



数据结构与算法（一）

张铭 主讲

采用教材：张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008. 6（“十一五”国家级规划教材）

<http://www.jpku.pku.edu.cn/pkujpku/course/sjjg>

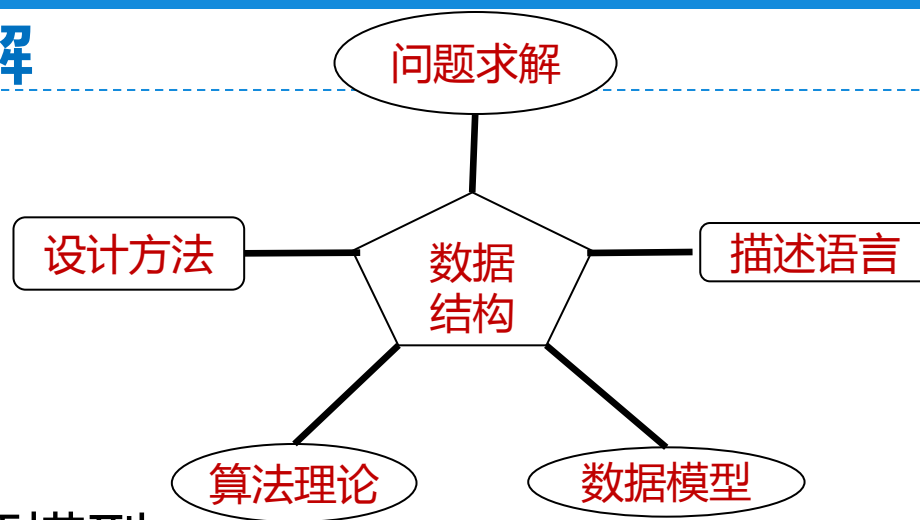


第1章 概论

- 问题求解
- 数据结构及抽象数据类型
- 算法的特性及分类
- 算法的效率度量
- 数据结构的选择和评价

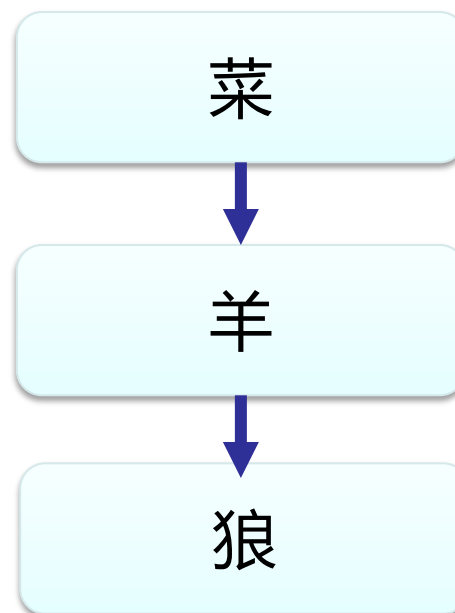
1.1 问题求解

- 编写计算机程序的目的？
 - 解决实际的应用问题
- 问题抽象
 - 分析和抽象任务需求，建立问题模型
- 数据抽象
 - 确定恰当的数据结构表示数学模型
- 算法抽象
 - 在数据模型的基础上设计合适的算法
- 数据结构 + 算法，进行程序设计
 - 模拟和解决实际问题



1.1 问题求解

农夫过河



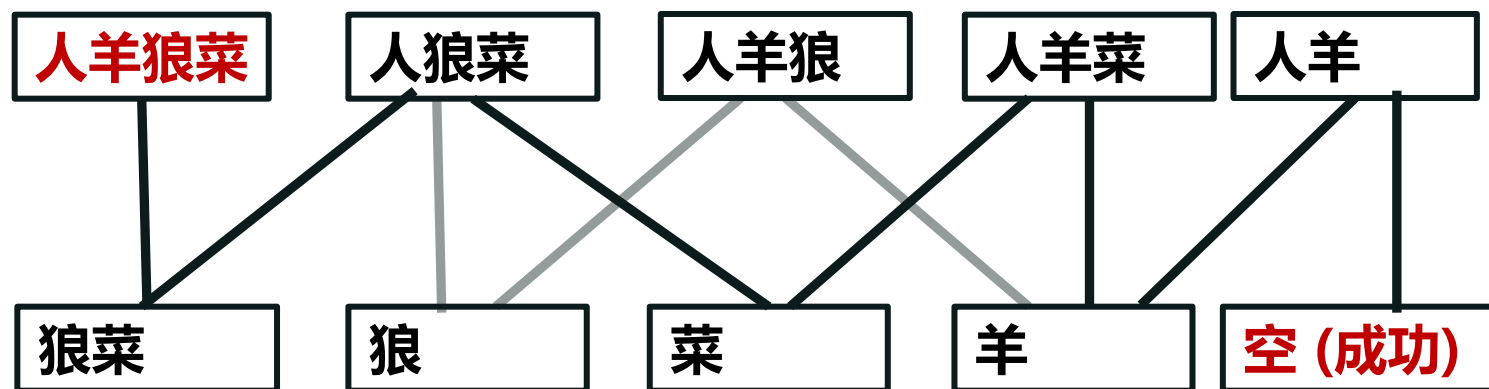
1.1 问题求解

• 问题抽象：人狼羊菜乘船过河

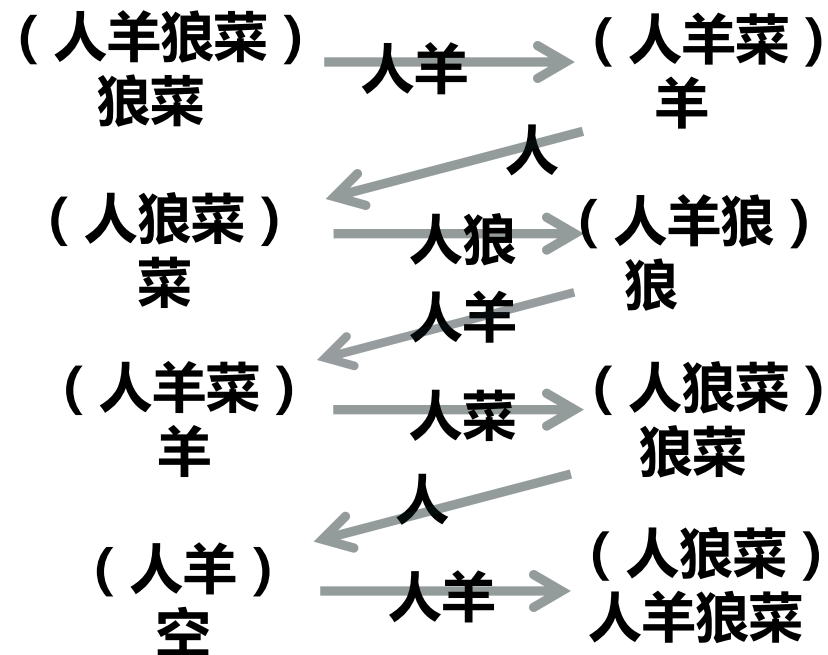
- 只有人能撑船
- 船只有两个位置（包括人）
- 狼羊、羊菜不能在没有人时共处

• 数据抽象：图模型

- 不合理状态：狼羊、人菜、羊菜、人狼、狼羊菜、人
- 顶点表示“原岸状态”——10种（包括“空”）
- 边：一次合理的渡河操作实现的状态转变



农夫过河

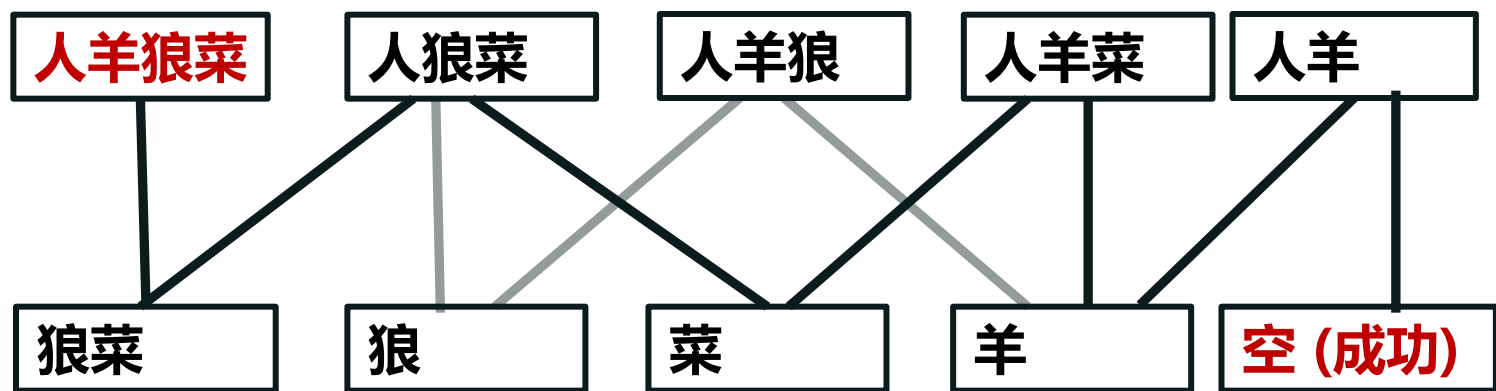


1.1 问题求解

农夫过河

- 数据结构
 - 相邻矩阵
- 算法抽象：
 - 最短路径

0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	1	1	0	0
0	0	0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	0	1	1
1	1	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0





