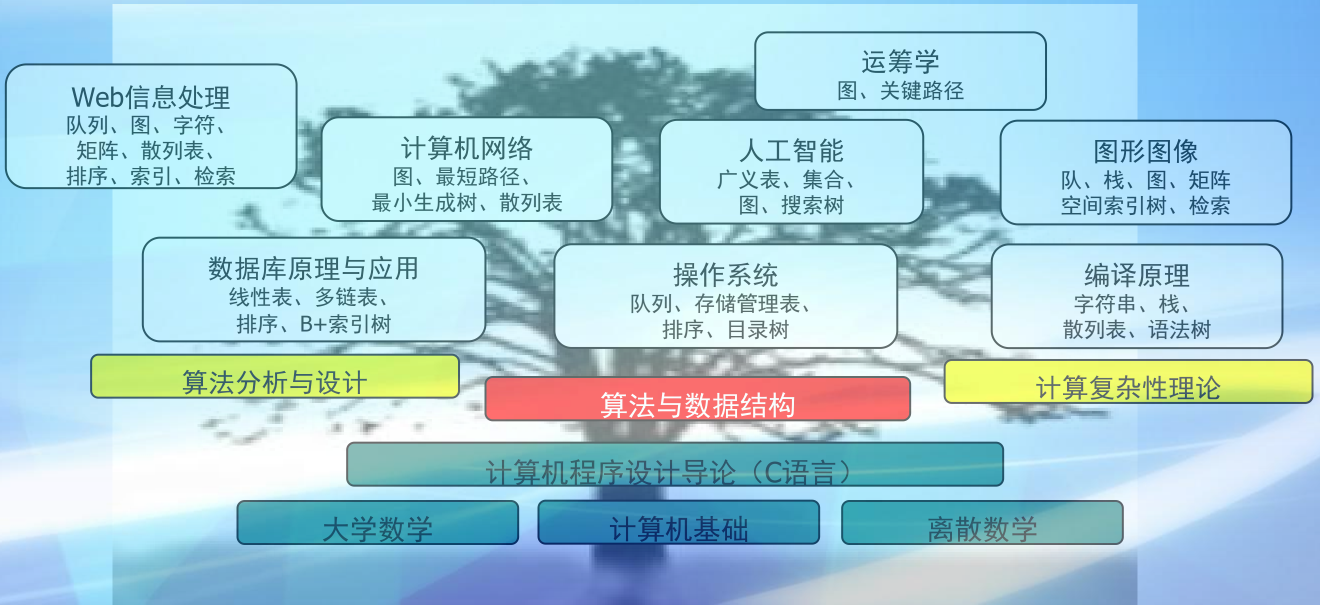
算法的评价

《数据结构》

主讲人：陈卫卫



斐波那契函数

$$f_i = \begin{cases} 1 & i = 0,1 \\ f_{i-1} + f_{i-2} & i \geq 2 \end{cases}$$

方法一：递归方法

```
  int fib(int n)
  {
1.    if (n < 2)
2.        f = n;  // base cases
3.    else
4.        f = fib( n-1 )+ fib( n-2 );
5.    return f;
   }
```

方法二：递推方法

```
  int fib(int n)
  { int  f0=1,f1=1,f2, i=2;
1.   if (n < 2)
2.       return( n);  // base cases
3.   while (i<=n)
4.      {   f2=f1+f0;
5.          f0=f1, f1=f2;
6.          i++;
       }
7.   return f;      }
```

