第2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
- 空间复杂性
- 时间复杂性
- 渐近符号
- 递归函数的求解

2.1 算法与程序性能概述

2.1.1 算法与程序

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[①]的操作序列。

程序是算法用某种程序设计语言的具体实现。

① 这里的有穷不是一个纯数学概念，而是在实际上是合理的，可接受的。

1. 算法的特性

一个算法必须具有下列五个重要特性。

① 有穷性。一个算法必须总是(对任何合法的输入值)在执行有穷步之后结束,且每一步都可在有穷时间内完成。

② 确定性。算法中每一条指令必须有确切的含义,读者理解时不会产生二义性。并且,在任何条件下,算法只有唯一的一条执行路径,即对于相同的输入只能得出相同的输出。

③ 可行性。算法中描述的操作都可以通过已经实现的基本运算执行有限次来实现。

④ 输入。一个算法有零个或多个的输入,这些输入取自于某个特定的对象集合。

⑤ 输出。一个算法有一个或多个的输出。这些输出是同输入有着某些特定关系的量。

【注】如果只满足②~⑤,只能叫做计算过程。如果②中执行路径不满足唯一性,则称为非确定性算法。

2. 程序的特性

程序可以不满足算法的性质①即有穷性。例如，操作系统是一个在无限循环中执行的程序，因而不是一个算法。

2.1.2 算法描述与算法设计策略

1. 算法描述

描述算法有多种方式，如自然语言方式、图形表格方式等。本教材主要采用C++17，并使用STL，编译器使用Dev-C++。有时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的方式来描述算法。伪代码使用类似于C/C++的语法。

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 <queue> // queue
```

```
#include "queue.hpp" // minheap, maxheap等，第1章介绍
#include "Matrix.hpp" // 矩阵，第4章介绍
#include "ios.hpp" // to_string, io等，第1章介绍
```

```
// 保证只需定义<和==就可以在std中使用!=, <=, >, >=
namespace std { using namespace rel_ops; }
namespace std { const auto inf = INFINITY; } // INFINITY缩写为inf
using namespace std;
```

2. 常用的算法设计策略

分治方法、贪心方法、动态规划方法、回溯方法、分支限界方法、概率方法等。

2.1.3 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

- 如果对于解决一个问题有多种可选的方案，那么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。

2.2 空间复杂性

2.2.1 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。有些编译器带有选项，如优化模式、覆盖模式等。所取的选项不同，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处理硬件，那么，每个浮点操作可以转化为一条机器指令。否则，必须生成仿真的浮点计算代码，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。

2.2.2 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 两个数作为实例特征。

2. 空间复杂性组成部分分析

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。

令 $S(P)$ 表示程序 P 需要的空间，有 $S(P)=C+S_P(\text{实例特征})$，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 $S_P(\text{实例特征})$。实例特征的选择一般受到相关数的数量以及程序输入和输出规模的制约。

2.2.3 举例

1. 顺序搜索

- 问题。在X[0 to n-1]中搜索v，若找到则返回所在的位置，否则返回-1。
- 源程序。
- 实例特征。采用数组的长度n作为实例特征。
- 分析。数组地址X需4字节，参数v需4字节，n需4字节，局部变量i需4字节，常量-1需4字节，共需20字节，独立于n，所以 $S_{\text{Find}}(n)=0$。
- 注意。数组X所需要的空间是在其他函数中分配的，不能算作函数Find所需要的空间。

```
template<class T, class K>
int Find(T X[], int n, const K &v)
{   for(int i = 0; i < n; ++i)
        if(X[i] == v) return i;
    return -1;
}
```

2. 阶乘

- 源程序。分别给出累积计算和递归计算的源程序。
- 实例特征。取自变量n作为实例特征。
- 累积计算。n，i和result的空间与n无关，因此，$S_{\text{Factorial}}(n)=0$。
- 递归计算。递归栈空间包括参数n（4字节）以及返回地址的空间（4字节），每次调用Fact共需8字节。嵌套调用一直进行到n = 1。递归深度为n，所以需要$8n$字节的递归栈空间。$S_{\text{Fact}}(n)=8n$。

```
int Factorial(int n)
{   int result = 1;
    for(int i = 1; i <= n; ++i)
        result *= i;
    return result;
}
```

```
int Fact(int n)
{   if(n <= 1) return 1;
    return n * Fact(n - 1);
}
```

2.3 时间复杂性

2.3.1 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、写等所需的时间，从而得到计算 t_P 的公式。令 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，在多数情况下仅仅是对 t_P 进行估计。

2.3.2 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。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。寻找X[0 to n-1]中的最大元素。

② 源程序。

