第2章

程序性能分析

本章内容

- 空间复杂性
  - 组成
  - 分析
- 时间复杂性
  - 组成
  - 分析
- 简单的搜索(查找)方法、排序方法及矩阵运算等
  - 搜索方法：顺序搜索、折半搜索
  - 排序方法：按名次排序、选择排序、冒泡排序、插入排序
  - 矩阵运算：相加、转置、相乘

2.1 程序性能

- **程序性能(program performance)**，是指运行一个程序所需要的内存大小和时间。
  - 空间复杂性
  - 时间复杂性

- 确定一个程序的性能有两种方法。
  - 分析的方法
    - 进行性能分析
  - 实验的方法
    - 进行性能测量

2.2 空间复杂性

- **空间复杂性(Space complexity)**
  - 是指运行完一个程序所需要的内存大小。
- 程序所需要的空间构成 ：
  - 指令空间(Instruction space)
  - 数据空间(Data space)
  - 环境栈空间(Environment stack space)

空间复杂性组成—指令空间

- **指令空间**
  - 指令空间是指用来存储经过编译之后的程序指令所需的空间。

- 程序所需要的指令空间的数量取决于如下因素：
  - 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。

计算表达式a+b+b*c+(a+b-c)/(a+b)+4 的代码

- P40 图2-1 不同的编译器将产生不同的程序代码

```
LOAD    a
ADD     b
STORE   t1
LOAD    b
MULT    c
STORE   t2
LOAD    t1
ADD     t2
STORE   t3
LOAD    a
ADD     b
SUB     c
STORE   t4
LOAD    a
ADD     b
STORE   t5
LOAD    t4
DIV     t5
STORE   t6
LOAD    t3
ADD     t6
ADD     4
```

a)

```
LOAD    a
ADD     b
STORE   t1
SUB     c
DIV     t1
STORE   t2
LOAD    b
MUL     c
STORE   t3
LOAD    t1
ADD     t3
ADD     t2
ADD     4
```

b)

```
LOAD    a
ADD     b
STORE   t1
SUB     c
DIV     t1
STORE   t2
LOAD    b
MUL     c
ADD     t2
ADD     t1
ADD     4
```

c)

空间复杂性组成—数据空间

- **数据空间：**数据空间是指用来存储所有常量和变量所需的空间。
  - 简单变量和常量；
  - 动态数组空间；
  - 动态类实例空间。

空间复杂性组成—数据空间

类型	空间大小（字节数）	范围
bool	1	{true,false}
char	1	[−128,127]
unsigned char	1	[0,255]
Short	2	[−32 768,32 767]
unsigned short	2	[0,65 535}
long	4	$[-2^{31},2^{31}-1]$
unsigned long	4	$[0,2^{32}-1]$
int	4	$[-2^{31},2^{31}-1]$
unsigned int	4	$[0,2^{32}-1]$
float	4	± 3.4 E ± 38(7 位）
double	8	± 1.7E ± 308（15 位）
long double	10	± 1.2E ± 4392（19 位）
pointer	2	(near,_cs,_ds,_es,_ss 指针）
pointer	4	(far,huge 指针）

Unit: 字节

图 2-2 32位计算机上 C++中简单变量占用空间(p41)

空间复杂性组成—环境栈

- **环境栈：**
  - 环境栈用来保存函数调用返回时恢复运行所需要的信息。
  - 函数被调用时，将被保存在环境栈中的数据：
    - **返回地址**。
    - 被调用的函数的所有**局部变量**的值以及形**式参数**的值(仅对于递归函数而言，与编译器有关)。
  - 实际使用的编译器将影响环境栈所需要的空间。

空间复杂性组成—环境栈

- **递归空间栈：**
  - 递归函数所需要的栈空间
  - 大小依赖于局部变量、形式参数以及递归的层次

```
rSum(a,n)
    rSum(a,n-1)
        rSum(a,n-2)
            .
            .
            .
                rSum(a,1)
                    rSum(a,0)
```