1.1 问题求解

思考：问题求解过程

- 农夫过河问题 —— 最短路径模型
 - 问题抽象？
 - 数据抽象？
 - 算法抽象？
 - 不妨编程序模拟实现
- 还有其他模型吗？

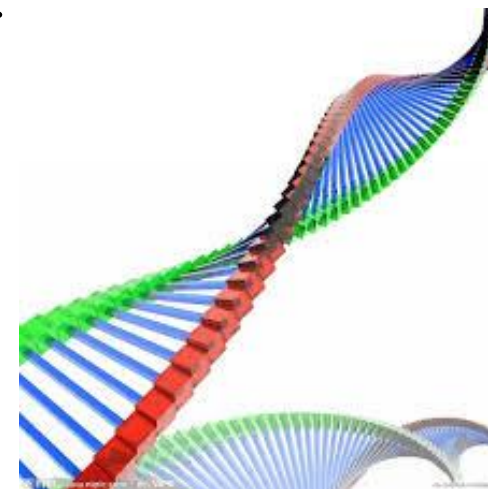
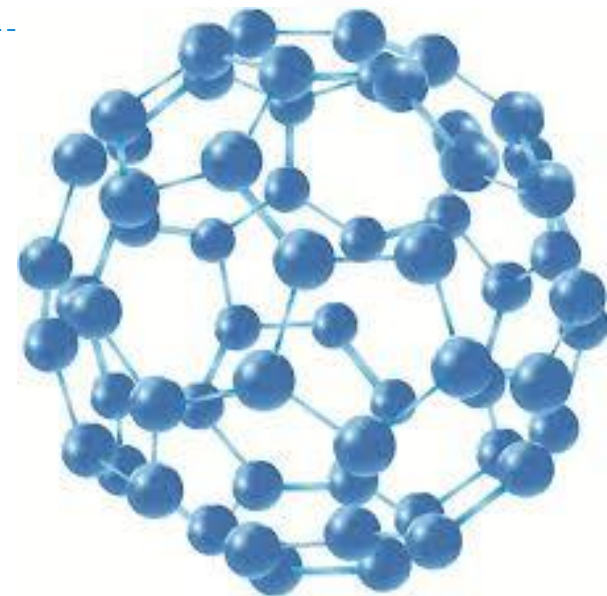
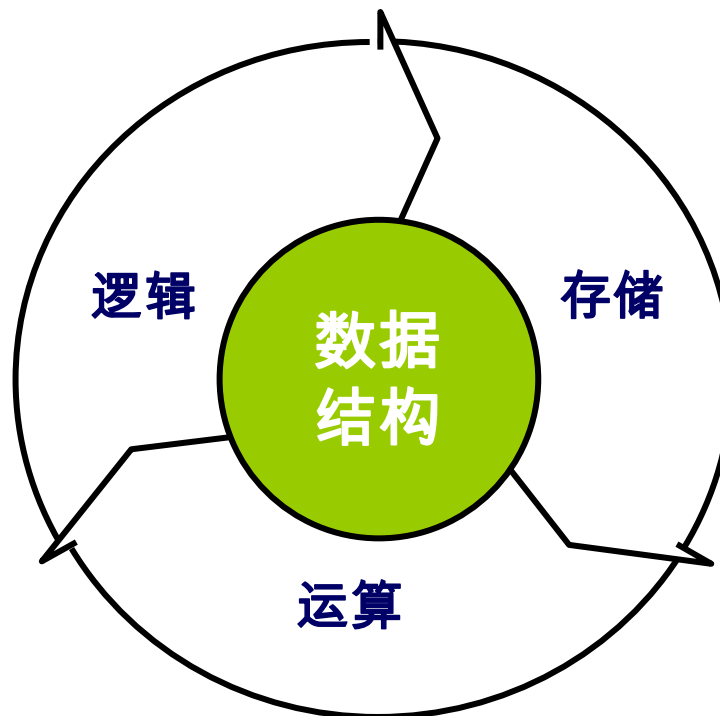


第1章 概论

- 问题求解
- 数据结构及抽象数据类型
- 算法的特性及分类
- 算法的效率度量
- 数据结构的选择和评价

1.2 什么是数据结构

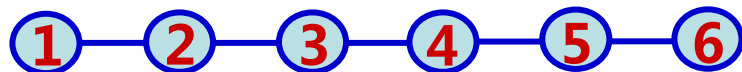
- **结构**: 实体 + 关系
- **数据结构**:
 - 按照**逻辑关系**组织起来的一批数据,
 - 按一定的**存储方法**把它存储在计算机中
 - 在这些数据上定义了一个**运算**的集合



1.2 什么是数据结构

数据结构的逻辑组织

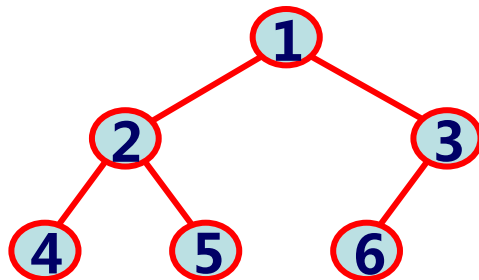
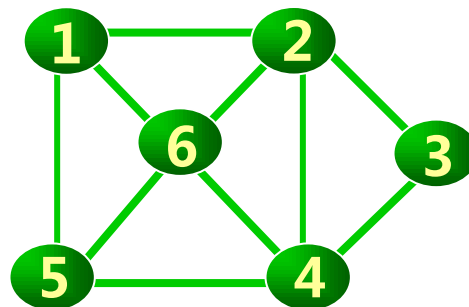
• 线性结构



