

DATA STRUCTURES AND ALGORITHMS

Textbook:

Fundamentals of Data Structure in C++, Silicon Press, 2006

Instructor:

金远平 ypjin@seu.edu.cn

The PPT can be downloaded from

工作区：软件学院/文件

Teaching assistants:

**严祥光 18251627206,
719155827@qq.com**

**王辉 13770729185,
582336641@qq.com**

Total class hours: 64

Total lab. hours: 16

**On Wednesday evening 6:30pm-
10:00pm of week 5, 8, 11, 14.**

At Room 262, 235, Computer Building

Home work:

- Should be handed to teaching assistants on Wednesday of week 3, 5, 7, 9, 11, 13, 15
- All 71141+7114202—229 (total 42) to 严祥光
- 7114230—236+all 71Y14 (total 42) to 王辉

Evaluation:

Course Attendance: 10%,

Exercises and Projects: 20%,

**Final Examination (Textbook and
Course Notes allowed): 70%**

References:

- 1 金远平, 数据结构(C++描述), 清华大学出版社, 2005
- 2 T. A. Standish, Data Structures, Algorithms & Software Principles in C, Addison-Wesley Publishing Company, 1994

Chapter 1

Basic Concepts

Purpose:

Provide the tools and techniques necessary to design and implement large-scale software systems, including:

- **Data abstraction and encapsulation**
- **Algorithm specification and design**
- **Performance analysis and measurement**
- **Recursive programming**

1.1 Overview: System Life Cycle

(1) Requirements

specifications, purpose, input, output

(2) Analysis

break the problem into manageable pieces,
bottom-up, top-down

(3) Design

data objects, operations on them, abstract
data type, algorithm specification and
design

- (4) Refinement and coding**
representations for data object, algorithms
for operations , components reuse
- (5) Verification and maintenance**
testing, error removal, update

1.3 Data Abstraction and Encapsulation

Definition: **Data Encapsulation** or **information Hiding** is the concealing of the implementation details of a data object from the outside world.

Definition: **Data Abstraction** is the separation between the *specification* of a data object and its *implementation*.

Mobile phone example.

Definition: A **Data Type** is a collection of *objects* and a set of *operations* that act on those objects.

predefined and user-defined:
char, int, arrays, structs, classes.

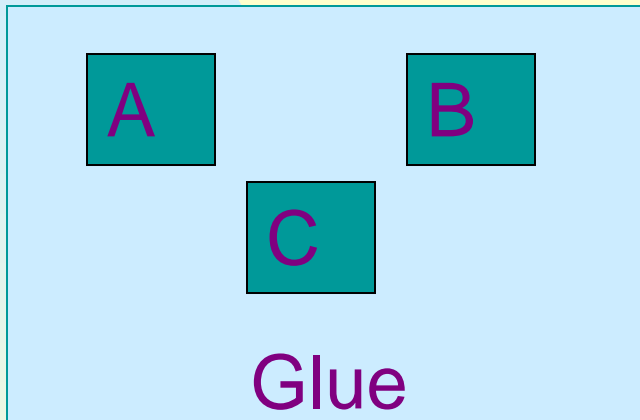
Definition: An **Abstract Data Type (ADT)** is a data type with the specification of the objects and the specification of the operations on the objects being separated from the representation of the objects and the implementation of the operations.

Benefits of data abstraction and data encapsulation:

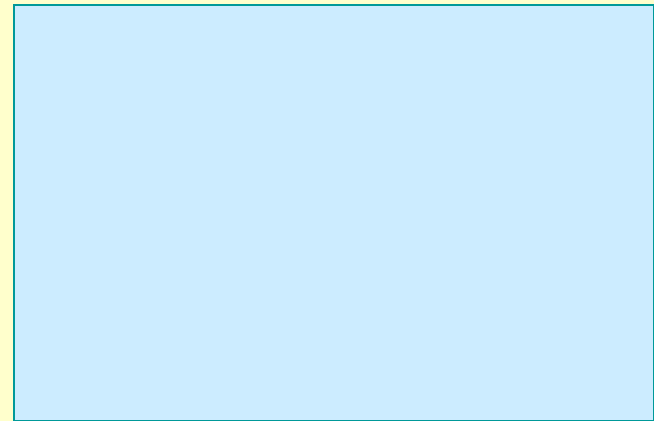
- (1) Simplification of software development**
 - data types A, B, C + Glue**
 - (a) a team of 4 programmers**
 - (b) a single programmer**

(2) Testing and debugging

Code with data abstraction



Code without data abstraction



Unshaded areas represent code to be searched for bugs.

