

第一章 复杂性分析初步

程序性能 (program performance) 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 (performance analysis) 与性能测量 (performance measurement), 前者采用分析的方法, 后者采用实验的方法。

考虑空间复杂性的理由

- ✓ 在多用户系统中运行时, 需指明分配给该程序的内存大小;
- ✓ 想预先知道计算机系统是否有足够的内存来运行该程序;
- ✓ 一个问题可能有若干个不同的内存需求解决方案, 从中择优;
- ✓ 用空间复杂性来估计一个程序可能解决的问题的最大规模。

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时间的上限 (用户可接受的);
- ✓ 所开发的程序需要提供一个满意的实时反应。

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及空间的复杂性, 最好采取加权的方式进行评价。但是随着计算机技术的迅速发展, 对空间的要求往往不如对时间的要求那样强烈。因此我们这里的分析主要强调时间复杂性的分析。


§ 1 空间复杂性

- **程序所需的空間**主要包括: 指令空间、数据空间、环境栈空间。

 **指令空间** 用来存储经过编译之后的程序指令。大小取决于如下因素:

- ✓ 把程序编译成机器代码的编译器;
- ✓ 编译时实际采用的编译器选项;
- ✓ 目标计算机。

注: 所使用的编译器不同, 则产生的机器代码的长度就会有差异; 有些编译器带有选项, 如优化模式、覆盖模式等等。所取的选项不同, 产生机器代码也会不同。采用优化模式可以缩减机器代码, 但要多消耗时间。此外, 目标计算机的配置也会影响代码的长度。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。

 **数据空间** 用来存储所有常量和变量的值。

- ✓ 存储常量和简单变量; 所需的空間取决于所使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間及动态分配的空間；
- ✓ 计算方法：


结构变量所占空间等于各个成员所占空间的累加；数组变量所占空间等于数组大小乘以单个数组元素所占的空间。例如

`double a[100]`；所需空间为 $100 \times 8 = 800$

`int matrix[r][c]`；所需空间为 $4 \times r \times c$

C++基本数据类型（32 位字长机器）

类型名	说明	字节数	范围
<code>char</code>	字符型	1	-128~127
<code>signed char</code>	有符号字符型	1	-128~127
<code>unsigned char</code>	无符号字符型	1	0~255
<code>short[int]</code>	短整型	2	-32768~32767
<code>signed short[int]</code>	有符号短整型	2	-32768~32767
<code>unsigned short[int]</code>	无符号短整型	2	0~65535
<code>int</code>	整型	4	-2147483648~2147483647
<code>signed[int]</code>	有符号整型	4	-2147483648~2147483647
<code>unsigned[int]</code>	无符号整型	4	0~4294967295
<code>long[int]</code>	长整型	4	-2147483648~2147483647
<code>signed long[int]</code>	有符号长整型	4	-2147483648~2147483647
<code>unsigned long[int]</code>	无符号长整型	4	0~4294967295
<code>float</code>	单精度浮点型	4	约 6 位有效数字
<code>double</code>	双精度浮点型	8	约 12 位有效数字
<code>long double</code>	长双精度浮点型	16	约 15 位有效数字

 **环境栈空间**—保存函数调用返回时恢复运行所需要的信息。当一个函数被调用时，下面数据将被保存在环境栈中：

- ✓ 返回地址；
- ✓ 所有局部变量的值、递归函数的传值形式参数的值；
- ✓ 所有引用参数以及常量引用参数的定义（此部分参看附录 1）。
- **实例特征**：决定问题规模的那些因素，主要指输入和输出数据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， n 作为实例特征；两个 $m \times n$ 矩阵的加法，以 n 和 m 两个数作为实例特征。

注：一般情况下，指令空间的大小对于所解决的待定问题不够敏感。常量及简单变量所需要的数据空间也独立于所解决的问题，除非相关数的大小对于所选

定的数据类型来说实在太太。这时，要么改变数据类型要么使用多精度算法重写该程序，然后再对新程序进行分析。定长复合变量和未使用递归函数所需要的环境栈空间都独立于问题的规模。递归函数所需要的栈空间主要依赖于局部变量及形式参数所需要的空间。除此外，该空间还依赖于递归的深度（即嵌套递归调用的最大层次）。

- 程序所需要的空间可分为两部分：

$$S(P) = c + S_p(\text{实例特征})$$

- ✚ **固定部分 c**，它独立于实例的特征。主要包括指令空间、简单变量、常量以及定长复合变量所占用的空间；
- ✚ **可变部分 S_p** ，主要包括复合变量所需的空间（其大小依赖于所解决的具体问题）、动态分配的空间（依赖于实例的特征）、递归栈所需的空间（依赖于实例特征）。