图 2-3　程序 1-31 的嵌套调用层次

空间复杂性组成—结论

- 无法精确地分析一个程序所需要的空间。
- 可以确定程序中某些部分的空间需求，这些部分**依赖于所解决问题实例的特征**。
- **实例特征**：这些特征包含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）
  - 例：
    - 对n个元素进行排序；
      - 实例特征：n
    - 两个n×n矩阵相加；
      - 实例特征：n
    - 两个m×n 矩阵相加：
      - 实例特征： m,n

空间复杂性度量

- $S(p)=c+S_p$(实例特征)
  - c: **固定部分**，它独立于实例的特征。
    - 包含指令空间(即代码空间)、简单变量及定长复合变量所占用空间、常量所占用空间等等。
  - S_p(实例特征): **可变部分**
    - 动态分配的空间（这种空间一般都依赖于实例的特征);
    - 递归栈空间(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。
      - 递归的最大深度(即嵌套递归调用的最大层次)。

空间复杂性度量

- 分析程序的空间复杂性：
  - 确定实例的特征
  - 估算Sp(实例特征)

例2-2 顺序查找

```
template<class T>
int sequentialSearch(T a[], const T& x, int n)
{
  // 在数组a[0:n-1]中查找x
 //如果找到，则返回元素所在位置，否则返回 -1 .
   int i;
   for (i=0; i < n && a[i] !=x; i++);
   if (i==n) return -1 ;
            else  return  i;
}
```

实例特征 :n

a, x, n, i, 0, -1: 需要空间与n无关

$S_{sequentialSearch}(n)=0$

例2-3 $S_{sum}(n)$

程序 1-30

```
template<class T>
T sum(T a[], int n)
 { //返回数组元素a[0]至a[n-1] 的和
   T theSum=0;
   for (int i = 0; i < n; i++)
         theSum += a[i];
   return  theSum;
 }
```

实例特征 :n

a, n, i, theSum , 0: 空间与n无关

$S_{sum}(n)=0$

例2-4 $S_{rSum}(n)$

程序1-31

```
template<class T>
T rSum(T a[], int n)
{
    if (n > 0)
        return  rSum(a, n-1) + a[n-1];
    return  0;
}
```

- 实例特征 :n
- a, n, 返回地址
  4, 4, 4 (字节)

嵌套调用的层次:

rSum(a, n)
rSum (a, n-1)
rSum (a, n-2)
……
rSum (a, 1)
rSum (a, 0)

递归的深度:
嵌套调用的最大层次

$S_{rSum}(n)=12(n+1)$

例2-6 $S_{permutations}(n)$

```
template<class T>
void permutations(T list[], int k, int m)
{ //生成list[k:m]的所有排列方式 , 输出前缀是 list[0:k-1] 后缀是 list[k:m]的所有排列方式
  int i;
  if (k = =m) {// list[k:m]只有一个排列
     copy(list,list+m+1,ostream_iterator<T>(cout, " "))  ;
          cout << endl;
       }
  else // list[k:m ]有多个排列方式，递归地产生这些排列方式
       for (i=k; i <= m; i++) {
            swap (list[k], list[i]);
            permutations (list, k+1, m);
            swap (list[k], list[i]);
           }
}
```

permutations(list[], 0, n-1)

$S_{permutations}(n) = ?$

填空题 2分

设置

```
int factorial(int n)
{ //计算 n!
   if (n<=1) return 1;
   else   return n*factorial(n-1);
}
```

递归深度为 [填空1]

Sfactorial (n) = [填空2]

作答

2.3 时间复杂性

1. 为了比较两个完成同一功能的程序的时间复杂性；
2. 为了预测随着实例特征的变化，程序运行时间的变化量。

2.3.1 时间复杂性的组成

- 影响一个程序空间复杂性的因素也都能影响程序的时间复杂性。

- T(p)=编译时间+运行时间
  - 编译时间与实例的特征无关
  - 运行时间通常用“t_p(**实例特征**)”来表示

- 估算运行时间的方法：
  - **操作计数**(operation counts)：找出一个或多个关键操作，确定这些关键操作所需要的执行时间；
  - **步数**(step counts)：确定程序总的步数。