```
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还执行了其它比较，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. 选择排序*

① 源程序。

```
#include "algorithm.h"
template<class T> // 对数组X[0 to n - 1]中元素排序
void SelectionSort(T X[], int n)
{   for(int size = n; size > 1; --size)
        swap(*max_element(X, X + size), X[size - 1]);
    // max_element需要size - 1次比较
}
```

② 复杂性分析。关键的操作是比较。程序首先找出最大元素，把它移动到X[n-1]，然后在余下的n-1个元素中再寻找最大的元素，并把它移动到X[n-2]。如此进行下去。每次调用max_element需要执行size-1次比较，总的比较次数为 $1+2+\ldots+(n-1)=n(n-1)/2$ 。每调用一次swap()需要三次元素移动，总的移动次数为 $3(n-1)$ 。忽略了for循环的额外开销等其它开销。

3. 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。如数组[5, 3, 7, 4, 11, 2]的冒泡过程如下。

比较5和3，交换；比较5和7，不交换；比较7和4，交换；比较7和11，不交换；比较11和2，交换。一次冒泡后，原数组变为[3, 5, 4, 7, 2, 11]。如图2-1所示。

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2-1　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 n)
{   bool swapped = false;
    for(int i = 0; i < n - 1; ++i)
        if(X[i] > X[i + 1])
            swap(X[i], X[i + 1]), swapped = true;
    return swapped;
}
```

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，最后将原数组按递增顺序排列。

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 i = n; i > 1 ; --i)
        if(!Bubble(X, i)) return;
}
```

④ 复杂性分析。在程序Bubble中，for循环的每一回都执行了一次比较和三次元素的移动，因而总的操作数最多为 $4(n-1)$。在程序BubbleSort中，对于每个i，调用函数Bubble(X, i)需要执行最多 $4(i-1)$ 次操作，因而总的操作数最多为 $4(2-1)+\ldots+4(n-1)=2n(n-1)$。这里忽略了for循环的额外开销等其它开销。

```
bool Bubble(T X[], int n)
{   bool swapped = false;
    for(int i = 0; i < n - 1; ++i)
        if(X[i] > X[i + 1])
            swap(X[i], X[i + 1]), swapped = true;
    return swapped;
}
void BubbleSort(T X[], int n)
{   for(int i = n; i > 1 ; --i)
        if(!Bubble(X, i)) return;
}
```

4. 最好的、最坏的和平均的操作计数

观察Bubble程序可以发现，Bubble程序执行交换的次数不仅依赖于实例特征，还与数组元素的具体值有关。交换次数可在0到$n-1$之间变化。例如，若数组是递增的，则每一次冒泡都不需要交换数据，而如果数组是递减的，则第一次冒泡就要交换数据$n-1$次。由于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数。

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

```
bool Bubble(T X[], int n)
{   bool swapped = false;
    for(int i = 0; i < n - 1; ++i)
        if(X[i] > X[i + 1])
            swap(X[i], X[i + 1]),
            swapped = true;
    return swapped;
}
void BubbleSort(T X[], int n)
{   for(int i = n; i > 1 ; --i)
        if(!Bubble(X, i)) return;
}
```

- P 的最好（Best）操作计数为

$$O_P^B(n_1,n_2,\ldots,n_k)=\min_{I\in S(n_1,n_2,\ldots,n_k)} O_P(I)$$

- P 的最坏（Worst）操作计数为

$$O_P^W(n_1,n_2,\ldots,n_k)=\max_{I\in S(n_1,n_2,\ldots,n_k)} O_P(I)$$

- P 的平均（Mean）操作计数为

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

- P 的期望（Expectation）操作计数为（操作计数的数学期望）

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

其中，$p(I)$ 是实例 I 可被成功解决的概率。

在顺序搜索Find(T X[], const T& v, int n)中，取n作为实例特征，关键操作是比较。此时，比较的次数并不是由n唯一确定的。若n=100，v=X[0]，那么仅需要执行一次操作；若v不是X中的元素，则需要执行100次比较。

每当进行一次不成功的查找，就需要执行 n 次比较。对于成功的查找，最好的比较次数是1，最坏的比较次数为 n 。若假定每个实例出现的概率都是相同的，则成功查找的平均比较次数为

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

5. 插入排序算法

① 一次插入。为了在有序数组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2-2演示了在数组{3, 5, 6}中插入元素4的过程。