注：一个精确的分析还应当包括编译期间所产生的临时变量所需的空间，这种空间与编译器直接相关联，在递归程序中除了依赖于递归函数外，还依赖于实例特征。但是，在考虑空间复杂性时，一般都被忽略。

程序 1-1-1 利用引用参数

```
template < class T >
T Abc(T& a, T& b, T& c)
{
    return a+b+c+b*c\
        +(a+b+c)/(a+b)+4;
}
```

在程序 1-1-1 中，采用数据类型 T 作为实例特征。由于 a, b, c 都是引用参数，在函数中不需要为它们的值分配空间，但需保存指向这些参数的指针。若每个指针需要 2 个字节，则共需要 6 字节的指针空间。此时函数所需要的总空间是一个常量，而 $S_{\text{Abc}}(\text{实例特征})=0$ 。

若函数 Abc 的参数是传值参数，则每个参数需要分配 $\text{sizeof}(T)$ 的空间，于是 a, b, c 所需的空间为 $3 \times \text{sizeof}(T)$ 。函数 Abc 所需的其他空间都与 T 无关，故 $S_{\text{Abc}}(\text{实例特征})=3 \times \text{sizeof}(T)$ 。

程序 1-1-2 顺序搜索

```
template < class T >
int SeqSearch(T a[], const T& x, int n)
{
```

```

//在 a[0:n-1]中搜索 x, 若找到则返回所在的位置, 否则返回-1
int i;
for (i=0; i<n && a[i]!=x; i++);
if (i==n) return -1;
return i;
}

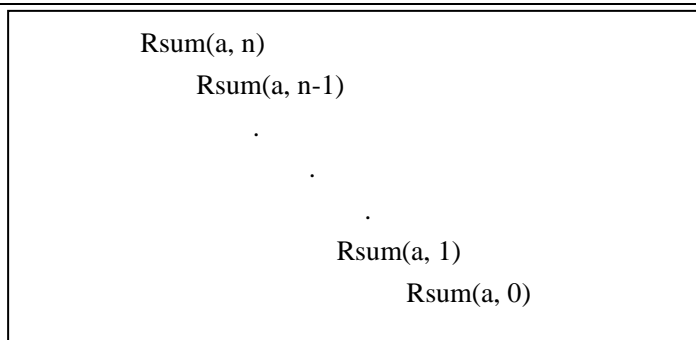
```

在程序 1-1-2 中, 假定采用被查数组的长度 n 作为实例特征, 并取 T 为 `int` 类型。数组中每个元素需要 4 个字节, 实参 x 需要 4 个字节, 传值形式参数 n 需要 4 个字节, 局部变量 i 需要 4 个字节, 整数常量 -1 需要 2 个字节, 所需要的总的空间为 18 个字节, 其独立于 n , 所以 $S_{\text{SeqSearch}}(n)=0$ 。这里, 我们并未把数组 a 所需要的空间计算进来, 因为该数组所需要的空间已在定义实际参数 (对应于 a) 的函数 (如主函数) 中分配, 不能再加到函数 `SeqSearch` 所需要的空间上去。

再比较下面两个程序, 它们都是求已知数组中元素的和。在第一个程序中采用累加的办法, 而在第二个程序中则采用递归的办法。我们取被累加的元素个数 n 作为实例特征。在第一个函数中, a , n , i 和 $tsum$ 需要分配空间, 与上例同样的理由, 程序所需的空间与 n 无关, 因此, $S_{\text{Sum}}(n)=0$ 。在第二个程序中, 递归栈空间包括参数 a , n 以及返回地址。对于 a 需要保留一个指针, 对于 n 则需要保留一个 `int` 类型的值。如果是 `near` 指针 (需占用 2 个字节), 则返回地址需占用 2 个字节, n 需占用 4 个字节, 那么每次调用 `Rsum` 函数共需 8 个字节。该程序递归深度为 $n+1$, 所以需要 $8(n+1)$ 字节的递归栈空间。 $S_{\text{Rsum}}(n)=8(n+1)$ 。

累加 a[0:n-1]	递归计算 a[0:n-1]
<pre> template<class T> T Sum(T a[], int n) {//计算 a[0:n-1]的和 T tsum=0; for (int i=0; i<n; i++) tsum+=a[i]; return tsum; } </pre>	<pre> template<class T> T Rsum(T a[], int n) {//计算 a[0:n-1]的和 if (n>0) return Rsum(a, n-1)+a[n-1]; return 0; } </pre>