- 线性表（表，栈，队列，串等）

• 非线性结构

- 树（二叉树，Huffman树，二叉检索树等）
- 图（有向图，无向图等）

• 图 \supseteq 树 \supseteq 二叉树 \supseteq 线性表

1.2 什么是数据结构

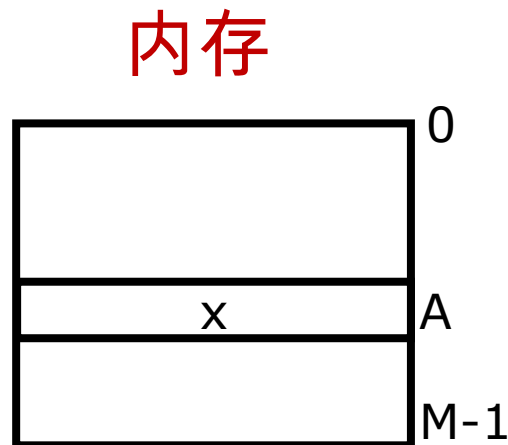
数据的存储结构

- 逻辑结构到物理存储空间的**映射**

计算机主存储器（内存）

- **非负整数**地址编码，**相邻单元**的集合

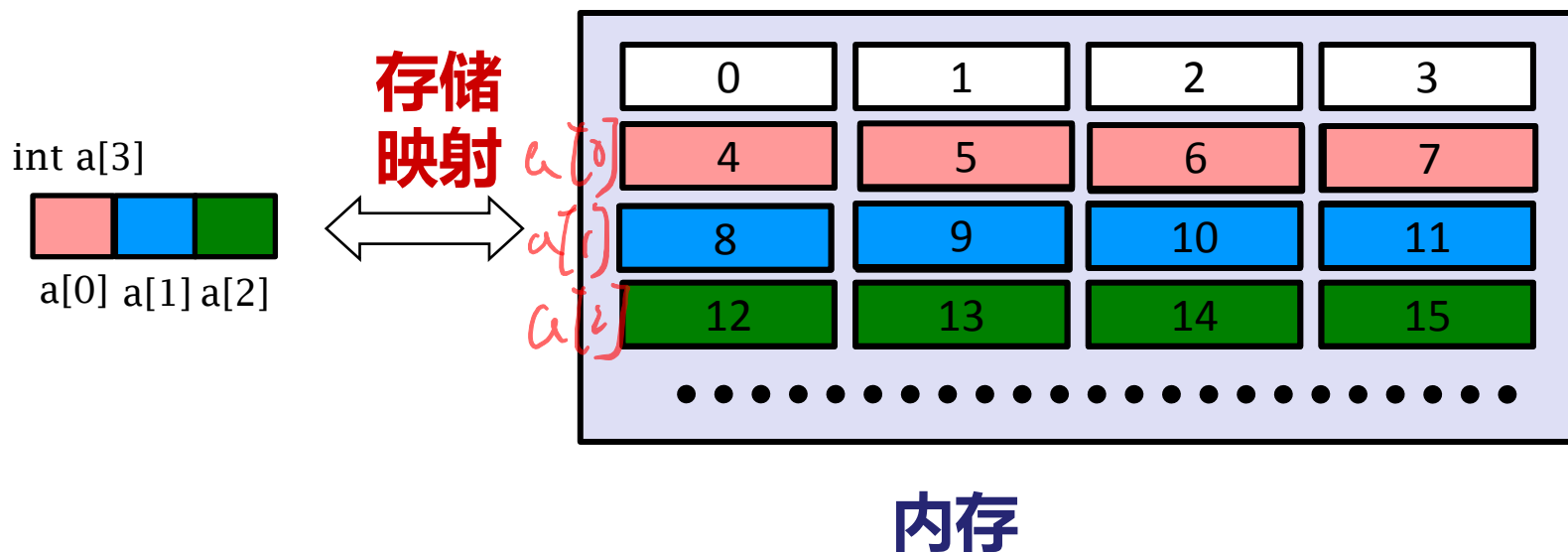
- 基本单位是字节
- 访问不同地址所需时间基本相同（即随机访问）



1.2 什么是数据结构

数据的存储结构

- 对逻辑结构 (K, r) ，其中 $r \in R$
 - 对结点集 K 建立一个从 K 到存储器 M 的单元的映射： $K \rightarrow M$ ，对于每一个结点 $j \in K$ 都对应一个**唯一的连续存储**区域 $c \in M$



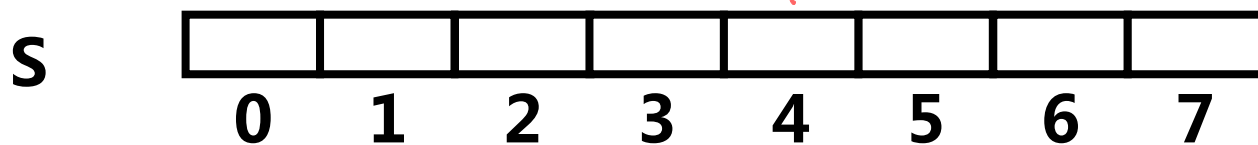
1.2 什么是数据结构

数据的存储结构

- 关系元组 $(j_1, j_2) \in r$
(其中 $j_1, j_2 \in K$ 是结点)

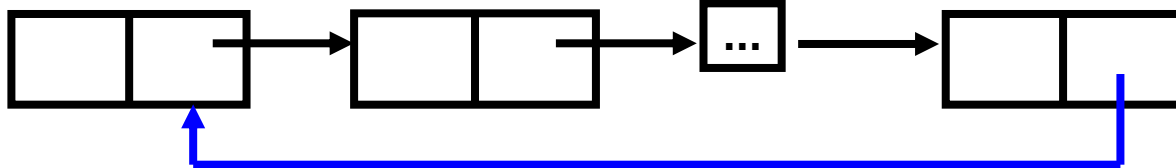
- 顺序：存储单元的顺序地址

线性化



- 链接：指针的地址指向关系

(非线性的主要组织方式)

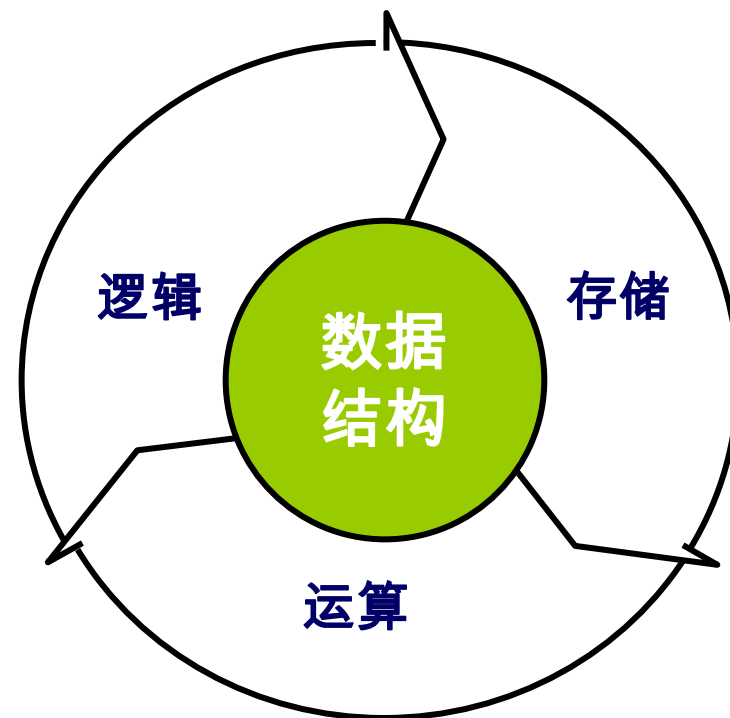


- 四类：**顺序**、**链接**、**索引**、**散列** ★

1.2 什么是数据结构

抽象数据类型

- 简称**ADT** (Abstract Data Type)
 - 定义了一组运算的数学模型
 - 与物理存储结构无关
 - 使软件系统建立在数据之上(面向对象)
- **模块化**的思想的发展
 - 隐藏运算实现的细节和内部数据结构
 - 软件复用





1.2 什么是数据结构

ADT 不关心存储细节

—— 例，C++ 版本括号匹配算法

```
void BracketMatch(char *str) {  
    Stack<char> S; int i; char ch;  
    // 栈可以是顺序或链式的，都一样引用  
    for(i=0; str[i]!='\0'; i++) {  
        switch(str[i]) {  
            case '(': case '[': case '{':  
                S.Push(str[i]); break;  
            case ')': case ']': case '}':  
                if (S.IsEmpty( )) {  
                    cout<<"右括号多余!";  
                    return;  
                }  
            else {
```

```
                ch = S.GetTop( );  
                if (Match(ch,str[i]))  
                    ch = S.Pop( );  
                else {  
                    cout << "括号不匹配!";  
                    return;  
                }  
            } /*else*/  
        } /*switch*/  
    } /*for*/  
    if (S.IsEmpty( ))  
        cout<<"括号匹配!";  
    else cout<<"左括号多余";  
}
```

1.2 什么是数据结构

C 的顺序栈括号匹配算法 (与链式略不同)

```
void BracketMatch(char *str) {  
    SeqStack S; int i; char ch;  
    InitStack(&S);  
    for(i=0; str[i]!='\0'; i++) {  
        switch(str[i]) {  
            case '(': case '[': case '{':  
                Push(&S, str[i]); break;  
            case ')': case ']': case '}':  
                if (IsEmpty(&S)) {  
                    printf("\n右括号多余!");  
                    return;  
                }  
            else {
```

```
                GetTop (&S,&ch);  
                if (Match(ch,str[i]))  
                    Pop(&S,&ch);  
                else {  
                    printf("\n括号不匹配!");  
                    return;  
                }  
            } /*else*/  
        } /*switch*/  
    } /*for*/  
    if (IsEmpty(&S))  
        printf("\n括号匹配!");  
    else printf("\n左括号多余" );  
}
```




1.2 什么是数据结构

C 的链式栈括号匹配算法 (与顺序栈不同)

```
void BracketMatch(char *str) {
    LinkStack S; int i; char ch;
    InitStack(&S);
    for(i=0; str[i]!='\0'; i++) {
        switch(str[i]) {
            case '(': case '[': case '{':
                Push(&S, str[i]);
                break;
            case ')': case ']': case '}':
                if (IsEmpty(S)) {
                    printf("\n右括号多余!");
                    return;
                }
                else {
```

```
                GetTop (&S,&ch);
                if (Match(ch,str[i]))
                    Pop(&S,&ch);
                else {
                    printf("\n括号不匹配!");
                    return;
                }
            } /*else*/
        } /*switch*/
    } /*for*/
    if (IsEmpty(S))
        printf("\n括号匹配!");
    else printf("\n左括号多余" );
}
```



