作用与等价变换之后完全的 enqueue 序列生成的队列得到的值是一样的。

图 2.15 队列 ADT 的规范。构造函数是 create; isEmpty 和 front 是存取函数；enqueue 和 dequeue 是处理函数。类中比 **Object** 一般的节点的规范以类似的方法定义。

队列可以通过数组非常有效率的实现（所有的操作都在 $\Theta(1)$ ）。复杂一点，如果队列的可能增长到的最大容量未知，其增长常用数组成倍增长技术（参见 6.2 节）。这些实现细节都可以由队列 ADT 来向 client 隐藏。

2.5 动态集 ADT

*动态集*是一个集合，其元素在算法使用集合的过程中会改变。通常，这个算法的目的是构建集合本身。但是为了达到目的，在构造集合的过程中，算法需要存取集合以决定如何继续构造工作。动态集 ADT 的一套正确的操作差异很大，依赖于使用它的算法和应用的需要。标准的动态集，例如优先级队列、Disjoint 集合需要联合和查找操作，以及按字典索引。他们在后面的小节描述。

动态集对他们的数据结构有严格的强制要求。本小节介绍的 ADT 不可能在常数事件内实现所有的操作。有所取舍是必须的，对于不同的应用必须有不同的实现才能达到最优的效

率。高效率的查找可能必须用一些非常高级和复杂的实现，我们将在第 6 章接触这些实现方法。

2.5.1 优先级队列 ADT

优先级队列是一种类似 FIFO 队列的结构（2.4.2 节），但是它的元素的顺序与元素的*优先级*相关，而不是元素进队列的时间。元素的优先级（也叫做“key”）是由插入操作支持的一个参数，而不是一种先天属性。我们将假定优先级的类型是 **float**。我们还将假设元素的类型是 **int**，因为这是在大多数最优化应用中的类型。实践中，元素有一个 **int** 类型的标识，以及一个相关联的数据域；这个标识一定不能和“key”向混淆，“key”是优先级域的传统名字。

当每一个元素插入优先级队列时，概念上指它以优先级的顺序插入。一个可以被检查和移除的元素是当前优先级队列中*最重要的*（译注：优先级最高的）元素。Actually what occurs behind the scenes is up to the implementation, as long as outward appearances are consistent with this view.（译注：理解不了^_^）

优先级的概念既可以是最重要的元素有有最小的优先级（消耗观点），也可以是最重要的元素有最大的优先级（效益观点）。在最优化问题中，消耗观点比较流行，而且优先级队列操作的的传统名字也反映了这个观点：**getMin**,**deleteMin**,**decreaseKey**。

优先级队列一个重要的应用是叫做*堆排序*的排序方法（4.8 节），这个名字开始于用*堆*来实现优先级队列。在*堆排序*中，*最大的* key 被认为是最重要的，所以应该叫作 **getMax** 和 **deleteMax**。

与 FIFO 队列不同，优先级队列不能在 $\Theta(1)$ 中实现它的所有操作。必须参考具体算法的需要，或是整体效率来在所有的方法中有选择的实现。这些问题将在各种使用优先级队列的算法中研究。除了*堆排序*（4.8 节），还有一族称为*贪婪算法*的算法，它们都使用优先级队列，包括 Prim 和 Kruskal 的 minimum-spanning-tree 算法（8.2 节和 8.4 节），Dijkstra 的 single-source-shortest-path 算法（8.3 节），以及许多 NP 难题的近似算法（第 13 章）。*贪婪算法*是算法设计的一种主要范例（paradigm）。

现在让我们回到优先级队列 ADT 的规范，在图 2.17 和 2.18 中展示。显然与 FIFO 队列类似。一个主要的区别是删除操作叫 **deleteMin**，如同它的名字，删除优先级（最小 key）最小的元素，而不是最现进队列的元素。

另一个主要的区别是优先级顺序可以用 **decreaseKey** 操作来进行重新排列。但是这个操作和 getPriority 函数可能被*堆排序*算法的实现和其他不需要这种能力的算法的实现所忽略；这两个操作给规范和实现都增加了相当大的复杂性（就像我们在 6.7.1 节中看到的）。我们称不带 decreaseKey 和 getPriority 的 ADT 为*基本*优先级队列。

2.5.2 联合－查找 ADT（Disjoint 集合）

联合－查找 ADT 以它最重要的两个操作命名，但是有时也叫做 Disjoint 集合 ADT。一开始，所有感兴趣的元素都在带构造函数 **create** 的分开的单独的集合中，或者他们增加了单独的处理函数 **makeSet**。**find** 存取函数返回元素的*集合 id*。通过一个 union 操作，两个集合可以连接，连接之后他们不在作为单独的集合存在。因此没有元素可以长期存在一个集合中。一般在实际中，元素都是整数，而且集合 id 是集合中的一些特定元素，叫做 *leader*。

但是，理论上讲，元素可以是任何类型而且集合 id 不需要与元素类型一致。

没有办法“遍历”一个集合中的所有元素。注意与 in-tree 的相似性（2.3.5 小节），在 in-tree 中也没有办法遍历所有元素。事实上，in-tree 可以有效的实现 Union-Find ADT。Union-Find ADT 的实现细节在 6.6 节中描述。UnionFind 的规范在图 2.19 中。

Priority create()
Precondition: 无
Postconditions: If pq=create(), pq 引用一个新创建的对象且 isEmpty(q)=**true**;
boolean isEmpty(PriorityQ pq)
Precondition: none.
Int getMin(PriorityQ pq)
Precondition: isEmpty(pq)=**false**.
void insert(PriorityQ pq, int id, float w)
Precondition:如果实现了 decreaseKey（参见图 2.18），则 id 必须已经在 pq 中了。
Postconditions:标识是 id 优先级是 w 的元素插入。

1. isEmpty(pq)=false;
2. 如果实现了 getPriority(参见图 2.18)，则 getPriority(pq, id)=w。
3. getMin(pq)的值参见下面的注解

void deleteMin(PriorityQ pq)
Precondition: isEmpty(pq)=**false**.
Postconditions:

1. 如果从 create 之后，deleteMin 的数量少于 insert 的数量，则 isEmpty(pq)=**false**，否则是 true。
2. getMin(pq)的值参见下面的注解

注解：将/pq/（在没有操作之前的状态）抽象成对 $((id_1, w_1),(id_2, w_2),...(id_k, w_k))$ 的序列，其中 w_i 代表元素 id_i 的优先级，对以 w_i 的降序排列。则 insert(pq,id,w)高效的按顺序插入(id,w)到队列中，降 pq 扩展到 k+1 个元素。同样的，deleteMin(pq)高效的删除序列/pq/的第一个元素，使得 pq 只有 k-1 个元素。最后 getMin(pq)返回 id_1。

图 2.15 *基本*优先级队列ADT的规范。构造函数是create; isEmpty 和 getMin 是存取函数；insert 和 deleteMin 是处理函数。*完全*优先级队列 ADT 的附加操作在图 2.18 中。其他元素类型不是 **int** 的规类与之类似。

float getProooiority(PriorityQ pq, int id)
Precondition: id 在 pq 中。
void decreaseKey(PriorityQ pq, int id, float w)
Precondition: id 在 pq 中，且 w<getPriority(pq,id)。就是说，新的优先级 w 要求比已经存在的相同元素的优先级要小。
Postconditions:isEmpty(pq)依然是 **false**。getPriority(pq,id)=w。getMin(pq)的值参见注解。
注解：如图 2.17 中的注解，将/pq/（在没有操作之前的状态）抽象成对 $((id_1, w_1),(id_2, w_2),...(id_k, w_k))$ 的序列，对以 w_i 的降序排列。同样的所有的 id_s 是唯一的。

则，decreaseKey（pq,id,w）需要存在$1 \le i \le k$有$id = id_i$，这个函数将高效的从队列/pq/中移除(id_i, w_i)，再以 w 为顺序插入(id_i, w)。最后序列仍然保持 k 个元素。与函数操作之前一样，getMin(pq)返回id_i。

图 2.18 完全优先级队列 ADT 的附加操作。其他操作参看图 2.17。这里 getPriority 是存储函数，decreaseKey 是处理函数。

UnionFind create(int n)
Precondition:无
Postconditions: 如果 sets=create(n)，则 sets 引用一个新创建的对象;对于$1 \le e \le n$有 find(sets,e)=e，对于 e 的其他值是未定义的。
int find(UnionFind sets, int e)
Precondition:集合{e}已经创建了，可以是被 makeSet(sets, e)创建也可以是 create 创建的。
void makeSet(UnionFInd sets, int e)
Precondition:find(sets,e)未定义
Postcondition:find(sets,e)=e;就是说，e 是包含 e 的独立集合的集合 id。
void union(UnionFInd sets, int s, int t)
Precondition:find(sets,e)=s 以及 find(sets,t)=t，就是说 s 和 t 都是集合 id 或是 leader。还有$s \neq t$
Postcondition: 令/sets/引用操作之前的集合。则对于所有的 x 有 find(/sets/,x)=s 或者 find(/sets/,x)=t，我们现在有 find(sets,x)=u。u 的值将是 s 或 t。所有其他的调用再其他操作之前返回同样的值。
图 2.19 UnionFind ADT 的规范。构造函数是 create；find 是存取函数，makeSet 和 union 是处理函数

2.5.3 字典 ADT

字典是一种一般性的相联存储结构。就是说，，每个元素有某种类型的标识和需要存储和检索的信息。信息是与标识相*关联*的。ADT 的名字“字典”来自它与传统字典的相似性，传统字典中每一个词条也有它的标识，定义，读音，以及其他相关信息。但是相似性是有限的，因为在字典 ADT 中元素标识不一定是有序的。一个字典的重要方面是可以在任何时间检索任何信息。

字典的规范在图 2.20 中给出。这些规范是初步的，直到 DictId 的类型确定之后。DictId 是字典条目的*标识*的类型。通常他们是内建类型 **String**，或是原始类型 **int**，或是一个包含多个类型的组织者类。使用字典 ADT 来设计的一个优点是我们可以在使用字典 ADT 的算法设计好了之后再来讨论字典的规范。我们可以创建一个空的字典，再将对(id,info)存入其中。我们可以查询任意 id 是不是在字典内，如果在我们可以检索出里面的信息。对于本书中的应用，都不需要删除操作，但是其他应用，可能会需要一个删除操作。

字典 ADT 在设计动态算法（第 10 章）中非常有用。字典也可以方便的记录外部的名字（通常是输入的字符串），所以程序可以决定之前它是否看到过这个名字，还是第一次看到这个名字。例如，编译器需要记录之前看到过那些数据名字和过程的名字。

Dict create()
Precondition:无
Postconditions: 如果 d=create()，则
1. d 引用一个新创建的对象；
2. 对于任意 id 有 member(d,id)=false。

boolean member(Dict d, DictId id)
Precondition:无

Object retrieve(Dict d, DictId id)
Precondition: member(d)=true; （译注：似乎应该是 member(d,id)=true;）

void store(Dict d,DictId id, Object info)
Precondition: 无
Postconditions:
1. retrieve(d,id)=info;
2. member(d,id)=true;

图 2.20 字典 ADT 的规范，当 DictId 指定为某个具体类型才可以。构造函数是 create; member 和 retrieve 是存取函数；store 是处理函数。类中比 **Object** 一般的信息数据的规范以类似的方法定义。

第三章 递归与归纳

3.1 概述

公认 Massachusetts Instiute of Technology 的 John McCarthy 教授（后就职与 Stanford University）是最早意识到程序设计语言中递归重要性的人。他在设计 *Algol60*（Pascal，PL/I 和 C 的先驱时）贯彻了他的结论，他还开发了 *Lisp*。*Lisp* 引入了递归数据结构，递归过程和函数。本节中的概念都是在 Lisp 中定型的。递归的价值在 20 世纪 70 年代 intense 算法开发期间得到了认同，今天几乎所有流行的预言都支持递归。

递归和归纳是密切相关的。为了使关系更清楚，本章中归纳的表述是形式化的。在字面上讲，一个归纳证明就是一个递归证明。由于结构上的相似性，通过归纳证明递归过程是很简单的。（和前一章一样，在本章中，我们使用“函数”更一般的名字“过程”，Java 的术语是“方法”。）

递归树在 3.7 节引如，提供了一个通用的分析递归过程的时间消耗的 framework。我们将解决许多通用的递归模式，结论以定理的形式总结。

3.2 递归过程

对递归在计算机中到底是如何工作的清晰的理解，对于递归的思考，手动执行递归代码以及分析递归过程的时间消耗都是很有帮助的。我们将首先回顾如何在*活动帧*（*activation frames*）中实现过程调用，以及这个过程是怎么支持递归的。但是，对于大多数涉及递归过

程的设计和分析的活动，我们都想在比活动帧框架要高的层次上来考虑。为了帮助读者达到这个目的，我们引入 Method99，这实际上是一个帮助我们设计递归方案的思考上的技巧。

3.2.1 活动帧框架和递归过程调用

本节对过程调用的实现方式给出一个概要和抽象的描述，这个实现正是递归的工作基础。更详细的描述，请参考本章后面的注释和引用书目。

在运行时（runtime）单个过程调用最基本的存储单元叫作*活动帧*。这个帧为过程的局部变量、实际参数和编译器的“临时变量”提供了存储空间，如果过程有返回值的化还包括返回值的空间。它还包括了其他薄记需要的空间，比如返回的地址，这个地址指示在本过程退出之后程序该执行的指令在那里。因此它提供了对*本次调用唯*一的过程的帧。

从*帧堆栈*（经常简称为“堆栈"）分配存储空间的代码由编译器产生，这部分代码作为过程调用的一部分。这个空间由一个特定的寄存器引用，一般叫作*帧指针*，因此只要过程调用开始，它就知道它的局部变量、输入参数和返回值都存储在什么地方。每一个活动的过程调用都有一个独立的活动帧。一个过程调用从过程进入到过程退出都是有效的。如果发生递归，所有同时存在的递归过程都有不同的帧。当过程调用（无论递归与否）退出，它的活动帧将被自动的释放，于是这块空间就可以被将来的过程调用使用。显示活动帧状态的手工执行代码叫做*活动追踪*（*activation trace*）。

例 3.1 Fibonacci 数列的活动帧

图 3.1 展示了当 main 函数执行 x=fib(3)时，Fibonacci 函数中活动追踪的指针位置。fib 的伪代码如下

```
int fib(int n)
{
    int f, f1,f2;
1:  if(n<2)
2:      f=n;
3:  else
    {
4:      f1=fib(n-1);
5:      f2=(fib(n-2);
6:      f=f1+f2;
    }
7:  return f;
}
```

这段代码申明了一些局部变量，编译器通常为局部变量产生临时的空间，于是我们就可以更细致的研究活动帧。事实上如同许多递归定义的函数一样，fib 函数可以写成一条“怪物”语句：return n<2?n:fib(n-1)+fib(n-2);。但是这种形式让我们追踪活动记录不是很方便。

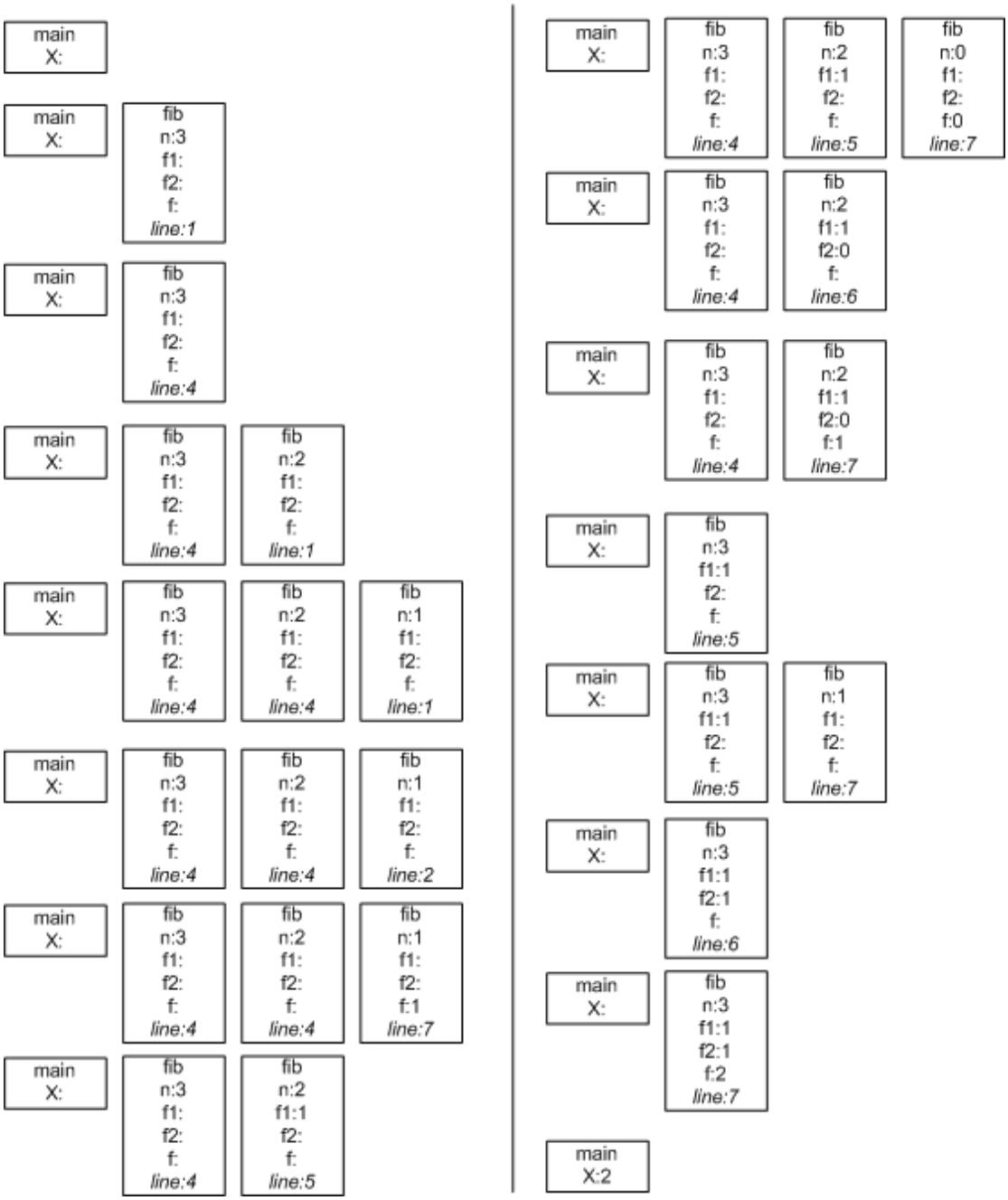


图 3.1 fib 函数的活动堆栈追踪：堆栈顶在右边。堆栈快照的序列排列在左列，再到右列。

图 3.1 中左列顶行是在 fib(3)调用之前帧堆栈，下一行是 fib 进入之后帧堆栈。帧下面显示的行号是将要被执行的，如果帧不是帧堆栈顶部的则行号是执行中的一行。程序的执行流总是在栈顶的活动帧“中”，所以其他帧下面的行号展示了在过程调用结束时执行流将到达的新的位置的堆栈帧。每一个局部变量的值展示在冒号后面。没有值的变量则是还没有初始化。

随后的一行展示了执行到第 4 行，此时另一个函数调用发生。（就是说递归还没有开始。）为了节约空间，下一行忽略了从 1 行到 4 行的过程，仅显示在下一个函数调用之后的第 4 行。这次调用从 2 到 7 行，又一个同样的调用过程；f 接受到值，这次调用将返回。列的最后一行展示了前一次调用返回的是 1；，返回值被存储到 f1，第 5 行将被执行。

右列的顶部展示了当函数调用 fib(0)到达第 7 行的情形；它将返回了。fib(1)函数调用产生的活动帧空间现在已经被复用了。接下来的 3 列展示了调用 fib(2)的完成。它的返回值被存储再调用 fib(3)的活动帧中 f1 内。这个活动帧一直到第 5 行。下一个函数调用再次扩展堆栈。之后堆栈就像前面一样重复工作了。

我们令一个*简单语句*是一个过程中不再调用过程的语句。就像上面提到的 fib 的代码，可以把过程写成每行一个过程调用或是简单语句的线性序列。假设每一个简单语句运行常数时间，而且薄记一个函数调用的环境（设置下一个活动帧等）也是常数时间。因此有：

引理 3.1 在不带 while 或 for 循环，但是可能有递归过程调用的计算过程中，总的计算时间是 $\Theta(C)$，这里 C 是在计算中过程调用的总数（包括作为过程调用的函数）。

但是任意过程的规模 L 必须是自不变的；就是说不会因输入的不同而不同。在任意固定算法中，对于算法中所有的过程都有一个最大的 L。每一次运行算法的总时间，确定是栈顶不同时间活动帧代表的过程调用的时间的总和。还可以合理的假定，由于薄记工作的存在，每一个活动帧都会消耗时间，即使它是立即“返回”的。这些给我们分析递归计算的运行时间的强有力工具。

定理 3.2 在不带 while 或 for 循环但是可能包含递归调用的计算中，总的计算时间是 $\Theta(C)$，这里 C 是在计算过程中调用的总数（包括作为过程调用的函数）。

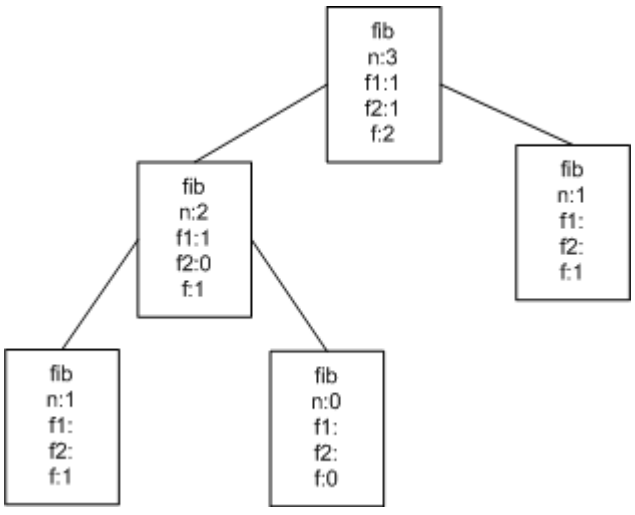


图 3.2 fib(3)的活动树

进一步的，我们可以定义*活动树*（*activation tree*）来作在算法一次运行中所有过程调用

的持久记录。每一个节点代表一个不同的过程调用，记录的位置是在过程将要返回的那一点。根是算法的入口点。每一个节点的父节点是这个节点创建时在帧堆栈顶部的那个节点。每一个节点的子节点从左到右是他们的活动帧创建的顺序。图 3.2 是个例子。

先序遍历活动树将是活动帧创建的顺序，树中节点的数量与执行时间是成比例的。执行期间任何帧堆栈的快照都是对应树中的某条从根开始的线路。（在第 7.4.1 节的深度优先查找时我们回到这种对应性）在 3.7.3 小节中我们会检查活动树和递归等式之间的关系。这对分析低回算法是很有帮助的。

3.2.2 递归的提示－Method 99

对于高级的算法开发，递归是一种必不可少的设计技术。对于递归的深层次的讨论超出了本书的范围，但是这里有个小提示。可以参考本章后面的 Notes 和 References 作进一步阅读。

Identify to youself some "unit of measure" for the size of the problem your function or procedure will work on。然后假定你的 task 只有一个过程叫做 p，p 可以处理 *0－100 的规模*。这就意味着，设计解决方案的时候你必须考虑至少 100 的规模——这就是你的"假想 precondition。"

同样的，假定你可以调用一个给定的*子例程*，叫 p99，他只做你的过程要做的事情，有同样的函数原型，只是他的"fantasy precondition"是规模 0 到 99。允许你使用这个子例程（只要参数符合他的 precondition），你不必写他的代码。

第二个线索是弄清楚你的程序非递归的情况。将非递归的情况弄到尽可能的小。你的过程将总是从测试这种非递归情况开始，也叫做*基本情况*（*base case*）。

最后的约定是决定 p 输入问题的规模刚好是 100"too expensive"。（我们可以使假想规模是 1,000,000,000，但是 method 999,999,999 太长了。）但是将他的规模决定曾 0 或是更小的常量是完全可行的。

现在 Method 99 指出了通过调用 p99 来写 p 的一种方法。（你不需要写 p99，所以不要想他。）当然，如果 p 是一个很简单的情况，则不需要调用 p99。关键的一点是，当 p 检测到问题不能立即解决的时候，他就需要创建一个子问题 p99 以解决之，这满足下列 3 种情况：

1. 子问题的规模比 p 问题的规模小。
2. 子问题的规模比最小的规模要大（这里最小的规模是 0）。
3. 子问题满足 p99 的其他 preconditions（和 p 的 precondition 是一样的。）。

在我们的假想中，子问题保证满足 p99 的规模约束（为什么？）。

如果你可以按这种方式分解问题，你就基本上解决了问题了。只是完成 p 的代码，在需要的时候调用 p99。

让我们实践一下写一个 delete(L,x)，就是从 IntList 中删除元素 x，返回一个新的包含所有 L 的元素但是不包含 x 的 IntList。X 也可能不包含在 L 中。（2.3.2 小节讨论了 IntList ADT，他的 cons 是构造函数，first 和 rest 是存取函数，常量 nil 表示是空表。）

为了适用 Method99， 我们假设我们只需要考虑 list 最多含有 100 个元素的情况，而且我们有 delete99 可以使用。显然，如果我们可以从 L 中消除一个元素（比如第一个），则我们可以让 delete99 去处理 rest（L）。我们不知道 rest（L）中还有多少元素，但是我们采取了精明的实际的态度：如果 L 有 100 个元素，则调用 delete99 就是 ok 的。如果超过 100 个是不会发生的（在我们的假想中）我们仅假设 delete 只能处理 100 个元素或是更少的。

由于第二个线索，我们需要测试最坏的情况。什么是最坏的情况呢？既然允许 x 不在 L

中，那么空表也是可能的。除了空表，还有一种情况需要马上解决而不需要调用 delete99：即 x 是 L 的第一个元素。在这种情况下，我们只需要简单的返回 rest(L)就完成了目标。

现在我们就用 Method99 来实现 delete 过程。

```
IntList delete(IntList L, int x)
{
    IntList newL, fixedL;
    if(L==nil)
        newL=L;
    else
        if(x==first(L))
            newL=rest(L);
        else
            fixedL= delete99(rest(L, x);
            newL= cons(fisrt(L, fixedL);
        return newL;
}
```

Ok，为了完成工作，只需要将子例程的 99 去掉，改成递归调用。

delete 的过程也适合*普通搜索例程*（参见定义 1.12）：如果在没有数据就失败；如果这个数据搜索到了就成功（在上面的例子里是删除了）；否则继续搜索剩下的数据。

3.2.3 递归过程的包装器

经常的一个任务有一部分仅在开始或结束的时候做一次。在这种情况下，你需要将非递归过程独立出来，再调用递归过程。我们称这个非递归过程是递归过程的*包装器*。有时候包装器只是简单的初始化递归过程的参数。例如二分查找（算法 1.4）需要一个包装器使第一次递归有范围。包装器可能简单如下：

```
int orderSearch(int[] E, int n, int k)
{
    return binarySearch(E, 0, n-1, K);
}
```

3.3 什么是证明?

再开始介绍证明之前，让我们对什么是证明做一个回顾。在 1.3.3 小节提到了，逻辑是一个规范化自然语言语句的系统，通过逻辑我们推理的更精确。证明是用逻辑语句推理的*结果*。本节描述详细的证明。实践中人们常忽略很多细节，将细节留给读者去填充；这样的书写更精确的讲叫做*证明框架*。

定理、引理和推论都是可以被证明的语句，他们之间的区别并不是十分明显。一般来说，人们对引理的兴趣不是它本身，引理的重要在于它能帮助证明人们感兴趣的命题，一般称为定理。推论一般是定理的简单结果，但是并不是不重要。不管被证明的语句叫“命题”、“定理”、“引理”、“推论”或是其他什么术语，证明过程是一样的。我们将使用命题作为“一般性”的术语。

证明是符合逻辑规则的语句序列。每一个语句在形式语法层面上讲都是一个 *complete*

sentence：它有一个主语和一个谓词，等等。尽管数学符号提供了一种缩写，但是语句还是代表一个 complete sentence。例如，“x=y+1”代表“x 等于 y+1”，这是一个完整的句子，而“y+1”自己就不是一个句子。

这里精确的将逻辑语句组合成证明的 inference rules 可以无遗漏的列出来，我们将给出比较多的非正式的规则。最重要的规则在 1.3.3 小节给出，等式 1.29 到 1.31。每一个语句都可以从下面的事实得出新的结论

- 众所周知的，且不是你将要证明的（例如，数学恒等式），或
- 你要证明的定理的假定（前提），或
- 在前面的证明中已经建立的语句（中间结论），或是
- inductive hypothesis 的实例，在 3.4.1 小节讨论。

证明的最后一个语句必须是要证明命题的结论。当一个证明分好几种情况时，每一种情况度必须遵循上面的。

每一个语句不仅要给出新的结论，而且必须给出支持结论的事实。直接支持新结论的语句称为新结论 justification。含糊的 justification 是大多数逻辑错误的原因。

定理或命题的格式

你需要证明的命题有两个部分，*假设*（也叫做*前提 premises* 或 *hypotheses*）和*结论*。我们将结论称为*目标语句*。通常，命题有一个型如“对于集合 W 中所有的元素 x”的部分，目标语句是关于 x 的。（可能在语句中有几个类似 x 的变量。）实践中，集合 W（世界的缩写）是一族集合比如自然数、实数或是一族数据结构比如 list、树、图。令要证明的命题有如下形式

$$\forall x \in W\left[A(x) \Rightarrow C(x)\right] \qquad (3.1)$$

这里 $A(x)$ 表示假设，而 $C(x)$ 表示结论或叫做目标语句。符号 $\Rightarrow$ 读作“蕴涵”。方括号仅是为了更可读；他们和园括号一样。在自然语言中命题语句经常是这样的形式“对于所有的 W 中的 x，如果 A(x)，则 C(x)。”用词有各种变化。通常，我们需要将大多数自然语句“揉”成一种向上面这样的标准形式，在我们要证明之前，我们需要知道 x，W，A(x)，C(x)分别代表什么。

例 3.2

一个命题可能表示为

命题 3.3 对于常数 $\alpha < \beta$，$2^{\alpha n} \in o(2^{\beta n})$。

将其规范成等式 3.1 的形式，

命题 3.4 对于所有 $\alpha \in R$，对于所有 $\beta \in R$，如果 α 和 β 是常数且 $\alpha < \beta$，则 $2^{\alpha n} \in o(2^{\beta n})$。

让我们来检查两者的对应关系。显然二元组 (α, β) 对应 x，$R \times R$ 对应 W。引理的前提 A(x)

有 3 条语句：“α 是常数”，“β 是常数”且“$\alpha < \beta$”。结论 C(x)是“$2^{\alpha n} \in o(2^{\beta n})$”。

双列证明格式

我们现在描述一种双列格式用于证明的表示。这种格式的目的是为了让证明中的 justifications 更清晰；右列包含所有的 justifications。每一个证明语句占用代行号的一行。每一个在左边的*新结论*它的 justifications 都在右边。引用前面已经证明的语句是通过它的行号。
例 3.3
例 3.2 的语句将用双列格式来证明。定理和证明都写的非常详细，作为一个证明中如何写 justifications 的范例。我们包括所有的所有的证明者通常会期待读者自己天上的部分。

定理 3.5 对于所有 $\alpha \in R$，对于所有 $\beta \in R$，如果 α 和 β 是常数且 $\alpha < \beta$，则 $2^{\alpha n} \in o(2^{\beta n})$。

证明

语句	Justification
1．首先我们想展示 $\lim_{n\to\infty} \frac{2^{\beta n}}{2^{\alpha n}} = \infty$	
2．$\frac{2^{\beta n}}{2^{\alpha n}} = 2^{(\beta-\alpha)n}$	数学恒等式
3．$\beta - \alpha > 0$ 且都是常数	定理的前提＋数学恒等式
4．$\lim_{n\to\infty} 2^{(\beta-\alpha)n} = \infty$	（3）＋已知的数学定理
5．$\lim_{n\to\infty} \frac{2^{\beta n}}{2^{\alpha n}} = \infty$	（2）＋（4）和带入
6．$2^{\alpha n} \in o(2^{\beta n})$	（5）＋o 集合的定义（定义 1.17）

几点注意事项：

1. 除了组成证明主题的语句，一般还包括一些“路线”或是“计划”语句来说明下面的证明小段的目的，或是要使用的方法论，或是为了完成证明还需要的段落等等。

   第一行就是一个*计划*语句，不包含任何结论。因此它不能被后面的语句引用，它也不需要 justification。它告诉读者要成功证明的一个中间目标。这个目标在第 5 行完成。

2. 最后一行的新结论就是定理的目标语句。

3. 其他的所有行都被当作 justification 被引用了没有任何浪费。

为了让证明完整、流畅，以双列格式写出一些结论的细节是一个很好的风格。通常完成证明框架的细节是有益的，可以让你知道每一个新语句是怎么得出的。

3.4 归纳证明

归纳证明是证明一个关于无限个对象的集合的语句时的一种机制，通常也是唯一的机制。我们这里描述的归纳方法通常叫做*强*归纳。强归纳是用于大多数关于算法和数据结构的证明中最简单的形式。即使有时候不需要强归纳，但是使用使用强归纳也并不比使用他的弱化版本要难。因此我们采用“one-size-fits-all”的原则，总是使用强归纳。

我们将会看到递归和归纳（强归纳）配合的非常完美。从大方面讲，证明是困难的，我们需要尽可能的使得证明可理解，可靠精确。归纳证明和递归过程直接结构的类似性是在证明复杂算法时的重要工具。

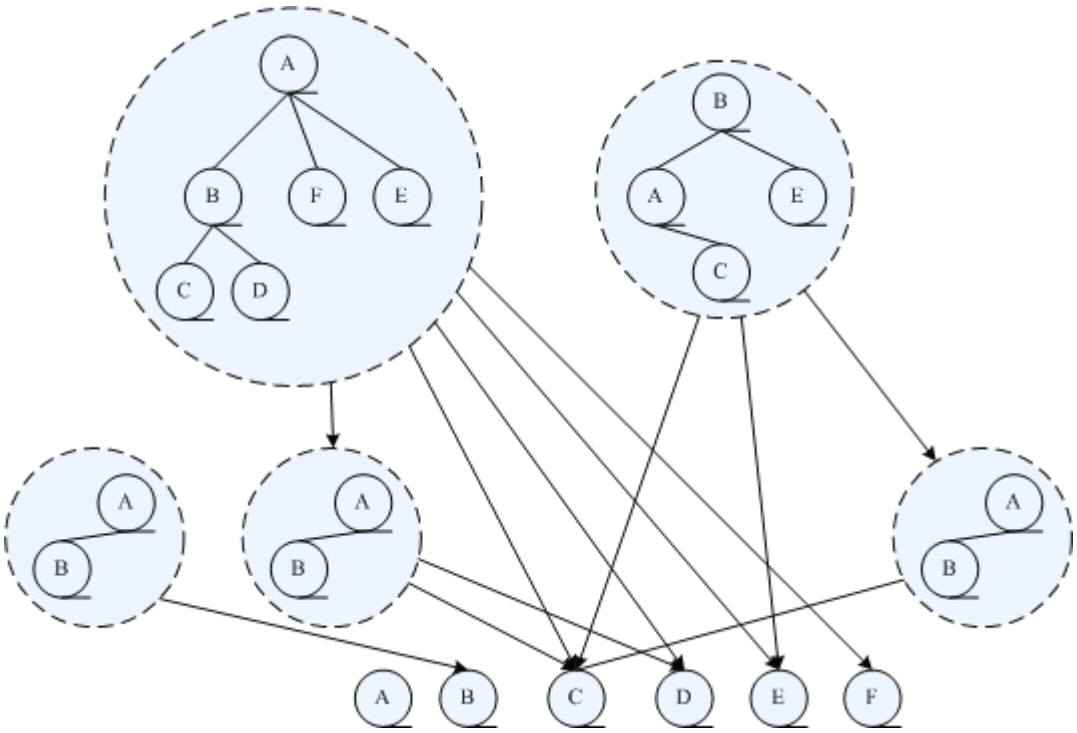


图 3.3 展示的树的集合之间的 *subtree* partial oreder

在大多数情况下归纳法是在自然数集合（非负整数，1.5.1 小节）或是正整数集合上。当然，归纳法在更一般的集合也是有效的，只要他们提供两个属性：

1. 集合是部分有序的；就是说，在某些元素队上定义有顺序关系，但是可能并不是定义在所有的元素对上。
2. 集合中没有无限的降序元素链。

例如，在*所有*整数的集合上就不能使用归纳法。

树就是一个部分有序的集合，经常对树使用归纳法。常用的部分顺序是：$t_1 < t_2$，当 t_1 是 t_2 的 *proper 子树*时（参见图 3.3）。后面我们将看到图有一个类似的部分顺序。在这样的集合上进行的归纳叫*结构化归纳法*。

典型的需要归纳法来证明的定理包括：数学公式的定理、数据结构属性的定理、递归等式的定理、这些经常在分析递归过程的运行时间时用到。3.5 节覆盖了那些需要证明过程完成了它的目标并会正确的终止的定理。3.6 节覆盖了典型的递归等式。

3.4.1 归纳证明的模式

归纳证明的第一件事就是：

There is no such thing as "n+1" in an induction proff.

不幸的是，很多读者学到了另外一面。为什么我们把这个教条放在这里呢？

答案在我们早先给出的动机中——为了将递归和归纳联系起来。我们知道一个递归过程是*创建和解决小的子问题*，再合并子问题来解决主要问题。我们希望我们的递归证明也采用这个模式。对于证明来说，"主要问题"是要求的定理，而"子问题"是要求的定理的小的实例，这些小的实例可以合并起来证明主要的实例。实践中，这些小的实例差不多直接对应递归过程创建的精确的子问题。

所有的递归证明遵循一个通用的模式，我们称之为 *induction schema*。Schema 最关键的部分是 *inductive hypothesis* 的正确引入。首先，我们给出一个证明的例子，之后我们描述一般的 schema，下面是几个例子。

例 3.4

下面定理的证明展示了 induction schema，在例子之后我们将描述其一般形式。以**黑体**表示的步骤几乎会出现在所有归纳证明的细节中。下面给出了证明的细节，我们使用前面提到的双列证明格式。

命题 3.6 对于所有 $n \geq 0$，$\sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{n(n+1)(n+2)}{6}$。

证明

语句	Justification
1. **证明归纳 n**，来给出和的上限	

2. **基本情况是** n=0	
3. 此时等式的两边都等于 0	数学
4. **对于 n>0，假设** $\sum_{i=1}^{k}\frac{i(i+1)}{2}=\frac{k(k+1)(k+2)}{6}$ **对于所有** $k\geq 0$ **且 k<n 成立**	
5. $\sum_{i=1}^{n-1}\frac{i(i+1)}{2}=\frac{(n-1)n(n+1)}{6}$	带入 k=n-1
6. $\sum_{i=1}^{n}\frac{i(i+1)}{2}=\sum_{i=1}^{n-1}\frac{i(i+1)}{2}+\frac{n(n+1)}{2}$	数学
7. $\sum_{i=1}^{n}\frac{i(i+1)}{2}=\frac{(n-1)n(n+1)}{6}+\frac{n(n+1)}{2}$	（5）+（6）
8. $\frac{(n-1)n(n+1)}{6}+\frac{n(n+1)}{2}=\frac{n(n+1)(n+2)}{6}$	数学
9. $\sum_{i=1}^{n}\frac{i(i+1)}{2}=\frac{n(n+1)(n+2)}{6}$	（7）+（8）

一行一行的解释。

1. n 是主要的归纳变脸。注意命题的形式是 $\forall n\in N[A(n)\Rightarrow C(n)]$。

   这里 $A(n)$ 是简单 *true*，而 $C(n)$ 是等式。

2. 一个归纳证明始终有两种主要情况，称为*基本情况*和*归纳情况*。有时是基本情况（s）。

3. *证明*基本情况（s）。

4. 引入辅助变量 k，进行递归假设。注意递归假设是 $A(n) \Rightarrow C(n)$ 的形式，回到这个命题， $A(k)$ 仅仅是 *true*；

   注意 k 的范围包括基本情况。

   注意 k 必须严格小于 n；否则我们的假设就已经包括了我们要证明的。

   递归假设语句的标志着递归情况证明的开始。

5. 使用递归假设。（注意我们将“拉到 n”。）辅助变量 k 在这里假设到了 n-1。既然我们证明 n>0 的情况，k 的这个值满足 $0 \le k < 0$，就像在第 4 行要求的。

   辅助变量 k 可以在其他行假设到范围内的其他值。这是强归纳的一个好处。在这个简单的例子中，恰好不需要将 k 假设到别的值。

6. Justification 是一个标准数学等式，假设读者都知道。

7. Justification 指出前面的两行支持这个新结论，但是不符合用到的*推论规则*，假设读者能指出来。

8. 采用了另一个数学等式。实际中，第 6 行到第 9 行可以合并成一行，假设读者可以指出这些步骤。但是，这样的浓缩可能导致许多错误的“证明”。证明者必须小心哪些精确的步骤需要写出来。

9. 结论恰好是目标语句 $C(n)$。

前面的证明遵循了一种模式，这种模式可以泛化成下面的 schema。注意一般的术语，命题可以是定理、引理、推论或是其他 term without changing the proof process。

定义 3.1 归纳证明模式

首先我们解释下面模式用到的符号。**黑体**的内容重要表示重要的内容。在角括号“<”“>”之间的条目是需要根据证明的命题来替换的。类似的，变量 x 和 y 也是根据命题变化的。他们在集合 W（world）之中。

逻辑语句 $C(x)$ 叫做*目标语句*。逻辑语句 $A(x)$ 叫做*命题假设*（或是假设 s，如果它是一个合取式）。变量 x 叫做主要归纳变量（或简称归纳变量）。变量 y 叫做辅助变量。

一个形如 $\forall x \in W[A(x) \Rightarrow C(x)]$ 的命题的归纳证明由下面的几部分组成。

1. **证明是归纳 x**，<描述 x>。
2. **基本情况是（情况 s 是）**<基本情况>。
3. <*证明*基本情况的目标语句，即带入基本情况，就是 $C(base-case)$>。
4. **对于<X>大于<基本情况>，假设对于所有的 $y \in W$ 且 $y < x$ 都有** $[A(y) \Rightarrow C(y)]$。
5. <*证明*目标语句，$C(x)$，就是命题出现的。>

一个归纳证明有两个主要情况：基本情况合和递归情况。模式的第 2 步定义了基本情况；第 3 步证明基本情况下定理成立。第 4 步定义了递归情况，而且使用了递归假设。第 5 步证明递归情况下定理成立，这通常是证明的主要部分。$C(x)$ 的证明可以由下面部分支持：

1. x 大于<基本情况>；
2. 命题的假设，$A(x)$（不是 $A(y)$）；
3. 任意数量的归纳假设的实例，$[A(y) \Rightarrow C(y)]$，为辅助变量 y 带入严格小于 x 的 W 的元素。

和普通的证明一样，证明中已经证明的结论，外部标识、定理都可以使用。

三个样板语句允许不带 justification，因为他们没有给出任何结论；他们只是简单的解释了证明的模式以及定义了一些符号。他们是

- “证明是归纳 x…”
- “基本情况是…”
- “对于 x> <基本情况>，假设对于所有的 y<x 都有 $[A(y) \Rightarrow C(y)]$。”

后两个语句将证明分成了两种情况：x 是基本情况，x 大于基本情况。两种情况必须覆盖整个 W，即覆盖所有 x 所有的范围。

归纳模式的变体

1. 如果假设 $A(x)$ 并不实际依赖于 x，则递归假设可以简化成：假设 $C(y)$ 对于所有 $y \in W$ 且 y 小于 x 都成立。你必须能解释 why simplification is justified y referring back to

the justifications for proof statements.

