碁峰
www.gotop.com.tw

21世纪高校
计算机系列教材

数据结构
（Java版）

蔡明志　编著

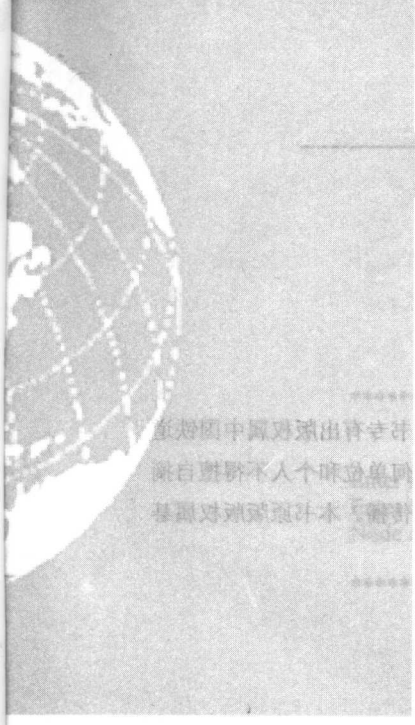
引进台湾原版成熟教材■

注重内容的实用性，培养学生的专业能力■

适合高校电子信息类各专业选用■

经过全国高等院校计算机基础教育研究会著名专家学者、教授的评估与审定■

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE



数据结构（Java 版）

蔡明志　编著

中国铁道出版社
CHINA RAILWAY PUBLISHING HOUSE

北京市版权局著作权合同登记 图字：01-2005-4561 号

版 权 声 明

图书在版编目（CIP）数据

数据结构：Java 版/蔡明志编著. —北京：中国铁道出版社，2006.6
（21 世纪高校计算系列教材）
ISBN 7-113-07197-X

Ⅰ.数… Ⅱ.蔡… Ⅲ.①数据结构—高等学校—教材②JAVA 语言—程序设计—高等学校—教材 Ⅳ.①TP311.12②TP312

中国版本图书馆 CIP 数据核字（2006）第 063403 号

书　　名：数据结构（Java 版）
作　　者：蔡明志
出版发行：中国铁道出版社（100054，北京市宣武区右安门西街 8 号）
策划编辑：严晓舟　秦绪好
责任编辑：苏　茜　谢立和　黄园园
封面制作：白　雪
责任校对：刘　洁
印　　刷：北京鑫正大印刷有限公司
开　　本：787×1092　1/16　**印张**：21.5　**字数**：528 千
版　　本：2006 年 7 月第 1 版　2006 年 7 月第 1 次印刷
印　　数：1～5 000 册
书　　号：ISBN 7-113-07197-X/TP・1911
定　　价：28.00 元

前　言

数据结构（Data Structure）是信息学科中的核心课程之一，也是基础和必修的科目，鉴于它的重要性，本书作者将在学校的教学讲义编辑成了本书。

本书作者从事了多年的数据结构教学，具有相当丰富的经验，了解应如何阐述数据结构的每一个主题，并尽可能地以图文并茂的方式表达，使其能达到事半功倍的效果。

传统数据结构的程序大部分以 C 或 C++语言编写，本书则以 Java 语言编写，主要是因为 Java 语言是面向对象的语言，而且又是跨平台的，近年来普遍受到用户的青睐，为了顺应这股潮流，书中也以 Java 程序来验证数据结构的一些重要问题。

本书在内容的编排上也费了一番心思。全书共分 13 章，分别为第 1 章算法分析、第 2 章数组、第 3 章栈与队列、第 4 章链表、第 5 章递归、第 6 章树结构、第 7 章堆结构、第 8 章平衡二叉查找树、第 9 章 2-3 树与 2-3-4 树、第 10 章 B 树、第 11 章图结构、第 12 章排序及第 13 章查找。

在本书的编写上，尽量以简单易懂的方式来进行说明，这区别于一般市面上的“翻译书”。因为作者已将每一主题做了深入的了解，同时深知学生不易弄懂的地方，因此所写出来的内容一定不会让读者感到模棱两可。某些数据结构的重要名词或说明，如果以中文表达不合适，则用英文替代，以保持原有的风貌。

