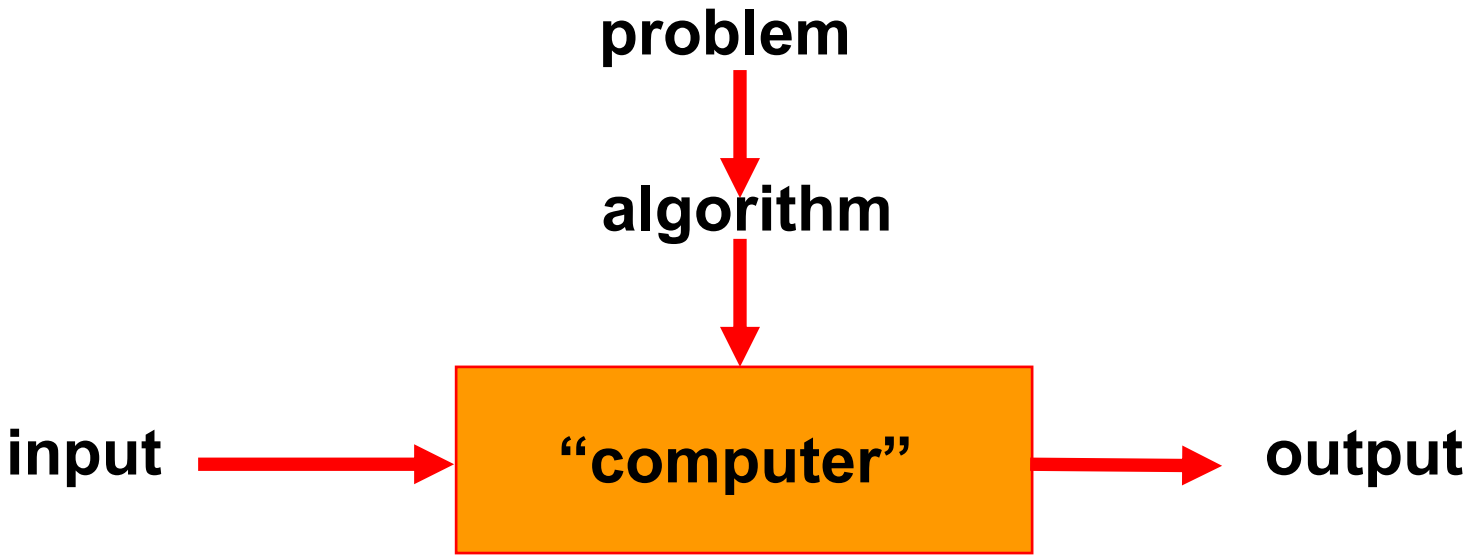
算法基础

深圳大学计算机与软件学院
卢亚辉

- 课程介绍
- 算法的概念
- 算法正确性分析
- 算法时间效率分析
  - 理论分析
  - 经验分析

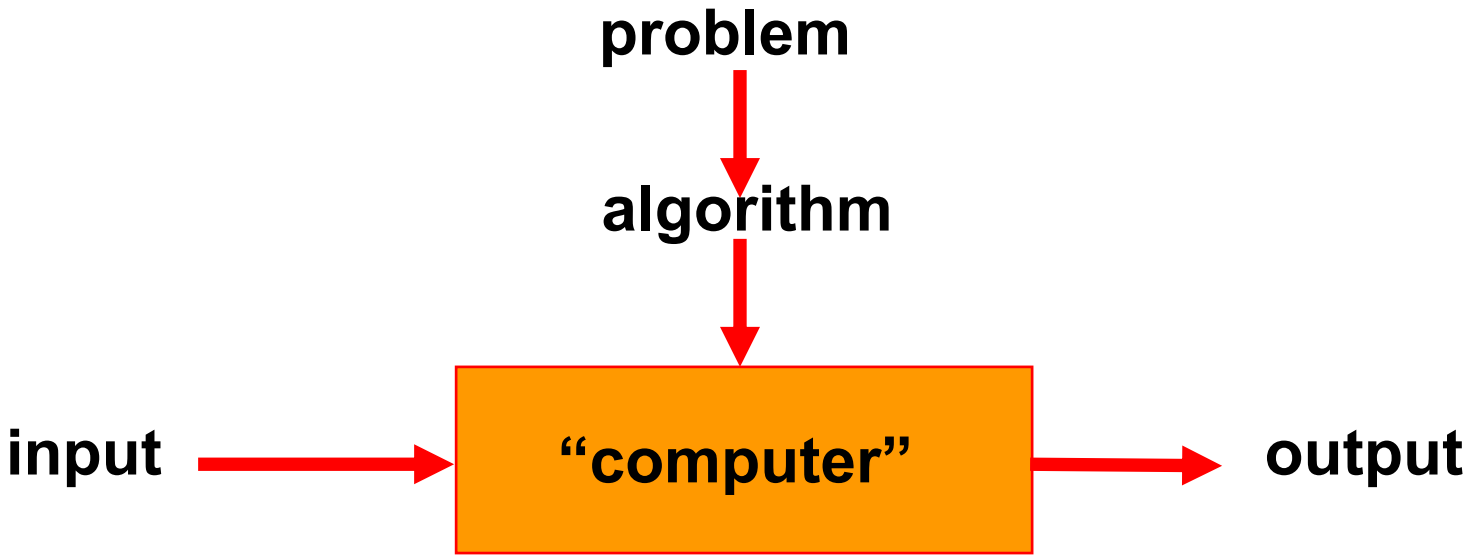
1.1 算法基本概念—什么是算法？

- 算法是一个良定义的计算过程，以一个或一些值作为输入，产生出一个或一组值作为输出。Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- 算法是一个计算步骤的序列，用以将输入转换为输出。An algorithm is thus a sequence of computational steps that transform the input into the output



1.1 算法基本概念—什么是算法？

- 将算法看作一个工具，用于解决良说明的计算问题。问题陈述用概括的词语说明了期望的输入输出关系（specification，规范）。算法描述了特定的计算过程，用于获得该输入输出关系（implimentation，实现）。
- We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship.



