湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课程名称	算法设计与分析
章节、专题	算法分析技术
教学目标及基本要求	掌握常用的算法分析技术，能够分析算法或程序的空间复杂性和时间复杂性。
教学重点	时间复杂性分析，渐近复杂性估计
教学难点	渐近复杂性估计
教学内容与时间分配	（1）算法与程序性能概述（0.5课时） （2）算法描述与常用的支持程序（1课时） （3）空间复杂性的组成与分析（0.5课时） （4）时间复杂性的组成与分析（0.8课时） （5）渐近复杂性的估计（1.2课时） 共计4课时。
习　　题	第1.6.1节（基础知识题），第1.6.2节（实验题）

第1章　算法分析技术

本章介绍常用的算法分析技术，包括程序性能、空间复杂性、时间复杂性、渐近符号、递归函数求解等内容。

- 算法与程序性能概述
- 算法描述与常用的支持程序
- 空间复杂性的组成与分析
- 时间复杂性的组成与分析
- 渐近复杂性的估计

1.1 算法与程序性能概述

1.1.1 算法的特性

算法（Algorithm）是对特定问题求解步骤的一种描述，是为解决一个或一类问题给出的一个确定的、有穷的操作序列。通常，算法必须具有下列五个重要特性。

（1）有穷性。一个算法总是对任何合法的输入在执行有穷步后结束，且每一步都能在有穷时间内完成。

（2）确定性。算法中的每一条指令都必须有确切的含义，理解时不会产生歧义。并且，在任何条件下，算法只有唯一确定的执行路径，即相同的输入必定生成相同的输出。

（3）可行性。算法中的操作都可以通过已经实现的基本运算执行有限次实现。

（4）输入。一个算法有零个或多个输入，以刻画运算对象的初始情况。零个输入是指算法本身指定了初始条件。

（5）输出。一个算法有一个或多个输出，以反映对输入数据加工后的结果。

【注】如果操作序列只满足特性（2）~（5），则只能叫做计算过程。如果在特性（2）中执行路径不满足唯一性，则称为非确定性算法。

1.1.2 常用的算法设计策略

常用的算法设计策略有分治方法、贪心方法、动态规划方法、回溯方法、分枝限界方法、概率方法等。后续章节会分章介绍这些算法设计策略。

1.1.3 程序性能

程序性能是指运行一个程序所需的内存大小和时间多少。所以，程序的性能一般指程序的空间复杂性和时间复杂性。性能评估主要包含两方面，即性能分析与性能测量，前者采用分析的方法，后者采用实验的方法。

1. 考虑空间复杂性的理由

- 在多用户系统中，需要指明分配给该程序的内存大小。
- 想预先知道计算机是否有足够的内存来运行该程序。
- 一个问题可能有若干个内存需求不同的解决方案。
- 用空间复杂性估计一个程序可能解决的问题的最大规模。

2. 考虑时间复杂性的理由

- 某些计算机用户需要提供程序运行的时间上限。
- 程序需要提供一个满意的实时反应。

3. 选取方案的原则

- 当解决一个问题有多种可选方案时，方案的选取要基于这些方案之间的性能差异。
- 对于各种方案的时间和空间复杂性，最好采取加权的方式进行评价。

1.1.4 评价算法优劣的基本标准

一个算法的优劣应该从以下几方面来评价。

（1）正确性。在合理的数据输入下，能够在有限的运行时间内得到正确的结果。

（2）可读性。一个好的算法，首先应便于人们理解和相互交流，其次才是机器可执行。可读性强的算法有助于人们对算法的理解，而难懂的算法易于隐藏错误，且难于调试和修改。

（3）健壮性。当输入的数据非法时，好的算法能适当地作出正确反应或进行相应处理，而不会产生一些莫名其妙的输出结果。

（4）高效性。高效性包括时间和空间两个方面。时间高效是指算法设计合理，执行效率高，可以用时间复杂度来度量；空间高效是指算法占用存储容量合理，可以用空间复杂度来度量。时间复杂度和空间复杂度是衡量算法的两个主要指标。

1.2 算法描述方式与常用的支持程序

1.2.1 算法描述方式

描述算法有多种方式，如自然语言方式、图形表格方式和伪代码方式等。本书主要采用C++ 17描述算法，并直接使用C++ STL中的常用算法函数和常用数据结构。为了更好地阐明算法的思路，有时还采用伪代码或自然语言等方式描述算法，其中伪代码使用类似于C/C++的语法。

1.2.2 几个常用的I/O函数

本书后续章节介绍的算法多次用到输出数组或容器中的元素等函数，这里给出这些函数的C++源程序，保存在文件ios.hpp中。

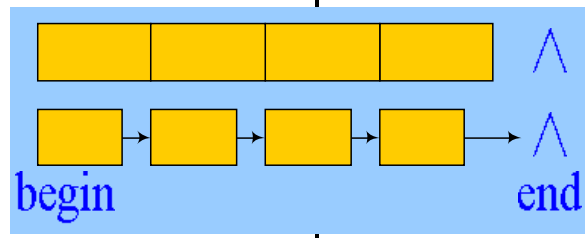
1. 预处理

```
// ios.hpp, C++ 11
#pragma once // 避免重复编译
#include<iostream> // io
#include<sstream> // ostringstream
#include<vector>
#include<set>
```

2. 数组或容器中的元素输出到字符串

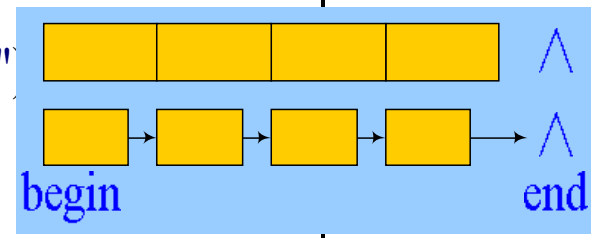
（1）基础函数。位于专门指定的名字空间，用于输出指定范围内的元素，范围通过迭代器指定。

```
// 数组或容器中的元素输出到字符串
namespace ios_aux // 基础函数, 输出指定范围内的元素
{
    template<class It> // 指定范围内的元素
    std::string to_string(It F, It L, const char sp[]= " ")
    {   if(F == L) return ""; // 没有元素
        std::ostringstream out;
        out << *F; // 首元素前不用分隔符
        // 后续元素前加上分隔符
        for(++F; F != L; ++F) out << sp << *F;
        return out.str();
    }
}
```