嵌套调用一直进行到 $n=0$ 。可见, 递归计算比累加计算需要更多的空间。



上述递归程序的嵌套调用层次

§ 2 时间复杂性

● 时间复杂性的构成

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，一般情况下，一次编译过的程序可以运行若干次，所以，人们主要关注的是运行时间，记做 t_p (实例特征)；如果了解所用编译器的特征，就可以确定程序 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（这里主要是问题实例的规模），则有如下的表示式：

$$t_p(n) = c_a \text{ADD}(n) + c_s \text{SUB}(n) + c_m \text{MUL}(n) + c_d \text{DIV}(n) + c_c \text{CMP}(n) + \dots$$

其中， c_a ， c_s ， c_m ， c_d ， c_c 分别表示一次加、减、乘、除及比较操作所需的时间，函数 ADD ， SUB ， MUL ， DIV ， CMP 分别表示代码 P 中所使用的加、减、乘、除及比较操作的次数；

注：一个算术操作所需的时间取决于操作数的类型（`int`、`float`、`double` 等等），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。估算运行时间的方法主要有两种：A. 选一种或多种关键操作，确定这些关键操作所需要的执行时间（对于前面所列举的四种算术运算及比较操作，一般被看作是基本操作，并约定所用的时间都是一个单位）及每种关键操作执行的次数；B. 确定程序总的执行步数。

● 操作计数

首先选择一种或多种操作（如加、乘和比较等），然后计算这些操作分别执行了多少次。关键操作对时间复杂性影响最大。

程序 1-2-1 寻找最大元素

```

template<class T>
int Max(T a[], int n)
{//寻找 a[0:n-1] 中的最大元素

```

```

int pos=0;
for (int i=1; i<n; i++)
    if (a[pos]<a[i])
        pos=i;
return pos;
}

```

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，则操作计数将增加到一个常数倍。

程序 1-2-2 n 次多项式求值

```

template<class T>
T PolyEval(T coeff[], int n, const T& x)
{//计算 n 次多项式的值，coeff[0:n]为多项式的系数
T y=1, value=coeff[0];
for (int i=1; i<=n; i++)    //n 循环
    {
        //累加下一项
        y*=x;                //一次乘法
        value+=y*coeff[i];    //一次加法和一次乘法
    }
return value;
}                                //3n 次基本运算

```

Horner 法则：

$$P(x) = (\dots (c_n * x + c_{n-1}) * x + c_{n-2}) * x + c_{n-3}) * x + \dots * x + c_0$$

程序 1-2-3 利用 Horner 法则求多项式的值

```

template<class T>
T Horner(T coeff[], int n, const T& x)
{//计算 n 次多项式的值，coeff[0:n]为多项式的系数
T value=coeff[n];
for(i=1; i<=n; i++)    //n 循环
    T value=value*x+coeff[n-i]; //一次加法和一次乘法

```

```
    return value;
}
```

//2n 次基本运算

这里，关键操作是加法与乘法运算。在程序 1-2-2 中，for 循环的每一回都执行两次乘法运算和一次加法运算。所以程序的总的运算次数为 $3n$ 。在程序 1-2-3 中，for 循环的每一回都执行乘法与加法运算各一次，程序总的运算次数为 $2n$ 。再考察下面两例：

程序 1-2-4 计算名次

```
template<class T>
void Rank(T a[], int n, int r[])
{//计算 a[0:n-1]中元素的排名
    for(int i=0; i<n; i++)
        r[i]=0; //初始化
    //逐对比较所有的元素
    for(int i=1; i<n; i++)
        for(int j=0; j<i; j++)
            if(a[j]<=a[i]) r[i]++;
            else r[j]++;
}
```

在这两个程序中，关键操作都是比较。在 1-2-4 中，对于每个 i ，执行比较的次数是 i ，总的比较次数为 $1+2+\dots+n-1=(n-1)n/2$ 。

(在此，for 循环的额外开销、初始化数组 r 的开销、以及每次比较 a

中两个素时对 r 进行的增值开销都未列入其中。在程序 1-2-5 给出的排序中，首先找出最大元素，把它移到 $a[n-1]$ ，然后在余下的 $n-1$ 个元素中再寻找最大的元素，并把它移到 $a[n-2]$ 。如此进行下去，此种方法称为选择排序(SelectSort)。从程序 1-2-1 中已经知道，每次调用 $\text{Max}(a, \text{size})$ 需要执行 $\text{size}-1$ 次比较，所以总的比较次数为 $1+2+\dots+n-1=(n-1)n/2$ 。每调用一次函数 Swap 需要执行三元素移动，所需要总的移动次数为 $3(n-1)$ 。

考虑**冒泡排序**(SubbleSort)。冒泡排序是从左向右逐个检查，对相邻的元素进行比较，若左边的元素比右边的元素大，则交换这两个元素的位置。如数组

程序 1-2-5 选择排序

```
template<class T>
void SelectSort(T a[], int n)
{//对数组 a[0:n-1]中元素排序
    for(int size=n; size>1; size--)
        { int j=Max(a, size);
          Swap(a[j], a[size-1]);
        }
}
```