2. 如果归纳需要多于一个的有次级情况，比如 Fibonacci 数列等式（1.13），可以有两个或多个基本情况。但是，最好仅引入归纳情况必须要的元素，因为每一个基本情况都需要他自己的证明。
3. 当归纳一个数据结构的时候比如 list、tree、图或其他仅有部分顺序的集合，可以有多个基本情况元素。在图 3.3 中 6 个单个数是基本情况。

3.4.2 归纳证明一个递归过程

下一个例子展示了归纳和递归是怎样一起工作的。我们将要证明的引理是关于一个计算 2-tree 的 external 路径长度的过程的，这在低限分析中很有用（参见 4.7.3 小节）。External 路径长度在一些其他的问题中自然的被提出。首先我们需要一些定义。

定义 3.2 External 节点和 2-tree

二叉树基本类型中除了空树外的基本情况，是只有一个节点的树，这个节点的类型与树剩下节点的类型不同。这种节点叫 *external 节点*。一个由 external 节点组成的树叫*叶子*，它没有任何子树了。其他节点的类型是 internal 节点，它必须有两个孩子。这样的二叉树叫 2-tress，因为每个节点要么有两个孩子，要么没有。

注意，如果我们将 2-tree 中所有的 external 节点替换成空树，则会剩下一个正常的，不严格的二叉树。在 2-tree 中检查一个节点是否是叶子节点通常不需要判断这个节点是否有孩子，因为叶子节点的类型与 internal 节点不一样。练习 3.1 展示了一个 2-tree 的 external 节点必须比 internal 节点要至少多一个。

定义 3.3 External 路径长度

在 2-tree 树 t 中，t 的 *external 路径长度*是根到所有 external 节点路径长度的和。路径的长度是路径的边数。

可选的，2-tree 的 external 路径长度可以递归定义如下：

1. 叶子的 external 路径长度是 0。
2. 令 *t* 是一个非叶子节点，它有左子树 *L* 和右子树 *R*（两者都可能是叶子）。t 的 external 路径长度等于 *L* 的 external 路径长度＋*L* 的 external 节点数＋*R* 的 external 路径长度＋*R* 的 external 节点数。（t 的 external 节点的数量是 L 和 R 的 external 节点数量的和。）

两个定义的等价性是很显然的，因为每一条从 t 的根到 *L* 中的 external 节点的路径都比从 *L* 的根到这个节点的路径要多 1，*R* 是类似的。

```
EplReturn calcEpl( TwoTree t)
{
    EplReturn ansL, ansR;   // 子树返回的
    EplReturn ans =new EplReturn(); // 要返回的
1.  if( t 是叶子)
```

```
2.          ans.epl=0; ans.extNum=1;
3.    else
      {
4.          ansL= calcEpl(leftSubtree(t));
5.          ansR= calcEpl(rightSubtree(t));
6.          ans.epl=ansL.epl+ ans.epl+ ansL.extNum+ ansR.extNum;
7.          ans.extNum =ansL.extNum +ansR.extNum;
      }
8.    return ans;
}
```

图 3.4 计算 2-tree external 路径长度的函数。返回类型 EplReturn 是一个组织者类，用于函数返回 2 个值 epl 和 extNum。

2.3.3 小节的二叉树遍历框架可以简单的扩展成计算 external 路径长度。参数的类型是 **TwoTree**，其定义类似与 **BinTree**，除了最小的树是叶子以外，**BinTree** 的最小树是空树。基本情况也要做相应的修改。函数需要返回两个值，所以我们需要需要定义一个组织者类（参见 1.2.2 小节），类名叫 **EplReturn**，有两个整数成员 **epl** 和 **extNum**，分别表示 external 路径的长度和 external 节点的数目。我们看到函数只是简单的实现了递归定义。我们现在能证明关于 calEpl 的引理.

引理 3.7 令 t 是任意 2-tree。令 epl 和 m 分别是 calcEpl(t)返回的 epl 和 extReturn 的值。则：

1. epl 是 t 的 external 路径的长度。
2. m 是 t 的下 external 节点的数目
3. $epl \geq m \lg(m)$

证明 在证明引理之前，让我们先把引理的语句和我们前面的证明模式关联起来，等式（3.1）。注意为了阅读把引理分成了几个句子。因此 t 是主要的递归变量而 W 是所有 2-tree 的集合。第二个句子代表假设，所以表示 $A(t)$。最后的三个结论组成 $C(t)$。和前面的例子一样，粗体表示了在任何递归证明中的关键部分。

证明是归纳 calcEpl 的参数 t，t with the “subtree” partial order。**基本情况是** t 是叶子。到达 calcEpl 的第 2 行时，epl=0 和 m=1，这符合（1）（2），且 $0 \geq 0$ 也符合（3）。

当 t 不是叶子，**假设引理对于所有 s 都成立，其中** s 是 t 的严格子树。也就是说，如果 epl_s 和 m_s 是 calcEpl(s)返回的值，则 m_s 是 s 的 external 节点的数量，epl_s 是 s 的外部路径的长度，$epl_s \geq m_s \lg(m_s)$。令 L 和 R 分别表示 t 的左右子树。这里必须要求是严格子树，因此可以采用递归假

设。因为 t 是不是叶子，将执行从第 4 行到第 7 行的代码，则有下面的等式

$$epl = epl_L + epl_R + m_L + m_R$$

$$m = m_L + m_R$$

通过递归假设和 exteranl 路径长度的递归定义，*epl* 是 t 是 exteranl 路径长度。t 的每一个 external 节点都是同样是 L 或 R 的 external 节点，所以 m 是 t 的 external 节点的数目。

剩下的是证明 $epl \geq m\lg(m)$。我们注意到（参见练习 3.2）函数 $x\lg(x)$ 在 $x > 0$ 时是凸的，所以我们可以使用引理 1.3。通过归纳假设我们有

$$epl \geq m_L \lg(m_L) + m_R \lg(m_R) + m$$

$$m_L \lg(m_L) + m_R \lg(m_R) \geq 2(\frac{m_L + m_R}{2}) \lg(\frac{m_L + m_R}{2})$$

≥的传递性，

$$epl \geq m(\lg(m) - 1) + m = m\lg(m)$$

推论 3.8 一颗 2-tree 的 external 路径长度 epl 有低限：$epl \geq (n+1)\lg(n+1)$。

证明 每一个有 n 个 internal 节点的 2-tree 都有(n+1)个 external 节点（参见练习 3.1）采用引理 3.7。

经常发生递归过程需要用组织者类返回多个参数的情况，即使只需要一个量，但是返回多个参数有利于简化程序。在这个例子里面，extNum 不需要在查询，然是在递归过程中返回它会极大的简化剩下的计算。对于其他的例子，参见练习 3.13，这个练习要求你设计一个函数来计算树顶点的独立集合的最大权。

3.5 证明过程的正确性

Things should be made as simple as possible—but not simpler.

—Albert Einstein

一般认为证明一个程序的正确性试一个绝望的困难任务，*一般的程序*。然而，证明正确

性对于解决问题和产生正确工作的程序来说是有价值的活动。诀窍是*以已经证明为正确的程序的风格去写程序*。我们称之为*证明友好风格*。我们想证明应该是能帮助我们，而不是成为额外的负担。让证明帮助我们的方法是以证明友好风格写程序，或者至少能在证明友好风格和有效率的风格之间改变。

在这一节里面我们建立一套证明方法，从简单的开始逐渐复杂，但是一直会控制复杂的程度。我们发展的风格将会用在这本书里面。基础是单赋值 paradigm 和递归。单赋值 paradigm 在 3.5.3 小节引入。

3.5.1 定义和术语

Block 是一段代码，block 有唯一的入口和唯一的出口。Blocks 是程序代码和过程代码的主要部分。一个*过程*是一个带名字的 block。过程常使用一些*参数*（*parameters*），以我们的理解来说，参数可以是*输入*也可以*输出*的。为了简化问题，我们假设没有参数是既用于输入也用于输出；可以设计两个参数分别完成输入和输出。还有，我们假设输入参数在过程执行过程中*不会被修改*。如果需要修改，将把参数复制到一个局部变量中。这个约定允许我们将输入参数当作 postconditions without having to specify that we are referring to the values at the time of entry.

一个*函数*是一个带输出参数的过程；如果有多个输出参数我们可以假设他们被封装在一个*组织者类*对象（参见 1.2.2 节），因此他们可以被一个 **return** 语句返回。因为仅有一个出口点，所以 **return** 语句必须是这个出口点。这个形式允许我们将函数当作一个特殊的过程处理。

一个过程通常不引用*非局部数据*，即不在过程的头，也不在过程的体中定义的任何数据。实际上，如果一个过程没有输出参数，调用他能产生效果的唯一方法是引用非局部数据。同样的，过程可以定义局部数据。在过程执行的过程中，过程的参数也可以认为是一种局部数据。

一个在过程中的 block 也可以引用*非局部数据*，即在 block 之外定义的数据。这也叫做全局数据。这可以是外层 block 封装的数据，或者是过程之外的数据，总之是符合使用的语言的可见规则，能访问到的数据。

对于数组必须特别说明一下。如果将数组作为一个参数传递，数组的*引用*被认为是局部数据，但是数组的*内容*被认为是非局部数据。类的的，在 Java 中，一个对象的*引用*是局部的，但是对象的*实例域*是非局部的。更新非局部数据对于某些算法的效率非常重要，但是为证明正确性带来了很大的麻烦。

3.5.2 基本控制结构

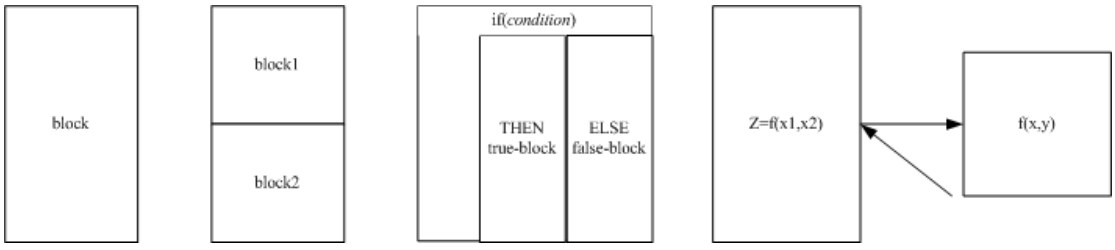


图 3.5 基本控制结构

控制结构是导致不同的 block 被执行的机制。一开始我们仅考虑 3 种控制结构（参见图 3.5）：*顺序*（block1，之后 block2），*选择*（if *条件* then block1，else block2），和 *过程条用*。在我们的基本证明方法学中省略 for 和 while 循环是特意的。在发展出基本方法学之后，我们讨论 3 种构造的适应性（在 3.5.4 小节）。

我们能不用循环写出任意可用的程序吗？吃惊的答案是“yes”。可能使用递归，，而且用递归一般比用循环更简单。

“证明正确性”意味着证明一个过程的特定逻辑语句。例如“limited warranty”，statements are phrased carefully, so that sweeping that a proof would be hopelessly difficult. 现在我们描述这些语句德形式。

定义 3.4 Precondition, postcondition，和规范

Precondition 是一个关于输入参数和一个 block（包括过程和函数）的局部数据的逻辑语句，在 block 进入的时候，precondition 应该是真。*Postcondition* 是一个关于输入参数、输出参数、block 的非局部数据的逻辑语句，在 block 退出的时候，postcondition 应该是真。Block 的规范是 preconditions 和 postconditions，他们描述了 block 正确的行为。

每一个 block（包括过程和函数）必须有规范，如果我们想证明它的正确性的话。为了证明正确的行为，证明下面形式的引理就足够了。

命题 3.9 （通用正确性引理形式）如果当 block 进入的时候所有的 *preconditions* 都满足，则 block 退出的时候所有的 *postconditions* 都应该是真。

假设一个 block 可以分为*顺序结构*：block1 then block2。为了证明 block 的正确性，证明如下形式的引理就足够了：

命题 3.10 （顺序结构正确性引理形式）

1. Block 的 preconditions 包含了 block1 的 preconditions。
2. Block1 的 postconditions 包含了 block2 的 preconditions。
3. Block2 的 postconditions 包含了 block 的 postconditions。

假设一个 block 可以分为选择结构：if（*condition*）then true-block，else false-block。为了证明 block 的正确性，证明如下形式的引理就足够了：

命题 3.11 （选择结构正确性形式）

1. Block 的 preconditions 逻辑与 *condtion* 为真包含了 true-block 的 preconditions。
2. True-block 的 postcondition 逻辑与 *condition* 为真（在 true-block 进入的时刻）包含了 block 的 postcondition。
3. Block 的 preconditions 逻辑与 *condtion* 为假包含了 false-block 的 preconditions。

4. False-block 的 postcondition 逻辑与 *condition* 为假（在 false-block 进入的时刻）包含了 block 的 postcondition。

图 3.6 展示了命题 3.10 和 3.11 是如何进行组合以获得类似命题 3.9 的证明形式的。

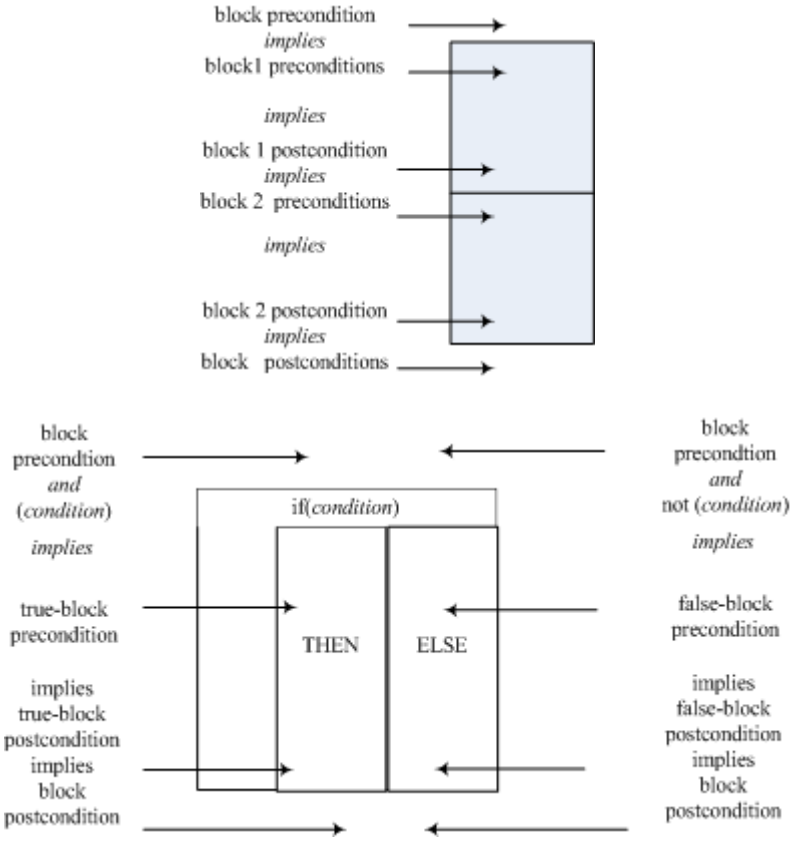


图 3.6 用来证明*顺序*和*选择*结构 block 的 preconditions 能推倒出 postconditions 的推倒链

假设一个 block 由一个过程调用组成。为了证明 block 的正确性，证明如下形式的引理就足够了：

命题 3.12 （过程调用正确性引理形式）

1. Block 的 preconditions 包含了以实际参数调用过程的 preconditions。
2. Block 的 postconditions 包含了以实际参数调用过程的 postconditions。

重要的是注意为了证明包含过程调用的 block 的正确性我们不需要证明过程调用本身的正确性；过程调用的正确性是单独的结果。

我们描述的证明的结构，这使得我们可以证明一个 block 的正确性，但是还没有进入如何证明特定程序语句的细节。这是很有技术性和复杂的主题。

例如，假设我们看到 java 语句，“x=y+1”。在语句之后我们可以知道什么逻辑语句（也就是什么是语句的 postcondition）？似乎 postcondtion 是等式 x=y+1。但是假设语句是“y=y+1”？或是假设我们有一个语句序列“x=y+1, y=z”？

实践中，人们依赖“common sense”arguments，而不是正式的证明方法。不是试图指出过程代码隐含什么逻辑语句，我们关注描述的 postcondition 是否能到达，而且试图对于结论得出 *ad hoc* arguments。下一个主题讨论更好的方式。

3.5.3 单赋值范例

在早期对证明友好的编程风格的研究中，研究者发现引起证明困难的是主要是两种构造：*goto* 语句和*赋值*语句。消除赋值语句是不切实际的，所以早期的研究者开始消除 goto 语句，结果导致了结构化编程的兴起。不幸的是，即使没有 goto 语句，证明通常还是困难的。

最近，消除赋值语句的问题被重新提出，出的方法是消除重新赋值（overwriting）赋值语句。就是说，在变量创建之后，变量只有一个值；一旦赋值之后就不能被覆盖。既然，变量的值在变量的生存期之内不能改变，那么关于变量论证就简单了。这就是*单赋值范例*。

很多程序设计语言有一体化的单赋值范例约束，比如 Prolog、ML、Haskell、Sisal（for Streams and iteration in a single assignment language）和 SAC（单赋值 C）。

在其他程序设计语言中，包括 C、Fortran 和 Java，也可以在不改变程序的行为的情况下转换成单赋值。这种转换用于编译优化和并行代码检测。研究发现采用单赋值形式之后，程序分析可以达到很深的程度（参见本章后面的 Notes 和 References）我们能在日常的编程利用单赋值范例的优点吗？

*单赋值范例*不能通用，但是可以非常简单的用于非循环的局部变量。非循环代码包含递归过程调用的代码，所以这个限制还不至于严重单赋值范例无用的地步。，事实上，Sisal 的编译器讲 **for** 和 **while** 循环转换成递归过程调用，所以它能在转换后的程序中采用单赋值范例。由于强制使用了单赋值范例，Sisal 编译器能自动分析哪一段代码可以并行执行。但是，单赋值范例的限制形式也可以和 **while** 和 **for** 循环一起使用。

复习一下本小节前面提到的使得证明变困难的赋值语句。在单赋值范例中“x=y+1;”*隐含*等式 x=y+1 在整个程序中 x 有一个值。麻烦的语句是“y=y+1;”和“x=y+1; y=z”，这两个违反单赋值范例的语句，将 y 第二次赋值了。

在没有循环的过程中，x 和 y 可以是局部变量，我们总是可以通过定义额外的局部变量做我们想做的计算。

例 3.5

为了修正语句“y=y+1;”我们写“y1=y+1;”来代替，我们得到了有效的等式 y1=y+1。为了修正“x=y+1;y=z”我们必须有两个有效等式，x=y+1 和 y1=z。在两种情况中，过程这个分支后面所有对 y 的引用都可以改成 y1，因为 y1 是修改后的值。

让我们看一个更普通的困难：一个变量仅在选择结构一个的分支里面才更新，但是在分支后面使用时混合了分支的两种情况。

例 3.6

考虑下面的代码片段

```
1.if(y<0)
2.    y=0;
3.x=2*y;
```

根据我们前面说的，我们必须定义一个局部变量 y1，然后将第二行替换成“y1=0”。但是第 3 行呢？显然我们不知道该使用 y 还是 y1。解

决方案是：如果局部变量在选择结构的一个分支里面赋值，则在所有的分支中赋以正确的值。在这种情况下，使用多条赋值语句但是仅有一条赋值语句将会执行。修订后的符合单赋值范例的代码是

```
1.if(y<0)
2.    y1=0;
3.else
4.    y1=y;
5.x=2*y1;
```

现在我们对于变量有了一个清晰的逻辑关系（$\Rightarrow$表示推导出，$\wedge$表示合取）：

$$(y<0 \Rightarrow y1=0)\wedge(y\geq 0 \Rightarrow y1=y)\wedge(x=2y1)$$

创建额外变量的做法可能会导致效率下降，但是实际的编译器优化可以轻易的判断，如果原来的 y 没有再被引用了它的空间用来存储 y1。

然而，记住单赋值范例，虽然可以使用局部变量，但是对于有数组时是不适用的。需要更新数组元素是很经常的事情，显然我们不能因为修改了一个元素就重新定义一个完整的数组。即使我们这样做了，我们还是会在试图得到任何描述数组状态的逻辑语句时遇到困难。需要更新对象的实例域时也会遇到同样的问题。

转换没有循环的过程

如果我们相对 while 或 for 循环采用本节讨论的证明工具，and the procedure is reasonably compact，最简单的方法可能就是将循环转换成一个递归过程。

例 3.7

算法 1.1 给出了一个顺序查找的迭代过程。下面的代码给出递归版本，并且使用单赋值范例。命题 3.9 到 3.12 用于证明其正确性。（我们不证明算法 1.1 的正确性，它将一个变量赋予多个值导致了它的复杂。）

复习*通用查找例程*的模式（参看定义 1.12）：如果再没有数据了，失败；否则查找一个数据；如果是我们想找的，成功；否则查找剩下的数据。这个模式在下面的过程中清晰的给出。

算法 3.1 顺序查找，递归

输入： E, m, num, K, 这里 E 是一个有 num 个元素的数组（索引从 0, …, num-1），K 是要查找的元素，m≥0 是要查找的数组段的启示索引。为了简化，我们假定 K 和 E 的元素都是整数，就和 num 一样。

输出： ans，一个在 E 中 K 的位置，范围是 m≤ans<num，或是-1，如果 K 在范围内没有找到的话。

注意：顶层调用必须是 ans=seqSearchRec(E, 0, num, K)。

```
int seqSearchRec(int[] E, int m, int num, int K)
{
    int ans;
1.  if(m>= num)
2.      ans=-1;
    else
    {
3.      if( E[m]= =K)
4.          ans=m;
5.      else
6.          ans= seqSearchRec(E, m+1m num, K);
    }
7.  return ans;
}
```

注意 ans 出现在三个赋值语句中，但是他们在不同的分支里面，所以这是符合单赋值范例的。让我们来看看如何采用已知的命题来验证过程的正确性。

首先，我们需要公式化 seqSearchRec 的 preconditions：

1. m≥0
2. 对于 m≤i< num，E[i]是初始化的。

现在我们给出目标或是 postcondition，即在第 7 行必须是 true 的。

1. 如果 ans=-1，则对于 m≤ j < num， E[i]≠K。
2. 如果 ans≠-1，则 m≤ ans< num，且 E[ans]=K。

现在，使用命题 3.9，我们展示了如果当进入 seqSearchRec 的时候 precondition 是满足的，则过程调用结束的时候 postcondition 就是满足的。我们发现过程分离到三种选择情况中，，第 2,4,6 行，最后汇聚到第 7 行的 **return** 语句。命题 3.11 用于每种情况。对于每一个选择，导致选择的条件是可以用于证明每一个分支都导致 postcondition 的额外的信息。

如果到达第 2 行，则第 1 行的条件是 true（m≥num）。第 2 行之后，ans= -1，之后就到了第 7 行。此时，第 1 行条件为 true 包含了没有索引在

范围≤ i < num 内，所以条件 1 满足。条件 2 也是 true，因为它的前提是 false（复习 1.3.3 小节）。

类似的，如果达到第 4 行，第 1 行的条件是 false（所以 m<num），且第 3 行的条件是 true（E[m]=K）。第 4 行自己建立了等式（ans=m）。合并这些域，条件 2 是正确的。等式（ans =m）和 precondition 1 可以得出 postcondition 1 的前提是 false；因此 postcondition 1 是 true。

最后，如果到达第 6 行，第 1 行和第 3 行的条件都是 false（所以 m<num 且 E[m]≠K）。首先，我们需要展示我们在第 6 行以实参“正确的调用”了 seqSearchRec。就是说，当我们初始化这些实际参数的时候，我们需要验证 seqSearchRec 的 preconditions：

1. 如果 m≥0，则 m+1≥0。

2. 范围 m+1, …, num+1 包含在 m, …, num-1，所以 E[i]是初始化的。

现在，根据命题 3.12，*我们可以得出结论第 6 行的过程调用符合他的 postconditions*。既然第 6 行赋予 ans 的值就是调用返回的值，它满足调用的 postcondition（以实参 m+1）。这些 postconditions 和参数 E[i]≠K 隐含了当前调用的 postconditions（实参是 m）。例如如果返回-1，隐含了 E[m+1],…, E[num-1]都不包含 K，所以 E[m],…, E[num-1]也不包含 K。另外 ans≥m+1，也有 ans≥m。

因此我们展示了无论第 7 行如何到达，需要的 postconditions 都能达到。剩下的唯一问题是是否存在可能性第 7 行永远无法到达，即进入了无限递归中。3.5.6 小节讨论这个问题。

联系 3.6 要求读者使用本小节的技术证明找两个整数最大公约数的欧几里得算法的正确性。

3.5.4 带循环过程

命题 3.9 到 3.12 给我们了一个在没有 **for** 和 **while** 循环的情况下证明正确性的框架。只要有循环，单赋值范例通常就是不可能的了，因为必须定义一个循环变量，indexed both by 过程中的行号和 by number og passes through the loop to keep track of all the values taken on by the same 程序变量。Then 必须小心的 trace 每一个变化中的历史值。我们相信相对于规格化过程本身来说，将循环转换成递归过程在实践上更简单。

事实上，一旦我们理解了循环和递归版本之间的关系，通常没有必要实际的完成转换。作为一个预处理步骤，我们必须下列：

1. 在循环体内部最宽范围内申明一个局部变量，遵循单赋值范例。就是说，在一轮仅给变量一个值。
2. 因为变量必须更新（而且必须声明在循环外面），在循环体的终结处做所有的更新。

这些规则最小化了必须考虑到的不同情况的数量。

将一个 **while** 循环重新表示成递归的一般规则是：

1. 在循环内更新的变量变成一个过程的输入参数。他们在循环入口的初始值对应顶层递归调用的实际参数值。我们称这些为*活动参数（active parameters）*.

2. 循环中引用的但是早先的定义的，

For 循环的规则类似。

在 3.7 图中展示了一个斐波那契函数的转换。注意 n 是一个传递的参数。

```
int factLoop(int n)
{
    int k, f;
1.  k=1;
2.  f=1;
3.  while(k≤n)
4.  {
5.      int fnew = f*k;
6.      int knew= k+1;
7.      k= knew; f=fnew;
8.  }
9.  return f;
}
```

```
int fact(int n)
{
9.  return factRec(n,1,1);
}

int factRec(int n, int k, int f)
{
    int ans;
3a. if(k>n)
3b.     ans =f;
4.  else
    {
5.      int fnew = f*k;
6.      int knew =k +l;
7.      ans =factRec(n, knew, fnew);
    }
    return ans;
}
```

图 3.7 while 循环转换成递归函数。与转换不相关的花括号省略了。

3.5.5 正确性证明作为调试工具

证明正确性最大的实际价值一——即使是非常不正式“心算”证明——在于证明能在编码和测试没有开始之前就常常精确指出了 bug 所在。有时证明仅是完整的理顺一下过程的 preconditions 和 postconditions，并*把他们作为注释写在代码中*。（即使你的证明将是一个“心算”，你也不应该逃避这步写文档的工作。）

许多程序 bugs 是过程的 preconditions 和调用过程时的实际条件不匹配，而且这种不匹配是简单和显而易见的。另外大部分是因为 postconditions 不匹配。这种错误通常在考虑命题 3.12 之后就会显而易见。

当问题比较微妙，一切看上“看上去都对”，你必须试图使用命题来为一个一个合适大小的 block 来构在证明。这个过程就是问每一个 block，“假设这个 block 完成了会怎样？”，再问“它完成需要那些条件？”现在其一个 block 是否正确的达到这些要求？

如果代码中有一个 bug，而且你推理的很小心，*证明中断的点告诉你 bug 的所在*。就是，bug 就在两个 block 中穿过边界附近，在你发现 postconditions 和 preconditions 不匹配的地方。

例如，在递归顺序搜索（算法 3.1）中，如果第 1 行的条件错写成了（m≥num-1），则 postcondition 1 在第 2 行之后将不能得到，bug 就定位了。

另一个例子，假设算法 3.1 中 1-2 行和 3-4 行交换一下位置，就是说，第 1 行变成“if(E[m]==k)”。我们给出的证明中所有提到的语句都重复行号上的变化，但是过程有一个 bug。为了在检查期间找出 bug，我们必须注意到*所有*计算一个表达式的语句的 precondition

是表达式中的所有数据元素都已经给出值，也就是我们不能 access 未初始化的变量和实例域。如果 m 可能是大于 num-1 的数，我们不能保证这一点。再一次的，证明试图暴露 bug，但是仅在我们检查的很仔细的时候才行。在我们复查代码的时候问一下“数据元素初始化了吗？”是一个很好的习惯。

3.5.6 递归过程的终止

命题 3.9 到 3.12，3.5.2 描述的引理等有递归过程的场合，展示什么叫*部分*正确，因为他们没有给出过程是否该终止。为了完成完全正确的证明，必须展示每一次递归调用都得到比原来要小的问题。

这时首先需要证明递归调用的 precondition 是满足的，其次是讨论传递给递归调用的结构或是“问题规模”比调用者的要小。作为另一个正确性问题，实践中人们使用合理的一般的讨论，而不用带公理和推论的正式的证明。

在许多情况下，问题的规模是一个非负整数，比如子范围的元素数量，链表中元素的数量，诸如此类。例如在算法 3.1 中，在 3.5.2 小节“问题规模”方便的定义为 n=(num-m)，未检查元素的数量。每次递归调用为导致减少 1，最终会减少到 0。

在有些情况下，one can use directly a partial order defined on the structure being passed as an imput parameter, such as the *subtree* partial order(see 图 3.3)。例如，在一个二叉树遍历过程中，如图 3.4，如果过程的输入参数是树 T，而 T 不是基本情况，则 T 的子树比 T“要小”in partial order。因此在形式正确的二叉树结构上的递归过程会结束。

为了技术上的正确，在二叉树上做递归的过程必须有一个 precondition，即输入参数 T 是正确的二叉树——没有环。规范 BinaryTree 抽象数据结构类型一个动机（如同我们在 2.3.3 小节做的）是使得这个条件自动满足。

3.5.7 二分查找的正确性

我们现在证明递归过程 **binarySearch** 的正确性（二分查找的细节参看算法 1.4）。这作为一个用归纳法证明递归过程的示范。一个归纳证明建立在全部的不带循环递归过程正确性的基础之上；就是说，it establishes that the procedure terminates, as well as establishing that its preconditions imply its postconditions.（如果递归过程调用子例程，则子例程的正确性要在递归过程被证明之前作为一个*假设*添加。）

我们定义 binarySearch 的问题规模 n= last –first+1，E 在查找范围中的条目的数量。图 3.8 只是为了方便做的一个重复。

> **引理 3.13** 对于 n≥0，如果 **binarySearch(E, first, last, K)**被调用，问题的规模是(last- first +1)=n，E[first], …, E[last]在非递减阶，则如果 K 不在 E 中 first, …, last 的范围中返回-1，否则如果 K=E[index]返回 index。
>
> **证明** 证明归纳 n，问题的规模。基本情况是 n=0。这时第 1 行是 true，到达第 2 行，返回-1。
>
> 对于 n>0，假设 binarySearch(E, f, l, K)在 0≤k<n 的问题规模 k 上满足引理，f 和 l 是任意索引满足 k=l-f+1。因为 n>0，第 1 行市 false，first≤last，流程到第 4 行，然后是第 5 行。根据前面提到的不等式和等式

$mid = \lfloor (first + last)/2 \rfloor$，得 first≤mid≤last。因此 mid 在搜索范围内 。

如果第 5 行是 true，过程在第 6 行得到了结果。

现在假定第 5 行市 false。从前面的两个不等式和 n 的定义，我们有（根据≤的传递性）

$$(mid - 1) - first + 1 \le (n - 1)$$

$$last - (mid + 1) + 1 \le (n - 1)$$

所以对于第 8 行和第 10 行的两个递归调用递归假设都是适用的。

```
int binarySearcha(int[] E, int first, int last, int K)
{
1    if( last< first)
2         index =-1;
3    else
     {
4         int mid = (first +last)/2;
5         if(K== E[mid])
6              index=mid;
7         else
          {
7              if(K<E[mid])
8                   index= binarySearch(E, first, mid-1, K);
9              else
10                  index= binarySearch(E, mid+1, last, K) ;
          }
11   return index;
}
```

图 3.8 binarySearch 过程，算法 1.4 的重复

现在，如果第 7 行是 true，则执行第 8 行。直接就可以发现 binarySearch 的 precondition 是满足的，因为第 8 行的实际参数只有第 3 个参数改变了，而且是变小了。因此我们假定调用了完成了 binarySearch 的目标。如果第 8 行的调用返回一个正的索引，这解决了当前了的问题。如果第 8 行的调用返回-1，这意味着 K 不再 E 的范围 first, …, mid-1 内。但是第 7 行隐含了 K 不再范围 mid, …, last 内，所以从当前的过程返回-1 是正确的。如果第 7 行市 false，则执行第 10 行，是类似的。

关于证明需要强调的一点是，在我们（能）假定地 8 行和第 10 行的调用完成了他们的目标之前，我们需要验证调用的 preconditions。既然许多逻辑错误是由调用过程而没有满足过程的 precondition 引起的，这种检查能揭示很多 bugs。

3.6 递归等式

递归等式是定义在自然数上的函数，T(n)，它的值表达式中有一项或多项是小于 n 的整数的 T。换句话说，T(n)是归纳定义的。如所有的归纳一样，基本情况是单独定义，递归等式仅处理比基本情况大的 n。尽管我们的主要兴趣在用递归等式作为一种描述算法（运行时间，关键字的比较次数，或者其他的重要工具）的函数式工具，但这不是递归等式唯一的用途。许多有趣的数学函数可以用递归等式定义，比如 Fibonacci 数列，等式（1.13）。

用递归等式来描述递归过程是很自然的事情。本节的目的是展示如何从递归等式的代码中衍生出递归等式，并描述算法中一些经常出现的 patterns。3.7 节探索了如何解一些常见的递归等式。因为不同资源的数量可能需要度量（时间，空间，关键字比较次数等），我们将使用通用的术语 *cost* 来给出递归等式的界限。

我们首先要指定一种度量方法来度量，需要解的递归过程的问题规模：令规模是 n。递归等式左边将是 T(n)。为了构建递归等式的右边我们需要估计递归过程有多少不同的块，这些块的 cost 与 n 的函数关系。经常有些块的 cost 是常数。We can just call all constants 1 if we are satisfied with an answer that is within a constant factor.

在我们的术语学中，*子例程（subroutine）*是任意的过程，且不递归调用我们要分析的过程；就是说，子例程中的调用序列不能有一个调用自己的。与子例程相关的数量一般带角标 S。与递归调用相关的数量一般带角标 R。

如果过程没有循环，合并每一个块的最坏情况分析是简单的。

1. 对于块的*序列*，*增加*单独的 cost
2. 对于可选的块，包括基本情况，取选择项中*最大*的。
3. 如果块包含子例程调用，指出实际参数与 n 的函数关系。为了简化问题，假设只需要一个参数规模，称为 $n_s(n)$。我们需要知道子例程的 cost 函数 T_s。则这个调用的的 cost 是 $T_s(n_s(n))$。
4. 如果块包含递归调用，指出实际参数与 n 的函数关系，称为 $n_r(n)$。则递归调用的 cost 是 $T_r(n_r(n))$。这里的 T 与递归等式左边的一样。

等式右边出现的不含函数 T（等式左边的）的部分称为*非递归 cost*。要把这个 cost 和过程调用总的 cost 区分出来，过程总的 cost 也包含 T。

合并块的平均行为分许需要区别的对待*可选*的 construct。对于所有可选项应该求数学期望。Also，if the size of subproblems（$n_r(n)$和 $n_s(n)$）can vary for different inputs，子例程的 cost 和递归调用的 cost 需要平均。由于这个原因，平均行为分析通常认为比最坏情况分析要难。

例 3.8

对于简单应用的规则，考虑算法 3.1 的递归函数

seqSearchRec(E, m, num, K)

为了方便在这里重复一下：

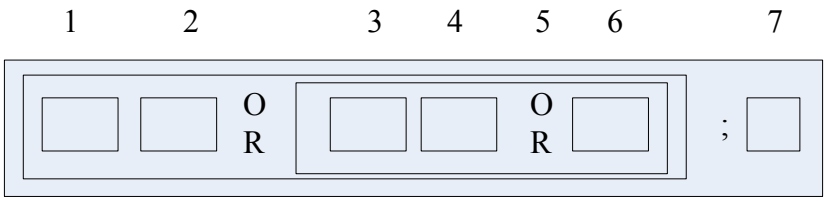
```
1. if(m≥num)
2.     ans=-1;
3.else if(E[m]==K)
4.     ans=m;
```

5.**else**

6. ans=seqSearchRec(E, m+1, num, K);

7.**return** ans;

我们指定问题规模为要数组 E 的元素个数。因此 n=num-m，这里 m 和 num 是当前调用的第 2 个和第 3 个参数。让我们将过程分解为块，这样就可以采用规则了。块可以通过行号的范围来描述。整个过程是 1-7。可以参照下面的 diagram，“OR”表示*可选项*，“；”表示*序列*。



基本情况是块 2-2，基本情况需要排除在递归等式之外，递归等式只考虑非基本情况。所有最里面的块 s 都是简单语句，除了 6-6。当 cost 是运行时间时，我们假设简单语句需要常量的运行时间，并用 1 代表任意常量时间（于是 1+1=1）。当 cost 是某些特定操作的数量时，则操作需要计数。For definiteness，我们将假设 cost 是数组元素比较的次数。因此第 3 行 cost 是 1，而在这个 cost 模型下其他简单语句没有 cost。

注意算法 3.1 中的第 6 行的调用，我们看到实际参数的第 2 个和第 3 个是 m+1 和 num，所以它的问题规模是 num-(m+1)=m-1。所以，6-6 的 cost 是 T(n-1)。整个块的 cost 由两部分构成，合并可选语句的最大项，合并序列块。块 2-2 作为一个可选项被排除。注意 1-7 是 1-6 和 7-7 的和，给出 seqSearchRec 递归等式的右边

$$(0+(1+\max(0,T(n-1)))+0$$

简单来说，我们看到递归等式是 T(n)=T(n-1)+1。这种情况下非递归 cost 是 1。

基本情况总是一个小问题，因此当 cost 时时间是，我们假设他们总是单独 cost。但是这里我们计算比较次数，所以 T(0)=0。

例 3.9

对于其他例子，考虑二叉查找，算法 1.4，在 3.5.7 小节有重复。问题规模是 n=last-first+1。消耗的度量是关键字比较，所以第 5 行消耗是 1。第 8 行和第 10 行的递归调用的问题规模是 n/2 和(n-1)/2，但是这是可选的，所以其合并的消耗取最大值，而不是两者的和。剩下的语句中没有关键字比较，所以递归等式是

$$T(n)=T(m/2)+1$$

在这个过程中，出现了两次递归调用实际上只有一次。在第 4 章我们将遇到排序过程，有些排序中有两次递归调用，他们的递归等式的右边会有每一次递归调用代表的项。

如果规模 $n_r(n)$或 $n_s(n)$不是很清楚就有了新的问题。例如，在 n 个节点的二叉树遍历中（2.3.3 小节），我们知道左子树和右子树总共有 n-1 个节点，但是我们不知道两边各是多少。假设我们引入额外变量 r 表示右子树的规模。则我们得到递归等式

$$T(n) = T(n-1-r) + T(r) + 1, \quad T(0) = 1$$

巧合的是，我们通过代入法能知道这个等式为 T(n)=2n+1，即使我们不知道 r。一般我们没有这么幸运，而且对于不同的 r 可能结果不一样。这个问题在快速排序中就出现了（4.4.3 小节）。

普通递归等式

我们能描述许多类递归等式，他们经常遇到且能通过标准方法解决（某种程度上）。在所有情况下“子问题”指主要问题比较小的实例，需要递归解决的。符号 b 和 c 是常量。

分而治之： 在分而治之范例的许多情况中，子问题的大小是 n/2 或 n 乘以某个固定的比例。例如我们已经看过的二分查找（1.6 节），我们在第四章要学习的：归并（4.6 节）和堆操作（4.8.3 小节）。使用 4.6 节的归并作为例子，归并排序做的比较数目是：

$$T_{MS}(n) = T_{MS}(\frac{n}{2}) + T_{MS}(\frac{n}{2}) + M(n), \quad T_{MS}(1) = 0 \tag{3.2}$$

M(n)的消耗是归并子例程产生的。在我们能解决 $T_{MS}(n)$之前，我们需要知道 M(n)是什么。

一般对于分而治之类型问题，规模 n 的主问题可以分解为 b 个（b≥1）规模是 n/c（c>1）的子问题。还有一部分非递归消耗 f(n)（为了分解问题，且/或合并子问题的解到主问题）。

$$T(n) = bT(\frac{n}{c}) + f(n) \tag{3.3}$$

我们称 b 是*分支因子*。

Chip and Conquer：规模 n 的主问题可以“切掉一部分”成一个规模是 n-c 的子问题（c>0），再带一个 f(n)的非递归消耗（为了分解问题，且/或合并子问题的解到主问题）。

$$T(n) = T(n-c) + f(n) \tag{3.4}$$

Chip and Be Conquered：规模 n 的主问题可以“切掉一部分”成 b 个（b>1）规模是 n-c 的子问题（c>0），再带一个 f(n)的非递归消耗（为了分解问题，且/或合并子问题的解到主问题）。我们称 b 是*分支因子*。

$$T(n) = bT(n-c) + f(n) \tag{3.5}$$

如果子问题有不同的规模，但是所有的子问题规模都在 $n-c_{max}$ 到 $n-c_{min}$ 之间，责低限可以分别使用 c_{max}，c_{min} 代替 c 来求。这种情况在练习 3.11 中考虑。

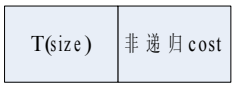
下一节我们探讨用数学方法分析这些典型的递归等式。

3.7 递归树

递归树提供了一种分析我们已知递归等式的递归过程的消耗的工具（运行时间，关键字

比较次数或是其他测量方式)。首先我们将以一个例子展示如何从递归等式生成一个递归树，然后我们将描述一般的过程。从一般过程中，我们将能派生出几种一般的解决方案（引理 3.14，引理 3.15，定理 3.15，定理 3.17，等式 3.12 和 3.13）。这些解决方案覆盖许多实际算法分析中出现的递归等式，而且当遇到并不符合标准形式的递归等式时，他们能作为一个粗略的 guide。It is not necessary to follow all of the technical details in this section to be able to apply the general solutions mentioned.