（2）输出前n个元素。输出数组或容器中的前n个元素。

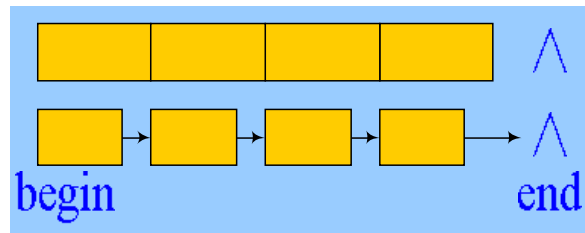
```
namespace std
{
    using ios_aux::to_string; // 输出指定范围内的元素
    template<class T> // 数组的前n个元素
    string to_string(T X[], int n, const char sp[] = " ")
    {   return to_string(X, X + n, sp);
    }
    template<class Cont> // 容器的前n个元素
    string to_string(const Cont &X, int n, const char sp[] = " "
    {   auto F = begin(X), L = next(F, n);
        return to_string(F, L, sp);
    }
```



3. 使用operator<<输出容器的元素

本书后续章节介绍的示例使用的容器对象只有vector对象和set对象，所以这里只给出vector对象和set对象的输出函数，其他容器对象的输出函数可类似实现。

```
// 使用operator<<输出vector和set的全部元素
template<class T>
ostream &operator<<(ostream &out, const vector<T> &X)
{   return out << to_string(begin(X), end(X));
}
template<class T>
ostream &operator<<(ostream &out, const set<T> &X)
{   return out << to_string(begin(X), end(X));
}
}
```



1.2.3 priority_queue的特殊化

堆分为最小堆和最大堆2种，是一种带有优先级的队列。在最小堆中，队首（堆顶）元素一定是最小的元素；在最大堆中，队首（堆顶）元素一定是最大的元素。

在C++ STL中，可用priority_queue表示堆。但是，priority_queue使用起来不是很方便。为了方便后续章节使用，这里对priority_queue进行特殊化，专门定义最小堆和最大堆，并将该定义保存在文件queue.hpp中。

```
// queue.hpp, C++ 11
#pragma once // 避免重复编译
#include <queue> // priority_queue, queue
namespace std // 在名字空间std中定义
{
    // 最小堆, 将STL的priority_queue特殊化, 需要大于运算
#ifndef minheap
    template<class T> // 使用using定义模板别名
    using minheap = priority_queue<T, vector<T>, greater<T>>;
#endif
    // 最大堆, 将STL的priority_queue特殊化, 需要小于运算
#ifndef maxheap
    template<class T> // 使用using定义模板别名
    using maxheap = priority_queue<T, vector<T>, less<T>>;
#endif
    // 保证只需定义<和==就可以在std中使用!=, <=, >, >=
    using namespace rel_ops;
}
```

1.2.4 一些常用的预处理

在使用C++描述算法时，为了避免过于拘泥于细节，这里将一些多处用到的预处理组织成文件Algorithm.h，内容如下。

```
// Algorithm.h
#pragma once
#include <algorithm> // 常用算法, swap, fill, copy, max_element, sort, etc.
#include <numeric> // 数值算法, accumulate, iota, etc.
#include <cmath> // abs, sqrt, log2, isinf, INFINITY, etc.
#include <functional> // function, etc.
#include <vector> // vector
#include <set> // set
#include <stack> // stack
#include <queue> // priority_queue, queue
#include <map> // map
#include "queue.hpp" // minheap, maxheap等, 第1章介绍
#include "ios.hpp" // to_string, io等, 第1章介绍
#include "Matrix.hpp" // 矩阵, 第2章介绍
using namespace std;
namespace std { const auto inf = INFINITY; } // INFINITY缩写为inf
```

1.3 空间复杂性的组成与分析

1.3.1 空间复杂性的组成

程序所需要的空间主要由以下部分构成。

- 指令空间。存储经过编译之后的程序指令。
- 数据空间。存储所有常量和所有变量值所需要的空间。
- 环境栈空间。保存函数调用返回时恢复运行所需要的信息。

1. 指令空间

程序所需指令空间的大小取决于如下因素。

- 把程序编译成机器代码的编译器。所使用的编译器不同，则产生的机器代码长度就会有所差异。
- 编译时实际采用的编译器选项。采用的选项不同，产生的机器代码也会不同。
- 目标计算机。目标计算机的配置也会影响代码的规模。例如，如果计算机具有浮点数处理硬件，那么，每个浮点数操作可以转化为一条机器指令。否则，必须生成仿真的浮点数计算代码，使整个机器代码加长。

2. 数据空间

分成以下两部分。

- 存储常量和简单变量。取决于所用的计算机和编译器，以及变量与常量的数目。
- 存储复合变量。包括数据结构所需的空间及动态分配的空间。

3. 环境栈空间

调用一个函数时，下列数据保存在环境栈中。

- 返回地址。
- 所有局部变量的值、传值形式参数的参数值。
- 所有引用参数的定义。

1.3.2 空间复杂性的分析

1. 实例特征

所谓实例特征是指决定问题规模的那些因素（如输入和输出的数量或相关数的大小）。如对 n 个元素进行排序、$n \times n$ 矩阵的加法等，都可以用 n 作为实例特征；而两个 $m \times n$ 矩阵的加法应该用 n 和 m 作为实例特征。