本书每一章的每一小节几乎都有练习题，目的是让读者能检测一下对这一小节的了解程度。为方便读者学习，将书中练习的参考答案放在本社网站的下载专区中，网址：http://www.tqbooks.net/download.asp。不过要提醒读者的是，一定要先做完练习再查看答案，否则会事倍功半。

本书由台湾碁峰资讯股份有限公司授权，经中国铁道出版社计算机图书中心审选。书中若有疏误或不妥之处，欢迎各位专家和读者批评指正。

编者

2006 年 5 月

目　录

第 1 章　算法分析

1.1　算法

算法（Algorithm）是一个解决问题的有限步骤过程，也可以说是在解答过程中的一步步过程。举例来说，现有一问题为：判断数字 X 是否在一已排序好的数字串 S 中，其算法为：从串 S 的第一个元素开始依次比较，直到 X 被找到或是数字串 S 已到结尾。假如 X 被找到，则显示出 Yes；否则，显示出 No。

可是当问题变得很复杂时，上述叙述性的算法就难以表达出来。因此，算法大都先以类似程序语言的方式来表达，然后利用用户所熟悉的程序语言来执行。本书则直接以 Java 程序语言来编写，因此笔者假设读者已具备编写 Java 语言的能力。

读者是否常常会问这样的一个问题："他的程序写得比我好吗？"，答案不是因为他是班上第一名，所以他所写出来的程序一定就是最好的。而是应该用一种比较科学的方法来比较，常用的方法是利用性能分析（Performance Analysis）方法来进行评估，而性能分析方法通常可分为时间复杂度分析（Time Complexity Analysis）和空间复杂度分析（Space Complexity Analysis），由于时间复杂度分析常被人们使用，因此我们选择此种分析方法来对性能进行评估。首先必须求出程序中每个语句的执行次数（其中"{"和"}"不加以计算），最后将这些执行次数加起来，再求出其 Big-O，所求出的结果就是这个程序的时间复杂度（关于 Big-O 的内容，1-2 节有详细的说明）。让我们先看下面 4 个例子。

1.1.1　数组元素相加

数组元素相加是将数组中每个元素的值加起来，其所对应的 Java 程序如下：

```
public static int sum(int arr[],int n)
{
  int i,total=0;
  for(i=0;i<n;i++)
  total+=arr[i];
  return total;
}
```

执行次数
1
n+1
n
1
2n+3

其中在 for(i=0; i<n; i++)循环体内会重复执行 n 次（0，1，2，…，n－1），但在 i=n 的时候 for 本身仍然会判断，所以 for 语句一共执行了 n+1 次。

1.1.2　矩阵相加

矩阵相加表示将相对应的元素相加，如 $\begin{bmatrix}5 & 6\\7 & 8\end{bmatrix}+\begin{bmatrix}2 & 3\\3 & 4\end{bmatrix}=\begin{bmatrix}7 & 9\\10 & 12\end{bmatrix}$，程序如下：

```
public static void add(int a[][],int b[][],int c[][],int m,int n)
{
 for (int i=0; i < n; i++)
 for (int j=0; j < n; j++)
 c[i][j] = a[i][j] + b[i][j]
}
```

执行次数
n+1
n(n+1)
n^2
$2n^2+2n+1$

注意：for 语句本身执行 n+1 次，进入循环体后，才执行 n 次。同时，我们假设两个矩阵皆为 n × n 元素。

1.1.3 矩阵相乘

矩阵相乘为 $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}$，对应的程序如下：

```
public static void mul(int a[][], int b[][], int c[][], int n)
        {
      int i,j,k,sum;
            for(i=0; i < n; i++)
           for(j=0;j<n;j++){
                 sum=0;
               for(k=0;k<n;k++)
               sum=sum+a[i][k]*b[k][j];
             c[i][j]=sum;
          }
                    }
```

执行次数
1
n+1
n(n+1)
n^2
$n^2(n+1)$
n^3
n^2
$2n^3+4n^2+2$
n+2

1.1.4 顺序查找

顺序查找是指在一个数组中，由第 1 个元素开始依次查找，我们假设要找的数据一定在数组中，其程序如下：

```
public static int search(int data[],int target, int n)
        {
        int i;
    for (i = 0;i < n; i++)
                if (target == data[i])
                return i;
          }
```