每一个递归树的节点有两个域，*size* 和*非递归 cost* 域。一个节点表示如下：



规模域代表这个节点处 T 的实际参数。我们加上递归名字 T 来提醒我们规模域不是 cost。

例子 3.10 简单的分而治之的递归树

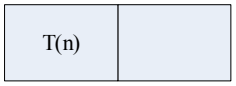
考虑递归等式：

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n$$

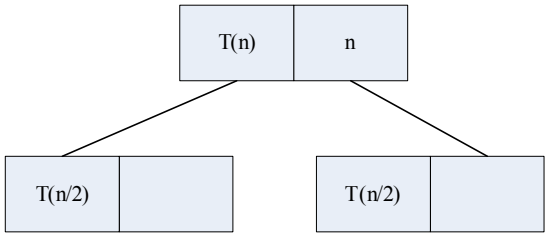
这是等式（3.3）的特殊情况，b=2 和 c=2。这是归并排序的一个稍微简单的形式，它会在很多地方出现。我们将一步一步的构造出相应的递归树。第一步，是以辅助变量重写等式（在递归假设中有一个类似的辅助变量），这有助于避免替换错误。我们称这个递归等式为我们的工作拷贝。

$$T(k) = T(\frac{k}{2}) + T(\frac{k}{2}) + k \qquad (3.6)$$

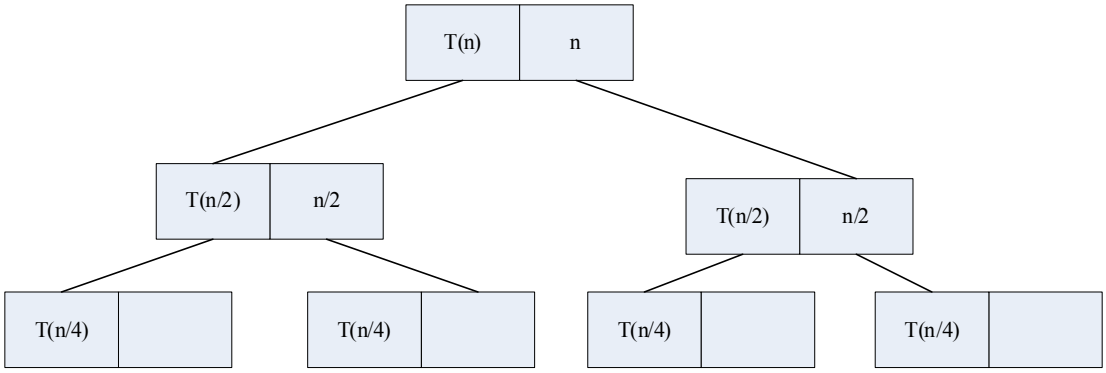
只要节点的 size 域知道了一个节点就可以创建了；之后我们就可以用 size 域计算非递归消耗的值。我们准备创建 T(n)递归树的 root 节点；这里 size=n。



决定非递归 cost 域和没有完成孩子节点的过程叫*扩展*节点。我们将节点的规模域添上准备扩展，这里是 n，替换掉我们工作拷贝 k，等式（3.6）。右边成为 T(n/2)+T(n/2)+n。所有带 T 的部分变成孩子节点，剩下的部分变成节点的非递归 cost，如下



既然同样深度的所有节点看上去一样，我们可以在分支中产生他们。一般情况下，每一个不完全节点必须根据自己的规模域产生。这里所有规模域都是 n/2，所以我们用 n/2 代入等式（3.6）中的 k，而且我们看到右边应该是 T(n/4)+T(n/4)+n/2。所以现在我们有



我们可以继续下一层，直到我们看出数的模式。图 3.9 展示了数扩展到另外一层；有 8 个未完成孩子，其细节没有显示出来。这里我们可以看到以深度为函数的规模参数, $n/2^d$, 非递归 cost 恰好也是 $n/2^d$。（复习下，为了方便，我们将根的深度定义为 0）。在这个简单的例子中，有同样数深度所有节点都是一致的，但是这不总是成立的。

我们在下一页总结生成一颗递归树的规则 s。

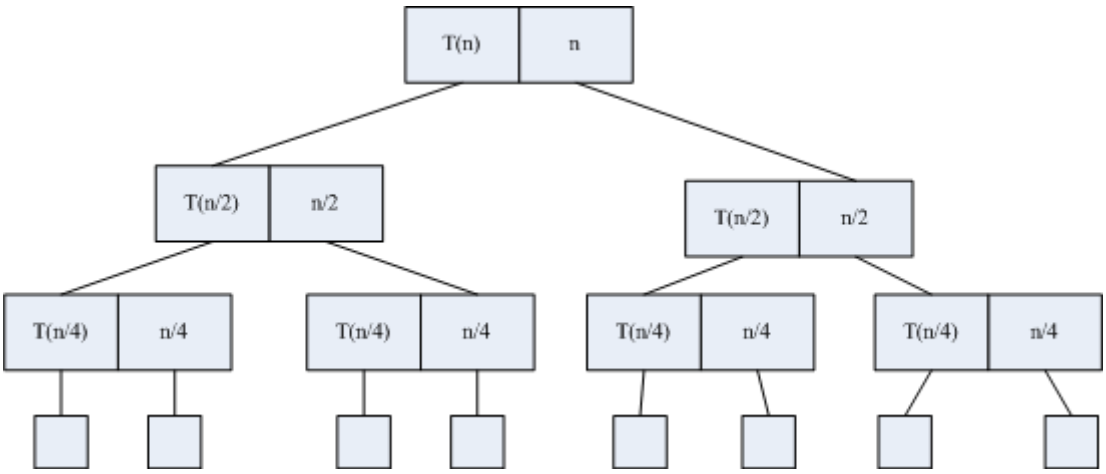


图 3.9 三层递归树。8 个未完成孩子的规模域没有显示。

定义 3.5 递归树规则

1. 递归等式的*工作拷贝*使用和原来不同的变量；称为*辅助变量*。这里令 k 是辅助变量。原来递归等式的左边（假设是 T(n)）变成递归树根节点的规模域。

2. 一个未完成节点有一个规模域，但是没有非递归 cost。

3. 决定给递归 cost 和未完成节点孩子的过程称为*扩展*节点。我们取出待扩展节点的规模，代入递归等式的辅助变量 k 中。在右边包含 T 的部分变成节点的孩子；所有剩下的部分称为节点的非递归调用。

4. 扩展基本情况给出非递归调用，对于基本情况没有孩子。

为了简化表示形式，我们假定递归等式的基本情况的 cost 不为 0。如果等式的基本情况的 cost 是 0，我们可以只计算最小的不为 0 情况，将它作为基本情况。

事实上，我们经常假定基本情况 cost 是 1，for definiteness。如果必要可以做变动。

在任意递归树的子树里面，下面的等式成立：

根的规模=Σ扩展节点的非递归 cost

+Σ未完成节点的规模 （3.7）

这很容易由归纳法证明。在基本情况下，T(n)=T(n)。一次扩展后，根节点被扩展，而孩子是未完成的，所以等式（3.7）给出的是原始的递归等式，以此类推。

例 3.12 递归树

在例子 3.10 的 7 个节点递归树中（带 4 个未完成节点），等式（3.7）表示为 T(n)=n+2(n/2)+4T(n/4)=2n+4T(n/4)。

计算递归树的技术是：首先计算每一深度所有节点非递归 cost 的和；称为树这一层的*行和*（*row-sum*）。然后计算所有行和的和。继续图 3.9 的例子，有些行和已经展示在图 3.10 中了。

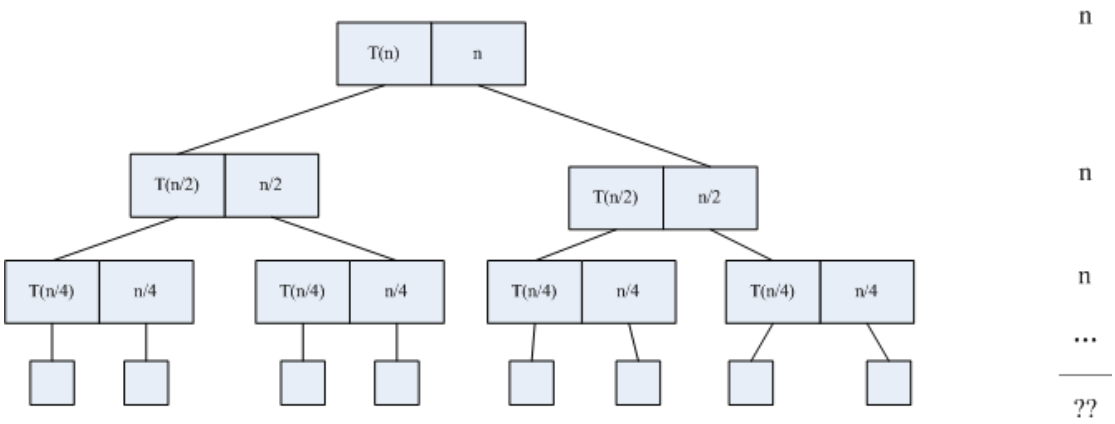


图 3.10 递归树中非递归 cost 的和。头 3 行的行和列在右边

为了计算行和的总和，（通常）需要知道递归树的最大深度。也就是规模约简到基本情况的层数。

例 3.12 递归树计算

对于例子 3.10（参看图 3.10）我们观察规模与节点深度的函数是 $n/2^d$，所以基本情况发生在 d=lg(n)。既然每一行的和室 n，树的总和是 T(n)=nlog(n)。

3.7.1 分而治之， 一般情况

使用例子 3.10 到例子 3.12 同样的步骤，我们能估计一般分而治之递归等式（等式（3.3），为了方便这里重复一下），以得到 T(n)的渐进阶。

$$T(n) = bT\left(\frac{n}{c}\right) + f(n) \qquad (3.8)$$

这小节得到这种技术，但是引理和定理可以被理解，而且可以被单独使用。

首先，我们可以看到规模参数以因数 c 每次递减（我们令 c=2 为例）。因此基本情况发生在$(n/c^D)=1$ 时，这里 D 是基本情况节点的深度。解方程得 $D=\lg(n)/\lg(c)\in\Theta(\log(n))$。然而，我们必须跳出所有深度行和都一样的情况。

知道树*有多少*叶子是很有用的。分支因数是 b，所以深度 D 的节点数是 $L=b^D$。为了方便：$\lg(L)=D\lg(b)=(\lg(b)/\lg(c))\lg(n)$。系数 lg(n)经常出现，我们给它一个名字。

定义 3.6 临界指数

对于等式（3.3）中的 b 和 c，我们定义临界指数为

$$E=\frac{\lg(b)}{\lg(c)}$$

通过引理 1.1，第 8 部分，对于 E 这样形式的对数可以使用任意方便的底，只要分子分母同时还就行。以这个符号，有如下引理：

引理 3.14 等式（3.8）递归树的叶子数量近似是 $L=n^E$，这里 E 是定义 3.6 中定义的临界指数。

假设叶子中的非递归 cost 是 1，这告诉我们树的 cost *至少*是 n^E。即使叶子的非递归 cost 是 0，在叶子的上层也肯定会有非 0 的 cost 的节点（或者极端情况下叶子之上有常数层非0）。但是这层中仍然是 $\Theta(n^E)$节点，所以保持的 $\Omega(n^E)$底限。

让我们总结下目前得到的结论。

引理 3.15 用前面使用的符号，我们近似有：

1. 递归树有深度 D=lg(n)/lg(c)，所以有许多行和。
2. 第 0 层的行和是 f(n)，即根的非递归 cost。
3. 第 D 层行和是 n^E，假设基本情况是 1，或者 $\Theta(n^E)$，in any event。
4. T(n)的值，也就是等式（3.8）的解，是树中所有非递归 cost 的和，也就是所有行和的和。

在许多实际情况中，行和形式是一个几何多项式（或者近似的上下界都是多项式）。几何多项式的形式 $\sum_{d=0}^{D} ar^d$（1.3.2 小节）。常量 r 称为*比率*。Quit a few simplification occour in practice that are based on the principle of 定理 1.13 的第 2 部分，which stated that, for a 几何多项式 whose 比率不是 1，the sum is in Θ of its largest term。由这个定理和引理 3.15，我们可以得出结论：

定理 3.16（Little Master Theorem）以前面讨论的符号，和等式（3.8）定义的 T(n)：

1. 如果行和组成了一个几何多项式（从第 0 行数的顶层开始），这里 E 是定义 3.6 定义的*临界指数*。就是说，cost 与递归树的叶子数量成比例。

2. 如果行和保持常量，$T(n) \in \Theta(f(n)\log(n))$。

3. 如果行和组成一个递减几何多项式，则 $T(n) \in \Theta(f(n))$，这和根的 cost 成比例。

证明 在情况 1 中和由最后一项支配。在情况 2 中有相当于 $\Theta(\log(n))$的项 。在情况 3 中和由最后一项支配。

为了更抽象适用范围更广，将这个定理泛化。泛化是很有用的，当等式（3.8）中的函数牵涉到对数时，因为之后行和可能不是很整洁。（对于更一般的版本，参见联系 3.9。）

定理 3.17 （Master Theorem）以前面讨论的术语，递归等式的解

$$T(n) = bT(\frac{n}{c}) + f(n) \qquad (3.9)$$

（重复等式 3.3 和 3.8）有如下解的形式，这里 $E=\lg(b)/\lg(c)$是定义 3.6 定义的*临界指数*。

1. 对于任意正数 ε 如果 $f(n) \in O(n^{E-\varepsilon})$，则 $T(n) \in \Theta(n^E)$，与递归树的叶子数量成比例。

2. 如果 $f(n) \in O(n^E)$，则 $T(n) \in \Theta(f(n)\log(n))$，as all node depths contribute about equally。

3. 对于任意正数 ε 如果 $f(n) \in \Omega(n^{E-\varepsilon})$，并且对于 $\delta \geq \varepsilon$ 有 $f(n) \in O(n^{E+\varepsilon})$，则 $T(n) \in \Theta(f(n))$，与递归树根的的非递归 cost 成比例。

（可能没有一种情况是适用的。）

证明 深度 d 的节点有 b^d 个节点，每一个都有非递归 cost $f(n/c^d)$。因此我们对于等式（3.8）的解忧下面的通用表达式：

$$T(n) = \sum_{d=0}^{\lg(n)/\lg(c)} b^d f(\frac{n}{c^d}) \qquad (3.10)$$

我们将只给出一个框架证明，它遵循定理 3.16 给的理由。（参看 Notes 和 References 有完全的证明）考虑情况 3。忽略系数，对于某些正数 ε，f(n) 关于 $nE+\varepsilon$ 。所以

$$f(\frac{n}{c^d}) \approx \frac{n^{E+\varepsilon}}{(c^d)^{E+\varepsilon}} \approx \frac{f(n)}{c^{Ed+\varepsilon d}}$$

则 $b^d f(n/c^d)$是 $f(n)b^d/(c^{Ed}c^{\varepsilon d})$。但是 $c^E=b$ through standard identities，所以分母是 c^{Ed} 和分子 b^d。我们最后有 $f(n)/c^{\varepsilon d}$，给 d 一个递减的几何多项式。其他情况的分析类似。

3.7.2 Chip and Conquer，或者 Be Conquered

一个不同的图片融合了等式（3.4）和（3.5）。如果分支因子大于 1，我们有等式（3.5），这里为了方便重复一下：

$$T(n) = bT(n-c) + f(n) \qquad (3.11)$$

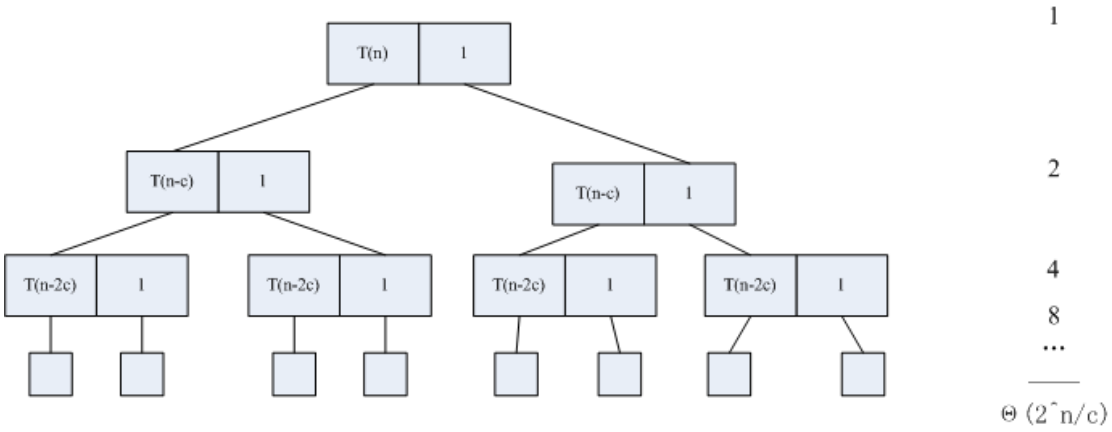


图 3.11 在一个 chip-and-be-conquered 递归树的非递归 cost 的和

图 3.11 展示了等式（3.11）一个例子的递归树，令 f(k)=1。既然当深度增加 1 规模以 c 递减，基本情况发生 d=n/c。

如图 3.11 中展示的，树的总数是以 n 为指数。这保持了最喜欢的假设即 f(n)=1。下面的一般表达式可以通过检查等式（3.11）的递归树得到：

$$T(n) = \sum_{d=0}^{n/c} b^d f(n-cd) = b^{n/c} \sum_{h=0}^{n/c} \frac{f(ch)}{b^h} \quad \text{（等式 3.11 的解）} \qquad (3.12)$$

这里第 2 个和使用了 h=(m/c)-d，所以 h 在叶子处是 0，朝向根逐渐增加。在大多数实践情况下，最后一个和是 $\Theta(1)$，于是有 $T(n) \in \Theta(b^{n/c})$。这个函数以 n 的指数增长。一个更一般的 chip-and-beconquered 情况在联系 3.11 中提及。如我们在 1.5 节看到的，以指数增长的算法在稍大的输入规模时将不能解决其最坏情况。

但是，如果等式（3.11）（等式 3.4 给出）分支因子 b 是 1，则等式（3.12）更一般的表达式变得友好一点：

$$T(n) = \sum_{d=0}^{n/c} f(n-cd) = \sum_{h=0}^{n/c} f(ch) \approx \frac{1}{c}\int_0^n f(x)dx \quad \text{（等式 3.4 的解）} \qquad (3.13)$$

例如，如果 f(n)是一个多项式 n^{α}，则 $T(n) \in \Theta(n^{\alpha+1})$。可选的，如果 f(n)=log(n)，则 $T(n) \in \Theta(n\log(n))$。（参看 1.3.2 小节）

总的来说，我们有两个工具来计算递归过程的 cost：递归树和递归等式。他们是一种信息的两种表现形式。人们发展了许多技术来计算经常出现的树和等式形式。及时不是很适合标准形式时可以用代入，递归树依然表示了递归等式正确的解；它只是可能会难以求解。

*3.7.3 为什么递归树能工作

这小结给出了一个给定递归等式和它的递归树的联系，以及编写好的用来计算递归解的函数。读者可以略过本节，而不会丢失任何连续性。

一种可视化递归树的方式是想象我们实际上编写了一个简单的递归函数（称为evalT(k)）来计算递归等式的值，比如等式 3.2 到等式 3.5。这个函数的 activation 树非常精确的对应了递归树。递归等式看上去如 T(k)=f(k)+… （带 T 的项）。我们假设我们的递归函数 evalT 有一个参数 k（代表问题规模），而一个局部变量 nonrecCost 来存储非递归 cost 的值（也就是 f(k)）。忽略基本情况，假设 evalT 的代码如下：

```
nonrecCost=f(k);
return nonrecCost+ … （带 evalT 的项）
```

这里带 evalT 的项只是模仿了递归等式右边带 T 的项。

递归等式的*递归树*，以 T(n)为根，将是 evalT(n)的 *activation 树*。（这假设非递归 cost 函数 f(k)可以用简单语句计算。译注：即没有函数调用。）

可以看出，整个树的 nonrecCost 值得和就是顶层调用返回的值。我们假定顶层调用时 evalT(n)，就是为了计算递归等式 T(n)的值。

第四章 排序

4.1 概述

在本章中我们将学习几个排序的算法，就是将集合中的元素按顺序排列。将一组元素排序的问题是第一个广泛研究的计算机科学问题。已知最好的分而治之的许多算法设计范例都是排序算法。在 20 世纪 60 年代，当商业数据处理大规模使用自动化时，在安装计算机的地方排序程序是运行最多的程序。一个软件公司靠着更好的排序程序在行业内能领先好几年。以今天的硬件水品，排序性能的关键有所改变。在 20 世界 60 年代，低速存储器（磁带或磁盘）和主存之间的传输速度是主要的性能瓶颈。主存一般只有 100,000 字节，而要排序的文件大多比这要大一个数量级上。主要的研究都是适合这种类型的排序算法。现在，主存容量增长了 1,000 倍（也就是约 100MB），或者高达 GB。所以大多数文件都能装入主存了。

有好几个原因要学习排序算法。首先，经常需要排序。以字母顺序排序电话簿或字典以使他们更容易使用，在计算机中操作大容量有序的集合比不无序的要容易。其次，人们开发了如此多的排序算法（本章将讲解很多种），而且学习很多排序算法会给你深刻的影响，即你可以从很多方面来考虑同样的一个问题。如何改进一个给出的算法，如何在多个算法之间取舍，本章对算法的讨论一定会让你更清楚的理解这两个问题。最后，排序是少数几个我们能简单的推导出最坏情况和平均行为的强低限的问题。说低限是强的是指已经有算法近似的达到了最小的工作量。因而我们已经根本上最优化了排序算法。

在大多数算法的描述中，我们假定要排序的集合存储在数组中，所以访问任意位置元素的时间都是常数；称为*随机访问 random access*。但是，有些算法对于排序文件和链表也是很有用的。当集合仅以顺序方式访问时，我们使用术语序列 sequence，来强调集合的结构可能是顺序文件或是链表，也可以是数组。如果数组的索引定义为 0,…,n-1，则数组的*范围 range*或*子范围 subrange*是指两个给定索引之间连续的数组项，一般写做 first 和 last，0≤first，last≤n-

1。如果 last<first，称范围是空。

我们假设集合中每一个要排序的元素都包含一个标识，叫做*关键字 key*，这是一个线性集合的元素，两个关键字可以比较以决定他们哪一个大或是两者相等。我们总是以非递减顺序排序关键字。集合中的每一个元素都可以包含除 key 之外的信息。当在排序过程中，key 重排列之后，相应的信息也会跟着重排列，但是有时我们仅关心 key，而不显示的给出数组项剩下的信息。

4.2 小节到 4.10 小节考虑的算法都是只通过比较关键字（和复制他们）进行排序的一类算法，这类算法必须不对关键字做其他操作。我们称这些是“通过比较关键字排序的算法”，或简称“基于比较的算法”。分析这类算法主要的工作是计算比较关键字的次数。在 4.7 小节建立了这类算法需要的比较次数的低限。4.11 小节讨论可以对关键字使用其他的排序算法，以及不同的度量工作量的方法。

本章的算法叫做*内部排序 internal sorts* 因为假设数据都在计算机高速随机访问的主存中。当集合数据量过大而不能放在主存中时性能问题的焦点就有所不同。对存储在外部、低速存储设备中且对数据的访问有约束的数据集合排序算法称为*外部排序 external sorts*。参看本章后面的 Notes 和 References 找到这类算法更多的信息。

当分析排序算法时，我们将考虑他们将使用多少额外空间（超过输入数据的那一部分）。如果额外空间的大小相对与输入是一个常数，算法称为 *in place*。

为了帮助算法更清楚，我们使用 **Element** 和 **Key** 作为类型标识符，但是认为 **Key** 是一个数值类型，我们对 Key 使用“=,<,>”等关系运算符。当书中使用类似“E[i].key<x”关键字比较表达式时，如果实际的类型是非数值的（例如 **String**），Java 程序提供了语法来调用方法，比如“less(E[i].key, x)”。许多语言都是这样。

> *Java 语言提示*：通过 Java 中的 Comparable 接口，可以写一个能比较各种关键字类型的过程，类型 Key 必须体换成关键字 Comparable。细节在附录 A 给出。复习一下 Java 中的数组申明为 Element[] arrayName。

4.2 插入排序

从插入排序开始是一个很好的选择，因为它的思想是很自然和很一般的，而且计算它的最坏时间和平均行为时间也很简单。它也是我们在第 4.10 小节讨论的更快的算法要用到的部分。

4.2.1 策略

我们从有 n 个元素以任意序列的 E 开始，如图 4.1。（）插入排序可以用于关键字是任意线型顺序集合的情况，但是对于小棍的插图，将棍的高想象成关键字。）

假设我们已经将序列的一些小段排序了。图 4.2 显示了一个左边的 5 个已经排完序的快照。下一步就是通过把剩下的元素插入到它正确的位置来增加已排序列的长度。

令 x 是要插入已排序列的下一个元素，就是说，x 是未检查段的最左边元素。第一步我们把 x“搬走”（就是将它拷贝到局部变量），在 x 的位置留一个*空位*。然后我们依次将 x 与空位左边的元素比较，只要 x 比左边的元素大，我们就这个元素搬到空位，留下一个新的空位。就是说，空位向左移动了一步。重复这个过程直到，我们没有元素移到空位中，或是当前空位左边的元素小于或等于 x。将 x 插入空位，如图 4.3。为了让算法开始，我们只需要

将第一个元素看作是一个有序段就可以了。现在我们将这些步骤规范化成移个过程，我们假定序列是一个数组；当然对于 list 或其他的线性序列结构也是可以的。

图 4.1 未排序元素

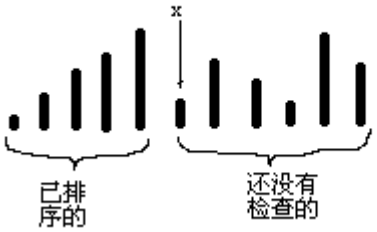


图 4.2 部分排序元素

图 4.3 将 x 插入道正确的位置

int shiftVac(Element[] E, **int** vacant, Key x)

Precondition: vacant 是非负的。

Postcondition: 令 xLoc 是返回的值。则：

1. E 中的索引小于 xLoc 的元素在它原来的地方，且 key 都小于或等于 x。
2. E 中的元素位置在 xLoc+1 直到 vacant 的，大于 x 的元素都将在 shiftVac 调用之后把位置移动一格。

图 4.4 shiftVac 的规范

4.2.2 算法和分析

现在我们仔细的实现排序过程。让子例程 shiftVac(E, vacant, x)做搬运的工作，直到把空位移动到按顺序 x 应该在的地方。过程返回空位的索引，叫做 xLoc。Precondition 和 postcondition 都在图 4.4 中。换句话说，shiftVac 完成图 4.2 到图 4.3 的转换。现在 insertionSort 可以一直调用 shiftVac，使得左边的段越来越长，直到所有的元素都在有序段中。

shiftVac 过程是一个典型的*普通搜索例程*（定义 1.12）。如果没有数据了，fail；否则查

找一个元素，如果它是我们要找的元素，成功；否则继续没有检查过的元素。因为有两种结束情况，可以用一个 **while** 循环来表示，再使用 **break** 对于一种或多种结束情况。递归形式是直接的。

```
int shiftVacRec(Element[] E, int vacant, Key x)
{
    int xLoc;
1:  if(vacant==0)
2:      xLoc= vacant;
    else
3:      if(E[vacant-1].key<=x)
4:          xLoc=vacant;
5:      else
        {
6:          E[vacant]= E[vacant-1];
7:          xLoc=shiftVacRec(E, vacant-1,x);
        }
8:  return xLoc;
}
```

为了验证第七行的递归的正确性，我们注意到递归过程在一个更小的范围内工作，而且它的第 2 个元素是非负的，所以是满足 precondition（图 4.4 中的）的。（你可以检查调用链来论证 vacant-1 为什么是非负的——它为什么不是负的呢？）正确性是显然的，如果我们可以假定第七行的递归完成了它的目标的话。

尽管 shiftVacRec 过程非常简单，如果我们想象一下 E 的第 n 个元素插入时候的活动跟踪情况，算一下递归的深度或是堆栈的深度可能增长到规模 n。这对于很大的 n 是不可接受的。因此这种情况下，当让所有的事情都工作正常之后，递归必须改成迭代。（试图优化不能工作的程序是徒劳。）目的与其说是为了节约时间不如说是为了节约空间。实际上如果打开优化，许多编译器将自动执行这个转换过程。下面的完整算法把包含了 shiftVac 的迭代版本。

算法 4.1 插入排序

Input：数组 E，数组的大小。索引从 0 到 n-1。

Output：E，其元素是升序。

Remark：shiftVac 子例程的规范在图 4.4 中给出。

```
void insertionSort(Element[]E, int n)
{
    int xindex;
    for(xindex=l; xindex<n; xindex++)
    {
```

```
            Element current= E[xindex];
            Key x=current.key;
            int xLoc= shiftVac(E, xindex, x);
            E[xLoc]=current;
        }
        return;
    }

    int shiftVac(Element[]E, int xindex, Key x)
    {
        int vacant, xLoc;
        vacant= xindex;
        xLoc=0; //  假定失败
        while（vacant>0）
        {
            if(E[vacant-1].key<=x)
            {
                xLoc=vacant; //  成功
                break;
            }
            E[vacant]=E[vacant-1];
            vacant--;
        }
        return xLoc;
    }
```

最坏情况复杂度

为了分析，我们使用 i 代表 xindex。对于每一个 i 的值，关键字比较的最大次数是 i（在一次 shiftVac 的迭代调用中，或是 shiftVacRec 的顶层递归 中）。则总数是

$$W(n) \le \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

注意，我们已经建立了最坏情况的上界；花点时间来验证对输入确实有 n(n-1)/2 次关键字比较。一种最坏情况是当关键字是颠倒顺序的时候（就是说是递减的时候）。所以

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

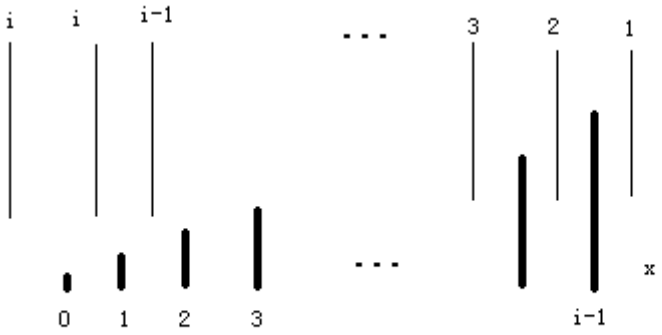


图 4.5 决定 x 的位置需要的比较次数

平均行为

我们假定输入关键字的所有排列都是等概率的。我们首先将决定将一个新关键字插入到原来已经排序好的片段中需要多少次关键字比较，就是以任意 i（用于 xindex）调用一次 shiftVac 会有多少次比较。为了简化分析，我们假设关键字是不同的。（分析过程十分类似与第一章做对序列查找做的分析。）

有 i+1 个位置可能放置 x。图 4.5 展示了插入的位置决定了要做多少次比较。

x 属于任一位置的概率是 1/(i+1)。（这依赖于一个事实，即算法之前不知道 x 的任何信息。如果算法事先知道关于 x 的信息，，我们不必假设 x 对于每一个 x 的值都有相等的概率。）因此在 shiftVac 中找到第 i 个元素的平均比较次数是

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

现在对于所有的 n-1 次插入，

$$A(n) = \sum_{i=1}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - 1 - \sum_{j=2}^{n}\frac{1}{j},$$

这里我们带入 j=i+1 得到最后的和。我们从等式（1.16）可以看出 $\sum_{j=1}^{n}(1/j) \approx \ln n$，我们可以忽略 1 preceding the sum to make the lower limit j=1。忽略低阶项，我们有

$$A(n) \approx \frac{n^2}{4} \in \Theta(n^2)$$

空间

显然，插入排序是一个 in-place 排序方法，当使用迭代版本的 shiftVac 时。当使用递归版本时，frame 堆栈可能扩展到 Θ(n)。

4.2.3 特定排序算法的行为低限

考虑一个元素得到关键字是 x，当插入排序比较 x 它左边的关键字时，他恰好占据了“vacant”位置。则在每次比较之后，插入排序都不移动元素，或者这是简单的交换两个相邻的元素。我们将展示所有的有限制排序算法——即每次比较之后“局部”移动元素的那些算法——必须做和插入排序工作量类似的工作。

N 的元素的排列可以描述为一个一对一的集合 N={1, 2, …, n}到自己的函数。对于 n 个元素有 n！种排列。令在未排序序列 E 中的元素是 $x_1, x_2, \ldots, x_n$。为了在这里简化符号，让我们假设排序之后的元素存储在 1, …, n 而不是 0, …, n-1。排列 π，在 1≤i≤n 时，π(i)是 x_i 在排序完成之后的位置。不失一般性，我们可以假设关键字是整数 1, 2, …, n，这样我们就可以认为最小关键字是 1，第二小的是 2，以此类推。则未排序的输入是 π(1), π(2), …, π(n)。例如，考虑输入序列 2, 4, 1, 5, 3。π(1)=2 意味着第一个关键字 2 属于第 2 个位置。π(2)=4，因为第 2 个关键字属于第 4 个位置，如此类推。我们将以排列 π 标识序列 π(1), π(2), …, π(n)。

排列 π 的一个*倒置*（*inversion*）是一个对（π(i), π(j)）有 i<j 而 π(i)> π(j)。如果（π(i), π(j)）是一个倒置，序列中的第 1 个和第 j 个关键字不符合他们的顺序。例如排列 2, 4, 1, 5, 3 有 4 个倒置(2, 1), (4, 1), (4, 3)和(5, 3)。如果排序算法移除在每次比较之后至少移除一个倒置（也就是交换相邻元素），则在输入 π(1), π(2), …, π(n)上执行的比较次数至少是 π 的导致个数。所以我们考虑倒置。

容易得到对于一个排列有 n(n-1)/2 个倒置。（哪一个？译注：逆序时）因此任何每次比较移除至少一个倒置的排序算法最坏行为肯定是 Ω(n²)。

为了得到这样的排序算法关键字比较的平均行为的底限，我们计算排列中倒置的的平均数。每一个排序 π 有一个*转置排列*（*transpose permutation*）π(n), π(n-1), …, π(1)。例如，2, 4, 1, 5, 3 的转置是 3, 5, 1, 4, 2。每一个排列有一个唯一的转置而且当 n>1 时两者是不同的。令 i 和 j 是 1 到 n 之间的整数，假设 j<i。则（i, j）是一个排列 π 的倒置，但是对于 π 的倒置来说就不是。有 n(n-1)/2 对整数。因此每对排列有 n(n-1)/2 倒置，因此平均是 n(n-1)/4。综上所述，排列的倒置平均数是 n(n-1)/4，我们证明下面的定理。

> **定理 4.1** 任意通过比较关键字排序的算法，如果在每一次比较之后至少减少一个逆序，则在最坏情况下至少做 n(n-1)/2 次比较，平均情况下至少做 n(n-1)/4 次比较（当有 n 个元素时）。

既然插入排序在最坏情况下做 n(n-1)/2 次关键字比较，在平均情况下近似 $n^2/4$，它已经是做“局部”交换的算法中最好的了。当然肯定有更好的策略，但是如果存在更快的算法，他们必须一次移动元素超过一个位置。

4.3 分而治之

分而治之算法设计范例背后的原理是，通常情况下解决几个小型要比解决一个大型问题要容易。4.4 小节到 4.8 小节的算法都使用了分而治之的逼近。他们将一个问题*分解*成许多同样类型的小问题（本章中是分成更小的集合再排序），之后递归的*解决*这些小问题（以同样的方法），最后*合并*以得到原始输入的解决。为了终止递归我们，直接解决一些小的问题。但是，插入排序仅仅分解了一个元素创建了一个子问题。

我们已经看过了分而治之的一个主要的例子——二分查找（1.6 节）。将主要的问题分解成两个子问题，其中一个甚至不需要去解决。

一般的，我们可以通过图 4.6 中的框架过程来描述分而治之算法。

为了设计一个分而治之算法，我们必须设定子例程 directlySolve，divide 和 combine。小的实例的数量分解到 k。对于输入规模 n，令 B(n)是执行 directlySolve 的步数，令 D(n)是执行 divide 的步数，令 C(n)是执行 combine 的步数。则当基本情况 T(n)=B(n)描述算法工作量的一般形式的递归等式

$$T(n) = D(n) + \sum_{i=1}^{k} T(size(I_i)) + C(n) \qquad \text{对于 n>smallSize}$$

当 n≤smallsize 时，是基本情况 T(n)=B(n)。对于许多分而治之算法而言，分割和合并的步骤都很简单，T 的递归等式比通用形式要简单。Master 定理（定理 3.17）给出了分而治之递归等式的一般解决方法。

```
solve(I)
    n=size(I);
    if( n≤ smallsize)
        solution=directlySolve(I);
    else
    {
        divide I into I1, …, Ik.
        For each i∈{1, …, k}:
            Si =solve( Ii);
        solution =combine (S1, …, Sk);
    }
    return solution;
```

图 4.6 分而治之骨架

接下来几节介绍的快速排序和归并排序，两者的区别就在他们分割问题和合并解的方式不一样，或者说是排序子集的方式不一样。快速排序的特点是“艰难的划分，简单的合并”，而归并排序是“简单的划分，艰难的合并”。除了过程调用的簿记工作外，我们将看到所有的“实际工作”都在“艰难的”节中。两个排序过程都有子例程做他们“hard”部分，这些子例程在他们自己的 **rights** 内是非常有用的。对于快速排序重点是 **partition**，它是通用 framework 中 **divid** 步骤；**combin** 步骤没什么内容。对于归并排序重点是 **merge**，它是 combin 步骤；divide 步骤则是简单的计算。两个算法划分问题成两个子问题。但是，对于归并排序，两个子问题是等规模的（可能差一个元素），而对于快速排序，甚至子划分都是不能保证的。这个不同导致了两者性能上的明显不同，在两个算法各自的分析中有详细 discovery。

从顶层看，堆排序（4.8 节）不是一个分而治之算法，但是它使用的堆的操作时分而治之策略的。堆排序的加速形式使用了更完善的分而治之算法。

在后面的章节中，分而治之策略将会出现许多次。在第 5 章，应用分而治之查找集合的中间元素。（通用问题称为 selection 问题）。在第 6 章，我们使用分而治之生成二叉查找树，以及它的平衡版本红黑树。在第 9 章，我们使用它解决它的路径问题，比如传递闭包。在第 12 章，我们用它解决许多矩阵和矢量问题。在第 13 章，图的着色问题也用到分而治之。第 14 章中，分而治之在并行计算中以另一种形式出现。

4.4 快速排序

快速排序是最早发现的分而治之算法；它由 1962 由 C.A.R.Hoare 发表。 直到现在它也是实际使用中最快的算法之一。

4.4.1 快速排序的策略

快速排序的策略是将元素重新排列，使得数组中“小”的关键字都在“大”的关键字前面（“难度分解”）。之后 Quicksort 递归的排列“小”关键在和“大”关键字子范围，最后整个数组就有序了。对于数组来说，在“合并”（combination）步骤没有什么要做的，但是 Quicksort 也可以用于 list（参见练习 4.22），在这种情况下“合并”步骤连接两个 lists。我们为了简单，描述的是数组的实现。

令 E 是一个数组，令 first 和 last 是当前 Quicksort 要排列子范围的的第一个和最后一个元素。最开始的时候 first=0，last=n-1，n 是数组元素的个数。

Quicksort 算法从它要排序的子范围中选择一个元素，叫做 pivot 元素，它的关键字叫做 pivot，然后将 pivot 元素移到一个局部变量中，在数组中留一个空位。暂时，我们假设子范围最左边的元素就是 pivot 元素。

Quicksort 将 pivot 传递给 Partition 子例程。这个子例程重排列剩下的元素，找到 splitPoint 的索引，splitPoint 满足：

1. 对于 first≤i<splitPoint，E[i].key<pivot；
2. 且对于 splitPoint<i≤last，E[i].key≥pivot。

注意现在有一个空位在 splitPoint 的位置。

之后 Quicksort 将 pivot 元素放回 E[splitPoint]，这就是它正确的位置，pivot 元素在后面的递归调用中不在参与，就是它正确的位置了（参见图 4.7）。这个就完成了“分解“的过程，然后 Quicksort 开始递归调用自己解决由 Partition 创建的子问题。

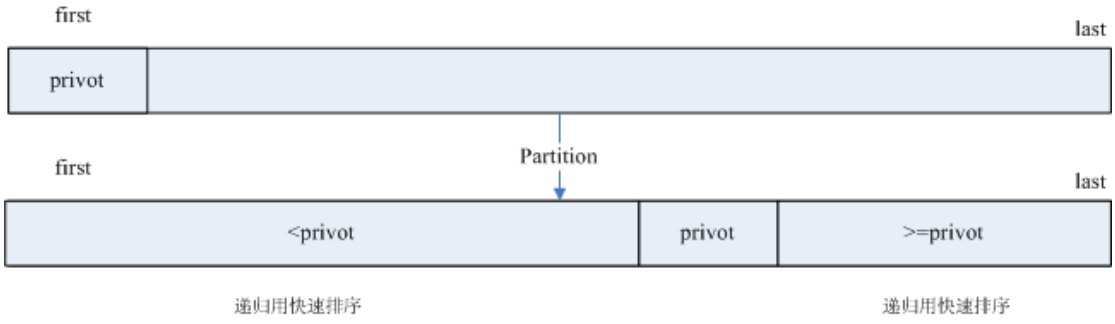


图 4.7 快速排序

Quicksort 过程可能选择 E[first]和 E[last]中的任意一个元素来做 partition，这作为一个预处理步骤。无论那个元素被选择，先移动到局部变量 pivot 中，如果它*不是* E[first]，则 E[first]移动到它的位置，保证 E[first]在 Partition 调用时是一个空位。其他的选择 pivot 的策略在 4.4.4 小节中有叙述。

算法 4.2 快速排序

输入：数组 E 和索引 first，last，其中元素 E[i]定义为 first≤i≤last。

输出：E[first]… E[last]是一个排序好的数组，和原来的数组元素相同。

```
void quickSort( Element[] E, int first, int last)
{
    if( first<last)
    {
        Element pivotElement =E[first];
        Key pivot = pivotElement.Key;
        int splitPoint = partition( E, pivot, first, last);
        E[splitPoint]= pivotElement;
        quickSort(E, first, splitPoint -1);
        quickSort(E, splitPoint+1, last);
    }
    return;
}
```

4.4.2 Partition 子例程

所有比较和移动元素的工作都在 Partition 子例程中。Partition 有许多不同的策略可以使用；不同的策略带来不同的优点和缺点。我们在这里展示一个，在练习中留下另一个。策略的关键在于如何完成元素的重排列。一个非常简单的解决方案是降元素移动到一个临时数组中，但是这样做的问题是如何将他们重新到正确的位置。

我们这里描述的 partition 方法本质上来说是 Hoare 描述的方法。As motivation, remember that the lower bound argument in Section 4.2.3 showed that, to improve on Insertion Sort, it is necessary to be able to move an element many positions after one compare. 这里空位初始时在 E[first]。这里我们想要小的元素都在左边的范围里面，而且我们想要元素移动的距离都尽可能的长，所以很自然的我们从 E[last]开始查找小元素（小于 pivot 的元素）。找到一个的时候，我们将这个元素移动到空位（现在是 first）。这时将在小的那个元素的位置留下一个空位；我们称它为 highVac。这种情况展示在图 4.8 的头两个数组图中。

我们知道所有索引大于 highVac（直到 last）的元素都是大于或等于 pivot 的。可能的话，

某个其他的大于 pivot 的元素应该移动到 highVac。同样的，我们希望元素移动的距离尽可能的长，于是很自然的从 first+1 查找大的元素。当我们找到的时候，我们移动这个元素到空位（现在是 highVac），之后会留下一个新空位，我们称它是（lowVac）。我们知道所有索引小于 lowVac（从 first 开始）的元素小于 pivot 的。

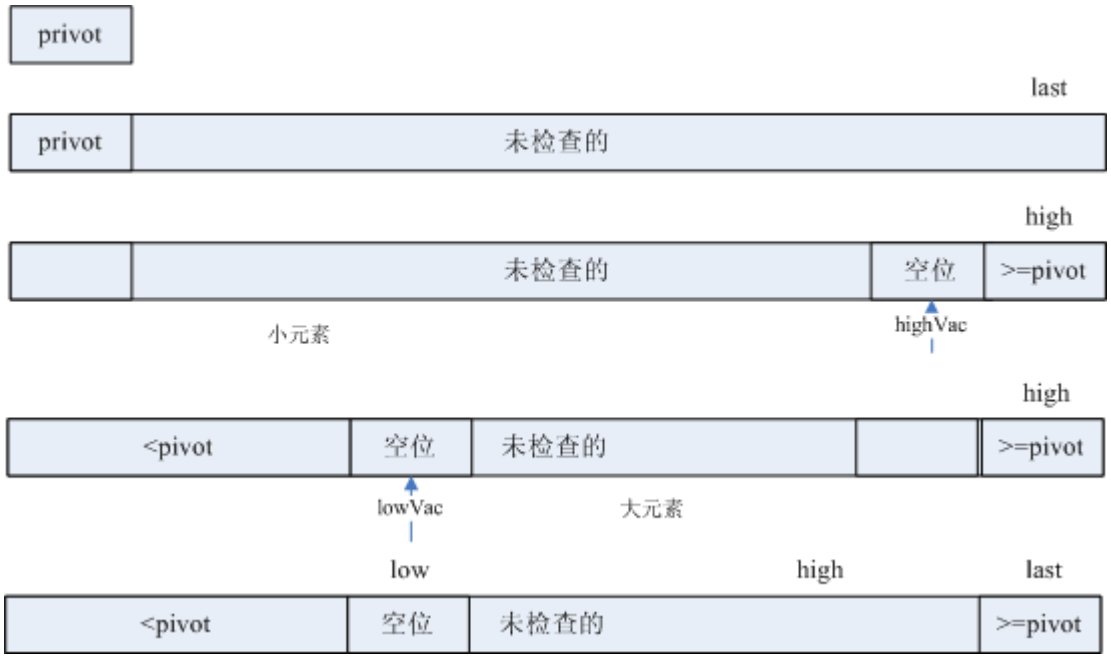


图 4.8 Partition 第一个周期的过程

最后我们更新 low 和 high 变量准备下一次循环，如图 4.8 最后一行显示的。在第一个循环开始的时候，从 low+1 到 high 之间的元素还没有被检查，E[low]是空位。我们可以按刚才描述的方式重新开始，在 high 之后查找小的元素，将它移动到 low vacancy，之后从 low+1 开始往前找一个大的元素，将它移动到 highVac，在 lowVac 的位置创建一个新的空位。最后 lowVac 和 highVac 碰到一起了，意味着所有的元素都和 pivot 比较过了。

Partition 过程以刚才描述的循环实现，使用子例程以方便组织代码。子例程 extendLargRegion 扫描右边，越过大的元素直到遇到一个小的元素，遇到之后将小的元素移动到空位返回，或者直到空位而没有一个小的元素。在后一种情况下，partition 就完成了。在前一种情况下，返回新的空位的位置，调用第二个子例程。子例程 extendSmallRegion 类似，除了是向左搜索，以及越过小的元素找到大的元素。

一开始，small-key 范围（low 的左边）和 large-key 范围（high 的右边）都是空的，空位是 middle-region 的最左边（在此刻，它是整个数组）。每一次调用子例程 **extendlargRegion** 或是 **extendSmallRegion**，都会将 middle-region 至少缩减 1 个，然后将空位留在 middle-region 的最后。两个子例程还保证每次只有一个小的元素进入 small-region，一个大的元素进入 large-region。这可以参考他们的 postcondition。当 middle-region 缩减到只有一个位置时，这个位置就是空位，作为 splitPoint 返回。下面的问题作为一个联系：在 **partition** 的 **while** 循环中，middle region 的两个边界是什么，最终空位在什么地方？尽管 Partition 过程可以“make do”只用更少的变量，但是我们定义的每一个变量都有它自己的含义，在练习中有简化的版本。

为了避免在 partition 的 while 循环中额外的比较，在第 5 行没有测试 highVac=lowVac，这将预示所有的元素都将被 partitioned。因此，当循环结束的时候 high 可能只比 low 小 1，而从逻辑上来说应该相等。然而，high 在循环结束后不你呢个 accessed，所以这个差异是无