2. 各组成部分的分析

（1）指令空间。指令空间的大小对于所解决的问题不是很敏感。

（2）常量及简单变量所需空间。常量及简单变量所需空间独立于所解决的问题。

（3）复合变量及动态分配所需空间。

- 定长复合变量及定长动态分配所需空间通常独立于问题的规模。
- 变长复合变量及变长动态分配所需空间通常与问题的规模有关。

（4）环境栈。

- 如果没有使用递归函数，那么环境栈独立于实例特征。
- 在使用递归函数时，实例特征通常会影响环境栈所需空间。
- 递归函数所需的栈空间主要依赖于局部变量及形式参数所需要的空间。
- 递归函数所需的栈空间还依赖于递归的深度（即嵌套递归调用的最大层次）。

3. 程序所需要的空间

综合以上分析，一个程序所需要的空间可分为以下两部分。

（1）固定部分。独立于实例特征，主要包括指令空间、简单变量以及定长复合变量占用的空间、常量占用的空间。

（2）可变部分。主要包括复合变量所需空间、动态分配的空间、递归栈所需要的空间。

- 复合变量所需的空间依赖于所解决的具体问题。
- 动态分配的空间和递归栈所需要的空间依赖于实例特征。

若令$S(P)$表示程序P需要的空间，则有$S(P)=C+S_P$(实例特征)，其中C表示固定部分所需空间，是一个常数，S_P表示可变部分所需空间。

在分析程序的空间复杂性时，一般着重于估算S_P(实例特征)。其中，实例特征的选取通常受相关数的数量以及程序输入和输出规模的制约。

1.3.3 举例

以秦九韶算法为例说明。该算法是南宋数学家秦九韶在《数书九章》中记载的算法，欧洲人称为Horner算法，用于计算多项式 $a_0x^n+a_1x^{n-1}+\cdots+a_{n-1}x+a_n$ 的值。该算法的基本思路是将上述多项式改写为 $\left(\cdots\left(a_0x+a_1\right)x+\cdots+a_{n-1}\right)x+a_n$ 的形式。

1. 累积计算

- 源程序。

```
auto Qin(const double a[], double x, int n)
{   double result = a[0];
    for(int i = 1; i <= n; ++i)
        result = result * x + a[i];
    return result;
}
```

- 实例特征。取次数n作为实例特征。
- 空间复杂性。数组地址a需4字节，参数x需8字节，n需4字节，局部变量i需4字节，result需8字节，共需28字节，独立于n，因此，$S_{\text{Qin}}(n)$=0。

2. 递归计算

- 源程序。

```
auto Horner(double a[], double x, int n)
{   if(n <= 0) return a[0];
    return Horner(a, x, n - 1) * x + a[n];
}
```

- 实例特征。取次数n作为实例特征。
- 空间复杂性。递归栈空间包括参数a（4字节）、x（8字节）、n（4字节）以及返回地址的空间（4字节），每次调用Horner共需20字节。因为嵌套调用一直进行到$n=0$，所以递归深度为$n+1$，需要$20(n+1)$字节的递归栈空间，即$S_{\text{Horner}}(n)=20(n+1)$。

1.4 时间复杂性的组成与分析

1.4.1 时间复杂性的构成

1. 时间复杂性的组成要素

- 一个程序所占时间$T(P)=$ 编译时间 $+$ 运行时间。
- 编译时间与实例特征无关，而且，一个编译好的程序可以运行若干次，所以，人们主要关心运行时间，记做t_P(实例特征)。

2. 时间复杂性的计算

根据编译器的特征可以确定代码 P 进行加、减、乘、除、比较、读、写等操作所需的时间，从而得到计算 t_P 的公式。用 n 代表实例特征，则

$$t_P(n)=c_a\,\mathrm{Add}(n)+c_s\,\mathrm{Sub}(n)+c_m\,\mathrm{Mul}(n)+c_d\,\mathrm{Div}(n)+\cdots$$

其中，c_a, c_s, c_m, c_d 等常量分别表示一次加、减、乘、除等操作所需的时间，$\mathrm{Add}, \mathrm{Sub}, \mathrm{Mul}, \mathrm{Div}$ 等函数分别表示代码 P 中加、减、乘、除等操作的次数。

3. 时间复杂性计算的难度

- 一个算术操作所需的时间取决于操作数的类型，所以，有必要对操作按照数据类型进行分类。
- 在构思一个程序时，影响t_P的许多因素还是未知的，所以，通常只对t_P进行估计。

1.4.2 操作计数

选定一种或多种操作，计算这些操作分别执行多少次。这些操作通常称为关键操作，关键操作对时间复杂性影响最大。

下面举例说明如何使用操作计数的方法来估计算法的时间复杂性。

1. 寻找最大元素

（1）问题。从$X[0..n)$中寻找最大元素。

（2）源程序。为方便其他算法调用，将该程序保存在文件Max.h中。

```
// Max.h
#pragma once
#include "algorithm.h"
```

```
template<class T>
int Max(T X[], int n)
{   int pos = 0;
    for(int i = 1; i < n; ++i)
        if(X[pos] < X[i]) pos = i;
    return pos;
}
```

（3）复杂性分析。这里的关键操作是比较。for循环中共进行了n-1次比较。Max还执行了其他比较，如for循环每次比较之前都要比较一下i和n。此外，Max还进行了其他的操作，如初始化pos、循环控制变量i的增量。这些一般都不包含在估算中，若纳入计数，则操作计数将增加一个常量。

2. 插入算法

（1）问题。向升序数组中插入一个新元素。

（2）算法思路。为了向升序数组X中插入元素v，首先需要从数组尾部开始找到第一个不大于v的元素，然后将v插入该元素之后。图1-1演示了向数组{3, 5, 6}插入元素4和7的过程。