执行次数
1
n+1
n
1
2n+3

练习题

试回答下列程序中 x = x+1;语句执行了多少次？

```
for(i=1;i<=n;i++)
  for (j=i;j<=n;j++)
    x=x+1;
for (i=1;i<=n;i++){
  k=i+1;
  do {
     x=x+1;
  } while(k++<=n);
}
```

1.2 时间复杂度 Big-O

如何去计算完成一段程序或算法所需要的执行时间呢？在程序或算法中，每一条语句（statement）的执行时间为：①此语句执行的次数；②每次执行此语句所需的时间。两者相乘即为此语句的执行时间。由于执行每条语句所需的时间必须考虑到机器和编译器的功能，通常假设所需的时间为固定的，因此通常只考虑执行的次数即可。

算完程序中每一条语句的执行次数，并将其相加后，再利用 Big-O 来表示此程序的时间复杂度（Time Complexity）。

Big-O 的定义如下：

f(n)=O(g(n))，当且仅当唯一存在正整数 c 及 n_0，使得 $f(n)\leqslant cg(n)$，对所有的 n，$n\geqslant n_0$。

上述的定义表示可以找到 c 和 n_0，使得 $f(n)\leqslant c\times g(n)$，此时，我们称 f(n)的 Big-O 为 g(n)。请看下列例子：

$3n+2=O(n)$，因为可找到 c=4，$n_0=2$，使得 $3n+2\leqslant 4n$

$10n^2+5n+1=O(n^2)$，因为可以找到 c=11，$n_0=6$ 使得 $10n^2+5n+1\leqslant 11n^2$

$7\times 2^n+n^2+n=O(2^n)$，因为可以找到 c=8，$n_0=4$ 使得 $7\times 2^n+n^2+n\leqslant 8\times 2^n$

$10n^2+5n+1=O(n^3)$，可以很明显地看出，原来 $10n^2+5n+1\in O(n^2)$，而 n^3 又大于 n^2，当然 $10n^2+5n+1=O(n^3)$是没问题的。同理也可以得知 $10n^2+5n+1\neq O(n)$，因为 f(x)没有小于等于 $c\times g(n)$。

由上面的几个例子得知 f(n)为多项式，表示一程序完成时所需要计算的时间，而其 Big-O 只要取其最高次方的项即可。

根据上述的定义，得知数组元素值相加总的时间复杂度为 O(n)，矩阵相加的时间复杂度为 $O(n^2)$，而矩阵相乘的时间复杂度为 $O(n^3)$，顺序查找的时间复杂度为 O(n)。

其实可以加以证明，当 $f(n)=a_mn^m+\cdots+a_1n+a_0$ 时，$f(n)=O(n^m)$

【证明】

$$f(n)\leqslant \sum_{i=0}^{m}|a_i|n^i$$

$$\leqslant n^m\times\sum_{i=0}^{m}|a_i|n^{i-m}$$

$$\leqslant n^m\times\sum_{i=0}^{m}|a_i|，对\ n\geqslant 1\ 而言$$

可推出：$f(n)\in O(n^m)$，因为可将 $\sum_{i=0}^{m}|a_i|$ 视为 c，而 n^m 为 g(n)

即 Big-O 取其最大指数的部分即可，因此前面所讲的例子中，数组元素相加的 Big-O 为 O(n)，矩阵相加的 Big-O 为 $O(n^2)$，而矩阵相乘的 Big-O 为 $O(n^3)$。Big-O 的图形表示如图 1-1 所示。

例如有一程序执行的时间为 n^2+10n，则其 Big-O 为 $O(n^2)$，表示程序执行所花的时间最多如 n^2，换个角度说，就是在最坏的情况下也不会大于 n^2。

以 Big-O 的定义：$n^2+10n\leqslant 2n^2$，当 c=2，$n\geqslant 10$ 时，可推出：$n^2+10n\in O(n^2)$。Big-O 的图形表示如图 1-2 所示。

