
# 数据结构与算法设计

**主讲老师：徐跃东**

2023-2024学年第一学期

# 任课教师：徐跃东

- 2013年9月加入复旦大学
- 研究方向
  - AI for Networking：　　网络资源分配
  - Network for AI：　　分布式机器学习

- 联系方式
  - 邮件：ydxu@fudan.edu.cn
  - 实验室主页：http://medianet.azurewebsites.net/
    http://ee.fudan.edu.cn/Data/View/812

# 助教信息

- 助教信息
  - ***

    Email:
  - ***

    Email:
  - ***

    Email:
- Disclaimer：讲课内容主要来自于原参考教材课件、国内外相关课程的课件、网络论坛等

# 授课知识点


线性表
数组
栈
队列
串
广义表
二叉树
树
图
查找
内排序
外排序
分治算法
贪心算法
动态规划
散列表
NP完全问题
复杂度分析

# 参考教材

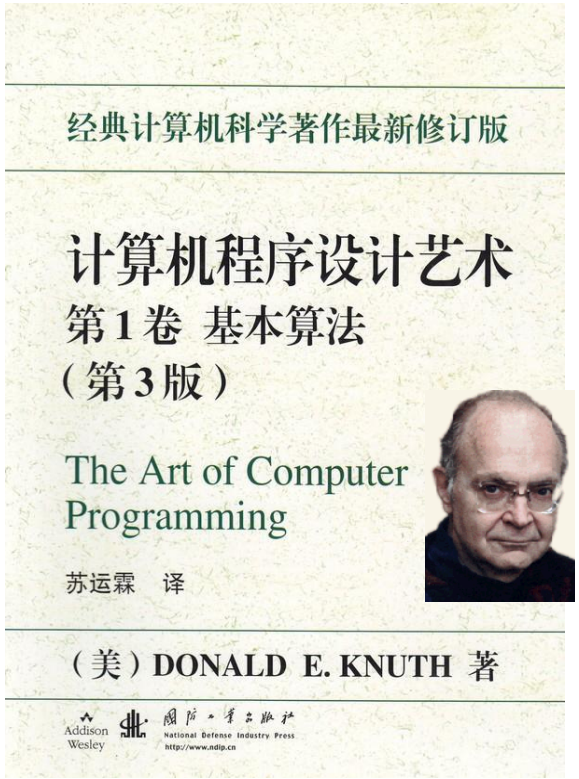
- 数据结构参考教材
  - 《数据结构》(C语言版)　清华大学出版社　严蔚敏等 编著
  - 《数据结构教程》　清华大学出版社　李春葆等 编著
  - 《数据结构》 (用面向对象方法与C++语言描述)
    清华大学出版社　殷人昆等 编著

- 算法设计参考教材
  - 《算法导论》(中英文)　机械工业出版社　T.H. Cormen 等著
  - 《数据结构与算法设计》(C语言描述)
    机械工业出版社　M.A. Weiss 等著
  - 《计算机程序设计艺术》 人民邮电出版社　Donald E. Knuth 著

# 作业与综合成绩

- 作业提交
  - 源码：学号注册、OJ系统提交
  - 分析：“实验报告”文件夹，pdf/doc/docx格式
- 作业量
  - 每周编程题 2 题
    (1题课本例题、1题课外例题)
- 课程项目
  - 约4个工作量超平时作业的题目
- 作业+PJ总代码量
  - 约 2000 行~2500 行
- 组成部分
- (视情况微调)
  - 平时作业：15% (20%)
  - 期中考试：15% (10%)
  - 课程项目：10%
  - 期末考试：60%