(3) Reusability

data structures implemented as distinct entities of a software system

(4) Modifications to the representation of a data type

a change in the internal implementation of a data type will not affect the rest of the program as long as its interface does not change.

1.5 Algorithm Specification

1.5.1 Introduction

Definition: An **algorithm** is finite set of instructions that, if followed, accomplishes a particular task.

Must satisfy the following criteria:

- (1) **Input** Zero or more quantities externally supplied.
- (2) **Output** At least one quantity is produced.
- (3) **Definiteness** Clear and unambiguous.

(4) Finiteness Terminates after a finite number of steps.

(5) Effectiveness Basic enough, feasible

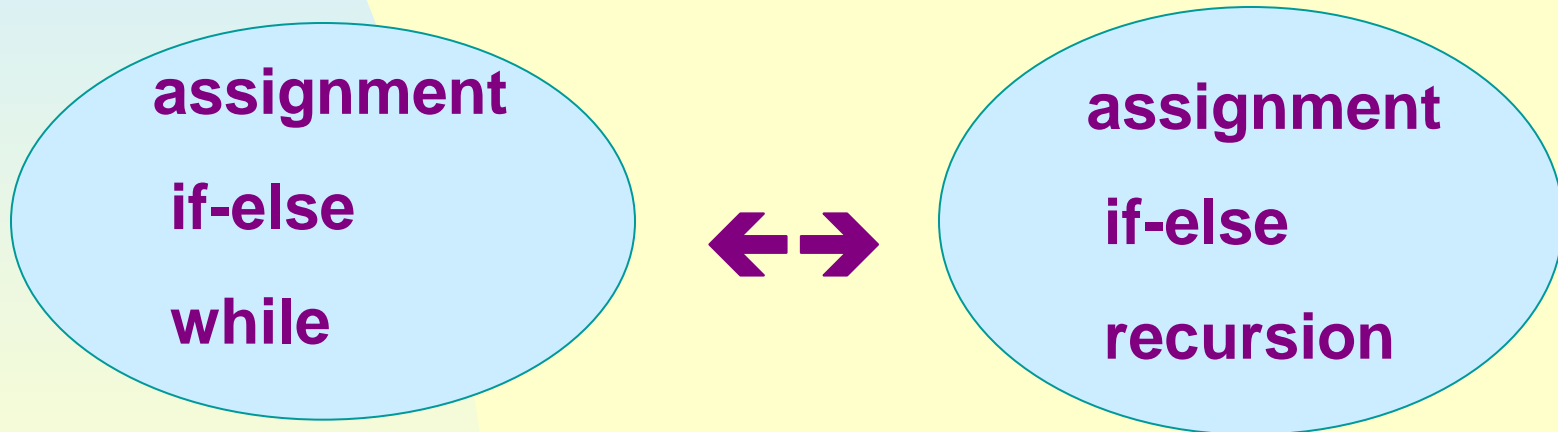
Compare: algorithms and programs

1.5.2 Recursive Algorithms

Direct recursion: functions call themselves.

Indirect recursion: call other functions that again invoke the calling function.

Expressiveness:



Recursion is especially appropriate for recursively defined problems.

Example 1.4 *Recursive binary search*

```
int BinarySearch (int *a, const int x, const int left, const int
right)
// search the sorted array a[left:right] for x
{
    if (left <= right) {
        int middle = (left + right) / 2;
        if (x < a[middle]) return BinarySearch(a, x, left, middle - 1);
        else if (x > a[middle])
            return BinarySearch(a, x, middle + 1, right);
        return middle; // x == a[middle]
    }
}
```

```
} // end of if  
return -1; // not found  
}
```

To invoke, use: `BinarySearch(a,x,0,n-1)`

Example 1.5 Permutation Generator

Problem: for a set of $n \geq 1$ elements, print all possible permutations of it.

