Chapter 1 Introduction to Data Structures

信息科学与技术学院　　　黄方军

data_structures@163.com

东校区实验中心B502

课程安排

讲授3学时/周， 实验2学时/周（独立完成）

教学用书： Robert L. Kruse and Alexander J. Ryba “Data Structures and Program Design in C++”，高教出版社

课程资料地址：

ftp://172.18.184.29/黄方军老师/

参考书：

1． **Sartaj Sahni**，数据结构、算法与应用（**C++**语言描述，影印版），机械工业出版社

1.1 为什么学习数据结构？

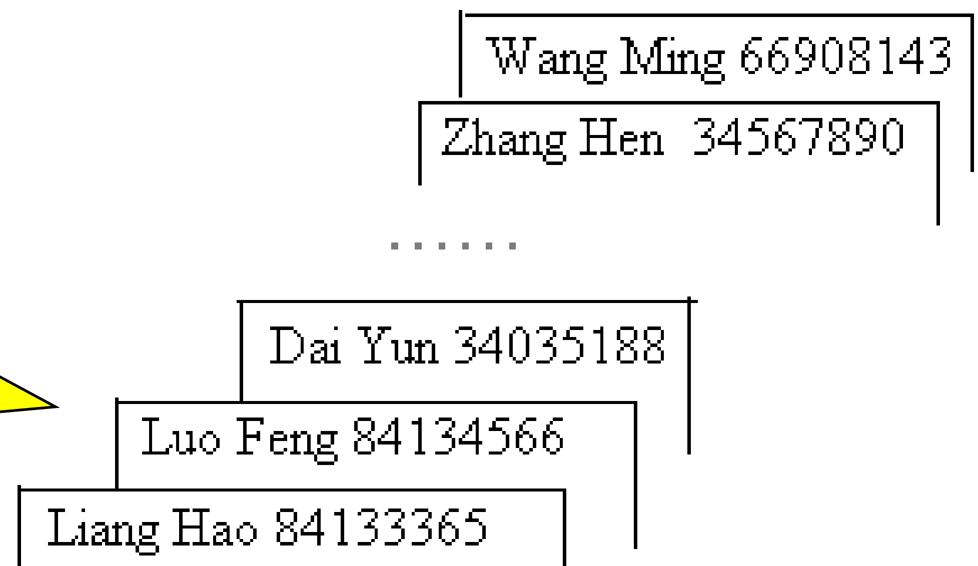
- 什么是数据结构？
- 为什么学习数据结构？达到什么目的？
- 应该掌握哪些内容？技能？
- 如何学习？

1.1 从电话号码查询说起

问题：你有一叠亲友和客户等的名片，需要在其中查找某个人的卡片。

- **解法1：顺序查找**
- **卡片组织：顺序（线性逻辑结构）**

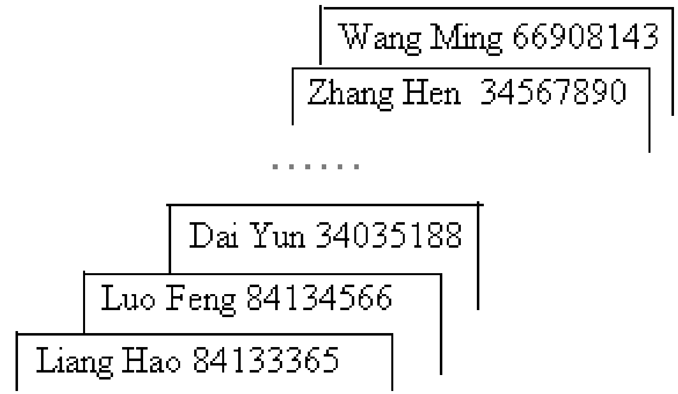
如何组织数据（逻辑关系）以便实现数据上的处理



1.1 从电话号码查询说起

解法1的实现(1)：

- 使用数组存储卡片；



(Liang Hao, 84133365)	(Luo Feng, 84134566)	…	(Wang Ming, 66908143)

用C++表示：

```
struct Card {
  string name;
  string phone;
} ;
 Card phones[100];
```

如何用C/C++表示数据，即数据的存储结构

1.1 从电话号码查询说起

选择数据存储方法

用C++表示：

```
struct Card {
  string name;
  string phone;
};
```

解法1的实现(1)：

- 算法的实现；

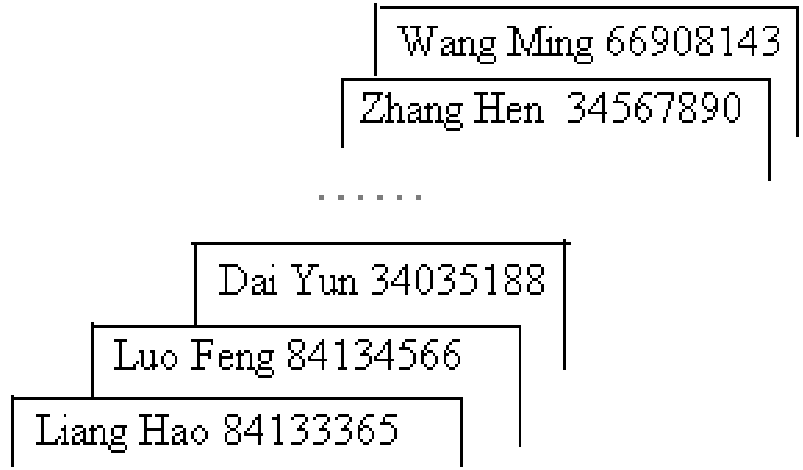
```
int sequentialSearch(Card[] st, int n,  string const &target) {
 //在查找表st中顺序查找其关键字为k的记录。如果查找成功，则返回
 //该元素在查找表中的下标(0—n-1); 否则，返回表的长度n，表明查找失败。
  int i;
 for (i = 0; i<n && st[i].name != target;i++);
 return i;
}
```

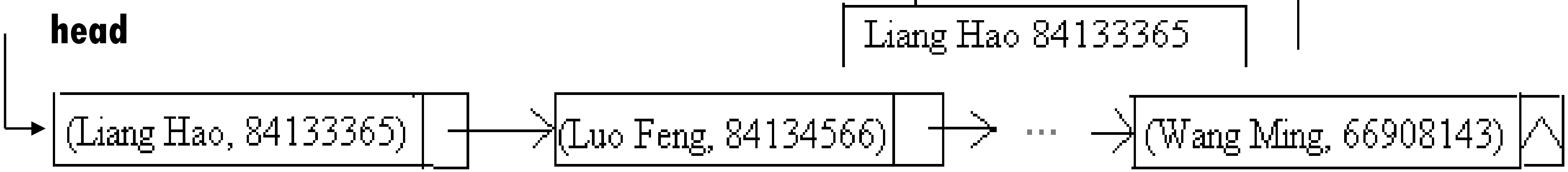
You may pack the data the method in a class.