例子

- 排序问题的形式化定义（statement，specification）

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- 问题的实例由计算出问题的解所必须的输入组成。an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

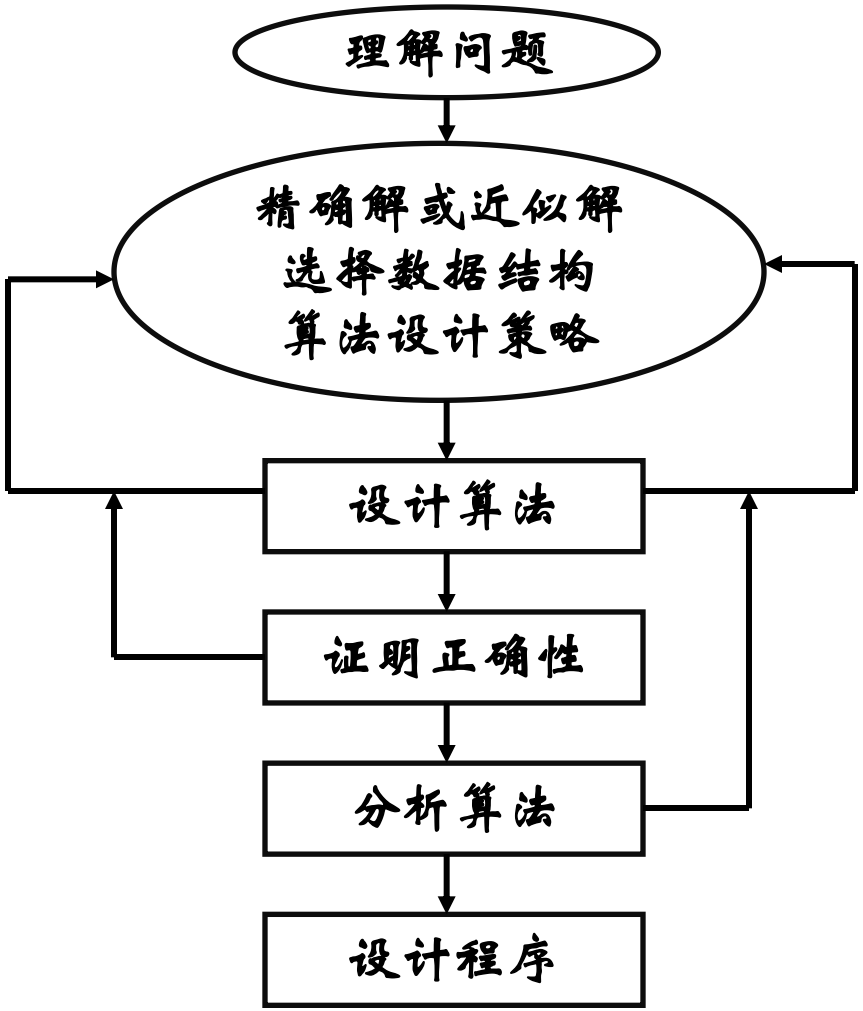
算法的正确性

- 算法是正确的
  - 如果对于每个输入实例，算法都能输出正确的输出并停机
  - 正确的算法解决了给定的计算问题。
- An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem.
- 不正确的算法
  - 对某些输入实例不停机（没计算完，不知道是否能得到正确输出）
  - 停机，给出错误的答案
- An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.
- Contrary to what you might expect, incorrect algorithms can sometimes be useful, if we can control their error rate. We shall see an example of an algorithm with a controllable error rate in Chapter 31 when we study algorithms for finding large prime numbers.
- Ordinarily, however, we shall be concerned only with correct algorithms.

1.1 算法基本概念—算法

- 算法是若干指令的有穷序列，满足性质：
  - 输入：有外部提供的量作为算法的输入。
  - 输出：算法产生至少一个量作为输出。
  - 确定性：组成算法的每条指令是清晰，无歧义的。
  - 有限性：算法中每条指令的执行次数是有限的，执行每条指令的时间也是有限的。
  - 可行性：算法是能够有效解决问题的

算法基本概念—问题求解过程



最大公约数

- 两个不全为0的非负整数m和n的最大公约数gcd(m,n)是能够整除m和n的最大正整数

- 60和24的最大公约数是多少？
- 怎么证明它是最大公约数
- 如何描述这个计算过程？

- 方法1：欧几里得算法
- 方法2：连续整数检测法
- 方法3：质因数法

方法1：欧几里德算法

- 重复应用下列等式，直到m mod n等于0；
- gcd(m ,n)=gcd(n ,m mod n) (m mod n 表于m除以n之后的余数)