Solution:

look at (a,b,c) , the answer is by printing:

- (1) a followed by all permutations of (b,c)
- (2) b followed by all permutations of (a,c)
- (3) c followed by all permutations of (b,a)

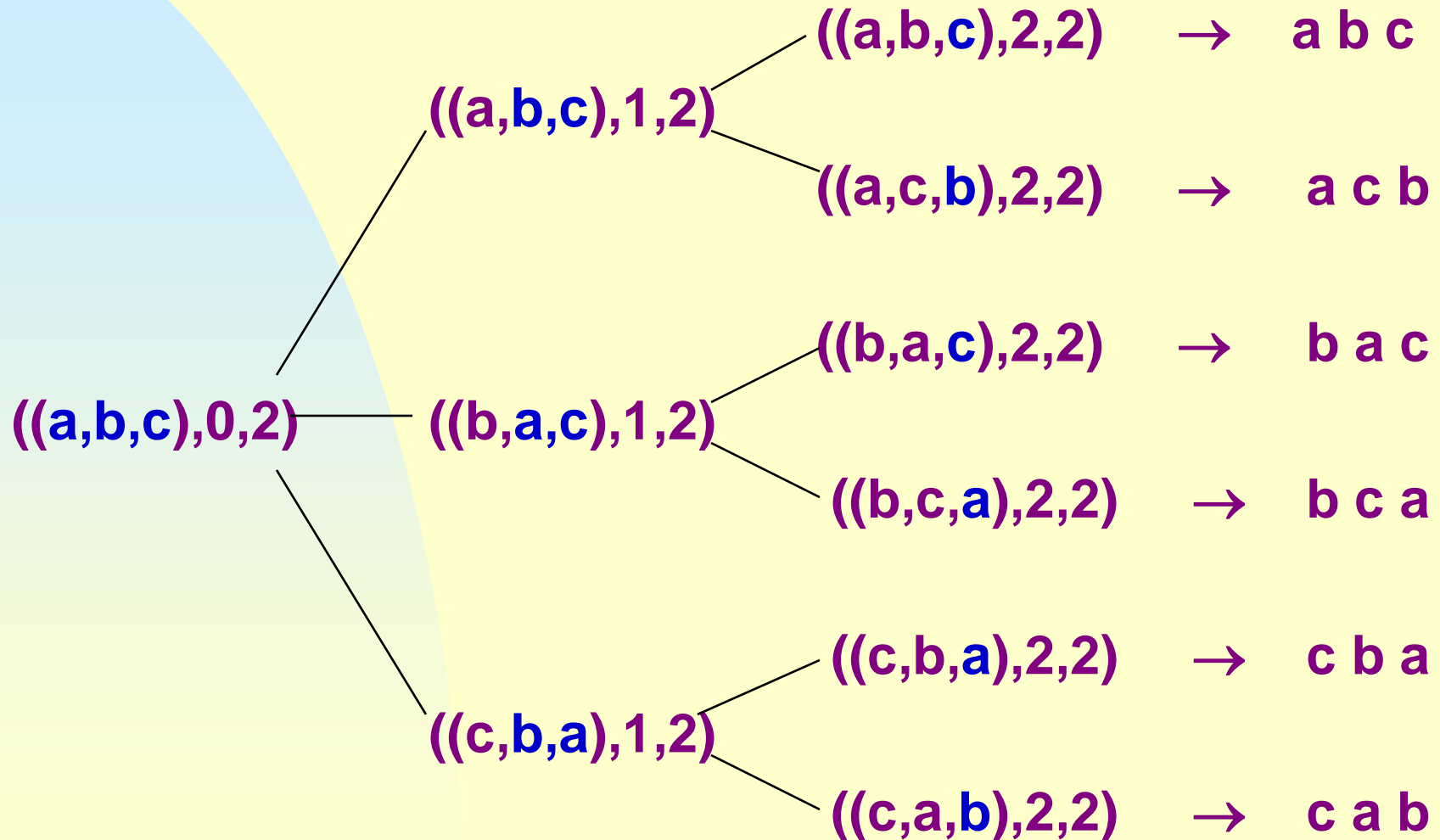
the clue to recursion: we can solve the problem for a set of n elements if we have an algorithm that works on $n-1$ elements.

```
void Permutations (char *a, const int k,const int m)
// generate all permutations of a[k],...,a[m]
{
    if (k ==m) {    // output permutation
        for (int i=0; i<=m; i++) cout << a[i] << " ";
        cout << endl;
    }
}
```