1.1 从电话号码查询说起

解法1的实现(2):

- 使用链表存储卡片；





用**C++**表示：

```
struct Node{
  Card  data;
  Node * next;
};
 Node *head;
```

如何设计/选择存储数据的方式（存储结构）便于数据上操作（如查找）的实现

1.1 从电话号码查询说起

解法1的实现(2)：

- 使用链表存储卡片；

```
struct Node{
  Card  data;
  Node * next;
};
```

```
int sequentialSearchLinked(Node *head, string const &target) {
 //在查找表st中顺序查找其关键字为k的记录。如果查找成功，则返回
 //该元素在查找表中的下标(0—n-1); 否则，返回表的长度n，表明查找失败。
  Node *p=head;
  int n=0;
  while (p!=NULL&&(p->data).name!=target) {
     p=p->next;
     n=n+1;
  };
  return n;
}
```

如何评价以上两种实现?
掌握评价算法的基本技能

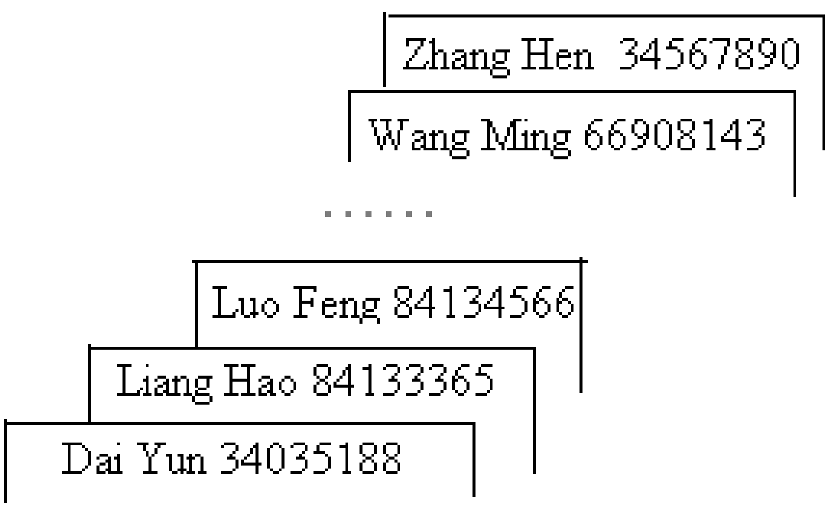
1.1 从电话号码查询说起

解法2：二分查找。

条件：先把卡片按照姓名排序；

组织结构：线性逻辑结构;