学习目标和要求

1.算法的评价标准

2.大O记号和常用阶的高低

3.时间复杂性的计算方法

算法的评价标准

- 算法评价（评估，评测）称为算法分析（Algorithm analysis）
- 算法的评价标准

（1）算法的**正确性**

（2）算法的**有效性**

算法的正确性

能满足具体问题的需求，且对所有的合法的输入数据都正确

- 算法的正确性是最起码的，也是最重要的
- 一个正确的算法应当对所有合法的输入数据都能“计算”出正确的结果。

例，一个排序算法，只有对任意n个数据都能完成排序工作的算法才是正确的排序算法

算法的正确性

如何证明算法的正确性？

评价方法：

- **人工证明**

  归纳法

- **调试**

  精心挑选具有“代表性”的数据

  只能验证算法有错，不能证明算法无错

算法的有效性

- 算法的**时间复杂性**（time complexity）

  算法对时间的需求

  同一问题，算法执行时间越短，效率越高

- 算法的**空间复杂性**（space complexity）

  算法对空间的需求（存储空间）

算法的时间复杂性

- 算法的时间复杂性是输入数据量n的函数，记为T(n)，描述算法执行过程中所需的时间用量与问题规模n之间的**函数关系**
- 评价算法的时间复杂性，就是设法找出T(n)和n的函数关系，即计算出T(n)