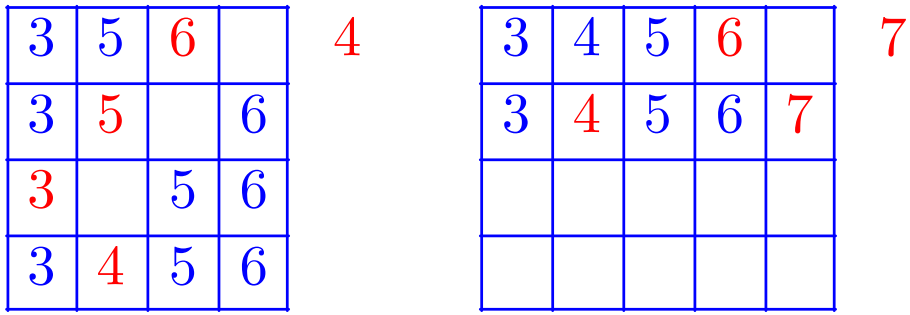


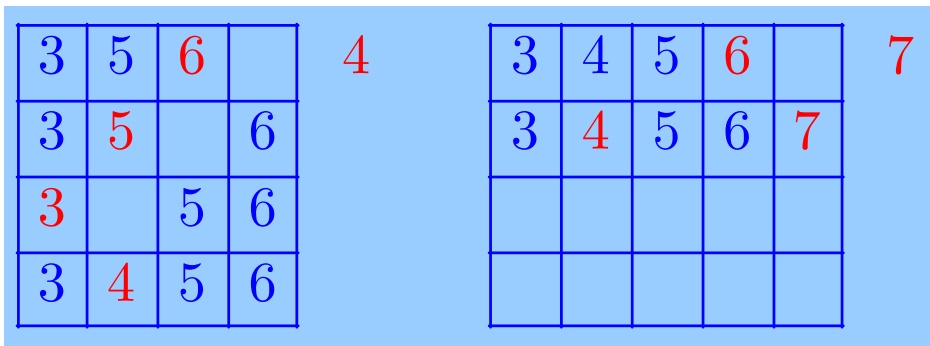
图1-1　向升序数组插入元素

【注】除问题背景隐含或指定了排列顺序以外，在本书中，插入和排序等问题涉及的排列顺序均默认为升序。

（3）算法描述。

```
#include "algorithm.h"
template<class T, class T2> // 在升序组X[m]中插入v
void Insert(T X[], int m, const T2 &v)
{   int i; // 从尾部开始找到第一个不大于v的元素
    for(i = m - 1; i >= 0 and X[i] > v; --i)
        X[i + 1] = X[i];
    X[i + 1] = v;
}
```

（4）复杂性分析。以原数组X的大小m为实例特征，程序的关键操作是v与X中元素的比较。容易看出，for循环中最多进行m次这样的比较，因而总的操作数最多为m。这里同样忽略了for循环的额外开销和元素移动等其他开销。

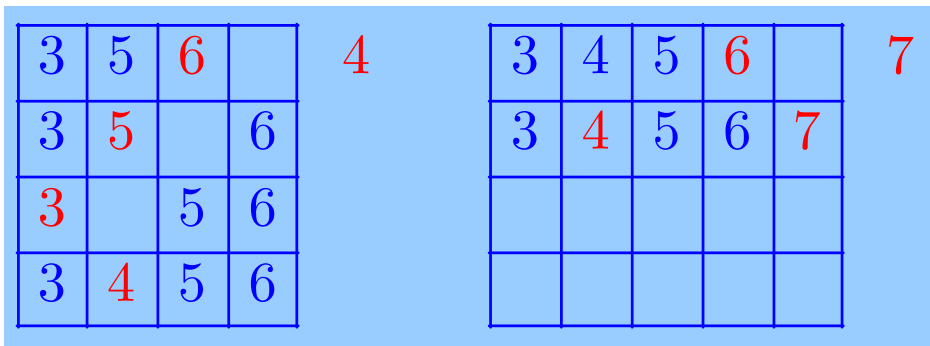


3. 最好、最坏和平均操作计数

（1）问题引入。观察Insert()程序可以发现，Insert()程序执行比较的次数不仅依赖于实例特征，还与数组元素和待插入元素的具体值有关，比较次数可在1到m之间变化。例如，若待插入元素正好插入到数组尾部，则只需进行1次比较，而如果待插入元素正好插入到数组首部或首部元素之后，则必须进行m次比较。

由Insert()程序执行比较的次数可知，操作计数不总是由实例特征唯一确定，所以，人们通常还关心最好的、最坏的和平均的操作计数。

```
template<class T, class T2>
void Insert(T X[], int m, const T2 &v)
{   int i;
    for(i = m - 1; i >= 0 and X[i] > v; --i)
        X[i + 1] = X[i];
    X[i + 1] = v;
}
```



（2）相关定义。令 P 表示一个程序，将操作计数 $O_P(n_1,n_2,\ldots,n_k)$ 视为实例特征 $n_1,n_2,\ldots,n_k$ 的函数。用 $O_P(I)$ 表示程序实例 I 的操作计数，$S(n_1,n_2,\ldots,n_k)$ 表示程序 P 的具有实例特征 $n_1,n_2,\ldots,n_k$ 的实例集合，则

- P 的最好（Best）操作计数为

$$O_P^B(n_1,n_2,\ldots,n_k)=\min\left\{O_P(I):I\in S(n_1,n_2,\ldots,n_k)\right\}$$

- P 的最坏（Worst）操作计数为

$$O_P^W(n_1,n_2,\ldots,n_k)=\max\left\{O_P(I):I\in S(n_1,n_2,\ldots,n_k)\right\}$$

- P 的平均（Mean）操作计数为

$$O_P^M\left(n_1,n_2,\ldots,n_k\right)=\frac{1}{\left|S\left(n_1,n_2,\ldots,n_k\right)\right|}\sum\left\{O_P(I):I\in S\left(n_1,n_2,\ldots,n_k\right)\right\}$$

- P 的期望（Expectation）操作计数为（操作计数的数学期望）

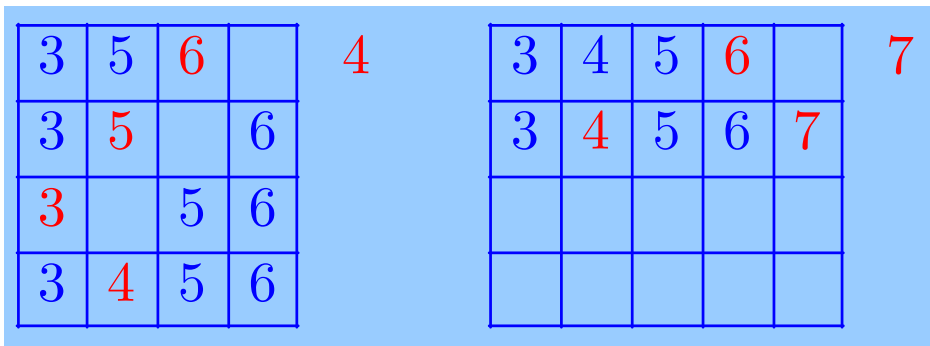
$$O_P^E\left(n_1,n_2,\ldots,n_k\right)=\sum\left\{p(I)\times O_P(I):I\in S\left(n_1,n_2,\ldots,n_k\right)\right\}$$