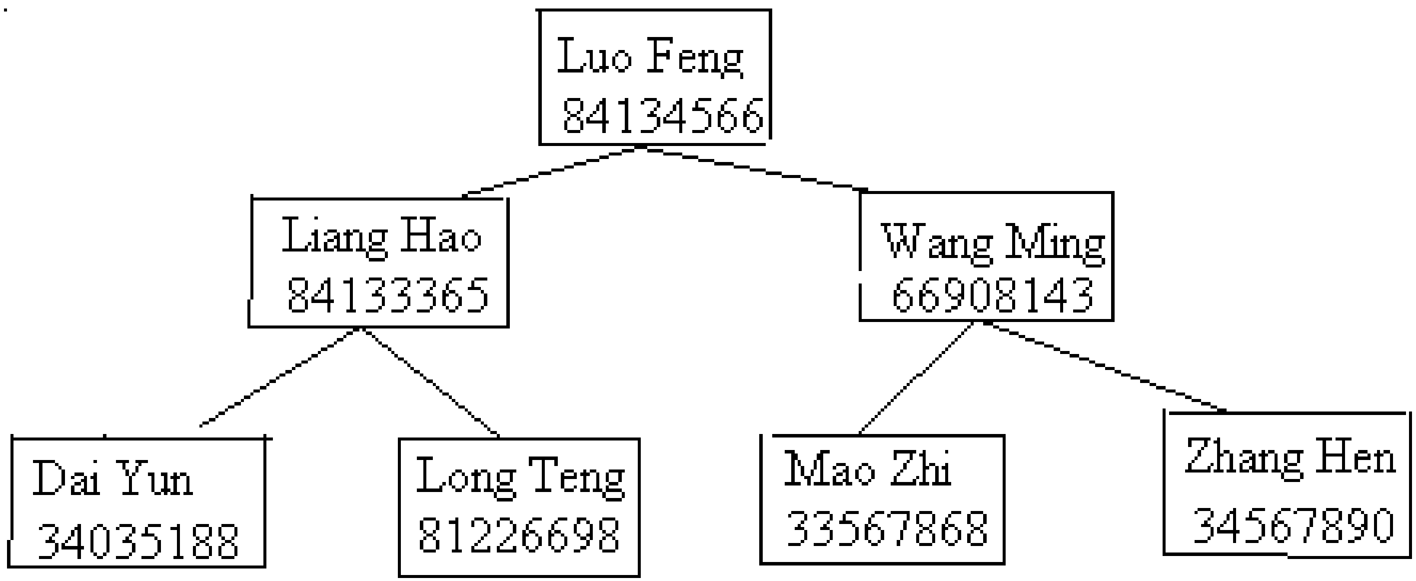
存储结构：连续结构（数组，又称 随机存取结构）



1.1 从电话号码查询说起

解法3：二叉查找树；

数据组织（逻辑关系）：二叉树型结构；

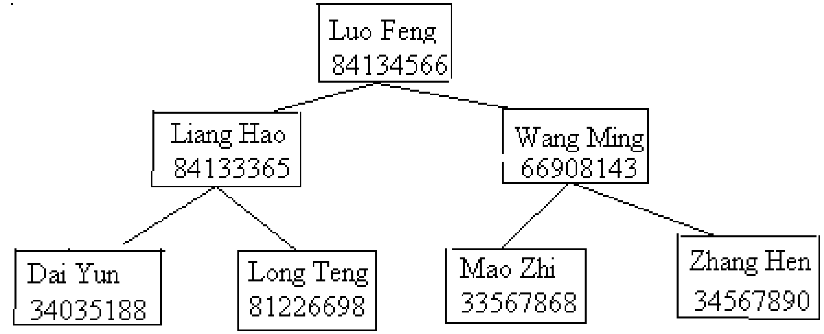


1.1 从电话号码查询说起

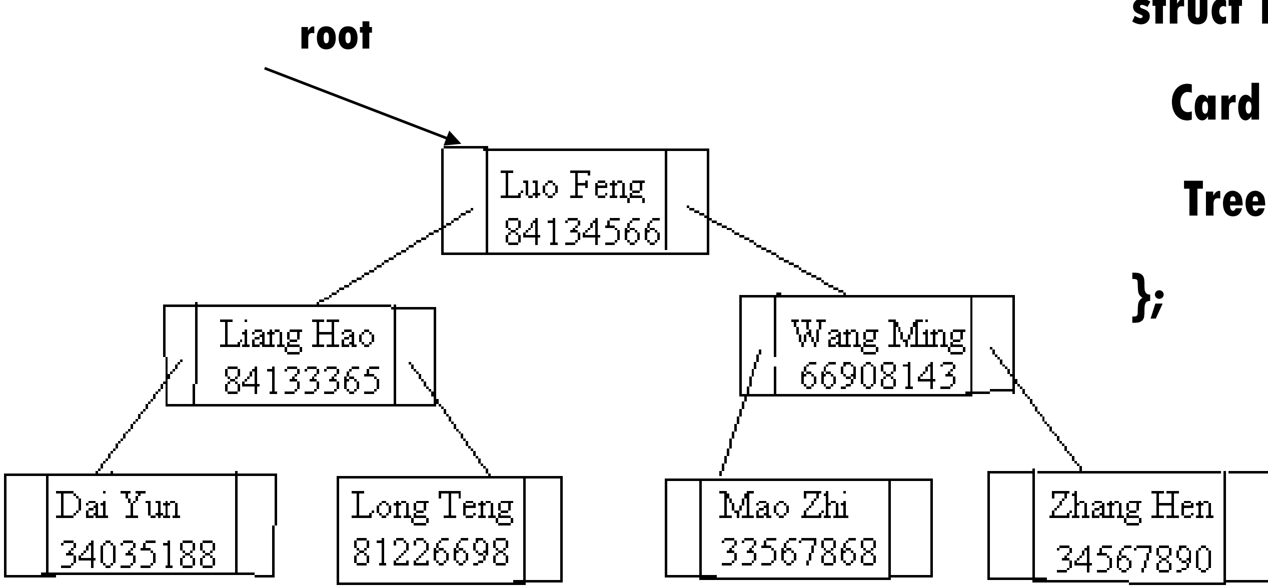
解法3：二叉查找树；

数据组织：二叉树型结构；

存储结构：二叉链表；




```
struct TreeNode{
   Card data;
    TreeNode *llink,*rlink;
};
```



1.1 从电话号码查询说起

解法4：使用联合容器map

```
int main() {
  map<stirng, string> phones; //初始化一个map容器
  phones[“Dai Yun"]      = “34035188”; //插入记录
  phones[“Liang Hao"]   = “84133365”;
  phones["Zhang Hen"]  = “34567890”;
  phones[“Wang Ming"] = “66908143”;
  phones[“Luo Feng"]     = “84134566”;
  map<string,string>::iterator cur = phones.find("Zhang Min");
  if (cur != phones.end())
  //如果查找失败，cur指向容器“末尾”，否则指向找到的记录
     cout << "Zhang Min's phone is " << (*cur).second << endl;
  else
     cout << "Zhang Min doesn't exist in phones." <<endl;
}
```

map是一个解决查找问题的数据结构

1.1 从电话号码查询说起

- 处理的数据是什么？需要什么样的数据操作？
- 如何组织数据（逻辑结构）？
- 如何存储数据（存储结构）？
- 如何实现操作（算法的实现）？
- 常见特定数据组织形式与操作构成ADT（抽象数据类型）, 其实现便是一个数据结构，解决问题的一个组件/工具；
- 是否有数据结构和算法适用于当前问题？
- 如何评价不同的解法（算法的时间复杂度和空间复杂度）？

1.3.1 Value Parameters

Formal parameters & Actual parameters

```
#include<iostream>
using namespace std;
int Abc(int a, int b, int c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4;
}

int main()
{
  cout << Abc(2,3,4) << endl;
}
```

Formal parameters

Actual parameters

1.2.2 Template Functions

```
#include<iostream>
using namespace std;
float Abc(float a, float b, float c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4;
}

int main()
{
    cout << Abc(2,3,4) << endl;
}
```

1.2.2 Template Functions

```
#include<iostream>
using namespace std;
template<class T>
T Abc(T a, T b, T c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4;
}

int main()
{
  cout << Abc(2,3,4) << endl;
}
```

1.2.3 Reference Parameters

```
#include<iostream>
using namespace std;
template<class T>
T Abc(T& a, T& b, T& c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4;
}
void main(void)
{
  int x = 2, y = 3, z = 4;
  cout << Abc(x,y,z) << endl;
}
```

