题的一类特殊方法——它不是问题的答案，而是经过精确定义的[①]、用来获得答案的求解过程。因此，无论是否涉及计算机，特定的算法设计技术都可以看作是问题求解的有效策略。著名的计算机科学家科努思(Donald Knuth)是这样论述这个问题的："受过良好训练的计算机科学家知道如何处理算法，如何构造算法、操作算法、理解算法以及分析算法，这些知识远不只是为了编写良好的计算机程序而准备的。算法是一种一般性的智能工具，一定有助于我们对其他学科的理解，不管是化学、语言学、音乐还是另外的学科。为什么算法会有这种作用呢？我们可以这样理解：人们常说，一个人只有把知识教给别人，才能真正掌握它。实际上，一个人只有把知识教给**计算机**，才能真正掌握它，也就是说，将知识表述为一种算法……比起简单地按照常规去理解事物，用算法将其形式化会使我们获得更加深刻的理解。"

算法研究的核心问题是时间(速度)问题。人们可能有这样的疑问：既然计算机硬件技术的发展使得计算机的性能不断提高，算法的研究还有必要吗？

计算机的功能越强大，人们就越想去尝试更复杂的问题，而更复杂的问题需要更大的计算量。现代计算技术在计算能力和存储容量上的革命仅仅提供了计算更复杂问题的有效工具，无论硬件性能如何提高，算法研究始终是推动计算机技术发展的关键。下面看几个例子。

1. 检索技术

20 世纪 50 年代 ～ 60 年代，检索的对象是规模比较小的数据集合。例如，编译系统中的标识符表，表中的记录个数一般在几十至数百这样的数量级。

20 世纪 70 年代 ～ 80 年代，数据管理采用数据库技术，数据库的规模在 K 级或 M 级，检索算法的研究在这个时期取得了巨大的进展。

20 世纪 90 年代以来，Internet 引起计算机应用的急速发展，海量数据的处理技术成为研究的热点，而且数据驻留的存储介质、数据的存储方法以及数据的传输技术也发生了许多变化，这些变化使得检索算法的研究更为复杂，也更为重要。

近年来，智能检索技术成为基于 Web 信息检索的研究热点。使用搜索引擎进行 Web 信息检索时，经常看到一些搜索引擎前 50 个搜索结果中几乎有一半来自同一个站点的不同页面，这是检索系统缺乏智能化的一种表现。另外，在传统的 Web 信息检索服务中，信息的传输是按 pull 的模式进行的，即用户找信息。而采用 push 的方式，是信息找用户，用户不必进行任何信息检索，就能方便地获得自己感兴趣的信息，这就是智能信息推送技术。这些新技术的每一项重要进步都与算法研究的突破有关。

2. 压缩与解压缩

随着多媒体技术的发展，计算机的处理对象由原来的字符发展到图像、图形、音频、视频等多媒体数字化信息，这些信息数字化后，其特点就是数据量非常庞大，同时，处理多媒

① 算法固有的精确性限制了它所能够解决的问题种类，比如说，我们无法找到一个使人生活快乐的算法，也不能找到一个使人富有和出名的算法。

体所需的高速传输速度也是计算机总线所不能承受的。因此，对多媒体数据的存储和传输都要求对数据进行压缩。声音文件的 MP3 压缩技术说明了压缩与解压缩算法研究的巨大成功，一个播放 3～4 分钟歌曲的 MP3 文件通常只需 3MB 左右的磁盘空间。

3. 信息安全与数据加密

在计算机应用迅猛发展的同时，也面临着各种各样的威胁。一位酒店经理曾经描述了这样一种可能性："如果我能破坏网络的安全性，想想你在网络上预订酒店房间所提供的信息吧！我可以得到你的名字、地址、电话号码和信用卡号码，我知道你现在的位置，将要去哪儿，何时去，我也知道你支付了多少钱，我已经得到足够的信息来盗用你的信用卡！"这的确是一个可怕的情景。所以，在电子商务中，信息安全是最关键的问题，保证信息安全的一个方法就是对需要保密的数据进行加密。在这个领域，数据加密算法的研究是绝对必需的，其必要性与计算机性能的提高无关。