- **时间单位**

  每执行一条基本语句耗用一个时间单位

  （比较粗些）

- T(n)=执行基本语句的总条（次）数

大体上能够说明算法的时间性能

$$T(n)=O(f(n))$$

f(n)：运行时间增长率的上界

T(n)是f(n)的大O函数

只求T(n)的最高阶
忽略其低阶项和常系数

简化T(n)的计算；较客观地反映当n很大时，算法的时间性能

常用阶（由低到高）

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(n^3)$

$O(n^c)$

（有效算法）

常数阶——最快

对数阶——底数无关

线性阶

很理想的阶

平方阶

c是常数，多项式阶

$O(2^n)$

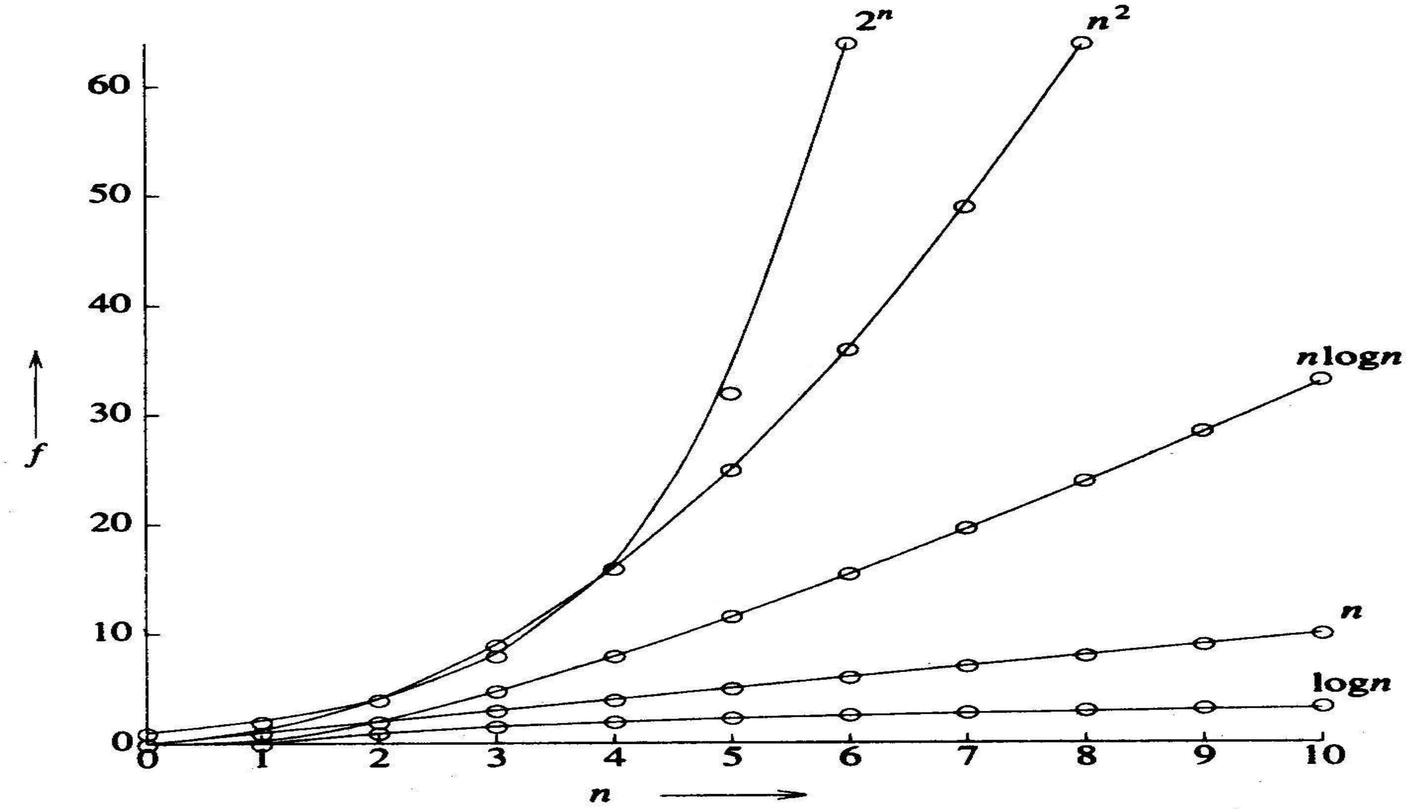
$O(n!)$

(无效算法)

指数阶

阶乘阶

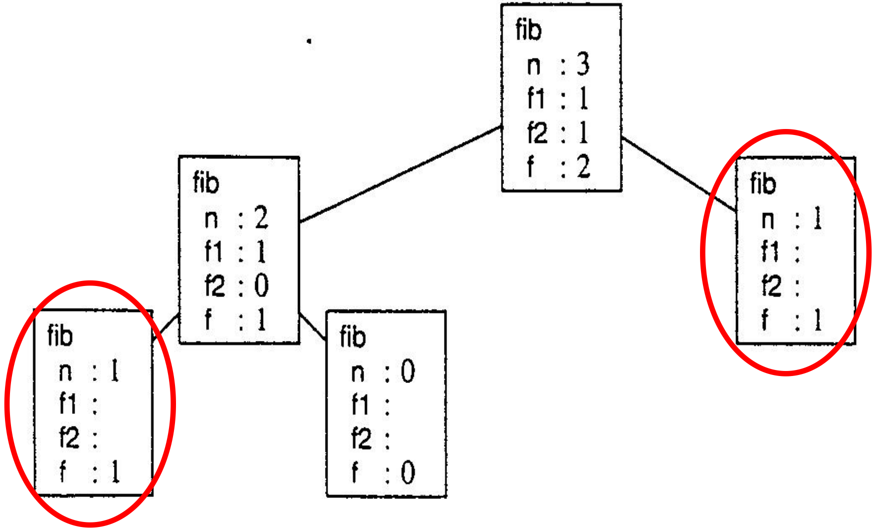
常用阶（由低到高）



示例：求斐波那契函数值的算法时间复杂性

方法一：递归方法

```
   int fib(int n)
   {
1.    if (n < 2)
2.       f = n;  // base cases
3.    else
4.       f = fib( n-1 )+ fib( n-2 );
5.    return f;
   }
```



$$T(n) = O(\Phi^n) \qquad (\Phi = \frac{1+\sqrt{5}}{2})$$

$$\begin{cases} T(n) = 1 & n = 0,1 \\ T(n) = T(n-1) + T(n-2) & n > 1 \end{cases}$$

示例：求斐波那契函数值的算法时间复杂性

方法一：递归方法

```
   int fib(int n)
   {
1.    if (n < 2)
2.        f = n;  // base cases
3.    else
4.        f = fib( n-1 )+ fib( n-2 );
5.    return f;
   }
```

×

$$T(n)=O(\Phi^{n}) \qquad (\Phi=\frac{1+\sqrt{5}}{2})$$

方法二：递推方法

```
   int fib(int n)
   { int  f0=1,f1=1,f2, i=2;
1.   if (n < 2)
2.       return( n);  // base cases
3.   while (i<=n)
4.     {   f2=f1+f0;
5.         f0=f1, f1=f2;
6.         i++;
       }
7.   return f;       }
```