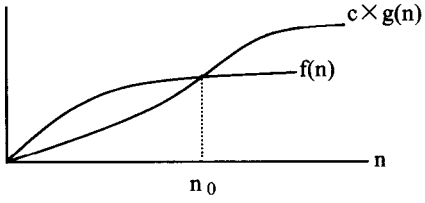


图 1-1　Big-O 的图形表示

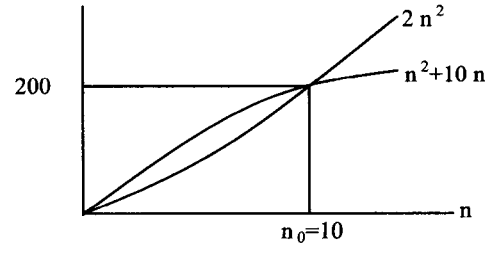


图 1-2　n^2+10n 的 Big-O 的图形表示

一般常见的 Big-O 类别如表 1-1 所示。

表 1-1　常见的 Big-O

Big-O	类　别
O(1)	常数时间 （constant）
$O(\log_2 n)$	对数时间 （logarithmic）
O(n)	线性时间 （linear）
$O(n^2 \log_2 n)$	对数线性时间 （log linear）
$O(n^2)$	平方时间 （quadratic）
$O(n^3)$	立方时间 （cubic））
$O(2^n)$	指数时间 （exponential）
O(n!)	阶乘时间 （factorial）
$O(n^n)$	n 的 n 次方时间

一般而言，这几种类别按照 O(1)，O(logn)，…，O(n!)，$O(n^n)$的排列顺序表示程序的运行效率越来越差，也可用另一种方式表示：

$O(1) < O(\log_2 n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$

$O(1) < O(\log_2 n)$表示后者所花的时间大于前者，因此效率上前者较后者优。往后我们可以利用 Big-O 来衡量程序或算法的效率。当 n 越大时，越能显示出其中的差异，如表 1-2 所示。若某位同学程序的 Big-O 为 $O(n\log_2 n)$，而你的程序为 O(n)，则你的程序和那位同学的相比，执行效率要高些。

表 1-2　各种 Big-O 的比较

n	$\log_2 n$	$n\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4 096	65 536
32	5	160	1 024	32 768	4 294 967 296

从表 1-2 可以看出，当 n 越来越大时，$n\log_2 n$、n^2、n^3 和 2^n 之间的差距便越来越大，如 n=32 时，$\log_2 n$ 才为 5，但 n^2 就等于 1 024，n^3 更大，已为 32 768，而 2^n 此时已达到 4 294 967 296，之间的差距是相当大的，在表 1-2 中省略了 n!，因为当 n=32 时，结果中的数字差不多有三十几位。各位读者只要清楚 Big-O 类别之间的排列即可。

除了 Big-O 之外，用来衡量效率的方法还有Ω 和 Θ，以下是它们的定义。

Ω 的定义如下：

$f(n)=\Omega(g(n))$当且仅当存在正整数 c 和 n_0，使得 $f(n) \geqslant c \times g(n)$，对所有的 n，$n \geqslant n_0$。

请看下面几个例子：

$3n+2=\Omega(n)$，因为可找到 c=3，$n_0=1$

使得 $3n+2 \geqslant 3n$

$200n^2+4n+5=\Omega(n^2)$，因为可找到 c=200，$n_0=1$

使得 $200n^2+4n+5 \geqslant 200n^2$

$10n^2+4n+2=\Omega(n)$，因为从定义得知 $10n^2+4n+2=\Omega(n^2)$，

由于 $n^2>n$，当然 $10n^2+4n+2$ 也可以为$\Omega(n)$。

Θ的定义如下：

$f(n)=\Theta(g(n))$，当且仅当存在正整数 c_1、c_2 及 n，使得 $c_1\times g(n)\leqslant f(n)\leqslant c_2\times g(n)$，对所有的 n，$n\geqslant n_0$。