操作计数

- **关键操作：**
  - 关键操作对时间复杂性的影响最大。
- **操作计数：**
  - 找出一个或多个关键操作，确定这些关键操作所需要的执行时间；

例2-7 最大元素

```
程序1-37
template<class T>
int indexOfMax(T a[], int n)
{//查找数组a[0:n]的最大元素
    if (n<=0)  throw illegalParameterValue("n must be > 0");
    int indexOfMax=0;
    for (int i=1; i < n; i++)
    if (a[indexOfMax] < a[i])
            indexOfMax=i;
    return indexOfMax;
}
```

实例特征：n；关键操作：比较

n<=0:比较次数0

n>1:比较次数 n-1

比较次数:max(0, n-1)

例2-8 多项式求值

$$P(x)=\sum_{i=0}^{n} C_i X^i$$

程序2-3

```
template <class T>
T  polyEval(T coeff[], int n, const T& x)
{//计算n阶多项式在x出的值， 系数为coeff[0:n]
  T y=1, value=coeff[0] ;
  for (int i=1; i≤n; i++)
   {
     y *=x;
     value +=y * coeff [ i ] ;
     }
   return value;
}
```

- 实例特征:n；关键操作：加法、乘法
  - 加法的次数为n，乘法的次数为2n。

例2-8 多项式求值

$$P(x)=\sum_{i=0}^{n} c_i x^i$$

Horner法则:

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

程序2-4

```
template <class T>
T  horner(T coeff[], int n, const T& x)
{//计算n阶多项式在x出的值，系数为coeff[0:n]
  T value=coeff[n] ;
  for(int i=1; i≤n; i++)
        value=value * x + coeff[n-i] ;
return value;
}
```

- 加法的次数为n，乘法的次数为n。

例2-9 名次计算

- 元素在序列中的名次(rank):
  - 序列中所有比它小的元素数目加上在它左边出现的与它相同的元素数目。
- 例如，
  - 序列：　　数组a=[4, 3, 9, 3, 7]
  - 各元素的名次为r =[2, 0, 4, 1, 3]

例2-9 名次计算

程序2-5

```
template <class T>
void rank(T a[], int n, int r[])
   { //给数组a[0:n-1]中的n个元素排名次
     //结果在r[0:n-1]中返回

   //初始化
   for (int i=0; i<n; i++)
         r[i]=0;

   //比较所有元素对
   for (int i=1; i<n; i++)
         for (int j=0; j<i; j++)
               if (a[j]≤a[i])   r[i]++ ;
               else    r[j]++;
   }
```

- **关键操作：**
  数组a的元素比较
- **比较次数：**
  $1+2+3+\ldots+n-1$
  $=(n-1)*n/2$

例2-10 按名次排序(计数排序)

- 程序2-6 利用附加数组的计数排序

```
template <class T>
void rearrange (T a[], int n, int r[])
{ //使用附加数组u, 将元素排序

//创建附加数组u
T *u=new T[n];

//数组a中的元素移动到u中名次对应的位置
for (int i=0; i < n; i++)
     u[r[i]]=a[i];

//数组u中的元素移动到a中
for (int i=0; i < n; i++)
    a [i]=u[i] ;

   delete [ ]u;
}
```

元素的移动次数： 2n

例2-15 按名次排序 (原地重排)

程序2-11

```
template<class T>
void rearrange(T a[], int n, int r[])
{ //  原地重排数组元素
for (int i=0; i < n; i++)
//把正确的元素移到a[i]处
while (r[i] !=i) {
      int t=r[i];
      Swap(a[i], a[t]);
      Swap(r[i], r[t]);
    }
}
```

- **比较次数：**

rank求r[]: (n-1)*n/2

共： (n-1)*n/2+n-1

- **交换次数:**

0------2(n-1)

- **移动次数:**

0------3*2*(n-1)

2.3.3 最好、最坏和平均操作计数

- **最好操作计数**:所有程序实例操作计数中，最小操作计数
- **最坏操作计数**:所有程序实例操作计数中，最大操作计数
- **平均操作计数**:

- 许多程序的平均操作计数不好确定，可以给出程序的最好、最坏操作计数

  - 例：程序2-11 原地重排数组元素，
  - **最少交换次数**:0；**最多交换次数**: 2(n-1)
  - **最少移动次数**:0；**最多移动次数**: 3*2*(n-1)