害的。

图 4.10 展示了一个小的例子。Partition 的细节操作只在它第一次调用时展示。注意小的元素积累到 low 的左边，而大的元素积累到 high 的右边。

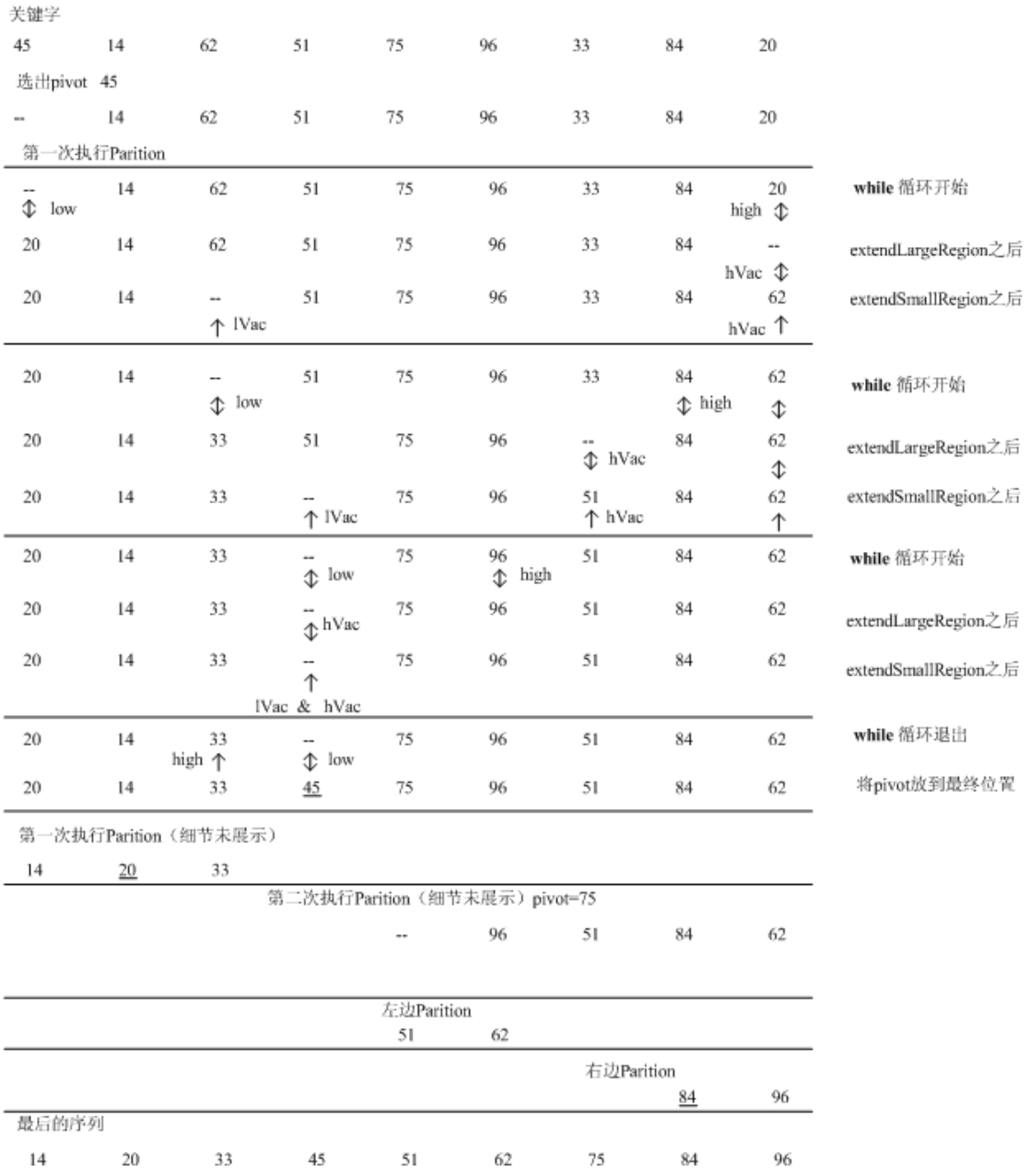


图 4.10 快速排序例子

算法 4.3 Partition

输入：数组 E，partition 的轴 pivot，索引 first 和 last，元素 E[i]定义为 first+1 ≤i≤last，且 E[first]是空位。假设 first<last。

输出：令 splitPoint 是返回值。原来在 first+1, …, last 之间的元素被重新排列成两个子范围：

1. E[first], …, E[splitPoint-1]之间元素的关键字都小于 pivot，而

2. E[splitPoint+1], …, E[last]之间元素的关键字都大于等于 pivot。

还有，first≤splitPoint≤last，E[splitPoint]是空位。

过程：参看图 4.9。

```
int partition(Element[] E, Key pivot, int first, int last)
{
    int low, high;
1.  low=first, high=last;
2.  while(low<high)
    {
3.      int highVac= extentLargeRegion(E,pivot, low, high);
4.      int lowVac=extendSmallRegion(E, pivot, low+1, highVac);
5.      low= lowVac; high=highVac-1;
    }
6.  return low; // 这是 splitPoint;
}

/** Postcondition for extendLargeRegion:
  *E[lowVac+1], …, E[high]最右边的关键字<pivot 被移动到 E[lowVac]
  *被移动元素的索引作为返回值。如果不存在这样的元素，lowVac 被返回。
  */
int extendLargeRegion(Element[] E, Key pivot, int lowVac, int high)
{
    int highVac, curr;
    highVac=lowVac; // 考虑没有 key<pivot 的情况
    curr=high;
    while(curr>lowVac)
    {
        if(E[curr].key<pivot)
        {
            E[lowVac]=E[curr];
            highVac=curr;
            break;
        }
        curr--; // 簿记
    }
    return hgihVac;
}

/** Postcondition：（练习）*/
int extendSmallRegion(Element[] E, Key pivot, int low, int highVac)
{
    int lowVac, curr;
```

```
    lowVac= highVac;
    curr=low; //考虑没有 key>=pivot 的情况
    while(curr< highVac)
    {
        if(E[curr].key>= pivot)
        {
            E[hgihVac]=E[curr];
            lowVac=curr;
            break;
        }
        curr++;   // 簿记
    }
    return lowVac;
}
```

图 4.9 算法 4.3 的过程

4.4.3 快速排序的分析

最坏情况

Partition 把每一个关键字与 pivot 比较，所以如果在它工作的范围有 k 个位置，它将做 k-1 次比较（第一个位置是空位）。如果 E[first]是要分割范围最小的元素，则 splitPoint=first，这将导致分割成一个空的子范围（没有元素比 pivot 小）和一个 k-1 个元素的子范围。因此所以，每次 Partition 调用时如果 pivot 是最小的元素，则总的比较次数是

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

这与插入排序和 Max 排序一样坏。而且奇怪的是，最坏情况在关键字已经有序的时候发生！快速排序是不是有点虚假广告的味道？

平均行为

在 4.2.3 小节里面我们展示了如果排序算法每次比较至少从排列中移除一个倒置，则它在平均情况下至少要做$(n^2-n)/4$ 次比较（定理 4.1）。但是快速排序没有这个限制。Partition 算法可以移动一个元素穿过整个数组，一次移动消除 n-1 个倒置。快速排序因为它的平均行为得到它的名字。

我们假设关键字是不同的，且所有关键字的排列都是等概率出现。令 k 是数组要排序范围中的元素的个数，令 A(k)是这个规模排序要做的平均关键字比较次数。假设下一次 Partition 将 pivot 移动到了本次子范围的第 i 个位置（图 4.11），从 0 开始计数。Partion 做了 k-1 次关键字比较，下一次要排序的子范围分别有 i 个元素和 k-1-i 个元素。

在 Partiition 完成之后对于我们的分析是很重要的，子范围（first, … splitPoint-1）内没有两个关键字相互进行过比较，所以这个子范围内所有的排列依然是等概率的。同理

（splitPoint+1, …, last）也是一样。这一条有下面的递归。

splitPoint 每种可能的位置都是等概率的（1/k），令 k=n，我们有递归等式

$$A(n) = n - 1 + \sum_{i=0}^{n-1} \frac{1}{n}(A(i) + A(n-1-i)) \quad \text{对于 } n \geq 2$$

$$A(1) = A(0) = 0$$

检查和中的项以简化递归等式。形如 A(n-1-i)的项 run from A(n-1) down to A(0)，所以他们的和以 A(i)的和同样。则我们可以丢弃 A(0)项，的得到

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \qquad \text{对 } n \geq 1 \tag{4.1}$$

这是一个比我们前面看到的要复杂的递归等式，因为 A(n)的值依赖于早先的值。我们可以尝试使用一些技巧来解这个等式，或者我们可以猜一个解然后用归纳法证明这个解正确。后一种技术特别适合递归算法。学习两种方法都是有益的，所以两种方法我们都会做。

为了猜 A(n)的形式，让我们考虑一种快速排序工作的最好的情况。假设每次 Partition 执行都将原来的范围分成两个等长的子范围。既然我们只是想从侧面猜测快速排序在平均情况下有多块，我们将子范围的大小近似为 n/2 而不管这是不是一个整数。比较的次数描述为：

$$Q(n) \approx n + 2Q(n/2)$$

可以采用 Master 定理了（定理 3.17）：b=2，c=2，E=1，f(n)=n^1。因此 Q(n)∈ Θ(nlogn)。所以如果 E[first]每次都接近 split 位置，则快速排序的比较次数将在 Θ(nlogn)。这比 Θ(n^2)有显著的提高。但是如果关键字所有的排列都是等概率出现的话，还会得到这么好的结果吗？我们将证明之。

定理 4.2 令 A(n)由递归等式（4.1）定义。则，对于 n≥1，A(n)≤cnlnn，c 为常数。（注意：我们已经切换到自然数算法以简化证明中的一些计算。c 的值将在证明中找到。）

证明 证明归纳要排序的元素个数 n。基本情况是 n=1。我们有 A(1)=0，c1ln1=0。

当 n>1 时，假设 A(i)≤ciln(i)，对于 1≤i<n，c 是某个常数。通过等式（4.1）和归纳假设。

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \leq n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} ci\ln(i)$$

我们可以用积分划定边界（参看等式 1.16）：

$$\sum_{i=1}^{n-1} ci\ln(i) \leq c\int_1^n x\ln x dx$$

使用 1.3.2 小节给出的等式（1.15）有

$$\int_1^n x\ln x dx = \frac{1}{2}n^2\ln(n) - \frac{1}{4}n^2$$

所以

$$A(n) \le n-1+\frac{2c}{n}(\frac{1}{2}n^2\ln(n) - \frac{1}{4}n^2) = cn\ln n + n(1-\frac{c}{2}) - 1$$

为了满足 A(n)≤cnlnn，第二项和第三项应该是负数或者 0。对于 c≥2 的情况，第二项小于等于 0。所以我们可以令 c=2 并得出结论 A(n)≤2nlnn。

对于 c<2 的情况可以做类似的分析 A(n)>cnlnn。既然 lnn≈0.693lgn，我们有：

推论 4.3 在平均情况下，假设所有的输入的排列是等概率，快速排序（算法 4.2）做的平均比较次数对于大的 n，近似为 1.386nlgn，n 是输入集合的规模。

*更精确的平均行为

尽管我们已经建立了快速排序的平均行为，回到递归等式（等式 4.1）依然是有益的，尝试直接解这个等式可以得到更多的 leading term。这一小节使用一些复杂的数学手段，可以跳过本节不会影响学习的连续性。

由等式（4.1）我们有

$$A(n) = n-1+\frac{2}{n}\sum_{i=1}^{n-1}A(i) \qquad (4.2)$$

$$A(n-1) = n-2+\frac{2}{n-1}\sum_{i=1}^{n-2}A(i) \qquad (4.3)$$

如果我们用（4.2）减去（4.3），大多数项都约掉了。既然和需要乘上不同的因子，我们需要一个 slightly more complicated bit of algebra。非正式的，我们计算

n×等式（4.2）-（n-1）×等式（4.3）

于是

$$nA(n) - (n01)A(n01) = n(n-1) + 2\sum_{i=1}^{n-1}A(i) - (n-1)(n-2) - 2\sum_{i=1}^{n-2}A(i)$$

$$= 2A(n-1) + 2(n-1)$$

有

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

现在令

$$B(n) = \frac{A(n)}{n+1}$$

对于 B 的递归等式是

$$B(n) = b(n-1) + \frac{2(n-1)}{n(n+1)} \qquad B(1) = 0$$

引用等式（1.11），我们过程留给读者验证

$$B(n) = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\sum_{i=1}^{n} \frac{1}{i} - 4\sum_{i=1}^{n} \frac{1}{i(i+1)}$$

$$\approx 2(\ln n + 0.577) - 4n/(n+1)$$

因此

$$A(n) \approx 1.386 n \lg n - 2.846 n \text{。}$$

空间使用

第一眼看去似乎快速排序是一种 in-place 排序。但是不是。当算法在子范围上工作时，所有其他范围的起至范围都要存放在 frame stack 中（译注：递归调用带来的），栈的规模依赖于子范围的数量。当然就是依赖于 n。在最坏情况下，Partition 每次分隔一项；递归的深度是 n。因此最坏情况下的空间使用数量是 Θ(n)。下面描述的一个算法的改进能显著的减少栈的尺寸。

4.4.4 基本快速排序算法的改进

选择枢轴

我们已经看到如果 Partition 使用的 pivot 能分成两个基本一样大小的段时，快速排序就会很好。（它的位置是 Partition 返回的 splitPoint。）选择 E[first]作为 pivot 时可能导致快速排序很糟糕，而本来这时排序会是很简单的，比如当数组已经是有序的情况。有许多其他的选择 pivot 的策略。一种是选一个 first 和 last 之间的随机整数 q，令 pivot=E[q].key。另一个是另 pivot 是 E[first], E[(first+last)/2], E[last]的中间一个。（两种情况下，在 Partition 之前 E[first]元素都将先和 pivot 元素交换。）两种策略都需要在选择 pivot 时做一点额外工作，但是他们会减少快速排序程序的平均运行时间。

可选的 Partiton 策略

文中展示的 Partition 版本在平均情况下做最少的元素移动，与其他 partitioning 策略比较。这个子例程比较清晰，and coding these in-line would save some overhead；但是有些编译器可以自动做这种优化。其他优化考虑在本章后面的 Notes 和 References 中提及。在练习中有一个可选版本，连续中的版本易于理解和编程，但是稍慢。

小集合排序

实际应用中快速排序对于小的集合效果不佳，因为有太多的过程调用。但是，快速排序算法对于大到 n 的规模也会分解成小的子集合再递归排序他们。因此，当子集合小的一定程度时，算法就变的低效率了。这个问题可以通过下面的方法来补偿，选择一个小的 smallSize，当集合的大小小于或等于 smallSize 的时候采用简单非递归的排序，在改进过的算法中调用了 smallSort。（此时插入排序是个不错的选择。）

```
quickSort(E, first, last)
{
    if( last –first)>smallsize)
    {
        pivotElement=E[first];
        pivot=pivotElement.Key;
        int splitPoint=partition(E, pivot, first, last);
        E[splitPoint]= pivotElement;
        quickSort(E, first, splitPoint-1);
        quickSort(E, splitPoint+1, last);
    }
    else
        smallSort(E, first, last);
}
```

这个方案的一个变体是跳过调用 smallSort。则当快速排序退出时，数组不是有序的，但是所有的元素都只需要移动小于 smallSize 的位置就能到达它正确（排序好的）的位置。（为什么？）因此作为后处理，进行一次插入排序将是非常有效的，而且不会像前面那个版本中的 smallSort 调用做很多相同的比较。

smallSize 该取多大呢？最好的选择依赖于算法具体的实现（也就是使用的计算机，以及程序的细节），因而我们可以在 overhead 和比较之间做一下取舍。10 是一个合理的值。

堆栈空间优化

我们观察到 Quicksort 的递归调用深度可能很深，最坏情况下可能达到 n（在 Partition 每次之分解一个元素的时候）。许多 pushing 和 poping 的都是不必要的。Partition 之后，程序开始排序子范围 E[first]….E[solitPoint-1]；之后它必须排序 E[splitPoint+1]…E[last]。

第二个递归调用是过程的最后一句话，所以可以用我们前面在插入排序中用于 shiftVac 的方式将第二个递归转换成迭代。第一个递归依然在那里，所以只是部分的消除了递归。

尽管只有过程中只有一个递归调用了，我们仍然要关心递归深度。当每次递归调用只能比前一次稍微小一点时，递归深度就是个问题。所以我们使用的第二个技巧是避免递归在大的子范围做递归。通过保证每次递归调用在接近它的“父”递归调用一半的子范围上，递归深度保证维持在 lgn。两个 ideal 合并在下面的版本中，“TRO”表示“尾递归优化（tail recursion optimization）”。想法是在每次 partition 之后，下一次递归将在小的那个子范围上，大的子范围将直接用 while 循环处理。

```
quickSortTRO(E, first, last)
{
```

```
        int first1, last1, first2, last2;

        first2= first; last2= last;
        while( last2- first2 >=1)
        {
            pivotElement = E[first2];
            pivot= pivotElement.Key;
            int splitPoint=partition(E, pivot, first2, last2);
            E[splitPoint]= pivotElement;
            if(splitPoint<= (first2+ last2)/2)
            {
                first1=first2; last1=splitPoint -1;
                first2= splitPoint+1; last2 =last2;
            }
            else
            {
                first1= splitPoint +1; last1= last2;
                first2= first2;    last2 =splitPoint-1;
            }
        }
        return;
    }
```

结合使用这些改进手段

前面我们独立的讨论几种修改方法，但是他们可以协调的组合在一个程序中。

Remarks

实际中，快速排序程序对大的 n 平均而言运行的很快，是广泛使用的一种排序方法。但是在最坏情况下，快速排序很糟糕。类似插入排序（参看 4.2 小节），Maxsort 和冒泡排序（练习 4.1 和 4.2），快速排序最坏情况是 $\Theta(n^2)$，但是与其他不同，快速排序平均行为在 $\Theta(n\log n)$。有排序算法最坏情况能在 $\Theta(n\log n)$吗，或者我们可以建立最坏情况的底限就是 $\Theta(n^2)$。让我们再次检查通用的技术，看看如何改进最坏情况下的行为。

4.5 归并有序序列

在这一节中我们回顾一下下面问题的直接解决方案：给定两个升序序列 A 和 B，归并两者得到一个新的序列 C。归并有序的子序列是归并排序的关键策略。归并有序子序列同样有许多的方法，在练习中有提及。衡量归并算法的度量是关键字比较的次数。

令 k 和 m 是分别是 A 和 B 元素的数量。令 n=k+m 是“问题的规模”。假设 A 和 B 都不是空的，我们可以立即决定 C 的第一个元素：就是 A 和 B 的第一个元素中小的那个。C 剩下的部分呢？假设 A 的第一个元素是比 B 的小。则 C 剩下的部分肯定是 A 除第一个元素外

所有的元素和 B 所有元素的归并。这只是我们开始那个问题的一个小一点的版本。如果 C 的第一个元素是来自 B 的，问题是对称的。两种情况下剩下问题的规模（构造 C 剩下的部分）都是 n-1。Method99（3.2.2 小节）。

如果我们假定我们仅需要归并规模是 100 的问题，而且我们可以调用 merge99 来归并规模是 99 的问题，则问题就解决了。伪代码如下：

```
merge(A,B,C)
{
    if(A 是空)
        rest of C =rest of B
    else
        if(B 是空)
            rst of C =rest of A
        else
            if (first of A<= first of B)
            {
                first of C= first of A
                merge99(rest of A, b, rest of C)
            {
            else
            {
                first of C= first of B
                merge99(A, rest of B,rest of C)
            }
    return
}
```

现在只需要讲 merge99 换成 merge 就是一个递归解决方案了。

一旦有一个解决，我们就可以想办法讲它变成一个迭代解决方案。理想的情况是对于所有的连续数据结构可用，但是在不指明的情况下，我们的算法仅限于数组。我们在迭代每次开始的时候引入 3 个索引来跟踪”rest of A”,”reat of B”和”rest of C”。（这些索引在递归版本中是参数。）

算法 4.4 归并

输入：数组 A 有 k 个元素，B 有 m 个元素，两个数组都是升序的。

输出：C，一个数组包含 n=k+m 个来自 A 和 B 的元素，同样以升序排列。C 是传入的，算法填充它。

```
void merge(Element[] A, int k, Element[] B, int m, Element[] C)
{
    int n=k+m;
    int indexA=0, indexB=0, indexC=0;
    // indexA 是 rest of A 最开始的位置，B 和 C 类似。
```

```
        while(indexA<k && indexB<M)
        {
            if(A[indexA].key<=B[indexB].key)
            {
                C[indexC]=A[indexA];
                indexA++;
                indexC++;
            }
            else
            {
                C[indexC]B[indexB];
                indexB++;
                indexC++;
            }
        }
        if(indexA>=k)
            Copy B[indexB,…,m-1] to C[indexC,…,n-1].
        else
            Copy A[indexA,…,k-1] to C[indexC,…,n-1].
    }
```

4.5.1 最坏情况

只要 A 和 B 中的关键字有一次比较，至少有一个元素移动到 C，而且不会在检查这个元素了。在最后一次比较之后，至少有两个元素——两个参加比较的——还没有进入 C。小的那个马上移动进去了，但是现在 C 至多有 n-1 个元素，而再没有比较了。剩下的工作就是将某个数组剩下的元素移动到 C 不做任何比较。所以最多有 n-1 次比较，最坏情况下所有的 n-1 次比较，即 A[k-1]和 B[m-1]刚好是 C 的最后两个元素时。

4.5.2 归并的优化

下面我们将演示，当 k=m=/2 时在基于比较的算法的最坏情况下，算法 4.4 是最优的。就是说，对于任何基于比较的算法为了在 k=m=n/2 的时候要想所有的输入都正确，至少*存*

*在某些*输入必须要做 n-1 次比较。（这并不是说，对于*某些特定*的输入有算法要比算法 4.4 要好。）考虑完 k=m=n/2 之后，我们在看看 k 和 m 的其他关系。

定理 4.4 所有通过比较关键字归并两个含有 k=m=n/2 元素有序数组的算法，在最坏情况下至少要做 n-1 次比较。

证明 假设我们给出任意归并算法。令 a_i 和 b_i 分别是 A 和 B 的第 i 个元素。我们发现这两个关键字被选出则，算法必须比较 a_i 到 b_i（$0 \leqslant i<m$），以及 a_i 到 b_{i+1}（$0 \leqslant i<m-1$）。

4.5.3 空间使用

因为所有的元素都要拷贝到 C，从算法 4.4 看出归并总数是 n 的条目需要 2n 个元素的空间。然而有些情况下，需要额外空间的数量可能减少。比如，序列是链表。则在合并完成之后 A,B 就不需要了，A 和 B 的元素重新组成了 C。

假定输入的序列存储在数组中，且有 k≥m。如果 A 有足够的空间来放 n=k+m 个元素，则只需要 A 中的额外 m 个元素就可以了。

4.6 归并排序

Quicksort 的问题是 Partition 不能总是将数组分成两个相等的部分。归并排序将数组分成两相等的部分，再分别排序（当然是递归的）。之后它归并两个已经有序的部分（参看图 4.13）。因此，使用 4.3 节的分而治之技术，“分”仅仅是计算子范围的中间索引没有任何关键字比较；“和”是归并。

我们假定归并改成合并数组两个相邻的子范围，将合并之后的数组放回原来的空间。它的参数现在是 E、要合并的子范围的索引是 first、mid、last；排序的子序列是 E[first]…E[mid]和 E[mid+1]…E[last]，最后完成排序的序列是 E[first]…E[last]。这样修改之后，merge 子例程还是需要分配额外的空间。将在 4.5.3 小节中讨论。

算法 4.5 Mergesort

输入：数组 E 索引 first，last，E[i]（$first \leq i \leq last$）

输出：E[first]…E[last]是已经有序的元素。

```
void mergeSort(Element[] E, int first, int last)
{
    if(first<last)
    {
        int mid=(first+last)/2;
        mergeSort(E,first,mid);
        mergeSort(E,mid+1, last);
```

```
            merge(E,first,mid,last);
        }
    }
```

我们发现学生经常将 Merge 和 Mergesort 搞混。

归并分析

首先找到最坏情况下 Mergesort 关键字比较的渐进阶。我们定义问题的规模 n=last-first+1，要排序的元素个数。

***更精确的归并排序分析**

根据下一节求出的低限，有时候需要得到最坏情况下比较次数的的精确值。我们可以看出 Mergesort 非常接近低限。读者可以跳过

4.7 基于比较的排序的低限

插入排序和快速排序做的关键字比较次数在最坏情况下在 $\Theta(n^2)$。我们能用归并排序加以改进，使得最坏情况在 $\Theta(n\log n)$。我们能做的更好吗？

在本节中我们

4.7.1 排序算法的判定树

4.7.2 最坏情况的低限

4.7.3 平均行为的低限

4.8 堆排序

快速排序重新排列原来数组的元素，但是不能保证对一个个问题进行完全的分解，因此它有一个最坏的情况。归并排序可以保证问题的分解，对最坏的情况下也接近最优，但是它不能在原来数组重新排列；它需要相当大的辅助的空间。堆排序重新排列了数组的元素，它的最坏情况 $\Theta(n\log n)$，这是最优的增长率，所以综合了快速排序和归并排序的优点。它的缺点是有比其他两个大的常数因子。但是新版本的 Heapsort 减少了常数因子，使得堆排序可以和快速和归并一起竞争了。我们称这个新版本叫 *Accelerated Heapsort*。因此 *Accelerated Heapsort* 也是一种排序的选择方案。

4.8.1 堆

堆排序用到一种叫*堆*的数据结构，它是一颗特殊的二叉树。堆的定义包括数据结构和节点上数据的条件，叫做 *partial order tree property*。非正式的，堆*结构*是一个完全二叉树，它的叶子节点的右面部分可能不全。（参见图 4.17）堆提供了对 ADT 优先级队列的的有效实现（2.5.1 小节）。在堆中，最高优先级的的数据放在根节点。根据这种优先排序的观点，根节点可以是最小值（小顶堆）或是最大值（大顶堆）。对于一个升序的堆排序排序的堆，我们需要建立一个大顶堆，我们将从这方面来描述堆，其他情况我们可能需要建立一个小顶堆 。

我们将使用两个术语，S 是一个关键字是线型的元素的集合，T 是高度为 h 的二叉树，它的节点属于 S。

定义 4.1 堆结构

二叉树 T 是一个堆*结构*，当且仅当它符合下面的条件：

1．T 的高度至少为 h-1 且为完全二叉树

2．所有的叶子的深度为 h 或 h-1

3．所有到深度为 h 的叶子的路径都在到深度为 h-1 的叶子的路径的左侧

深度为 h-1 的最右边的内部节点可能有左孩子，没有右孩子。其他的内部节点都有 2 个孩子。堆结构的另一个名字是*左完全二叉树*。

定义 4.2 partial order tree property

树 T 是（大的）partial order tree 当且仅当每一个节点都大于等于它的每一个孩子（如果存在孩子）。

一颗完全二叉树是一个堆结构。当添加新元素到堆时，他们必须从左自友加到最底层，而且如果移除节点，必须一处最底层最右边的节点，如果移除节点之后仍然是一个堆的话。注意根必须是堆中最大的元素。

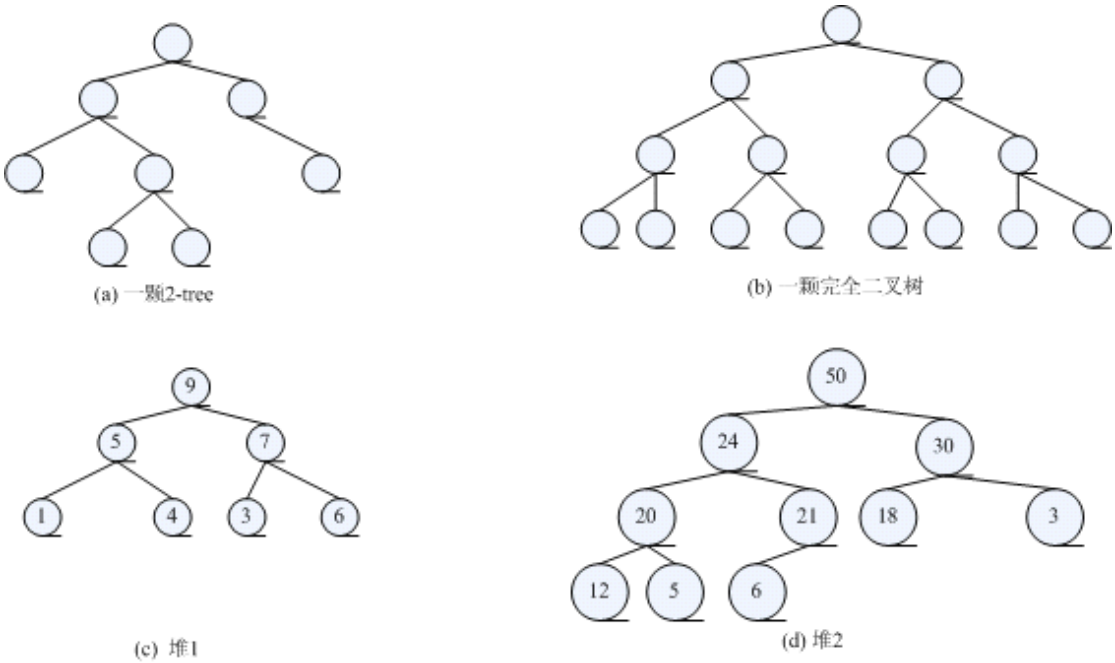


图 4.17 2-trees，完全二叉树和堆

4.8.2 堆排序的策略

如果要排序的元素在一个堆中，则我们通过重复下面的过程构造一个逆序列：把堆的根元素（剩下的最大的元素）移出，移走了最大元素之后重新排列元素使得剩下的元素仍然是一个堆。这个操作就是优先级队列 ADT 的 deleteMax 操作。（我们可以用小顶堆来得到一个升序序列，之所以用大顶堆是因为我们在 4.8.5 要学习一种更有效率的实现方法。）

既然第一步是要建立一个堆，之后不断调用 deleteMax（这个函数调用某个重新构造一个逆序列堆排列元素），这看起来不象一个得到排序算法的有效的策略。但是把它发展成一个排序算法很容易。因此我们在这里大概的给出策略，等会给出细节。和往常一样，我们假设 n 个元素存放在数组 E 中，但是这次我们假设索引从 1,…,n，因为这样我们看堆的实现方式时比较清楚。暂时我们假设堆（叫 *H*）在别处。

```
heapSort(E, n)
{
    Construct H from E, E 是要排序的 n 个元素的集合；
    for(i=0;i>=1; i--)
    {
        curMax=getMax(H);
        deleteMax(H);
        E[i]=curMax;
    }
}
```

图 4.18 的第一和最后的图片展示了一个例子，分别是 for 循环开始和结束之后的情况。中间的图片展示了 deleteMax 和 fixHeap 调用时重新排列的步骤，fixHeap 由 deleteMax 调用。

```
deleteMax(H)
{
```

拷贝 H 最底层最右边的元素到 K
删除 H 最底层最右边的元素
fixHeap(H,K)
}
显然，几乎所有的工作都由 fixHeap 完成。

我们现在需要一个算法来构造堆，还需要一个算法来 fixHeap。因为 fixHeap 可以用来解决构造堆的问题，所以我们下面考虑 fixHeap。

4.8.3 修正堆

fixHeap 过程恢复堆结构的 partial order tree property，堆结构必须是除了根以外其余部分都是符合要求的。特定的，当 fixHeap 开始的时候，我们有一个根是“空位”的堆结构。两个子树都是 partial order tree，而我们有一个额外的元素，叫 K，要插入到堆中。既然根是一个空位，我们从根开始，令 K（空位节点）下降到它正确的位置。到空位正确的位置之后，K（准确的说是 K 的关键字）必须大于或等于它的两个孩子，这样每一步中 K 需要和当前空位节点的大的孩子比较。如果 K 大于（或是等于）它就要可以被插入到等前空位节点处。否则，大的孩子被移动到空位，重复之。

例 4.1 FixHeap 实战

fixHeap 的动作展示在图 4.18 的第二到第五图片中。稍微回退一点，第一图片展示了初始时的状态，即 4.8.2 小节中 heapSort 的 for 循环开始时的状态。首先，fixHeap 将 H 的根关键字 50 拷贝到 curMax，使得树的根是一个空位；然后，它调用 deleteMax，它将最底层最右边的关键字 6 拷贝到 K，实际的删除该节点。

这就到了第二图片，这是 fixHeap(H,K)开始。空位节点的大的孩子是节点 30，这比 K 大，所以 30 移到空位位置，空位下降，到第三图片。同样的，空位节点的大的孩子仍然大于 K，所以空位节点继续下降。下一次，空位节点是一个叶子，所以 K 可以被插入到这个位置，H 的 partial order tree properter 就恢复了。

尽管堆的树结构是其本质且容易理解 HeapSort，我们之后将用数组来表示堆和子堆，而没有显式的边。

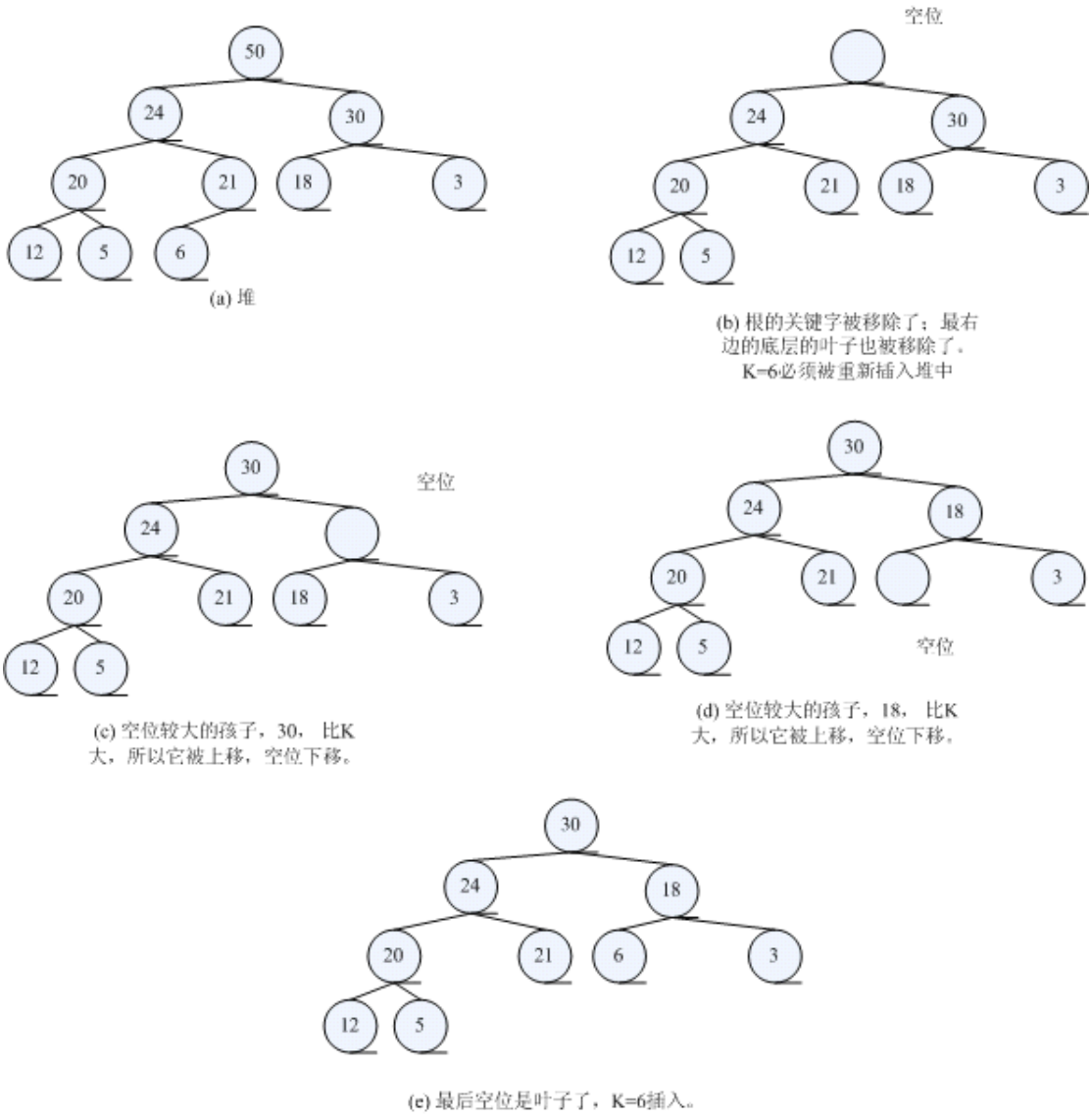


图 4.18 删除根的元素，重现建立 partial order tree property

算法 4.6 FixHeap （Outline）

输入：一个非空二叉树带有一个“空位”根，且它的左子树和右子树都是 partial order 树，待插入元素 K。H 的 type 叫**堆**。H 的节点假设类型为 **Element**。

输出：一个二叉树 H，其元素是 K 和原来 H 的元素，满足 partial order tree property。

注意：H 的结构不改变，但是节点的内容改变。

```
fixHeap(H, K)   //OUTLINE
{
    if(H 是叶子)
```

```
            插入 K 到 H 的根上
        else
            设 置 largerSubHeap 成 leftSubtree(H) 或
            rightSubtree(H)，选择两者中根的关键字大的。只
            要 rightSubtree 不是空，则会有一次关键字比较。
        if( K.key ≥ root(largerSubHeap).key)
            将 K 插入 H 的根
        else
        {
            插入 root(largerSubHeap) 到 H 的根;
            fixHeap(largerSubHeap, K);
        }
        return;
    }
```

引理 4.12 fixHeap 过程对于高度是 h 的堆在最坏情况下比较关键字 2h 次。

证明 每一次过程调用最多有两次关键字比较，每次递归调用树高减少 1.（在决定 largerSubHeap 时有一次隐式比较。）

4.8.4 构造堆

假设我们开始把所有的元素都任意的放入堆；就是说，partial order tree property 对于任意子树都不是必须的。fixHeap 算法可以通过分而治之策略来建立 partial order tree property。两个子树可以被递归转换成堆，然后调用 fixHeap 来将根元素下降到它正确的位置，因此合并两个子堆和根就成了一个堆。基本情况是由一个节点组成的树（例如一个叶子）；它已经是一个堆了。下面的算法就是这种思想。

算法 4.7 构造堆

输入：堆结构 H 但是不一定符合 partial order tree property。

输出：有相同元素的 H，已经重新排列使之满足 partial order tree property。

```
void constructHeap(H)
{
    if ( H 不是叶子)
    {
```

```
            constructHeap(H 的左子树);
            constructHeap(H 的右子树);
            Element K=H 的根;
            fixHeap(H, K);
        }
        return;
    }
```

如果我们拆开这个分而治之的算法，我们会看到实际的工作从靠近叶子的节点开始。(这是一种后序遍历。）参见图 4.19。练习 4.38 要求你实现一个迭代版本的 constructHeap。

正确性

定理 4.13 过程 constructHeap 为它的参数 H 建立 partial order tree property。

最坏情况分析

constructHeap 的递归等式依赖于 fixHeap 耗费的时间。定义问题的规模是 n，n 是堆结构 H 中节点的数量，我们看到 fixHeap 需要大约 2lg(n)次关键字比较。令 r 表示 H 右子堆的节点的数目。我们有

$$W(n) = W(n-r-1) + W(r) + 2\lg(n) \qquad \text{对于 } n>1$$

尽管堆是尽可能平衡的，r 可以小到 n/3。因此，当 costructHeap 是分而治之算法时，它的两个子问题不一定是相等的。对于任意的 n 在数学上解决这个递归式子有点困难。我们先解决 $N=2^d-1$ 时的，就是说是一颗完全二叉树，则对于 n 在 N/2 到 N 之间，W(N)是 W(n)的上界。

对于 $N=2^d-1$，左子树和右子树的节点数量是相等的，所以递归等式变为

$$W(N) = 2W(\frac{1}{2}(N-1)) + 2\lg(N) \qquad N>1$$

现在使用 Master 定理（定理 3.17)。我们令 b=2，c=2（N/2 与(N-1)/2 之间的区别可以忽略)，E=1，以及 f(N)=2lg(N)。现在选ε=0.1（或者任何小于 1 的小数）适用定理的情况 1：$2\lg(N) \in O(n^{0.9})$。则 $W(N) \in \Theta(N)$。

现在，回到一般的 n，既然 $N \le 2n$，$W(n) \le W(N) \in \Theta(2n) = \Theta(n)$。因此，构造堆是线型时间！（另一个方法在练习 4.39 中。）

现在仍然不清楚 Heapsort 是一个好的算法；它似乎需要额外的空间。下面考虑堆的实现。

4.8.5 堆的实现和堆排序

二叉树通常用链式结构实现，每个节点包含指针（或是其他种类的引用）指向它的子树。设置和使用每个结构需要额外的时间和额外的空间存放指针。但是，我们可以完全不用指针来存储和使用堆。在堆中，只有 d-1 层是满的情况下 d 层才可能有节点，所以一个堆可以可以按层存放在数组中，每一层从左自右一次存放节点。图 4.20 显示了如何存放图 4.17 中的两个堆。用这个方案，我们必须能快速的找到一个节点的子节点，以及快速的决定一个节点是不是叶子。将根存放在索引 1，而不是 0，是很重要的，这关系到我们将要讲到的公式。

堆排序分析

4.8.6 加速堆排序

回忆一下，fixHeap 处理根是空位，但是根的两个子树都满足 partial order tree property。一个新的节点将插入，但是它可能太小不能放在根的位置。这个元素下降到左或右子树，直到它合适的位置。

假设一个关键字开始的时候位置太低，就是说，对于它当前的位置它大了。bubbleUpHeap 将它向根“升起”。事实上，“升起”会更简单，因为，元素只需要与它的父节点比较——不需要决定左孩子大还是右孩子大。bubbleUpHeap 过程是对 fixHeap（算法 4.9）的自然的补充。在写出 bubbleUpHeap 的细节之后，我们将看到如何使用扩展因子来将 Heapsort 加速大约一倍。

我们继续假设是大顶堆，因为这是 Heapsort 使用的类型。准确的说，bubbleUpHeap 将给出元素 K 和一个空位位置，这样将元素放到空位的位置可能太低；K 可能大于它的父节点。

过程让空位到根的路径上的小的元素下降，空位上升，知道为新元素找到正确的位置。（对于索引 i，父节点是$\lfloor i/2 \rfloor$。）操作类似于插入排序中将一个新元素插入到前面已经有序序列中的过程。

算法 4.10 Bubble-Up Heap

输入：一个表示堆结构的数组；整数表示的 root 和 vacant，一个元素 K 要插入到 vacant 或是 vacant 的祖先，并且保持 E 的 partial order tree property。作为一个 precondition，如果不算 vacant，E 有 partial order tree property。

输出：以 root 为根且 K 已经插入的子堆，且有 partial order tree property。

注意：E 的结构没有改变，但是它的节点的内容改变了。

```
void bubbleUpHeap(Element[] E, int root, Element K, int vacant)
{
    if(vacant==root)
```

```
            E[vacant]=K;
        else
        {
            int parent= vacant/2;
            if(K.key<=E[parent].key)
                E[vacant]=k;
            else
            {
                E[vacant]=E[parent];
                bubbleUpHeap(E, root, K, parent);
            }
        }
    }
```

将一个元素通过 bubbleUpHeap 升起一层只需要一次比较。算法 4.10 可以用来实现在堆中插入一个元素（参见练习 4.41）。

联合 bubbleUpHeap 对 fixHeap 做一个简单的修改，允许我们将 Heapsort 比较的次数减约一半左右。这使得 Heapsort 可以在比较次数上和归并排序竞争了，他们的最坏情况下的系数一样了（1）。元素的移动次数，没有减少，但是数量级和归并一样了。堆排序的优点是不用辅助数组。

主要的思想是简单的。fixHeap 将一个元素下降一层需要两次比较。但是我们可以避免其中一次比较——

分析

本质上，

4.9 四种排序算法的比较

表 4.1 综合了迄今为止讨论过的四种排序算法分析的结果。尽管归并排序在最坏情况下最接近最优，但是有比较次数更少的算法。4.7 节得到的底限是相当好的。它对于有些 n 的值是很精确的；就是说，排序$\lceil \lg n! \rceil$次比较就足够，对于一些 n 的值。也知道$\lceil \lg n! \rceil$并不是对于所有 n 的值都足够。例如$\lceil \lg 12! \rceil = 29$，但是已经证明排序 12 个元素在最坏情况下至少需要 30 次比较。参看本章后面的 Notes 和 References，哪种排序算法在最坏情况下最接近底限。

算法	最坏情况	平均情况	空间使用

插入排序	$n^2/2$	$\Theta(n^2)$	In place
快速排序	$n^2/2$	$\Theta(nlogn)$	额外空间和 logn 成比例
归并排序	nlgn	$\Theta(nlogn)$	额外空间和 n 成比例
堆排序	2nlgn	$\Theta(nlogn)$	In place
加速堆排序	nlgn	$\Theta(nlogn)$	In place

表 4.1 四种排序算法分析的结果。Entries are number of comparisons and include the leading terms only。

4.10 Shell 排序

Shell 排序（以发明者 Donald Shell 命名）用的技术是有趣的，而且算法易于编程，运行的相当快。只是它的分析是非常困难，而且是不完全的。

4.10.1 算法

Shell 排序

4.10.2 分析和注意事项

4.11 基数排序

对于 4.2 到 4.10 的排序算法，对于关键字的假设只有一个：他们是线型的集合。基本的操作是比较两个关键字。如果我们对关键字作更多的假设，我们能可以对关键字做其他的操作。本节学习一种新的算法，叫“桶排序”，“基数排序”或是“分配排序”。

4.11.1 使用关键字的属性

假设关键字是打印在卡片上的名字，一个卡片一个名字。为了用手将他们按字母顺序排序，我们首先根据首字母将他们分成 26 堆，或者按几个字母一堆分好；用其他方法将每一堆卡片按字母顺序排好，比如类似插入排序；最后合并排好序的堆。如果关键字是 5 位十进制树，我们可以根据第一位将他们分成 10 堆。如果关键字是 1 到 m 之间的整数，我们可以每间隔 k 分一堆，[1, m/k]，[m/k+1, 2m/k]等等。上面的例子中，都是根据检查关键字的一个单独字母或一位数字，或是与预先定好的值比较，来将关键字分发到不同的堆中。然后再将堆单独排序在组合。以这种方式排序的算法与前面讨论所的算法都不同，因为要使用这种方法我们必须知道关键字的结构或是顺序。