1.2 什么是数据结构

抽象数据类型ADT

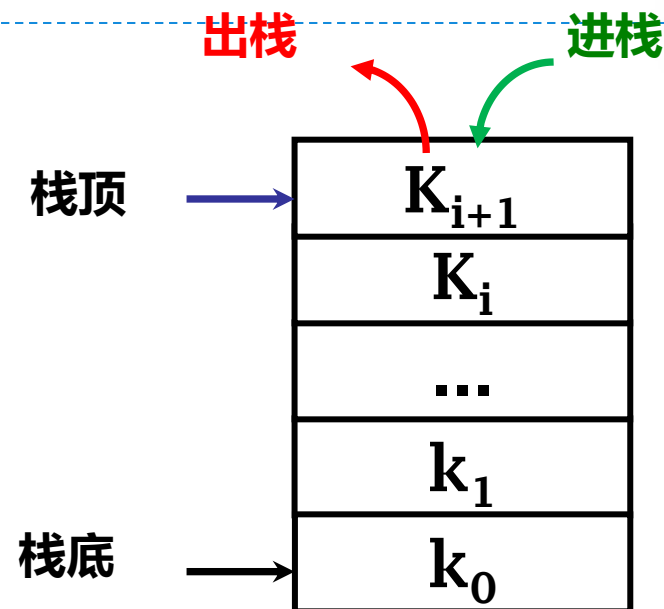
- 抽象数据结构二元组
 <数据对象D, 数据操作P>
- 先定义逻辑结构, 再定义运算
 - **逻辑结构**: 数据对象及其关系
 - **运算**: 数据操作

1.2 什么是数据结构

例：栈的抽象数据类型ADT

- 逻辑结构：线性表
- 操作特点：**限制访问端口**
 - 只允许在一端进行插入、删除操作
 - 入栈 (push)、出栈 (pop)、取栈顶 (top)
 - 判栈空 (isEmpty)

```
template <class T>           // 栈的元素类型为 T
class Stack {
public:                       // 栈的运算集
    void clear();            // 变为空栈
    bool push(const T item); // item入栈，成功返回真，否则假
    bool pop(T & item);      // 弹栈顶，成功返回真，否则返回假
    bool top(T& item);       // 读栈顶但不弹出，成功真，否则假
    bool isEmpty();          // 若栈已空返回真
    bool isFull();           // 若栈已满返回真
};
```





1.2 什么是数据结构

思考：关于抽象数据类型ADT

- 怎么体现逻辑结构？
- 抽象数据类型等价于类定义？
- 不用模板来定义可以描述 ADT 吗？



第1章 概论

- 问题求解
- 数据结构及抽象数据类型
- **算法的特性及分类**
- 算法的效率度量



问题 —— 算法 —— 程序

目标：问题求解

- **问题 (problem)** 一个函数
 - 从输入到输出的一种映射
- **算法 (algorithm)** 一种方法
 - 对特定问题求解过程的描述，是指令的有限序列
- **程序 (program)**
 - 是算法在计算机程序设计语言中的实现

1.3 算法

算法的特性

	Q		
			Q
Q			
		Q	

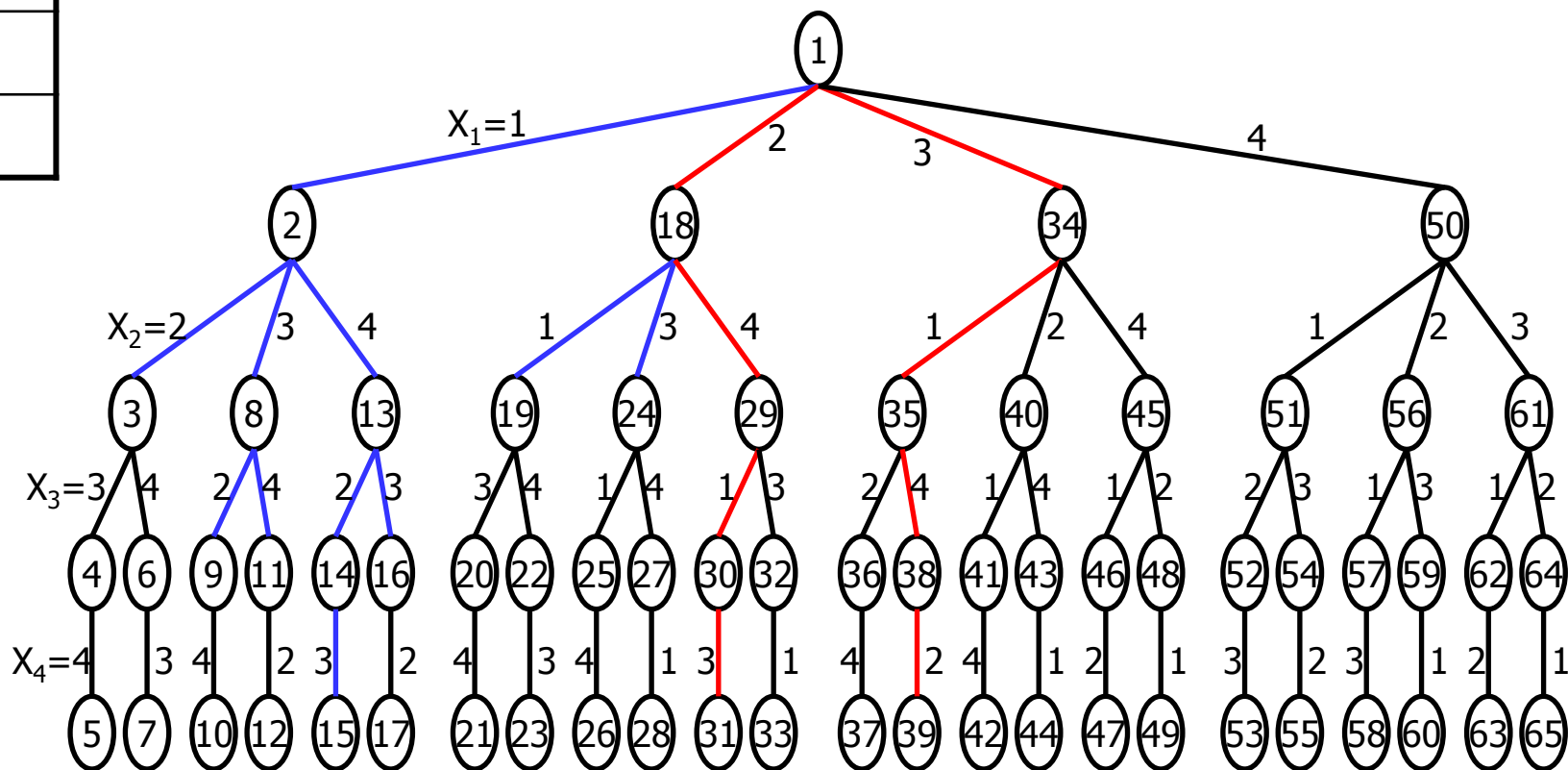
- 通用性
 - 对参数化输入进行问题求解
 - 保证计算结果的正确性
- 有效性
 - 算法是有限条指令组成的指令序列
 - 即由一系列具体步骤组成
- 确定性
 - 算法描述中的下一步应执行的步骤必须明确
- 有穷性
 - 算法的执行必须在有限步内结束
 - 换句话说，算法不能含有死循环

1.3 算法

皇后问题（四皇后）

- 解 $\langle x_1, x_2, x_3, x_4 \rangle$ （放置列号）
- 搜索空间：4叉树（排列树）

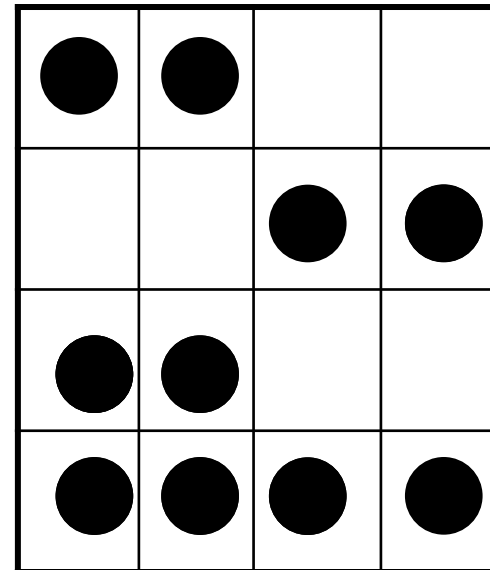
	Q		
			Q
Q			
		Q	



1.3 算法

基本算法分类

- **穷举法**
 - 顺序找 K 值
- **回溯、搜索**
 - 八皇后、树和图遍历
- **递归分治**
 - 二分找 K 值、快速排序、归并排序
- **贪心法**
 - Huffman 编码树、最短路 Dijkstra 算法、最小生成树 Prim 算法
- **动态规划**
 - 最短路 Floyd 算法





1.3 算法

顺序找k值

```
template <class Type>
class Item {
private:
    Type key;
```

```
public:
    Item(Type value):key(value) {}
    Type getKey() {return key;}
    void setKey(Type k){ key=k;}
};
```

```
vector<Item<Type>*> dataList;
```

```
template <class Type> int SeqSearch(vector<Item<Type>*>& dataList, int length,
Type k) {
    int i=length;
    dataList[0]->setKey (k);
    while(dataList[i]->getKey() !=k) i--;
    return i;
}
```

```
// 关键码域
// 其它域
```

```
// 取关键码值
// 置关键码
```

```
// 将第0个元素设为待检索值，设监视哨
```

```
// 返回元素位置
```