1.2.4 Const Reference Parameters

```
#include<iostream>
using namespace std;

template<class T>
T Abc(const T& a, const T& b, const T& c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4;
}
void main(void)
{
  cout << Abc(2,3,4) << endl;
}
```

1.2.4 Const Reference Parameters

```
#include<iostream>
using namespace std;

template<class Ta, class Tb, class Tc>
Ta Abc(const Ta& a, const Tb& b, const Tc& c)
{
  return a+b+b*c+(a+b-c)/(a+b)+4;
}
void main(void)
{
  cout << Abc(2,3,4) << endl;
}
```

1.2.4 Recursive Functions

$$f(n)=\begin{cases}1 & n\le 1\\ nf(n-1) & n>1\end{cases}$$

```
#include<iostream>
using namespace std;

int Factorial(int n)
{// Compute n!
  if (n <= 1) return 1;
  else return n * Factorial(n - 1);
}
void main(void)
{
  cout << "5! = " << Factorial(5) << endl;
}
```

1.2.4 Recursive Functions

```
#include<iostream>
using namespace std;
template<class T>
T Sum(T a[], int n)
{// Return sum of numbers a[0:n -1].
  T tsum = 0;
  for (int i = 0; i < n; i++)
    tsum += a[i];
  return tsum;
}
void main(void)
{
  int a[6] = {1, 2, 3, 4, 5, 6};
  cout << Sum(a,6) << endl;
}
```

1.2.4 Recursive Functions

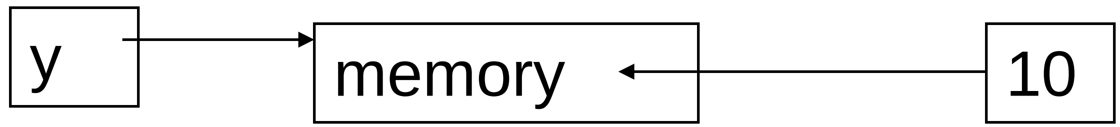
```
#include<iostream>
Using namespace std;
template<class T>
T Rsum(T a[], int n)
{// Return sum of numbers a[0:n - 1].
   if (n > 0)
      return Rsum(a, n-1) + a[n-1];
   return 0;
}

void main(void)
{
   int a[6] = {1, 2, 3, 4, 5, 6};
   cout << Rsum(a,6) << endl;
}
```

1.3 Dynamic memory allocation

- The operator *new*
- The operator *delete*

1.3.1 The operator *new*

```
int *y;
 y=new int;
 *y=10;
```



or

```
    int *y=new int(10);
```

or

```
 int *y;
y=new int(10);
```

1.3.2 One-Dimensional Arrays

First One:

```
x = 5;
float myArray[x] = {1, 3, 4.6, 7, 8}
```

Second One:

```
float *x = new float [n];
```

1.3.4 The Operator *Delete*

Free space allocated by *new*

delete y;
delete [ ] x;

1.4 Class

```
enum sign {plus, minus};
class Currency {
  public:
    // constructor
    Currency(sign s = plus, unsigned long d = 0,
                        unsigned int c = 0);
    // destructor
    ~Currency() {}
    bool Set(sign s, unsigned long d,
                  unsigned int c);
    bool Set(float a);
    Currency& Increment(const Currency& x);
    void Output() const;
  private:
    sign sgn;
    unsigned long dollars;
    unsigned int cents;
};
```

1.5.1 What is testing

```
void OutputRoots(T a, T b, T c)
{// Compute and output the roots of the
quadratic.

   T d = b*b-4*a*c;
   if (d > 0) {// two real roots
           float sqrtd = sqrt(d);
           cout << "There are two real roots "
                << (-b+sqrtd)/(2*a) << " and "
                << (-b-sqrtd)/(2*a)
                << endl;}
```

1.5.1 What is testing

```
else if (d == 0)
        // both roots are the same
        cout << "There is only one distinct root "
            << -b/(2*a)
            << endl;
      else //  complex conjugate roots
          cout << "The roots are complex"
              << endl
              << "The real part is "
              << -b/(2*a) << endl
              << "The imaginary part is "
              << sqrt(-d)/(2*a) << endl;
  }
```

1.5.2 Designing test data

- Black Box Methods
- White Box Methods

1.7 Performance of program

The performance of a program, we mean the amount of computer memory and time needed to run a program.

- Space complexity（空间复杂度）.
- Time complexity （时间复杂度）.

1.7.1 Components of Space complexity

- Instruction space
  - The compiler used to compile the program into machine code.
  - The compiler options in effect as the time of compilation.
  - The target computer.

1.7.1 Components of Space complexity

a+b+b*c+(a+b-c)/(a+b)+4

```
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)

1.7.1 Components of Space complexity

- Data space
  - Space needed by constants and simple variables.
  - Component variables such as the array (structures and dynamically allocated memory)
  - …

1.7.1 Components of Space complexity

- Data space

Type	space	range
char	1	-128-127
short	2	-32768-32767

…

1.7.1 Components of Space complexity

- Environment space
  - The return address.
  - The values of all local variables and value formal parameters in the function being invoked (recursive functions only)
  - ……

1.7.1 Summary of Space complexity

Divided the total space needed by a program into two parts.

- A fixed part.
- instruction space
- Space for simple variables
- Fixed-size component variables
- Space for constants
- ……

1.7.2 Summary of Space complexity

- A variable part.
- Components variables whose size depends on the particular problem instance being solved.
- Recursion stack space
- …

1.7.2 Summary of Space complexity

The space requirement $S(P)$ of any program P may therefore be written as

$$S(P) = c + S_P \text{ (instance characteristics)}$$

1.7.3 Examples

Example 1.1

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

T is instance characteristics:
S_{abc} (instance characteristics)=0

Magnitude of a, b and c is instance characteristics:
S_{abc} (instance characteristics)=0

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

T is instance characteristics:
S_{abc} (instance characteristics) =3*sizeof(*T*)

Magnitude of a, b and c is instance characteristics:
S_{abc} (instance characteristics)=0

1.7.3 Examples

Example 1.2-Sequential Search