1.1.2　算法及其重要特性

算法(algorithm)被公认为是计算机科学的基石。通俗地讲，算法是解决问题的方法。严格地说，算法是对特定问题求解步骤的一种描述，是指令的有限序列，此外，算法还必须满足下列 5 个重要特性(如图 1.1 所示)：

(1) 输入：一个算法有零个或多个输入。算法的输入来源于两种方式：一种是从外界获得数据，另一种是由算法自己产生被处理的数据。

(2) 输出：一个算法有一个或多个输出。既然算法是为解决问题而设计的，那么算法实现的最终目的就是要获得问题的解。没有输出的算法是无意义的。

(3) 有穷性：一个算法必须总是(对任何合法的输入)在执行有穷步之后结束，且每一步都在有穷时间内完成。

(4) 确定性：算法中的每一条指令必须有确切的含义，不存在二义性。并且，在任何条件下，对于相同的输入只能得到相同的输出。

(5) 可行性：算法描述的操作可以通过已经实现的基本操作执行有限次来实现。

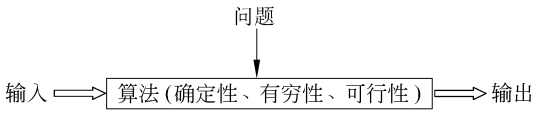

图 1.1　算法的概念

概 念 回 顾

算法和**程序**不同。**程序**(program)是对一个算法使用某种程序设计语言的具体实现，原则上，算法可以用任何一种程序设计语言来实现。算法的有穷性意味着不是所有的计算机程序都是算法。

1.1.3 算法的描述方法

算法设计者在构思和设计了一个算法之后，必须清楚准确地将所设计的求解步骤记录下来，即描述算法。常用的描述算法的方法有自然语言、流程图、程序设计语言和伪代码等。下面以欧几里得算法（用辗转相除法求两个自然数 m 和 n 的最大公约数）为例进行介绍。

1. 自然语言

用自然语言描述算法，最大的优点是容易理解，缺点是容易出现二义性，并且算法通常都很冗长。欧几里得算法用自然语言描述如下：

① 输入 m 和 n；

② 求 m 除以 n 的余数 r；

③ 若 r 等于 0，则 n 为最大公约数，算法结束，否则执行第④步；

④ 将 n 的值放在 m 中，将 r 的值放在 n 中；

⑤ 重新执行第②步。

2. 流程图

用流程图描述算法，优点是直观易懂，缺点是严密性不如程序设计语言，灵活性不如自然语言。欧几里得算法用流程图描述如图 1.2 所示。

在计算机应用早期，使用流程图描述算法占有统治地位，但实践证明，除了一些非常简单的算法以外，这种描述方法使用起来非常不方便。如今，只能在早期有关算法的教材中找到它的踪影了。

3. 程序设计语言

用程序设计语言描述的算法能由计算机直接执行，而缺点是抽象性差，使算法设计者拘泥于描述算法的具体细节，忽略了“好”算法和正确逻辑的重要性，此外，还要求算法设计者掌握程序设计语言及其编程技巧。

欧几里得算法用 C ++ 语言书写的程序如下：

```
#include <iostream.h>
int CommonFactor(int m, int n)
{
    int r=m % n;
    while (r!=0)
    {
        m=n;
        n=r;
        r=m % n;
    }
    return n;
```