# 数据结构历史沿革




- 1968年美国人Donald E. Knuth开创了数据结构的最初体系。

- 他所著的《计算机程序设计艺术》（共7卷）第一卷《基本算法》是第一本较系统地阐述数据的逻辑结构和存储结构及其操作的著作。

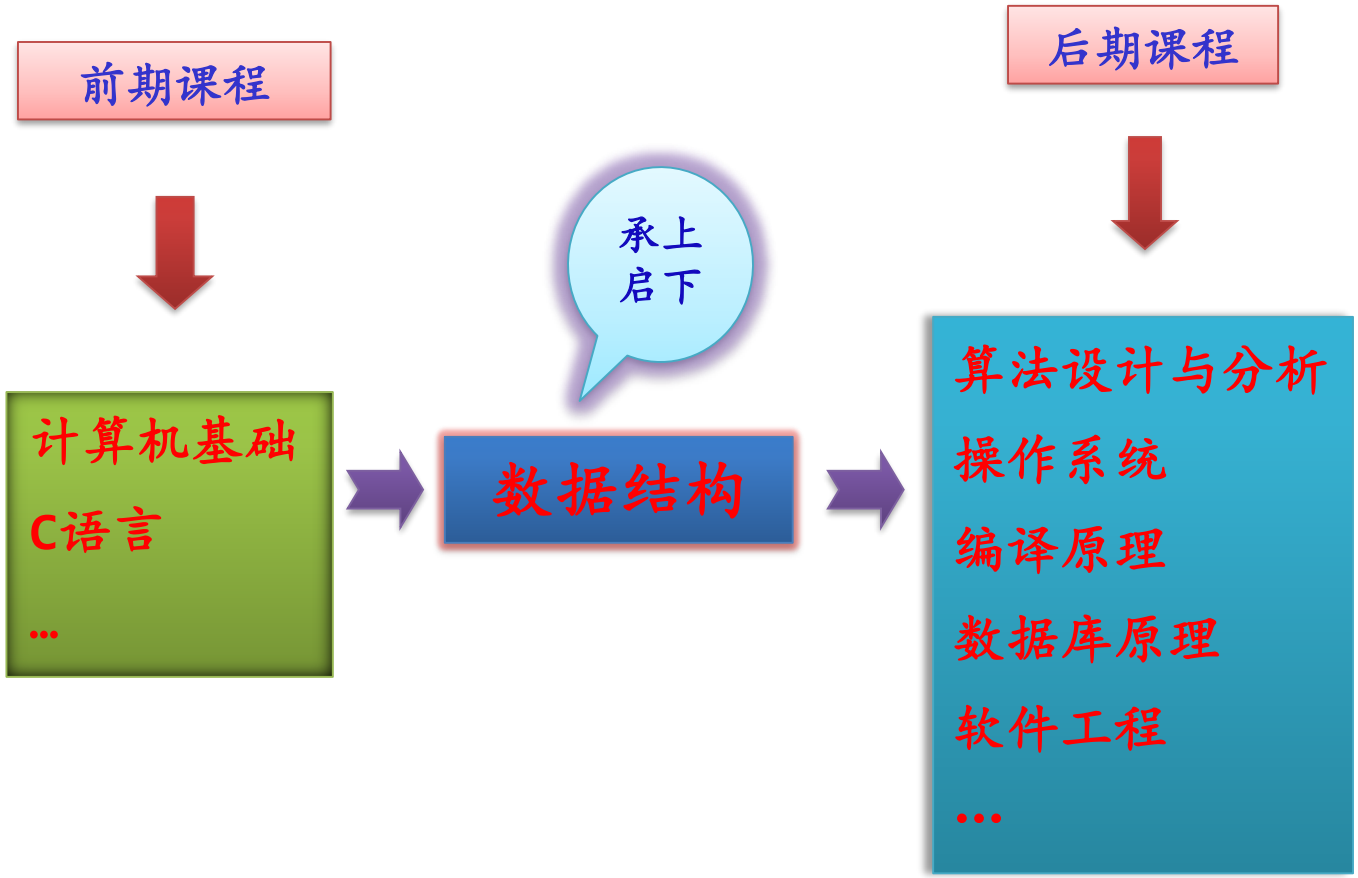
数据结构问题起源于程序设计!

# 1 “数据结构”课程的内容

- 各种数据的逻辑结构描述。
- 各种数据的存储结构表示。
- 各种数据结构的运算定义。
- 设计实现运算的算法。
- 分析算法的效率。

基本数据组织和数据处理方法

# 2 “数据结构”在计算机课程体系中的地位



# Why study algorithms and data structures?

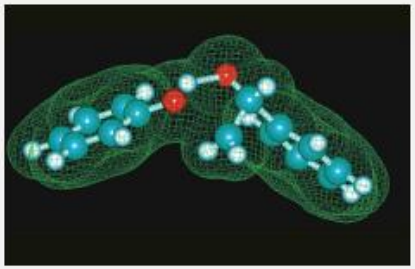
- **Their impact is broad and far-reaching**