以下面几个例子加以说明：

$3n+1=\Theta(n)$，因为可以找到 $c_1=3$、$c_2=4$ 且 $n_0=2$

使得 $3n\leqslant 3n+1\leqslant 4n$

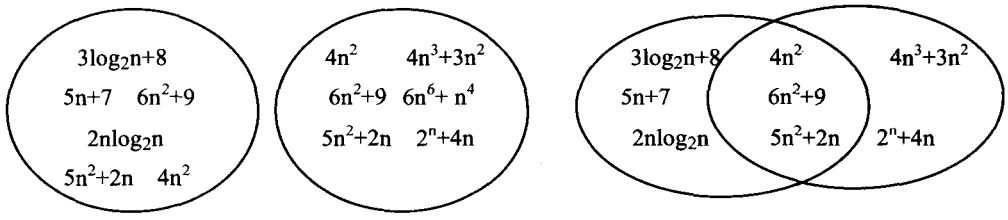
$10n^2+4n+6=\Theta(n^2)$，因为只要 $c_1=10$、$c_2=11$ 且 $n_0=10$

便可得 $10n^2\leqslant 10n^2+4n+6\leqslant 11n^2$

注意，$3n+2\neq\Theta(n^2)$，$10n^2+n+1\neq\Theta(n)$

读者可加以思考一下。

图 1-3 分别为 Big-O、 Ω、 Θ 的表示形式。



（a） $O(n^2)$　　（b）$\Omega(n^2)$　　（c）$\Theta(n^2)$，只有中间交集部分

图 1-3　Big-O、Ω、Θ的表示形式

有些问题，只要知道其做法便可求出 Big-O，下面我们举一些例子来说明。

顺序查找（Sequential Search）的情况可分为 3 种，第一种为最坏的，当要查找的数据放在文件的最后一个，需要 n 次才会查找到（假设有 n 个数据在文件中）；第二种为最好的，与第一种刚好相反，表示需要查找的数据在第一个，只要搜索 1 次就可以找到；最后一种为中间情况，其平均查找的次数为：

$$\sum_{k=1}^{n}(k\times(1/n))=(1/n)\times\sum_{k=1}^{n}k=(1/n)(1+2+\cdots+n)=1/n\times(n(n+1)/2)=(n+1)/2$$

因此得知其 Big-O 为 O(n)。

二分查找法（Binary Search）的情形和顺序查找不同，二分查找法是数据已经排好序，由中间的数据（mid）开始比较，便可以知道要查找的关键字（key）是落在 mid 的左边还是右边，之后，再将右边或左边中间的数据拿来与关键字相比，而每次所要调整的只是每个数据段的起始地址或是最终地址如图 1-4 所示。

例如：

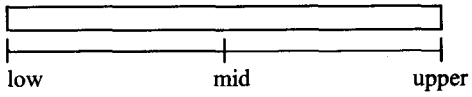


图 1-4　二分查找法

当 key > data[mid]时，low = mid +1，而 upper 不变，如图 1-5 所示，此例为调整起始地址。

当 key < data[mid]时，upper = mid -1，而 low 不变，如图 1-6 所示，此例为调整最终地址。

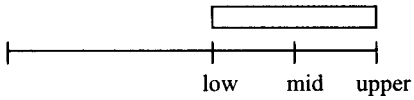


图 1-5　调整起始地址

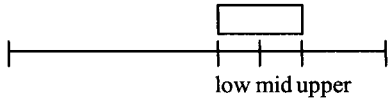


图 1-6　调整最终地址

若此时 key == data[mid]，则表示找到了查找的数据，从图 1-6 得知二分查找法每执行一次，则 n 会减半，第 1 次为 n/2，第 2 次为 $n/2^2$，第 3 次为 $n/2^3$，……，假设第 k 次时 n=1 比较结束，则 $n/2^k=1$。由于 $n/2^k=1$，所以 $n=2^k$，两边取 $\log_2$ 得到 $k=\log_2 n$，因此二分查找法的时间复杂度为 $O(\log_2 n)$。

二分查找法的 Java 程序如下：

```
public static void binsrch(int A[],int n,int x,int j)
{
   lower=1;
   upper=n;
   while(lower<=upper) {
      mid=(lower+upper)/2;
      if(x>A[mid])
         lower=mid+1;
      else if(x<A[mid])
         upper=mid-1;
      else {
         j=mid;
         System.out.println( "Found, " + x + " is #" + mid + "record." );
      }
   }
}
```