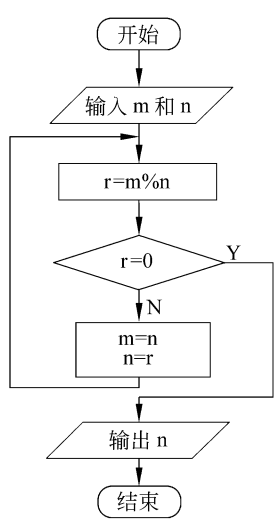

图 1.2 用流程图描述算法

```
}
void main( )
{
  cout<<CommonFactor(63, 54)<<endl;
}
```

4. 伪代码

伪代码(pseudocode)是介于自然语言和程序设计语言之间的方法,它采用某一程序设计语言的基本语法,操作指令可以结合自然语言来设计。至于算法中自然语言的成分有多少,取决于算法的抽象级别,抽象级别高的伪代码自然语言多一些。计算机科学家从来没有对伪代码的书写形式达成过一种共识,只是要求了解任何一种现代程序设计语言的人都能很好地理解。

欧几里得算法采用符合 C ++ 语法的伪代码描述如下:

伪代码

算法 1.1——欧几里得算法

```
1. r=m % n;
2. 循环直到 r=0
   2.1  m=n;
   2.2  n=r;
   2.3  r=m % n;
3. 输出 n;
```

伪代码不是一种实际的编程语言,但在表达能力上类似于编程语言,同时极小化了描述算法的不必要的技术细节,是比较合适的描述算法的方法,被称为“算法语言”或“第一语言”。

1.1.4　算法设计的一般过程

算法是问题的解决方案,这个解决方案本身并不是问题的答案,而是能获得答案的指令序列。不言而喻,由于实际问题千奇百怪,问题求解的方法千变万化,所以,算法的设计过程是一个灵活的充满智慧的过程,它要求设计人员根据实际情况具体问题具体分析。可以肯定的是,发明(或发现)算法是一个非常有创造性和值得付出精力的过程。

在设计算法时,遵循下列步骤可以在一定程度上指导算法的设计[①]。

1. 理解问题

在面对一个算法任务时,算法设计者往往不能准确地理解要求他做的是什么,对算法希望实现什么只有一个大致的想法就匆忙地落笔写算法,其后果往往是写出的算法漏洞百出。在设计算法时需要做的第一件事情就是完全理解要解决的问题,仔细阅读问题的

① 这个一般过程并不是一个绝招,能为任意的问题设计算法,一个公认的事实是——这样的绝招是不存在的。

描述，手工处理一些小例子。

对设计算法来说，这是一项重要的技能：准确地理解算法的输入是什么？要求算法做的是什么？即明确算法的入口和出口，这是设计算法的切入点。

2. 预测所有可能的输入

算法的输入确定了该算法所解问题的一个实例。一般而言，对于问题 P，总有其相应的实例集 I，则算法 A 若是问题 P 的算法，意味着把 P 的任一实例 input∈I 作为算法 A 的输入，都能得到问题 P 的正确输出。

预测算法所有可能的输入，包括合法的输入和非法的输入。事实上，无法保证一个算法（或程序）永远不会遇到一个错误的输入，一个对大部分输入都运行正确而只有一个输入不行的算法，就像一颗等待爆炸的炸弹。这绝不是危言耸听，有大量这种引起灾难性后果的案例。例如，许多年以前，整个 AT&T 的长途电话网崩溃，造成了几十亿美元的直接损失。原因只是一段程序的设计者认为他的代码能一直传送正确的参数值，可是有一天，一个不应该有的值作为参数传递了，导致了整个北美电话系统的崩溃。

如果养成习惯——首先考虑问题和它的数据，然后列举出算法必须处理的所有特殊情况，那么可以更快速地成功构造算法。

3. 在精确解和近似解间做选择

计算机科学的研究目标是用计算机来求解人类所面临的各种问题。但是，有些问题无法求得精确解，例如求平方根、解非线性方程、求定积分等，有些问题由于其固有的复杂性，求精确解需要花费太长的时间，其中最著名的要算旅行商问题（即 TSP 问题，是指旅行家要旅行 n 个城市，要求经历各个城市且仅经历一次，并要求所走的路程最短），此时，只能求出近似解。