例2-13 顺序搜索

```
template<class T>
int sequentialSearch(T a[], const T& x, int n)
{ // 在数组a[0:n-1]中搜索x
  //如果找到，则返回该元素的位置, 否则返回-1 .
  int i;
  for (i=0; i < n && a[i] !=x; i++);
  if (i==n) return -1 ;
  else return  i;
}
```

顺序搜索时间复杂性

- 搜索(查找)：成功，不成功
- 不成功查找:比较次数:n
- 成功查找:
  - 最少的比较次数:1
  - 最多的比较次数:n
  - 平均比较次数:假定所有的数组元素都是不同的，并且每个元素 被查找的概率是相同的。

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

例2-11 选择排序

程序2-7 选择排序

```
template <class T>
void selectionSort(T a[], int n)
{//给数组a[0:n-1]的n个元素排序
  for (int size=n; size>1; size--)
    {
        int j=indexOfMax(a, size);
        swap(a[j] , a[size-1]);
     }
 }
```

- indexOfMax(a, size):
比较次数:size-1

- **选择排序时间复杂性:**
- **比较次数:**
1+2+3+...n-1=(n-1)*n/2
- **移动次数:** 3(n-1)

例2-16 选择排序(及时终止的)

```
template<class T>
void selectionSort(T a[], int n)
{//及时终止的选择排序
 bool sorted=false;
for (int size=n; !sorted && (size > 1); size--)
    {int indexOfMax=0;
     sorted=true;
     for (int i=1; i < size; i++)
     //找最大元素
     if (a[indexOfMax]≤a[i]) indexOfMax=i;
           else sorted=false;  //无序

     swap(a[indexOfMax], a[size - 1 ]) ;
     }
}
```

比较次数:

最好的情况： n-1

最坏的情况：

(n-1+n-2+…+3+2+1)

=(n-1)n/2

交换次数:

最好的情况： 1;

最坏的情况：(n-1)

例2-12 冒泡排序

程序2-8 一次冒泡

```
template <class T>
void bubble (T a[], int n)
{ //把数组a[0:n-1]中最大的
  //元素通过冒泡移到右边
for (int i=0; i<n-1; i++)
  if(a[i]>a[i+1])
       swap(a[i], a[i+1]) ;
}
```

程序2-9 冒泡排序

```
template <class T>
void bubbleSort (T a[], int n)
{ //对数组a[0:n-1]进行冒泡
   排序
  for (int i=n; i>1; i--)
       bubble(a , i) ;
}
```

比较次数:

(n-1+n-2+…+3+2+1)
=(n-1)n/2

交换次数:

最好的情况 :0
最坏的情况 : (n-1)n/2

例2-17　及时终止的冒泡排序

程序2-13

```
template<class T>
bool bubble(T a[], int n)
{ //把a[0:n-1]中最大元素冒泡至右端
 bool swapped=false; //尚未发生交换
 for (int i=0; i<n-1; i++)
 if (a[i] > a[i+1]) {
   swap(a[i], a[i+1]);
 swapped=true; //发生了交换
}
return swapped;
}
template<class T>
void bubbleSort(T a[], int n)
{//及时终止的冒泡排序
  for (int i=n; i>1 && bubble(a, i); i--) ;
}
```

比较次数:

最好的情况:　n-1

最坏的情况 :

(n-1+n-2+…+3+2+1)

=(n-1)n/2

交换次数:

最好的情况 :　　0

最坏的情况 :　(n-1)n/2

向有序数组中插入元素

- 向有序数组a[0:n-1]中插入一个元素。a中的元素在执行插入之前和插入之后都是按递增顺序排列的。
- 例如，向数组a[0:5] = [1, 2, 6, 8, 9, 11]中插入4，得到的结果为a[0:6]=[1, 2, 4, 6, 8, 9, 11]。
- 当为新元素找到欲插入的位置时，必须把该位置右边的所有元素分别向右移动一个位置。对于本例，需要移动11, 9, 8和6，并把4插入到新空出来的位置a[2]中。

例2-14 向有序数组中插入元素

程序2-10

```
template<class T>
 void insert(T a[], int n, const T& x)
 {//向有序数组a[0:n-1]中插入元素x ;
   //假设数组a 的容量大于n
      int i;
      for (i=n-1; i≥0 && x < a[i]; i--)
                a[i+1]=a[i];
      a[i+1]=x;
 }
```