算法的表示方法

- An algorithm can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed
- 算法可以说明为英语，计算机程序，甚至硬件设计。唯一的要求是这些说明必须提供一个对计算过程的精确描述。

欧几里德算法原始描述

命题 *给定两个正整数，求它们的最大公因子。*

设 A 和 C 是两个给定的正整数；求它们的最大公因子。如果 C 整除 A，则 C 就是 C 和 A 的一个公因子，因为它也整除本身。而且事实上显然它是最大的，因为再无比 C 大的数能整除 C 了。

但如果 C 不整除 A，则不断地从 A、C 两数中之大者减去小者，直到得出整除前一数的某个数。这种情况最终一定要发生，因为如果得到的是 1，则它就整除前一个数。

现在设 E 是 A 除以 C 的正余数；设 F 是 C 除以 E 的正余数，而且设 F 是 E 的一个因子。由于 F 整除 E 和 E 整除 $C-F$，故 F 也整除 $C-F$；但它也整除本身，所以它整除 C。而且 C 整除 $A-E$；因此 F 也整除 $A-E$，但它也整除 E；因此它整除 A。于是它是 A 和 C 的一个公因子。

现在论证它也是最大的。如果 F 不是 A 和 C 最大的公因子，则将有某个更大的数整除它们。设这样一个数是 G。

现在由于 G 整除 C 而 C 整除 $A-E$，故 G 整除 $A-E$，但也整除整个 A，所以它整除余数 E，但 E 整除 $C-F$；因此 G 也整除 $C-F$，但 G 也整除整个 C，所以它整除余数 F；即，一个较大的数整除一个较小的数，这是不可能的。

因此，没有大于 F 的数能整除 A 和 C，所以 F 是它们的最大公因子。

推论 这一论证使得下列结论成为显然：整除两个数的任何数必整除它们的最大公因子。

证毕。

算法的表示方法：自然语言

(1) 自然语言

优点：容易理解

缺点：冗长、二义性

使用方法：粗线条描述算法思想

注意事项：避免写成自然段

欧几里德算法

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。

算法的表示方法：流程图

(2) 流程图

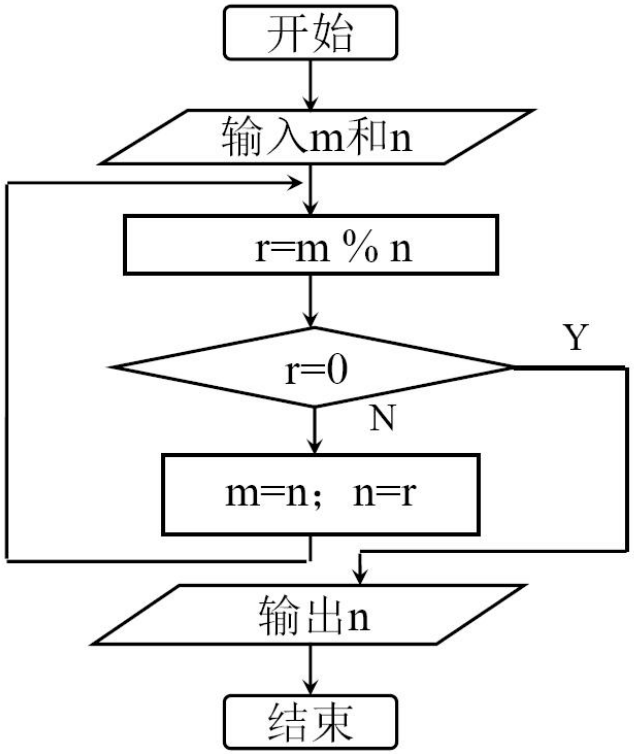
优点：流程直观

缺点：缺少严密性、灵活性

使用方法：描述简单算法

注意事项：注意抽象层次

欧几里德算法



算法的表示方法：程序设计语言

(3) 程序设计语言

优点：能由计算机执行

缺点：抽象性差，对语言要求高

使用方法：算法需要验证

注意事项：将算法写成子函数

欧几里德算法

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

算法的表示方法：伪代码

伪代码（Pseudocode）：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。

优点：表达能力强，抽象性强，容易理解

欧几里德算法

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

方法2：连续整数检测算法

- 第一步：将min{m,n}的值赋给t。
- 第二步：m除以t，如余数为0。进入第三步；否则，进入第四步。
- 第三步：n除以t，如果余数为0，返回t的值作为结果；否则，进入第四步。
- 第四步：把t的值减1。返回第二步。

- 思考：
  - 为什么t的值要每次减小而不是增加？
  - 这个算法有什么bug？

方法3：质因数法

- **第一步**：找到m的所有质因数。
- **第二步**：找到n的所有质因数。
- **第三步**：从第一步和第二步求得的质因数分解式中找出所有的公因数
  - （如果p是一个公因数，而且在m和n的质因数分解式分别出现过pm 和pn次，那么应该将p重复min{pm, pn}次 ）。
- **第四步**：将第三步中找到质因数相乘，其结果作为给定数字的最大公约数。

- 思考：如何实现找到m的所有质因数?

$$60 = 2 \times 2 \times 3 \times 5$$

$$24 = 2 \times 2 \times 2 \times 3$$

$$\gcd(60, 24) = 2 \times 2 \times 3 = 12$$

问题和思考

- 这三个算法
  - 是否都能够找到正确的最大公因数？
  - 每一步是否都是确定无歧义的？
  - 每一步是否都是可行的？
  - 算法是否能在有限步内结束？