我们将在后面展示*基数排序*算法的细节。为了区分同类新的其他算法，我们使用“桶排序”做这类算法的通用名字。

桶排序有多块

一个桶排序算法有 3 步：

1. 分发关键字
2. 单独排序桶
3. 合并桶

每一步工作的类型都是不同的，所以我们选一个通用的基本操作来计数的策略不好用了。假设有 k 个桶。

在分发阶段，算法检查每个关键字一次（不管是检查某个特定的位域，或是将关键字与常数个特定值比较。）然后算法决定关键字属于那个桶。这一步可能牵涉到拷贝关键字或是设置指针引用什么的。第一步的一个合理的实现应该将操作步数控制在 $\Theta(n)$。

为了排序桶，假设我们使用一种基于比较关键字的排序算法，即对于 m 个元素的桶需要 S(m)次比较。令 n_i 是第 i 个桶中的元素个数。第二步算法要做 $\sum_{i=1}^{k} S(n_i)$ 次比较。

第三步，合并桶，最坏情况下需要将所有的元素从桶拷贝到一个 list 中；操作的数量是 $O(n)$。

因此主要的工作是在排序桶时。假设 S(m)在 $\Theta(m\log m)$。则如果关键字均匀的分布在桶中，算法在第二步要做 $ck(n/k)\lg(n/k)=cn\lg(n/k)$次比较，这里 c 是一个依赖于在桶中使用排序算法的常数。增加桶的数量 k 将减少比较的数量。如果我们选 k=n/10，则要做 nlg10 次比较，桶排序的时间将在 n 的线性之内，假设关键字是均匀的分发且第一步的运行时间不依赖于 k 的话。（As a caveat, the fewer elements pre bucket, the less likely it is that the distribution will be even。）但是在最坏的情况下，所有的元素都装进一个桶里面，第二步中要对整个 list 排序，而第一步和第三步都变成了浪费。因此在最坏情况下，桶排序会变的极度低效。如果关键字的分布是可以预先知道的，则可以调整每个桶保存关键字的范围，使得所有的桶都接受近似相等数量的元素。

桶排序需要的空间依赖于桶是如何存储的。如果每一个桶都是连续空间的集合（例如数组），则每一个桶都必须分配足够的空间以存储一个桶的最多可能的元素数量，也就是所有的元素 n。因此排序 n 个元素可能需要 kn 个空间。随着桶数量的增加，算法的速度增加但是使用的空间也随之增加。链表可能会 好点；仅需要 $\Theta(k+n)$ 的空间（n 个元素占用 n 个空间，k 个表头占用 k 个空间）。将关键字分发到桶中可能需要构造 list 的节点。但是每个桶中的元素如何排序呢？快速排序和归并排序，先前讨论的两种最快的排序方法，都可以在链表上简单的实现（参看练习 4.22 和连续 4.28）。如果桶的数量很大，桶内的元素的数量很少，可以使用慢一点的算法。插入排序也也可以简单的修改之后就用于链表（参看练习 4.11）。如果每个桶都是大约 n/k 个元素，归并排序对每个桶平均做大约$(n/k)(\lg(n)-\lg(k))$次比较，总共做 $n(\lg(n)-\lg(k))$次比较。再次强调，随着 k 增加，速度也增加，但是空间也增加。

你可能奇怪为什么我们不使用桶排序算法递归的创建更小的桶。这里有几个原因。簿记工作马上就会超出控制；指向桶头的指针以及最后合并元素需要信息不得不不断的进栈出栈。由于每一个递归调用都要簿记信息，算法不能依赖于最后将问题分解成一个桶装一个元素，所以将用到其他的排序算法来排序小的桶。因此如果一开始就使用了合适多的桶，在递归桶排序就会得到的少，失去的多。但是尽管递归分发关键字到桶是低效的，但有时利用这种思想还是很有用的。

4.11.2 基数排序

假设关键字是 5 位数字。一个递归的算法，就像刚建议的，可以首先根据最左边也是影响最大的首位数字将关键字分发到 10 个桶中，之后在递归的继续根据第二位的数字将他们分发到更多的桶中去。直到桶完全有序之后再合并，从而有一个很大的簿记量。令人吃惊的是，如果关键字首先按他们*最小影响*位（或者位域，字母，域）分发的桶中，则桶在再次分发之前就可以合并了。桶排序的问题就彻底解决了。如果关键字是 5 位，则算法分发关键字到桶中再合并桶 5 次就可以了。算法从右自左分发关键字，就像图 4.27 所展示的。

这样总能工作吗？在最后一趟中，当两个关键字分发到一个桶中是因为他们都以 9 开始，什么保证了他们之间的相对顺序是对的呢？在图 4.27 中，关键字 90283 和 90583 仅在第三个数字不同，他们除了第三次每次都分发在不同的桶中。在第三趟之后，在桶按顺序合并之后，分发在同一个桶中两个关键字的相对顺序没有再改变，这些关键字之间保持着相对顺序。一般的，如果两个关键字不同的最左边的位是第 i 位（从右算），他们在第 i 趟之后其相对位置将保持不变。这个命题可以被直接的归纳证明。

这种排序方法用于卡片排序机。一种古老的机器，这种机器做分发步骤；每次分发完之后收集合并继续分发。

分发到堆或桶，可以通过卡片上的一列，一个数字位，或是关键字的位域来控制。算法之所以叫*基数排序*是因为它将数字看作特定基数的数字。在图 4.27 的例子中，基数是 10。如果关键字是 32 位正整数，算法可以使用 4 位位域，即把基数当作 16。它将分发他们到 16 个桶中。因此基数也是桶的个数。在下面的基数排序算法中，我们假设分发是基于位域的。位域先从关键字的低位提取。如果可能位域的数目保持在常数不依赖于 n（输入元素的个数）。一般情况下，基数（桶的个数）随着 n 而增长。通用的选择是最大的整数 w，$2^w \leqslant n$。则每一个域有 w 位宽。如果关键字稠密的分布（例如，在一个 n 的多项式范围内），这种策略有一个常数的域。

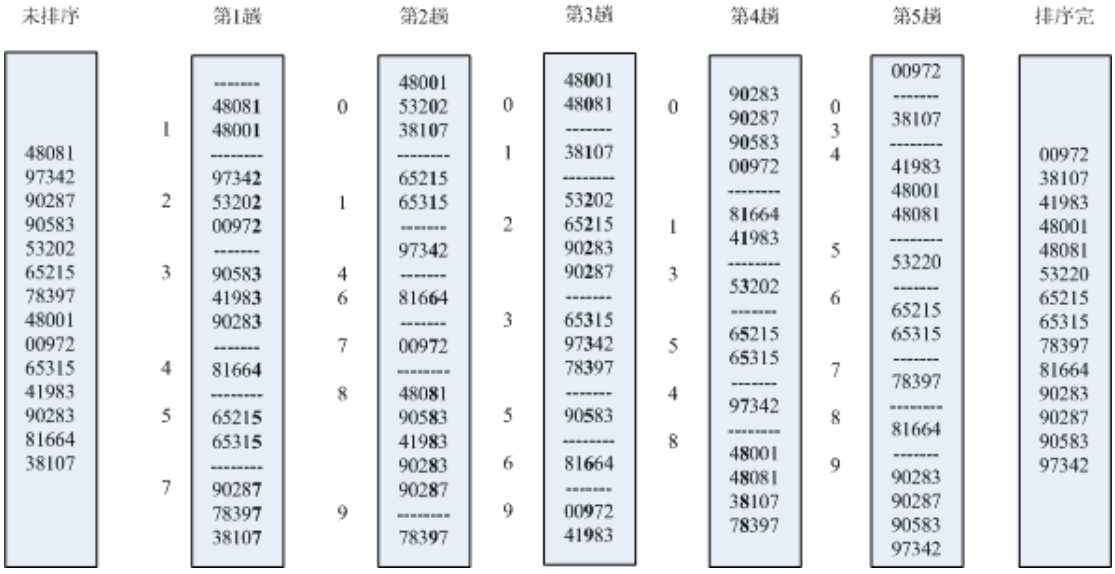


图 4.27 基数排序

数据结构展示在图 4.28 中，图中显示的是图 4.27 例子中的第三步。注意每一个桶 list 都是逆序的，因为新元素都是附加在 list 的前面。但是合并过程在合并的时候再次反转他们，所以最后合并是正确的顺序。

算法 4.13 基数排序

输入：未排序 list L，分发用的桶的数量 radix，关键字分发次数 numFields。

输出：排序的 list, newL。

注意：distribute 过程反转桶中的 lists，combine 过程再次反转它们（以逆序循环），所以合并完得到了想要的顺序。用 List ADT 的操作来操作链表（参看图 2.3）.

```
List radixSort(List L, int radix, int numFields)
{    List[] buckets= newList[radix];
     int field;                          // 关键字中域的数量
     List newL;

     newL= L;
     for(field=0; field<numFields; field++)
     {    // 将桶数组初始化为空；
          distribute(newL, buckets, radix, field);
          newL=combine(buckets, radix);
     }
     return newL;
}

void distribute(List L, list[] buckets, int radix, int field)
{
     List remL;

     remL=L;
     while(remL!=null)
     {    Element K=first(remL);
          int b=maskShift(field, radix, K.key);
          // maskShift(f, r, key)选择 key 中第 f 个域（从右开始），
          // 基于基数 r。
```

```
            // 结果 b 的范围是 0,…, radix-1, b 是 K 所在的桶。
            buckets[b]=cons(K, buckets[b]);
            remL= rest(remL):
        }
        return;
}

List combine(List[] buckets, int radix)
{   int b;   //桶的数量
    List L, remBucket;

    L= null;
    for(b= radix-1; b>=0; b--)
    {   remBucket=buckets[b];
        while(remBucket!=null)
        {   Key K= first(remBucket);
            L=cons(K, L);
            remBucket= rest(remBucket);
        }
    }
    return L;
}
```

分析与注意事项

分发一个关键字需要解析出一个域，还有一个连接操作；步数是一个常数。所以对所有的关键字，分发是Θ(n)步。类似的，合并也是Θ(n)步。分发和合并的趟数是 numFields，也是关键字分成多少个域。如果它可以控制在常数之内，基数排序总的执行步数在 n 的线性时间之内。

我们的基数排序的实现使用了Θ(n)的额外空间用于链域，提供的基数是以 n 相关的。其它的不使用链的实现也将使用Θ(n)的额外空间。

第五章 Selection 和 Adversary Arguments

5.1 概述

在这章中我们学习几个可以被归类到一个通用名字 *selection* 的问题。找到一个集合的中间元素是一个总所周知的例子。除了找到有效解决问题的算法，我们还将找到这类问题的*低限*。我们会介绍一种被称为*虚拟对手*（adversary arguments）的广泛应用的技术来建立低限。

5.1.1 Selection 问题

假设 E 是包含 n 个元素的数组，数组元素的关键字属于某个线型集合，令 k 是 $1 \le k \le n$ 的一个整数。selection 问题是找到 E 中第 k 小的元素。这样的元素叫做*有 rank k*。与我们学过的排序算法一样，我们将假定唯一的操作是比较两个关键字（以及拷贝或是移动元素）。在这一章中关键字和元素是一样的，因为我们注意的是关键字比较的次数，而且我们一般不关心元素移动。还有，当关键字在一个数组中存放时，我们将使用位置 1,…,n 而不是 0,…,n-1 以符合一般的 rank 术语，位置 0 只是简单的放在那里不用。

在第一章中我们解决了一个 k=n 的 selection 问题，那个问题是简单的找到最大的元素。我们考虑了一个最简单的算法做了 n-1 次关键字比较，之后我们证明了没有算法可以低于这个值。对于查找最小的元素，即 k=1 解决方法是一样的。

另一个常见的 selection 问题是 $k = \lceil n/2 \rceil$，就是找到中间的元素，或者叫 *median* 元素。median 对于概括一个大的集合的数据是很有用的，比如一个国家中所有人的收入或是某个职业的收入，房子的价格或是大学入学考试中的分数。例如新闻报道不是用包含所有数据的集合，而是告诉我们一个平均值。可以在Θ(n)内简单的计算 n 个元素的平均值吗？我们如何才能有效的计算平均值？

当然，所有的 selection 问题都可以通过对 E 排序来解决；之后无论我们想要 rank k，E[k]就是解。排序需要Θ(nlogn)次比较，我们可以知道对于有些值 k，selection 问题可以在线型时间内解决。直觉上讲似乎找到 median 是 selection 问题中最难解决的。我们可以在线型时间内找到 median 吗？或是我们能建立一个找到 median 的低限吗，可能是Θ(nlogn)？我们将在本章回答这个问题，我们还将描述一个 selection 问题的框架算法。

5.1.2 低限

目前为止我们已经将判定树作为建立低限的主要手段。算法判定树的内部节点表示了算法要执行的比较的次数，而叶子表示了输出。（1.6 节的查找算法中，内部节点同样表示输出。）最坏情况下比较的次数是树的高度；对于 L 个叶子，高度至少是 $\lceil \lg L \rceil$。

在 1.6 小节中我们使用判定树得到最坏情况下查找问题的低限是 $\lceil \lg(n+1) \rceil$。这是二分查找的精确比较次数，所以判定树给出了最好可能的低限。在第 4 章我们使用判定树来得到

排序的低限是$\lceil \lg n! \rceil$或是粗略解$\lceil n\lg n - 1.5n \rceil$。有些算法非常接近这个低限，再一次的，判定树给出了非常可信的结果。但是，判定树对于 selection 问题就不是很好用了。

一个 selection 问题的判定树至少有 n 个叶子因为集合中 n 个关键字中的每一个都可能是输出，即第 k 小的值。因此我们得出树的高度（最坏情况下的比较次数）至少是$\lceil \lg n \rceil$。但是这不是一个好的低限；我们已经知道在最简单的情况下查找最大的元素需要至少 n-1 次比较。判定树在那里错了呢？在查找最大元素的判定树中，有些输出出现至少在多个叶子上，事实上将有多于 n 个叶子。参考练习 5.1，将要求你画出 FindMax（算法 1.3）在 n=4 时的判定树。因为我们不知道一种简单的方法知道一个结果到底包含多少重复的叶子，所以判定树不能给出好的低限。

代替判定树，我们用一种叫 adversary argument 的技术来为 selection 问题建立更好的低限。这种技术在下面描述。

5.1.3 虚拟对手 Adversary Arguments

假设我们和朋友玩一个猜谜游戏。你选出一个时间（月份和日期），朋友将通过问是/不是来猜这个时间。你必须迫使你的朋友问尽可能多的问题。如果第一个问题是，“是在冬天吗？”而你是一个好的对手，所以你回答“不是"，因为 3 个季节比一个季节要多。对于问题，“月份的第一个字母在字母表的前半吗？”你回答“是"。但是这是欺骗吗？你根本没有选出一个真实的日子！事实上，你不会选出一个确定的月和日期，直到为了不让你的答案前后矛盾位置。这不是一个玩猜谜游戏的好方法，但是但是它在找一个算法的低限时时个正确的方法。

假设我们有一个算法，我们认为它是有效的。想象有一个对手想证明它不是有效的。在每一个算法做判定的点（例如一次关键字比较），对手告诉我们判定的结果。对手选择它的回答使得算法更难找到结果，就是说要做更多的判定。你可以认为对手逐渐构造一个算法的“坏”的输入。对手的回答的唯一限制是必须前后一致；就是必须*有*输入可以使对手的回答是正确的。如果对手可以迫使算法执行 f(n)步，则 f(n)就是最坏情况下算法执行步数的低限。在练习 5.2 中对排序和归并的关键字比较进行了分析。

事实上，“设计对抗对手”在解决基于比较的问题时通常是一个有效的技术。In thinking about what comparison to make in a given situation, imagine that the adversary will give the least favorable answer—then choose a comparson where both outcomes are about equally favorable. 这种技术的细节在 5.6 节讨论。但是，这里我们主要感兴趣的是对手策略在求低限中的角色。

5.1.4 竞标赛

在本章剩下的部分我们解决 selection 问题的算法，并用 adversary arguments 来讨论各种情况的低限，包括 median。在大多数算法和 arguments 中，我们使用比赛或是竞标赛，来描述比较的结果。比较找到的大的称为*胜者*，小的称为*败者*。

5.2 查找 max 和 min

这小节中，我们分别用 max 和 min 来代表 n 个元素中的最大关键字和最小关键字。

我们采用算法 1.3 来找到 max，从集合中去处 max。再次使用相反的算法 1.3 在剩下的 n-1 个元素中找到 min，这样就找到了 max 和 min。因此 max 和 min 可以在(n-1)+(n-2)次比较完成。这不是最优的。尽管我们知道（第一章）独立的查找 max 和 min 都至少需要 n-1 次比较，但是两个一起找的时候有些工作可以“共享”。练习 1.25 要求给出了一个查找 max 和 min 的只需要约 3n/2 次关键字比较的算法。一个解决方法（）n 是偶数是将关键字两两比较，做 n/2 次比较，然后在胜者中查找 max，在败者中查找 min。如果 n 是奇数，最后的关键字即可能是胜者也可能是败者。两种情况下，总的比较次数都是$\lceil 3n/2 \rceil - 2$。本节中，我们将给出一个对手策略来展示这个解决是最优的。特别的，在剩下的部分中，我们证明：

定理 5.1 在 n 个关键字中通过比较找 max 和 min 的任意算法在最坏情况下都必须做至少 3n/2-2 次关键字比较。

证明 为了建立低限，我们假定关键字是不同的。为了知道一个关键字 x 是 max，一个关键字 y 是 min，一个算法必须知道其他的所有关键字都直接或间接的败于 x，其他关键字都直接或间接胜过 y。如果我们计胜和败为一个单位的信息，则算法必须知道（至少）2n-2 个单位的信息来保证给出的是正确回答。我们给出一个对手策略来回答比较结果，对手对于每个比较给出的结果都是尽可能的使得到的信息更小。想象对手构造一个特定的输入集合来回答算法的比较。

我们对算法进行中任何情况下关键字的状态标记如下：

关键字状态	含义
W	至少胜过一次而从未败过
L	至少败过一次而从未胜过
WL	至少胜过一次败过一次
N	还没用比较过

每一个 W 或 L 是一个单元的信息。状态 N 不携带信息。对手策略在表 5.1 中描述。主要一点是，除了两个关键字都没有比较过的情况，对手给出的回答至少最多提供一个单位的新信息。我们需要验证如果对手符合下面的规则，它的回答构成某些输入。之后我们需要展示这种策略迫使算法任意算法都必须做理论宣称的比较次数。

算法比较的 x 和 y 的状态	对手的回答	新的状态	新信息的单位
N, N	x>y	W, L	2
W, N 或 L, N	x>y	W, L 或 WL, L	1
L, N	x<y	L, W	1
W, W	x>y	W, WL	1
L, L	x>y	WL, L	1
W, L 或 WL, L 或 W, WL	x>y	没有改变	0
WL, WL	与赋值一致	没有改变	0

表 5.1 对于 min 和 max 问题的对手策略

观察表 5.1 中除了最后一行的情况，对手会让还没有败过的关键字继续胜，让没有胜过的关键字继续败。考虑第一种可能性；假设算法比较 x 和 y，对手选择 x 作为胜者，x 还没有败过。即使赋给 x 的值比赋给 y 的值要小，只要不与先前给出的回答矛盾对手就可以改变 x 的值，使之比 y 大。其他的情况，比如从没有胜过的关键字要继续败，处理方式类似——如果需要就减少关键字的值。所以对手可以用表 5.1 中的规则构造一致的输入来回答算法的比较。这展示在下面的例子中。

为了完成 5.1 的证明，我们只需要证明对手规则迫使任何算法都要做至少 3n/2-2 次比较才能得到 2n-2 单位的信息。唯一的一次比较能得到两个单位信息的情况是两个比较的元素以前都没有参加过比较。假设 n 是偶数。算法可以做最多 n/2 次能得到 2 个单位信息比较，所以算法可以通过这种形式的比较得到 n 个单位的信息。对于每一个其他比较，算法最多得到一个单位的信息。算法还需要 n-2 单位的信息，所以她必须做至少 n-2 次比较。因此为了得到2n-2的单位的信息，算法必须做至少n/2+n-2=3n/2-2 次比较。读者可以简单的检查 n 是奇数的情况，是至少 3n/2-3/2 次比较。完成了定理 5.1 的证明。

比较	x1		x2		x3		x4		x5		x6	
	状态	值	状态	值	状态	值	状态	值	状态	值	状态	值
x1,x2	W	20	L	10	N	*	N	*	N	*	N	*
x1,x5	W	20							L	5		
x3,x4					W	15	L	8				
x3,x6					W	15					L	12
x3,x1	WL	20										
x2,x4			WL	10			L	8				
x5,x6									WL	5	L	3
x6,x4							L	2			WL	3

表 5.2 对手策略找 max 和 min 的例子

例 5.1 使用对手规则构造输入

表 5.2 的第一列展示一个可能被有些算法得出的比较序列。剩下的列展示了对手赋给关键字的状态和值。（还没有赋值的关键字以星号标识。）第一行之后的每一行仅包含当前比较的条目。注意当 x3 和 x1 比较时（第 5 次比较），对手增加了 x3 的值，因为 x3 要胜。之后对手改变了 x4 和 x6 的值以和他的规则一致。5 次比较之后，除了 x3 的每一个关键字都至少失败了一次，所以 x3 是 max。最后一次比较之后，x4 是唯一没有胜过的关键字，所以它是 min。在这个例子里，算法做了 8 次比较；6 个关键字最坏情况下的的低限（前面证明了）就是 3/2×6-2=7。

5.3 查找次大的关键字

我们找到并去除最大的元素，然后在剩下的集合中找到最大的，这样就找到了次大的元

素。还有更有效法的方法吗？我们可以证明一个方法是最优的吗？本小节回答这些问题。

5.3.1 概述

整个小节中，我们分别用 max 和 secondLargest 表示最大和次大的元素。为了简化问题和算法，我们假定关键字都是不同的。

次大的元素可以用 FindMax（算法 1.3）做 2n-3 次比较来找到，但是这似乎不是最优的。我们必须期望在查找 max 的过程中发现的一些信息用于降低查找 secondLargest 的比较次数。特别的，任何关键字如果除了失败于 max 之后还失败于别的关键字，则它肯定不是 secondLargest。在第二趟查找的过程中可以被忽略的关键字在查找 max 中的过程中都可以被发现。（如何保存他们的位置将稍后讨论。）

在一个 5 个元素的集合上的使用算法 1.3，结果可能如下：

比较	胜者
x1, x2	x1
x1,x3	x1
x1,x4	x4
x4,x5	x4

则 max=x4，次大的元素可能是 x5 或是 x1 因为 x2 和 x3 都失败于 x1。因此这个例子中还需要做一次比较就可以找到 secondLargest。

但是可能在我们第一趟查找 max 时我们没有得到任何有用的查找 secondLargest 的信息。如果 max 是 x1，则其他每一个元素都只与 max 比较了一次。这是不是意味着找到 secondLargest 最坏情况下必须做 2n-3 次吗？不。前面说的都是使用算法 1.3 的情况。没有算法可以用小于 n-1 次算法找到 max 的，但是其他算法可以提供更多的信息用来在第二趟中减少比较的次数。下面描述的竞标赛方法提供了更多的信息。

5.3.2 竞标赛方法

竞标赛方法之所以叫这个名字是因为它以竞标赛的方法执行比较。关键字分对，“分轮”比较，胜者进入下一轮。（如果有一轮的关键字数量是奇数，他们中有一个简单的等着进入下一轮。）锦标赛可以描述成图 5.1 中显示的二叉树。每一个叶子包含一个关键字，而每一层的父层都是这一层胜出的胜者。根包含最大的元素。同算法 1.3 一样，需要 n-1 次比较找到 max。

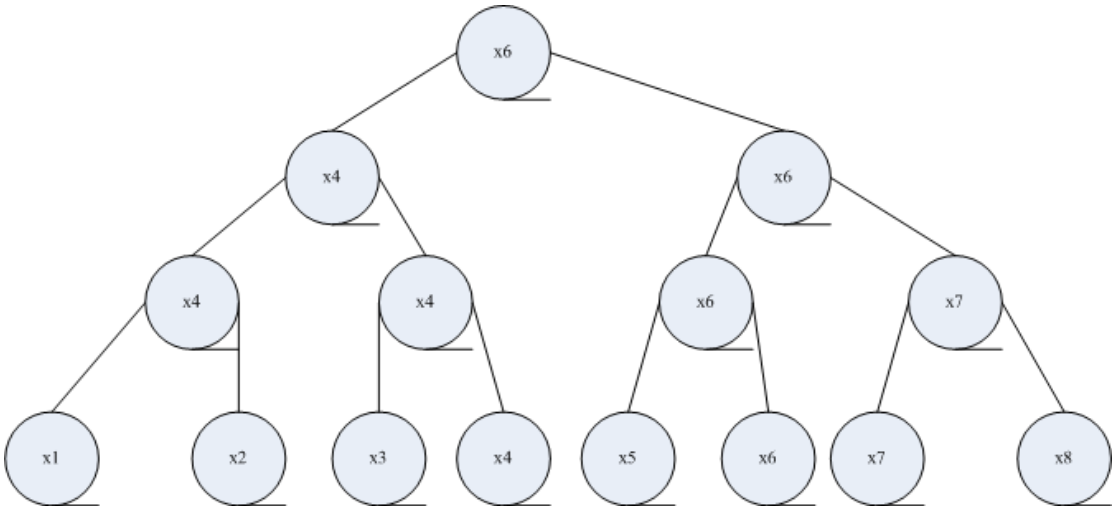


图 5.1 竞标赛的例子；max=x6；次大元素可能是 x4,x5,x7

在查找 max 的过程中，除了 max 之外的每一个关键字都失败一次。有多少直接失败与 max？如果 n 是 2 的整数次幂，这里有 $\lg n$ 轮；一般情况下是 $\lceil \lg n \rceil$。既然每一轮中都有 max 在，则最多有 $\lceil \lg n \rceil$ 个元素仅失败与 max。他们是 secondLargest 所有可能的候选。算法 1.3 可以用与在 $\lceil \lg n \rceil$ 个元素中查找最大的，需要做 $\lceil \lg n \rceil - 1$ 次比较。因此最坏情况下竞标赛查找 max 和 secondLargest 共需要 $n + \lceil \lg n \rceil - 2$ 次比较。这比 2n-3 是一个进步。我们可以更好吗？

5.3.3 An Adversary Lowe-Bound Argument

我们考虑的两个方法都是首先找到最大的元素。这不是浪费。任何查找 secondLargest 的算法必须也找到 max，因为为了知道这个关键字是次大的，首先必须知道它不是最大的；就是说它必须有一次比较失败。在 secondLargest 必须失败的那次比较中胜出的那个就是 max。

5.3.4 查找 max 和 secondLargest 竞标赛方法的实现

为了找到 max 进行的竞标赛我们需要一种方法保存每一轮的胜者。在通过竞标赛找到 max 之后，只有这些关键字参与查找 secondLargest。在我们还不知道那一个是 max 的时候我们如何保存那些只败于 max 的关键字呢？既然竞标赛的概念就是一棵二叉树，自然的想到了 4.8.1 小节的*堆结构*。对于 n 个元素的集合，需要一个有 2n-1 个节点的堆结构；就是 E[1],…,E[2*n-1]。一开始，将元素放在位置 n,…,2n-1。随着竞标赛的过程，将逐步用胜者填写 1,…,n-1（反着填）。练习 5.4 显示了额外的细节。这个算法使用线型的额外空间，运行时间也是线型的。

5.4 Selection 问题

假设我们想找到一个数组 E 中 n 个元素的中间的元素（我们想元素有 rank$\lceil n/2 \rceil$）。在前面的小节中，我们找到了有效的方法找到 rank 接近一端或是另一端（比如最大，最小，同时最大最小，次大的）的方法。练习探索了很多变种，但是所有的解决这些问题的技术在我们不是对付一端的情况时都将失去效率，对于查找 median 都没有用。如果我们找到一个解决方案可以有效的简单排序整个集合，则需要新的思想。

5.4.1 一个分而治之接近

假设我们能将关键字分成两个集合，S_1 和 S_2，这样所有在 S_1 中的关键字都小于 S_2 中的关键字。则

5.5 查找 median 的低限

我们假设有 n 个元素的集合 E 且 n 是奇数。我们将

5.6 Designing Against an Adversary

设计一个对手可能是一种强大的技术

第六章 动态集&查找

6.1 概述

动态集合是在计算规程中它的关系会改变的集合。在有些应用程序中，，集合一开始是空的，在计算的过程中元素一个个加入。通常集合可能的最大规模预先是不知道的。另一种应用程序一开始有一个大的集合，在计算过程中逐步删除元素（通常在集合为空的时候结束）。有些应用程序既删除也添加元素。开发了许多数据结构来表示动态集合。那种数据结构更有效率依赖于需要的操作和访问模式。首先我们描述数组成倍增长技术，他是一个基本工具。接着我们引为平摊（amortized）时间分析的基础，，这是一项为了展示各种动态集合实现的效率的常用的技术。最后我们考察许多流行的用来表示动态集合的数据结构。他们作为适当的 ADT 的实现而出现。

红-黑树提供了一种平衡二叉树的形式，它对于有效的实现二叉查找树是很有用的。二叉查找树和 hash 表是字典 ADT 的两种流行实现。

动态等价关系在许多应用中存在，它们的操作与 Union-Find ADT 紧密相关。使用 In-Tree ADT，在一些情况下是非常高效的实现。

优先级队列是许多算法的工具，尤其是贪婪算法。对优先级队列 ADT 两种高效的实现

是二叉堆（也用于堆排序）和 pairing forests，也称为 lazy pairing heaps。

本章将介绍这些主题。为了进一步阅读和更深层次的学习，请参考本章后面的 Note 和 References。

6.2 数组翻倍

引起动态集合并的典型情况是我们在计算开始时并不知道需要多大的数组。分配一个“满足需要的最大的”的数组通常是非常令人不满意，但是这是最普通的解决方法。一个简单的更有灵活性的方案是，开始的时候只分配一个小的数组，当空间不组时就加倍数组。为了完成这项工作我们必须记录当前数组用了多少，以及当前分配了多少。Java 通过 length 域自动记录了当前分配了多少，但是对前一个参数是程序员的责任，它依赖于具体的应用。

假设我们组织了一个类 **setArray**，有两个域 **setSize** 和 **elements**，后一个是 element 类型的数组，这里我们假设类型是 Object。最开始我们如下够这个类的对象：

```
setArray mySet= new setArray（）;
mySet.setSize=0;
mySet.elements=new Object[100];
```

在每次向 mySet 中添加元素的时候，setSize 都将增加。但是在插入新元素之前，都要确保还有空位，若没有了，把数组的大小翻倍。这个过程是分配一块两倍原来大小的新数组，然后把所有元素移动到新数组中。代码如下：

```
if(mySet.setSize = =mySet.element.length)
    arrayDouble(mySet);
继续插入新元素。.
```

在 arrayDouble 子例程如下:

```
arrayDouble(set)
{
    newLength=2*set.elements.length;
    newElements=new Object[newLength];
    将 set.elements 数组的元素移动到 newElements 数组中。
    set.elements=newElements;
}
```

耗时的大部分是移动元素。但是，我们现在将展示以这种存储方式插入 n 个元素的总系统开销是Θ(n)。

假设插入第(n+1)个元素将触发数组翻倍操作，令 t 是将数据从旧数组转移到新数组的花费（假设 t 是常数）。在这次数组翻倍操作中要进行 n 个元素的转移。但是在前一次操作中是 n/2 个元素转移，再之前是 n/4，等等。从集合创建以来所有元素转移不会超过 2tn。

这是可以*平摊*或者铺开偶尔出现昂贵操作的简单例子，所以每一步的平均开销是一个常数时间低限。平摊时间分析在下一节解释。.

6.3 平摊（Amortized） 时间分析

就像我们在前一节看到的，下面的情况可能发生：对于同一种类型每一个操作做的工作变化很大，但是一个长的操作序列总的时间远小于序列的长度乘以每个操作的最坏时间。在动态集和他们相关的操作中这种情况经常出现。叫*平摊（Amortized） 时间分析*的技术用来

为这种情况提供更精确的时间分析。名字 *amortized* 来自于（很宽松的解释）分解一个大帐单的商业记账实践， which was actually incurred in a single time period, over multiple time periods that are related to the reason for incurring the cost. 在算法分析的时候，一个大的操作被展开成许多操作，这里其他操作没有这么耗时。这一小节给出 amortized 时间分析的一个大概框架。技术在概念上简单的，尽管为了对不同的问题得到有效的方案需要做点创新。

假设我们有一个 ADT，我们想用 amortized 来分析它的操作。我们使用术语 *individual operation(单体操作)*来表示一个操作的一个执行步骤。平摊时间分析基于下面的等式：

$$amortized\ cost = actual\ cost + accounting\ cost \qquad (6.1)$$

它采用了整个计算的过程中每一个单体操作。创新的部分是要为每一个单体操作设计一个 *accounting costs* 系统，以达到下面两个目标。

1. 合法的操作序列中的每一个（从创建 ADT 对象开始分析起）操作的 accounting costsd 的和是非负。
2. 尽管 actual cost 可能在很大范围内波动，分析每一个 amortized cost 是可行的（例如，它相当的有规律）

如果这两个目标都达到了，则序列总的 *amortized* cost（从创建 ADT 开始）就是总的 *actual* cost，*且*总的 amortized cost 是经得起分析的。

直观的来说，总的 accounting cost 类似与储蓄账户。当好天气的时候，我们为雨季存储一些。当雨季来的时候，就是比较昂贵但是不常出现的的操作，we make a withdrawal。但是，为了保持偿付能力，不能忽视保持我们帐户的平衡。

设计一个 accounting costs 系统的主要思想是“正常”操作必须有一个正的 accounting cost，

例 6.1 用成倍增长的数组实现栈 accounting 方案

考虑栈 ADT，它有两个操作，push 和 pop，它以数组实现。（这个例子中，我们将忽略访问的代价，因为他们不改变栈，所以代价是O(1)。在必要的时候使用在第 6.2 小节中描述的数组翻倍来扩大数组。当没有发生数组大小变化时 push 或者 pop 的实际消耗都是 1。如果进行了数组翻倍（从 n 到了 2n）拷贝了 n 个元素到新的数组，则此时 push 的实际消耗是 1+nt 其中 t 是一个常数。（练习 6.2 考虑了一种方案，push 和 pop 都可能导致数组改变大小。）

最坏情况下 push 的实际时间是Θ(n)。看到这个最坏情况实际时间，似乎这个实现是及其低效的，因为这些操作都可能在Θ(1)时间内完成。但是 amortized 时间分析会给出一个更精确的结果。我们可以用下面的记账方案：

1. 不需要数组翻倍的 push 的 *accounting cost* 是 2t。
2. 需要将 n 扩展到 2n 的 push 的 *accounting cost* 是-nt+2t.
3. pop 的 *accounting cost* 是 0

accounting costs 中的系数 2 可以在栈创建的时候选到足够大，accounting costs 的和可以永不为负。非正式的用假设简化问题，假设在栈的大小为 N、2N、4N、8N…的时候会有数组翻倍发生。考虑最坏情况，只有 push 发生。“account balance”——accounting costs 的净和——会增长到 2Nt，之后第一个负的变化会将他减少到 Nt+2t，之后在第二次数组翻倍之前它

又会增长到 3Nt，到数组翻倍的时候又降到 Nt+2t。当它增长到 5Nt 时，会减少到 Nt+2t，增长到 9Nt 时会减少到 Nt+2t，等等。因此这时一个有效的栈 ADT 的 accounting 方案（）。

越复杂的数据结构通常需要越复杂的计数方案，就需要更创新的方法去想出来。在本章的后面的小节中（6.6.6 和 6.7.2 小节）我们将遇到需要平摊时间分析来展示其效率的 ADTs 以及实现

6.4 红黑树

红黑树是满足特定结构要求的二叉树。这些结构要求暗示有 n 个节点的红黑树的高不超过 2lg(n+1)。就是说，它的高比有 n 个节点的最平衡的二叉树多一个 2 倍常数因子。红黑树最流行的用处是做二叉查找树，但是这不是它唯一的应用。本节展示如何用红黑树来高效的保持平衡二叉查找树（平衡度刚被提及）。其他保持平衡二叉查找树的方案在本章后面的 Note 和 References 中提到。我们选择红黑树是因为其删除过程在所有选择方案中是最简单的。

在介绍了一些符号之后我们将复习二叉查找树。之后我们介绍红黑树要求的结构特性，并介绍如何在插入和删除的时候有效的维护之。

红黑树是一个 **RBtree** 类的对象，它的实现可能与 2.3.3 小节的 BinTree ADT 非常的类似；但是规范和接口非常不同。这是因为红黑树比以 BinTree ADT 实现的一般二叉树有更多的目的，并且有改变它结构的操作，而 BinTree ADT 没有定义这样的操作。同 BinTree ADT 一样，以 null 表示一颗空树。红黑树的操作有 rbtInsert、rbtDelete 和 rbtSearch。他们分别在树中插入，删除或查找给定的关键字。与 BinTree ADT 不同，红黑树没有提供直接访问子树的操作。但是在理解其子树是二叉查找树但是不一定是红黑树之后，就可以添加相应的操作。

RBtree 类非常适合实现字典 ADT，或其他需要平衡二叉树的 ADT。注意，红黑树是

Properly Drawn 树

Properly Drawn 树的思想可以帮助我们理解二叉查找树和红黑树的许多概念。本书用 properly drawn 树画所有的插图。

定义 6.1

一棵树在二维平面内是 properly drawn 的，如果：

1. 每一个节点是一个点，每一条边是一条线段或是一段连接父母到孩子的弧线。（在画图中一个节点是一个圈或是类似的插图，他们的“点”可以认为是中心，边认为从点出发。）
2. 一个节点左孩子和右孩子分别画在节点的左边和右边，水平位置。
3. 对于任意边 uv，u 开始于父节点，没有

空树作为 external 节点

对于二叉查找树，特别是对红黑树来说，将空树当作一种特殊类型的节点是方便的，空树的节点称为 *external node*。

6.4.1 二叉查找树

在二叉查找树中，节点的关键字满足下面的约束。
定义 6.3 二叉查找树 property

6.4.2 二叉树的旋转

6.4.3 红黑树的定义

6.4.4 红黑树的大小和深度

引理 6.2 令 T 是一个 RB_k 树。就是说，令 T 是一个有 black height h 的红黑树。则：

1. T 至少有 2^k-1 个 internal 黑节点。
2. T 至少有 4^k-1 个 internal 节点。
3. 任一黑结点的深度约两倍于它的 black depth。

令 A 是 ARB_k 树。就是说，令 A 是一个有 black height h 的近似红黑树。则：

1. A 至少有 2^k-2 个 internal 黑节点。
2. A 至少有 $\frac{1}{2}4^{h}-1$ 个 internal 节点。
3. 任一黑结点的深度约两倍于它的 black depth。

6.4.5 红黑树中插入

红黑树定义了基于颜色和 black height 的约束。理想的插入是在红黑树中插入一个红节点，这样就保证了 black height 的约束仍然是满足的。但是新的红节点可能会造成红节点成了红节点的孩子。我们可以通过合并颜色和结构来修复这个犯规且保持 black-height 的约束。

插入关键字 K 的第一步是在 BST 中查找 K 应该在的位置，当然因为 K 不在 BST 中所以会找到一个 external 节点（空树，参见算法 6.1）。下一步是将空树替换成仅包含一个节点 K 的树。最后一步，递归调用的出口，是修复颜色的犯规。任何时候都不会违反 black-height 的规范。

例 6.2 红黑树插入的第一步

在写出完整的算法之前，让我们考虑看看在图 6.5 的红黑树中插入关键字 70 会产生什么情况。在整个树中，70 与根比较，大，所以下降到右子树。然后 70 与 60 比较，接着下降到右子树，这里 70 与 80 比较。现在转到了左子树，且遇到了 external 节点。

6.5 哈希

哈希是一种常用来实现字典 ADT 的技术，尽管字典也可以用其他技术实现。想象一下在一个应用中，我们能给每一个可能关键字都赋予一个唯一的数组索引。这样，定位、插入、删除元素都是非常简单和快速的。

当然一般情况下，关键字空间（所有可能关键字的集合）都过于庞大。一个典型的例子字符串的关键字空间，比如人名。假设名字最多有 20 个字母或空格。关键字空间超过 2^{100} 个元素。就是说如果我们使用数组，将需要 2^{100} 个单元来为每一个字符串赋予一个索引，这完全是不可实现的。即使关键字空间是巨大的，在一个特定的应用程序中仅会碰到一小部分。实际元素的集合可能在几百之内，或者小于百万。一个有 4 百万个元素的足够给每一个元素一个不同的索引，这样规模的数组大小是可行的。

哈希的目的就是为了将巨大的关键字空间映射到一个合理的小的整数范围内。关键字转换之后的值称之为关键字的*哈希码*，计算转换过程的称为*哈希函数*。我们可以根据元素的哈希码用数组存储每一个元素。

“哈希”这个名字原来是根据早期的处理关键字的做法来的，即“chopping up”关键字，选取关键字的特定几位来做哈希码。

哈希函数的工作是以某种方式将关键字赋予一个整数，这种方式可能会失败的将一个“典型”的 n 个元素的集合中两个元素映射到同一个整数。当这种情况发生时，这个事件称为一个*碰撞*。为了减少碰撞出现的机会，如果我们有 n 个元素，典型的情况我们使用 2n 作为哈希码的范围。

最普通的，但是不是唯一的方式，使用哈希来维护一张*哈希表*。哈希表是一个索引 0, ..., h-1 的数组 H；就是说表有 h 个条目。H 的条目称为*哈希单元*。哈希函数将每一个关键字映射到 0,…,h-1 之间的整数。

例 6.9 哈希

对于一个简单的例子，假设关键字空间是 4 位整数，现在需要把他们转换到 0, …, 7 之间的整数。我们选择哈希函数：

$$hashCode(x) = (5x \bmod 8)$$

假设我们实际的集合由 6 个重要的历史日期组成：1055、1492、1776、1812、1918 和 1945.他们映射到 0,…,7 如下：

hash code	0	1	2	3	4	5	6	7
key	1775			1055	1492	1945	1918	
					1812			

如果我们有一个由 8 个条目组成的哈希表，元素可以根据他们的哈希码存储，他们将分散在整个表中。但是，有些元素有同样的哈希码，所以必须为这种事件做预防措施。在这个例子中，关键字 1492 和 1812 *碰撞*了。意味着他们映射得到了同样的哈希码。

两个议题为哈希表设计地址：哈希函数怎么设计，碰撞如何处理？两个议题相当的独立。找一个合适的哈希函数依赖于具体的应用。我们将讨论碰撞问题。

6.5.1 封闭地址哈希

6.5.2 开放地址哈希

6.5.3 哈希函数

6.6 动态等价关系和联合查找程序

动态等价关系出现在集合和图的各种问题里面。联合查找抽象数据类型为处理动态等价关系提供了一种方法。尽管它有非常高效（也很简单!）的实现，但分析是很麻烦的。

其应用包括最小生成树，在 8.4 小节讨论，以及本章最后提到的一些问题。

6.6.1 动态等价关系

集合 S 上的一个*等价关系* R 是一个 S 上的二元关系，且是自反，对称和传递的（1.3.1 小节）。就是说，对于所有的 S 中的 s，t 和 u，都符合下面的属性：如果 sRt 且 tRs，则 sRs; 如果 sRt 且 tRu 则 sRu。S 中一个元素 s 的等价类是 S 的一个子集，其中包含所有等价与 s 的元素。等价类是 S 的一个划分，就是说他们是不相交的，他们的联合是 S。符号≡将用来表示等价关系。

本小节的问题是表示、修改在计算过程中改变的等价关系以及回答相关的问题。等价关系开始是相等关系，就是说，集合中的每一个元素都与自己相等。问题是处理一个指令的序列，指令有两种类型，这里s_i和s_j都是 S 的元素：

1. $s_i \equiv s_j$

2. 使得$s_i \equiv s_j$（在$s_i \equiv s_j$并不是真的情况下）。

问题 1 的回答是“是”或“不是"。正确的回答依赖于已经收到的类型 2 的指令；回答是“是"当且仅当指令“使得$s_i \equiv s_j$”已经收到或是$s_i \equiv s_j$ can be derived by applying the reflexive, symmetric,and transitive properties to pairs that were explicitly made equivalent by the second type of instruction. The response to the latter, that is, the MAKE 指令, is to modify the data structure that represents the equivalence relation so that later instructions of the first type will be answered correctly.