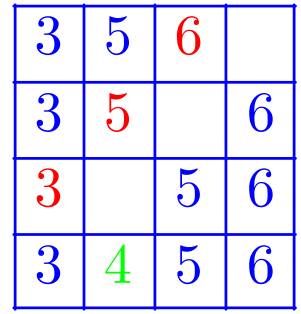

图2-2 在有序数组中插入元素

插入过程的C++描述如下，保存在文件InsertionSort.h中。

```
// InsertionSort.h
#pragma once
template<class T, class T2> // 在有序组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 && X[i] > v; --i)
        X[i + 1] = X[i];
    X[i + 1] = v;
}
```

在Insert中假定X中元素在插入v前后都是升序排列的。取数组X的大小 m 作为实例特征，程序的关键操作是v与X中元素的比较。显然，最少比较次数为1，发生在v插入数组尾部时；最多比较次数是 m，发生在v插入数组首部时。

② 插入排序。将原数组中的元素依次插入结果数组中（可以使用原数组）。

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

程序Insert的关键操作是v与X中元素的比较，最少比较次数为1，最多比较次数是 m。从而，在程序InsertionSort中所执行的比较次数，最好情况（正序数组）下是 $n-1$，最坏情况（逆序数组）下是

$$1+2+\ldots+(n-1)=n(n-1)/2$$

假定v有相等的机会插入任何一个可能位置（共 $n+1$ 种可能）。如果v插入到位置 $i+1$， $i\geq 0$，则比较次数是 $n-i$，若v插入到位置0，则比较次数为 n。所以，平均比较次数为

$$[(n-0)+(n-1)+\ldots+(n-(n-1)+n]/(n+1)$$
$$=n/2+n/(n+1)$$

从而，程序InsertionSort执行的平均比较次数为

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

2.3.3 执行步数

1. 程序步

利用操作计数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。所谓程序步是一个语法或语意上的程序片断，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。

可以通过创建全局变量counter来确定一个程序为完成预定任务所需要的执行步数。将counter引入程序语句中，当执行程序中的一条语句时，就为counter加上该语句所需要的执行步数。

2. 执行步数统计举例

```
int counter = 0;
template<class T>
void reverse(T X[], int n)
{   for(int i = 0, j = n - 1; i < j; ++i, --j)
    {   ++counter; // 对应于for循环(最多n/2步)
        T t = X[i];
        X[i] = X[j];
        X[j] = t;
        counter += 3; // 对应于循环体(最多3n/2步)
    }
    ++counter; // 对应于最后一次for循环
}
```

取 n 为实例特征，程序reverse总的执行步数最多为 $2n+1$ 。

3. 执行步数统计表

可以使用执行步数统计表来统计程序执行步数。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 T1, class T2>			
void replace(T X[], int n, const T1 &v1, const T2 &v2)			
{ for(int i = 0; i < n; ++i)	1	$\leq n+1$	$\leq n+1$
if(X[i] == v1) X[i] = v2;	1	$\leq n$	$\leq n$
}			
总计	$\leq 2n+1$		

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。

2.4 渐近符号

确定程序的操作计数和执行步数的目的是为了比较两个完成同一功能的程序的时间复杂性，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 $\left[T(n)-\overline{T}(n)\right]\Big/T(n) \to 0$，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 $n \to \infty$ 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，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。为此引入渐近符号。

在下面的讨论中，用 $f(n)$ 表示一个程序的时间或空间复杂性，它是实例特征 n（$n \geq 0$）的函数，且 $f(n)$ 对于 n 的所有取值均为非负实数。

2.4.1 渐近符号*O*（上界）

1. 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。上界函数通常取单项的形式。表2-2给出了一些常用的渐近函数。

表2-2 常用的渐近函数

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

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。
- $f=O(f)$。

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 $f(n)$。

- $3n+2=O(n^2)$ 是松散的界限。
- $3n+2=O(n)$ 是较好的界限。

② 不要产生错误的界限。例如 $n^2+100n+6$，当 $n<3$ 时，$n^2+100n+6<106n$，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”。

5. 大O比率定理

【大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 $\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}\le c$。

根据大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

使用大O比率定理容易证明下列常用的上界关系。

【常用上界关系定理】对于任何正数 x 和 ε，$\log^{-x} n = O(1)$，$x = O(\log^{\varepsilon} n)$，$\log^{x} n = O(\log^{x+\varepsilon} n)$ ，$\log^{x} n = O(n^{\varepsilon})$ ，$n^{x} = O(n^{x+\varepsilon})$ ，$n^{x} = O(2^{n})$ 和 $2^{n} = O(n!)$ 等上界关系都成立。