# Why study algorithms and data structures?

- **They may unlock the secrets of life and of the universe.**

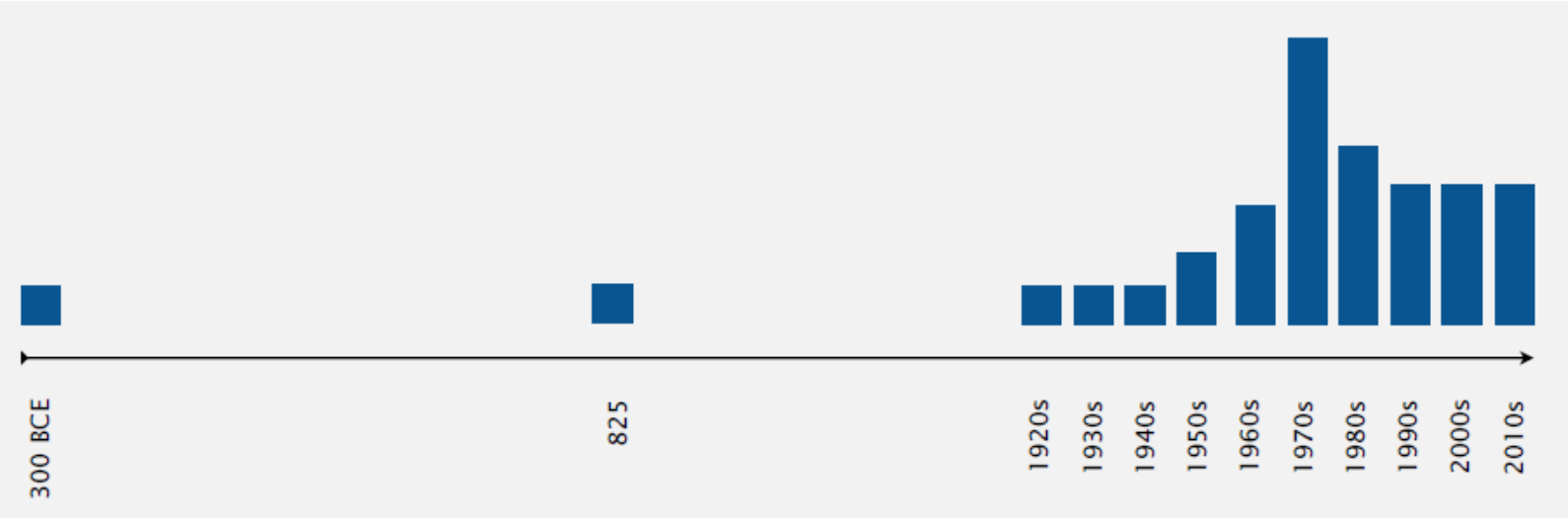
> *" Computer models mirroring real life have become crucial for most advances made in chemistry today…. Today the computer is just as important a tool for chemists as the test tube."*
>
> — *Royal Swedish Academy of Sciences (Nobel Prize in Chemistry 2013)*

Martin Karplus, Michael Levitt, and Arieh Warshel



# Why study algorithms and data structures?

- **Old roots, new opportunities**
  - Study of algorithms dates at least to Euclid
  - Named after Muhammad ibn Mūsā al-Khwārizmī
  - Formalized by Church and Turing in 1930s
  - Some important algorithms were discovered by undergrads



# Why study algorithms and data structures?

- **To become a proficient programmer**

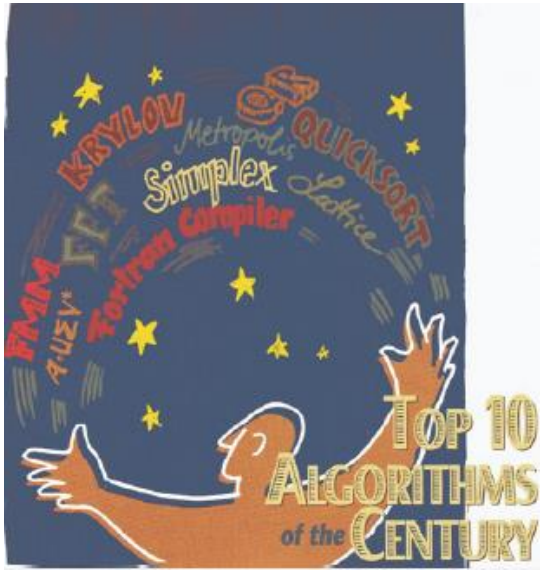
*" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "*

*— Linus Torvalds (architect of Linux and git)*

# Why study algorithms and data structures?

- **For intellectual stimulation**

*" For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. " — Francis Sullivan*