```
template<class T>
int SequentialSearch(T a[], const T& x, int n)
{
   int i;
   for (i = 0; i < n && a[i] != x; i++);
   if (i == n) return -1;
   return i;
}
```

n is instance characteristics:
Assume T is int.
2 bytes for a, x, n, i, 0, -1, respectively.
The total data space needed is 12 bytes.
Since this space is independent of n,
$S_{SequantialSearch}$ (n)=0

Example 1.3

```
template<class T>
T Sum(T a[], int n)
{// Return sum of numbers a[0:n -1].
   T tsum = 0;
   for (int i = 0; i < n; i++)
      tsum += a[i];
   return tsum;
}
```

Space is required for a, n, i, and tsum, respectively.
Since this space is independent of n, $S_{Sum}(n)=0$

1.7.3 Examples

Example 1.4

```
template<class T>
T Rsum(T a[], int n)
{// Return sum of numbers a[0:n - 1].
   if (n > 0)
      return Rsum(a, n-1) + a[n-1];
   return 0;
}
```

The recursion stack space includes space for the formal parameters a and n and the return address.

The depth of recursion is n+1. So S_{Rsum} (n)=6(n+1)

1.7.4 Components of time complexity

The time $T(P)$ taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. This run time is denoted by

t_P (instance characteristics).

For example,

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + \ldots$$

1.7.4 Components of time complexity

Two more manageable approaches to estimating run time are

- **Identify one or more key operations and determine the number of times these performed.**
- **Determine the total number of steps executed by the program.**

1.7.4 Operation Counts

Example *1.5-Max element*

```
template<class T>
int Max(T a[], int n)
{// Locate the largest element in a[0:n-1].
   int pos = 0;
   for (int i = 1; i < n; i++)
     if (a[pos] < a[i])
        pos = i;
   return pos;
}
```

The total number of element comparisons is *n-1*. the function max does other comparisons (each iteration of the for loop is preceded by a comparison between *i* and *n*) that are not included in the estimate. Other operations such as initializing *pos* and incrementing the for loop index *i* are also not included in the estimate.

1.7.4 Operation Counts

Example *1.6-Polynomial Evaluation*

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

```
template<class T>
T PolyEval(T coeff[], int n, const T& x)
{
   T y = 1, value = coeff[0];
   for (int i = 1; i <= n; i++) {
      y *= x;
      value += y * coeff[i];
      }
   return value;
}
```

The number of additions is n, and the enumber of multiplications is $2n$.

1.7.4 Operation Counts

Example *1.7-Horner's Rule*

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

```
template<class T>
T Horner(T coeff[], int n, const T& x)
{
   T value = coeff[n];
   for (int i = 1; i <= n; i++)
      value = value * x + coeff[n - i];
   return value;
}
```

The number of additions is *n*, and th enumber of multiplications is *n*.

1.7.4 Operation Counts

Example *1.8-Ranking*

```
template<class T>
void Rank(T a[], int n, int r[])
{
   for (int i = 0; i < n; i++)
      r[i] = 0;  // initialize
   for (i = 1; i < n; i++)
      for (int j = 0; j < i; j++)
         if (a[j] <= a[i]) r[i]++;
         else r[j]++;
}
```

$$a = [4,3,9,3,7]$$

$$r = [2,0,4,1,3]$$

The number of element comparisions is 1+2+3+…+(n-1) = $(n-1)n/2$.

1.7.4 Operation Counts

对rank后的数据从小到大排序

Example 1.9-*Rank sort*

```
void Rearrange(T a[], int n, int r[])
{
   T *u = new T [n+1];
   for (int i = 0; i < n; i++)
      u[r[i]] = a[i];
   for (i = 0; i < n; i++)
      a[i] = u[i];
   delete [] u;
}
```

$a = [4,3,9,3,7]$

$r = [2,0,4,1,3]$

The complete sort requires $(n-1)n/2$ comparisons and $2n$ element moves.

1.7.4 Operation Counts

先找最大的数再从小到大排序

Example 1.10-*Selection sort*

```
template<class T>
void SelectionSort(T a[], int n)
{
   for (int size = n; size > 1; size--) {
      int j = Max(a, size);
      Swap(a[j], a[size - 1]);
      }
}
```

The complete sort requires $(n-1)n/2$ comparisons and $3(n-1)$ element moves.
注：共n-1次循环，每次循环中Swap需要进行3次移动。

1.7.4 Operation Counts

寻找数组中最大数的函数

Example 1.11-*Maximum Element*

```
template<class T>
int Max(T a[], int n)
{// Locate the largest element in a[0:n-1].
   int pos = 0;
   for (int i = 1; i < n; i++)
     if (a[pos] < a[i])
        pos = i;
   return pos;
}
```

1.7.4 Operation Counts

交换数组中两个元素的值

Program 1.11-*Swap Two Values*

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

1.7.4 Operation Counts

冒泡法将最大元素移到最右端位置，利用冒泡法来进行排序

Example 1.12-*Bubble Sort*

```
void Bubble(T a[], int n)
{// Bubble largest element in a[0:n-1] to right.
  for (int i = 0; i < n - 1; i++)
    if (a[i] > a[i+1])
       Swap(a[i], a[i + 1]);
}
void BubbleSort(T a[], int n)
{// Sort a[0:n - 1] using bubble sort.
  for (int i = n; i > 1; i--)
    Bubble(a, i);
}
```

The number of comparisons between pairs of elements of *a* is ***n*-1**.

The number of element comparisons **(*n*-1)n/2**.

Best, Worst, and Average Operation Counts

Best Operation Counts:

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

Worst Operation Counts:

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

Average Operation Counts:

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

$$O_P^{AVG}(n_1, n_2, \ldots, n_k) = \sum_{I \in S(n_1, n_2, \ldots, n_k)} (P(I) * operation_P(I))$$

1.7.4 Operation Counts

Example 1.13-*Sequential Search*

在数组中搜索某一元素，计算最优、最差、以及平均搜索次数