- 上述三种算法哪个理解起来比较容易？
- 上述三种算法哪个执行起来比较快？

算法解决哪种问题

- 人类基因工程：DNA的10万个基因，30亿个化学基对的序列（生物信息学）
- 互联网：寻找路由（24章）、搜索引擎查询（11章，32章）
- 电子商务：
  - 使得货物与服务能够以电子的方式洽谈与交换 Electronic commerce enables goods and services to be negotiated and ex_x0002_changed electronically
  - 公钥加密和数字签名（31章）
- 制造业和商业：
  - 分配稀有资源以最有利的方式allocate scarce resources in the most beneficial way（29章）

数据结构

- 数据结构是一种存储和组织数据的方式，以便于访问（读）和修改（写）。
- 没有一种单一的数据结构对所有用途都有效，因此有必要知道几种数据结构的优势和局限。
- data structure is a way to store and organize data in order to facilitate access and modifications. No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.
- 第三部分，数据结构
- 第五部分，高级数据结构
- 着重于对数据结构的效率分析

Algorithms as a technology

- 效率
  - Different algorithms devised to solve the same problem often differ dramatically in their efficiency.
- Algorithms and other technologies
- Having a solid base of algorithmic knowledge and technique is one characteristic that separates the truly skilled programmers from the novices. With modern computing technology, you can accomplish some tasks without knowing much about algorithms, but with a good background in algorithms, you can do much, much more.

注记

本章注记

有许多全面介绍算法这一主题的书都是非常不错的，如 Aho、Hopcroft 和 Ullman[5，6]，Baase 和 Van Gelder[26]，Brassard 和 Bratley[46，47]，Goodrich 和 Tamassia[128]，Horowitz，Sahni 和 Rajasekaran[158]，Kingston[179]，Knuth[182，183，185]，Kozen[193]，Manber[210]，Mehlhorn[217，218，219]，Purdom 和 Brown[252]，Reingold，Nievergelt 和 Deo[257]，Sedgewick[269]，Skiena[280]，以及 Wilf[315]等著作。Bentley[39，40]和 Gonnet[126]对算法设计中一些更具体实际的问题进行了讨论。对算法这一领域的综述可以参见《Handbook of Theoretical Computer Science，Volume A》[302]，以及《CRC Handbook on Algorithms and Theory of Computation》[24]。Gusfield[136]、Pevzner[240]、Setubal 和 Medinas[272]、Waterman[309]等教材则对计算生物学中用到的各种算法做了概述性的介绍。

- 课程介绍
- 算法的概念
- 算法正确性分析
- 算法时间效率分析
  - 理论分析
  - 经验分析

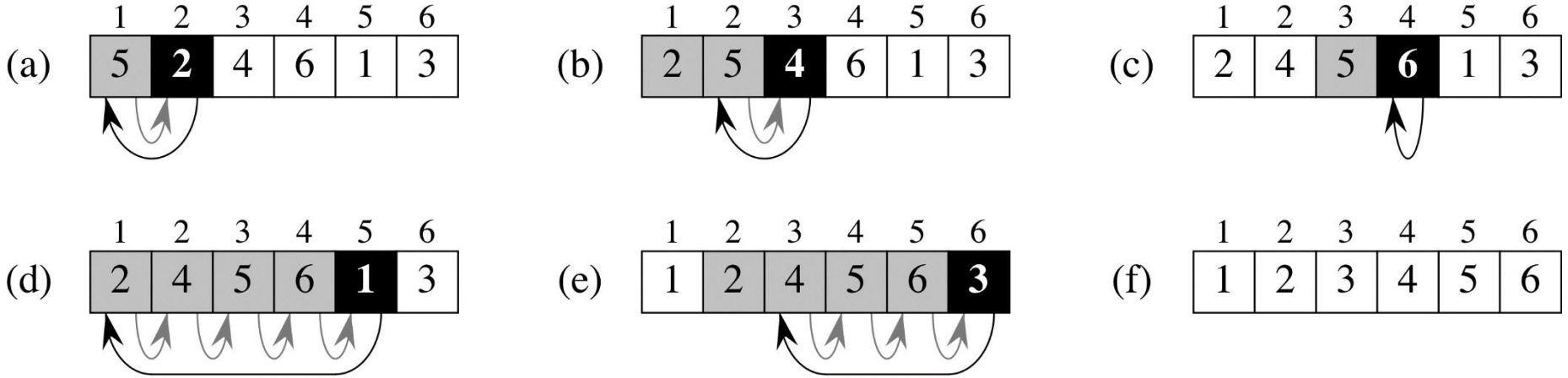
插入排序 (1)

排序问题

输入： n个数的一个序列 $\langle a_1, a_2, \ldots, a_n \rangle$.