√

$$T(n)=O(n)$$

示例：求斐波那契函数值的算法时间复杂性

方法一：递归方法

```
int fib(int n)
{
    if (n < 2)
        f = n;  // base cases
    else
        f = fib( n-1 )+ fib( n-2 );
    return f;
}
```

方法二：递推方法

```
int fib(int n)
{ int  f0=1,f1=1,f2, i=2;
    if (n < 2)
        return( n);  // base cases
    while (i<=n)
      {   f2=f1+f0;
          f0=f1, f1=f2;
          i++;
      }
    return f;      }
```

设计算法时，应当选用有效算法，并且尽量设法降低它的时间耗用量，以提高算法的运行速度。

算法的最坏情况和平均情况

为什么要区分两种情况 ？

- 有些算法因分枝等因素，对不同的输入数据（即使输入数据量都是n）耗用时间会有所不同，而且往往相差很大
- 为使评价更客观，更有说服力，通常需要分几种情况讨论算法的时间性能
- 在算法理论分析上，最常见的是分别计算出**最坏情况**下和**平均情况**下算法的时间复杂性（也称最坏性态和平均性态）

最坏情况和平均情况

最坏情况（worst-case）

具有相同输入数据量的不同输入数据

算法时间用量的最大值

10
20
30
40
50
60

60
50
40
30
20
10

最坏情况和平均情况

最坏情况（worst-case）

具有相同输入数据量的不同输入数据

算法时间用量的最大值

用$T_W(n)$表示

说一个算法是“好”的，总是希望它在任何情况下运行速度都快———最坏情况下算法的时间性态可以表述算法的这一特征。

最坏情况和平均情况

平均情况

对于所有相同输入数据量的各种不同数据，
算法耗用时间的“平均值”
用$T_E(n)$表示

加概率：期望情况（expected-case）

等概率：平均情况（average -case）

平均情况

对于所有相同输入数据量的各种不同数据，

算法耗用时间的“平均值”

用$T_E(n)$表示

从实用角度看，有些算法遇到最坏情况的可能性极小极小，在大多数情况下是快的。算法的平均性态可以表述算法的这一特征

从最坏情况和平均情况两个角度分析算法的时间性能，可以给算法作出更为客观的评价

空间复杂性

算法空间用量函数S(n)

算法执行时所需空间：占用内存单元数

通常只计算辅助空间用量

不计原始数据所占的空间

为了节省时间，先作“预处理”，多记一些信息（多占空间）——时空转换

时空折中方案（space versus time trade-offs）

示例——循环地右移k位问题

❖ **问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。**

方法一：K遍右移法

（1）算法文字描述

每遍将n个元素循环右移一位，经k遍右移，而完成。

示例——循环地右移k位问题

- 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法一：K遍右移法

（1）算法文字描述

每遍将n个元素循环右移一位，经k遍右移，而完成。

例如（n=6，k=2），

原始排列	10	20	30	40	50	60
第一遍，循环右移一位	60	10	20	30	40	50
第二遍，循环右移一位	50	60	10	20	30	40

示例——循环地右移k位问题

- 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法一：**K遍右移法**

（2）算法形式化描述

```
void method1(int  a[N], int n, int k)  //移位函数
{ int  x,i,j;
  for(j=0;j<k;j++)
  {  x=a[n-1];
        for(i=n-2;i>=0;i--)a[i+1]=a[i];
        a[0]=x;
  }
}
```

示例——循环地右移k位问题

- 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法一：K遍右移法

（2）算法形式化描述

```
   void method1(int  a[N], int n, int k)  //移位函数
   { int  x,i,j;
1.   for(j=0;j<k;j++)
2.   {  x=a[n-1];
3.        for(i=n-2;i>=0;i--)a[i+1]=a[i];
4.        a[0]=x;
     }
   }
```