有时需要根据问题以及问题所受的资源限制，在精确解和近似解间做选择。

4. 确定适当的数据结构

在结构化的程序设计时代，著名的计算机学者沃思（Wirth）提出了“算法＋数据结构＝程序”的观点，断言了算法和数据结构是构成计算机程序的重要基础。在面向对象的程序设计时代，数据结构对于算法设计和分析仍然是至关重要的。本书所讨论的很多算法设计技术都是基于精心设计的数据结构。

确定数据结构通常包括对问题实例的数据进行组织和重构，以及为完成算法所设计的辅助数据结构。在很多情况下，数据结构的设计直接影响基于该结构之上设计的算法的时间性能。

概 念 回 顾

数据结构是指相互之间存在一定关系的数据元素的集合。按照视点的不同，数据结构分为逻辑结构和存储结构。数据结构从逻辑上分为 4 类：集合、线性结构、树结构和图结构，常用的存储结构有两种：顺序存储结构和链接存储结构。

5. 算法设计技术

现在，设计算法的必要条件都已经具备了，如何设计一个算法来解决一个特定的问题呢？这正是本书讨论的主题。

算法设计技术(algorithm design technique，也称算法设计策略)是设计算法的一般性方法，可用于解决不同计算领域的多种问题。本书讨论的算法设计技术已经被证明是对算法设计非常有用的通用技术，包括蛮力法、分治法、减治法、动态规划法、贪心法、回溯法、分支限界法、概率算法、近似算法等。这些算法设计技术构成了一组强有力的工具，在为新问题(即没有令人满意的已知算法可以解决的问题)设计算法时，可以运用这些技术设计出新的算法。算法设计技术作为问题求解的一般性策略，在解决计算机领域以外的问题时，也能发挥相当大的作用，读者在日后的学习和工作中将会发现学习算法设计技术的好处。

6. 描述算法

在构思和设计了一个算法之后，必须清楚准确地将所设计的求解步骤记录下来，即描述算法。描述算法的常用方法有自然语言、流程图、程序设计语言和伪代码等，其中伪代码是比较合适的描述算法的方法。因为 C ++ 语言的功能强，而且大多数读者都比较熟悉，所以，本书对于算法的描述采用符合 C ++ 语法的伪代码，使得算法的描述简明清晰，既不拘泥于 C ++ 语言的实现细节，又容易转换为 C ++ 程序。

7. 跟踪算法

逻辑错误无法由计算机检测出来，因为计算机只会执行程序，而不会理解动机。经验和研究都表明，发现算法(或程序)中的逻辑错误的重要方法就是系统地跟踪算法。跟踪必须要用“心和手”来进行，跟踪者要像计算机一样，用一组输入值来执行该算法，并且这组输入值要最大可能地暴露算法中的错误。即使有几十年经验的高级软件工程师，也经常利用此方法查找算法中的逻辑错误。

8. 分析算法的效率

算法有两种效率：时间效率和空间效率，时间效率显示了算法运行得有多快，空间效率则显示了算法需要多少额外的存储空间，相比而言，我们更关注算法的时间效率。事实上，计算机的所有应用问题，包括计算机自身的发展，都是围绕着“时间——速度”这样一个中心进行的。一般来说，一个好的算法首先应该是比同类算法的时间效率高，算法的时间效率用时间复杂性来度量。

9. 根据算法编写代码

现代计算机技术还不能将伪代码形式的算法直接“输入”进计算机中，而需要把算法转变为特定程序设计语言编写的程序，算法中的一条指令可能对应实际程序中的多条指令。在把算法转变为程序的过程中，虽然现代编译器提供了代码优化功能，但仍需要一些

标准的技巧，比如，在循环之外计算循环中的不变式、合并公共子表达式、用开销低的操作代替开销高的操作等。一般来说，这样的优化对算法速度的影响是一个常数因子，可能会使程序提高10%～50%的速度。