0	1	2	3	4	5	6	7	8
	17	35	22	18	93	60	88	52



1.3 算法

二分法找 k 值

对于已排序顺序线性表

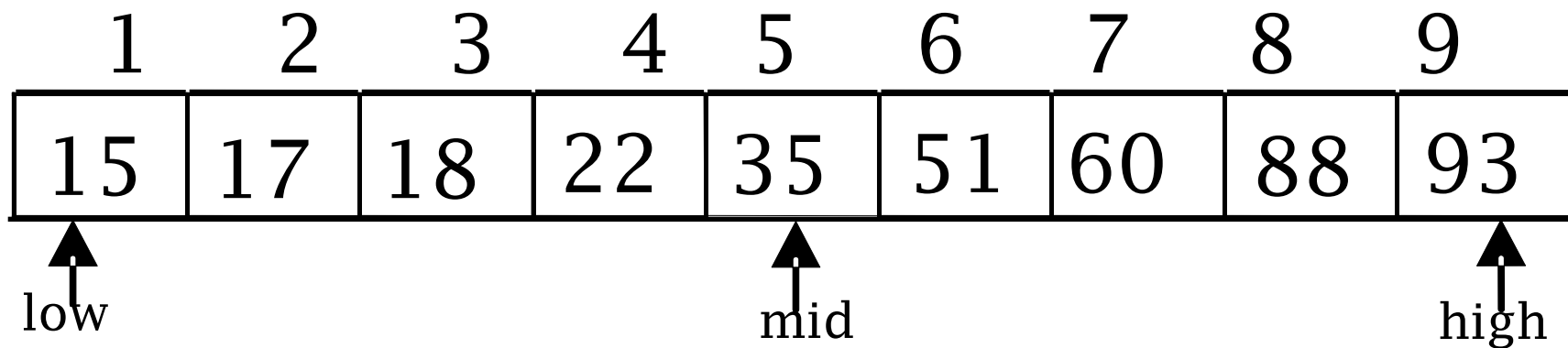
- 数组中间位置的元素值 k_{mid}
 - 如果 $k_{\text{mid}} = k$ ，那么检索工作就完成了
 - 当 $k_{\text{mid}} > k$ 时，检索继续在前半部分进行
 - 相反地，若 $k_{\text{mid}} < k$ ，就可以忽略 mid 以前的那部分，检索继续在后半部分进行
- 快速
 - $k_{\text{mid}} = k$ 结束
 - $k_{\text{mid}} \neq k$ 起码缩小了一半的检索范围

1.3 算法

二分法找 k 值

```
template <class Type> int BinSearch (vector<Item<Type>*>& dataList,
int length, Type k){
    int low=1, high=length, mid;
    while (low<=high) {
        mid=(low+high)/2;
        if (k<dataList[mid]->getKey())
            high = mid-1;           // 右缩检索区间
        else if (k>dataList[mid]->getKey())
            low = mid+1;           // 左缩检索区间
        else return mid;           // 成功返回位置
    }
    return 0;                       // 检索失败，返回0
}
```

二分法检索图示



检索关键码18 low=1 high=9 K=18

第一次 : mid=5; array[5]=35>18
high=4; (low=1)

第二次：mid=2; array[2]=17<18
low=3; (high=4)

```
第三次：mid=3; array[3]=18=18
        mid=3 ; return 3
```

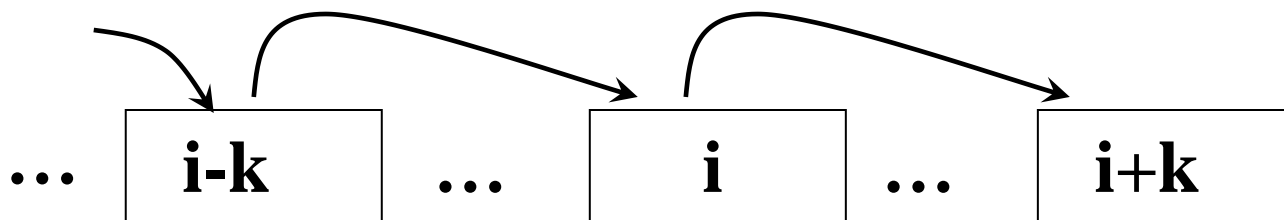
1.3 算法

思考：算法的时空限制

设计一个算法，将数组 $A(0..n-1)$ 中的元素循环右移 k 位，假设原数组序列为 $a_0, a_1, \dots, a_{n-2}, a_{n-1}$ ；移动后的序列为 $a_{n-k}, a_{n-k+1}, \dots, a_0, a_1, \dots, a_{n-k-1}$ 。要求只用一个元素大小的附加存储，元素移动或交换次数与 n 线性相关。例如， $n=10, k=3$

原始数组：0 1 2 3 4 5 6 7 8 9

右移后的：7 8 9 0 1 2 3 4 5 6





第1章 概论

- 问题求解
- 数据结构及抽象数据类型
- 算法的特性及分类
- **算法的效率度量**



1.4 算法复杂性分析

算法的渐进分析

$$f(n) = n^2 + 100n + \log_{10} n + 1000$$

- 数据规模 n 逐步增大时，
 $f(n)$ 的增长趋势
- 当 n 增大到一定值以后，计算公式中影响最大的就是 n 的幂次最高的项
 - 其他的常数项和低幂次项都可以忽略



1.4 算法复杂性分析

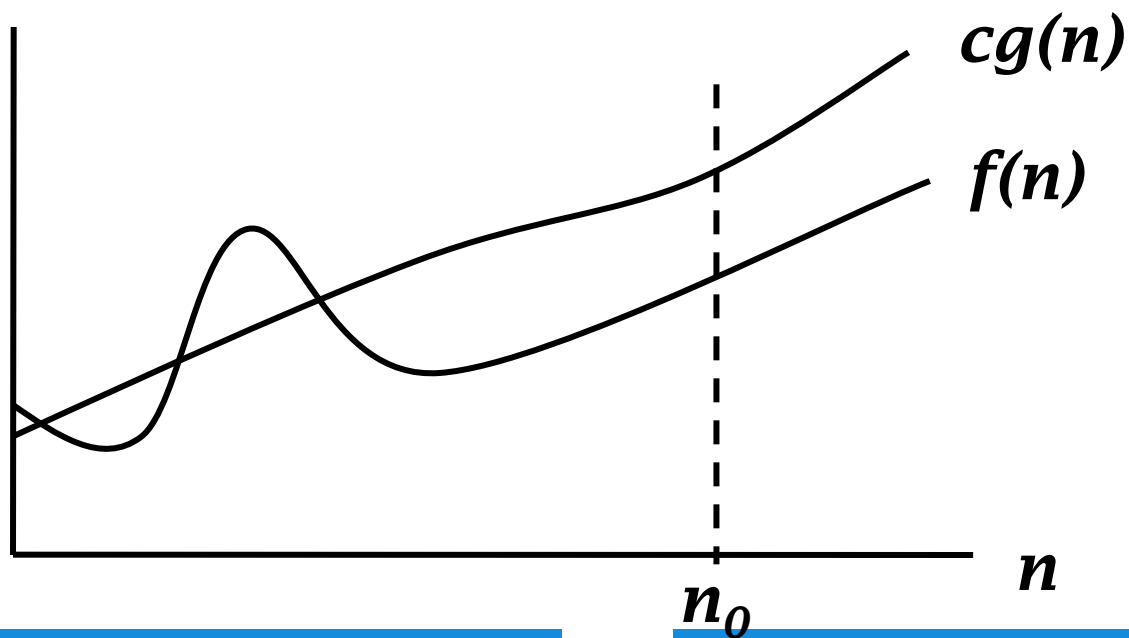
算法渐进分析：大O表式法

- 函数 f, g 定义域为自然数，值域非负实数集
- **定义**：如果存在正数 c 和 n_0 ，使得对任意的 $n \geq n_0$ ，都有 $f(n) \leq cg(n)$ ，
- 称 $f(n)$ 在集合 $O(g(n))$ 中，简称 $f(n)$ 是 $O(g(n))$ 的，或 $f(n) = O(g(n))$
- 大 O 表示法：表达函数增长率上限
 - 一个函数增长率的上限可能不止一个
- 当上、下限相同时则可用 Θ 表示法