向有序数组中插入元素

- **比较次数:**
- 最少的比较次数:1
- 最多的比较次数:n
- 平均比较次数:
  - 假定x 有相等的机会被插入到任一个可能的位置上(共有n+1个可能的插入位置)。
  - 如果x 最终被插入到a的i+1处，i≥0，则执行的比较次数为n-i。如果x 被插入到a[0]，则比较次数为n。

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

例2-18 插入排序

程序2-14

```
template<class T>
void insertionSort(T a[], int n)
{//对a[0:n-1]实施插入排序
    for (int i=1; i < n; i++) {
    T t=a[i];
    insert(a, i, t);
    }
}
```

比较次数:

最好的情况 :n-1

最坏的情况:n(n-1)/2

例2-18　另一种插入排序

```
程序2-15
template<class T>
void insertionSort(T a[], int n)
{ //对a[0:n-1]实施插入排序
    for (int i=1; i<n; i++) {
     //将a[i]插入a[0:i-1]
    T t=a[i];
    int j;
    for (j=i-1; j≥0 && t<a[j]; j--)
          a[j+1]=a[j];
    a[j+1]=t;
    }
}
```

- 最好情况：
  - 数组a最初已经有序。
- 最坏情况：
  - 数组a倒序。

- 比较次数：

最好的情况 :n-1

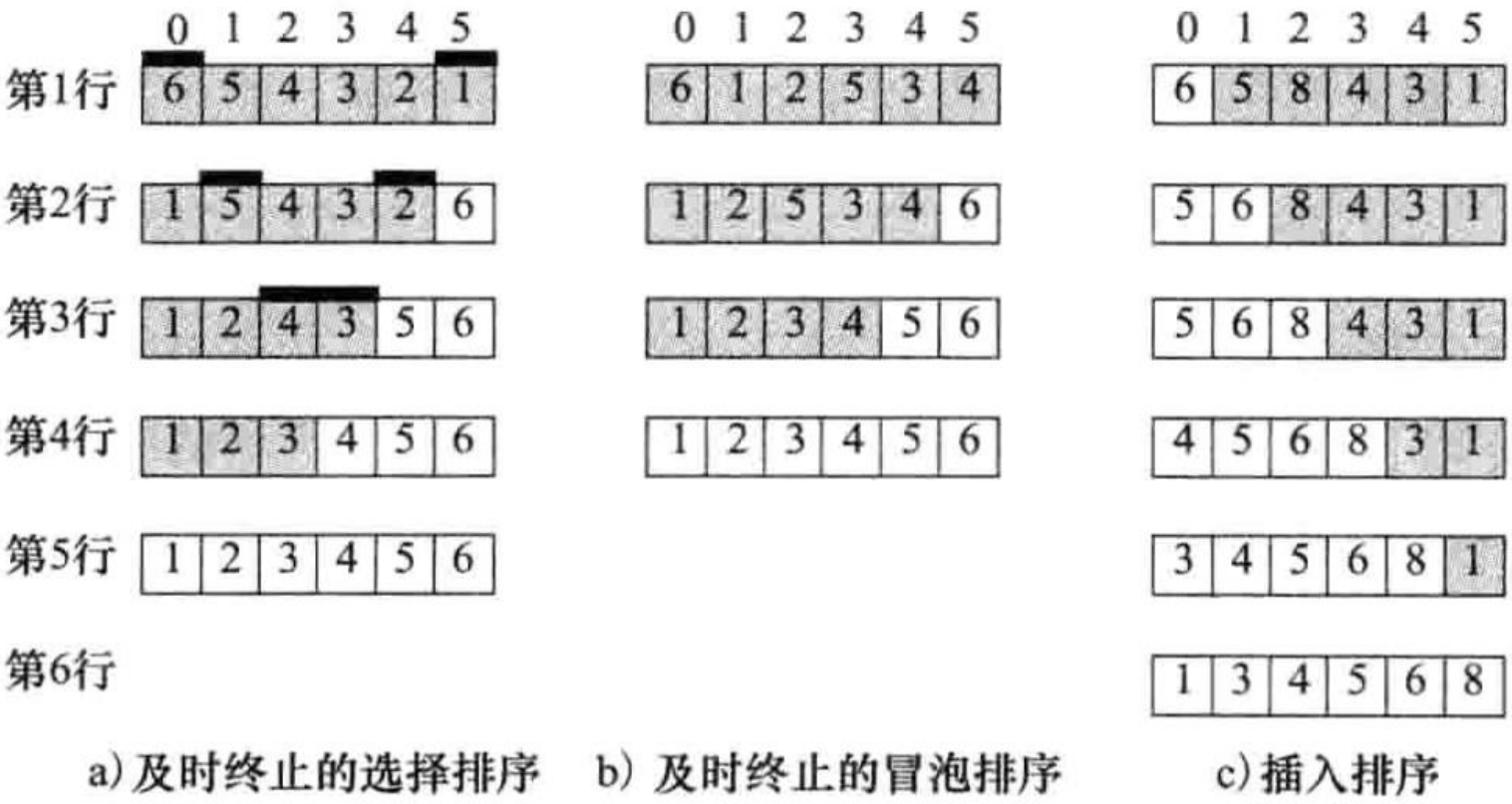
最坏的情况:n(n-1)/2

- 移动次数：

最好的情况 :2(n-1)

最坏的情况: 2(n-1)+n(n-1)/2

排序举例



a) 及时终止的选择排序　b) 及时终止的冒泡排序　c) 插入排序

排序算法时间复杂性比较

算法	最好	最坏
计数排序		
比较	n(n-1)/2+n	n(n-1)/2+n
移动	0	6(n-1)
选择排序		
比较	n-1	n(n-1)/2
移动	3	3(n-1)
冒泡排序		
比较	n-1	n(n-1)/2
移动	0	3*n(n-1)/2
插入排序		
比较	n-1	n(n-1)/2
移动	2(n-1)	2(n-1)+n(n-1)/2

课堂练习