考虑下面的例子这里 S={1, 2, 3, 4, 5}。指令的序列在左列。右列显示了回答——yes 或 no，或者是这里定义的关系的等价类。

开始时的等价类：{1}, {2}, {3}, {4}, {5}

1. IS $2 \equiv 4$? No
2. IS $3 \equiv 5$? No
3. MAKE $3 \equiv 5$ {1}, {2}, {3,5}, {4}

4. MAKE $2 \equiv 5$ {1}, {2, 3, 5}, {4}
5. IS $2 \equiv 3$? Yes
6. MAKE $4 \equiv 1$ {1,4}, {2, 3, 5}
7. IS $2 \equiv 4$? No

6.6.2 一些浅显的实现

为了比较不同实现策略，我们将计算每一个策略对于不同类型的操作的次数，假设我们要处理一个包含 n 个元素的集合 S，其中 MAKE 或 IS 指令的总数有 m 个。我们从检查两种简单的数据结构开始：矩阵和数组。

一个矩阵表示一个等价关系需要 n^2 个单元（或者已经是对称的情况下，需要 $n^2/2$）。对于一个 IS 指令只需要检查一个条目；但是一个 MAKE 指令可能需要拷贝整个行。一个 m 条 MAKE 指令的序列（考虑最坏情况有 m 条 MAKE 指令 1 条 IS 指令）可能需要至少 mn 步操作。

通过使用数组空间的使用数量可以减少到 n。eqClass，这里 eqClass[i]是一个包含等价类 s_i 的标签或是名字。指令 $s_i \equiv s_j$？需要查找和比较 eqClass[i]和 eqClass[j]。对于 MAKE $s_i \equiv s_j$，每一个条目都必须与 eqClass[i]比较，如果相等赋值为 eqClass[j]。此时，一个包含 m 条 MAKE 指令的 m（当然最坏情况）至少需要 mn 步操作。

两种方法都有低效率的一方面——第一个的拷贝过程第二个的查找过程（对于 eqClass[i]）里面的元素。更好的解决方案是使用连接来避免额外的操作。

6.6.3 联合－查找程序

MAKE 指令的效果是将 S 的两个子集合并成一个。如果我们有办法找到一个元素是否在一个集合中，我们就能简单的回答 IS 指令。联合查找 ADT（2.5.2 小节）提供了这些操作。一开始，对每一个 S 的元素运行 makeSet，得到 n 个单独的集合。查找和联合操作将用于实现下面的等价指令：

IS $s_i \equiv s_j$	MAKE $s_i \equiv s_j$
t=find(s_i);	t=find(s_j);
u=find(s_i);	t=find(s_j);
(t==u)?	union(t,u)

我们认为 create(n)是 Create(0), makeSet(1), makeSet(2), …, makeSet(n)的简写。这假定 S={1,…, n}。这个结果是一个集合的聚集，每一个包含一个单独的元素 i，$1 \le i \le n$。如果在程序运行期间需要一个一个的增加元素，不是一开始就包含所有的，我们假定 makeSet 运行在一个可数的序列上，是没有间隔的：makeSet(1), makeSet(2), ..., makeSet(k)。如果元素的数目不是自然树，字典 ADT（2.5.3 小节）可以用来进行翻译。

因此我们将注意力转到 makeSet，union 和 find 操作，以及可以简单实现他们特定的数

据结构。我们将通过 in-tree 来表示每一个等价类或子集。回顾 In-Tree ADT（2.3.5 小节）提供了下面的操作：

makeNode	构造一个只有一个节点的树
setParent	改变节点的父节点
setNodeData	为节点设置整数值
isRoot	如果节点没有父节点返回 true
parent	返回节点的父节点
nodeData	返回节点的数据

每一个根将被作为一个树的标签或标识符。指令 r=find(v)找到包含 v 的根，然后将它赋值给 r。union 的参数必须是根；union(t,u)将根 t 和 u（$t \neq u$）的树合并到一起。

In-Tree ADT 使得它实现 union 和 find 很简单。为了合并根 t 和 u 得到将 u 作为根的 in-tree，就像 union 要求的简单的执行 in-tree 操作 setParent(t,u)。为了找到节点的根，使用重复的使用 parent 操作，直到找到一个祖先它的 isRoot 操作返回 **true**。实现 create 和 makeSet 也是简单的，使用 in-tree 操作 makeNode 就可以。

如果 in-tree 的节点是 1, …, n，这里 n=|S|，我们可以在一个 n+1 个元素的数组中实现 in-tree。既然这是实践中的通常情况，而且为了更好的关注的本质问题，本小节剩下的部分我们将采用这个假设。我们可以“具体化”（un-abstract）in-tree，用简单的访问数组元素 parent[i]来代替调用 parent(i)和 setParent(i)。我们将一个节点的 parent 值设置为-1 表示其是 in-tree 的根，所以 isRoot 不用额外的数组。另一个数组保存 nodeData，但是这个名字对于这个应用过于宽泛了，所以我们将给出一个更具体的名字权（weight），预计 weighted union 方法将在后面描述。如果不知道会有多少元素时，可以使用数组翻倍（6.2 节）技术。

使用数组方便编码；但是，为了理解算法的逻辑，最好还是技术底层的 in-tree 结构，将数组元素的访问理解成 in-tree 的操作。使用数组实现 in-tree 将在好多算法中使用，所以它是值得记住的。

一个 create(n)操作（认为它是 n 个 makeSet）紧跟着 m 个任意顺序的 union 或 find 操作是输入，或者一个长度 m 的联合查找程序（Union-Find Program）。就是说一开始的 makeSet 不计算在输入的长度里面。为了简化讨论，我们假设在 create 之后没有 makeSet。如果 makeSet 在后面出现，也将得出相同的分析结论（练习 6.31）。

我们用访问 parent 的次数来衡量工作的多少；每一次访问不管是*查找*还是*赋值*，我们假设他们的时间都是O(1)。（这使得总的操作次数和 parent 的访问次数是成比例的这一点变的清楚。）每一个 makeSet 和 union 做一次 parent 赋值，每一个 find(i)做 d+1 次 parent 查找，这里 d 是节点 i 在树总的深度。parent 赋值和查找连起来叫做 *link* 操作。

图 6.19 的程序创建了图 6.20(a)显示的树，做了 n+n-1+(m-n+1)n 次 link 操作。使用这个方法的范例，在最坏情况下 Union-Find 程序是Ω(mn)。（我们知道 m>0；否则我们要写Ω(mn+n)。）不难证明没有程序能超过 mn+n 次 link 操作所以最坏情况是Θ(mn)。这与前面描述的方法比没有优势。我们将改进 union 和 find 操作的实现。

6.6.4 Weighted Union

图 6.19 中程序的消耗之所以高是因为由 union 指令构造的树，图 6.20(a)展示的，有太高的高度。通过更巧妙的 union 的实现可以保持树很矮，从而减少消耗。令 wUnion(“weighted union”）是策略：将节点数量少的树作为节点数量多的树的子树（同时，如果两个树的节点相同，令第一颗树是第二课树的子树）。（练习 6.22 检查了使用高度来代替节点数量做 weight 的可能性。）为了区别 union 操作的两个实现，我们叫第一个 unwUnion。对于 wUnion，每

一个树的节点数量存储在 weight 数组里面（对应 ADT 中的 nodeData）。实际上，仅根需要值。wUnion 必须比较节点的数量，计算新树的大小，为 parent 和 weight 赋值。wUnion 的消耗仍然是一个小的常数，包括一个 link 操作。现在如果我们返回到图 6.19（叫做 P）来看使用 wUnion 需要多少工作量，我们会发现 P 不在是一个正确的程序，因为 union 的参数从 3 到 n-1 都不是根了。我们可以扩展 P 到 P`来，需要将行如 union(i,j)的程序替换成下面 3 个指令： t=find(i); u=find(j);union(t,u);。则使用 wUnion，P`仅需要 2m+2n-1 次 link 操作！图 6.20 展示了使用 unwUnion 和 wUnion 的 P 和 P`构造的树。我们不能得出结论 wUnion 在任何情况下都是线型时间；P`不是 wUnion 的最坏情况。下面的引理帮助我们得到最坏情况的上界。

引理 6.6 如果 union(t,u)通过 wUnion 实现——就是说，根为 u 的树作为子树附加到根为 t 的树上，当且仅当根为 u 的树节点数量较少，否则就是根为 t 的树作为子树附加到根为 u 的树上——则，在任意序列的 union 指令之后，任何有 k 的节点的高最多是$\lfloor \lg k \rfloor$

证明 用 k 的归纳来证明之。k=1 的情况；一个节点的树是高度 0，就是$\lfloor \lg 1 \rfloor$。现在假设 k>1，并且任何由 union 指令构造包含 m 个节点的树，对于 m<k 有最多$\lfloor \lg m \rfloor$的高。考虑图 6.21 中的树 T，有 k 个节点高度是h，它由树 T1 和 T2 通过 union 指令构成。如图中的假设，T2 的根 u 附加到 T1 的根 t。令 k1 和 h1 分别是 T1 的节点数量和高度，同样的 k2 和 h2 分别是 T2 的节点数量和高度。通过归纳假设，$h_1 \le \lfloor \lg k_1 \rfloor$，且 $h_2 \le \lfloor \lg k_2 \rfloor$。新树的高度 $h = \max(h_1, h_2 + 1)$。显然，$h_1 \le \lfloor \lg k \rfloor$。因为 $k_2 \le k/2$，$h_2 \le \lfloor \lg k \rfloor - 1$ 且 $h_2 + 1 \le \lfloor \lg k \rfloor$。所以两种情况下都有 $h \le \lfloor \lg k \rfloor$。

定理 6.7 一个作用在 n 个元素上的，规模 m 的 Union-Find 程序，如果使用 wUnion 和直接查找，最坏情况下要执行 $\Theta(n + m \log n)$ 次 link 操作。

证明 对于 n 个元素，创建一个最多有 n 个节点的树，最多有 n-1 次 wUnion 指令可以用。由于引理，每一颗树至少有高度$\lfloor \lg n \rfloor$，所以每一个查找最多是$\lfloor \lg n \rfloor + 1$。每一个 wUnion 做一次 link 操作，所以 m 个 wUnion 或 find 操作花费的上界就是 m 个 find 操作的花费。因此 link 操作的总的数量小于 $m(\lfloor \lg n \rfloor + 1)$，是 $O(n + m \log n)$。

练习 6.23 的程序需要Ω(n+mlogn)步来构造。

wUnion（与 create 和 makeSet 一样）的算法非常容易写；我们留在练习中。

6.6.5 路径压缩

通过称为路径压缩的过程还可以改进 find 操作的实现，从而更进一步的提高 Union-Find 程序的执行速度。给出参数 v，cFind（“压缩（compressing）查找”）parent

wUnion 和 cFind 的兼容性

*6.6.6 wUnion 和 cFind 的分析

6.6.7 应用

6.7 优先级队列 with a Decrease Key Operation

回顾优先权队列 ADT（2.5.1 小节）的主要 access 函数是 **getBest**，这里 best 可能是最大或者最小。一个完整的最小化优先权队列 ADT 的操作有：

构造器： **create**

Access 函数： **isEmpty**, **getMin**, **getPriority**

Manipulation 过程： **insert**, **deleteMin**, **decreaseKey**

这些名字适合最大化优先权队列。还可以增加一个 **delete** 操作，这个操作删除任意关键字。

Partial order tree（定义 4.2）是经常用于实现优先权队列的一类数据结构。“最好”元素在 partial order 树的根，所以它可以在常数时间内得到。Partial order 数的几种实现已经发展了很多年了。“堆”这个词经常出现在这些实现的名字中，因为最早的 partial order 树的结构被它的发明者命名为“堆”。

二分堆

6.7.1 The Decrease Key Operation

第七章 图和图的遍历

7.1 概述

大量的问题可以转换成某种图的问题。这些问题不仅来源于计算机的相互连接，而且来自科学、工业、商业。解决许多图问题的有效算法的发展，对人类解决所有这些领域中实际问题的能力有重要的影响。然而，许多重要的图问题没有有效的解决。对于已有解决的，也不能肯定当前已知的解决方法是否是最有效的或是将来能不能有本质的提高。

这一章中我们将引入图的定义和基本属性。之后我们将介绍遍历图的主要方法。以图的

遍历为基础，可以证实许多自然的问题都可以非常有效的解决——在线性时间之内。不严格的将，我们可以称之为“简单”图问题，这不仅是说解决方法可以简单的找到或是可以简单的写出程序，而且指一旦写出程序可以解决非常大的实例（比如图有上 1M 个节点）。

继续我们不严格的分类，“中级”图问题需要多项式时间解决，但是需要比一次图遍历更多的工作量。就是说，对于每一个“中级”问题，算法在一个固定幂的多项式时间内解决问题，比如 n^2 或 $n^d d$ 是一个固定数。许多此类问题的在后面的章节中讲到；参见第 8，9，14 章。在今天强劲的计算机上解决比较大的问题是可以实现的（比如上 k 或 10k 个节点）。

当然我们有“困难”图问题，目前不知道有在多项式时间内解决的方法。此类问题当图有 50 个或是 100 个节点时就没有已知的方法可以解决了，即使允许计算机运行一年的时间。然而，我们当前的知识不能排除找到有效算法的可能性。这些问题真正代表了我们知识的前沿，他们将在第十三章的讨论他们中的一些。

图问题一个迷人的方面是：只要稍微改变一下问题的形式，可能问题就会改变其分类简单，中级或是困难。因此一个问题和已知问题的类似程度，以及改变一个问题的那个方面会使的他们变的简单，中级或是困难对于解决一个新问题是很有帮助的。

7.2 定义和表示

非正式的，一个图是点（顶点或节点）的有限集合，他们中的一部分通过线或是箭头（边）连接起来。如果边是无向的或是“双向的”，则图是*无向图*。如果边是有向的或是“单边的”，则图是*有向图*。“有向图”一半简写成 *digraph*。“无向图”有时简称“图”，但是这有时候会引起误解因为人们经常将无向图和有向图都简单的叫“图”。我们经常在可能有歧义的上下文中使用特定的属于。一般的讨论“图”既指无向图也指有向图。

7.2.1 一些例子

图是许多问题和结构的形象的抽象，包括运筹学、计算机科学、电子、经济、数学、物理、化学、通信、博弈论和许多其他的方面。考虑下面的例子

例 7.1 航空路线图

一个航空路线图可以用无向图表示。点表示城市；两个城市用线连接当且仅当一个中途不停的飞机在他们之间相互通行。图 7.1（假想的）是 California 几个城市的航空路线。

例 7.2 流程图

流程图表示了一个过程内的控制流，或是一个过程中数据或物质的流动。流程图的框是点；流程图的箭头是边。图 7.2 以 Pascal 语法显示了一个例子。

例 7.3 二元关系

二元关系在 1.3.1 小节中定义。定义 R 是集合 S={1,…,10}上的由有序对(x,y)组成的二元关系，其中 x 是 y 的整数倍；就是说 $x \neq y$ 且 y/x 的

余数是 0。回忆 xRy 是 $(x, y) \in R$ 的一种表示。在图 7.3 的有向图中，点是 S 的元素，当且仅当 xRy 时 x 和 y 之间有连接边。注意 R 是传递的：如果 xRy 且 yRz，则 xRz。

例 7.4 计算机网络

点是计算机。线（无向图）或箭头（有向图）是通信线路。图 7.4 展示了两个例子。

例 7.5 集成电路

点可以是二极管、三极管、电容、开关等等。如果两者之间有电路连接则有线。

前面 5 个例子已经展示了：无向图和有向图提供了一种直接的对分离对象的关系的抽象形式，包括物理对象和他们的关系（比如，通过航空线路、高速公路和铁路连接的城市）以及抽象对象的关系（比如二元关系和程序控制结构）。

这些例子同时提出了一些我们可能希望回答的关于所表示的对象以及关系的问题，这些问题可以转换成图的术语。这样的问题可以通过工作在图上的算法回答。例如，问题“在 San Diego 和 Sacramento 之间有直通航线吗？”可以转换成“图 7.1 中顶点 SD 和 SAC 和有边吗？”考虑下面的问题：

1. 从 San Diego 飞到 Sacramento 最便宜的路线是那条？
2. 那条路有最短的飞行时间？
3. 然后一个城市的机场因为坏天气而关闭了，你仍然可以在每对城市之间飞行吗？
4. 如果网络中的一台电脑 down 掉了，网络中的每对计算机之间还能传递消息吗？
5. 一个点到另一个点之间道路的流量有多大？
6. 一个给定的二元关系是可传递的吗？
7. 给定一个流程图有循环吗？
8. 如何连接不同的电路端口使得连接所有的需要的电线最少？

本章和下面的章中我们将学习算法回答大多数问题。

7.2.2 基本的图定义

本小节主要是图的定义和基本注意。许多申明和定义都同时适用于有向和无向图，我们使用公用的符号表示两者共同的部分。当然，无向和有向图不一样的部分我们将特别注明。

定义 7.1 有向图

一个*有向图*或 *digraph* 是一个对 G=(V,E)这里 V 是一个集合，它的元素是*顶点*，E 也是一个集合它的元素是 V 元素的有序对。顶点通常也叫*节点*。E 的元素叫*边*，或是*有向边*，或是弧。对于 E 中的有向边(v, w)，v 叫做尾，w 叫做头；(v,w)表示为一个有向箭头 v→w。在文本中我们简单的写作 vw。

在二元关系例子中（例 7.3，图 7.3），

$V = \{1,2,...,10\}$

$$E = \{(1,2),...,(1,10),(2,4),(2,6),(2,8),(2,10),(3,6),(3,9),(4,8),(5,10)\}$$

定义 7.2 无向图

一个*无向图*或 *graph* 是一个对 G=(V,E)这里 V 是一个集合，它的元素是*顶点*， E 也是一个集合它的元素是 V 元素的无序对。顶点通常也叫*节点*。E 的元素叫*边*，或是为了强调叫做*无向边*。每一个边都可以认为是包含两个元素 V 的子集；因此符号{v, w}表示无向边。用 v-w 表示。在文本中我们简单的写作 vw。当然对于无向图 wv=vw。

例如对于例子 7.1 和图 7.1，我们有

$$V = \{SF, OAK, SAC, STK, FRES, LA, SD\}$$

$$E = \begin{Bmatrix} \{SF,STK\}, & \{SF,SAC\}, & \{SF,LA\}, & \{SF,SD\}, & \{SF,FRES\}, & \{SD,OAK\}, \\ \{SAC,LA\}, & \{LA,OAK\}, & \{LA,FRES\}, & \{LA,SD\}, & \{FRES,STK\}, & \{SD,FRES\} \end{Bmatrix}$$

无向图的定义暗示了不存在连接顶点自己的图：边定义为包含两个元素，一个集合不能有重复的元素（定义 1.3.1）。

定义 7.3 子图,对称图,完全图

一个图 G=(V,E)的子图 G'=(V',E'),V'⊆V,E'⊆E.由图的定义还需要满足 E'⊆V'×V'.

对称图是这样的有向图:右边 VW 则必有反方向边 WV.每个无向图有对应的对称图,把无向边解释成相反方向的一对有向边.

完全图(通常是无向图)是每对顶点都有边的图.

边 VW 可说它是依附于顶点 V 和 W 的.

7.2.3 图的表示和数据结构

邻接矩阵

邻接表数组

7.3 图的遍历

大部分算法是为解决检查或处理图的顶点和边的问题的.广度优先查找和深度优先查找提供了一种高效访问每个顶点和边仅一次的访问策略.(术语深度优先查找和深度优先遍历是可互换的,广度优先查找和广度优先遍历也是类似的).从而,许多算法以此为基础运行在随输入图的大小呈线性增长的时间里.

7.3.1 深度优先查找

例 7.6 深度优先查找
让我们从下图中的顶点 A 开始深度优先查找。为了简化，当我们有多条边可以探索时，我们将按字母顺序选择一条

7.3.2 广度优先查找

7.3.3 深度优先查找和广度优先查找的比较

7.4 有向图的深度优先查找

7.4.1 深度优先查找和递归

7.4.2 使用深度优先查找连通分量

7.5 有向图的强连通分支

7.6 无向图的深度优先查找

7.7 无向图的二连通分支

在 7.2 节，我们提出了这些问题：

1. 如果一个城市的机场因为坏天气而关闭了，你仍然能在其他的任意两个城市之间飞行吗？
2. 如果网络中一台计算机崩溃了，消息仍然能在其他的任意两台网络中计算机之间传递吗？

本节中明我们仅考虑无向图。作为一个图问题：

问题 7.1

如果任一顶点（与之相关联的边）从连通图中移除了，剩下的子图仍然是连通的吗？

这个问题在代表通信或运输网络的图中是非常重要的。如果移除之后就使得图不在连通，找出这些顶点也是很重要的。本节的目的就是找到有效的算法来回答这些问题。这个算法是由 R.E.Tarjan 发现的，这个算法也是早期体现出深度优先查找的强大威力的算法。

7.7.1 关节顶点和二连通分支

我们从建立一些技术和基本属性开始。

定义 7.21 二连通分支

连通无向图 G 是*二连通*的，当移除它的任一顶点以及与顶点相连的边之后，图仍然是连通的。

无向图的一个*二连通分支*（简写为 bicomponent）是最大的二连通子图，就是说，不被更大的二连通自图所包含。

显然，移除一个关节顶点会留下一个不连通的图，所以一个连通图是*二连通*的，当且仅当它没有关节顶点。观察这一点，尽管二连通分支将边划到不相交集合，但是并不严格得划分顶点；有些顶点在多个分支里面。（那些顶点是？）

第八章 图的最优化问题和贪婪算法

8.1 概述

本章中，我们将学习许多图的可以被贪婪算法精确解决的最优化问题。典型的，在最优化问题中需要做一系列的选择，以取得最小的消耗，或者得到最大的收益。贪婪方法由一系列选择组成，每一个单独的选择都使结果向“短期”（译注：局部的）最优解靠近，通常局部最优解不需要很多花费就可以得到。一旦作出了一个选择，它不能被撤销，即使都后面发现它其实是一个糟糕的选择。由于这个原因，贪婪算法对很多问题不能找到精确的最优解。但是，对于本章讨论的问题我们可以证明适当的贪婪策略会产生最优的解决方案。在第十三章中，我们将看到非常类似的贪婪方法无法得到一些问题的解。在第十章中我们也将看到一些贪婪方法不能解决的问题。

本章首先给出一个无向图中找最小生成树的算法，由 R.C.Prim 给出；一个很相关的在有向和无向图中找最短路径的算法，由 E.W.Dijkstra 给出；以及第二个找最小生成树的算法，由 J.B.Kruskal 给出。所有的算法使用优先权队列来从候选集合中选择当前最好的选择。

8.2 Prim 最小生成树算法

第一个问题我们将学习找一个连通、带权、无向图的最小生成树。对于非连通图，很自然的就可以扩展到每一个连通分支。我们在 7.4.2 小节看到，连通分支可以在线型时间内找到。

最小生成树仅对无向带权图有意义，所以本小节虽有的“图”都指“无向图”，权都是边的权。复习符号 G=(V, E, W)，W 是一个函数，它给 E 的每一个边赋予一个权。这仅是数学上的描述。在实现中，一般没有“函数”；每一条边的权简单的存储在边的结构里面。

8.2.1 最小生出树的定义和例子

定义 8.1 最小生成树

连通无向图 G=(V, E)的*生成树*是 G 的一个子图，它是一颗无向树且包含 G 的所有顶点。在带权图 G=(V, E, W)上，子图的权是子图所有边的权的和。带权无向图的最小生成树（简写为 MST）是权最小的生成树。

有很多情况必须找最小生成树。无论什么时候想找最便宜的方式连接一些终端站点。，比如城市、电子终端、计算机、工厂使用路，电线或是电话线，解决的方法都是找最小生成树，其中边是所有可能的连接，权是可能连接的花费。找最小生成树也是各种路由算法的重要子问题，即访问图的每一个顶点的有效的路径。

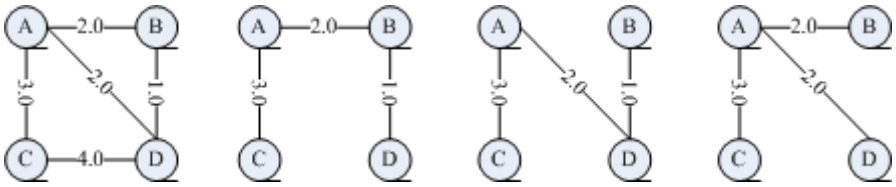


图 8.1 一个图和它的一些生成树：其中两个是最小生成树。

就像图 8.1 中展示的简单例子，一个带权图可以有许多最小生成树。事实上，这个例子中将一个最小生出树转换成另一个的方法在 8.2.3 小节中讨论，这个图是后面讨论最小生成树时一直使用的图。

8.2.2 算法的概况

因为无向图是连通的，且任意节点都可以认为是根，一个自然的找最小生成树的想法是每次给起始顶点“增加”一条边。我们必须首先使用标准的遍历算法，深度优先和广度优先遍历算法。如果我们可以使用其中的一个框架解决这个问题，则我们就可以在线性时间解决问题，这肯定是最佳的。你必须花点时间尝试使用这些搜索算法的思想，并构造标准搜索算法不能找到最小的例子（练习 8.1）

使我们相信简单的遍历似乎不能解决问题的时候，考虑到这是个最优化问题，下一个自然的想法使尝试*贪婪方法*。贪婪方法的思想是：不断选择取短期消耗最小的行动，以期望大量的短期最小消耗加起来得到一个小的总体消耗。（可能的回溯是指：小的短期消耗可能导致下一步不可避免的会是一个很大的消耗。）我们会有一个很自然的想法，即在我们增加边的时候增加短期消耗最小的边：简单的增加可以增加到树的边（译注：不会将树变成图），且有最小的权。Prim’s 算法就是这样一个贪婪实现。

有了解决问题的思路，还有两个问题。它工作正确吗？它运行的有多快？我们已经提及，一系列短期消耗可能导致到不好的情况，所以即使我们得到一颗生成树，我们还需要考虑它的权是不是所有生成树中最小的。同样，既然我们需要在每一步都从每一条边中选一条，而且候选集合在每一步之后都会变，我们需要选一种数据结构使得这些操作比较高效率。在解决了通用思想后，我们将回到这些问题。

Prim’s 算法开始要任意的选一开始顶点，在通过不断选出新的顶点和边“分支出”树的部分。新边连接新的顶点到前面生成的树。在整个算法过程中，顶点可以划分成 3 个不相交的集合：

1. *树*的顶点：当前在树中的，

2. *边缘*的顶点：不在树中，但是与树中的顶点邻接，
3. *没有检查*的顶点：剩下的全部。

算法关键的步骤是从边缘顶点中选一个顶点和附带边。实际上，因为权是边的权，选择的焦点在边，不在顶点。Prim's 算法总是选择最小权的从树的顶点到边缘的顶点的边。算法的一般结构如下：

primmest(G, n)

初始化所有的顶点为*没有检查*。
选择任意顶点 s 开始树；将它划分到*树*。
划分所有与 s 邻接的顶点为*边缘*。
当有边缘顶点时：

选择一条权最小的从树顶点 t 到边缘顶点 v 的边；
将 v 划分到*树*；添加边 tv 到树；
将所有与 v 邻接的*边缘*顶点划分到*没有检查*的顶点。

例 8.1 Prim's 算法，一次迭代

图 8.2(a)展示了一个带权图。假设 A 是开始的顶点。循环开始之前的步骤导致到图 8.2(b)。在第一次循环的迭代之后，边缘顶点中的最小权边是 AB。因此 B 被添加到树，与 B 邻接的没有检查顶点划分到边缘顶点，导致了图 8.2(c)。

我们能确定这个策略会导致一个最小生成树吗？在短期贪婪的策略是一个好的长期策略吗？在这种情况，是。下两个小节讨论所有最小生成树的公共特性，使用这些特性展示 Prim's 算法每一步构造的树都是这颗树的子图上的最小生成树。在 8.2.5 小节我们将讨论实现的问题。

8.2.3 最小生出树的属性

图 8.1 展示了一个带权图可能有多个最小生成树。事实上，最小生成树有一个公共特性，用这个特性我们可以将一颗最小生成树一步一步转换成另一颗最小生成树。考察这些特性也有助于我们更倾向无向树（also help us to become more familiar with undirected trees, generally）。

定义 8.2 最小生成树特性

给出连通带权图 G=(V, E, W)，令 T 是 G 的任意生成树。假设 G 的每一条不在 T 中的边 uv，如果 uv 增加到 T，使得产生了一个回路，而 uv 是这个回路中权最大的边。则树 T 有*最小生出树属性*（简称为 MST 属性）。

首先，让我们看看定义的意思。之后我们将证明这个名字是很恰当的（译注：即符合这一特性的确实是最小生成树）；仅仅是叫“最小生成树属性”并不意味着可以不作任何事情就说它是最小生成树！

例 8.2 最小生成树属性

根据定义一颗无向树连接树中的任意两个顶点，且没有环。图 8.1 就是一个简单图，我们称之为 G，还有 3 颗生成树。首先另 T 是（b）代表

的树。假设我们增加一条 G 的不属于 T 的边到 T 上，得到新图 G_1（我们将增加权值是 2 的那条）。这产生了一个回路（G_1 中唯一的回路）。（为什么呢？译注：生成树两个顶点间必然有一条路径，增加一条不属于生成树的路径，必然和生成树中的路径构成回路）回路中其他的边的权值最多是 2，即不超过加入边的权值。可选的，如果我们加入权值 4 的边，则回路中其他边的权值最多是 4（事实上是 4）。这里只有两条可选的边，所以 T 有最小生成树属性。（c）代表的树也是一样。

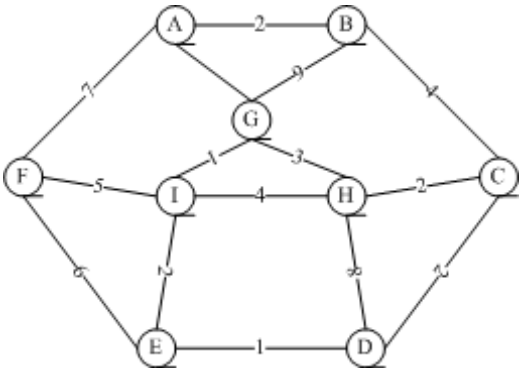
但是，现在考虑令 T 是让（d）代表的树，增加权值 1 的边。这次回路中有些边的权值就超过 1 了。因此（d）代表的树*不符合*最小生成树属性。注意我们可以去掉回路中任意一条边得到新图 G_2，G_2 也必然是一颗树，事实上还是一颗生成树。在练习 8.2 中证明了这个事实，这个事实用于证明下面的引理和定义。既然有一条边的权值大于 1，那么我们选择去掉这条权值大于 1 的边。这意味着 G_2 的权值要小于 T，所以 T 不是最小生成树。

引理 8.1 在一个连通，带权图 G=(V, E, W)中，如果 T1 和 T2 是两颗符合 MST 属性的生成树，则他们有相同的权。

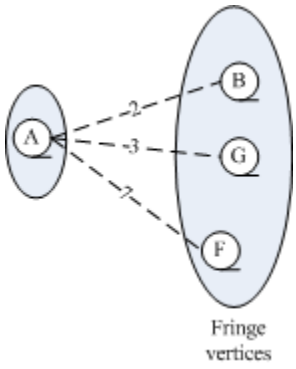
证明 令 k 是在 T1 中但是不在 T2 中的边的数目，证明用归纳 k 的方式。（在 T2 中不在 T1 中的边的数目 k 也是类似的。）基本情况是 k=0；在这种情况下，T1 和 T2 是相同的，所以他们有相等的权。

对于 k>0，令当 0 到 k-1 条边都不同时，引理是符合的，则当有 k 条边不同时。令 uv 是在 T_1 中或 T_2 中但是不同时在两者中的最小权的边。假设 $uv \in T_2$; $uv \in T_1$ 是等价的。考虑 T_1 中 u 到 v 唯一的路径：$w_0, w_1, \ldots$ wp，这里 w_0=u，w_p=v，$p \geq 2$。这条路径必然包含有不在 T_2 中的边。（为什么？译注：因为 $uv \in T_2$，如果 T_1 中 u 到 v 的边都在 T_2 中，则 T_2 中存在两条从 u 到 v 的路径，与 T_2 是树的属性不符。p 必须不小于 2 是同样的道理。）令 w_iw_{i+1} 就是这样的边。由 T_1 的 MST 属性知，w_iw_{i+1} 的权*不大于* uv 的权。但由于 uv 是不同时在 T_1T_2 的权最小的边，所以 w_iw_{i+1} 的权*不小于* uv 的权。因此 $W(w_iw_{i+1})=W(uv)$。增加 uv 到 T_1 形成回路，则移出 w_iw_{i+1}，打开回路，得到新的生成树 T_1'，新的生成树与 T_1 有相同的权。但是 T_1' 与 T_2 仅有 k-1 条边不一样，由于归纳假设，两者有相同的权，因此 T_1 和 T_2 有相同的权 。

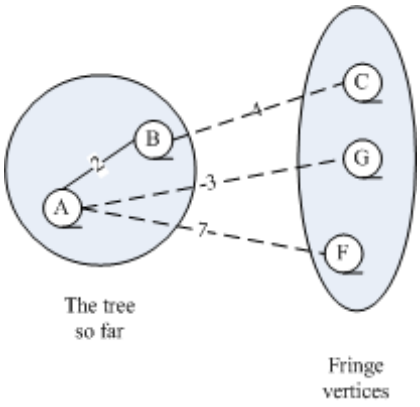
引理的证明也向我们展示了一种一步一步将任何最小生成树 T_1 转换成另一颗最小生成树 T_2 的方法。选择一条 T_2 中但不在 T_1 中权最小的边 uv。找到 T1 中 u 到 v 的路径。路径中肯定有一条不在 T2 中但权与 uv 相等的边，xy。（参见练习 8.3）。移出 xy 增加 uv。这样就朝 T_2 前进了一步。重复，直到得到 T_2。



（a）带权图



（b）开始选择时树和边缘顶点



（c）在选择一条边和顶点之后：BG 没有画出，因为 AG 是到达 G 更好的选择（有更低的权）

图 8.2 Prim's 算法一次迭代之后：实线表示树的边，虚线表示到边缘顶点的边

定理 8.2 在连通带权图 G=(V, E, W)，树 T 是最小生成树当且仅当 T 有 MST 属性。

证明 （仅当）假设 T 是 G 的最小生成树。假设存在一条不在 T 中的边 uv，将 uv 添加到 T 之后构成回路，回路中有另一条边 xy 使得 W(xy)>W(uv)。则移出 xy 得到新的生成树，新的生成树的权小于 T，与 T 是最小生成树的假设矛盾。

（当）假设 T 有 MST 属性。令 T_{min} 是 G 的*任意*最小生成树。由前面的证明可知 T_{min} 有 MST 属性。由引理 8.1，T 和 T_{min} 有相同的权。

8.2.4 Prim's MST 算法的正确性

我们现在使用 MST 属性来展示 Prim's 算法确实构造了一颗最小生成树。这个证明用了一种在采用归纳时经常出现的形式：通过归纳证明的语句有时比我们感兴趣的定理更详细。所以我们首先将这更详细的语句作为*引理*证明。之后定理简单的吸取引理感兴趣的部分。以这种观点，定理非常象递归过程的“包装”，就像 3.2.2 小节讨论的。

引理 8.3 令 G=(V, E, W)是连通的带权图，n=|V|；令 T_k 是一个由 Prim's 算法创建的有 k 个顶点的树，其中 k=1, …, n；令 Gk 是 G 中 T_k 的顶点组

成的子图（例如，uv 是 Gk 的一条边，当这条边在 G 中，且 u、v 都在 T_k 中。）则在 Gk 中 T_k 有 MST 属性。

证明 证明是对 k 的归纳。基本情况是 k=1。这种情况下，G_1 和 T_1 仅包含起始节点 s 没有边， 所以在 G_1 中 T_1 有 MST 属性。

对于 k>1 时，假设对于 1≤j<k 都有在 G_j 中 T_j 有 MST 属性。假设 Prim's 算法加入增加到树中的第 k 个顶点是 v，v 到 T_{k-1} 中顶点的边分别是 $u_1v,\ldots, u_dv$。For definiteness，假设 u_1v 是最小的边，因此被算法选中。我们需要验证 T_k 有 MST 属性。就是说，如果 xy 是任意在 G_k 中但不在 T_k 的边，则将 xy 加入 T_k 中后形成的回路中 xy 是权最大的边。如果 x≠v 且 y≠v，则 xy 也在 G_{k-1} 中，但是不在 T_{k-1} 中，所以根据归纳假设，它增加到 T_{k-1} 之后是回路中权最大的边。但是这个回路同样是 T_k 的回路，所以这种情况下 T_k 也有 MST 属性。剩下的就是检查 xy 是 $u_2v,\ldots, u_db$ 其中一条时（因为 u_1v 已经在 T_k 中了）。对于 d<2 时我们已经作了，现在假设 d≥2。

引用图 8.3 会在剩下的证明中帮助我们的。考虑 v 到 T_k 中 u_i 路径，对于任意 i，2≤i≤d。假设这些路径上存在边的权大于 u_iv 的权，至少有 u_1v 的权（如果不存在，MST 属性就满足了。）特别的，令路径是 $v, w_1,\ldots, w_p$，这里 $w_1=u_1$ 且 $w_p=u_i$。则 $w_1,\ldots, w_p$ 是 T_{k-1} 中的路径。令 w_aw_{a+1} 是路径中*第一条*权比 $W(u_iv)$大的边，令 w_bw_{b+1} 是路径中*最后*一条权比 $W(u_iv)$大的边（可能 a+1=b；看图 8.3）。我们认为如果 w_a 和 w_b 是由 Prim's 算法构造的话，则 w_a 和 w_b 不能存在于 T_{k-1}。假设 w_a 在 w_b 之前加入树。则从 w_1（就是 u_1）到 w_a 的路径上所有的边将在 w_aw_{a+1} 和 w_bw_{b+1} 之前被加入，因为他们有更小的权，且 u_1v 也将在他们之前加入。类似的，如果 w_b 在 w_a 之前加入树，则 u_iv 也将在 w_aw_{a+1} 或 w_bw_{b+1} 之前加入。但是 u_1v 或者 u_iv 都不在 T_{k-1} 中，所以 $w_1,\ldots, w_p$ 中没有边的权大于 $W(u_iv)$，T_k 符合 MST 属性。

定理 8.4 Prim's 算法输出最小生成树

证明 在引理 8.3 的术语中，$G_{n=G}$ 和 T_n 是算法的输出，所以 T_n 在 G 中有 MST 属性。由定理 8.2 知，T_n 是 G 的最小生成树。

8.2.5 用优先权队列有效的管理边缘

在算法循环的每一个迭代之后，就可能有新的边缘顶点，而且下一次待选择的边集合也会改变。图 8.2(c)建议我们需要考虑树的顶点和边缘顶点之间所有的边。在选择 AB 之后，BG 成了一个待选项，但是要丢弃 BG 因为 AG 的权值比 BG 低，AG 将是到达 G 的更好的选择。如果 BG 的权值比 AG 低，则丢弃 AG。对于每一个边缘顶点，我们需要保持到树的一条边即可：权值最小的那条。我们称这样的边为*候选边（candidate edges）*。

优先权对立 ADT（2.5.1 小节）恰好有我们实现 8.2.2 小节算法所需要的操作。Insert 操作加入一个顶点到优先权队列。getMin 操作可以用于选取边缘顶点，

使用优先权队列 ADT 操作，高层算法如下。一个子例程 updateFringe 用来处理顶点邻接

初步分析

在不知道优先权队列是如何实现时，我们能算法的运行时间吗？第一步是求出每一个 ADT 操作消耗多少时间。然后我们可以写出表达式

8.2.6 实现

算法主要的数据结构（除了图本身以外）

8.2.7 分析（时间和空间）

我们现在完成了算法 8.1 在 G=(V,E,W)的分析。

8.2.8 The Pairing Forest Interface

8.2.9 低限

到底找到最小生成树需要多少工作？我们宣称任何一个最小生成树算法在最坏情况下需要Ω(m)，因为它必须以某种方式处理图的每一条边。令 G 是一个连通的带权图，每一条边至少有权值 2，并假设有一个算法对 G 中的边 xy 完全不做任何事情。则 xy 肯定不在算法输出的树 T 中。将 xy 的权改为 1。这可能不改变算法的行为，因为算法不检查 xy。但是由定理 8.2 知，现在 T 不再符合 MST 属性，它不再是一颗最小生成树。事实上，为了得到一颗更“轻”的树，只需要简单的 xy 加入 T 中，然后移除回路中的另一条边就可以了。因此算法不检查 xy 是错误的。

8.3 单源最短路径

在 7.2 小节中我们简要的考虑了在航空路线图上如何找到两个城市间最好的路线的问题，比如图 7.8。使用的标准是我们关心的机票的价格，我们观察最好的——就是说最便宜的——从 San Diego 到 Sacramento 的方式是在 Los Angeles 中转一下。这是一个非常普通的带权图的问题的实例，或叫应用：在两个特定的顶点之间找到最小权的路径。

最坏情况下，基本上成了两个问题：找到一对特定顶点 s 和 t 之间的最小权路径；找到从 s 到每一个可以从 s 到达的顶点的最小权路径。这两个问题一样的复杂。后一个问题叫做*单源最短路径问题*。两者采用同样的算法。

本节考虑在带权有向图和无向图上找从一个源顶点到每一个其他顶点最小权路径的问题。路径的权（长度或是耗费）是路径上所有边的权的和。当权被解释成距离时，一个最小权路径叫做最短路径，而且这是最常用的名字。（唉，它混合了术语权，耗费和长度。）

我们是如何确定图 7.4 中 SD 到 SAC 的最短路径的呢？事实上，对于这个简单的例子我们使用了很不规则的方法，充满了假设，比如城市之间的费用和距离是成比例的，地图是按比例精确画的。则我们捡了一条*看上去*短的路线，然后加起来了耗费。最后，我们检查了许

多其他的路线（somewhat haphazardly）没有注意到有改进，所以我们宣布问题解决了。这是一个很难用于计算机编程的算法。我们发现这确实能诚实的回答上面的问题；人们一般使用非常不严格的方式来解决问题，特别是对非常小的数据集合。

实践中，当 V 中包含上百，上千，甚至百万个顶点的时候，在图中找最短路径的问题就麻烦了。理论上算法必须考虑所有可能的路径而且比较他们的权值，但是实践中这可能需要很长的时间，可能有几个世纪。为了尝试找到更好的方法，找到最短路径的一些通用的性质是很有帮助的，然后再来看他们是否能发现一个更有效的方法。我们展示的算法是 E.W.Dijkstra 提出的。这个算法需要边的权都是非负的。其他一些不需要这个限制的算法在本章后面的 Notes 和 References 中提到。

8.3.1 最短路径的属性

一般的，当尝试解决一个大问题的时候，我们想把它分解成小的问题。What can we say about shortest paths between distant nodes, in terms of shortest paths between less distant nodes? 我们可以使用某种形式的分而治之方法吗？假设路径 P 是 x 到 y 的最短路径而 Q 是 y 到 z 的最短路径。那么是否意味着 P 接上 Q 就是 x 到 z 的最短路径？不用很长的时间就可以找出一个例子来说明它不对。但是它的逆命题是对的。下面的引理的证明留到练习中。

> **引理 8.5** （最短路径的属性）在带权图 G 中，假设 x 到 y 的最短路径由从 x 到 y 的路径 P 和从 y 到 z 的路径 Q 组成。则 P 是 x 到 y 的最短路径，Q 是 y 到 z 的最短路径。

假设我们要尝试找 x 到 z 的最短路径。可能一条直接的边 xz 存在，且提供了最短的路径。但是，如果最短的路径包含 2 条或更多的边，则引理告诉我们它可以被分解成两条路径，每一条比原来包含的边少，每一条都是自己的最短路径。为了发展这个算法，我们需要建立一些分解路径的有组织的方案。

例 8.3

8.3.2 Dijkstra's 最短路径算法

本小节中我们学习 Dijkstra's 最短路径算法；它非常类似与前一小节的 Prim's 最小生成树算法。

> **定义 8.3**
>
> 令 P 是一条带权图上 G=(V, E, W)上的非空路径，由 k 个边 xv_1，v_1v_2，…，$v_{k-1}y$（可能 $v_1 = y$）。P 的权，表示为 W(p)是权 $W(xv_1)$，$W(v_1v_2)$,…，$W(v_{k-1}y)$。如果 x=y，认为 x 到 y 有空路径。空路径的权是 0。
>
> 如果 x 到 y 没有路径的权小于 W(p)，则 P 称为最短路径，或是最小权路径。

前面的定义小心的表达了负权的可能性。但是，在本节中我们假定权是非负的。在这个限制下，最短路径可以限制成一条简单路径。

问题 8.1 单源最短路径

给出带权图 G=(V, E, W)和一个源顶点 s。问题是找出 s 到每一个顶点 v 的最短路径。

在开始之前，我们必须考虑我们是否需要一个全新的算法。假设我们使用最小生成树的算法，从 s 开始。由算法构造的树中到 v 的路径是否就是 s 到 v 的最短路径？考虑图 8.4 中最小生成树中 A 到 C 的路径。它*不是*最短路径；路径 A,B,C 更短。

Dijkstra's 最短路径算法通过增加到 s 的距离来找 s 到其他顶点的最短路径。算法，类似于 8.2 小节的 Prim's MST 算法，从一个顶点（s）开始，通过选择到新顶点的特定的边来"向外分支"。由算法生成的树叫*最短路径树*。（叫这个名字不意味着它就是；必须证明在树中的路径确实是最短路径。）

同样类似于 Prim's MST 算法。Dijkstra's 算法是贪婪算法；它总是选择出现的"最近"的顶点的边；但是这里的"最近"是到 s 最近，而不是"到树最近"。顶点被分成 3（不相交）部分如下：

4. *树*的顶点：当前在树中的，
5. *边缘*的顶点：不在树中，但是与树中的顶点邻接，
6. *没有检查*的顶点：剩下的全部。

同 Prim's 算法一样，我们记录唯一的到边缘顶点的候选边（当前找到最好的）。对每一个边缘顶点 z，这里至少有一个树中的顶点 v，使得 vz 是 G 的一条边。对于每一个这样的 v 都有（唯一的）树中的从 s 到 v 的路径（可能 s=v）；d(s, v)表示路径的权。添加一条边 vz 到这条路径，，得到 s 到 z 的路径，它的权是 d(s,v)+W(vz)。z 的候选边是 vz，因为 d(s,v)+W(vz)是当前所有树中颗选择顶点最小的。

例 8.4 最短路径树的增长
看图 8.9(a)的图。每一个无向边都被看作是…….