```
template<class T>
int SequentialSearch(T a[], const T& x, int n)
{
  int i;
  for (i = 0; i < n && a[i] != x; i++);
    if (i == n) return -1;
  return i;
}
```

The average count for a successful search is

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

1.7.4 Operation Counts

Example 1.14-*Insertion into a Sorted Array*

```
template<class T>
void Insert(T a[], int& n, const T& x)
{
   int i;
   for (i = n-1; i >= 0 && x < a[i]; i--)
      a[i+1] = a[i];
   a[i+1] = x;
   n++;
}
```

在排序后的数组中插入某一元素，计算最优、最差、以及平均搜索次数

The average count for a successful search is

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

1.7.4 Operation Counts

根据已有的索引对数组中的元素进行排序，查看需要交换的次数

Example 1.15-*Rank Sort Revisited*

```
template<class T>
void Rearrange(T a[], int n, int r[])
{
  for (int i = 0; i < n; i++)
    while (r[i] != i) {
      int t = r[i];
      Swap(a[i], a[t]);
      Swap(r[i], r[t]);
    }
}
```

$$a = [4,3,9,3,7]$$

$$r = [2,0,4,1,3]$$

The number of swaps performed varies from a low of zero to a high of **2(*n*-1)**

1.7.4 Operation Counts

Example 1.16-*Selection Sort Revisited*

在每一次将最大值交换到最右端时先检查下数组是否已经是按从小到大排序了

```
template<class T>
void SelectionSort(T a[], int n)
{
  bool sorted = false;
  for (int size = n; !sorted && (size > 1); size--)
  {
    int pos = 0;
    sorted = true;
    for (int i = 1; i < size; i++)
      if (a[pos] <= a[i]) pos = i;
      else sorted = false; // out of order
    Swap(a[pos], a[size - 1]);
    }
}
```

The best case for the early-terminating version of selection sort arises when the array a is sorted to begin with. Now the outer for loop iterates just once, and the number of comparisons between elements of a is n-1. In the worst case the outer for loop is iterated until size=1 and the number of comparisons is $(n-1)n/2$.

1.7.4 Operation Counts

利用冒泡法对数组中的元素进行排序，每次排序前进行判断，看是否已正确排序

Example 1.17-*Bubble Sort Revisited*

```
bool Bubble(T a[], int n)
{// Bubble largest element in a[0:n-1] to right.
  bool swapped = false; // no swaps so far
  for (int i = 0; i < n - 1; i++)
    if (a[i] > a[i+1]) {
      Swap(a[i], a[i + 1]);
      swapped = true; // swap was done
      }
  return swapped;
}

void BubbleSort(T a[], int n)
{// Early-terminating version of bubble sort.
  for (int i = n; i > 1 && Bubble(a, i); i--);
}
```

The worst-case number of comparisons is unchanged from the original version **$(n-1)n/2$**. The best case number of comparisons is $(n-1)$.

1.7.4 Operation Counts

Example 1.18-*Insertion Sort*

利用数据插入方式来对数组中的元素进行排序

```
template<class T>
void InsertionSort(T a[], int n)
{// Sort a[0:n-1].
   for (int i = 1; i < n; i++) {
      // insert a[i] into 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;
      }
}
```

$$a = [2,1,6,8,9,11]$$

The best case number of comparisons is n-1 and the worst case number of comparisons is $(n-1)n/2$.

1.7.5 Step Counts

Example 1.19

```
template<class T>
T Sum(T a[], int n)
{
   T tsum = 0;
   count++; // for tsum = 0
   for (int i = 0; i < n; i++) {
     count++; // for the for statement
     tsum += a[i];
     count++; // for assignment
     }
   count++; // for last execution of for statement
   count++; // for return
   return tsum;
}
```

1.7.5 Step Counts

Example 1.19-Simplified Version of 1.19

```
int count = 0;
template<class T>
T Sum(T a[], int n)
{
   for (int i = 0; i < n; i++)
      count += 2;
   count += 3;
   return 0;
}
```

1.7.5 Step Counts

Example 1.20

```
template<class T>
T Rsum(T a[], int n)
{// Return sum of numbers a[0:n - 1].
   count++; // for if conditional
   if (n > 0)
         {count++; // for return and Rsum invocation
          return Rsum(a, n-1) + a[n-1];}
   count++; // for return
   return 0;
}
```

$$
\begin{aligned}
t_{Rsum}(n) &= 2+t_{Rsum}(n-1) \\
&= 2+2+t_{Rsum}(n-2) \\
&\;\;\vdots \\
&= 2n+t_{Rsum}(0) \\
&= 2(n+1)\ (n>=0)
\end{aligned}
$$

1.7.5 Step Counts

Example 1.21

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int
    cols)
{
  for (int i = 0; i < rows; i++)
    for (int j = 0;  j < cols; j++)
      c[i][j] = a[i][j] + b[i][j];
}
```

1.7.5 Step Counts

Example 1.21

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{// Add matrices a and b to obtain matrix c.
  for (int i = 0; i < rows; i++) {
    count++; // preceding for loop
    for (int j = 0;  j < cols; j++) {
      count++; // preceding for loop
      c[i][j] = a[i][j] + b[i][j];
      count++; // assignment
      }
    count++; // last time of j for loop
    }
  count++; // last time of i for loop
}
```

1.7.5 Step Counts

Example 1.21