14. 数组 a[0:8]=[g,h,i,f,c,a,d,b,e], 按名次排序结果 r[0:8]=[6,7,8,5, 2,0,3, 1,4]，请画出类似于图 2-5b 和图 2-5c 的图，显示原地重排函数（见程序 2-11）的排序过程。
15. 1）使用及时终止选择排序（见程序 2-12），对数组 a[0:9]=[9,8,7,6,5,4,3,2,1,0] 排序，画出类似于图 2-6a 的图，显示排序过程。

    2）对数组 a[0:8]=[8,4,5,2,1,6,7,3,0] 重复过程 1)。
16. 使用及时终止冒泡排序（见程序 2-13），对数组 a[0:9]=[4,2,6,7,1,0,9,8,5,3] 排序，画出类似于图 2-6b 的图，显示排序过程。
17. 使用插入排序（见程序 2-14），对数组 a[0:9]=[4,2,6,7,1,0,9,8,5,3] 排序，画出类似于图 2-6c 的图，显示排序过程。

步数 (Step counts)

- **操作计数方法**来估算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。
- **步数方法**，要统计程序/函数中所有操作部分的时间开销。

程序步(program step)可以定义为一个语法或语义意义上的程序片段，该片段的执行时间独立于实例特征。

- 100次加法，100次减法，1000次乘法可以视为一步.
- Return a+b+b*c+(a+b-c)/(a+b)+4可以视为一个程序步 .
- x=y可以视为一个程序步 .
- *n* 次加法不能视为一程序步.

确定步数方法1—stepCount

- 方法1：创建一个全局变量stepCount(其初值为0)，每当原始程序或函数中的一条语句被执行时，就为stepCount累加上该语句所需要的步数。

例： 2-19

```
template<class T>
T sum(T a[], int n)
    {//计算a[0:n-1]中元素之和
     T theSum=0;
     stepCount++; //对应于theSum=0
     for (int i=0; i<n; i++) {
          stepCount++; //对应于for循环的每一次条件判断语句
         theSum +=a[i];
          stepCount++; //对应于theSum +=a[i]
         }
     stepCount++; //对应于for循环的最后一个条件判断语句
     stepCount++; //对应于return语句
     return theSum;
     }
```