Dijkstra's 算法的一般结构如下：

dijkstraSSSP(G,n)

初始化所有的顶点为*没有检查*。
从指定的源顶点 s 开始树；将其作为树的根；
定义 d(s,s)=0。
将所有邻接的顶点划成*边缘*。
当还有边缘顶点时：
　在树的顶点 t 和边缘顶点 v 中选择一条边，使得 d(s,t)+W(tv)最小；
　将 v 划到*树*顶点；将边 tv 添加到树；
　定义 d(s,v)=(d(s,t)+W(tv))。
　将所有邻接 v 的*没有检查*顶点划成*边缘*顶点。

因为 d(s,y)+W(yz)作为候选边可能重复使用，它可以被计算一次并保存。为了有效的计算它当 yz 第一次变成候选边时，我们也为树中的每一个 y 保存 d(s,y)。因此我们可以使用数组 dist：

dist[y]=d(s,y)　　树中的 y；
dist[z]=d(s,y)+W(yz)　　对于在边缘的 z，yz 是 z 的候选边。

就像在 Prim's 算法中，在一个顶点和相应的候选边被选择之后，在数据结构中的信息必须为某些边缘顶点和以前的没有检查顶点更新。

例 8.5 更新距离信息
在图 8.9(d)中……..

这个方法正确吗？关键的步骤是选择下一个边缘顶点和候选边。对于任意候选边 yz，d(s,y)+W(yz)并不必须是 s 到 z 的最短距离，因为到 z 的最短路径可能不穿过 y。（例如在图 8.9 中，到 H 的最短路径并不经过 G，尽管 GH 是 c,d 和 e 的可选的一部分。）我们认为，如果 d(s,y)是每一个树顶点 y 的最短距离，在所有候选边中选出最小的 d(s,y)+W(yz)从而选出了 yz，则 yz 就是给出的最短路径。这个观点在下面的定理中证明。

定理 8.6 令 G=(V, E, W)是一个带非负权的图。令 V`是 V 的一个子集，令 s 是 V`是的成员。假设 d(s,y)是 G 中 s 到每一个 y∈V'的最短的距离。如果为了使 d(s,y)+W(yz)最小选择边 yz，其中 y 在 V`中，z 在 V-V`中，则由 s 到 y 的最短的路径更上 yz 就是 s 到 z 的最短路径。

证明 看图 8.10。假设 e=yz 被选择，令 $s,x_1,\ldots,x_r,y$,是 s 到 y 的最短路径（可能 y=s）。令 $P=s,x_1,\ldots,x_r,y,z$。W(P)=d(s,y)+W(yz)。令 $s,z_1,\ldots,z_a,\ldots,z$ 是 s 到 z 的最短路径；叫 P`。顶点 $z_a$ 是 P`中不在 V`中的第一个顶点（可能 $z_a=z$）。我们必须说明 W(p)≤W(P')。（如果 a=1，$z_0=s$，算法将选择 $z_{a-1}z_a$。）因为选择了 e

$$W(P)=d(s,y)+W(e)\le d(s,z_{a-1})+W(z_{a-1}z_a). \qquad (8.2)$$

由于引理 8.5，$s,z_1,\ldots,z_{a-1}$ 是 s 到 z_{a-1} 的最短路径，所以这条路径的权是 $d(s,z_{a-1})$。既然 $s,z_1,\ldots,z_{a-1},z_a$ 是 P`的一部分，且每条边都是非负的，

$$d(s,z_{a-1})+W(z_{a-1}z_a)\le W(P') \qquad (8.3)$$

联合(8.2)（8.3），W(p)≤W(P')。

定理 8.3 给出一个带非负权的图 G 和源顶点 s，Dijkstra's 算法计算出 s 到 G 中每一个可到达顶点的最短距离（最小权的路径的权）。

证明 通过将顶点不断加到最短路径树的序列来证明。细节留到 8.16 中。

8.3.3 实现

分析

对于 8.2.7 小节对 Prim's 最小生成树算法，算法 8.1，的分析可以不做任何改变搬到对 Dijkstra's 最短路径算法，算法 8.2，上来。Dijkstra's 算法在最坏情况下也运行在$\Theta(n^2)$。同样的低限$\Omega(m)$，空间需求也是$\Theta(n)$。

如果不可到达顶点的数量很多，则预处理的时候测试可到达性会更有效率，消除不可到达顶点，对可到达的重新编号 $1,\ldots n_r$。总的消耗是$\Theta(m+n_r^2)$，而不是$\Theta(n^2)$。

8.4 Kruskal's 最小生成树算法

令 G=(V, E, W)是有向带权无向图。8.2 节中我们学习了找图 G 最小生成树的 Prim's 算法（G 必须是连通图）。算法从任意顶点开始，通过“贪婪的”选择最小权的边不断分支出树。任何时候加入一条边都是一颗树。这里我们检查另一种用贪婪策略的算法。本节中所有的图都是无向图。

8.4.1 算法

Kruskal's 算法的一般思想如下。每一步它都在图剩下的边中选一条权最小的，但是丢弃与已选择的边构成回路的。任意时刻选择的边都可能相距很远，使得形成森林，而不必须是树。当所有边都处理过之后算法终结。

```
kruskalMST(G, n)
{
    R=E;
    F=空集合;
    while(R 非空)
    {   移除 R 中权最小的边，比如 vw;
        if(vw 不在 F 中形成回路)
            添加 vw 到 F;
    }
    return F;
}
```

在考虑如何实现这个思想之前，我们必须问它是否能工作。既然图可以不是连通的，我们首先需要一个定义。

定义 8.4 生出树集合

令 G=(V, E, W)是带权无向图。G 的*生出树集合*是一个树的集合，每一棵树都是 G 的一个连同分量，集合中的树都是连同分量的生成树。*最小生成树集合*是生成树集合中，边权的和最小的生成树的集合，就是说是最小生成树的集合。

首先，G 的每一个顶点都在某棵树中吗（如果图是非连通的，就可能有多棵树）？令 v 是 G 的任一顶点。如果至少一条边连在 v 上，令 S 是附加在 v 上所有边的集合，则第一条从 S 中取走的边将加入 F。但是如果 v 是孤立顶点（没有边与之相连），则它将不出现在 F 中，而且没有忽略它的话，需要单独考虑。

下一个问题是算法生成的是否是生成树集合，假设 G 没有孤立顶点。就是说，对 G 的每一个连通分量 F 中是否都有一棵精确的树？下面的引理提供了这个问题的一些理解。证明是简单的，留在练习中。

> **引理 8.8** 令 F 是森林；就是任意无向无环图。令 e=vw 是不在 F 中的边。
> 如果 e 和 F 中的边构成回路，当且仅当 v 和 w 属于 F 的同一个连通分支。

现在假设 G 的有些连通分量对应 Kruskal's 算法得到的森林 F 中的两棵或两棵以上的树 。G 中必然存在边连接这两颗树，称为 vw；v 和 w 属于 F 不同的连通分支。因此，当算法处理边 vw 时，它必然会得到 F'，F'在森林中有回路存在。所以 vw 不会添加到 F'。由于引理 8.8，v 和 w

8.4.2 分析

第九章 传递闭包、任意两点的最短路径

9.1 概述

本章学习两个相关的问题，这两个问题可以非正式的描述为下面两个关于一个图的所有*顶点对*的问题的答案：

1. u 到 v 有路径吗？
2. u 到 v 的*最短*路径？

在第七章和第八章我们见过了这两个问题的特殊情况，即第一个顶点是指定的，仅第二个顶点可以是图中任意顶点。本章中，我们学习更全面的问题。

本章表现的主要算法思想有非常广泛的应用。Klneene（为了综合正则语言，本书没有涉及），Warshall（为了传递闭包），Floyd（为了所有对的最短路径）各自独立的从不同的应用发现本章的算法思想。通常，被称为 Kleene-Floyd-Warshall 算法。整个这类问题都可以称为 *semi-ring closure* 问题。这超出了本书的范围。阅读本章后面的注释和引用文献得到进一步信息

9.2 二元关系的传递闭包

本节中，我们定义术语二元关系的传递闭包，并将他们的关系看作有向图的路径。我们也引入一些本章使用的符号。首先检查一种非常直接的算传递闭包的方法，之后会有复杂的方法。

9.2.1 定义和背景

令 S 是元素 s_1、s_2、...的集合。复习 1.3.1 小节，S 上的*二元关系*是一个 S×S 子集，称为 A。如果 $(s_i, s_j) \in A$，我们称 s_i 与 s_j 有关系 A，使用符号 $s_i A s_j$ 表示。

假设 S 有 n 个元素。关系 A 可以表示 n×n 布尔矩阵

$$a_{ij} = \begin{cases} true & \text{如果} s_i A s_j \\ false & \text{其他} \end{cases}$$

开始我们用这种表示方式，后来我们也考虑使用 1，0 代替 *true* 和 *false*；图表中也显示为 1，0。对于布尔矩阵，术语 *0 矩阵*，表示为 0，意味着矩阵所有的元素都是 *false*；*单位矩阵*，表示为 I，意味着矩阵对角线的元素是 true，其他的元素都是 false。

图中顶点集合之间的邻接关系

9.3 传递闭包的 Warshall's 算法

9.4 图的任意两点的最短路径

在第 8 章中，我们学习了 Dijkstra's 算法（算法 8.2），其在带权图中找到了特定*源顶点*到所有其他顶点的一条最短路径和距离。算法使用邻接表结构，在最坏情况下运行在 $\Theta(n^2)$ 时间。现在我们考虑下面的问题：

问题 9.1 任意两顶点之间的最短路径

给出带权图 G=(V, E, W)，有顶点 V={v_1,…v_n}，图用如下权矩阵表示

$$w_{i,j} = \begin{cases} W(v_i v_j) & v_i v_j \in E \\ \infty & v_i v_j \notin E \wedge i \neq j \\ \min(0, W(v_i v_j)) & v_i v_j \in E \\ 0 & v_i v_j \notin E \end{cases} \qquad (9.2)$$

计算 n×n 矩阵 D，其中 d_{ij} 表示 v_i 到 v_j 的最短距离。（*距离*就是最小权路径的权）。

图 9.3 是个例子。问题可以扩展成需要最短路径的*路由表*。如果存在负权的环，则有些顶点之间的距离不会是最短的路径——在环上绕任意圈得到的路径会更小。

计算 D 的（如果 G 没有负权）一个想法是对每一个顶点使用算法 8.2。但是，我们可以使用扩展的 Warshall's 算法，就是 R.W.Floyed。这个算法更精简，它消除了算法 8.2 的数据结构。

我们如何计算 D[i][j]？最短路径可能以任意顺序经过任意其他顶点。就像在 Warshall's 算法中的，我们以路径中间的数字最高的顶点来分类路径（参看定义 9.2）。

回顾引理 8.5 中的最短路径属性：如果 v_i 到 v_j 的最短路径经过中间顶点 v_k，则路径中 v_i 到 v_k 中的小段和 v_k 到 v_j 中的小段分别都是最短路径。如果我们选择 k 作为 v_i 到 v_j 路径中最高索引的中间顶点（假设路径包含一条以上的边），则出现的每一个小段中最高数字的中间顶点的索引必然都小于 k。（参看图 9.2，其展示了 Warshall's 算法一样的思想。）这提示在循环中计算矩阵 D，以下面的递归等式。

$$D^{(0)}[i][j] = w_{ij}$$

$$D^{(k)}[i][j] = \min(D^{(k-1)}[i][j], D^{(k-1)}[i][j] + D^{(k-1)}[k][j]) \qquad (9.3)$$

这里 w_{ij} 在等式（9.2）中定义。通过前面对最短路径属性的观察，同样的观点用在了引理 9.1 中，下面的引理可以证明（证明留在练习中）。

引理 9.3 对于每一个 0,…,n 中的 k，令 $d_{ij}^{(k)}$ 是 v_i 到 v_j 经过中间顶点 v_k 的简单路径的最小权，令 $D^{(k)}[i][j]$ 由等式（9.3）定义。则 $D^{(k)}[i][j] \le d_{ij}^{(k)}$。

例 9.3 计算距离矩阵

……..

等式（9.3）计算了一个矩阵序列：$D^{(0)}, D^{(1)}, \ldots, D^{(n)}$。既然计算 $D^{(k)}$仅用到了 $D^{(k-1)}$，我们没必要存储前面的矩阵，只需要两个 n×n 矩阵就可以了。事实上，仅要一个；计算可以在矩阵 D 内进行。既然矩阵的元素仅能不断变小，如果假设 $D^{(k-1)}[i][j]$被使用了，但是 $D^{(k)}[i][j]$ 可以代替它的位置，我们令 $D^{(k)}[i][j] \le D^{(k-1)}[i][j] \le d_{ik}^{(k-1)}$，计算过程会找到更好的路径的。

算法 9.4 （Floyed）任意两点间最短路径

9.5 通过矩阵操作计算传递闭包

第十章 动态规划

10.1 概述

Those who cannot remember the past are condemned to repeat it.

—George Santayana, The Life of Reason; or, The Phases of Human Progress(1905)

在计算机科学中动态规划已经演化成一种主要的算法设计范例。但是它的名字对于许多

人有点神秘。这个名字是 1957 年由 Richard Bellman 在描述一种最优控制问题时构造的。实际上，名字来源与描述的问题而不是解决问题的技术。这里 programming 表示“一系列选择”，比如规划一个无线电站。词动态的（dynamic）表示一种思想，即选择依赖于当前的状态，而不是在之前就决定好了。所以在原来意思里面，a radio show in which listeners call in request might might be said to be “dynamically programmed” to contrast it with the more usual format where the selections are decided before the show begins. Bellman 描述了一种解决“dynamically programming”问题的方法，这种方法成了后来好多计算机算法的灵感。这个方法的主要特点是将指数时间的问题转换成多项式时间。这称为了动态规发算法的主要特点。

本章不同与其他章，其他章中我们经常关注某个问题或应用领域并各方法来解决之；但是在本章中，我们关注一种技术，为不同应用领域开发动态规划解决方案的技术。

自顶向下的算设计时自然的且强大的。我们首先在通用方式上考虑和计划，之后在增加细节。我们通过将顶层复杂的问题分解成子问题来解决之。使用递归，我们通过将大型问题分解成同样的小规模问题来解决。分而治之，这种递归算法设计技术，被证明在得到快速的排序算法时是很有用的。但是象好的递归一样，如果不正确的控制，它仍然将变的非常低效。Fibonacci 数列提供了一个简单而深刻的例子。

例 10.1 递归的 Fibonacci 函数

回忆等式 1.13，Fibonacci 数列以递归等式定义 $F_n=F_{n-1}+F_{n-2}$ 对于 $n\geq 2$，边界值是 $F_0=0$，$F_1=1$。他们是递归定义的，所以自然而然的计算他们就使用例 3.1 给出的递归函数 fib(n)。但是，如图 7.13 所示，自然的递归计算有可怕的低效率，因为进行了大量的重复工作。图 7.13 实质上展示了 fib(6)的活动记录树。一般的，fib(n)的活动记录树是一颗深度为 n/2（最右边的路径最短）的完全二叉树，而且在更低的深度有更多的节点，所以运行时间至少是$\Omega(2^{n/2})$。练习 10.1 会给出一个精确的渐进阶。但是 F_n 可以由$\Theta(n)$简单语句的算法计算出，只需要计算并记录比 n 小的值就可以了，如果小的值都是已知的话计算 fib(n)只需要常量时间。（复习 3.2.1 小节，*简单语句*就是不调用函数，需要常量时间执行的语句。）假设有一个大数组 f，下面的代码：

```
f[0]=0; f[1]=1
for(i=2; i<=n; i++)
    f[u]=f[i-1]+f[i-2];
```

练习 10.2dispenses with the 数组。

一个动态规划算法为子问题存储结果或是解决方案，之后在需要他们来解决大型问题的时候，就可以直接查找他们而不是重新计算。因此动态规划特别适合那些递归算法会重复解决很多子问题的问题。

We will introduce a characterization of dynamic programming algorithms that provides a unified framework for a wide variety of published algorithms that might seem quite different on the surface. 这套 framework 允许将一个递归解决方案转换成一个动态规划算法，提供了分析动态规划算法复杂度的方法。

10.2 子问题图和他们的的遍历

就像早先说明的，问题经常通过分解成同类型小的问题来解决之，递归的解决了小问题再合并结果。假设我们有这样一个解决方案。我们可以基于问题和他们相关的子问题定义一个有向图。

定义 10.1 子问题图

假设对于问题有递归算法 A。*A 的子问题图*是有向图，图的顶点是问题的实例或输入，图的有向边 I->J 表示算法 A 在解决实例 I 时有一个直接的递归调用实例 J。（这里我们使用“I->J”而不是“IJ”是为了强调边是有向的。）和我们迄今为止考虑的其他图不同，以前的图都可以显示的表示为一个数据结构，子问题图是抽象的没有显示的表示。

令 P 是算法 A 的一个问题实例；就是说我们假设 A(P)不是一个递归调用。则 A(P)的子问题图是 A 的子问题图的可以从顶点 P 出发到达的一部分。

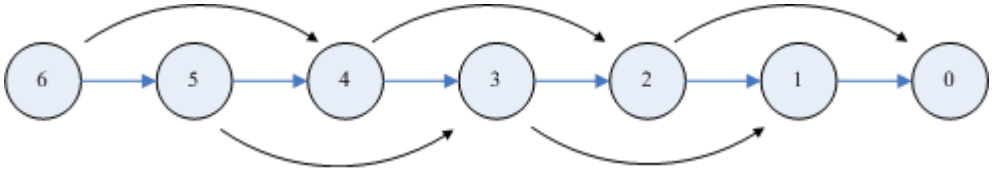


图 10.1 fib(6)的子问题图

例 10.2 Fibonacci 函数的子问题图

对于递归的 Fibonacci 函数，fib(n)，问题实例是非负整数，所以 F 的子问题图的顶点也是自然数。有向边是 $\{i \to i-1 \mid i \geq 2\} \cup \{i \to i-2 \mid i \geq 2\}$。尽管图是无限的，但是对于任意计算 fib(n)对应的子图（例如，可以从顶点 n 出发到达的部分）仅有 n+1 个顶点和 2n 条边。图 10.1 展示了这个例子。

如果算法 A 总是可终结的，则它的子问题图必然是无循环的。有向无环图（DAGs）在 7.4.6 小节中学习过，我们将有一个用到前面学习内容的机会。通过查找一个顶层递归调用（称为 **A**(*p*)）产生的活动帧树，显然树中的每一条路径就对应于一条 **A**(*p*)的子问题图中的一条边，这条边以顶点 P 为起点，以基本情况为结束点。基本情况节点没有出的边。在这个抽象图中顶点就是问题实例。有向边代表执行 **A**(*p*)的过程中进行的递归调用。

考虑

定义 10.2 递归算法的动态规划版本

给定递归算法 **A**，**A** 的*动态规划版本*记为 ***DP*(A)**，是一个

10.3 矩阵的乘法顺序问题

本节中我们将解决矩阵乘法的顺序问题，这是一个动态规划的经典例子。在下一节中我

们将学习一个起源与这个问题更本不相同的另一个问题，但是有一个非常类似的解决方法。两者和在一起是一个动态规划的好的入门。

本节的目的不是展示如何解决矩阵乘法的顺序问题，而是展示如何一步一步的使用规则来开发动态规划算法。我们希望这里的规则能帮助读者解决新的问题，能让读者有一种思想（什么时候该使用动态程序的策略）。但是因为我们要解释矩阵乘法顺序问题的解决方案，所以这个问题的“待遇”会比较好。

10.3.1 矩阵乘法顺序问题

假设我们想求在一串矩阵的连乘式子中，最好的乘法顺序。我们使用普通的矩阵乘法算法（算法 1.2），每一次我们乘上两个矩阵。一个 p×q 的矩阵乘以 q× r 的矩阵，需要作 pqr 次元素的乘法。这里可以有很重要的两点。第一，矩阵的乘法有结合率，即 A(BC)=(AB)C。第二，结合顺序的不同，工作量会有 很大的差别。考虑下面的例子。

例 10.4 不同的矩阵乘法顺序

我们要乘的矩阵的大小如下：

A_1 × A_2 × A_3 × A_4

30×1 1×40 40×10 10×25

下面的计算展示了不同的乘法顺序的不同计算量。

$((A_1A_2)A_3)A_4$ 30×1×40＋30×40×10＋30×10×25＝ 20700

$A_1(A_2(A_3A_4))$ 40×10×25＋1×40×25＋30×1×25 ＝ 11750

$(A_1A_2)(A_3A_4)$ 30×1×40＋40×10×25＋30×40×25 ＝ 41200

$A_1((A_2A_3)A_4)$ 1×40×10＋1×10×25＋30×1×25 ＝ 1400

对于一般的问题，假设我们给出矩阵 A_1，A_2，...，A_n，这里 A_i 的维数是 $d_{i-1}\times d_i$（$1\le i\le n$）。我们必须作多少次乘法，最小的消耗是多少？我们的消耗是元素乘法的次数。（当然还 有一些函数调用的开销。）现在我们将注意力放在找到最小消耗上；之后我们使算法“回忆“起最小是怎么达到的。我们将 A_k 和 A_{k+1} 之间的乘法表示为第 k 次 乘法。

10.3.2 贪婪的尝试

任何一种 n-1 次乘法的序列都是合法的，算法需要决定那一个的消耗是最小的。贪婪的

算法看上去是可行的。首先选择消耗最小的那次乘法，在这次乘法之后求出 矩阵的维数变化，继续选择最小消耗的乘法，重复之。例 10.4 就有这种策略。然而，在某些 3 个矩阵的序列时（仅有两次乘法），这种策略会失败。另一种贪婪的策略在练习 10.6 中。典型的，动态程序算法比贪婪算法要更 耗费时间，所以仅在无法找到贪婪策略来求最优时菜使用动态程序算法。

10.3.3 动态规划的解决方案

我们的下一个尝试是开发出一种递归算法。假设，在选择第一个乘法之后（序列中的位置 i），我们递归解决剩下的最优化问题。我们对每一个 i 都当作有效的第一个选择尝试一次，最后选择有最小组合消耗的 i。这叫做*回溯*算法，因为在尝试每一个完整的选择序列之后，算法回溯到最近的选择点之前，在尝试另一种选择；当这一点的选择都尝试过之后，算法回溯到更早的选择点再尝试新的选择，继续直到所有的选择都尝试过了。我们在 8 皇后问题见过这种方法（参看图 7.14）

假设维数 d0,..., dn 在数组 dim 中。我们保持数组不变，仅通过一个整数序列来标识子问题，整数序列表示剩下矩阵维数的索引。一开始索引序列是 0,...n。注意所有的序列的索引除了第一个和最后一个，都对应乘法运算符。

在第一个选择 i 之后，剩下问题的索引序列是 0,...,i-1,i+1...,n。就是说，选择的第一个乘法是 $A_i \times A_{i+1}$，其维数是 d_{i-1}，d_i 和 d_{i+1}。令 $B=A_i \times A_{i+1}$；则 B 的维数是 $d_{i-1} \times d_{i+1}$。剩下的子问题是乘

$$A_1 \times \cdots \times A_{i-1} \qquad \times \quad B \quad \times \qquad A_{i+2} \times \cdots \times A_n$$

$$d_0 \times d_1 \quad d_{i-2} \times d_{i-1} \qquad d_{i-1} \times d_{i+1} \qquad d_{i+1} \times d_{i+2} \qquad d_{n-1} \times d_n$$

假设索引序列本身存储在 0 开始的数组 seq 中，len 是 seq 的长度。方法的概要是

```
mmTry1(dim ,len, seq)
{
    if(len<3)
        bestCost=0;   // 基本情况一个数组或是没有
    else
    {   bestCost=无穷
        for(i=1; i<=len-1; i++)
        {
            c=seq[i]的位置做乘法的消耗；
            newSeq= seq 除去第 i 个元素；
            b= mmTry1(dim, len-1, newSeq);
            bestCost=min(bestCost, b+c);
        }
    }
    return bestCost;
}
```

这个算法的递归等式是 $T(n) = (n-1)T(n-1) + n$。这个解决方案在 $\Theta((n-1)!)$，但是我们的目标是通过将递归算法转换成动态规划算法来提升性能。

为了设计一个动态规划的版本，我们首先需要分析子问题图。从初始的问题可以到达多少子问题，就是索引序列 0,…,n 可以描述多少问题？这里我们遭遇了一个系列困难。尽管子序列是以更短的子区间开始的，但是随着子问题深度的不断加深，他们会变成越来越多的片段。例如当 n=10 的时候，选择了子序列 1,4,6,9，剩下的索引变成了 0,2,3,5, 7, 9,10。没有简单的方式来指定子序列。本质上来说，初时序列的每一个子序列（至少带 3 个元素的）都是一个可到达的子问题。这里有 2^n 个这样的子序列（参看练习 10.3），因此有指数量级的子问题。这个图大到无法有效的查找了。

这个展示了设计动态规划算法最重要的法则。子问题必须有*简单标识*。这限制了子问题图的最大规模（顶点的数量——还是可能有许多的边），并且可以为所有的标识编制字典（在需要解决的范围内（译注：即给定问题规模其子问题的规模是可以控制的））。回顾 *标识*，或 id 的含义，一个元素的标识在字典中唯一的表示该元素（2.5.3 小节）。字典中不同的*标识*对应不同的元素。因此，只要我们将字典的大小限制在输入的多项式时间之内，或者更小，那么我们就可以保证在深度优先搜索子问题图时在多项式时间之内。

基于这些考虑，我们意识到我们需要不同的思路来将问题压缩成子问题。在第一个矩阵乘法之后查找创建的子问题没有工作。通过选择最后一个矩阵乘法创建的子问题怎么样？假设最后一个矩阵乘法的位置是 i？这实际上创建了两个子问题：

1. 乘以 $A_1,\ldots, A_i$ 其维数索引是 0,…,i：

$$A_1(d_0 \times d_1) \times A_2(d_1 \times d_2) \times \cdots \times A_i(d_{i-1} \times d_i) = B_1(d_0 \times d_i)$$

2. 乘以 $A_{i+1},\ldots, A_n$ 其维数索引是 i,…,n：

$$A_{i+1}(d_i \times d_{i+1}) \times A_{i+2}(d_{i+1} \times d_{i+2}) \times \cdots \times A_n(d_{n-1} \times d_n) = B_2(d_i \times d_n)$$

最后一步乘以 B_1B_2 的消耗是 (d_0, d_i, d_n)。

并不能立即观察出这比我们一开始的方式好在那里。但是我们注意到每一个子问题都可以标识成一个整数对（目前为止）（0，i）和（i，n）。就是说，第一个子问题的索引序列是(0,1...,i)，但是既然元素是连续的，所以仅需要列出终点。（一个对（j-1,j）表示 Aj 是单独的，消耗是 0）就像前面说的，维数数组 $d_0,...,d_n$ 不用改变，可以作为一个全局的数组。

在进一步检查之前，我们看到一旦在子问题上选择了最后一个乘法的索引，创建出来的新子问题也可以用一个整数对来描述。例如，如果在子问题(i,n)中选择了 k，新的子问题是(i,k)和(k,n)。因此我们看到这种问题分解方法仅在子问题图中创建了 $\Theta(n^2)$ 个不同子问题。

就像前面一样，我们不知道最后选择那一个乘法会有多少开销，所以我们需要枚举所有的选择。mmTry2(dim,low, high)的目标就是找到（low,high)指定的子问题的最优消耗，这里low<high。其概要类似与 mmTry1：

```
mmTry2(dim, low, high)
{
1.  if(high-low==1)
2.        bestCost=0; // 基本情况：仅有一个矩阵
3.  else
```

```
4.      bestCost=无穷;
5.  for(k=low+1; k<=high-1; k++)
    {
6.      a=mmTry2(dim, low, k);
7.      b=mmTry2(dim, k, high);
8.      c=位置 k 处乘法的消耗，dim[low], dim[k], dim[high]
9.      bestCost=min(bestCost, a+b+c);
    }
10. return bestCost;
}
```

与 mmTry1 类似，这也是一个回溯算法。这个算法的精确递归等式是复杂的，但是我们可以得到一个简化的版本显示其大于 2^n(参看练习 10.4)。我们预计到这一点，因为，回溯算法典型的都是指数级别的，但是我们希望可以通过把自然的递归算法转换成动态规划来改进性能。

再一次的，让我们考虑子问题图，这里初始的问题用对(0,n)描述。顶点（子问题图的）通过整数对(i,j)标识，i<j 且 i,j 都在 0,…,n 的范围内，所以有约 $n^2/2$ 个。对于对(i,j)标识的子问题……

在下一个过程里，字典叫 cost，其元素的标识是一对整数。为了使用通用 ADT，我们必须将标识包装进一个组织者类，但我们可以定制的字典的接口，使之接受两个整数，low 和 high 作为两个参数。我们依然使用字典的操作 create, member, retrieve 和 store。行号与 mmTry2 区别。

```
mmTry2DP(dim, low, high, cost)
{
1.  if(high-low==1)
2.      bestCost=0; // 基本情况：仅有一个矩阵
3.  else
4.      bestCost=无穷;
5.  for(k=low+1; k<=high-1; k++)
    {
6a.     if(member(low, k)==false)
6b.         a=mmTry2DP(dim, low, k, cost);
6c.     else
6d.         a=retrieve(cost,low, k);

7a.     if(member(k, high)==false)
7b.         b=mmTry2DP(dim, k, high, cost);
7c.     else
7d.         b=retrieve(cost, k, high);

8       c=位置 k 处乘法的消耗，dim[low], dim[k], dim[high]
9       bestCost=min(bestCost, a+b+c);
    }
10a. store(cost, low, high,bestCost);
```

10b. **return** bestCost;

}

既然子问题通过范围 0,…,n 之内的整数对标识，那么字典可以用一个(n+1)×(n+1)矩阵来实现。在 mmTry2DP 中我们存储和检索子问题的最优解。在后面完整的算法里，cost 数组通过 last 数组来支持，last 数组包含了子问题的*最优*选择乘法。无穷的条目标识子问题还没有解决。我们将取消字典，直接访问数组。

我们已经看到 mmTry2 可以通过一些机械的步骤就转换成 mmTry2DP，其实现了 memoization。结果看上去很像 DFS 的框架：第 6 和第 7 行在一个循环中测试没有解决的子问题（没有发现的，或是白顶点），这个循环遍历了所有需要的问题（所有到邻接顶点的边）。已解决的子问题（黑顶点）仅仅查找一下。在第 10 行（postorder time）当前子问题变成已解决了（当前顶点标记为黑）。

o o o o o o o o o

从前一段提到的边，我们发现，减少第二个索引保持第一个索引将导致更低的拓扑数。

10.4 构造最优二叉树

在本节中我们考虑这样一个问题，如果我们知道对有些关键字的查找次数多于其他的，怎样在二叉查找树中排列关键字（来源于线型集合）使得平均查找次数最小。 在二叉查找树节点上的关键字符合定义 6.3 中给出的二叉查找树属性。回顾一下，以中序遍历二叉查找树得到的是关键字的升序序列。看图 10.3 的例子。读者可能希望在学习之前复习一下二叉查找树的遍历算法（算法 6.1）。

定义 10.3

我们采用下面的记号：

1. 定义 A(low, high, r)为子问题(low, high)是 K_r 被选为二叉查找树的根时的最小权消耗。
2. 定义 A(low, high)是子问题(low, high)对于所有根的最小权消耗。
3. 定义 p(low,high)=p_{low}+…+p_{high}；就是说，概率 that the key being search for is *some* key 在范围 K_{low} 到 K_{high} 中所有。我们称这个为*子问题的权*(low, high)。

如果对于包含 K_{low},…, K_{high} 的特定树，其检索的消耗权是 W（假定它是一颗完整的树，所以它的根的深度是 0），则如果它作为根的深度是 1 的子树时，则检索的消耗权是(W+p(low, high))。就是说，每一次搜索穿过整个子树都比原来是完整树时要多一次比较，所以多出的部分是 p(low, high)。（参看插图 10.4）这个关系允许我们递归的合并子问题的解决方案得到更大问题的解决方案。

$$A(low, high, r) = p_r + p(low, r-1) + A(low, r-1) + p(r+!, high) + A(r+1, high)$$

$$= p(low, high) + A(low, r-1) + A(r+1, high), \qquad (10.2)$$

$$A(low, high) = \min\{A(low, high, r) \mid low \le r \le high\} \qquad (10.3)$$

我们可以写一个递归过程计算 A(low, high)，基于等式(10.2)和(10.3)。但是，就像我们在 10。3 节学习的矩阵乘法顺序问题，我们将在递归解决方案观察到许多重复的工作。算法运行时间将是指数时间。再一次，为了避免重复工作，我们定义一个字典，用大小(n+2)×(n+1)的两个二维数组，叫做 cost 和 root。

就像在矩阵乘法顺序问题，子问题

10.5

第十一章 串匹配

11.1 概述

在这章里面我们学习检查一个特定子串（称为 pattern）是否出现在另一个字符串（称为 text）的问题。这个问题经常出现在

11.2 直接的解决方案

11.3 Knuth-Morris-Pratt 算法

11.4 Boyer-Moore 算法

11.5 近似串匹配

第十二章 多项式和矩阵

12.1 概述

这一章要讨论的问题是多项式估值（带和不带预处理系数），多项式乘法（作为一个离散傅立叶变换的例子）和矩阵向量乘法。这些任务所用到的操作一般是乘和 加。在旧计算机上，乘法比加法要做很多工作，有些算法是通过增加一些额外的加法来减少乘法的数量以此“提高”速度。因此他们的值依赖于乘法和加法的相对代 价。其他的算法用的是减少这两种操作的数量（对于超大型的输入）。

本章中的许多算法使用分而治之策略：带预处理系数的估值多项式（12.2.3 小节），Strassen's 矩阵乘算法（12.3.4 小节），和快速傅立叶变换（12.4 节）。

本章中许多结果的底限都是直接给出，没有给出证明。参考本章后面的注释和引用文献得到这些结果的更进一步的信息。

12.2 计算多项式函数的值

考虑多项式 $p(x)=a_nx^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$，系数是实数，n>=1。假设，系数 $a_0,a_1,\ldots a_n$ 和 x 给出，问题是计算 p(x)的值。

12.2.1 算法

12.2.2

12.3 矢量和矩阵乘法

12.4 快速傅立叶变换和卷积

傅立叶变换在工程、物理科学和数学中广泛使用。它的离散版本应用在内插值问题，求偏微分方程的解，电路设计，检晶仪中，在信号处理中应用非常广泛。离散傅立叶变换是最早的使用分而治之策略开发的算法比直接求解的渐进阶要低的问题。改进的算法叫快速傅立叶变换。

这个算法有深远的影响，因为许多其他的数学计算能表示为某种形式的傅立叶变换。有些情况下将原来的问题转换成傅立叶变换问题来解决，比直接解决原来的问题要快。卷积就是这样一个例子

定义 12.1 卷积

另 U 和 V 是 n 维向量，索引是 0 到 n-1。U 和 V 的*卷积*，表示为 U*V，定义为 n 维变量 W，其中 $W_i = \sum_{j=0}^{n-1} u_j v_{i-j}$，其中 $0 \le i \le n-1$ 且右边的索引以 n 为模。

例如，对于 n=5，

$$w_0=u_0v_0+u_1v_4+u_2v_3+u_3v_2+u_4v_1$$
$$w_2=u_0v_1+u_1v_0+u_2v_4+u_3v_3+u_4v_2$$
$$\ldots$$
$$w_4=u_0v_4+u_1v_3+u_2v_2+u_3v_1+u_4v_0$$

计算两个向量卷积的问题经常在概率问题，工程，和其他领域提出。本章讨论的符号多项式乘法是一个卷积运算。

在 n 维变量的离散傅立叶变换和两个 n 维变量的卷积中都可以用直接的方式求，需要 n^2 次乘法和略少于 n^2 次加法。我们将展示计算离散傅立叶变换的分而治之算法，只用 $\Theta(n\log n)$算术运算。这个算法（它有各种变体）就是快速傅立叶变换 FFT。后面我们将使用

FFT 在Θ(nlogn)时间内计算卷积。这个时间节省是很可观的。

贯穿本节所有的矩阵、数组和向量都是以 0 作为索引的开始。复数的单位根（roots to unity）和他们的一些基本属性将在 FFT 中使用。基本定义和需要的属性在本节的附录中复习了（12.4.3 小节）。对复数或是第 n 个单位根不是很熟悉的读者应该先读一下附录。

12.4.1 快速傅立叶变换

离散傅立叶变换将一个 n 维复向量（也就是元素都是复数的 n 维向量）变换成另一个 n 维复向量。变换 n 维实向量，可以认为是虚部都是 0 的 n 维复向量。

定义 12.2 快速傅立叶变换和矩阵

对于 n≥1，令ω be a primitive nth root of 1（译注：即$\omega^k \neq 1$，其中 k=1,2,3,4,…n-1），令 F_n 是 n×n 矩阵其元素是 $f_{ij}=\omega_{ij}$，这里 $0 \le i, j \le n-1$。n 维向量 $P=(p_0,p_1,\ldots p_{n-1})$的离散傅立叶变换是乘积 F_nP。

F_nP 的分量是

$$\omega^0 p_0 + \omega^0 p_1 + \cdots + \omega^0 p_{n-2} + \omega^0 p_{n-1}$$

$$\omega^0 p_0 + \omega^1 p_1 + \cdots + \omega^{n-2} p_{n-2} + \omega^{n-1} p_{n-1}$$

……

$$\omega^0 p_0 + \omega^i p_1 + \cdots + \omega^{i(n-2)} p_{n-2} + \omega^{i(n-1)} p_{n-1}$$

……

$$\omega^0 p_0 + \omega^{n-1} p_1 + \cdots + \omega^{(n-1)(n-2)} p_{n-2} + \omega^{(n-1)(n-1)} p_{n-1}$$

稍微改变一下形式，第 i 个分量是：

$$p_{n-1}(\omega^i)^{n-1} + p_{n-2}(\omega^i)^{n-2} + \cdots + p_1\omega^i + p_0$$

因而如果我们将 P 的分量解释为多项式 $p(x)=p_{n-1}x^{n-1} + p_{n-2}x^{n-2} + \ldots + p_1x + p_0$ 的系数，则第 i 个分量是 $p(\omega^i)$，而计算 P 的离散傅立叶变换等效计算多项式 p(x)在$\omega^0, \omega^1, \omega^2,\ldots, \omega^{n-1}$ 的值，也就是每一个 1 的第 n 重根。我们将以这种观点讨论这个问题。我们将首先开发一种分而治之的的递归算法再仔细优化它去掉递归。我们将假定 n 是 2 的整数次幂。（对于不是的情况可以调整算法。）

分而治之策略是将一个问题分解成小的实例，解决之，用小实例的解组合成原来实例。这里，为了计算 p 的 n 个点，我们计算两个小的原来的集合子集的多项式再适当的合并结果。复习$\omega^{n/2}=-1$ 以及 $0 \le j \le (n/2)-1$，$\omega^{(n/2)+j}=-\omega^j$。将 p(x)按奇数指数和偶数指数分为两组：

$$p(x) = \sum_{i=0}^{n-1} p_i x^i = \sum_{i=0}^{n/2-1} p_{2i} x^{2i} + x \sum_{i=0}^{n/2-1} p_{2i+1} x^{2i}$$

定义

$$p_{even}(x) = \sum_{i=0}^{n/2-1} p_{2i} x^i \text{ 和 } p_{odd(x)} = \sum_{i=0}^{n/2-1} p_{2i+1} x^i$$

则

$$p(x) = p_{even}(x^2) + xp_{odd}(x^2) \text{ 和 } p(x) = p_{even}(x^2) - xp_{odd}(x^2) \qquad (12.4)$$

等式（12.4）展示为了计算 $1, \omega, \ldots, \omega^{(n/2)-1}, -1, -\omega, \ldots, -\omega^{(n/2)-1}$ 的 p，算法需要计算 $1, \omega^2, \ldots, (\omega^{(n/2)-1})^2$ 的 p_{even} 和 p_{old}，然后做 n/2 次乘法（计算 $xp_{odd}(x^2)$）和 n 次加减法。多项式 p_{even} 和 p_{odd} 可以以同样的策略递归解决。就是说，他们是度为 n/2-1 的多项式，可以在 n/2 重根计算：$1, \omega^2, \ldots, (\omega^{(n/2)-1})^2$。显然当要计算的多项式是一个常量的时候，不需要做工作。

递归算法如下。

算法 12.4 快速傅立叶变换（递归版本）

算法 12.5 快速傅立叶变换

输入： 向量 $P=p_0, p_1, \ldots, p_{n-1}$ 是 n 个元素的 **Complex** 数组，$n=2^k$；整数 k>0。（为了处理 float 数组，可以将其拷贝到 **Complex** 数组 P 的实部，将虚部全部设为 0。）

输出： n 个元素的 **Complex** 数组 transform，P 的离散傅里叶变换。数组作为输入参数，算法填充之。

注意：我们假设类 **Complex** 提供了复数的算术运算以简化我们的伪代码。这个类在 Java 中是不存在的。

我们假设 omega 是一个全局的 Complex 数组，其中包含 1 的 n 重根：$\omega^0, \omega, \ldots, \omega^{n-1}$。Complex 数组 transform 初始化为包含第 k-1 层的值，不是叶子，参看图 12.5。π_k 是 {0, 1, …, n-1} 的排列。

12.4.2 卷积

12.4.3 附录：复数和单位根 Roots of Unity

复数的域 **C** 是结合 i（虚数单位，-1 的平方根）和实数 **R** 的域得到得的。**C**=**R**(i)={a+bi| a,b∈**R**}。如果 z=a+bi，a 叫做 z 的实部，b 叫做 z 虚部。另在在 $z_1=a_1+b_1i$，$z_2=a_2+b_2i$。则定义，

$$z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$$

$$z_1 z_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i$$

$$\frac{1}{z_1} = \frac{a_1}{a_1^2 + b_1^2} - \frac{b_1 i}{a_1^2 + b_1^2} \qquad z_1 \neq 0 + 0i$$

除法和减法可以用上面的等式简单的推导得到。

> *Java 语言提示：*为了在算法的伪代码中简化复数的表示和运算，我们假定 Complex 类型存在，而且可以做算术运算*/+和-；这样的类在 Java 中不存在，但是在 C++和 Fortran 中支持这种表示法。可以简单的定义类有两个实例域，re 和 im，float 或 double 类型。（数学书中常用缩写 re 和 im 来表示复数的 re 和 im 部分。）但是，程序员必须将算术运算符定义成函数（静态方法），在实际代码中使用函数调用而不是算术表达式。

一个复数可以表示复平面上的向量。

第十三章 $\mathcal{NP}$完全问题

13.1 概述

在前面的章节中我们学写了非常多种类的问题和算法。有些算法是很直接的，但是有些很复杂和需要技巧，但是所有这些算法的复杂性都在$O(n^3)$，这里 n 是问题规模定义。从本章的观点来看，我们将接受所有的算法 studied so far as having fairly low time requirements。再看一眼表 1.1。我们看到有些算法很复杂以致于不能用简单的多项式描述，这些算法求解一个大规模的输入运行的时间相当大。表的最后一列展示了如果复杂性是 2^n，算法即使对于很小的输入也是没有用的。在这一章里面我们将关注那些指数复杂度的问题，这些问题对于稍微大一点的输入就需要计算很多年，甚至好几个世纪。我们将展示一个定义，这个定义区分我们已经碰到的 *tractable*（“不是很难” 译注：可控制的，可管理的）问题和 *intractable*（“很难“，或是非常耗时的）问题。我们将学习一类重要的问题，这类问题有一个烦人的属性——至今我们还无法知道他们能不能有效的解决。已经找到解决这类问题的非精确算，但是没有一个能证明这些问题需要很多时间。因为许多这类问题是实际应用中频繁提出的优化问题，有效的算法有重大的现实意义。

13.2 $\mathcal{P}$和 $\mathcal{NP}$

本章中“$\mathcal{P}$”是一类问题，他们可以在“多项式时间”内解决。对于“$\mathcal{NP}$”的描述比较复杂。在得到形式化定义和定理之前，我们描述几个将贯穿本章的问题。之后我们给出 $\mathcal{P}$ 和 $\mathcal{NP}$ 的定义。

13.2.1 决策问题

本章描述的许多问题都来源于优化问题（他们成为*组合最优化问题*），但是他们也可以形式化为决策问题。$\mathcal{P}$ 和 $\mathcal{NP}$ 类问题，他们将在后面的小节中定义，就是一类决策问题。基本的，一个*决策问题*是一个只能有两个答案 *yes* 和 *no* 的问题。某些*输入*可以引起麻烦。一个*问题实例*是问题和特定输入的组合。经常的一个决策问题有两个部分：

1. *实例描述*部分定义了除输入外的其他信息。
2. *问题*部分给出了实际的 yes 或 no 问题；问题包含了定义在实例描述中的变量。

对于给定的输入决策问题正确的输出不是 *yes* 就是 *no*。因此，决策问题可以抽象为所有输入到集合{yes,no}的映射。

为了展示为什么输入的精确语句是重要的，考虑下面的两个问题：

1. *实例*：一个无向图 G=(V, E)。
   问题：G 包含一个有 k 个顶点的 clicque 吗？（一个 clique 是完全子图：子图中每一对顶点之间都有边连接。）
2. *实例*：一个无向图 G=(V, E)和整数 k
   问题：G 包含一个有 k 个顶点的 clicque 吗？