根据这些常用的上界关系，容易得出下列上界关系。

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

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

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

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

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

2.4.2 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 $f(n)\geq cg(n)$。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定理来估计复杂性函数的下界，而且有下述判定规则。

2. 大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 $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)}\leq c$。

2.4.3 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 $c_1g(n)\leq f(n)\leq c_2g(n)$。

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

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 $\lim\limits_{n\to\infty}\dfrac{g(n)}{f(n)}\le c_1$，$\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}\le c_2$。Θ比率定理是大$O$比率定理和Ω比率定理的综合。

4. 多项式情形的复杂性函数

【多项式阶函数定理】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，常用上界关系定理和多项式阶函数定理给出了一些直接确定这些界的阶函数（或叫渐近函数）的参考信息。

2.4.4 算法复杂性估计举例

1. 统计非零元素

如表2-4所示。

表2-4 统计非零元素的时间复杂性

语句	s/e	频率	总步数
template<class T>			
int CountNonZero(const T X[], int n)			
{ int k = 0;	1	$\Theta(1)$	$\Theta(1)$
for(int i = 0; i < n; ++i)	1	$\Theta(n)$	$\Theta(n)$
if(X[i] != 0) ++k;	1	$\Theta(n)$	$\Theta(n)$
return k;	1	$\Theta(1)$	$\Theta(1)$
}			
总计	$\Theta(n)$		

2. 冒泡排序

在Bubble()中，每次for循环需耗时 $\Theta(1)$。该循环共执行 n 次，因此，总时间复杂度为 $\Theta(n)$。在BubbleSort()中，对于每个i，调用Bubble(X, i)需耗时 $\Theta(i)$，从而总耗时最多为 $\Theta(1+2+\ldots+n)=\Theta(n^2)$。因此，在最坏情况下，BubbleSort()的总时间复杂性为 $\Theta(n^2)$。易知，在最好情况下，BubbleSort()的总时间复杂性为 $\Theta(n)$。

```
bool Bubble(T X[], int n)
{   bool swapped = false;
    for(int i = 0; i < n - 1; ++i)
        if(X[i] > X[i + 1])
            swap(X[i], X[i + 1]),
            swapped = true;
    return swapped;
}
void BubbleSort(T X[], int n)
{   for(int i = n; i > 1 ; --i)
        if(!Bubble(X, i)) return;
}
```

2.5 递归函数的求解

在分析程序的时间复杂性时，常常遇到递归函数。

2.5.1 一阶递归函数*

一阶递归函数比较简单，可以表示为

$$\begin{cases} f(n_0) = f_0 & n = n_0 \\ f(n) = af(n-1) + b & n > n_0 \end{cases}$$

采用直接推导的方法即可解得

$$f(n) = a^{n-n_0} f_0 + \begin{cases} (n-n_0)\,b & a = 1 \\ \dfrac{a^{n-n_0}-1}{a-1} b & a \neq 1 \end{cases}$$

2.5.2 二阶递归函数*

二阶递归函数可以表示为

$$\begin{cases} f(n_0) = f_0 \\ f(n_0+1) = f_1 \\ f(n) = af(n-1) + bf(n-2) \quad n > n_0+1 \end{cases}$$

可以用特征法来解。递归等式的特征方程为 $x^2 - ax - b = 0$ 。求出该方程的两个解 $x_1 = p, x_2 = q$ 。由韦达定理知 $a = p+q, b = -pq$ 。将它们代入递归等式得

$$\begin{aligned} f(n) - pf(n-1) &= qf(n-1) - qpf(n-2) \\ &= q(f(n-1) - pf(n-2)) \end{aligned}$$