输出： 输入序列的一个排列$\langle a'_1, a'_2, \ldots, a'_n \rangle$，满足$a'_1 \le a'_2 \le \cdots \le a'_n$

视频：舞动的排序



插入排序 (2)

算法伪代码

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 . . j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

伪代码的约定（Pseudocode conventions）

1）书写上的“缩进”表示程序中的分程序（程序块）结构。例如，从第 1 行开始的 **for** 循环的体包括了第 2～8 行，从第 5 行开始的 **while** 循环的体包括第 6～7 行。这种缩进风格也适用于 **if-then-else** 语句。用缩进取代传统的 **begin** 和 **end** 语句来表示程序的块结构，可以大大提高代码的清晰性。

2）**while**，**for**，**repeat** 等循环结构和 **if**，**then**，**else** 条件结构与 Pascal 中相同。然而，对 **for** 循环来说有一点小小的不同：在 Pascal 中，循环计数器变量在退出循环时是未定义的，但在本书中，在退出循环后，循环计数器的值仍然保持。于是，紧接着一个 **for** 循环之后，循环计数器的值就是第一个超出 **for** 循环终值的那个数字。我们在对插入排序的正确性证明中，也用到了这一性质。在第 1 行中，**for** 循环的头为 **for** $j \leftarrow 2$ **to** $length[A]$，因此，当此循环结束时，$j = length[A]+1$（或者可以说，$j=n+1$，因为 $n=length[A]$）.

3）符号“▷”表示后面部分是个注释。

4）多重赋值 $i \leftarrow j \leftarrow e$ 是将表达式 e 的值赋给变量 i 和 j；等价于赋值 $j \leftarrow e$，再进行赋值 $i \leftarrow j$。

5）变量（如 i，j 和 key 等）是局部于给定过程的。在没有显式说明的情况下，我们不使用全局变量。

伪代码的约定

6)数组元素是通过“数组名[下标]”这样的形式来访问的。例如，$A[i]$表示数组 A 的第 i 个元素。符号“..”用来表示数组中的一个取值范围，例如，$A[1..j]$就表示 A 的一个子数组，它包含了 j 个元素 $A[1]$，$A[2]$，…，$A[j]$。

7)复合数据一般组织成对象，它们是由属性(attribute)或域(field)所组成的。域的访问是由域名后跟由方括号括住的对象名形式来表示。例如，数组可以被看作是一个对象，其属性有 *length*，表示数组中元素的个数，如 *length*$[A]$就表示数组 A 中的元素个数。在表示数组元素和对象属性时，都要用到方括号，一般来说，通过上下文就可以看出其含义。

用于表示一个数组或对象的变量被看作是指向表示数组或对象的数据的一个指针。对于某个对象 x 的所有域 f，赋值 $y \leftarrow x$ 就使得 $f[y]=f[x]$。更进一步，如果有 $f[x]\leftarrow 3$，则不仅有 $f[x]=3$，同时 $f[y]=3$。换言之，在赋值 $y\leftarrow x$ 后，x 和 y 指向同一个对象。

有时，一个指针不指向任何对象。这时，我们赋给它 NIL。

8)参数采用按值传递方式：被调用的过程会收到参数的一份副本。如果它对某个参数赋值的话，主调过程是看不见这一变动的。当对象被传递时，实际传递的是一个指向该对象数据的指针，而对象的各个域则不被拷贝。例如，如果 x 是某个被调用过程的参数，在被调过程中的赋值 $x\leftarrow y$ 对主调过程来说是不可见的。但是，赋值 $f[x]\leftarrow 3$ 却是可见的。

9)布尔运算符“and”和“or”都具有短路能力。亦即，当我们求表达式“x and y”的值时，首先计算 x 的值。如果 x 的值为 FALSE，那么整个表达式的值就不可能为 TRUE 了，因而就无需再对 y 求值了。但是，如果 x 的值为 TRUE 的话，就必须进一步计算出 y 的值，才能确定整个表达式的值。类似地，在计算表达式“x or y”的值时，仅当 x 的值为 FALSE 时，才需要计算子表达式 y 的值。短路运算符允许我们写出如“$x\neq$NIL and $f[x]=y$”这样的布尔表达式，而不用担心当我们试图在 x 为 NIL 时计算 $f[x]$，会发生怎样的情况。