```
template<class T>
void Add( T **a, T **b, T **c, int rows, int cols)
{// Add matrices a and b to obtain matrix c.
   for (int i = 0; i < rows; i++) {
      for (int j = 0;  j < cols; j++) {
         c[i][j] = a[i][j] + b[i][j];
         count += 2;
         }
      count += 2;
      }
   count++;
}
```

If count is zero to begin with, it will be $2*rows*cols+2*rows+1$.

If rows>cols, then it is better to interchange the two for statements in Example 2.21. If this is done, the step count will become $2*rows*cols+2*cols+1$.

1.8 Asymptotic notation (O,Ω,Θ,*o*)

Two important reasons to determine operation and step counts are

- **To compare the time complexities of two programs that compute the same function**
- **To predict the growth in run time as the instance characteristics change.**

However, it has some limitations.

- **Focus on certain "key" operations.**
- **Instructions x = y and x = y+z+x/y count as one step.**

1.8.1 Big Oh Notation(O)

Common asymptotic functions

1	constant
$\log n$	logarithmic
n	linear
$n\log n$	*nlogn*
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

1.8.1 Big Oh Notation (O)

The big oh notation provides an upper bound for the function *f*.

Definition [Big oh] ***f*(*n*)=O(*g*(*n*)) iff positive constants *c* and n_0 exist such that $f(n) \le cg(n)$ for all *n*, $n \ge n_0$.**

Example

$$f(n) = 3n + 2 < 4n, \text{ for n} \ge 2, \ f(n) = O(n)$$

f(*n*) denotes the time or space complexity of a program

1.8.1 Big Oh Notation (O)

Example 1.22-*Linear Function*

Consider $f(n) = 3n+2$. When n is at least 2, $3n+2 \leq 3n+n \leq 4n$. So $f(n) = O(n)$. Thus $f(n)$ is bounded from above by a linear function.

Example 1.23-*Quadratic Function*

Consider $f(n) = 10n^2+4n+2$.

$f(n) = O(n^2)$.

1.8.1 Big Oh Notation(O)

Example 1.24-*Exponential Function*

Consider $f(n) = 6 \times 2^n + n^2$.

$f(n) = O(2^n)$.

Example 1.25-*Constant Function*

Consider $f(n) = 9$.

$f(n) = O(1)$.

1.8.1 Big Oh Notation(O)

Example *1.26-Loose Bounds*

For example,

$3n+3 = O(n^2)$;

$10n^2+4n+2 = O(n^4)$;

$6n2^n+20 = O(n^2 2^n)$

1.8.1 Big Oh Notation(O)

Example *1.27-Incorrect Bounds*

For example,

$3n+2 \neq O(1)$;

$10n^2+4n+2 \neq O(n)$;

$3n^2 2^n+4n2^n+8n^2 = O(2^n)$

1.8.1 Big Oh Notation(O)

Theorem 1.1

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = O(n^m)$.

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

$$\le n^m \sum_{0}^{m} |a_i| n^{i-m}$$

$$\le n^m \sum_{0}^{m} |a_i| \qquad for \;\; n \ge 1$$

1.8.1 Big Oh Notation(O)

Theorem 1.2-*Big oh ratio theorem*

Let *f*(*n*) and *g*(*n*) be such that $\lim_{n\to\infty} f(n)/g(n)$

Exists. *f*(*n*)=O(*g*(*n*)) iff $\lim_{n\to\infty} f(n)/g(n) \le c$

For some finite constant *c*.

1.8.1 Big Oh Notation(O)

Theorem 1.2-*Big oh ratio theorem*

Proof **if $f(n)=O(g(n))$, then positive c and an n_0 exist such that $f(n)/g(n) \le c$ for all $n \ge n_0$.**

Hence $\lim_{n\to\infty} f(n)/g(n) \le c$.

Next suppose that $\lim_{n\to\infty} f(n)/g(n) \le c$

It follows that an n_0 exists such that

$f(n) \le \max\{1,c\} * g(n)$ for all $n \ge n_0$.

1.8.1 Big Oh Notation(O)

- **$10n^2+3n+100=O(n^2)$, lim=10;**
- **$1000n^2=O(n^{2.1})$, lim=0**
- **称$g(n)$为$f(n)$的渐近上界(Asymptotic Upper Bound).**
- **约定O(1)代表常数.**
- **$g(n)$常取一些简单的初等函数,n^k,$\log_2 n$和$n^k\log_2 n$等:**

1.8.2 Omega Notation(Ω)

The omega notation provides an lower bound for the function *f*.