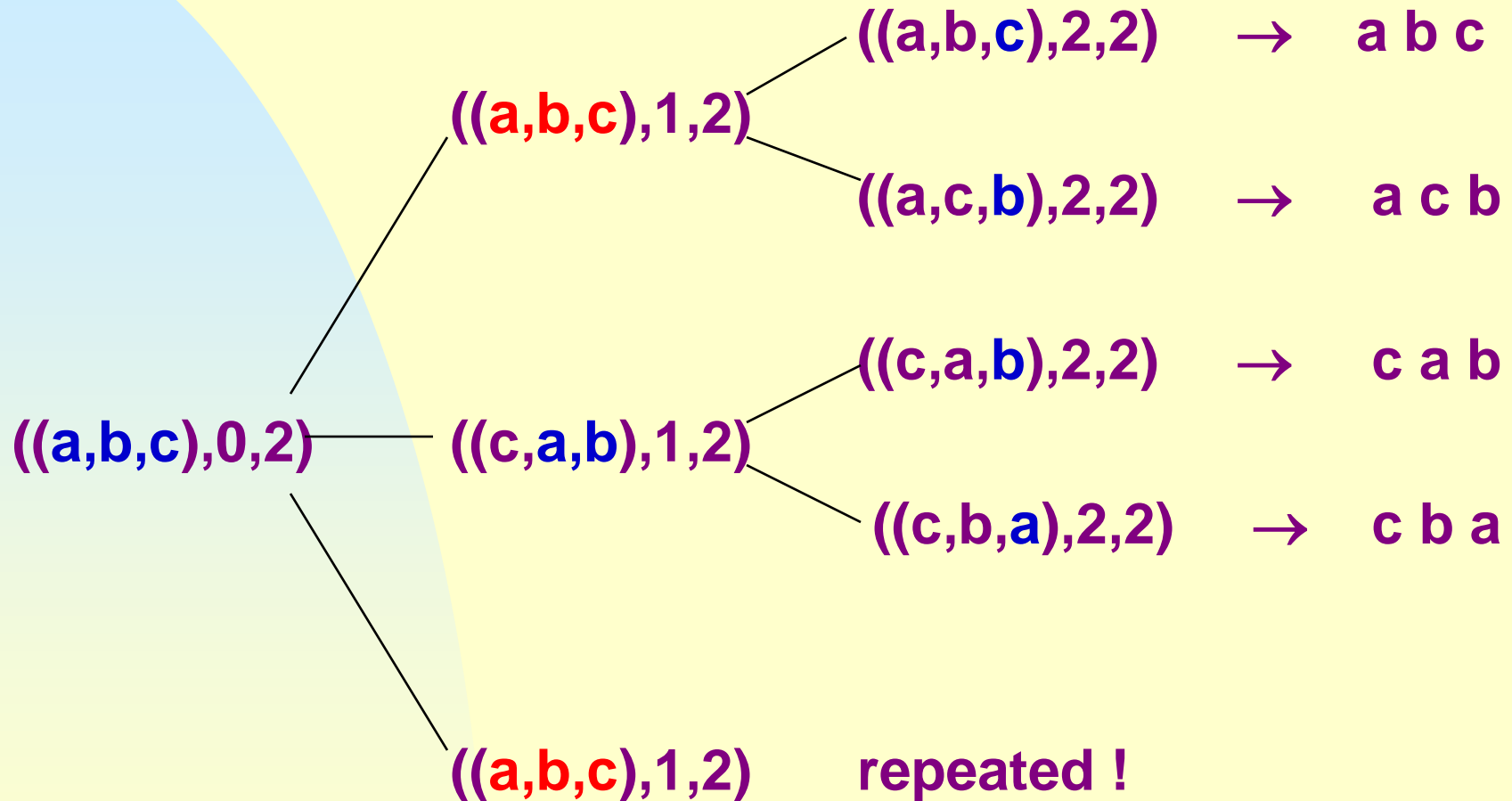
```
else { // a[k:m] has more than one permutation.  
    // generate recursively  
    for (i=k; i<=m; i++) {  
        swap(a[k], a[i]); // interchange a[k] and a[i]  
        Permutations(a,k+1,m); // all permutations of a[k+1:m]  
        swap(a[k], a[i]); // to return to the original configuration  
        // important  
    }  
}
```