算法设计的一般过程如图1.3所示。需要强调的是，一个好算法是反复努力和重新修正的结果，所以，即使足够幸运地得到了一个貌似完美的算法，也应该尝试着改进它。那么，什么时候应该停止这种改进呢？设计算法是一种工程行为，需要在资源有限的情况下，在互斥的目标之间进行权衡。设计者的时间显然也是一种资源，在实际应用中，常常是项目进度表迫使我们停止改进算法。

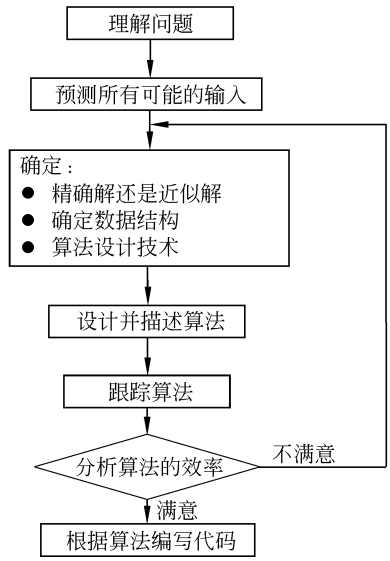

图1.3 算法设计的一般过程

1.1.5 重要的问题类型

就像生物学把自然界的所有生物作为自己的研究对象，计算机科学把问题作为自己的研究对象，研究如何用计算机来解决人类所面临的各种问题。在计算领域的无数问题中，或者由于问题本身具有一些重要特征，或者由于问题具有实用上的重要性，有一些领域的问题是算法研究人员特殊关注的。经验证明，无论对于学习算法还是应用算法，对这些问题的研究都是极其重要的。在本书中，我们将围绕下述问题展开对算法设计技术的讨论。

1. 查找问题

查找是在一个数据集合中查找满足给定条件的记录。对于查找问题来说，没有一种算法对于任何情况都是合适的。有的算法查找速度比其他算法快，但却需要较多的存储空间（例如Hash查找），有的算法查找速度非常快，但仅适用于有序数组（例如折半查找），如此，等等。此外，如果在查找的过程中数据集合可能频繁地发生变化，除了考虑查找操作，还必须考虑在数据集合中执行插入和删除等操作，这种情况下，就必须仔细地设计数据结构和算法，以便在各种操作的需求之间达到一个平衡。而且，组织用于高效查找的特大型数据集合对实际应用具有非常重要的意义。

2. 排序问题

简单地说，排序就是将一个记录的无序序列调整成为一个有序序列的过程。在对记录进行排序时，需要选定一个信息作为排序的依据，例如，可以按学生姓名对学生记录进行排序，这个特别选定的信息称为关键码。

排序的主要目的是为了进行快速查找，这就是为什么字典、电话簿和班级名册都是排好序的。出于同样的考虑，在其他领域的很多重要算法中，排序也被作为一个辅助步骤，例如，搜索引擎将搜索到的结果按相关程度排序后显示给用户。

迄今为止，已经发明的排序算法不下几十种，没有一种排序算法在任何情况下都是最好的解决方案，有些排序算法比较简单，但速度相对较慢；有些排序算法速度较快，但却很复杂；有些排序算法适合随机排列的输入；有些排序算法更适合基本有序的初始排列；有些排序算法仅适合存储在内存中的序列，有些排序算法可以用来对存储在磁盘上的大型文件排序，等等。

3. 图问题

算法中最古老也最令人感兴趣的领域是图问题，很多纷乱复杂的现实问题抽象出的模型都是图结构，例如，可以利用图研究化学领域的分子结构，解决高校排课问题，解决任务分配问题和车间调度问题，等等。

概 念 回 顾

图(graph)通常表示为 $G=(V,E)$，其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中顶点之间边的集合。若顶点 v_i 和 v_j 之间的边没有方向，则称这条边为无向边，用无序偶对(v_i,v_j)来表示；若从顶点 v_i 到 v_j 的边有方向，则称这条边为有向边(也称为弧)，用有序偶对$<v_i, v_j>$来表示。如果图的任意两个顶点之间的边都是无向边，则称该图为无向图，否则称该图为有向图。