步数：2n+3

确定步数方法1 —stepCount

- 例： 2-19

```
//简化版本
template<class T>
T sum(T a[], int n)
    {
    for (int i=0; i<n; i++)
         stepCount+=2;
    stepCount+=3;
    Return 0;
    }
```

步数：2n+3

递归函数—stepCount

- 例： 2-20

```
template<class T>
T rSum(T a[], int n)
{  //计算a[0:n-1]中元素之和
   stepCount++; //对应于if条件
   if (n > 0) {stepCount++;//对应于return和rSum 调用
                return rSum(a, n-1)+a[n-1];}
   stepCount++; //对应于return.
   return 0;
}
```

$t_{rSum}(n)=2+t_{rSum}(n-1)$，$n>0$

$t_{rSum}(0)=2$

在分析一个递归程序的步数时，通常可以得到一个计算步数的递归公式，这种递归公式被称为递推方程或递推。

递归函数—stepCount

- 可以采用重复替换的方法来求解递推方程

$t_{rSum}(n)=2+t_{rSum}(n-1)$，$n>0$

$t_{rSum}(0)=2$

$$
\begin{aligned}
t_{rSum}(n) &= 2+t_{rSum}(n-1) \\
&= 2+2+t_{rSum}(n-2) \\
&= 4+t_{rSum}(n-2) \\
&\cdots \\
&= 2n+t_{rSum}(0) \\
&= 2(n+1)
\end{aligned}
$$

确定步数方法1—stepCount

- 比较程序sum和程序rSum的步数：
  - 程序rSum的步数2n+2
  - 程序sum的步数2n+3
- 不能因此断定程序sum就比程序rSum慢，
  - 因为程序步不代表精确的时间单位。
  - rSum中的一步可能要比sum中的一步花更多的时间
- **步数可用来帮助我们了解程序的执行时间是如何随着实例特征的变化而变化的。**
- Sum的运行时间随着n的增加线性增长(其时间复杂性与实例特征n呈线性关系)。

确定步数方法2—建立步数表

- 确定步数方法2：建立步数表

语句	s/e	频率	总步数
T sum(T a[], int n)	0	0	0
{	0	0	0
T theSum=0;	1	1	1
for(int i=0;i<n;i++)	1	n+1	n+1
theSum +=a[i];	1	n	n
return theSum;	1	1	1
}	0	0	0
总计			2n+3

⑩s/e:语句每次执行所需要的步数,即执行该语句所产生的stepCount值的变化量。

步数表—s/e是变化的

- 例：对数组前置元素求和：

$$b[j]=\sum_{i=0}^{j} a[i], \text{其中} j = 0,1,\ldots,n-1$$

语句	s/e	频率	总步数
void Inef(T a[], T b[], int n)	0	0	0
{			
for (int j = 0; j < n; j++)	1	n+1	n+1
b[j] = Sum(a, j+1);	2j+6	n	n(n+5)
}	0	0	0
总计			n^2+6n+1

- Sum(a, m)的执行步数：2m+3;
- Sum(a, j+1)的执行步数：2(j+1)+3;
- 总步数：$\sum_{0}^{n-1}(2j+6) = n(n+5)$

步数表—最好、最坏、平均步数

- 例： 2-19 矩阵转置

语句	s/e	频率	总步数
void transpose(T **a, int rows)	0	0	0
{	0	0	0
for (int i = 0; i < rows; i++)	1	*rows*+1	*rows*+1
for (int j = i+1; j < rows; j++)	1	*rows*(*rows* +1)/2	*rows*(*rows* +1)/2
swap(a[i][j], a[j][i]);	1	*rows*(*rows* –1)/2	*rows*(*rows* –1)/2
}	0	0	0
总计			$rows^2$+*rows* +1)

步数表—最好、最坏、平均步数

- 例： 2-21 顺序搜索

最好情况步数：

语句	s/e	频率	总步数
int sequentialSearch(T a[],T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (int i=0; i<n && a[i]!=x; i++)	1	1	1
if (i= =n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总计			4