令 $F(n)=f(n)-pf(n-1)$，$F_1=f_1-pf_0$，则上式变成了关于 $F(n)$ 的一阶递归等式

$$\begin{cases}F(n_0+1)=F_1\\F(n)=qF(n-1) & n>n_0+1\end{cases}$$

解得 $F(n)=q^{n-n_0-1}F_1$。于是

$$f(n)-pf(n-1)=q^{n-n_0-1}F_1$$

由此得关于 $f(n)$ 的一阶递归等式

$$\begin{cases}f(n_0)=f_0\\f(n)=pf(n-1)+q^{n-n_0-1}F_1 & n>n_0\end{cases}$$

解得

$$f(n)=p^{n-n_0}f_0+\begin{cases}(n-n_0)q^{n-n_0-1}F_1 & p=1\\\dfrac{p^{n-n_0}-1}{p-1}q^{n-n_0-1}F_1 & p\neq 1\end{cases}$$

$$f(n)-pf(n-1)=q(f(n-1)-pf(n-2))$$

2.5.3 简化Master定理

1. 定理适用范围

当 $a>0$，$b>1$，$\alpha \geq 0$ 时，对于形如 $T(n)=aT(n/b)+X(n^{\alpha})$（其中，$X$ 代表 O、Ω、Θ 之一，n/b 可以理解为 $\lfloor n/b \rfloor$ 或 $\lceil n/b \rceil$）的递归函数，可以使用简化Master定理找到它们的渐近函数。在关于复杂性（往往是分治方法的复杂性）的递归函数中，会经常遇到这类递归函数。

2. 定理陈述

当 $a>0$，$b>1$，$\alpha \geq 0$ 时，递归函数 $T(n)=aT(n/b)+X(n^{\alpha})$（$X$ 代表 O、Ω、Θ 之一）的渐近函数为

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

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

3. 举例

① $T(n)=9T(n/3)+n$。这里 $a=9$，$b=3$，$\alpha=1$。由于 $\alpha<\log_3 9$，由简化Master定理可知 $T(n)=\Theta(n^{\log_3 9})=\Theta(n^2)$。

② $T(n)=7T(n/2)+O(n^2)$。这里 $a=7$，$b=2$，$\alpha=2$。由于 $\alpha<\log_2 7\approx 2.81$，由简化Master定理可知 $T(n)=O(n^{\log_2 7})\approx O(n^{2.81})$。

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

④ $T(n)=2T(n/2)+\Theta(n)$。这里 $a=2$，$b=2$，$\alpha=1$。由于 $\alpha=\log_2 2$，由简化Master定理可知 $T(n)=\Theta(n\log n)$。

⑤ $T(n)=3T(n/4)+n\log n$。对任何正数 ε，都有 $T(n)=3T(n/4)+O(n^{1+\varepsilon})$（根据常用上界关系定理）。这里 $a=3$，$b=4$，$\alpha=1+\varepsilon$。由于 $\alpha>\log_4 3$，根据简化Master定理可得 $T(n)=O(n^{1+\varepsilon})$。

⑥ $T(n)=2T(n/2)+n\log n$ 。对任何正数 ε，都有 $T(n)=2T(n/2)+O(n^{1+\varepsilon})$（根据常用上界关系定理）。这里 $a=2$，$b=2$，$\alpha=1+\varepsilon$。由于 $\alpha>\log_2 2$，根据简化Master定理可得 $T(n)=O(n^{1+\varepsilon})$。

2.6 练习题

1、写出下列函数的上界估计并说明结果的正确性。

（1）$T(n)=n^3+n^2\log n$

（2）$T(n)=n^4+n^{2.5}\log^{20} n$

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

2、求解下列递归函数。

（1）$T(n)=9T(n/3)+n$

（2）$T(n)=T(2n/3)+1$

（3）$T(n)=3T(n/4)+n\log n$（上界）

（4）$T(n)=2T(n/2)+n\log n$（上界）

（5）$T(n)=2T(n/2)+\Theta(n)$

（6）$T(n)=7T(n/2)+O(n^2)$

（7）$T(n)=49T(n/25)+n^{3/2}\log n$

（8）$T(n)=\sqrt{n}$

（9）$T(n)=\log n+\sqrt{n}$

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。

$f(n)=\log n^2$ ； $g(n)=\log n+5$

$f(n)=\log n^2$ ； $g(n)=\sqrt{n}$

$f(n)=n$ ； $g(n)=\log^2 n$

$f(n)=n\log n+n$ ； $g(n)=\log n$

$f(n)=10$ ； $g(n)=\log 10$

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

$f(n)=6*2^n$ ； $g(n)=100n^2$

$f(n)=2^n n^2\log n$；　$g(n)=3^{n+1}$

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、使用一种高级程序设计语言写出下列算法的程序，并分析其时间复杂性。

选择排序，冒泡排序，插入排序

6、对于 $T(n)=aT(n/b)+O(n^\alpha)$，请根据“若 $\alpha<\log_b a$，则 $T(n)=O(n^{\log_b a})$”证明“若 $\alpha<\log_b a$，则 $T(n)=\Theta(n^{\log_b a})$”。