有些图问题在计算上是非常困难的，这意味着，在能够接受的时间内，即使用最快的计算机，也只能解决这种问题的一个很小的实例，例如 TSP 问题。图问题中还有一个奇怪的现象：许多形式上非常类似的问题，解决它们的难度却相差很大。例如，在一个有向图中找出两点之间的最短路径问题存在多项式时间算法，但是在一个有向图中找出两点之间的最长路径问题至今没有找到一个多项式时间算法。

4. 组合问题

组合问题一般都是最优化问题，即寻找一个组合对象，比如一个排列、一个组合或一个子集，这个对象能够满足特定的约束条件并使得某个目标函数取得极值：价值最大或成本最小。

无论从理论的观点还是实践的观点，组合问题都是计算领域中最难的问题，其原因是：

① 随着问题规模的增大，组合对象的数量增长极快，即使是中等大小的实例，其组合对象的数量也会达到不可思议的数量级，产生组合爆炸；

② 还没有一种已知算法能在可接受的时间内，精确地求解绝大多数这类问题。

5. 几何问题

几何问题处理类似于点、线、面、体等几何对象。几何问题与其他问题的不同之处在于，哪怕是最简单、最初等的几何问题也难以用数字去处理。尽管人类对几何问题的研究从古代起便没有中断过，但是具体到借助计算机来解决几何问题的研究，还只是停留在一

个初级阶段。随着计算机图形图像处理、机器人和断层 X 摄像技术等方面应用的深入，人们对几何算法产生了强烈的兴趣。本书只讨论两个经典的计算几何问题：最近对问题和凸包问题。最近对问题是在给定平面上的 n 个点中，求距离最近的两个点，凸包问题要求找出一个能把给定集合中的所有点都包含在里面的最小凸多边形。

1.2 算法分析

算法分析(algorithm analysis)指的是对算法所需要的两种计算机资源——时间和空间进行估算，所需要的资源越多，该算法的复杂性就越高。不言而喻，对于任何给定的问题，设计出复杂性尽可能低的算法是设计算法时追求的一个重要目标；另一方面，当给定的问题有多种解法时，选择其中复杂性最低者，是选用算法时遵循的一个重要准则。随着计算机硬件性能的提高，一般情况下，算法所需要的额外空间已不是我们需要关注的重点了，但是对算法时间效率的要求仍然是计算机科学不变的主题。本书重点讨论算法**时间复杂性**(time complexity)的分析，对**空间复杂性**(space complexity)的分析是类似的。

1.2.1 渐进符号

算法的复杂性是运行算法所需要的计算机资源的量，这个量应该集中反映算法的效率，而从运行该算法的实际计算机中抽取出来。撇开与计算机软、硬件有关的因素，影响算法时间代价的最主要因素是问题规模。**问题规模**(problem scope)是指输入量的多少，一般来说，它可以从问题描述中得到。例如，对一个具有 n 个整数的数组进行排序，问题规模是 n；对一个 m 行 n 列的矩阵进行转置，则问题规模是 m 和 n。一个显而易见的事实是：几乎所有的算法，对于规模更大的输入都需要运行更长的时间。例如，需要更多时间来对更大的数组排序，更大的矩阵转置需要更长的时间。所以运行算法所需要的时间 T 是问题规模 n 的函数，记作 $T(n)$。

要精确地表示算法的运行时间函数常常是很困难的，即使能够给出，也可能是个相当复杂的函数，函数的求解本身也是相当复杂的。考虑到算法分析的主要目的在于比较求解同一个问题的不同算法的效率，为了客观地反映一个算法的运行时间，可以用算法中基本语句的执行次数来度量算法的工作量。**基本语句**(basic statement)是执行次数与整个算法的执行次数成正比的语句，基本语句对算法运行时间的贡献最大，是算法中最重要的操作。这种衡量效率的方法得出的不是时间量，而是一种增长趋势的度量。换言之，只考察当问题规模充分大时，算法中基本语句的执行次数在渐近意义下的阶，通常使用大 O、大 Ω 和 Θ 等 3 种渐进符号表示。