正确性证明

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j − 1].
      i = j − 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i − 1
      A[i + 1] = key
```

循环不变式：子数组A[1..j-1]包含了最初位于A[1..j-1]，目前已经排好序的各个元素

循环不变式主要用来帮助我们理解算法的正确性。对于循环不变式，必须证明它的三个性质：

初始化：它在循环的第一轮迭代开始之前，应该是正确的。

保持：如果在循环的某一次迭代开始之前它是正确的，那么，在下一次迭代开始之前，它也应该保持正确。

终止：当循环结束时，不变式给了我们一个有用的性质，它有助于表明算法是正确的。

正确性证明

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

初始化：首先，先证明在第一轮迭代开始之前，循环不变式是成立的。此时，$j=2$，[⊖]而子数组为 $A[1..j-1]$。亦即，它只包含一个元素 $A[1]$，实际上就是最初在 $A[1]$中的那个元素。这个子数组是已排序的(这一点是显然的)，这样就证明了循环不变式在循环的第一轮迭代开始之前是成立的。

保持：接下来，我们来考虑第二个性质：证明每一轮循环都能使循环不变式保持成立。从非形式化的意义上来看，在外层 **for** 循环的循环体中，要将 $A[j-1]$、$A[j-2]$、$A[j-3]$等元素向右移一个位置，直到找到 $A[j]$的适当位置时为止(第 4～7 行)，这时将 $A[j]$的值插入(第 8 行)。如果要更形式化地证明第二个性质成立的话，就需要陈述并证明对内层 **while** 循环也有一个循环不变式成立。但是，此处我们更倾向于暂时不陷入过于形式化的细节之中，而是依赖于非形式化的分析，来证明第二个性质对于外层循环是成立的。

终止：最后，分析一下循环结束时的情况。对插入排序来说，当 j 大于 n 时(即当 $j=n+1$ 时)，外层 **for** 循环结束。在循环不变式中，将 j 替换为 $n+1$，就有子数组 $A[1..n]$包含了原先 $A[1..n]$中的元素，但现在已排好序了。但是，子数组 $A[1..n]$其实就是整个数组！因此，整个数组就排好序了，这意味着算法是正确的。

- 课程介绍
- 算法的概念
- 算法正确性分析
- 算法时间效率分析
  - 理论分析
  - 经验分析

算法分析Analyzing algorithms

- 算法分析是指预测算法所需资源。Analyzing an algorithm has come to mean predicting the resources that the algorithm requires.
- 主要是计算时间。Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure.
- Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one. Such analysis may indicate more than one viable candidate, but we can often discard several inferior algorithms in the process.

RAM模型(Random-Access Machine)

假定一种单处理器计算模型—随机访问模型：

- 指令一条接一条执行，没有并发操作。
- 指令系统
  - 算术指令：加法、减法、乘法、除法、取余，向下取整、向上取整
  - 数据移动指令：装入、存储、复制
  - 控制指令：条件与无条件转移、子程序调用与返回。
  - 假设每条指令所需时间均为常量。不同指令的运行时间不同
  - 指令要具有现实性，比如，不能定义一条排序指令
- RAM模型中的数据类型有整数型和浮点实数型。一般不关心精度问题
- 指数运算是否为常量时间的指令？
  - 一般不是，但是左移指令是常量时间计算2^k。
  - 当k是一个足够小的正整数时，2^k当做一个常量时间的计算
- 不考虑内存层次的影响。

输入规模的度量

- **算法的运行时间严重依赖于问题的输入规模。一般地，算法的运行时间与输入规模同步增长。**
  - 例如，需要更多时间来对更长的数组排序，更大的矩阵相乘也需要花费更多时间，等等。
- **即使规模相同的两个不同输入，其运行时间也可能差别很大**

  例如：当插入排序在处理已排序好的*n*个元素数组和逆序排列的*n*个元素数组时。

仔细定义术语“运行时间running time”和“输入规模size of input”

输入规模

- 输入规模的的最佳notion依赖于于研究的问题。
- The best notion for input size depends on the problem being studied.
- 以n度量。For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input
- 以n的位数来度量。For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.
- 以m，n来度量。Sometimes, it is more appropriate to describe the size of the input with two numbers rather than one. For instance, if the input to an algorithm is a graph, the input size can be described by the numbers of vertices and edges in the graph.

运行时间的度量单位

- 为什么不用S、ms：
  - 依赖于特定计算机运行速度
  - 依赖于算法程序实现的质量
  - 依赖于适用那种编译器将程序转化成机器码
- 运行时间用基本操作执行的数量（执行步数）
- The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed.
- It is convenient to define the notion of step so that it is as machine-independent as possible.
- 执行每行伪代码需要常数时间。For the moment, let us adopt the following view. A constant amount of time is required to execute each line of our pseudocode. One line may take a different amount of time than another line, but we shall assume that each execution of the ith line takes time ci, where ci is a constant.