空间复杂性：$S(n)=1$

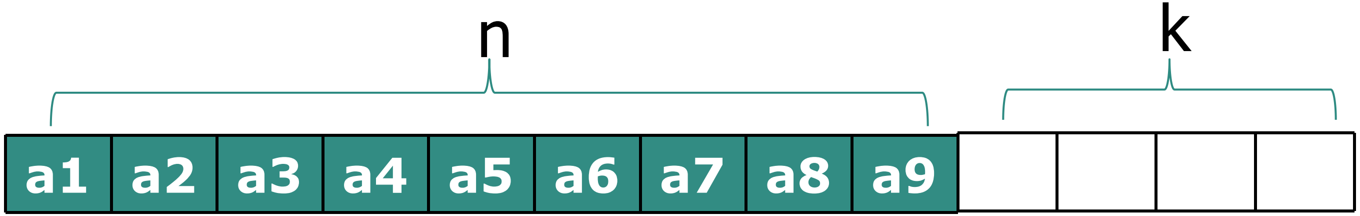
时间复杂性：$T(n)=k(n+1)$

- **问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里$0<k<n$ 。**

方法二：加长数组法

（1）算法文字描述

将数组a[n]的长度加长至a[n+k]后，将a[0]至a[n-1]右移k位，再将移出“数组”的k个元素移至数组左端。



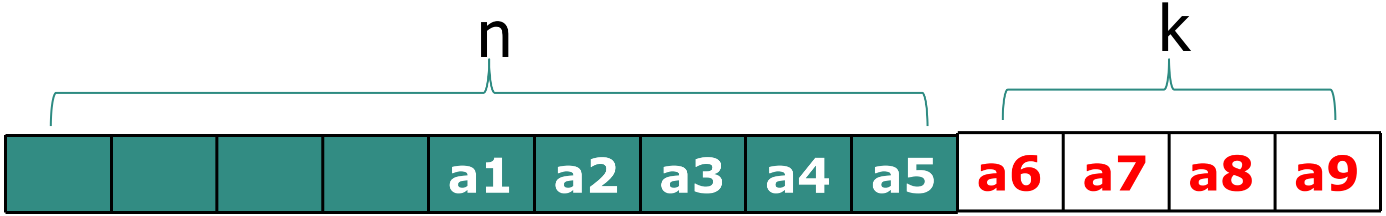
示例——循环地右移k位问题

- **问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。**

方法二：加长数组法

（1）算法文字描述

将数组a[n]的长度加长至a[n+k]后，将a[0]至a[n-1]右移k位，再将移出“数组”的k个元素移至数组左端。

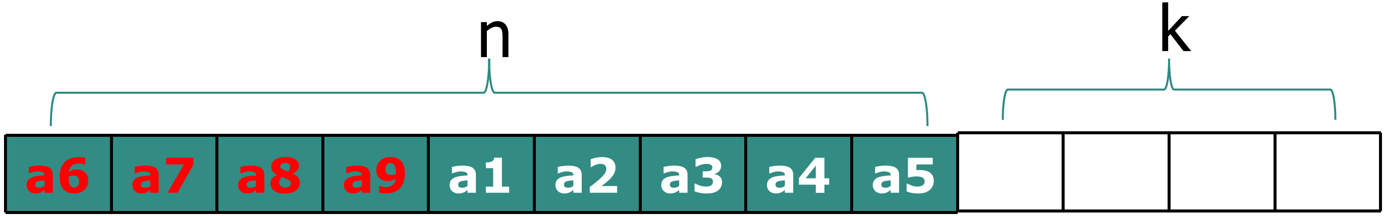


❖ 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法二：加长数组法

（1）算法文字描述

将数组a[n]的长度加长至a[n+k]后，将a[0]至a[n-1]右移k位，再将移出“数组”的k个元素移至数组左端。



示例——循环地右移k位问题

- **问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。**

方法二：加长数组法

（2）算法形式化描述

```
void method3(int a[], int n,int k)  //移位函数
{ int  i;
   for(i=n-1;i>=0;i--)a[i+k]=a[i]; //注意移动方向
   for(i=0;i<k;i++)a[i]=a[i+n];
}
```