其中函数 Max 定义如程序 1-2-1
而函数 Swap 由下述程序给出
程序 1-2-6 交换两个值

```
template<class T>
inline void Swap(T& a, T& b)
{//交换 a 和 b
    T temp=a; a=b; b=temp;
}
```

[5, 3, 7, 4, 11, 2]: 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。这个行程称为一次冒泡, 经一次冒泡后, 原数组变为[3, 5, 4, 7, 2, 11]. 可见, 数组中最大的数被移动到最右的位置。下一次冒泡只对数组[3, 5, 4, 7, 2]进行即可。如此进行下去, 最后将原数组按递增的顺序排列。

程序 1-2-7 一次冒泡

```
template<class T>
void Bubble(T a[], int n)void
{
    for(int i=0; i<n-1; i++)
        if(a[i]>a[i+1]) Swap(a[i], a[i+1]);
}
```

程序 1-2-8 冒泡排序

```
Template<class T>
BubbleSort(T a[], int n)
{
    for(int i =n; i >1; i --)
        Bubble(a, i);
}
```

在程序 1-2-7 中, for 循环的每一回都执行了一次比较和至多三次元素的移动, 因而总的操作数为 $4(n-1)$ 。在程序 1-2-8 中, 对于每个 i , 调用函数 $Bubble(a, i)$ 需要执行 $4(i-1)$ 次操作, 因而总的操作数为 $2n(n-1)$ 。这里我们照例忽略了非关键操作。

分析程序 1-2-7 发现, 在问题实例中, 如果数组本身是递增的, 则每一次冒泡都不需要交换数据, 而如果数组是严格递减的, 则第一次冒泡要交换数据 $n-1$ 次。这里我们都取数组元素个数为实例特征, 但操作数却是不同的。因而, 人们往往还会关心最好的、最坏的和平均的操作数是多少。令 P 表示一个程序, 将操作计数 $O_P(n_1, n_2, \dots, n_k)$ 视为实例特征 n_1, n_2, \dots, n_k 的函数。令 $operation_P(I)$ 表示程序对实例 I 的操作计数, $S(n_1, n_2, \dots, n_k)$ 为所讨论程序的具有实例特征 n_1, n_2, \dots, n_k 的实例之集, 则

P 最好的操作计数为

$$O_P^{BC}(n_1, n_2, \dots, n_k) = \min\{operation_P(I) \mid I \in S(n_1, n_2, \dots, n_k)\}$$

P 最坏的操作计数为

$$O_P^{WC}(n_1, n_2, \dots, n_k) = \max\{operation_P(I) \mid I \in S(n_1, n_2, \dots, n_k)\}$$

P 平均的操作计数为

$$O_P^{AVG}(n_1, n_2, \dots, n_k) = \frac{1}{|S(n_1, n_2, \dots, n_k)|} \sum_{I \in S(n_1, n_2, \dots, n_k)} operation_P(I)$$

P 操作计数的期望值为

$$O_p^{AVG}(n_1, n_2, \dots, n_k) = \sum_{I \in S(n_1, n_2, \dots, n_k)} p(I) \cdot \text{operation}_p(I)$$

其中, $p(I)$ 是可被成功解决的实例 I 出现的概率。

在前面的例子中, 一般计算的都是程序的最坏操作计数。如在冒泡程序 Bubble 中即是如此。在顺序搜索 SeqSearch($T\ a[], \text{const } T\ \&x, \text{int } n$) 中, 取 n 作为实例特征, 关键操作数是比较。此时, 比较的次数并不是由 n 唯一确定的。若 $n=100$, $x=a[0]$, 那么仅需要执行一次操作; 若 x 不是 a 中的元素, 则需要执行 100 次比较。当 x 是 a 中一员时称为成功查找, 否则称为不成功查找。每回不成功的查找, 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, 则平均比较次数为

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

再考察插入排序算法:

程序 1-2-9 向有序数组插入元素

```
template<class T>
void Insert(T a[], int& n, const T& x)
{ //向数组 a[0:n-1]中插入元素 x
  //假定 a 的大小超过 n
  int i;
  for(i=n-1; i>=0 && x<a[i]; i--)
    a[i+1]=a[i];
  a[i+1]=x;
  n++; //添加了一个元素
}
```

程序 1-2-10 插入排序

```
template<class T>
void InsertSort(T a[], int n)
{ //对 a[0:n-1]进行排序
  for(int i=1; i<n; i++) {
    T t =a[i];
    Insert(a, i, t);
  }
}
```