两个问题是一样的，但是第一个问题 k 不是输入的一部分，所以一个实例到另一个实例 k 是不能变的；换句话说，k 是*常量*。这个问题恰好可以在$O(k^2n^k)$内的算法解决。如果 k 是

一个常数，算法运行在多项式时间。在第二个问题中，k 是一个输入的一部分，所以它是变量。算法仍然运行在$O(k^2n^k)$，但是这个表达式不是一个多项式的，因为 n 的指数是一个变量。

13.2.2 一些简单的问题

我们将在这一章中学习一些问题。在有些情况下问题是实际应用产生问题的简化或抽象。经常的，困难的问题被简化以尝试做一些改进和得到一些认识，希望这些认识能用于改进原来的问题。

定义 13.1 图的着色和彩色数（chromatic number）

图 G=(V, E)的*着色*是一个映射 C: V→S，这里 S 是一个有限集合（“颜色”），如果 $vw \in E$ 则 C(V)≠C(w)；换句话说，邻接顶点不能赋以相同的颜色。

G 的*彩色数*，表示为χ(G)，是着色图 G 需要的最小颜色数（就是说，最小的使 G 的着色 C，|C(V)|=k 成立的 k）。

问题 13.1 图的着色

我们给出无向图 G= (V, E)要求着色。

优化问题：给出 G，决定χ(G)（并得出一个最优着色，就是说，一个仅用χ(G)种颜色的着色方案）。

决策问题：给出 G 和正整数 k，有没有一种仅用 k 种颜色的 G 的着色？（如果有，称 G 是 k 种可着色。）

图的着色问题是特定类型时序安排问题的抽象。例如，假设你的大学的期末考试需要在一个星期内安排，每天 3 场考试，即总共有 15 个“插槽”。有些课程，比如微积分 1 和物理 1，必须安排在不同的时间，因为有些学生同时修了这两门课。令 V 是课程的集合，令 E 是不恩那个一起考试的课程对的集合。则考试能不冲突的安排在 15 个插槽中，等价与图 G=(V ,E) 可以用 15 种颜色着色。

问题 13.2 带惩罚的任务调度

假设 n 个任务 $J_1,\ldots, J_n$ 在一个时刻只能执行一个。我们给出执行时间 $t_1, \ldots, t_n$，和时间期限 $d_1, \ldots, d_i$（从第一个任务开始的时间开始计算），和超过时间期限的惩罚 $p_1, \ldots, p_n$。假定执行时间，时间期限和乘法都是正整数。一个任务*调度*是{1, 2, …, n}的排列π，这里 $J_{\pi(1)}$表示首先要做的任务，$J_{\pi(2)}$表示第二要做的任务，依此类推。

对于一个特定的调度，如果任务的 $J_{\pi(2)}$在时间期限 $d_{\pi(j)}$之后完成则，第 j 的任务的惩罚表示为 P_j，定义为 $P_j=p_{\pi(j)}$，否则 $P_j=0$。对于特定的调度总的惩罚是

$$P_\pi = \sum_{j=1}^{n} P_j$$

优化问题：决定最小的可能惩罚（并找到最优的调度——有最小的总惩罚）。

决策问题：在输入中给出非负整数 k，是否存在一个调度使得 $P_\pi \le k$？

问题 13.3 装箱

假设我们有无限个容量为 1 的箱子，和 n 个大小 $s_1, \ldots, s_n$ 物体，这里 $0<s_i\le 1$（s_i 是有理数）。

优化问题：决定可以装下物体的最少的箱子数量（并找到最小的装箱方法）。

决策问题：给出整数 k，对象是否能装入 k 个箱子中？

装箱的引用包括，计算机的存储器中 packing 数据（例如，磁盘扇区上的文件，程序段中的内存页，不足一个内存字的域）和产品的装填顺序（例如纺织品和木材）以装入大的标准大小的箱子中。

问题 13.4 背包

假设我们有一个容量为 C 的背包（一个正整数）和 n 个大小为 $s_1, \ldots, s_n$ 的物体和“收益”$p_1, \ldots, p_n$（这里 $s_1, \ldots, s_n$ 和 $p_1, \ldots, p_n$ 是正整数）。

优化问题：找到背包能装下的最大收益（并找到达到这个收益的物体的子集）。

决策问题：给出 k，是否存在一个物体的子集能装入背包，且收益超过 k？

背包问题在经济计划和装填或是 packing 问题中有很多应用。例如，背包问题可以描述投资决定的问题，投资问题中“大小”是一项投资需要的资金，C 是总的资金，而投资的“收益”是期望的回报。在这个问题更复杂版本的应用中，对象是太空飞行中需要组织的任务或试验。他们的“大小”除了对仪器设备的需要外，还有对能源的需要，还需要一个小组花时间来执行他们。在太空飞行中，能源，时间都是有限的。每一个任务有它的价值。那种任务的组合有最大的总价值。

注意前面的 3 个问题都是以最小值描述的，但是背包问题是以最大值描述的。下一个问题是背包问题的简化版本。

问题 13.5 子集和

输入是一个正整数 C 和 n 个物体，物体的大小是正整数 $s_1, \ldots, s_n$。

优化问题：选取物体的子集，子集的和小于 C，那么最大的子集和是多少？

决策问题：是否存在一个物体的子集，他们的和恰好等于 C？

问题 13.6 可满足性

一个 popositional（或布尔）变量是一个可以赋值为 *true* 或 *false* 的变量。如果 v 是逻辑变量，则 $\overline{v}$，非 v，是 *true* 当且仅当 v 是 *false*。一个 *literal*（译注：指合取公式中的一项）是一个逻辑变量或是逻辑变量的非。一个逻辑公式以递归形式定义，逻辑公式是表达式，表达式是逻辑变量或是逻辑常量（也就是 true 和 false），或者由表达式加逻辑运算符组成。一个逻辑公式可以表示为多种形式，包括函数符号（例如，*and(x,y)*），运算符符号（例如，$(x \wedge y)$），或者表示为一颗树，每一个 internal 节点是逻辑运算符，每一个叶子是逻辑变量或者是 *true* 或 *false*。如果给每一个变量赋值，通过运算符的规则可以给公式计算一个真值。

逻辑公式的一种特定规范形式叫做 *合取范式*（*conjunctive normal form*），使用广泛。子句 *clause* 是一个由逻辑或（$\vee$）分隔的 literals 的序列。一个逻辑公式是合取范式（CNF），当它是由逻辑与（$\wedge$）分隔的子句序列。一个合取范式的逻辑公式的例子是

$$(p \vee q \vee s) \wedge (\overline{q} \vee r) \wedge (\overline{p} \vee r) \wedge (\overline{r} \vee s) \wedge (\overline{p} \vee \overline{s} \vee \overline{q})$$

这里 p，q，r 和 s 都是逻辑变量。贯穿本章“CNF 范式”始终指逻辑 CNF 范式。

一套逻辑变量的 *真值赋值 truth assignment* 是给集合中每一个变量赋予 true 或 false，换句话说是集合的布尔值函数。一套真值赋值 *满足*一个公式，当它使得这个公式的值为 *true*。注意一个 CNF 范式被满足，当且仅当它的每一个子句都是真，而子句是真当且仅当子句中至少一个 literal 是真。

决策问题： 给出一个 CNF 范式，存在一个真值赋值满足它吗？

这个决策问题叫 *CNF-可满足性*，或简称 *可满足性*，经常简写为 *CNF-SAT* 或 *SAT*。可满足性问题在自动定理证明中使用。这个问题在本章中作为中心例子。

下面的可满足性问题的简化，叫做 3-可满足性，3-CNF-可满足性，3-SAT，3-CNF-SAT，也是很重要的。（我们理出多个名字和简写是因为没有标准的术语，这个问题经常被提及。）

决策问题：给出一个 CNF 范式，它的每一个子句最多允许 3 个 literals，存在一个真值赋值满足它吗？

问题 13.7 哈密顿回路 Hamiltonian cycles 和哈密顿路径 Hamiltonian paths

无向图的 *哈密顿回路*是一个简单的回路，它穿过每一个定点一次且仅一次。有时候使用词 *cicuit* 代替 *回路*。

决策问题：给出一个无向图，它有哈密顿回路吗？

一个相关的优化问题是旅行商问题，或最小漫游路线，下面将描述这个问题。

无向图的哈密顿路径是一条简单路径，它穿过每一个定点一次且仅一次。

决策问题：给出一个无向图，它有哈密顿路径吗？

两个问题都可以推广到有向图，在有向图时他们叫“有向哈密顿回路（路径）问题”。一个哈密顿回路的变体是指定了开始和结束顶点。

问题 13.8 旅行商

这个问题最广泛的名字是旅行商问题 *traveling salesperson problem*（简写为 *TSP*），但是也叫*最小漫游问题 minimum tour problem*。商人想最小化访问一个区域内所有城市的费用（时间，或是距离），而且要回到出发点。其他应用包括垃圾收集车的路线和包裹速递的路线。

优化问题：给出一个完全带权图找到最小权的哈密顿回路。

决策问题：给出一个完全带权图和证书 k，存在权最多只有 k 的哈密顿回路吗？

传统版本将图当作无向的；就是说，边的两个方向的权都是一样的。哈密顿回路问题也有有向图的版本。

这些问题的有用性和显而易见的简单可能引起你的兴趣；在行动之前将邀请你设计解决他们的算法。

13.2.3 $\mathcal{P}$ 类

对于上一小结的问题没有已知的算法能保证在合理的时间内得出结果。我们 buhi 严格的定义“合理”，但是我们将定义一类 $\mathcal{P}$ 问题，它包括有合理的效率的算法。

定义 13.2 多项式低限

说一个算法是*多项式低限*的，当他的最坏情况复杂性是输入规模的多项式函数（也就是当有一个多项式 p，使得每一个规模为 n 的输入，算法都将在 p(n)步内终止）。

一个问题是多项式低限的，当它有一个多项式算法。

第一章到第十二章所有的算法都是多项式低限的，只有少数可选练习除外。

定义 13.3 $\mathcal{P}$ 类

$\mathcal{P}$ 是一类决策问题， 这类问题有多项式的低限。

$\mathcal{P}$ 仅定义在决策问题上,but you usually will not go wrong by thinking of the kinds of problems studied earlier in this book as being in $\mathcal{P}$。

用多项式低限作为分类标准似乎有点奢侈——多项式可以到非常大。但是，有好几个理

由选择多项式。

首先，尽管并不是 $\mathcal{P}$ 类中的每一个问题都有可接受的有效率算法，但是我们可以说如果一个问题不属 于 $\mathcal{P}$ 类，它一定是极度耗时间的，且在实践中几乎是不可解决的。本小节前面描述的所有问题可能都不属于 $\mathcal{P}$ 类；目前对于这些问题没有已知的算法能在多项式低限 内解决，而且这个领域的大多数研究者相信没有这样的算法存在。因此虽然 $\mathcal{P}$ 的定义可能太宽泛以致不能作为“问题是否在合理的时间内可解”的标准，但是它提供 了一个有效的标准——不在 $\mathcal{P}$ 中肯定是不可接受的。

其次，将多项式低限作为定义 $\mathcal{P}$ 的原因是多项式有好的“闭包”属性。一个复杂问题的算法可以通过许多简单问题的算法获得。简单算法可能需要其他算法的输出或 是中间结果。加法、乘法和 composition 简单算法的复杂性就可以得到复杂算法的复杂性。因为多项式对这些运算都是闭合的，任何以几个多项式低限的 算法通过自然方式构建起来的算法也是多项式低限的。没有更小的在用的复杂性低限函数对于这些运算是闭合的。

最后，使用多项式低限使得 $\mathcal{P}$ 独立与使用的计算形式模型。有好几个形式模型（算法的形式定义）被用于证明算法和问题复杂性的严格定理。这些模型允许的操作类 型，可用的内存资源和赋予不同操作的消耗都不一样。一个问题在一个模型中需要Θ(f(n)步，在另一个模型中可能需要多于Θ(f(n)步，但是对于所有的现实的模型，如果 一个问题是多项式低限，那么在别的模型中也是多项式低限。

13.2.4 $\mathcal{NP}$ 类

许多决策问题（包括我们所有的简单问题）都可以表达成一个存在问题：是否存在图 G 的 k 种颜色着色？是否存在一个真值赋值使得给定的 CNF 范式为真？对于给 定输入，“解决方案”是一个对象（例如，一个图的着色，或一个真值赋值），这个对象满足问题的标准从而得到一个 yes 回答（例如，图的着色最多用 k 种颜 色；真值赋值使得 CNF 范式为真）。“可能的解决方案”是一个简单的有恰当类别的对象——他可能满足也可能不满足标准。有时我们使用术语 certificate 来表示可能的解决方案。宽松的讲，$\mathcal{NP}$ 是一类决策问题，对于给定输入的一个可能的解决方案能快速的验证（在多项式时间内）是否是一个真正的解决方案（即它是否满足问题所有的要求）。更正式一点，问题的输入和可能的解决方案必须能用来自有限集合的符号串描述，例如，计算机键盘的字符集 合。我们需要使用一些符号来约定对图、集合、函数等等的描述。对于特定问题采用的这套约定称为问题的 encoding。串的大小就是里面符号的数量。所选 集合的符号串不是问题相关的有效编码 ；他们只是毫无疑义的组合 。正式的，一个问题实例的输入和可能的解决方案可以是字符集种的任意串。验证一个可能的解决方案包括验证串是一个有意义的（就是说，有正确的语法）所需要类别对象的描述，还有就是要检查它是否满足问题的标准。所以任一字符集的串都可以认为是问题实例的 certificate。

可能有一些决策问题对于“解决方案”和“可能的解决方案”没有自然的解释。抽象的说，一个决策问题仅是一个函数，函数的输入是一个串的集合，输出是 $\{yes,no\}$。$\mathcal{NP}$ 的一个形式化定义考虑所有的决策问题。定义使用我们马上将要定义的非确定性算法。尽管这样的算法在实践中是不现实或是没有用的。， 他们对于分类问题是很有用的。

定义 13.4 非确定性算法

一个*非确定性算法*有两个阶段和一个输出步：

1. 非确定性算法“猜”阶段。字符集的有些完整的任意串，*s*，“写在”内存指定的地方。算法运行的每一步，写的串可能不同。（这个串是certificate；它可以认为是一个问题解决方案的一个猜测，所以这个阶段称为猜阶段，但是 s 可能是无意义的串。）

2. 确定性“验证”阶段。一个确定性子例证开始执行。除了决策问题的输入外，子例程可能使用 *s*，或可能忽略 *s*。最终它返回 *true* 或 *false*——或者它进入死循环不会停止。（认为验证阶段就是检查 *s*，看 *s* 是不是决策问题输入的一个解决方案，也就是如果它为决策问题的输入得到 *yes* 回答。）

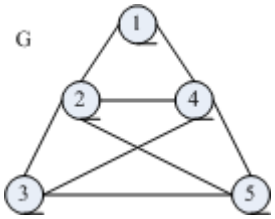
3. 输出步。如果验证阶段返回 *true*，算法输出 *yes*。否则没有输出。

执行非确定性算法一次执行的步数定义为两个阶段步数的和；就是说，步数是写 *s*（简化为 *s* 中字符的数量）加上第二步验证 *s* 执行的步数。

我们也可以以一种显式子例程结构描述非确定性算法。假设 genCertif 产生一个任意的 certificate。

```
void nondetA(String input)
{
    String s=getCertif();
    boolean checkOK=verifyA(input, s);
    if(checkOK)
        Output “yes”;
    return;
}
```

正式的说，一个算法对于任何输入都应该终止，以及只要给算法同样的输入就一定会有同样的输出。这对与非确定性算法是不一定的；对于特定的输入 x，一次运行 得到的输出（或者可能没有输出）可能和另一次得到的不一样，因为他们依赖于 s。那么一个非确定性计算对于特定决策问题和特定输入得到的“回答”（称为 **A**）是什么呢？A 对于 x 的回答定义为 *yes*，当且仅当有些 A 的运行过程给出了 yes 输出。如果对于所有 s 回答都是 no，就没有输出。使用我们 s 的非正式符号作为一个可能的解决方案。A 对于 x 的回答是 yes，当且仅当有一个可能的解决方案可以“工作”。



输入：k=4，n=5，G 的边：(1,2)，(1,4)，(2,4)，(2,3)，(3,5)，(2,5)，(3,4)，(4,5)

图 13.1 非确定性图着色的输入（例 13.1）

例 13.1 非确定性图着色

假设问题是决定一个无向图是否可以用 k 种颜色着色。非确定算法的第一阶段将写一个串 s，在第二阶段将其解释为一个可能的着色。串 s 可以解释为一个整数 $c_1,c_2,...,c_q$ 的列表，q 取决于 s 的长度。算法第二阶段

可以将这些整数解释成颜色赋予顶点：赋值 c_i 到 v_i。为了验证着色是否有效，第二阶段还需要：

1. 检查列表中是否有 n 种颜色（也就是 q=n）。
2. 检查每一个 c_i 是否都在 1,...,k 的范围内。
3. 扫描图的边的列表（或者扫描邻接矩阵），对于每一条边 v_iv_j 检查 $c_i \neq c_j$；也就是一条边的两个顶点的颜色不同。

如果所有的测试都通过，验证阶段返回 true，算法输出 yes。如果 s 不满足所有的要求，验证阶段可能返回 false 或进入死循环，对于这次执行算法不产生输入。

举个例子，令输入实例是图 13.1 中的 G，k=4，在这种情况下问题是，“G 可以被 4 种颜色着色吗？”为了更好的可读性，我们用字母代替数字表示颜色 B（蓝）、R（红）、Y（黄）和 O（橙）。这里有一个可能的 certificate 串 s 和验证阶段返回值的列表。

s	输出	原因
RGRBG	false	v2 和 v5 都是绿色且相邻
RGRB	fasle	不是所有顶点都着色了
RBYGO	false	使用了过多的颜色
RGRBY	true	有效的 4 种颜色着色
R%*,G@	false	错误的语法

既然（至少）有一个结果返回 true，所以对于输入（G，4）非确定性算法的回答是 yes。

如果对于每一个回答为 *yes* 的规模为 *n* 的输入有一个（固定的）多项式 *p*，有一些回答 *yes* 的输出最多在 *p(n)*步内得到答案，则认为一个非确定性算法有多项式低限。（If there is a (fixed) polynomial p such that for each input of size n for which the answer is yes, there is some execution of the algorithm that produces a yes output in at most p(n) steps，则认为一个非确定性算法有多项式低限。）

定义 13.5 类 *NP*

NP 是一类决策问题，这类问题有多项式低限的非确定性算法。（名字 *NP* 是指“*N*ondeterministic *P*olynomially bounded。”）

定理 13.1 图着色、哈密顿回路、哈密顿路径、带惩罚的任务调度、装箱、子集的和问题、背包问题、满足性和旅行商问题（问题 13.1 到 13.8）都属于 *NP*。

证明　证明是直接的，留在练习中。例如，前面描述的检查图的着色就可以简单的在多项式时间内完成。

定理 13.2　$\mathcal{P} \subseteq \mathcal{NP}$

证明　一个决策问题的普通（确定性的）算法是非确定性算法的特例，只有细微不同。如果 **A** 是一个决策问题的确定性算法，只需要令 **A** 是非确定性选法的第二阶段，但是 修改 A 使之只要都输出 *yes*，**A** 就返回 *true*，只要输出 *no* 就返回 *false*。**A** 只是忽略第一阶段写下的串的内容，继续按正常计算。一个非确定性算法可 以在第一阶段做 0 步工作（写一个空串），所以如果 **A** 运行在多项式时间，那么以带修改过的 **A** 做第二阶段的非确定性算法也能运行在多项式时间。如果 **A** 输出 *yes*，非确定性算法也输出 *yes*，否则什么也不输出。

问题是 $\mathcal{P}$=$\mathcal{NP}$?或是 $\mathcal{P}$ 是 $\mathcal{NP}$ 的真子集。换句话说，对于用非确定性“猜”的方法能在多项式时间解决，但是用普通算法不能在多项式时间解决的问题，非确定性算法比确定性算法更强大吗？如果一个问题属于 $\mathcal{NP}$，有多项式时间低限 p，那么如果我们检查所有长度不超过 $p(n)$的串（也就是说，为每个可能的串运行一次 非确定性算法的第二阶段），我们就可以（确定性的）给出正确的答案（*yes* 或 *no*）。检查每一个串需要的步数最多是 $p(n)$。麻烦是有太多的串要检查。如果我们的字符集包括 c 个字符，有 $c^{p(n)}$个长度为 $p(n)$的串。串的数量是 n 的指数，而不是 n 的多项式。当然有其他方法解决这个 问题：使用对象的某些属性和技巧来设计算法，使之不检查所有的可能。例如当排序时，我们不检查所有的 n！种排列 。本章讨论的问题的困难是这样的思路还没有 产生有效的算法；所有已知的算法不是检查所有的可能，就是用来检查工作的技巧还不足以在多项式低限内。

一般都相信 $\mathcal{NP}$ 是一个比 $\mathcal{P}$ 大的集合，但是还没有一个在 $\mathcal{NP}$ 中的问题被证明不在 $\mathcal{P}$ 中。许多 $\mathcal{NP}$ 中的问题（包括 13.2.2 小节中所有的简单问题）都没有已知的多项式低限算法，但是也没能证明这些算法有高于多项式低限。所以我们前面问到的问题 $\mathcal{P}$=$\mathcal{NP}$?，仍然悬而未决。

13.2.5 输入的规模

考虑下面的问题。

问题 13.9

给出一个正整数 n，存在整数 j,k>1，使得 n=jk?（就是说，n 是否是非素数？）

这个问题在 $\mathcal{P}$ 中吗？考虑下面的算法，它查找 n 的因子。

```
factor = 0;
for( j=2; j<n; j++)
{
    if(( n mod j ) ==0)
    {
        factor=j;
        break;
```

```
        }
    }
    return factor;
```

循环体执行的次数少于 n，而且可以确定（n mod j）能在$O(\log^2(n))$内计算，所以算法的运行时间保守估算是在$O(n^2)$。然而决定一个数是素数或者决定它不是素数的问题，目前*不知道*是否在 $\mathcal{P}$ 中，而且事实上因子分解大整数的难度是各种密码算法的基础，因为现在认为这是一个难题。这种矛盾的原因是什么呢？

最初测试算法的输入是整数 n，但是 n 的 size 是什么？直到现在，我们一直使用任意只要是方便和合理的输入规模的度量；它对于计数单独的字符 s 和 bits 并不重 要。当我们度量一个输入的规模时可能导致一个算法是多项式或是指数的，我们不得不小心对待。输入的规模是输入写成串的字符的数量。例如，如果 n=150， 我们写 3 个数字，而不是 150 个数字。因此一个以十进制表示的整数 n 有规模约 $\log_{10}n$。如果我们选择数字在计算机内部的表示法——二进制，那么 n 的规模约是 lgn。两者有一个常数因子的差别；$\log_2 n=\log_2 10\log_{10}n$，所以我们使用的数制不是关键。但是关键点是如果输入的规模是 $\log_{10}n$，而算法运行的时间是 n，则算法的运行时间是输入规模的指数函数（$n=10^s$）。因此前面决定 n 是否素数算法不属于 $\mathcal{P}$。目前没有已知算法能在多项式时间进行素数测试。问题“一个整数 n 是否是素数？”属于 $\mathcal{NP}$。

在本书前面我们考虑的问题中，我们使用的描述输入规模的的变量对应（多或少）输入数据的量。例如，我们使用待排序关键字的数量 n 作为排序问题输入的规模。 每一个待排序关键字都是以二进制表示的，但是既然有 n 个关键字，则输入至少有 n 个符号。所以，如果算法的复杂性是以 n 的多项式低限，对于精确的输入规模也是多项式低限。

同样的，我们使用(m+n)作为图输入的规模，但是所有的边需要显式的列出，所以在输入中至少要 m 个符号。尽管没有必要在输入中列出所有的 n 个顶点，在所有的关心每一个顶点的问题中都会附带上某些边，所以(n+m)最多是输入中符号数量的 3 倍。进一步的，如果如果算法的复杂性是以(n+m)的多项式低限，对于精确的输入规模也是多项式低限。

如果两个输入规模的度量中的每一个都在另一个的多项式函数内，则决定问题是否属于 $\mathcal{P}$ 将依赖于使用那一个度量。以排序问题为例，如果一个度量是关键字的数量 n，第二个是 nlg(最大的关键字)（最大以 bits 数衡量），我们有 $n \in O(n\log(\max Key))$ 和 $n\lg(\max Key) \in O(n^2)$。因而每一个度量都在另一个的多项式函数内。

所以我们一般不得不放弃完全精确的输入规模度量。但是我们必须小心行事，尤其当算法的运行时间以输入*值*表示时是一个多项式函数的时候，如同前面提到的素数测试问题。

前面提到的简单问题中有少数几个有动态规划的解决方法，乍一看似乎有多项式低限，但是同素数测试问题一样，没有。例如，复习一下子集和问题：是否存在 n 个对象（大小分别是 $s_1,s_2,...s_n$）的子集其和恰好等于 C？使用第十章的技术，可以用 n×C 的表格来表示，计算表格的数据只需要很少的步骤（参看练习 13.5a）。类似的不同版本的背包问题也有动态规划解决方案。

子集和问题的动态规划解决方案运行在$\Theta(nC)$。既然输入中有 n 个对象，n 是没有问题的，但是 C 的值比输入的指数要大（一般都是），因为 C 的数据至少需要 lgC 比特来表示。因此动态规划解决方案不是一个多项式低限算法。当然，如果 C 不太大，在实践中算法可能是有用的。

13.3 $\mathcal{NP}$ 完全问题

$\mathcal{NP}$ 完全是用来描述 $\mathcal{NP}$ 类中最困难的问题的术语，如果 $\mathcal{NP}$ 完全问题有多项式低限，则所有的 $\mathcal{NP}$ 问题都应该有多项式低限。

13.2.2 小节描述的有些简单问题似乎比其他的简单，事实上，他们的最坏情况复杂性却是不同（都是快速增长的函数，$2^{\sqrt{n}}$，2^n，$(n/2)^{n/2}$，$n!$ 等），但是令人惊讶的是，他们是等价的，即如果任一一个属于 P，他们都将属于 P。他们都是 $\mathcal{NP}$ 完全。

13.3.1 多项式约简 Polynomial Reductions

$\mathcal{NP}$ 完全的形式化定义使用了约简，或是一个问题到另一个问题的转换。假设我们想解决问题 **P** 且我们已经有另一个解决另一个问题 **Q** 的算法。假设我们还有一个函数 **T**，**T** 接受 P 的输入 x，得到 T(x)。使得 **P** 正确的输入也使得 Q 是 *yes*，当且仅当 **Q** 对于 T(x)是 *yes*。则通过组合 **T** 和 **Q** 的算法，我们得到一个 **P** 的算法。参看图 13.2。

13.3.2 一些已知的 $\mathcal{NP}$ 完全问题

13.3.3 什么使得问题困难？

13.3.4 优化问题和决策问题

13.4 近似算法

上百个重要的应用都是 $\mathcal{NP}$ 完全的。如果我们必须解决其中一个我们能做什么？有很多可能的途径。即使没有多项式低限算法存在，已知算法的复杂性还是有很大的 差别；我们通常可以尝试为一个开发一种最有效率的。我们可以关注平均行为而不是最坏情况 and look for algorithms that are better than others by that ctiterion, or more realistically, 我们能找到一种算法仅对最经常发生的输入能很好的工作；这个选择可能更加依赖于经验主义而不是严格的分析。

在这一节里面我们学习解决 $\mathcal{NP}$ 完全优化问题的一种不同的办法：使用快速（就是有多项式低限）算法，其不保证能得到最优解但是能比较接近最优解。这样的算法称之为*近似算法*或是*启发式算法*。启发是"拇指的规则（rule of thumb）"，就是通常似乎能找到更好情况的想法，但是可能不能证明确实是好。

在许多应用中一个近似解决方案就足够好了，特别是当需要考虑求解时间时。例如，你不能通过找到一个最优的任务调度方案来获胜，如果找到最优解的计算时间已经导致了最糟糕的惩罚的话。

策略或启发，就像他们的名字一样，被许多近似算法用来简化问题，然而他们能提供令人惊讶的好结果。他们中的许多是*贪婪*启发。在第 8 章我们学习了几种贪婪算法他们都得到了最优解；这一章里面贪婪算法不能得到最优解。为了得到近似算法的精确语句（他们的结

果好的何种程度，不是他们要运行多少时间），我们需要几个定义。再下面的段落中，假设一个特定优化问题 **P** 和也定的输入 *I*。

定义 13.9 可行解决方案集合

一个*可行解决方案*（*feasible solution*）是一个正确类型的对象，但是不必要是最优的一个。*FS(I)*是 *I* 的可行方案集合。

例 13.3 可行解决方案集合

对于图着色问题和一个输入图 G 来说，*FS(G)*是使用任意数量颜色 G 的所有有效的着色的集合。

对于装箱问题和输入 $I=\{s_1,...,s_n\}$，*FS(I)*是使用任意数量箱子所有有效的装箱方案的集合（也就是，all partitions of *I* into disjoint subsets $T_1,...,T_p$, for some p, such that total of the s_i in any subset is at most 1.）

对于一个任务调度问题的输入可行的解决方案是 n 个任务的全排列。

定义 13.10 价值函数

对于输入实例 I，和可行的解决方案 x，函数 *val*(I,x)返回优化参数的价值。

例 13.4 价值函数

1. 对于图的着色，*val*(G,C)是着色 C 使用的颜色数量。
2. 对于装箱问题，如果 $T_1,...,T_p$ 是输入 *I* 中对象可行的划分，则 $val(I,(T_1,...,T_p))=p$，使用的箱子的数量。
3. 对于任务调度，$val(I,\pi)=P_\pi$，调度π的惩罚。

读者识别可行的解决方案集合以及解决方案的价值函数一定很简单。

定义 13.4 最佳价值

依赖于具体问题，我们想找到一个解决方案，其最小化或是最大化的*价值*；令“最好”是分别是“最小”或是“最大”。则 $opt(I)=best\{val(I,x)\mid x\in FS(I)\}$。就是说，它是所有可行的解决方案能达到的最好的价值。*I* 的一个*最佳价值*是 *FS(I)*中的 x 有 *val(I,x)=opt(I)*。

定义 13.12 近似算法

一个问题的近似算法是多项式时间算法，它对于输入 I，输出 FS(I)的一个元素。

这里有好几种方式描述近似算法的质量。通常最有用的是算法输出的价值和最优解决方案的比值（尽管有时我们可能想看看两者之间的绝对差别）。令 **A** 是一个仅是算法。我们用 **A**(I)表示为输入 I 选择一个可行的解决方案 **A**。我们定义：

$$r_A(I) = \frac{val(I, A(I))}{opt(I)} \qquad \text{对于最小化问题} \qquad (13.3)$$

$$r_A(I) = \frac{opt(I)}{val(I, A(I))} \qquad \text{对于最大化问题} \qquad (13.4)$$

在两种情况下，$r_A \geq 1$。为了总结 **A** 的行为，我们将考虑最坏情况的比率。

13.5 装箱

装箱问题是一类问题的简化，这类问题在实践中经常遇到：如何组合或存储不同大小不同形状的对象，使得浪费的空间最小。

13.5.1

13.5.2 其他启发

13.6 背包

13.7 图的着色

13.7.1 一些基础技术

13.7.2 近似图着色是困难的

13.7.3 Wigderson's 图着色算法

13.8 TSP 旅行商问题

对于旅行商问题（TSP），我们给出一个完全带权图，我们想找到一个有最小权的漫游路线（经过所有顶点的回路）。这个问题在路由和调度问题中有大量的应 用。所以从理论上和实践上对这个问题的研究热情都很高。本小节展示一些简单的近似算法，然后给出一个定理（没有证明）这个定理指出大概一个好的近似算法不太可能存在。

13.8.1 贪婪策略

在第 8 章我们学习了两种在带权无向图中找最小生成树的贪婪算法（Prim's 算法和 Kruskal's 算法）。两个算法都能自然简单变成的 TSP 问题的算法。在这一小节中，我们讲考察这些方法。

复习优化问题的贪婪方法，由一个选择序列组成，其中每一个选择都朝向最好的“短期目标”，而这些“短期目标”是容易计算的。一旦作出了选择，就不能撤销， 即使后来的事实证明这是一个糟糕选择。一般的，贪婪策略有启发：启发似乎朝好的方向去，但是其中有很多并不总是导向最优解决方案或者并不总是有效率。在第 8 章，我们能证明 Prim’s 和 Kruskal's 对最小生成树的贪婪策略*总是*能有效率的产生最优解决方案。

复习 Prim‘s 算法，它从一个任意顶点开始，生成一颗树。在主循环的每一个迭代步骤它选一条已在树中顶点到边缘顶点之间的边；它“贪婪的”的选择权最小的。

另一方面，Kruskal's 算法“贪婪的”从图中剩下的边中选择权最小的，且不能和已选择的边构成回路的边。Kruskal's 算法任意时刻选出来的边构成的子图可以不是连同的；它是森林，不必要是树（最后必须是树）。

TSP 问题对应的贪婪策略 s 分别叫做最近邻居策略和最短连接策略。

13.8.2 最近邻居策略

*最近邻居策略*相当的简单。在 Prim's 算法中，当我们选择一条新边，我们能从树的任一顶点分支。这里我们构造的是回路不是树，所以我们

13.9.3 最短连接策略

13.8.4 TSP 的近似算法有多好？

不必惊讶这些简单的多项式时间 TSP 策略不能产生最小权的漫游路线。我们已经说了 TSP 是 NP 完全，可能没有算法能在多项式时间解决。（当然这不是说最近邻居策略和最短连接策略总是不能产生最优漫游路线，有时他们碰巧能产生最小权的漫游路线。）

最近邻居策略和最短连接算法是 TSP 的近似算法。我们能建立这两个算法产生的漫游路线的权和最优漫游路线的权之间差的低限吗？不幸的是，不能。考虑下面的定理。

> **定理 13.22** 令 **A** 是任一 TSP 问题的近似算法。如果有任一常量 c 对于所有实例 I 都有 $r_A(I)\le c$，则有 $\mathcal{P}=\mathcal{NP}$。
> **证明** 参看本章后面的 Notes 和 References。

这个定理说“保证”好的 TSP 的近似算法不太可能存在，就像不太可能存在 TSP 的多项式时间算法一样——即使“好”定义的非常宽松，允许为最优漫游路线的 权乘上任意常数因子。但是，如果我们以特定的属性约束输入的图，then there are approximation algorithms for the TSP with bounds on the weight of the tours produced。例如，如果图的边的权表示机场之间的距离，则应该满足三角不等式：

$$W(u,w)\le W(u,v)+W(v,w) \quad \text{对于所有 G 中的 u,v,w 都成立} \qquad (13.8)$$

练习 13.53 描述 TSP 的一个近似算法，如果图满足三角不等式，它保证了产生权最多是最优解两倍的漫游路线。

13.9 DNA 计算

当我们听到*计算机*（*computer*）时，我们想到的是现代的电子计算机。但是这个词有许多其他的含义。几个世纪以来，computer 指那些靠计算为生的人。计算技术的发展从手指开始（计数），到各种机械装置（算盘，加法机，卡片排序机），直到现代的电子计算机（从上十吨的庞然大物到桌上的个人计算机，到便携的嵌入系统）。没有理由认为计算机技术的发展会终止。下一代是什么？可能是基于 DNA，或生物技术的计算机。

13.9.2 DNA 背景

DNA 是脱氧核糖核酸，编码生物特征的基因的原料。这篇简单的背景介绍是为了更好的理解 DNA 是怎么应用到计算上的。从生物学的角度看，它稍微有点简单，有点不严密。

DNA 由一串化学上称为核苷的物质组成。DNA 中有 4 种核苷，每一种以名字的第一个字母表示：腺嘌呤（adenine A）、胞核嘧啶（cytosine C）、鸟嘌呤（guanine G）和胸腺嘧啶（thymine T）。我们可以用 4 个字母编码出任何信息，就像我们能用 0 和 1 编码出任何信息一样。现在可能合成包含特定核苷序列的 DNA 线；就是说创建任意表示数据的 串。

John Waston 和 Francis Crick 发现了 DNA 的双螺旋结构（这项工作为他们赢得了 Nobel 奖）。核苷构成互补对：A 和 T 互补，C 和 G 互补。如果两根核苷线在对应的位置有互补 的核苷，两条核苷线相互依附（且在双螺旋结构中互相缠绕）。例如，参看图 13.16（我们展示了依附在一起的核苷线，但是不是双螺旋的）。互补线互相依附 的事实在哈密顿路径的 DNA 算法中反复用到。有可能发生两条线即使在有些位置不互补也能依附在一起；这是 DNA 处理的一个属性，这可能导致麻烦。

Kary Mullis，一个化学家，开发了称为*聚合酶链反应 polymerase chain reaction*（PCR） 的 process，它复制小的 DNA 样本。（现在 PCR 在基因研究和 forengsics 中广泛使用，而 Mullis 也因为这项工作赢得了 Nodel 奖。）寻找符合我们要求的线的算法中，好几步都用到了 PCR。算法每一步使用的实际生物过程是复杂的，但是我们不得不理解他们以理解算法的逻辑。因此这都是我们需要的背景。

13.9.3 Adleman 的有向图和 DNA 算法

13.9.4 分析和评估

第十四章 并行算法

14.1 概述

整本书中我们用到的大部分计算模型都是通用的、确定性的、随机访问的计算机，这种计算机在一个时刻只能执行一个操作。有几次我们建立特定的模型来为特定的问题得出低限；这些不是通用计算机，但是他们也只能在一个时刻只能执行一个操作。我们将使用*顺序算法*这个术语来描述到目前为止我们学过的这种一个时刻执行一步的所有算法。（他们有时候也被称为*串行算法*。）在这一章里面我们将考虑并行算法，即在一个时刻可以有多个操作并行的执行，也就是说，在同一时刻对于一个问题有多个处理机在工作。

最近，随着多处理机系统变得便宜，连接多处理机的技术越来越先进，建立一个包含大量处理机的通用并行计算机称为可能。本章的目的就是介绍这些概念、形式化模型、技术和并行计算领域的算法。

并行算法对于许多应用程序来说是自然的。在图象处理处理中（例如，机器人的视觉系统）不同的场景可以同时处理，也就是并行的处理。并行可以加速图象显示的计算。在搜索问题中（例如，文献搜索、扫描新的故事和文本编辑），数据库或文本的不同部分可以平行的搜索。模拟程序经常要对被模拟系统中的大量的小部分做同样的计算；这些在每一个模拟步中都可以并行。人工智能程序（可能包含图象处理和大量的搜索）也可以从并行中得到好处。快速傅立叶变换（12.4 节）以专门的并行硬件实现。许多组合优化问题的算法（比如第 13 章描述的 $\mathcal{NP}$ 完全问题的优化版本）含有对巨大数量方案的检查；有些工作可以并行做。在其他应用领域并行计算也能简单的坚实的加速计算。

14.2 并行，PRAM 和其他模型

如果并行计算机中处理机的数量很少，比如 2 个或 6 个，则

14.2.1 PRAM

一个*并行随机访问机器*（*parallel random access machine* PRAM，读作”p ram”）包含 p 个通用处理器，P_0、P_1…、P_{p-1}，所有这些处理器都连接在一个巨大的共享的随机访问的存储器 M 上。这个存储器笨当作一个（非常巨大的）整数数组（参看图 14.1）。处理器有私有的，或者局部的存储器用于他们自己的计算，但是他们之间所有的通信都是通过共享存储器实现的。除非我们特别指出，算法

14.2.2 其他模型

尽管 PRAM 为在并行机器上开发和分析算法提供了一个良好的 framework，但是以实际硬件提供这个模型是困难或者是价格昂贵的。PRAM 假定了一个 复杂的通信网络，这个网络允许所有的处理器在同一处理周期访问任一内存单元，并且能在同一处理周期写任一单元。因此任一处理器能都能在两个处理周期内与另 一个处理器通信：一个处理器在一个处理周期将数据写入内存的一个位置，另一个处理器在下一个处理周期读那个内存位置。其他的并行模型没有这么一个共享内 存，因此限制处理器之间的通信。一个非常贴近实际硬件的模型是*超立方体*（hypercube）。一个超立方体有 2^d 个处理器，这里 d 是*维度*， 每一个处理器都和它临近的处理器连接。图 14.2(a)展示了一个维度是 3 的超立方体。每一个处理器都有自己的内存，且能通过消息传递与其他处理器通信。 每一个处理周期，每一个处理器都可以做计算，然后发消息给它相邻的一个邻居。为了与一个不邻接的处理器通信，处理器必须发送一个带路由信息的消息，消息中 指名了最终目的地；消息最多需要 d 次传递到达目的地。在一个有 p 个处理器的超立方体中，每一个处理器和 $\lg p$ 个其他的处理器相连。

另一类模型叫做*有限度网络*（bounded-degree networks），这种模型更进一步的限制连接 。在有限度网络中，每一个处理器最多直接与 d 个其他处理器连接，d 是常量。对于有限度网络有不同的设计； 图 14.2(b)展示一个 8×8 网络。超立方体和有限度网络比 PRAM 更贴近现实，但是他们的算法难以规范和分析。处理器之间消息的路由本身就是一个有趣 的问题，而这在 PRAM 中是不存在的。

PRAM，虽然不是非常实际，但是开发算法的时候 PRAM 是一种概念简单的模型。因此对于如何在其他的并行模型上模拟 PRAM 有很多努力，特别是那些不共享内存的模型。例如，每一个 PRAM 处理周期都能以大约O(logp)个有限度网络的处理周期模拟。所以我们能为 PRAM 开发一种算法，而且知道这个算法能被转换成实际机器的算法。这个转换可以通过转换程序自动进行。

在第 13 章中，我们定义了 $\mathcal{P}$ 类问题来帮助区分

14.3 一些简单的 PRAM 算法

在这一节中我们将引入一些 PRAM 计算使用的通用技术，还将开发一些简单的算法，两者都说明了一些 PRAM 算法的“flavor”，也都提供了后面使用的一些 blocks 和子例程（subroutine）。

14.3.1 二分扇入技术

考虑在 n 个关键字的数组中查找最大关键字的问题。我们对于这个问题有两个算法：算法 1.3 和在 5.3.2 小节中描述的竞标赛方法。在算法 1.3 中，我们顺序的将剩下的关键字与已经找到的 max 比较。每次比较之后，max 可能改变；我们不能并行的做下一次比较，因为我们不知道该使用那一个 max。但是在竞标赛方法中，元素在“一轮比赛”中分组， 比较。在成功的“一轮比赛”里面，胜者进入下一轮比赛（参看图 5.1）。最大的关键字在 $\lceil \lg n \rceil$ 轮找到。一轮中的所有比较都能同时执行。因此，竞标赛方法自然的是一个并行算法。

在竞标赛方法中，每一轮需要考虑的关键字的数量都会减半，因此每一轮需要的处理器

的数量减半。但是为了保持算法描述的的简短和清晰，我们在每一处理周期为 所有的处理器指定相同的指令。额外工作可能让人糊涂，所以首先看看图 14.3 是有帮助的。图 14.3 展示了实际需要的工作。直线代表*读*操作。折线表示写操作；一个处理器写（它能看到的最大关键字）到与处理器对应的内存单元（例如 P_i 写到 M[i]）。圆圈表示二元操作“比较”两个值；这里它是比较选出两个关键字中大的。“Bookkeeping” computations fit in around the reads and writes. 如果读线从 P_i 到 Pj 的列，意味着 P_i 从 M[j]读，也就是 P_j 写的地方。图 14.4 展示了所有处理器活动的一个完整的例子。阴影部分对应图 14.3，这些部分是影响结果的计算。

14.4 处理写冲突

PRAM 模型根据他们如何处理写冲突而变化。CREW（并行读，排他写）PRAM 模型要求每一步中只有一个单元只有一个处理器能写；同一时刻多于一个处理器写一个单元的算法是非法的。

有几种方法放宽 CREW 的限制，去掉限制后就是 CRCW（并行读并行写）模型（）。

1. 在公共写模型中，同一时刻多个处理器写一个单元是合法的当且仅当他们写的是一个值。
2. 在任意写模型中，当多个处理器同时写一个内存单元时，他们中的任意一个写成功。算法必须在无论哪个处理器赢得了写冲突的时候都能保证工作正常。
3. 在优先级写模型中，多个处理器试图同时写一个内存单元时，有最小索引的处理器能写成功。

这些 CRCW 模型成功的得到加强，而且他们都比 CREW 要强：一个对于前面的模型是合法正确的算法对于后面的都是合法和正确的，但是反之则不然。

模型的不同在于他们在解决不同的问题的时候速度不一样。为了展示这些区别，我们考虑计算 n bits 上的布尔 or 函数的问题。

14.4.1 n bits 上的布尔 or

用算法 14.1 的二分扇入方案，每一个处理器每一轮对 1 对 bits 执行 or 运算，问题在 Θ(logn)时间内解决。这种方法

14.4.2 一个在常数时间查找最大值的算法

14.5 归并和排序

14.5.1 并行归并

14.5.2 排序

14.6 查找连同分量

14.6.1 策略和技术

14.6.2 算法

14.6.3 算法的 PRAM 实现

14.6.4 分析

14.7 加 n 个整数的底限