以下为二分查找法与顺序查找法的比较表，假设要查找的数据存在于数组中。

二分查找法的时间复杂度为 $O(\log_2 n)$，顺序查找的时间复杂度为 O(n)。

从表 1-3 中的比较可以得知，二分查找法比顺序查找法效率高，那是因为二分查找法的 Big-O 为 $\log_2 n$，远比顺序查找法的 Big-O 为 n 来得好。

表 1-3　二分查找法与顺序查找法的比较

数组大小	二分查找	顺序查找
128	7	128
1 024	10	1 024
1 048 576	20	1 048 576
4 294 967 296	32	4 294 967 296

接下来讨论一个更有趣的例子——斐波纳契序列（Fibonacci number），其定义如下：

$f_0=0$

$f_1=1$

$f_n=f_{n-1}+f_{n-2}$　　当 $n \geqslant 2$ 时

因此

$f_2 = f_1 + f_0 = 1 + 0 = 1$

$f_3 = f_2 + f_1 = 1 + 1 = 2$

$f_4 = f_3 + f_2 = 2 + 1 = 3$

$f_5 = f_4 + f_3 = 3 + 2 = 5$

…

$f_n = f_{n-1} + f_{n-2}$

若以递归的方式进行计算的话，图形如图 1-7 所示。

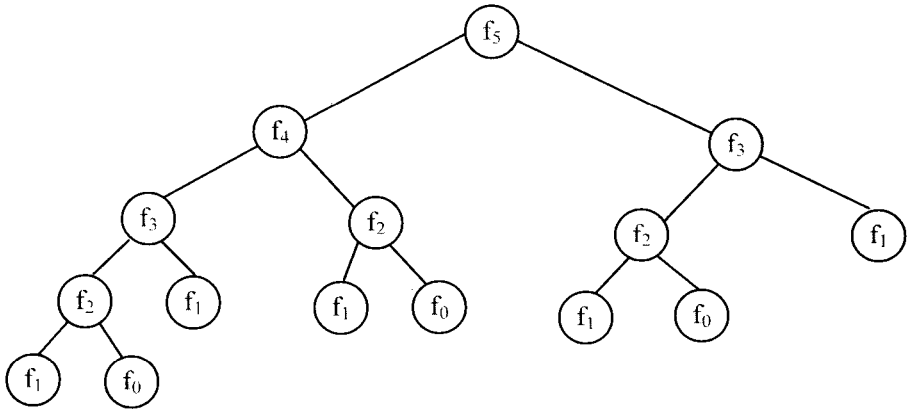


图 1-7　以递归方式进行计算

用递归方式，需计算的项目表，如表 1-4 所示。

表 1-4　用递归方式需计算的项目表

n 第 n 项）	需计算的项目数
0	1
1	1
2	3
3	5
4	9
5	15
6	25

当 n = 3（f_3），从表 1-4 可知计算的项目数为 5；n = 5 时，需计算的项目数为 15。因此可以用下列公式表示：

```
T(n)>2×T(n-2)
> 2 × 2 × T(n-4)
> 2 × 2 × 2 × T(n-6)
   …
> 2 × 2 × 2 × … × 2 × T(0)        ( 2 共有 n/2 次)
```

当 T(0) = 1 时，$T(n) > 2^{n/2}$，此时的 n 必须大于等于 2，因为当 n=1 时，$T(1) = 1 < 2^{1/2}$。

上述斐波纳契序列是以递归的方式算出，若改以非递归的方式计算的话，其 f(n)执行的项目为 n+1 项，Big-O 为 O(n)。由此可看出斐波纳契序列以非递归方式计算的效率比递归方式好，请参阅表 1-5，同时也说明某些问题以非递归方式来处理是较好的。有关递归的详细解说，请参阅第 5 章。

Java 程序的以递归方式计算斐波纳契序列：

```
int Fibonacci(int n)
{
   if(n==0)
     return 0;
   else if(n==1)
     return 1;
   else
      return(Fibonacci(n-1)+Fibonacci(n-2));
}
```