1. 大 O 符号

定义 1.1 若存在两个正的常数 c 和 n_0，对于任意 $n \geqslant n_0$，都有 $T(n) \leqslant c \times f(n)$，则称 $T(n)=O(f(n))$(或称算法在 $O(f(n))$中)。

大 O 符号用来描述增长率的上限，表示 $T(n)$的增长最多像 $f(n)$增长的那样快，也就是说，当输入规模为 n 时，算法消耗时间的最大值，这个上限的阶越低，结果就越有价值。

大 O 符号的含义如图 1.4 所示，为了说明这个定义，将问题规模 n 扩展为实数。

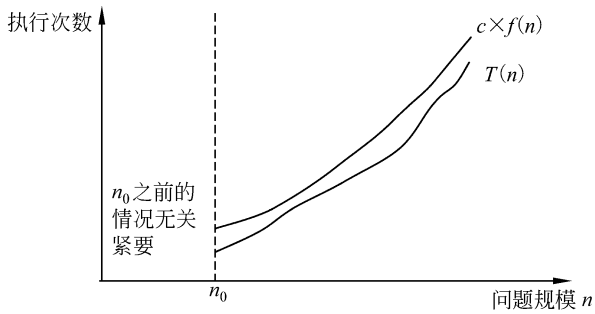

图 1.4　大 O 符号的含义

应该注意的是，定义 1.1 给了很大的自由度来选择常量 c 和 n_0 的特定值，例如，下列推导都是合理的：

$100n+5\leqslant 100n+n$(当 $n\geqslant 5$)$=101n=O(n)$　　($c=101, n_0=5$)

$100n+5\leqslant 100n+5n$(当 $n\geqslant 1$)$=105n=O(n)$　　($c=105, n_0=1$)

2. 大 Ω 符号

定义 1.2　若存在两个正的常数 c 和 n_0，对于任意 $n\geqslant n_0$，都有 $T(n)\geqslant c\times g(n)$，则称 $T(n)=\Omega(g(n))$(或称算法在 $\Omega(g(n))$中)。

大 Ω 符号用来描述增长率的下限，也就是说，当输入规模为 n 时，算法消耗时间的最小值。与大 O 符号对称，这个下限的阶越高，结果就越有价值。

大 Ω 符号的含义如图 1.5 所示。

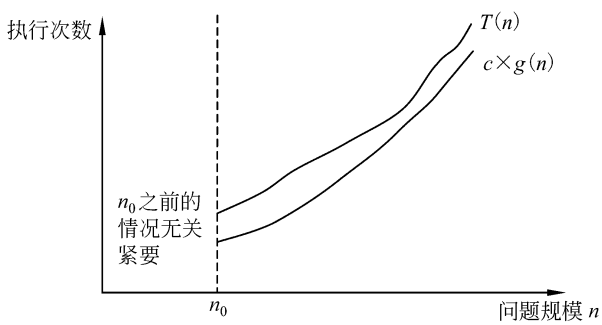

图 1.5　大 Ω 符号的含义

大 Ω 符号常用来分析某个问题或某类算法的时间下界。例如，矩阵乘法问题的时间下界为 $\Omega(n^2)$，是指任何两个 $n\times n$ 矩阵相乘的算法的时间复杂性不会小于 n^2，基于比较的排序算法的时间下界为 $\Omega(n\log_2 n)$，是指无法设计出基于比较的排序算法，其时间复杂性小于 $n\log_2 n$。

大 Ω 符号常常与大 O 符号配合以证明某问题的一个特定算法是该问题的最优算法，或是该问题中的某算法类中的最优算法。

3. Θ 符号

定义 1.3　若存在 3 个正的常数 c_1、c_2 和 n_0，对于任意 $n\geqslant n_0$，都有 $c_1\times f(n)\geqslant T(n)$