# Why study algorithms and data structures?

- **For fun and profit**

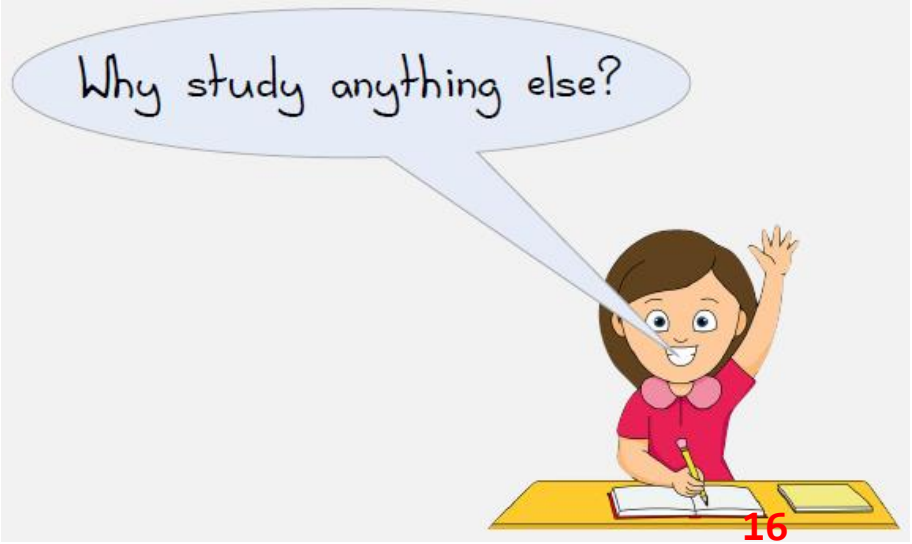
Google



ByteDance
字节跳动

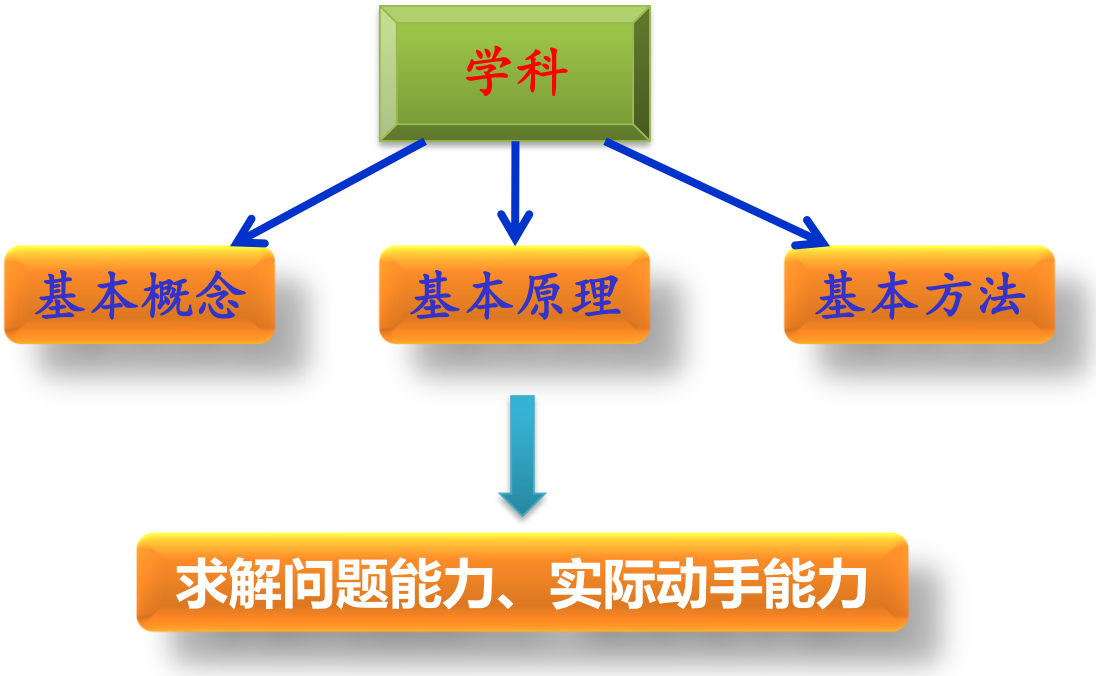
# Why study algorithms and data structures?