其中，$p(I)$ 是实例 I 可被成功解决的概率。

（3）举例说明。例如，在插入程序Insert(T X[], int m, const T2 &v)中，以原数组X的大小m为实例特征，程序的关键操作是v与X中元素的比较。此时，比较次数并不由m唯一确定。

- 最好操作计数。若待插入元素正好插入到数组尾部，则只需进行1次比较，这是最好的比较次数。
- 最坏操作计数。若待插入元素正好插入到数组首部或首部元素之后，则必须进行m次比较，这是最坏的比较次数。
- 平均操作计数。假定待插入元素v有相等的机会插入到数组任何一个可能位置（共$m+1$种可能）。由该假定可知，若v插入到位置$i+1$（$i \geq 0$），则比较次数是$m-i$，若v插入到位置0，则比较次数为m。所以，平均比较次数为$[(m-0)+(m-1)+...+(m-(m-1))+m]/(m+1)=m/2+m/(m+1)$。

```
template<class T, class T2>
void Insert(T X[], int m, const T2 &v)
{   int i;
    for(i = m - 1; i >= 0 and X[i] > v; --i)
        X[i + 1] = X[i];
    X[i + 1] = v;
}
```



4. 插入排序

（1）算法思路。将原数组中的元素依次插入结果数组中（可以使用原数组）。

（2）算法描述。

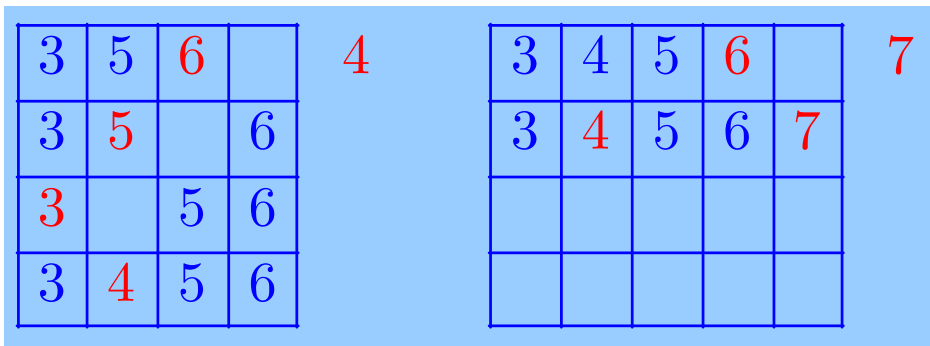
```
template<class T> // 插入排序，对X[0 .. n)排序
void InsertionSort(T X[], int n)
{   for(int m = 1; m < n; ++m)
        Insert(X, m, T{X[m]}); // 插入，需复制X[m]
}
```

（3）复杂性分析。

- 最好情况。若X是正序数组，则每次插入操作执行的比较次数都是1，从而排序操作执行的比较次数为n-1。
- 最坏情况。若X是逆序数组，则每次插入操作执行的比较次数都是m，从而排序操作执行的比较次数为1+2+...+(n-1)=n(n-1)/2。
- 平均情况。因为每次插入操作执行的平均比较次数是$m/2+m/(m+1)$，所以排序操作执行的平均比较次数为

$$\sum_{m=1}^{n-1}\left(m/2+m/(m+1)\right)\approx n(n-1)/4+(n-1)$$

```
void Insert(Seq &&X, int m, const T &v)
{   int i;
    for(i = m - 1; i >= 0 and X[i] > v; --i)
        X[i + 1] = X[i];
    X[i + 1] = v;
}
```



1.4.3 执行步数

1. 程序步

利用操作计数的方法估计程序的时间复杂性，忽略了所选择操作之外其他操作的开销。统计执行步数的方法则要统计程序中所有部分的时间开销。

执行步数也是实例特征的函数，每一步执行一个程序步。所谓程序步是一个语法或语义上的程序片段，该片段的执行时间独立于实例特征。例如，语句“return $a+b+b*c+(a+b+c)/(a+b)+4$;”和“$x=y$;”都可以作为一个程序步。

2. 执行步数统计

可以使用执行步数统计表来统计程序的执行步数。这里以顺序搜索为例说明执行步数统计表的使用方法。

（1）问题。在数组$X[0 .. n)$中顺序搜索指定元素v，若找到则返回该元素所在的位置，否则返回-1。

（2）算法描述与执行步数统计表。如表2-1所示。

表2-1　执行步数统计表

语句	步数	频率	总步数
template<class T, class T2>			
int Find(T X[], int n, const T2 &v)			
{ for(int i = 0; i < n; ++i)	1	$\leq n+1$	$\leq n+1$
if(X[i] == v) return i;	1	$\leq n$	$\leq n$
return -1;	1	≤ 1	≤ 1
}			
总计	$\leq 2n+2$		

其中，“步数”表示每次执行该语句所需要的执行步数（执行该语句所产生的counter值的变化量），频率是指该语句的总执行次数。

n	4	8	16	32	64
$1000n$	4000	8000	16000	32000	64000
2^n	16	256	65536	>4E9	>1.8E19

1.5 渐近复杂性的估计

确定程序的操作计数和执行步数都是为了比较完成同一功能的两个程序的时间复杂性，预测程序的运行时间随实例特征而变化的变化量。

设 $T(n)$ 是算法 A 的复杂性函数，如果存在函数 $\overline{T}(n)$，使得当 $n \to \infty$ 时有 $\left[T(n)-\overline{T}(n)\right]/T(n) \to 0$，则称 $\overline{T}(n)$ 是 $T(n)$ 当 $n \to \infty$ 时的渐近性态，或称 $\overline{T}(n)$ 是算法 A 当 $n \to \infty$ 时的渐近复杂性。一般来说，$\overline{T}(n)$ 比 $T(n)$ 简单。例如 $T(n)=10n^2+4n+2$，$\overline{T}(n)=10n^2$。