- This viewpoint is in keeping with the RAM model, and it also reflects how the pseudocode would be implemented on most actual computers.

分析插入排序算法

INSERTION-SORT(A, n)	*cost*	*times*
for $j = 2$ **to** n	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
$i = j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

- 假定每次执行第*i*行所花的时间都是常量c_i;
- 对 $j = 2, 3, \ldots n$, 假设t_j表示对那个值*j*执行*while*循环测试的次数。
- 当一个*for*或*while*循环按通常的方式（由于循环头中的测试）退出时，执行测试的次数比执行循环体的次数多1。

分析插入排序算法

- 插入排序的运行时间:

$$T(n) = c_1 * n + c_2 * (n-1) + c_4 * (n-1) + c_5 * \sum_{j=2}^{n} t_j + c_6 * \sum_{j=2}^{n} (t_j - 1) + c_7 * \sum_{j=2}^{n} (t_j - 1) + c_8 * (n-1)$$

  - 运行时间取决于t_j的值，其随输入问题的情况而定.

最好情况： 数组已排好序

- 当$i = j - 1$时，$A[i] \le key$；所有的t_j均为1.
- 运行时间

$$T(n) = c_1 * n + c_2 * (n-1) + c_4 * (n-1) + c_5 * (n-1) + c_8 * (n-1)$$

$$= (c_1 + c_2 + c_4 + c_5 + c_8) * n - (c_2 + c_4 + c_5 + c_8)$$

- $T(n)$可表示为$an + b$，其中常量a和b依赖于语句代价c_i

线性函数linear function

分析插入排序算法

最坏情况：数组已逆序排好

- 在整个循环测试中，总是$A[i] > key$。
- 必须将每个元素$A[j]$与整个已排序子数组$A[1..j]$中的每个元素进行比较。➔ $t_j = j$.

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n - (c_2 + c_4 + c_5 + c_8)$$

二次函数quadratic function

- $T(n)$表示为$an^2 + bn + c$，其中常量a, b和c 依赖于语句代价c_i

最坏情况与平均情况分析

- 通常关心最坏情况运行时间：
  - 算法的最坏情况运行时间给出了任何输入的运行时间的一个上界
    - The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer.
  - 对某些算法，最坏情况经常出现。
    - For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database.
    - In some applications, searches for absent information may be frequent.
  - 平均情况的时间复杂度往往与最坏情况一样。
    - 例子: 对插入排序而言，平均情况运行时间仍是n的二次函数

最坏情况与平均情况分析

- 平均情况分析：
  - 通常假定一个特定规模下的所有输入的“平均性”都是一样的
  - 也可以采用随机算法强制达到平均。

The scope of average-case analysis is limited, because it may not be apparent what constitutes an “average” input for a particular problem.

Often, we shall assume that all inputs of a given size are equally likely. In practice, this assumption may be violated, but we can sometimes use a randomized algorithm, which makes random choices, to allow a probabilistic analysis and yield an expected running time.

We explore randomized algorithms more in Chapter 5 and in several other subsequent chapters.

1.2插入排序算法平均情况分析（7）

- t_j理解为随机变量

$$E(t_j)=\frac{1}{j}(1+2+\cdots+j)=\frac{j+1}{2}\approx\frac{j}{2}$$

- 参看最坏情况下$t_j = j$，$T(n)$表示为$an^2 + bn + c$，则平均情况下的运行时间仍是n的二次函数

算法复杂性

- 算法复杂性 = 算法所需要的计算机资源；
- 算法的时间复杂性$T(n)$，其中n是问题的输入规模。
  - 最坏情况下的时间复杂性

    $T_{\max}(n) = \max\{ T(\mathrm{I}) \mid \mathrm{size(I)}=n \}$
  - 最好情况下的时间复杂性

    $T_{\min}(n) = \min\{ T(\mathrm{I}) \mid \mathrm{size(I)}=n \}$
  - 平均情况下的时间复杂性

    $T_{\mathrm{avg}}(n) = \sum_{size(I)=n} p(I)T(I)$

    其中I是问题的规模为n的实例，$p(I)$是实例I出现的概率
- 算法的空间复杂性$S(n)$；

增长量级 order of growth

- 简化：it is the rate of growth, or order of growth, of the running time that really interests us.

$$T(n)=\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n$$
$$-(c_2+c_4+c_5+c_8)$$

  - ignored the actual cost of each statement, using the constants ci to represent these costs.
  - an^2 + bn+c: expressed the worst-case running time as an^2 + bn+c for some constants a, b, and c that depend on the statement costs ci.