在程序 1-2-9 中假定: a 中元素在 x 被插入前后都是按递增的顺序排列的。我们取初始数组 a 的大小 n 作为实例特征, 则程序 1-2-9 的关键操作是 x 与 a 中元素的比较。显然, 最少的比较次数为 1, 这种情况发生在 x 被插在数组尾部的时候; 最多的比较次数是 n , 发生在 x 被插在数组的首部或者数组的第一元素之后的时候。如果 x 最终被插在 a 的 $i+1$ 处 ($i \geq 0$), 则执行的比较次数是 $n-i$ 。如果 x 被插在 $a[0]$ 之前的位置, 则比较的次数为 n 。假定 x 有相等的机会被插在任何一个可能的位置上, 则平均比较次数是

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

在程序 1-2-10 中所执行的比较次数, 最好情况下是 $n-1$, 最坏情况下是 $n(n-1)/2$ 。

虽然在这些简单例子中，我们都给出了平均操作数，但是在一般情况下，平均操作数不是很容易求得的，操作数的数学期望值就更不容易求得了。

● 执行步数

利用操作计数方法估计程序的时间复杂性注意力集中在“关键操作”上，忽略了所选择操作之外其他操作的开销。下面所要讲的统计执行步数(step-count)的方法则要统计程序/函数中所有部分的时间开销。执行步数也是实例特征的函数，通常做法是选择一些感兴趣的实例特征。如，若要了解程序的运行时间是如何随着输入数据的个数增加而增加的，则把执行步数仅看作输入数据的个数的函数。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;` 都可以作为程序步。一种直观的统计程序执行步数的方法是做执行步数统计表

矩阵加法程序执行步数统计表

语 句	s/e	频率	总步数
Void Addm(T **a, ...)	0	0	0
{	0	0	0
for(int i=0; i<rows; i++)	1	rows+1	rows+1
for(int j=0; j<cols; j++)	1	rows*(cols+1)	rows*cols+rows
c[i][j]=a[i][j]+b[i][j];	1	rows*cols	rows*cols
}	0	0	0
总 计			2*rows*cols+2*rows+1

其中 s/e 表示每次执行该语句所要执行的程序步数。一条语句的 s/e 就等于执行该语句所产生的 count 值的变化量。频率是指该语句总的执行次数。

实际统计中，可以通过创建全局变量 count 来确定一个程序或函数为完成其预定任务所需要的执行步数。将 count 引入到程序语句之中，每当原始程序或函数中的一条语句被执行时，就为 count 累加上该语句所需要的执行步数。

程序 1-2-11 矩阵加法与执行步数统计

```
template<class T>
void Addm(T **a, T **b, T **c, int rows, int cols)
{//矩阵 a 和 b 相加得到矩阵 c
    for(int i=0; i<rows; i++){
        count++; //对应于上一条 for 语句 （共执行了 rows 步）
```

```
for(int j=0; j<cols; j++){
    count++; //对应于上一条 for 语句(共执行了 rows×cols 步)
    c[i][j]=a[i][j]+b[i][j];
    count++; //对应于赋值语句          (共执行了 rows×cols 步)
}
count++; //对应 j 的最后一次 for 循环 (共执行了 rows 步)
}
count++; //对应 i 的最后一次 for 循环 (共执行了 1 步)
}
```

若取 rows 和 cols 为实例特征,则,程序 1-2-11 总的执行步数为 $2 \times \text{rows} \times \text{cols} + 2 \times \text{rows} + 1$ 。据此,若 $\text{rows} > \text{cols}$, 我们可以通过交换程序中 i 循环与 j 循环的次序而使程序所用时间减少。

§ 3 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着实例特征变化的变化量。设 $T(n)$ 是算法 A 的复杂性函数。一般说来,当 n 单调增加趋于 ∞ 时, $T(n)$ 也将单调增加趋于 ∞ 。如果存在函数 $\tilde{T}(n)$, 使得当 $n \rightarrow \infty$ 时有 $(T(n) - \tilde{T}(n)) / T(n) \rightarrow 0$, 则称 $\tilde{T}(n)$ 是 $T(n)$ 当 $n \rightarrow \infty$ 时的渐近性态, 或称 $\tilde{T}(n)$ 是算法 A 当 $n \rightarrow \infty$ 的渐近复杂性。因为在数学上, $\tilde{T}(n)$ 是 $T(n)$ 当 $n \rightarrow \infty$ 时的渐近表达式, $\tilde{T}(n)$ 可以是 $T(n)$ 中略去低阶项所留下的主项, 所以它无疑比 $T(n)$ 来得简单。