1.4 算法复杂性分析

大 O 表示法

- $f(n) = O(g(n))$, 当且仅当
 - 存在两个参数 $c > 0$, $n_0 > 0$, 对于所有的 $n \geq n_0$, 都有 $f(n) \leq cg(n)$
- iff $\exists c, n_0 > 0$ s.t. $\forall n \geq n_0 : 0 \leq f(n) \leq cg(n)$



n 足够大
 $g(n)$ 是 $f(n)$ 的上界



1.4 算法复杂性分析

大 O 表示法的单位时间

- 简单布尔或算术运算
- 简单 I/O
 - 指函数的输入/输出
例如，从数组读数据等操作
 - 不包括键盘文件等 I/O
- 函数返回

1.4 算法复杂性分析

大 O 表示法的运算法则

- **加法规则:** $f_1(n) + f_2(n) = O(\max(f_1(n), f_2(n)))$
 - 顺序结构, if 结构, switch 结构
- **乘法规则:** $f_1(n) f_2(n) = O(f_1(n) f_2(n))$
 - for, while, do-while 结构

```
for (i=0; j<n; i++)
```

```
    for (j=i; j<n; j++)
```

```
        k++;
```

}
}

$n - i$

?

$$\sum_{i=0}^{n-1} (n - i) = \frac{n(n-1)}{2} = \frac{n^2 - n}{2} = O(n^2)$$



1.4 算法复杂性分析

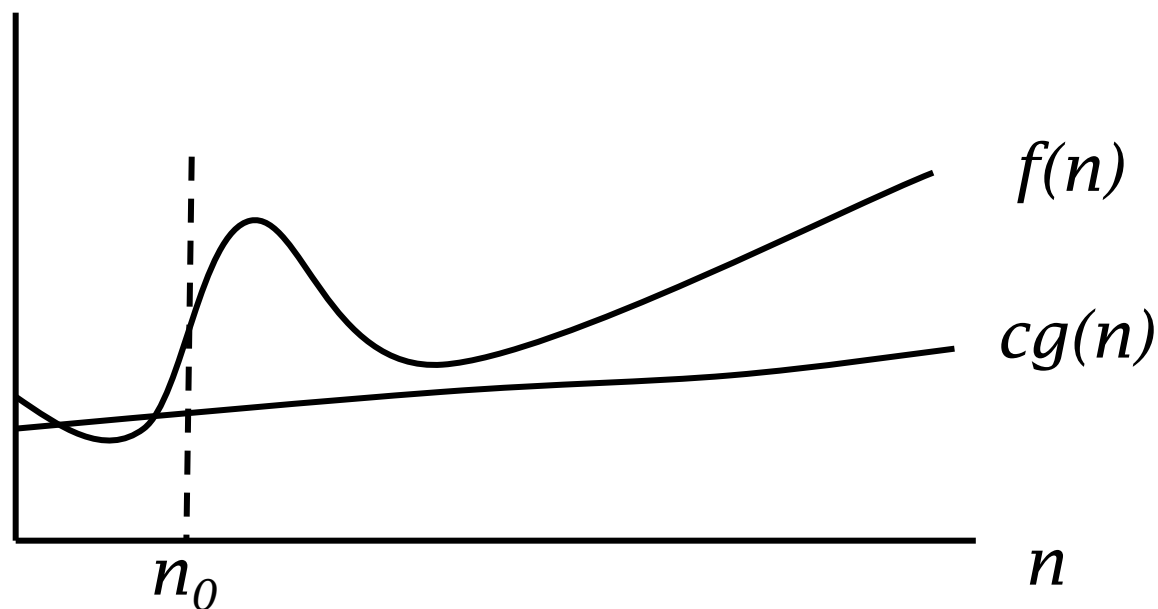
算法渐进分析：大 Ω 表式法

- **定义**：如果存在正数 c 和 n_0 ，使得对所有的 $n \geq n_0$ ，都有 $f(n) \geq cg(n)$ ，
则称 $f(n)$ 在集合 $\Omega(g(n))$ 中，或简称 $f(n)$ 是 $\Omega(g(n))$ 的，或 $f(n) = \Omega(g(n))$
- 大 O 表示法和大 Ω 表示法的唯一区别在于不等式的方向而已
- 采用大 Ω 表示法时，最好找出在函数增值率的所有下限中那个最“紧”（即最大）的下限

1.4 算法复杂性分析

大 Ω 表示法

- $f(n) = \Omega(g(n))$
 - iff $\exists c, n_0 > 0$ s.t. $\forall n \geq n_0, 0 \leq cg(n) \leq f(n)$
- 与大O表示法的唯一区别在于不等式的方向



n 足够大
 $g(n)$ 是 $f(n)$ 的下界



1.4 算法复杂性分析

算法渐进分析：大 Θ 表式法

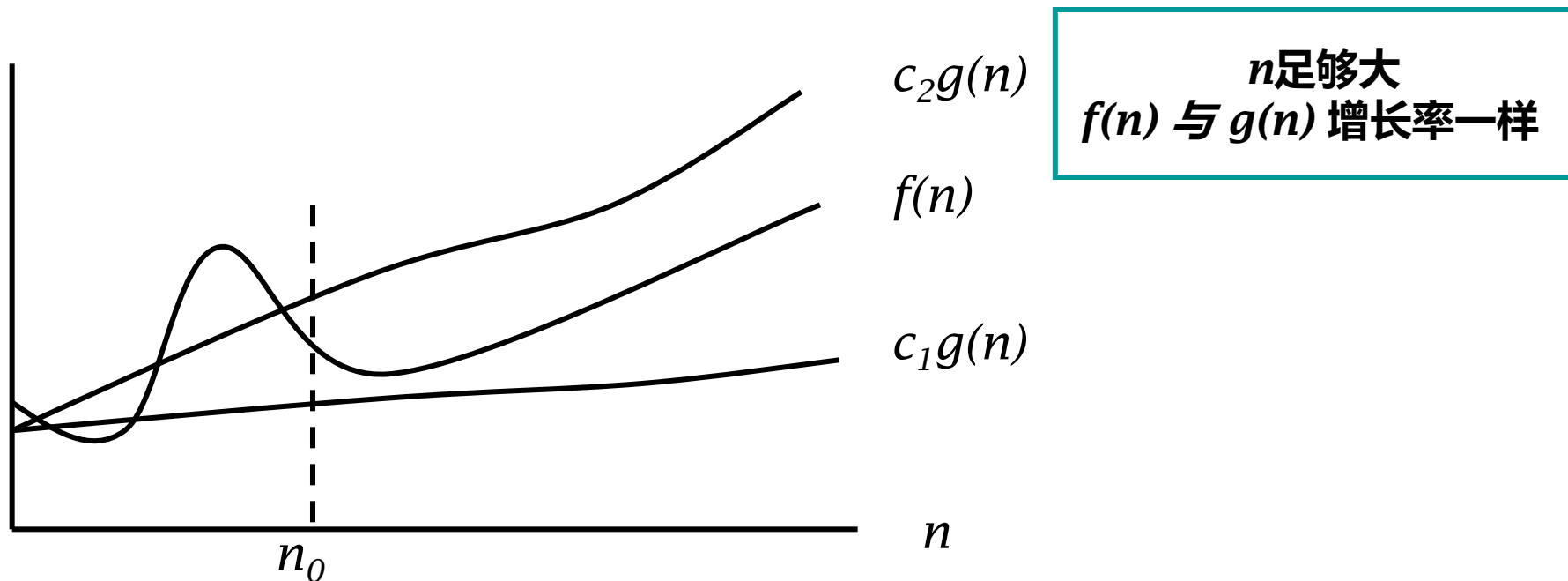
- 当上、下限相同时则可用 Θ 表示法
- 定义如下：
如果一个函数既在集合 $O(g(n))$ 中又在集合 $\Omega(g(n))$ 中，则称其为 $\Theta(g(n))$ 。
- 也即，当上、下限相同时则可用大 Θ 表示法
- 存在正常数 c_1, c_2 ，以及正整数 n_0 ，使得对于任意的正整数 $n > n_0$ ，有下列两不等式同时成立：