步数表—最好、最坏、平均步数

- 例： 2-21 顺序搜索

最坏情况步数：

语句	s/e	频率	总步数
int sequentialSearch(T a[],T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (int i=0; i<n && a[i]!=x; i++)	1	n+1	n+1
if (i= =n) return -1;	1	1	1
return i;	1	0	0
}	0	0	0
总计			n+3

步数表—最好、最坏、平均步数

- 例： 2-21 顺序搜索

X=a[j]

语句	s/e	频率	总步数
int sequentialSearch(T a[],T& x, int n)	0	0	0
{	0	0	0
int i;	1	1	1
for (int i=0; i<n && a[i]!=x; i++)	1	j+1	j+1
if (i==n) return -1;	1	1	1
return i;	1	1	1
}	0	0	0
总计			j+4

步数表—最好、最坏、平均步数

假定：数组a中的n个值都互不相同；
查找过程是成功的；
x与数组中任何元素相匹配的概率都是一样的。

$$t^{AVG}_{SequentialSearch}(n) = \frac{1}{n}\sum_{j=0}^{n-1}(j+4) = (n+7)/2$$

假设：数组a中的n个值都互不相同；
成功查找出现的概率为80%；
每个a[i]被查找的机会相同.

平均步数 ：

=0.8*(n+7)/2+0.2*(n+3)

=0.6n+3.4

步数表—最好、最坏、平均步数

- 例： 2-10 插入函数

语句	s/e	频率	总步数
void insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = n-1; i >= 0 && x < a[i]; i--)	1	1	1
a[i+1] = a[i];	1	0	0
a[i+1] = x;	1	1	1
n++; // 一个元素插入 a	1	1	1
}	0	0	0
总计			4

语句	s/e	频率	总步数
void insert(T a[], int& n, const T& x)	0	0	0
{	0	0	0
int i;	1	1	1
for (i = n-1; i >= 0 && x < a[i]; i--)	1	n+1	n+1
a[i+1] = a[i];	1	n	n
a[i+1] = x;	1	1	1
n++; // 一个元素插入 a	1	1	1
}	0	0	0
总计			2n+4

步数表—最好、最坏、平均步数

假定：数组a中的n个值都互不相同；
x插入数据任一位置的概率都是一样的。
如果x插入的位置是j, j≥0. 则步数为2n-2j+4，
平均步数为n+4

$$\frac{1}{n+1}\sum_{j=0}^{n}(2n-2j+4)=\frac{1}{n+1}\left[2\sum_{j=0}^{n}(n-j)+\sum_{j=0}^{n}4\right]$$

$$=\frac{1}{n+1}\left[2\sum_{k=0}^{n}k+4(n+1)\right]$$

$$=\frac{1}{n+1}[n(n+1)+4(n+1)]$$

$$=\frac{(n+4)(n+1)}{n+1}$$

$$=n+4$$

课堂练习

23. 试确定函数 squareMatrixMultiply（见程序 2-22）在两个 $n \times n$ 矩阵相乘时执行了多少次乘法。

程序 2-22　两个 $n \times n$ 矩阵的乘法

```
template<class T>
void squareMatrixMultiply(T **a, T **b, T **c, int n)
{// 将 n x n 矩阵 a 和 b 相乘得到矩阵 c
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
      {
         T sum = 0;
         for (int k = 0; k < n; k++)
            sum += a[i][k] * b[k][j];
         c[i][j] = sum;
      }
}
```

课堂练习

24. 试确定函数 matrixMultiply（见程序 2-23）在实现一个 $m \times n$ 矩阵与一个 $n \times p$ 矩阵相乘时执行了多少次乘法。

程序 2-23　$m \times n$ 矩阵与 $n \times p$ 矩阵的乘法

```
template<class T>
void matrixMultiply(T **a, T **b, T **c, int m, int n, int p)
{// 将一个 m×n 矩阵 a 和一个 n×p 矩阵 b 相乘得到矩阵 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;
      }
}
```

课堂练习

26. 函数 minmax（见程序 2-24）是查找数组 a 的最大元素和最小元素。令 n 为实例特征。试问 a 的元素之间有多少次比较？程序 2-25 是另一个查找方法。在最好和最坏情况下的比较次数分别是多少？试分析两个函数之间的相对性能。