进一步分析可知,要比较两个算法的渐近复杂性,只要确定出各自的阶即可知道哪一个算法的效率高。换句话说,渐近复杂性分析只要关心 $\tilde{T}(n)$ 的阶就够了,不必关心包含在 $\tilde{T}(n)$ 中的常数因子。所以,对 $T(n)$ 的分析又常常做进一步简化,即假设算法中用到的所有不同的运算(基本)各执行一次所需要的时间都是一个单位时间。

综上分析,我们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,首先给出常用的渐近函数。

常用的渐进函数

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 倍 $\log n$	$n!$	阶乘

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 (一般是输入规模) 的函数。由于一个程序的时间或空间需求是一个非负的实数, 我们假定函数 $f(n)$ 对于 n 的所有取值均为非负实数, 而且还可假定 $n \geq 0$ 。

渐近符号 O 的定义: $f(n) = O(g(n))$ 当且仅当存在正的常数 c 和 n_0 , 使得对于所有的 $n \geq n_0$, 有 $f(n) \leq cg(n)$ 。此时, 称 $g(n)$ 是 $f(n)$ 的一个渐近上界。

函数 f 至多是函数 g 的 c 倍, 除非 $n < n_0$ 。即是说, 当 n 充分大时, 在不计较相差一个非零常数倍的情况下, g 是 f 的一个上界函数。通常情况下, 上界函数取单项的形式, 如表 1 所列。

$C_0 = O(1)$: $f(n)$ 等于非零常数的情形。

$3n+2 = O(n)$: 可取 $c=4, n_0=2$ 。

$100n+6 = O(n)$: 可取 $c=101, n_0=6$ 。

$10n^2+4n+3 = O(n^2)$: 可取 $c=11, n_0=5$ 。

$6 \times 2^n + n^2 = O(2^n)$: 可取 $c=7, n_0=4$ 。

$3 \times \log n + 2 \times n + n^2 = O(n^2)$. 可取 $c=2, n_0=5$ 。

$n \times \log n + n^2 = O(n^2)$. 可取 $c=2, n_0=3$ 。

$3n+2 = O(n^2)$. 可取 $c=1, n_0=3$ 。

三点注意事项:

1. 用来作比较的函数 $g(n)$ 应该尽量接近所考虑的函数 $f(n)$ 。

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

2. 不要产生错误界。

$n^2+100n+6$, 当 $n < 3$ 时, $n^2+100n+6 < 106n$, 由此就认为 $n^2+100n+6=O(n)$, 这是不对的。事实上, 对任何正的常数 c , 只要 $n > c-100$ 就有 $n^2+100n+6 > c \times n$ 。同理, $3n^2+4 \times 2^n=O(n^2)$ 是错误的界。

3. $f(n)=O(g(n))$ 不能写成 $g(n)=O(f(n))$, 因为两者并不等价。

实际上, 这里的等号并不是通常相等的含义。按照定义, 用集合符号更准确些:
 $O(g(n)) = \{f(n) \mid f(n) \text{ 满足: 存在正的常数 } c \text{ 和 } n_0, \text{ 使得当 } n \geq n_0 \text{ 时, } f(n) \leq cg(n)\}$
 所以, 人们常常把 $f(n) = O(g(n))$ 读作: “ $f(n)$ 是 $g(n)$ 的一个大 O 成员”。

大 O 比率定理: 对于函数 $f(n)$ 和 $g(n)$, 如果极限 $\lim_{n \rightarrow \infty} (f(n)/g(n))$ 存在, 则
 $f(n) = O(g(n))$ 当且仅当存在正的常数 c , 使得 $\lim_{n \rightarrow \infty} (f(n)/g(n)) \leq c$ 。

例子 因为 $\lim_{n \rightarrow \infty} \frac{3n+2}{n} = 3$, 所以 $3n+2 = O(n)$;

因为 $\lim_{n \rightarrow \infty} \frac{10n^2+4n+2}{n^2} = 10$, 所以 $10n^2+4n+2 = O(n^2)$;

因为 $\lim_{n \rightarrow \infty} \frac{6 \cdot 2^n + n^2}{2^n} = 6$, 所以 $6 \cdot 2^n + n^2 = O(2^n)$;

因为 $\lim_{n \rightarrow \infty} \frac{n^{16} + 3 \cdot n^2}{2^n} = 0$, 所以 $n^{16} + 3 \cdot n^2 = O(2^n)$,

但是, 最后一个不是好的上界估计, 问题出在极限值不是正的常数。
 下述不等式对于复杂性阶的估计非常有帮助:

定理 1.3.1. 对于任意给定的正实数 c 、 x 和 ε , 有下面的不等式:

- 1) 存在某个 n_0 使得对于任何 $n \geq n_0$, 有 $(c \cdot \log n)^x < (\log n)^{x+\varepsilon}$ 。
- 2) 存在某个 n_0 使得对于任何 $n \geq n_0$, 有 $(\log n)^x < n$ 。
- 3) 存在某个 n_0 使得对于任何 $n \geq n_0$, 有 $(c+n)^x < n^{x+\varepsilon}$ 。
- 4) 存在某个 n_0 使得对于任何 $n \geq n_0$, 有 $n^x < 2^n$ 。
- 5) 对任意实数 y , 存在某个 n_0 使得对于任何 $n \geq n_0$, 有 $n^x (\log n)^y < n^{x+\varepsilon}$ 。

例子 根据定理 1, 我们很容易得出: $n^3 + n^2 \log n = O(n^3)$;
 $n^4 + n^{2.5} \log^{20} n = O(n^4)$; $2^n n^4 \log^3 n + 2^n n^5 / \log^3 n = O(2^n n^5)$ 。

符号 Ω 的定义: $f(n) = \Omega(g(n))$ 当且仅当存在正的常数 c 和 n_0 , 使得对于所有的 $n \geq n_0$, 有 $f(n) \geq c(g(n))$ 。此时, 称 $g(n)$ 是 $f(n)$ 的一个渐近下界。

函数 f 至少是函数 g 的 c 倍, 除非 $n < n_0$ 。即是说, 当 n 充分大时, 在不计较相差一个非零常数倍的情况下, g 是 f 的一个下界函数。类似于大 O 符号, 我

们可以参考定理 1.3.1 所列的不等式，来估计复杂性函数的渐近下界，而且有下述判定规则：

大 Ω 比率定理：对于函数 $f(n)$ 和 $g(n)$ ，如果极限 $\lim_{n \rightarrow \infty} (f(n)/g(n))$ 存在，则 $f(n) = \Omega(g(n))$ 当且仅当存在正的常数 c ，使得 $\lim_{n \rightarrow \infty} (f(n)/g(n)) \geq c$ 。

符号 Θ 的定义： $f(n) = \Theta(g(n))$ 当且仅当存在正的常数 c_1, c_2 和 n_0 ，使得对于所有的 $n \geq n_0$ ，有 $c_1(g(n)) \leq f(n) \leq c_2(g(n))$ 。此时，称 $f(n)$ 与 $g(n)$ 同阶。

函数 f 介于函数 g 的 c_1 和 c_2 倍之间，即当 n 充分大时，在不计较相差非零常数倍的情况下， g 既是 f 的下界，又是 f 的上界。

例子 $3n+2 = \Theta(n)$; $10n^2+4n+2 = \Theta(n^2)$ 。

$$5 \times 2^n + n^2 = \Theta(2^n);$$

Θ 比率定理：对于函数 $f(n)$ 和 $g(n)$ ，如果极限 $\lim_{n \rightarrow \infty} (g(n)/f(n))$ 与 $\lim_{n \rightarrow \infty} (f(n)/g(n))$ 都存在，则 $f(n) = \Theta(g(n))$ 当且仅当存在正的常数 c_1, c_2 ，使得 $\lim_{n \rightarrow \infty} (f(n)/g(n)) \leq c_1, \lim_{n \rightarrow \infty} (g(n)/f(n)) \leq c_2$ 。

例子： $t_{sum}(n) = 2n+3 = \Theta(n)$; $t_{Add}(m, n) = 2mn+2n+1 = \Theta(mn)$;

$t_{SeqSearch}^{AVG}(n) = \alpha(n+7)/2 + (1-\alpha)(n+3) = \Theta(n)$ ，其中 α 表示被查元素 x 在数组 a 中的概率。

比较大 O 比率定理和 Ω 比率定理，可知， Θ 比率定理实际是那两种情况的综合。对于多项式情形的复杂性函数，其阶函数可取该多项式的最高项，即

定理 1.3.2. 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

$$f(n) = O(n^m), \quad f(n) = \Omega(n^m), \quad f(n) = \Theta(n^m)$$

一般情况下，我们不能对每个复杂性函数直接去估计它们的渐进上界、渐进下界和“双界”，定理 1.3.1 和定理 1.3.2 给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。由这些信息可以给出多个函数经过加、乘运算组合起