Definition [Big oh] **$f(n)=\Omega(g(n))$ iff positive constants c and n_0 exist such that $f(n) \geq cg(n)$ for all n, $n \geq n_0$.**

Example

$$f(n) = 3n + 2 > 3n, \text{ for all n, } f(n) = \Omega(n)$$

$f(n)$ denotes the time or space complexity of a program

1.8.2 Omega Notation(Ω)

Example 1.28

$f(n) = 3n+2 > n$ for all n. So $f(n) = \Omega(n)$;

$f(n) = 10n^2+4n+2 > 10n^2$ for $n \geq 0$. So $f(n) = \Omega(n^2)$;.

1.8.2 Omega Notation(Ω)

Theorem 1.3

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$,

then $f(n) = \Omega(n^m)$.

Theorem 1.4-*Omega ratio theorem*

Let $f(n)$ and $g(n)$ be such that $\lim_{n\to\infty} g(n)/f(n)$

Exists. $f(n)=\Omega(g(n))$ iff $\lim_{n\to\infty} g(n)/f(n) \le c$

For some finite constant c.

1.8.2 Omega Notation(Ω)

- 称$g(n)$为$f(n)$的渐近下界
- 例如,

  $f(n)=0.001n^2-10n-1000=\Omega(n^2)$

  因为:$\lim f(n)/n^2=0.001$

1.8.3 Theta Notation(Θ)

The theta notation is used when the function *f* can be bounded both from above and below by the same function *g*.

Definition [Theta] $f(n)=\Theta(g(n))$ **iff positive constants** c_1 *and* c_2 **and** n_0 **exist such that** $c_1 g(n) \le f(n) \le c_2 g(n)$ **for all** n, $n \ge n_0$.

1.8.3 Theta Notation(Θ)

Theorem 1.5

If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Theorem 1.6-*Theta ratio theorem*

Let $f(n)$ and $g(n)$ be such that $\lim_{n\to\infty} f(n)/g(n)$ and $\lim_{n\to\infty} g(n)/f(n)$ Exists. $f(n) = \Theta(g(n))$ iff $\lim_{n\to\infty} f(n)/g(n) \le c$ and $\lim_{n\to\infty} g(n)/f(n) \le c$ for some finite constant c.

1.8.3 Theta Notation(Θ)

- 符号Θ
- 如果$f(n)=O(g(n))$同时 $f(n)=\Omega(g(n))$则$f(n)=\Theta(g(n))$,并称$f(n)$与$g(n)$同阶.
- Lim $f(n)/g(n)=c$, $0<c<\infty$,则

  $f(n)=\Theta(g(n))$
- $g(n)$取上述初等函数

1.8.4 Little Oh (o)

Definition [Little oh] $f(n)=o(g(n))$ **iff**

$f(n) = O(g(n))$ **and** $f(n) \neq \Omega(g(n))$.

Example 1.29

$3n+2 = o(n^2)$ as $3n+2 = O(n^2)$ and $3n+2 \neq \Omega(n^2)$.

1.8.5 Properties

	$f(n)$	Asymptotic
E1	c	$\oplus(1)$
E2	$\sum_{i=0}^{k} c_i n^i$	$\oplus(n^k)$
E3	$\sum_{i=1}^{n} i$	$\oplus(n^2)$
E4	$\sum_{i=1}^{n} i^2$	$\oplus(n^3)$
E5	$\sum_{i=1}^{n} i^k, k>0$	$\oplus(n^{k+1})$
E6	$\sum_{i=0}^{n} r^i, r>1$	$\oplus(r^n)$
E7	$n!$	$\oplus(n(n/e)^n)$

1.8.5 Properties

E8 $\sum_{i=1}^{n} 1/i$ $\oplus(\log n)$

$\oplus$ can be any one of O, Ω, and Θ

Figure 2.15 Asymptotic identities

1.9 Practical Complexities

Figure 1.5-Value of various functions

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Figure **1.5** Value of various functions

1.9 Practical Complexities

Figure 1.6-Plot of various functions

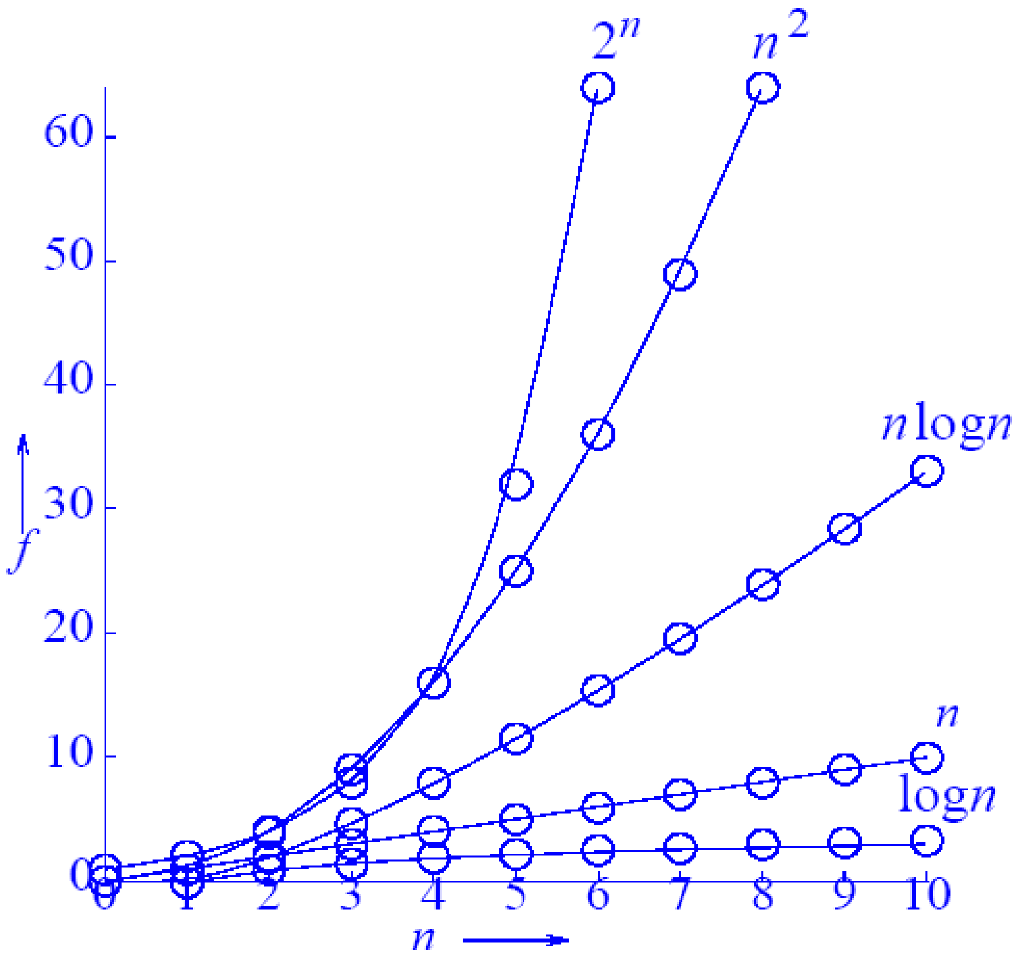


Figure 2.24 Plot of various functions

1.9 Practical Complexities

Figure 1.7-Run times on a 10^9 instruction per second

n	n	$n\log n$	n^2	n^3
1000	1mic	10mic	1milli	1sec
10000	10mic	130mic	100milli	17min
10^6	1milli	20milli	17min	32years

1.9 Practical Complexities

Figure 1.8-Run times on a 10^9 instruction per second

n	n^4	n^{10}	2^n
1000	17min	3.2×10^{13} years	3.2×10^{283} years
10000	116 days	???	???
10^6	3×10^7 years	??????	??????

1.10 Performance Measurement

```
#include <time.h>
void main(void)
{
  int a[1000], step = 10;
  clock_t start, finish;
  for (int n = 0; n <= 1000; n += step) {
    for (int i = 0; i < n; i++)
      a[i] = n - i; // initialize
    start = clock( );
    InsertionSort(a, n);
    finish = clock( );
    cout << n << ' ' << (finish - start) / float(CLK_TCK) << endl;
    if (n == 100) step = 100;
    }
  }
```