示例——循环地右移k位问题

- 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法二：加长数组法

（2）算法形式化描述

```
void method3(int a[], int n,int k)  //移位函数
{ int  i;
   for(i=n-1;i>=0;i--)a[i+k]=a[i]; //注意移动方向
   for(i=0;i<k;i++)a[i]=a[i+n];
}
```

空间复杂性：$S(n)=k$

时间复杂性：$T(n)=n+k$

示例——循环地右移k位问题

- 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法三：**置换圈法**

已经证明将数组a[n]的每个元素都循环地右移k位的结果，共产生m个等长的置换圈（元素只在圈移动），其中，m是n和k的最大公约数。根据这一结论，设计出按圈移动算法。

示例——循环地右移k位问题

方法三：置换圈法

```
int gcd(int m,int n)   //求m和n的最大公因子的函数
{ int r;       while(n) r=m%n, m=n, n=r;    return  m;      }
void  method5(int a[],int n,int k)            //移位函数
{  int i,j,m,p,q,s,x;
1.   m=gcd(n,k);                  //圈数
2.   s=n/m;                       //圈长度
3.   for(i=0;i<m;++i)             //循环处理m个圈
4.    { x=a[i];                   //取出圈头元素
5.      q=i;                      //圈头下标
6.      p=(q-k+n) %n;             //圈尾下标
7.      for(j=1;j<s;j++)          //圈内移位
8.      {  a[q]=a[p];       q=p;       p=(p-k+n) %n;     }
9.      a[q]=x;                   //圈头元素就位
    }}
```

示例——循环地右移k位问题

方法三：置换圈法

```
int gcd(int m,int n)   //求m和n的最大公因子的函数
{ int r;      while(n) r=m%n, m=n, n=
void  method5(int a[],int n,int k)
{  int i,j,m,p,q,s,x;
   m=gcd(n,k);              //圈数
   s=n/m;                   //圈长度
   for(i=0;i<m;++i)         //循环处理m
    { x=a[i];               //取出圈头元
      q=i;                  //圈头下标
      p=(q-k+n) %n;         //圈尾下标
      for(j=1;j<s;j++)      //圈内移位
      {  a[q]=a[p];     q=p;     p=(p-k
      a[q]=x;               //圈头元素就位
    }}
```

- 空间复杂性：S(n)=1。
- 每圈移位时，除圈头元素移动2次外，其余元素均需要移动1次。
- 又由于共有m个置换圈，所以，总的移动次数：T(n)=n+m。
- 又因为，m是n和k的最大公因子，故，m≤k。
- 元素移动总次数：T(n)≤n+k

示例——循环地右移k位问题

- 问题：设计算法将数组a[n]的每个元素都循环地右移k位，这里0<k<n 。

方法	时间复杂性T(n)（移动总次数）	空间复杂性S(n)	设计思路
1、k遍右移法	k(n+1)	1	直观
2、加长数组法	k+n	k	用空间换时间
3、置换圈法	≤k+n	1	好算法