- Their impact is broad and far-reaching
- They may unlock the secrets of life and of the universe
- Old roots, new opportunities
- To become a proficient programmer
- For intellectual stimulation
- For fun and profit



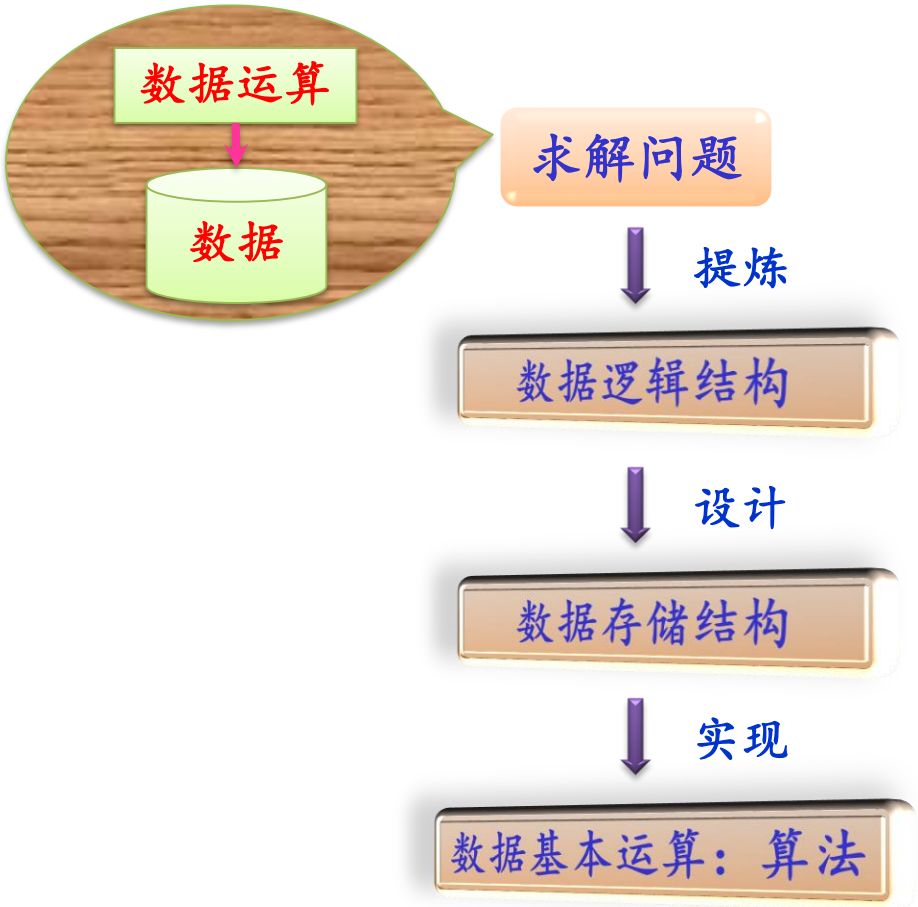
# 4 “数据结构”的学习目标

❶ 掌握数据结构的基本概念、基本原理和基本方法。




求解问题

❷ 掌握数据的逻辑结构、存储结构及基本运算的实现过程。



❸ 掌握算法基本的时间复杂度与空间复杂度的分析方法，能够设计出求解问题的高效算法。

设计优秀的算法，具有较小的时间复杂度与空间复杂度。

例如，求1 + 2 + … n。

算法1：

```
int fun1(int n)
{  int i, s=0;
   for (i=1;i<=n;i++)
      s+=i;
   return s;
}
```

算法2：

```
int fun2(int n)
{
   return (n+1)*n/2;
}
```

算法分析

**算法2好于算法1**

# 第1章 绪 论

1.1 什么是数据结构

1.2 算法及其描述

1.3 算法分析基础

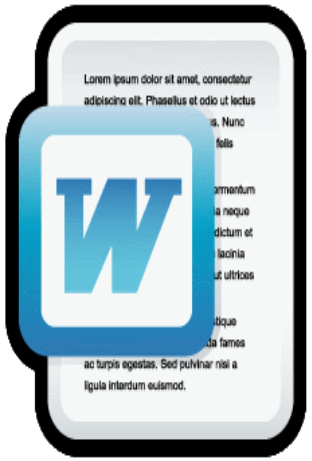
1.4 其他情况的算法分析

# 1.1 什么是数据结构

## 1.1.1 数据结构的定义

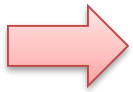
### 数据结构中的几个概念

- **数据**：所有能够输入到计算机中，且能被计算机处理的符号的集合。




Word文档


图像文档



都是数据

而数据结构中主要讨论结构化数据。

# 结构化数据示例

一个学生表

| 学号 | 姓名 | 性别 | 班号 |
| --- | --- | --- | --- |
| 1 | 张斌 | 男 | 9901 |
| 8 | 刘丽 | 女 | 9902 |
| 34 | 李英 | 女 | 9901 |
| 20 | 陈华 | 男 | 9902 |
| 12 | 王奇 | 男 | 9901 |
| 26 | 董强 | 男 | 9902 |
| 5 | 王萍 | 女 | 9901 |