要比较两个算法的渐近复杂性，只要确定出各自的阶，就可以确定哪个算法效率更高。换句话说，渐近复杂性分析只关心 $\overline{T}(n)$ 的阶，而不关心 $\overline{T}(n)$ 的常数因子。

上述讨论已经给出了简化算法复杂性分析的方法和步骤，即只考虑当问题的规模充分大时，算法复杂性在渐近意义下的阶。为了方便描述，科学家们引入了一些渐近符号。

在下面的讨论中，用 $f(n)$ 表示一个程序的时间复杂性或空间复杂性，它是实例特征 n（$n \geq 0$）的非减函数，且 $f(n) \geq 0$。

1.5.1 渐近符号O（上界）

1. 符号O的定义

$f(n)=O(g(n))$ 当且仅当存在正常数 c 和 n_0，使得当 $n\geq n_0$ 时，有 $f(n)\leq cg(n)$。此时，称 $g(n)$ 是 $f(n)$ 的一个上界。上界函数通常使用单项形式。

【注】可以将 $f(n)=O(g(n))$ 理解为在数量级上 $f(n)\leq g(n)$。

表2-2给出了一些常用的阶函数。

表2-2 常用阶函数

函数	名称	函数	名称
1	常数	n^2	平方
$\log n$	对数[①]	n^3	立方
n	线性	2^n	指数
$n\log n$		$n!$	阶乘

[①]如果没有特意指明底数或使用特殊的函数名称，本书中出现的对数函数都是指以 2 为底数的对数。实际上，复杂性分析中出现的对数函数是不关心底数的。

2. 几个例子

如表2-3所示。

表2-3　上界函数的例子

常数函数	$f(n)=C_0=O(1)$	可取 $c=C_0$，$n_0=0$
线性函数	$3n+2=O(n)$	可取 $c=4$，$n_0=2$
线性函数	$100n+6\ln n=O(n)$	可取 $c=102$，$n_0=6$
平方函数	$10n^2+4n+3=O(n^2)$	可取 $c=11$，$n_0=5$
平方函数	$n\times\log n+n^2=O(n^2)$	可取 $c=2$，$n_0=3$
指数函数	$6\times 2^n+n^2=O(2^n)$	可取 $c=7$，$n_0=4$
松散界限	$3n+2=O(n^2)$	可取 $c=3$，$n_0=2$
错误界限	$3n+2\neq O(1)$	

3. 运算规则

设 $f=O(F)$， $g=O(G)$，根据定义，容易证明下列运算规则。

- $\max(f,g)=O(F+G)$。
- $f+g=O(F+G)$。
- $fg=O(FG)$。
- 如果 $f=O(g)$，则 $f=O(G)$。
- $Cf=O(f)$，其中 C 是一个正的常数。
- $f=O(f)$。

【注】上述运算规则对后续两小节介绍的下界符号和双界符号也成立。

4. 三点注意事项

（1）不要产生松散的界限。用来比较的函数 $g(n)$ 应尽量接近待考虑函数 $f(n)$。例如，$3n+2=O(n)$ 是较好的界限，而 $3n+2=O(n^2)$ 是松散的界限。

（2）不要产生错误的界限。例如 $n^2+100n+6$，当 $n<10$ 时，$n^2+100n+6<110n$，如果由此认为 $n^2+100n+6=O(n)$ 就会产生错误的界限。事实上，对任何正常数 c，只要 $n>c-100$ 就有 $n^2+100n+6>c\times n$。同理，$3n^2+4\times 2^n=O(n^2)$ 也是错误的界限。

（3）$f(n)=O(g(n))$ 不能写成 $g(n)=O(f(n))$。

- 两者并不等价，因为这里的等号并不表示相等。
- $O(g(n))$ 表示所有满足条件“存在正常数 c 和 n_0，使得当 $n\geq n_0$ 时，有 $f(n)\leq cg(n)$”的 $f(n)$ 构成的集合。
- $f(n)=O(g(n))$ 常读作“$f(n)$ 是 $g(n)$ 的一个大 O 成员”，即 $f(n)\in O(g(n))$。

5. O比率定理

【O比率定理】对于函数 $f(n)$ 和 $g(n)$，如果 $\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}$ 存在，则 $f(n)=O(g(n))$ 当且仅当存在 $c>0$，使得 $\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}\le c$。

根据O比率定理，容易得出下列上界关系。

- 因为 $\lim\limits_{n\to\infty}\dfrac{3n+2}{n}=3$，所以 $3n+2=O(n)$。
- 因为 $\lim\limits_{n\to\infty}\dfrac{10n^2+4n+2}{n^2}=10$，所以 $10n^2+4n+2=O(n^2)$。
- 因为 $\lim\limits_{n\to\infty}\dfrac{6\times 2^n+n^2}{2^n}=6$，所以 $6\times 2^n+n^2=O(2^n)$。
- 因为 $\lim\limits_{n\to\infty}\dfrac{n^{16}+3n^2}{2^n}=0$，所以 $n^{16}+3n^2=O(2^n)$。这是一个松散的上界估计。

6. 一些常用的上界关系

使用O比率定理容易证明下列常用的上界关系。

【常用上界关系】对于任何正数α和ε，有$\log^{-\alpha} n = O(1)$，$\alpha = O(\log^{\varepsilon} n)$，$\log^{\alpha} n = O(\log^{\alpha+\varepsilon} n)$，$\log^{\alpha} n = O(n^{\varepsilon})$，$n^{\alpha} = O(n^{\alpha+\varepsilon})$，$n^{\alpha} = O(2^n)$和$2^n = O(n!)$。