Java 程序的以非递归方式计算斐波纳契序列：

```
Int Fibonacci(int n)
{
   int prev1, prev2, item, i;
   if(n==0)
      return 0;
   else if(n==1)
      return 1;
   else {
      prev2=0;
      prev1=1;
      for(i=2;i<=n;i++){
         item=prev1+prev2;
         prev2=prev1;
         prev1=item;
      }
      return item;
   }
}
```

表 1-5　斐波纳契序列以递归和非递归求解所花费的时间

计算第 n 项斐波纳契序列的值	递　归		非　递　归	
n	所计算的项目（2^{n2}）	所需执行的时间	所计算的项目（n+1）	所需执行的时间
40	1 048 576	1048μs	41	41ns
60	1.1×10^{8}	1s	61	61ns
80	1.1×10^{12}	18min	81	81ns
100	1.1×10^{15}	13 天	101	101ns
200	1.3×10^{30}	4×10^{13} 年	201	201ns

1 ns = 10^{-9} s　　1 μs = 10^{-6} s

从表 1-5 可以明显地看出，当 n 越大时计算第 n 项的斐波纳契序列所需执行的时间就越多，如第 100 项的斐波纳契序列以递归方式执行需要 13 天，而以非递归方式执行只需要 101ns，而当 n=200 时，以递归方式执行则需要 4×10^{13} 年，而以非递归方式执行则只需要 201ns。

练习题

1. 试问下列多项式的 Big-O 及找出 c 和 n_0，使其符合 $f(n) \leqslant c \times g(n)$。

   （1）$100n+9$　　（2）$1000n^2+100n-8$　　（3）$5\times 2^n+9n^2+2$

2. 试求下列多项式的Ω，并找出 c 和 n_0。

   （1）$3n+1$　　（2）$10n^2+4n+5$　　（3）$8\times 2^n+8n+6$

3. 试求下列多项式的Θ，并找出 c_1、c_2 及 n_0。

   （1）$3n+2$　　（2）$9n^2+4n+2$　　（3）$8n^4+5n^3+5$

1.3　思考题

1. 请计算下列程序中 x=x+1 的执行次数。

（1）
```
for(i=1;i<=n;i++)
for(j=i;j<=n;j++)
   for(k=j;k<=n;k++)
        x=x+1;
```

（2）
```
i=1;
while(i<=n){
     x=x+1;
     i=i+1;
}
```

（3）
```
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
     x=x+1;
```

（4）
```
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
   for(k=1;k<=n;k++)
        x=x+1;
```

（5）
```
for(i=1;i<=n;i++){
      j=i;
      for(k=j+1;k<=n;k++)
           x=x+1;
  }
```

（6）
```
for(i=1;i<=n;i++){
j=i;
while(j>=2){
 j/=5;
 x=x+1;
      }
}
```

（7）
```
k=100000;
while(k!=5){
   k/=10;
   x=x+1;
}
```

（8）
```
for(i=1;i<=n;i++){
k=i+1;
do{
   x=x+1;
} while(k>n);
}
```

2. 假设数组 A 有 10 个元素，分别为 2，4，6，8，10，12，14，16，18，20，请问若查找 1，3，13 及 21 这 4 个数，在下列的程序中，do … while 内的语句分别执行多少次？

```
i=1;
j=10; /* 因为 A 数组有 10 个元素 */
do{
   k=(i+j)/2;
   if(A[k] <= x)  /* x 为要查找的数据 */
     i=k+1;
   else
     j=k-1;
} while(i<=j);
```

3. 试问下列多项式 f(n)的 Big-O 为多少?

   （1）$f(n)=\sum_{i=1}^{n} i$　　（2）$f(n)=\sum_{i=1}^{n} i^2$　　（3）$f(n)=\sum_{i=1}^{n} x^i$

4. 试问下列多项式 f(n)的 Big-O?