  - an^2: We therefore consider only the leading term of a formula , since the lower-order terms are relatively insignificant for large values of n.
  - n^2: We also ignore the leading term's constant coefficient, since constant factors are less significant than the rate of growth in determining computational efficiency for large inputs.
  - P vs NP

- 课程介绍
- 算法的概念
- 算法正确性分析
- 算法时间效率分析
  - 理论分析
  - 经验分析

2.6 算法的经验分析

对算法效率做经验分析的通用方案

1. 了解实验的目的。
2. 决定用来度量效率的量度 M 和度量单位（单位时间内的操作次数）。
3. 决定输入样本的特性（它的范围、大小等等）。
4. 为实验准备算法（或者若干算法）的程序实现。
5. 生成输入样本。
6. 对输入样本运行算法（或者若干算法），并记录观察到的实验数据。
7. 分析获得的实验数据。

度量单位

在算法的程序实现中插入一些计数器（或者若干计数器）来对算法执行的基本操作次数进行计数。

第二种方法是记录待讨论算法的程序实现的运行时间。最简单的一种做法是利用系统命令，就像 UNIX 中的 time 命令一样。另一种测量程序段运行时间的做法是，在程序段的刚开始处（t_{start}）和才结束时（t_{finish}）查询系统的时间，然后计算这两个时间的差（$t_{finish} - t_{start}$）[1]。在 C 和 C++中，我们可以用 clock 函数来达到这个目的；在 Java 中，System 类的 currentTime*M*illis()方法提供了这个功能。

然而，我们很有必要把这样一些事实记在脑中。第一，一般来说，系统时间并不是十分精确的，对相同程序的相同输入重复运行多次，我们可能会得到有轻微差异的统计结果。一个明显的补救办法是进行多次这样的度量，然后取它们的平均值（或取中值）作为该样本的观察值。第二，由于现代计算机的速度很快，程序的运行时间可能被报告为 0，使得记录会完全失败。解决这种困境的标准做法是用一个特定的循环多次运行这段程序，度量总运行时间，然后除以循环的重复次数。第三，在一个运行分时系统（就像 UNIX）的计算机上，所报告的时间很可能包含了 CPU 运行其他程序的时间，很明显这完全有悖于实验的初衷。所以我们应该引起注意，并要求系统提供专门用于运行我们的程序的时间（在 UNIX 中，这个时间称为“用户时间”，time命令中就提供了这个功能）。

输入样本

样本。一般来说，我们的目标是用一个样本来代表一类“典型”的输入；所以，我们面临的挑战是理解什么输入是“典型”输入。对于有些类型的算法，比如本书后面将会讨论的旅行商问题的算法，研究人员制定了一系列输入实例用来作为测试的基准。但更常见的情况是，输入样本必须由实验者来确定。一般来说，我们必须作几方面的决定：样本的规模（一种比较明智的做法是，先从一个相对较小的样本开始，以后如有必要再加大），输入样本的范围（一般来说，既不要小得没有意义，也不要过分大），以及一个在所选择范围内产生输入的程序。就最后一个方面来说，输入的规模既可以符合一种模式（例如，1 000, 2 000, 3 000,...,10 000 或者 500, 1 000, 2 000, 4 000,...,128 000）也可以随机产生（例如，在最大值和最小值之间均匀分布）。

对于实验样本的规模，我们需要重点考虑的另一个因素是，是否需要包括同样规模样本的多个不同实例。如果我们预测，对于相同规模的不同实例，我们观测到的度量值会有相当的不同，那么，让样本中的每一种规模都包含多个实例是比较明智的（统计学中有许

随机数和伪随机数

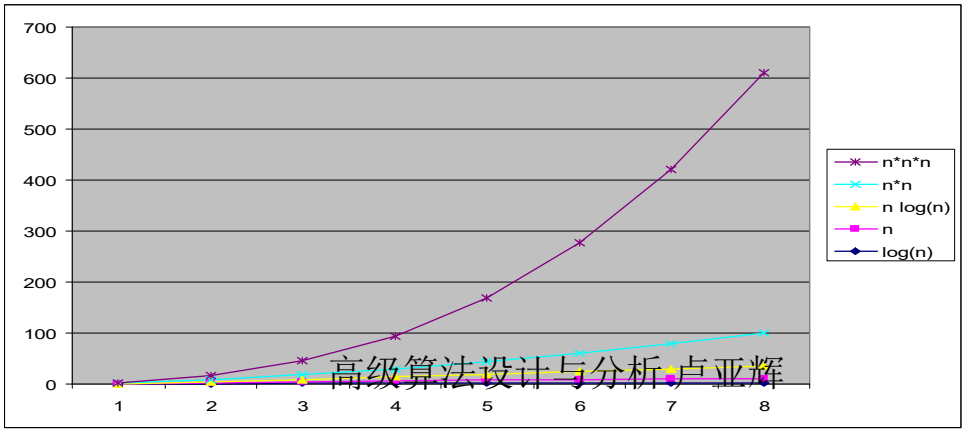
对于算法效率的经验分析来说，大多数情况下都需要产生一些随机数。即使我们决定对输入规模应用一种模式，我们仍然希望输入的实例会自动随机产生。目前，在一台数字式的计算机上产生随机数还是一个难题，因为原理上，这个问题只能近似解决。这就是为什么计算机科学家倾向与把这种数称为伪随机数。从实用的角度看，获取这种数的最简单和最自然的方法是利用计算机语言的函数库提供的随机数发生器。典型情况下，它会输出一个均匀分布在0和1区间中的（伪）随机变量的值。如果需要一个另外一种随机变量，我们应该做一个相应的变换。例如，x是一个均匀分布在区间$0 \le x < 1$上的连续随机变量，变量$y = l + \lfloor x(r-l) \rfloor$就会均匀地分布在$l$和$r-l$间的整数上，其中$l$和$r$是两个整数（$l < r$）。

数据的呈现方式

作为实验结果的经验数据需要记录下来，然后拿来做分析。数据可以用表格或者称为散点图的图形呈现，散点图就是在笛卡儿坐标系中用点将数据标出。任何时候只要可行，都应该同时使用这两种方法，因为这两种方法各有自己的利弊。

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

Table 2.1 Values (some approximate) of several functions important for analysis of algorithms



本章注记

1968 年，Knuth 发表了三卷题为《计算机程序设计艺术》(The Art of Computer Programming)[182，183，185]的著作中的第一卷。第一卷开创了现代计算机算法研究，并侧重于对算法运行时间的分析。对于本书中涉及的许多主题，这三卷著作都始终是有吸引力的和有价值的参考书。根据 Knuth 的说法，“算法”(algorithm)一词源自名字“al-Khowârizmî”，这是一位 9 世纪的波斯数学家。

Aho，Hopcroft 和 Ullman[5]主张将算法的渐近分析作为比较不同算法间相对性能的一种方法。他们还使利用递归关系来刻划递归算法运行时间的做法变得流行起来。

Knuth[185]以百科全书似的方式，分析了多种排序算法。在他对各种排序算法的比较(16.2 节)中，包括了精确的、比较执行步数这样的分析，类似我们这儿对插入排序所做的分析一样。Knuth 对插入排序的讨论包括了这一算法的几种变形，其中最重要的是 Shell 排序算法，由 D. L. Shell 提出，它对输入序列的周期性子序列采用插入排序，从而可以得到一种更快的排序算法。

Knuth 也对合并排序算法进行了分析。他提到了在 1938 年，有人发明了一种机械式校对机，它在一趟之内，就可以将两堆穿孔卡片合并成一堆。J. von Neumann 是计算机科学的先驱之一，他于 1945 年在 EDVAC 机上，特意为合并排序编写了一个程序。

Gries[133]介绍了程序正确性证明的早期历史，他提到了 P. Naur 在这一领域内的第一篇论文，并指出循环不变式是由 R. W. Floyd 提出的。Mitchell[222]编写的教材中介绍了程序正确性证明方面一些更新的进展。

练习

- P8 1-1