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

1.4 算法复杂性分析

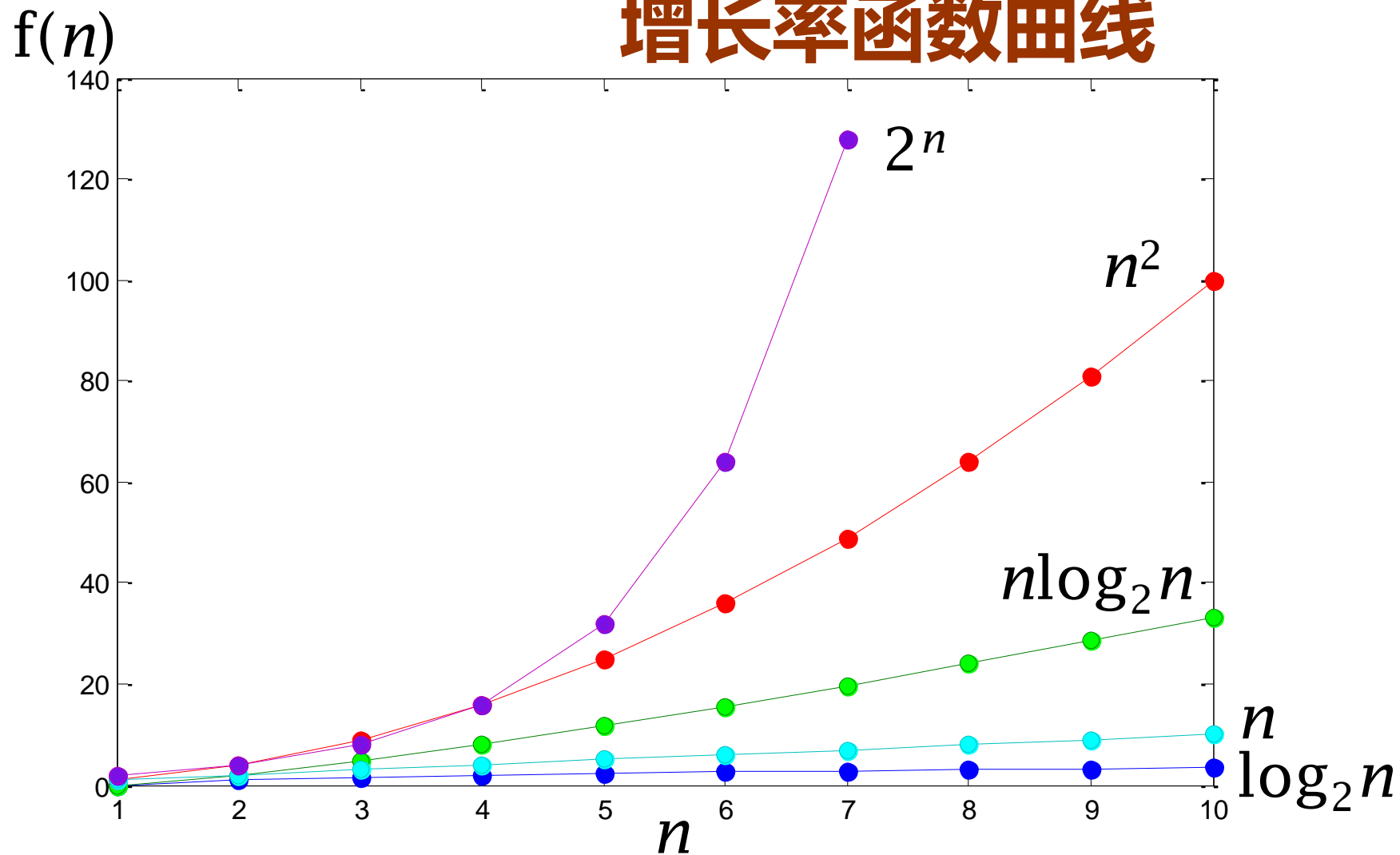
大 Θ 表示法

- $f(n) = \Theta(g(n))$
 - iff $\exists c_1, c_2, n_0 > 0$ s.t. $0 \leq c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0$
- 上、下限相同，则可用 Θ 表示法



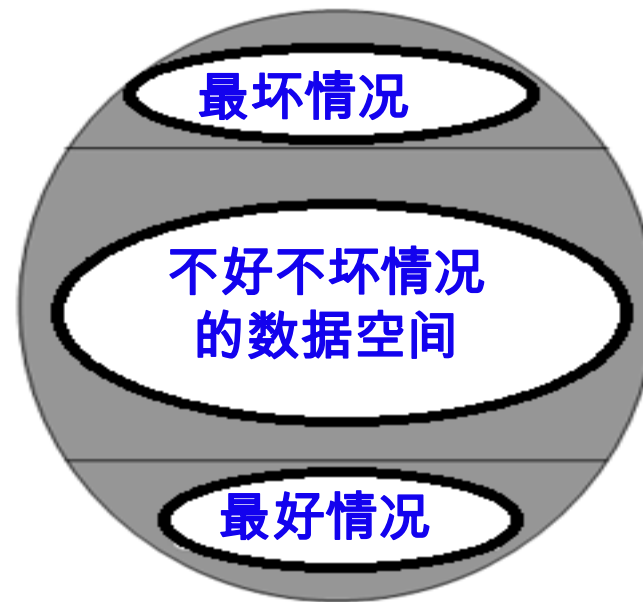
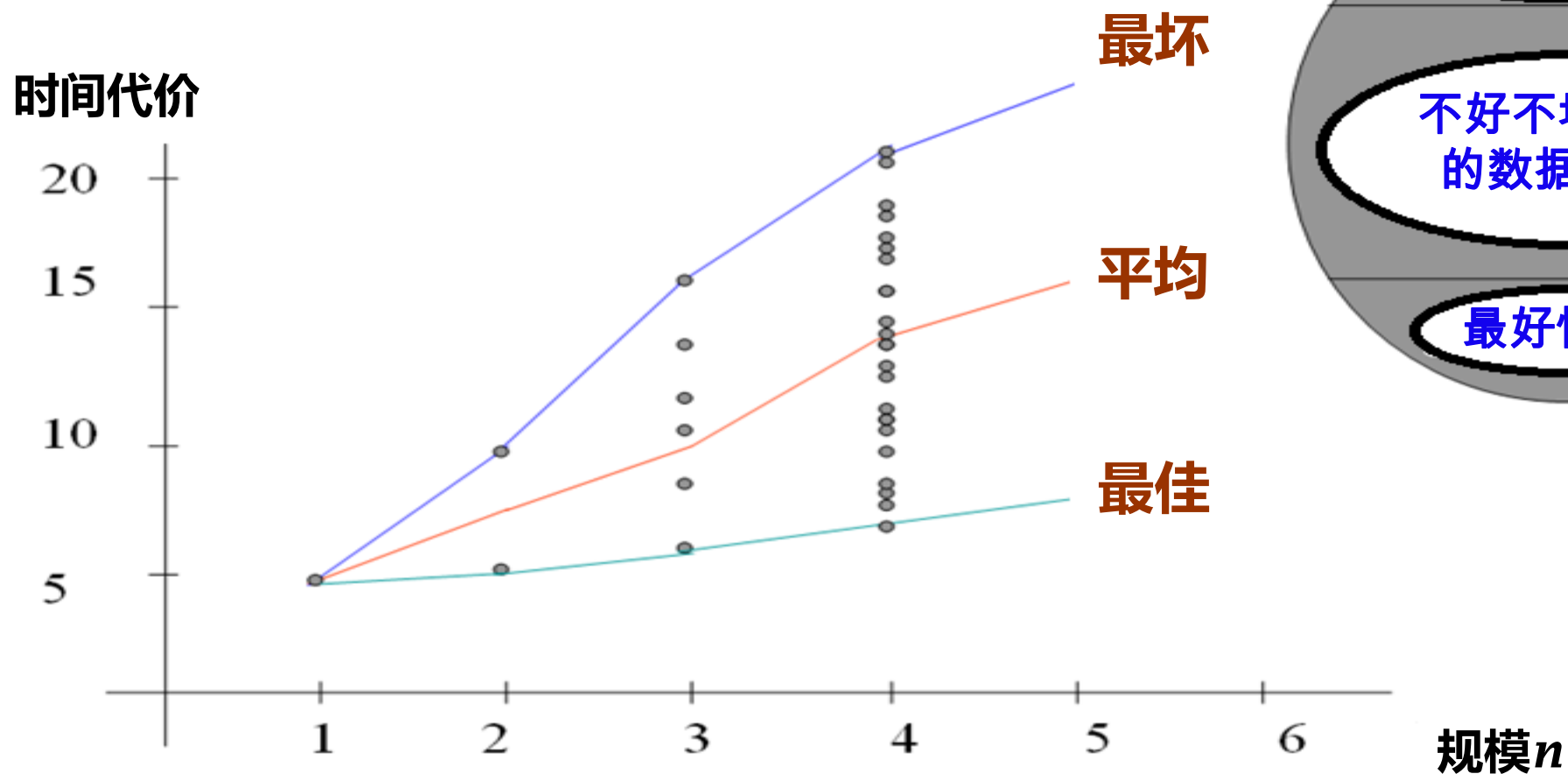
1.4 算法复杂性分析

增长率函数曲线



问题的输入数据空间

问题空间 vs 时间开销





1.4 算法复杂性分析

顺序找 k 值

- 顺序从一个规模为 n 的一维数组中找出一个给定的 K 值
- 最佳情况
 - 数组中第 1 个元素就是 K
 - 只要检查一个元素
- 最差情况
 - K 是数组的最后一个元素
 - 检查数组中所有的 n 个元素



顺序找 k 值——平均情况

- 如果等概率分布
 - K 值出现在 n 个位置上概率都是 $1/n$
- 则平均代价为 $O(n)$

$$\frac{1 + 2 + \dots + n}{n} = \frac{n + 1}{2}$$

1.4 算法复杂性分析

顺序找 k 值——平均情况

- 概率不等
 - 出现在第 1 个位置的概率为 $1/2$
 - 第 2 个位置上的概率为 $1/4$
 - 出现在其他位置的概率都是

$$\frac{1 - 1/2 - 1/4}{n - 2} = \frac{1}{4(n - 2)}$$

- 平均代价为 $O(n)$

$$\frac{1}{2} + \frac{2}{4} + \frac{3 + \dots + n}{4(n - 2)} = 1 + \frac{n(n + 1) - 6}{8(n - 2)} = 1 + \frac{n + 3}{8}$$