数据项(用于描述数据元素)

数据元素

- **数据元素：**是数据（集合）中的一个“个体”，它是数据的基本单位。
- **数据项：**数据项是用来描述数据元素的，它是数据的最小单位。
- **数据对象：**具有**相同性质**的若干个数据元素的集合，如整数数据对象是所有整数的集合。

默认情况下，数据结构中讨论的数据都是**数据对象**。

**数据结构**：是指带结构的数据元素的集合。

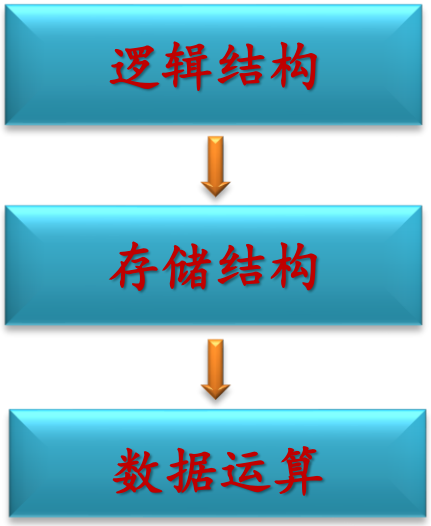
数据结构中讨论的元素关系主要是指**相邻关系**或**邻接关系**。

| 学号 | 姓名 | 性别 | 班号 |
|---|---|---|---|
| 1 | 张斌 | 男 | 9901 |
| 8 | 刘丽 | 女 | 9902 |
| 34 | 李英 | 女 | 9901 |
| 20 | 陈华 | 男 | 9902 |
| 12 | 王奇 | 男 | 9901 |
| 26 | 董强 | 男 | 9902 |
| 5 | 王萍 | 女 | 9901 |

相邻

不相邻

# 一个数据结构的构成：




- 数据元素之间的逻辑关系 ⇨ 数据的逻辑结构。
- 数据元素及其关系在计算机存储器中的存储方式 ⇨ 数据的存储结构（或物理结构）。
- 施加在该数据上的操作 ⇨ 数据运算。

# 1、数据的逻辑结构表示

数据的逻辑结构是面向用户的，它有多种表示形式。

## ❶ 学生表的逻辑结构表示1-表格

| 学号 | 姓名 | 性别 | 班号 |
|---|---|---|---|
| 1 | 张斌 | 男 | 9901 |
| 8 | 刘丽 | 女 | 9902 |
| 34 | 李英 | 女 | 9901 |
| 20 | 陈华 | 男 | 9902 |
| 12 | 王奇 | 男 | 9901 |
| 26 | 董强 | 男 | 9902 |
| 5 | 王萍 | 女 | 9901 |

直接来源于现实世界

## ❷ 学生表的逻辑结构表示2-二元组

**二元组**是一种通用的逻辑结构表示方法

一个二元组表示为：

$B=(D, R)$

其中，B是一种数据结构，它由数据元素的集合*D*和*D*上二元关系的集合*R*所组成。其中：

$D=\{ d_i \mid 1\le i\le n, n\ge 0\}$：数据元素的集合

$R=\{ r_j \mid 1\le j\le m, m\ge 0\}$：关系的集合

每个关系的用若干个序偶来表示：

- 序偶<x，y>（x，y∈D）⇒ x为第一元素，y为第二元素。
- x为y的前驱元素。
- y为x的后继元素。
- 若某个元素没有前驱元素，则称该元素为开始元素；若某个元素没有后继元素，则称该元素为终端元素。

序偶<x，y>表示x、y是有向的，序偶(x，y)表示x、y是无向的

| 学号 | 姓名 | 性别 | 班号 |
| --- | --- | --- | --- |
| 1 | 张斌 | 男 | 9901 |
| 8 | 刘丽 | 女 | 9902 |
| 34 | 李英 | 女 | 9901 |
| 20 | 陈华 | 男 | 9902 |
| 12 | 王奇 | 男 | 9901 |
| 26 | 董强 | 男 | 9902 |
| 5 | 王萍 | 女 | 9901 |