来的复杂函数的阶的估计。

小 o 符号定义: $f(n) = o(g(n))$ 当且仅当 $f(n) = O(g(n))$ 和 $g(n) \neq O(f(n))$ 。

一般情况下, $f(n) = o(g(n))$ 的充分必要条件是 $\lim_{n \rightarrow \infty} (f(n)/g(n)) = 0$ 。

最后给出折半搜索程序及算法复杂性估计, 这里假定被查找的数组已经是单调递增的。

程序 1-3-1 折半搜索

```
template<class T>
int BinarySearch(T a[], const T& x, int n)
{ // 在 a[0] ≤ a[1] ≤ ··· ≤ a[n-1] 中搜索 x
  // 如果找到, 则返回所在位置, 否则返回 -1
  int left=0; int right=n-1;
  while(left ≤ right) {
    int middle=(left+right)/2;
    if(x==a[middle]) return middle;
    if(x>a[middle]) left=middle+1;
    else right=middle - 1;
  }
  return -1; // 未找到 x
}
```

while 的每次循环 (最后一次除外) 都将以减半的比例缩小搜索范围, 所以, 该循环在最坏的情况下需要执行 $\Theta(\log n)$ 次。由于每次循环需耗时 $\Theta(1)$, 因此, 在最坏情况下, 总的时间复杂性为 $\Theta(\log n)$ 。

习题 一

1. 试确定下述程序的执行步数, 该函数实现一个 $m \times n$ 矩阵与一个 $n \times p$ 矩阵之间的乘法:

矩阵乘法运算

```
template<class T>
void Mult(T **a, T **b, int m, int n, int p)
{ //  $m \times n$  矩阵 a 与  $n \times p$  矩阵 b 相成得到  $m \times p$  矩阵 c
```

```

for(int i=0; i<m; i++)
    for(int j=0; j<p; j++){
        T sum=0;
        for(int k=0; k<n; k++)
            Sum+=a[i][k]*b[k][j];
        C[i][j]=sum;
    }
}

```

2. 函数 MinMax 用来查找数组 $a[0:n-1]$ 中的最大元素和最小元素，以下给出两个程序。令 n 为实例特征。试问：在各个程序中， a 中元素之间的比较次数在最坏情况下各是多少？

找最大最小元素 （方法一）

```

template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{//寻找 a[0:n-1] 中的最小元素与最大元素
    //如果数组中的元素数目小于 1，则还回 false
    if(n<1) return false;
    Min=Max=0; //初始化
    for(int i=1; i<n; i++){
        if(a[Min]>a[i]) Min=i;
        if(a[Max]<a[i]) Max=i;
    }
    return true;
}

```

找最大最小元素 （方法二）

```

template<class T>
bool MinMax(T a[], int n, int& Min, int& Max)
{//寻找 a[0:n-1] 中的最小元素与最大元素
    //如果数组中的元素数目小于 1，则还回 false
    if(n<1) return false;
    Min=Max=0; //初始化
    for(int i=1; i<n; i++){
        if(a[Min]>a[i]) Min=i;

```



```

    else if(a[Max]<a[i]) Max=i;
    }
    return true;
}

```

3. 证明以下关系式不成立:

1). $10n^2 + 9 = O(n)$;

2). $n^2 \log n = \Theta(n^2)$;

4. 证明: 当且仅当 $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ 时, $f(n) = o(g(n))$ 。

5. 下面那些规则是正确的? 为什么?

1). $\{f(n) = O(F(n)), g(n) = O(G(n))\} \Rightarrow f(n)/g(n) = O(F(n)/G(n))$;

2). $\{f(n) = O(F(n)), g(n) = O(G(n))\} \Rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$;

3). $\{f(n) = O(F(n)), g(n) = O(G(n))\} \Rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$;

4). $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \Rightarrow f(n)/g(n) = \Omega(F(n)/G(n))$;

5). $\{f(n) = \Omega(F(n)), g(n) = \Omega(G(n))\} \Rightarrow f(n)/g(n) = O(F(n)/G(n))$ 。

6). $\{f(n) = \Theta(F(n)), g(n) = \Theta(G(n))\} \Rightarrow f(n)/g(n) = \Theta(F(n)/G(n))$

6. 按照渐近阶从低到高的顺序排列以下表达式:

$$4n^2, \log n, 3^n, 20n, n^{2/3}, n!$$

7. 1) 假设某算法在输入规模是 n 时为 $T(n) = 3 \cdot 2^n$. 在某台计算机上实现并完成该算法的时间是 t 秒. 现有另一台计算机, 其运行速度为第一台的 64 倍, 那么, 在这台计算机上用同一算法在 t 秒内能解决规模为多大的问题?

2) 若上述算法改进后的新算法的时间复杂度为 $T(n) = n^2$, 则在新机器上用 t 秒时间能解决输入规模为多大的问题?

3) 若进一步改进算法, 最新的算法的时间复杂度为 $T(n) = 8$, 其余条件不变, 在新机器上运行, 在 t 秒内能够解决输入规模为多大的问题?

8. Fibonacci 数有递推关系:

$$F(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

试求出 $F(n)$ 的表达式。