根据这些常用的上界关系，容易得出下列上界关系。

（1）由常用上界关系，有$\log n = O(n)$，所以$n^3 + n^2 \log n = O(n^3)$。

（2）由常用上界关系，有$\log^{20} n = O(n^{1.5})$，所以

$$n^4 + n^{2.5} \log^{20} n = O(n^4)。$$

（3）由常用上界关系，有$\log^3 n = O(n)$和$\log^{-3} n = O(1)$，所以

$$2^n n^4 \log^3 n + 2^n n^5 \log^{-3} n = O(2^n n^5)$$

1.5.2 渐近符号Ω（下界）

1. 符号Ω的定义

$f(n)=\Omega(g(n))$ 当且仅当存在正常数 c 和 n_0，使得当 $n\geq n_0$ 时，有 $f(n)\geq cg(n)$。

【注】可以将 $f(n)=\Omega(g(n))$ 理解为在数量级上 $f(n)\geq g(n)$。

显然，$f(n)=\Omega(g(n))$ 当且仅当 $g(n)=O(f(n))$。根据该性质，可以使用前述常用上界关系来估计复杂性函数的下界，而且有下述判定规则。

2. Ω比率定理

【Ω比率定理】对于 $f(n)$ 和 $g(n)$，如果 $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)}$ 存在，则 $f(n)=\Omega(g(n))$ 当且仅当存在 $c>0$，使得 $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)}\leq c$。

1.5.3 渐近符号Θ（双界）

1. 符号Θ的定义

$f(n)=\Theta(g(n))$ 当且仅当存在正常数 c_1， c_2 和 n_0，使得当 $n\geq n_0$ 时，有 $c_1g(n)\leq f(n)\leq c_2g(n)$。

【注】可以将 $f(n)=\Theta(g(n))$ 理解为在数量级上 $f(n)=g(n)$。

显然，$f(n)=\Theta(g(n))$ 当且仅当 $f(n)=\Omega(g(n))$ 和 $f(n)=O(g(n))$ 都成立。

2. 例子

- $3n+2=\Theta(n)$
- $10n^2+4n+2=\Theta(n^2)$
- $5\times 2^n+n^2=\Theta(2^n)$

3. Θ比率定理

【Θ比率定理】对于函数 $f(n)$ 和 $g(n)$，如果 $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)}$ 与 $\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}$ 都存在，则 $f(n)=\Theta(g(n))$ 当且仅当存在正常数 c_1 和 c_2，使得 $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)}\le c_1$，$\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}\le c_2$。

4. 多项式情形的复杂性函数

【多项式阶函数】设 $f(n)=a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n+a_0$，如果 $a_m>0$，则由比率定理可知 $f(n)=O(n^m)$，$f(n)=\Omega(n^m)$，$f(n)=\Theta(n^m)$。

一般情况下，不能对每个复杂性函数去估计它们的上界、下界和双界，常用上界关系和多项式阶函数给出了一些直接确定这些界的阶函数的参考信息。

1.5.4 渐近复杂性估计举例

1. 选择排序

基本思路是依次将最大元素调至数组尾部，并同时减少待检查元素个数，算法描述和执行步数的渐近统计如表2-4所示。

表2-4 选择排序

语句	步数	频率	总步数
#include "Max.h"			
template<class T>			
void SelectionSort(T X[], int n)			
{ for(int m = n; m > 1; --m)	1	$\Theta(n)$	$\Theta(n)$
{ int i = Max(X, m); // 需m-1次比较	$\Theta(m)$	$\Theta(n)$	$\Theta(n^2)$
swap(X[i], X[m - 1]); // 最大元素调后	1	$\Theta(n)$	$\Theta(n)$
}			
}			
总计	$\Theta(n^2)$		

【注】调用Max()的总步数为$\Theta(1+2+...+n)= \Theta(n^2)$。

2. 冒泡排序

（1）冒泡。从左向右逐个检查，对相邻元素进行比较，若左边元素比右边元素大，则交换这两个元素。例如，数组[5, 3, 7, 4, 11, 2]的冒泡过程如图1-2所示。一次冒泡后，原数组变为[3, 5, 4, 7, 2, 11]。

5，3，7，4，11，2
3，5，7，4，11，2
3，5，7，4，11，2
3，5，4，7，11，2
3，5，4，7，11，2
3，5，4，7，2，11

图1-2　冒泡过程

冒泡过程的C++描述如下。

```
#include "algorithm.h"
template<class T> // 冒泡
bool Bubble(T X[], int m)
{   bool swapped = false;
    for(int i = 0; i < m - 1; ++i)
        if(X[i] > X[i + 1])
            swap(X[i], X[i + 1]), swapped = true;
    return swapped;
}
```

5, 3, 7, 4, 11, 2
3, 5, 7, 4, 11, 2
3, 5, 7, 4, 11, 2
3, 5, 4, 7, 11, 2
3, 5, 4, 7, 11, 2
3, 5, 4, 7, 2, 11

（2）排序。冒泡后，数组的最大元素移动到最右位置。下一次冒泡只对左边的元素进行，忽略最右元素。如此进行下去，最后将原数组按升序排列。

实际上，如果在某次冒泡过程中没有交换元素，则可结束排序过程。

排序过程的C++描述如下。

```
template<class T> // 排序
void BubbleSort(T X[], int n)
{   for(int m = n; m > 1 ; --m)
        if(not Bubble(X, m)) return;
}
```

（3）复杂性分析。在冒泡过程中，每次循环耗时$\Theta(1)$，共循环m-1次，因此，冒泡过程的时间复杂度为$\Theta(m)$。在最好情况下（原数组是正序的），冒泡排序只需调用一次冒泡过程，因此冒泡排序的最好时间复杂性为$\Theta(n)$。在最坏情况下（原数组是逆序的），冒泡排序对每个m都需要调用冒泡过程，因此，冒泡排序的最坏时间复杂性为$\Theta(1+2+...+n)=\Theta(n^2)$。

```
template<class T> // 冒泡
bool Bubble(T X[], int m)
{   bool swapped = false;
    for(int i = 0; i < m - 1; ++i)
        if(X[i] > X[i + 1])
            swap(X[i], X[i + 1]),
                swapped = true;
    return swapped;
}
```