   （1）$n^3+8^{10}n^2$　　（2）$5n^2-6$　　（3）$n^{1.001}+n\log_2 n$

（4）$n^2 2^n+n^3+n$　　（5）$\sum_{i=1}^{n} x^3$

5. 试问下列语句是否正确，如果不正确，请改正。

（1）$\log^2 n^2+9=O(n)$　　（2）$n^2 \log_2 n= O(n^2)$

（3）$48n^3+ 9n^2=\Omega(n^2)$　　（4）$48n^3+9n^2=\Omega(n^4)$

（5）$n!=O(n^n)$　　（6）$n^{k+\varepsilon}+n^k \log_2 n=\Theta(n^{k+\varepsilon})$　对所有的 k 和 ε，且 $k \geqslant 0$，$\varepsilon>0$

6. 有一冒泡排序（Bubble Sort）的程序如下，试求此程序的 Big-O。

```
public static void bubble_sort(int data[],int n)
{
   int i,j,k,temp,flag;
   for(i=0;i<n-1;i++){
      flag=0;
      for(j=0; j<n-1;j++)
         if(data[j]>data[j+1]){
               flag=1;
               temp=data[j];
               data[j]=data[j+1];
               data[j+1]=temp;
      }
      if(flag==1)
        break;
    }
 }
```

7. 下列有一段选择排序（Selection Sort）的程序，试求该程序的 Big-O。

```
 public static void select_sort(int data[], int size)
{
   int base, compare, min, temp, i;
   for(base=0;base<size－1;base++) {
   /* 将当前的数据与后面数据中最小的对调 */
     min = base;
     for(compare=base+1; compare<size; compare++) {
        if(data[compare]<data[min])
          min=compare;
          temp=data[min];
          data[min]=data[base];
          data[base]=temp;
          printf("Access:");
       for(i=0;i<size;i++)
         printf("%d  ", data[i]);
         printf("\n");
}
    }
       }
```

第 2 章　数组

2.1　数组表示法

在还没有谈到数组（Array）之前，先来看看线性表（Linear List）。线性表又称顺序表（Sequential List）或有序表（Ordered List）。其特性是每一项数据是依据它在链表的位置所形成的一个线性排列次序，所以 x[i]会出现在 x[i+1]之前。

线性表经常发生的操作如下：

- 取出列表中的第 i 项，$0 \leqslant i \leqslant n-1$。
- 计算列表的长度。
- 由左至右或由右至左读此列表。
- 在第 i 项加入一个新值，使原来的第 i，i+1，…，n 项变为第 i+1，i+2，…，n+1 项，从开始即之后的数据都要向后移动一位。
- 删除第 i 项，使其原来的第 i+1，i+2，…，n 项变为第 i，i+1，…，n–1 项，即在第 i 项之后的数据都会往前移动一位。

在 Java 程序语言中常利用数组实现线性表，以线性的对应方式将元素 a_i 置于数组的第 i 个位置上，若要读取 a_i，可利用 $a_i = a_0 + i \times d$ 来求得。其中 a_i 为相对地址，a_0 为数组的起始地址，d 为每一个元素所占的空间大小，但要注意的是，Java 的数组是从 0 开始的。

以下介绍的是数组的表示方法。

2.1.1　一维数组

假设一维数组（One Dimension Array）是 A(0:u*1)，且每一个元素占 d 个空间，则 $A(i) = l_0 + i \times d$，其中 l_0 是数组的起始地址。若每一元素，所占的空间为 d，且起始元素位置为 0，则数组 A 的每一元素所对应的地址如表 2-1 所示（假设 d=1）。

$A(i) = l_0 + i \times d$

表 2-1　数组 A

数组元素	A[0]	A[1]	A[2]	…	A[i]	…	A[u–1]
地　　址	l0	l0+1	l0+2	…	l0+(i)	…	l0+(u–1)

若数组是 A(t:u)，则 $A(i) = l_0 + (i-t) \times d$。

若数组 A(2:12)，则 A(6)与 A(2)起始点相差 4 个单位（6–2），相当于上述(i–t)。

若数组为 A(1:u)，表示数组的起始元素位置从 1 开始，则 $A(i) = l_0 + (i-1)d$，其中 d 为每一元素所占的空间大小。因此，我们必须注意数组起始元素的地址，例如，有一数组 A(0:100)，而起始地址 A(0)=100，d=2，则 A(16)=?

解：由于 $A(i) = 100 + (i) \times d$

所以 $A(16) = 100 + 16 \times 2 = 132$

假若数组为(t:u)，则

$A(i) = l_0 + (i-t) \times d$