To invoke: Permutation(a,0,n-1)

Try it with $n=3$ and $a[0:2]=(a,b,c)$



If the original configuration were not returned,



and the **b x x** permutations would be lost.

Exercises: P32-2, P33-14

1.7 Performance Analysis and Measurement

Definition: the **Space complexity** of a program is the amount of memory it needs to run to completion. The **Time complexity** of a program is the amount of computer time it needs to run to completion.

(1) Priori estimates --- Performance analysis

(2) Posteriori testing--- Performance measurement

1.7.1 Performance Analysis

1.7.1.1 Space complexity

The space requirement of program P:

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

We concentrate solely on S_p

Example 1.10

```
float Rsum (float *a, const int n) //compute  $\sum_{i=0}^{n-1} a[i]$  recursively
{
    if (n <= 0) return 0;
    else return (Rsum(a, n-1) + a[n-1]);
}
```

The instances are characterized by n , each call requires 4 words (n , a , return value, return address), the depth of recursion is $n+1$, so

$$S_{\text{rsum}}(n) = 4(n+1)$$

1.7.1.2 Time complexity

Run time of a program P:

$$t_p(\text{instance characteristics})$$

A **program step** is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of instance characteristics.

In P41-43 of the textbook, there is an detailed assignment of step counts to statements in C++.

Our main concern is on how many steps are needed by a program to solve a particular problem instance?

2 ways:

(1) count

(2) table

Example 1.12

```
count=0;
float Rsum (float *a, const int n)
{
    count++; // for if
    if (n <=0) {
        count++; // for return
        return 0;
    }
    else {
        count++; // for return
        return (Rsum(a,n-1)+a[n-1]);
    }
}
```

$$t_{\text{Rsum}}(0) = 2,$$

$$\begin{aligned} t_{\text{Rsum}}(n) &= 2 + t_{\text{Rsum}}(n-1) \\ &= 2 + 2 + t_{\text{Rsum}}(n-2) \end{aligned}$$

·

·

·

$$= 2n + t_{\text{Rsum}}(0) = 2n + 2$$

Example 1.14 Fibonacci numbers

```
1 void Fibonacci (int n)
2 { // compute the Fibonacci number  $F_n$ 
3   if (n <=1) cout << n<< endl; { //  $F_0=0$  and  $F_1=1$ 
4   else { // compute  $F_n$ 
5       int fn; int fnm2=0; int fnm1=1;
6       for (int i=2; i<=n; i++)
7       {
8           fn=fnm1+fnm2;
9           fnm2=fnm1;
10          fnm1=fn;
11      } //end of for
```

```
12     cout <<fn<<endl;  
13 } //end of else  
14 }
```

Let us use a table to count its total steps.

Line	s/e	frequency	total steps
1	0	1	0
2	0	1	0
3	1 (n > 1)	1	1
4	0	1	0
5	2	1	2
6	1	n	n
7	0	n-1	0
8	1	n-1	n-1
9	1	n-1	n-1

10	1	$n-1$	$n-1$
11	0	$n-1$	0
12	1	1	1
13	0	1	0
14	0	1	0

So

for $n > 1$, $t_{\text{Fibonacci}}(n) = 4n + 1$,

for $n = 0$ or 1 , $t_{\text{Fibonacci}}(n) = 2$

Sometime, the instance characteristics is related with the content of the input data set.

e.g., *BinarySearch*.

Hence:

- best-case
- worst-case,
- average-case.

1.7.1.3 Asymptotic Notation

Because of the inexactness of what a step stands for, we are mainly concerned with the magnitude of the number of steps.