每个学生记录用学号标识

二元组逻辑表示：

<1,8>, <8,34>, <34,20>, <20,12>, <12,26>, <26,5>

例如，如下数据为一个矩阵：

$$\begin{bmatrix} 2 & 6 & 3 & 1 \\ 8 & 12 & 7 & 4 \\ 5 & 10 & 9 & 11 \end{bmatrix}$$

对应的二元组表示为$B=(D, R)$，其中：

$D=\{2, 6, 3, 1, 8, 12, 7, 4, 5, 10, 9, 11\}$

$R=\{r1, r2\}$　　其中，r1表示行关系，r2表示列关系

$r1=\{<2,6>, <6,3>, <3,1>, <8,12>, <12,7>, <7,4>,$
$<5,10>, <10,9>, <9,11>\}$

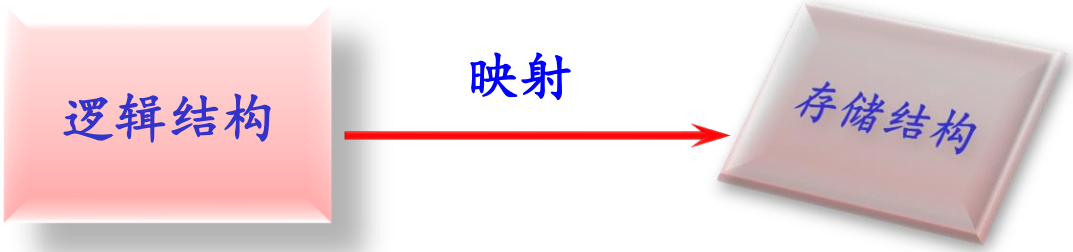
$r2=\{<2,8>, <8,5>, <6,12>, <12,10>, <3,7>, <7,9>,$
$<1,4>, <4,11>\}$

## ❸ 学生表的逻辑结构表示3-图形

在学生表中，用学号标识每个学生记录，其逻辑结构用图形表示如下：

## 2、数据的存储结构表示

数据在计算机存储器中的存储方式就是存储结构。




设计存储结构的这种映射应满足两个要求：

- 存储所有元素
- 存储数据元素间的关系

## ❶ 学生表存储结构1— 结构体数组

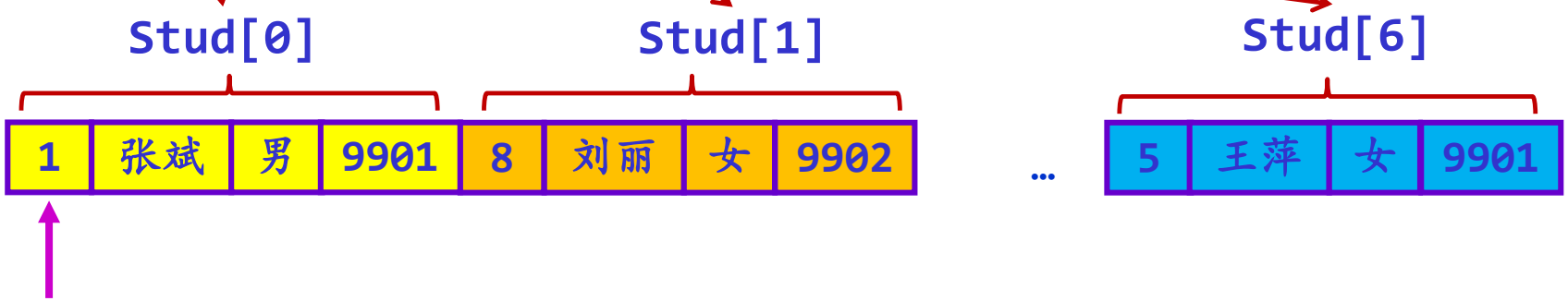
存放学生表的结构体数组Stud定义如下：

```
struct
{   int no;                 //存储学号
    char name[8];           //存储姓名
    char sex[2];            //存储性别
    char class[4];          //存储班号
}   Stud[7]={  {1,“张斌” ,“男” ,“9901”},
                            …,
                {5,"王萍","女","9901"}  } ;
```