算法的选用原则

- 当数据量不大时，低阶算法未必就快

例如，算法A1和A2的时间复杂性分别为

$$T1(n)=1000n$$

和

$$T2(n)=n^2$$

虽然$O(T1(n))<O(T2(n))$

当$n>1000$时，A1好于A2

当$n\leq1000$时，A2好于A1，因为$T2(n)\leq T1(n)$

算法的选用原则

- 总原则：能满足客观要求即可用
- 需要考虑如下因素：

（1）算法实现的难易程度

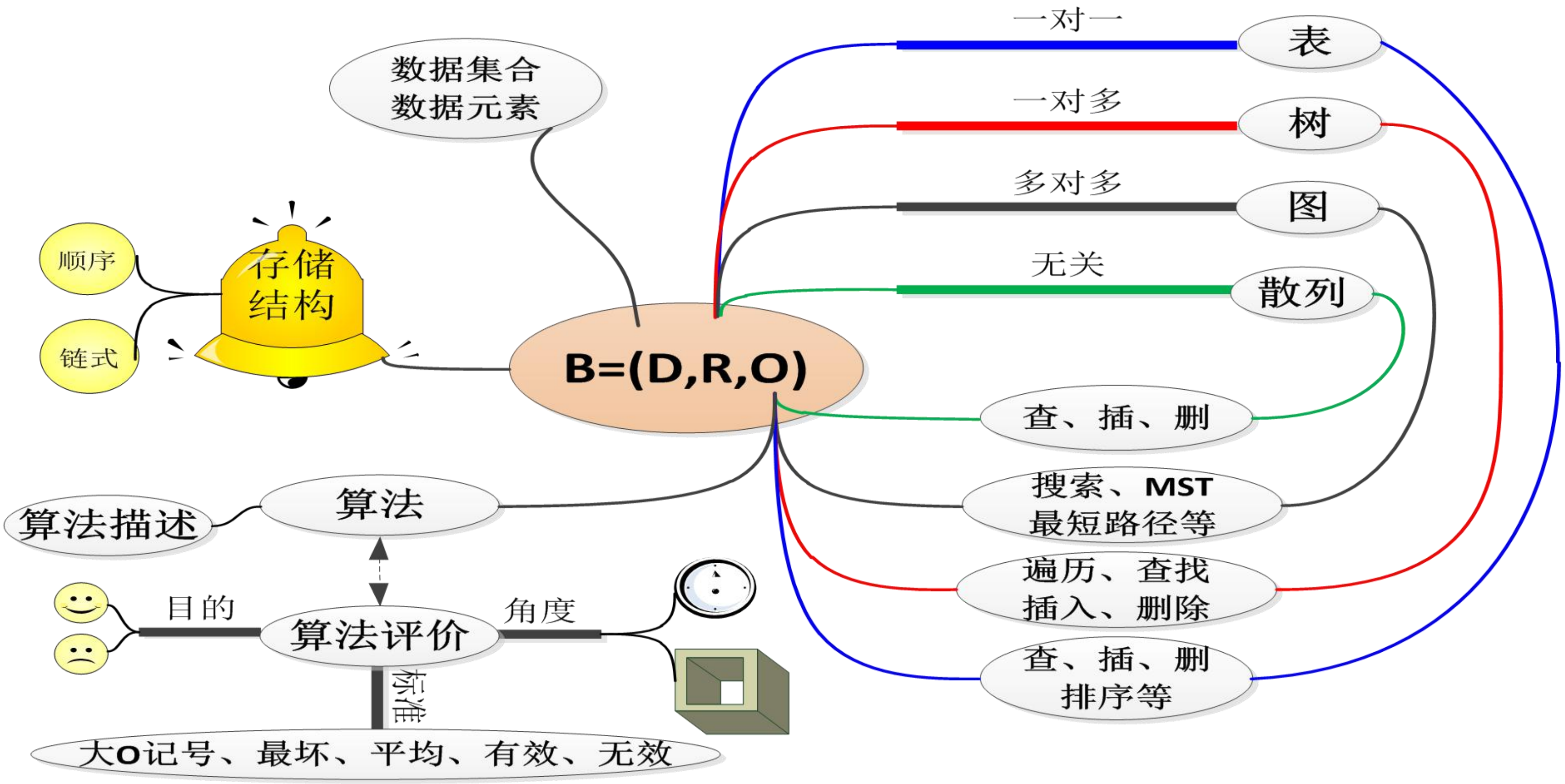
（2）算法使用的次数，否实时处理

（如通信、天气预报等）

（3）算法运行环境

输入数据量的大小范围

小结



数据集合
数据元素
顺序
链式
存储结构
B=(D,R,O)
一对一
表
一对多
树
多对多
图
无关
散列
查、插、删
搜索、MST
最短路径等
遍历、查找
插入、删除
查、插、删
排序等
算法描述
算法
目的
算法评价
角度
标准
大O记号、最坏、平均、有效、无效