1.3 算法

二分法找 k 值

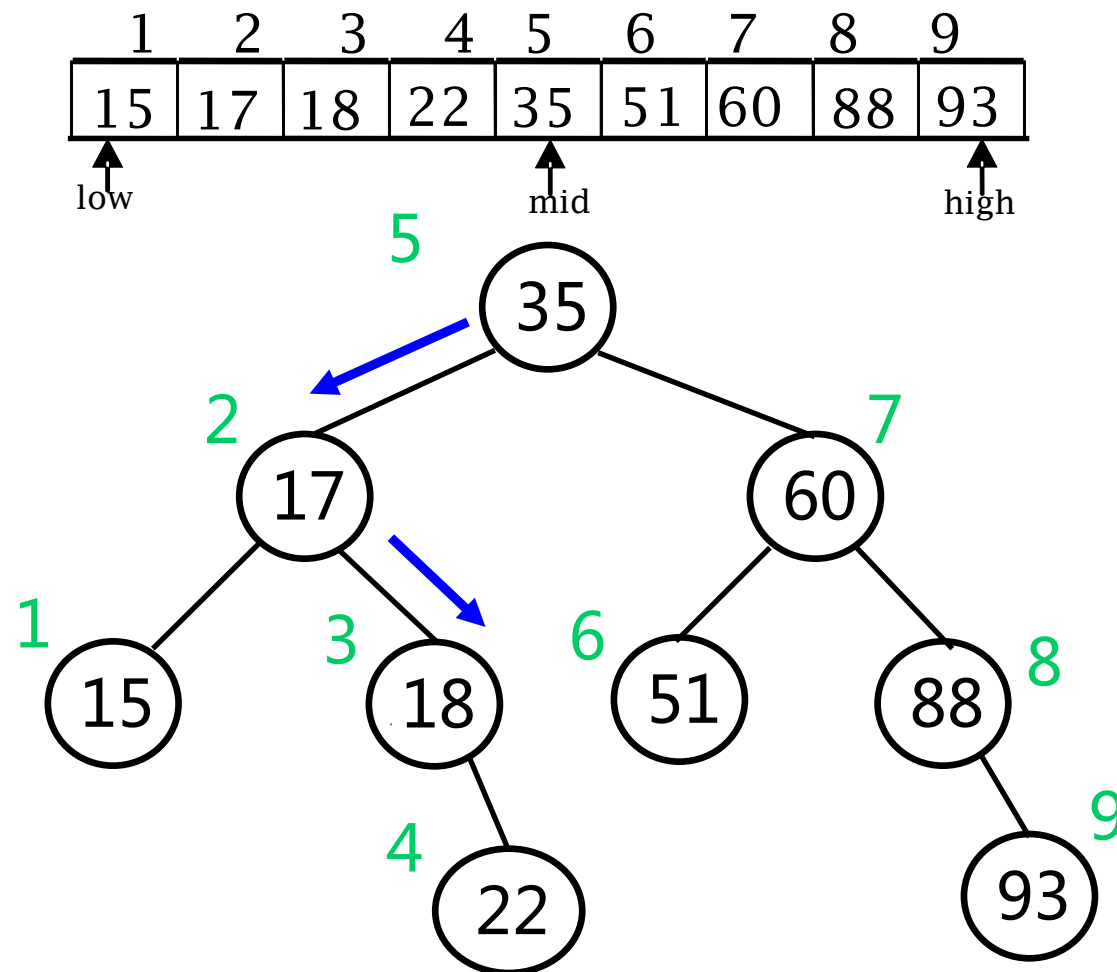
对于已排序顺序线性表

- 数组中间位置的元素值 k_{mid}
 - 如果 $k_{\text{mid}} = k$ ，那么检索工作就完成了
 - 当 $k_{\text{mid}} > k$ 时，检索继续在前半部分进行
 - 相反地，若 $k_{\text{mid}} < k$ ，就可以忽略 mid 以前的那部分，检索继续在后半部分进行
- 快速
 - $k_{\text{mid}} = k$ 结束
 - $k_{\text{mid}} \neq k$ 起码缩小了一半的检索范围

1.4 算法复杂性分析

二分法检索性能分析

- 最大检索长度为 $\lceil \log_2 (n+1) \rceil$
- 失败的检索长度是 $\lceil \log_2 (n+1) \rceil$
或 $\lfloor \log_2 (n+1) \rfloor$
- 平均检索代价为 $O(\log n)$
- 在算法复杂性分析中
 - $\log n$ 是以 2 为底的对数
 - 以其他数值为底，算法量级不变





1.4 算法复杂性分析

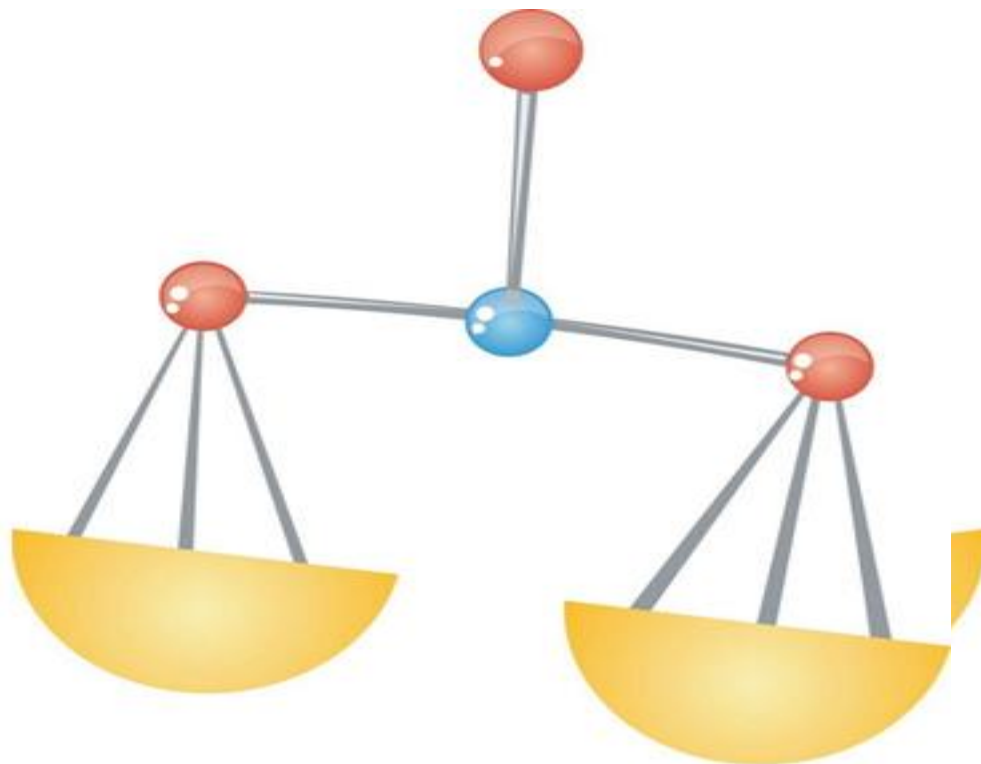
时间/空间权衡

- **数据结构**
 - 一定的空间来存储它的每一个数据项
 - 一定的时间来执行单个基本操作
- **代价和效益**
 - 空间和时间的限制
 - 软件工程

1.4 算法复杂性分析

时空权衡

- 增大空间开销可能改善算法的时间开销
- 可以节省空间，往往需要增大运算时间





1.4 算法复杂性分析

数据结构和算法的选择

- 仔细分析所要解决的问题
 - 特别是求解问题所涉及的数据类型和数据间逻辑关系
 - 问题抽象、数据抽象
 - 数据结构的初步设计往往先于算法设计
- 注意数据结构的可扩展性
 - 考虑当输入数据的规模发生改变时，
数据结构是否能够适应求解问题的演变和扩展



1.4 算法复杂性分析

思考：数据结构和算法的选择

- 问题求解的目标？
- 数据结构与算法选择的过程？



1.4 算法复杂性分析

思考：数据结构的三要素

以下哪几种结构是逻辑结构，而与存储和运算无关（ ____ ）。

- A. 顺序表
- B. 散列表
- C. 线性表
- D. 单链表

下面术语（ ____ ）与数据的存储结构无关。

- A. 顺序表
- B. 链表
- C. 队列
- D. 循环链表



数据结构与算法

谢谢聆听

国家精品课 “数据结构与算法”

<http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/>

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。 “十一五” 国家级规划教材