1.5.5 简化Master定理

在分析程序的时间复杂性时，会经常遇到形如 $T(n)=aT(n/b)+X(n^{\alpha})$ 的递归函数。其中，$a>0$，$b>1$，$\alpha\geq 0$，X 代表 O、Ω、Θ 之一，n/b 可以理解为 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$。可以使用简化Master定理找到这类函数的阶函数。

1. 定理陈述

【简化Master定理】当 $a>0$（通常会有 $a\geq 1$），$b>1$，$\alpha\geq 0$ 时，递归函数 $T(n)=aT(n/b)+X(n^{\alpha})$（$X$ 代表 O、Ω、Θ 之一）的阶函数为

$$T(n)=\begin{cases} X(n^{\log_b a}) & \alpha<\log_b a \\ X(n^{\alpha}\log n) & \alpha=\log_b a \\ X(n^{\alpha}) & \alpha>\log_b a \end{cases}$$

2. 举例说明

（1）$T(n)=T(n/2)+\Theta(1)$。这里 $a=1$，$b=2$，$\alpha=0$。由 $\alpha=\log_2 1$ 和简化Master定理可知 $T(n)=\Theta(n^0\log n)=\Theta(\log n)$。这是折半搜索的时间复杂性。

（2）$T(n)=T(n/2)+\Theta(n)$。这里 $a=1$，$b=2$，$\alpha=1$。由 $\alpha>\log_2 1$ 和简化Master定理可知 $T(n)=\Theta(n)$。这是选择算法的平均时间复杂性。

（3）$T(n)=2T(n/2)+\Theta(n)$。这里 $a=2$，$b=2$，$\alpha=1$。由 $\alpha=\log_2 2$ 和简化Master定理可知 $T(n)=\Theta(n\log n)$。这是归并排序算法和快速傅立叶变换的时间复杂性，也是快速排序算法的平均时间复杂性。

（4）$T(n)=7T(n/2)+O(n^2)$。这里 $a=7$，$b=2$，$\alpha=2$。由 $\alpha<\log_2 7$（$\log_2 7\approx 2.81$）和简化Master定理可知 $T(n)=O(n^{\log_2 7})$。这是使用Strassen算法快速计算两矩阵乘积的时间复杂性。

（5）$T(n)=T(2n/3)+50$。这里 $a=1$，$b=3/2$，$\alpha=0$。由 $\alpha=\log_{3/2}1$ 和简化Master定理可知 $T(n)=\Theta(n^0\log n)=\Theta(\log n)$。

$$T(n)=\begin{cases}X(n^{\log_b a}) & \alpha<\log_b a\\ X(n^{\alpha}\log n) & \alpha=\log_b a\\ X(n^{\alpha}) & \alpha>\log_b a\end{cases}$$

1.6 练习题

1.6.1 基础知识题

1. 写出下列函数的上界估计，并说明结果的正确性。

（1）$f(n)=n^3+n^2\log n$

（2）$f(n)=n^4+n^{2.5}\log^{20} n$

（3）$f(n)=2^n n^4\log^3 n+2^n n^5/\log^3 n$

（4）$f(n)=6*2^n+2^n n^2\log n+3^{n+1}$

2. 给出下列递归函数的阶函数。

（1）$f(n)=9f(n/3)+n$

（2）$f(n)=f(2n/3)+1$

（3）$f(n)=2f(n/2)+\Theta(n)$

（4）$f(n)=7f(n/2)+O(n^2)$

（5）$f(n)=49f(n/25)+n^{3/2}$

3. 试判定函数 $f(n)$ 和 $g(n)$ 的关系（ $f(n)=O(g(n))$ ， $f(n)=\Omega(g(n))$ ， $f(n)=\Theta(g(n))$ ），并说明结果的正确性。

（1） $f(n)=\log n^2$ ； $g(n)=\log n+5$

（2） $f(n)=\log n^2$ ； $g(n)=\sqrt{n}$

（3） $f(n)=n$ ； $g(n)=\log^2 n$

（4） $f(n)=n\log n+n$ ； $g(n)=\log n$

（5） $f(n)=\log^2 n$ ； $g(n)=\log n+\sqrt{n}$

（6） $f(n)=6*2^n$ ； $g(n)=100n^2$

4. 证明：对于多项式函数 $a_m n^m+a_{m-1}n^{m-1}+\cdots+a_1 n+a_0$ ，如果 $a_m>0$ ，则 $f(n)=O(n^m)$ ， $f(n)=\Omega(n^m)$ ， $f(n)=\Theta(n^m)$ 。

5. 假定 $a>0$ ， $b>1$ ， $\alpha\geq 0$ ，使用简化Master定理证明当 $\alpha<\log_b a$ 时，对于递归函数 $T(n)=aT(n/b)+O(n^\alpha)$ ，有 $T(n)=\Theta(n^{\log_b a})$ 。

1.6.2 实验题

1. 使用一种程序设计语言分别写出一种从一个数组中寻找最大元素和最小元素的算法，并分析其时间复杂性。

2. 使用一种程序设计语言写出选择排序算法的程序，并分析其时间复杂性。

3. 使用一种程序设计语言写出一种向一个有序数组中插入一个新元素的算法（需要判断原数组是升序还是降序），并分析其时间复杂性。

4. 使用一种程序设计语言写出插入排序算法的程序，并分析其时间复杂性。

5. 使用一种程序设计语言写出一种不重复的插入算法，该算法用于向一个不含重复元素的升序数组中不重复地插入一个新元素。

1.6.3 进阶训练题

假定 $a>0$，$b>1$，$\alpha \geq 0$，证明当 $\alpha<\log_b a$ 时，对于递归函数 $T(n)=aT(n/b)+O(n^{\alpha})$，有 $T(n)=\Theta(n^{\log_b a})$。注意，不能直接使用Master定理或简化Master定理的结论。