Definition [O]: $f(n)=O(g(n))$ iff there exist positive constants c and n_0 such that $f(n) \leq c g(n)$ for all n , $n>n_0$.

Example 1.13: $3n+2=O(n)$, $6 \cdot 2^n + n^2 = O(2^n), \dots$

Note $g(n)$ is an upper bound.

$n=O(n^2)$, $n=O(2^n)$, ..., for $f(n)=O(g(n))$ to be informative, $g(n)$ should be as small as possible.

In practice, the coefficient of $g(n)$ should be 1. We never say $O(3n)$.

$O(1)$	--- constant	$O(\log_2 n)$	--- logarithm
$O(n)$	--- linear	$O(2^n)$	--- exponential
$O(n^2)$	--- quadratic		
$O(n^3)$	--- cubic		

Theory 1.2: if $f(n)=a_m n^m+\dots+a_1 n+a_0$, then $f(n)=O(n^m)$.

When the complexity of an algorithm is actually, say, $O(\log n)$, but we can only show that it is $O(n)$ due to the limitation of our knowledge, it is ok to say so. This is one benefit of O notation as upper bound.

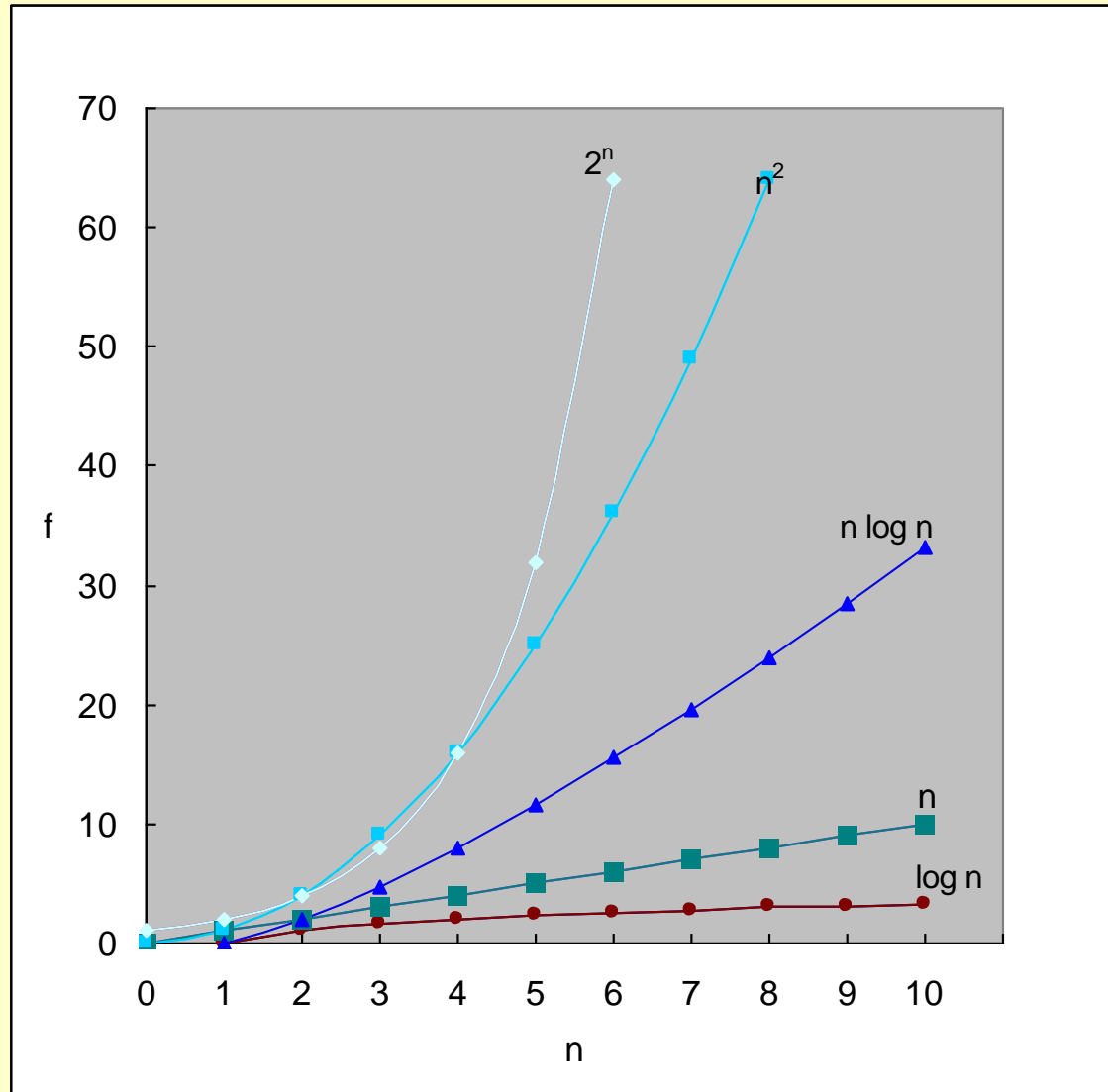
Self-study:

Ω --- low bound

Θ --- equal bound

1.7.1.4 Practical Complexity

The right figure shows how the various functions grow with n .



n	$f(n)=n$	$f(n)=n\log_2 n$	$f(n)=n^2$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01 μ s	.03 μ s	.1 μ s	10 μ s	10s	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	160 μ s	2.84h	1 ms
30	.03 μ s	.15 μ s	.9 μ s	810 μ s	6.83d	1 s
40	.04 μ s	.21 μ s	1.6 μ s	2.56ms	121d	18m
50	.05 μ s	.28 μ s	2.5 μ s	6.25ms	3.1y	13 d
100	.1 μ s	.66 μ s	10 μ s	100 ms	3171y	$4 \cdot 10^{13}$ y
10^3	1 μ s	9.66 μ s	1ms	16.67m		
10^4	10 μ s	130 μ s	100ms	115.7d		
10^5	100 μ s	1.66ms	10s	3171y		

Table 1.8: Times on a 1-billion-steps-per-second computer

1.7.2 Performance Measurement

Performance measurement is concerned with obtaining the actual space and time requirements of a program.

To time a short event it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

Let us look at the following program:

```
int SequentialSearch (int *a, const int n, const int x )  
{ // Search a[0:n-1].  
    int i;  
    for (i=0; i < n && a[i] != x; i++);  
    if (i == n) return -1;  
    else return i;  
}
```

```

void TimeSearch ( )
{
    int a[1000], n[20];
    const long r[20] = {300000, 300000, 200000, 200000,
        100000, 100000, 100000, 80000, 80000, 50000, 50000,
        25000, 15000, 15000, 10000, 7500, 7000, 6000, 5000,
        5000 };

    for ( int j=0; j<1000; j++ ) a[j] = j+1; //initialize a
    for ( j=0; j<10; j++ ) { //values of n
        n[j] = 10*j;  n[j+10] = 100*( j+1 );
    }

    cout << "n    total    runTime" << endl;
}

```

```

for ( j=0; j<20; j++ ) {
    long start, stop;
    time (&start);                                // start timer
    for ( long b=1; b<=r[j]; b++ )
        int k = seqsearch(a, n[j], 0 ); //unsuccessful search
    time (&stop);                                // stop timer
    long totalTime = stop - start;
    float runTime = (float) (totalTime) / (float)(r[j]);
    cout << " " << n[j] << " " << totalTime << " " << runTime
        << endl;
    }
}

```

The results of running *TimeSearch* are as in the next slide.

n	total	runTime	n	total	runTime
0	241	0.0008	100	527	0.0105
10	533	0.0018	200	505	0.0202
20	582	0.0029	300	451	0.0301
30	736	0.0037	400	593	0.0395
40	467	0.0047	500	494	0.0494
50	565	0.0056	600	439	0.0585
60	659	0.0066	700	484	0.0691
70	604	0.0075	800	467	0.0778
80	681	0.0085	900	434	0.0868
90	472	0.0094	1000	484	0.0968

Times in hundredths of a second, the plot of the data can be found in Fig. 1.7.

Issues to be addressed:

- (1) Accuracy of the clock
- (2) Repetition factor
- (3) Suitable test data for worst-case or average performance
- (4) Purpose: comparing or predicting?
- (5) Fit a curve through points

Exercises:

P72-10