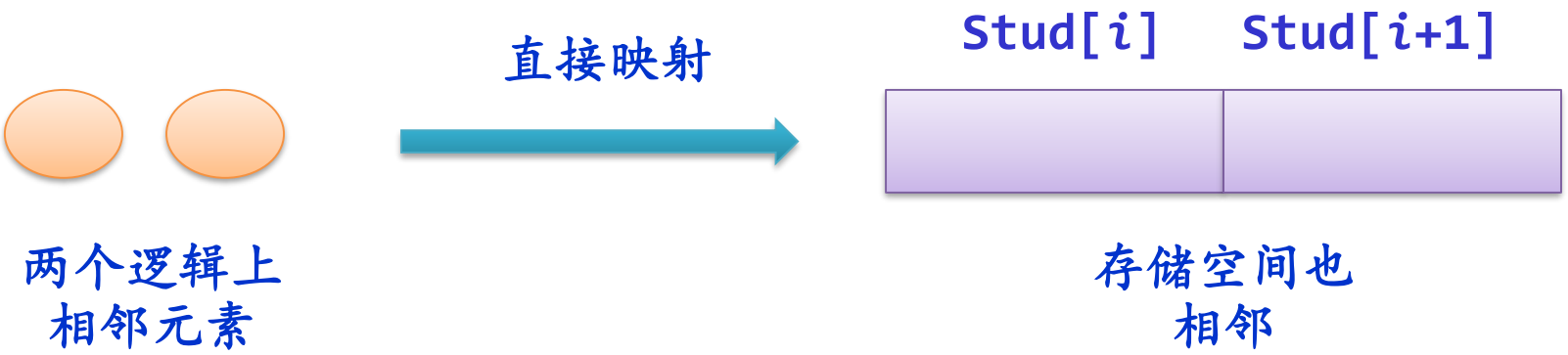
## 映射过程：

学生表的逻辑结构

| 学号 | 姓名 | 性别 | 班号 |
|---|---|---|---|
| 1 | 张斌 | 男 | 9901 |
| 8 | 刘丽 | 女 | 9902 |
| 34 | 李英 | 女 | 9901 |
| 20 | 陈华 | 男 | 9902 |
| 12 | 王奇 | 男 | 9901 |
| 26 | 董强 | 男 | 9902 |
| 5 | 王萍 | 女 | 9901 |




存储结构建立完毕




这种存储结构的特点：

- 所有元素占用一整块内存空间。
- 逻辑上相邻的元素，物理上也相邻。

顺序存储结构

## ❷ 学生表存储结构2— 链表

存放学生表的链表的结点类型StudType声明如下：

```
typedef struct studnode
{   int no;                          //存储学号
    char name[8];                    //存储姓名
    char sex[2];                     //存储性别
    char class[4];                   //存储班号
    struct studnode *next;           //存储指向下一个学生的指针
}   StudType;
```

## 映射过程：

学生表的逻辑结构

| 学号 | 姓名 | 性别 | 班号 |
|---|---|---|---|
| 1 | 张斌 | 男 | 9901 |
| 8 | 刘丽 | 女 | 9902 |
| 34 | 李英 | 女 | 9901 |
| 20 | 陈华 | 男 | 9902 |
| 12 | 王奇 | 男 | 9901 |
| 26 | 董强 | 男 | 9902 |
| 5 | 王萍 | 女 | 9901 |


存储结构建立完毕

这种存储结构的特点：

- 一个逻辑元素用一个结点存储，每个结点单独分配，所有结点的地址不一定是连续的。
- 用指针来表示逻辑关系。

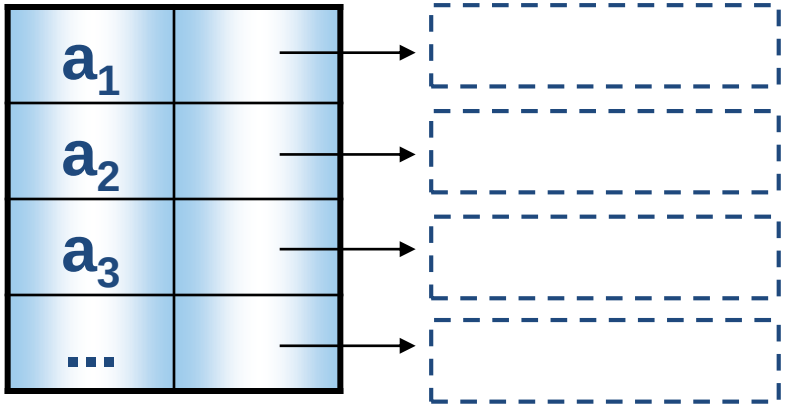
链式存储结构

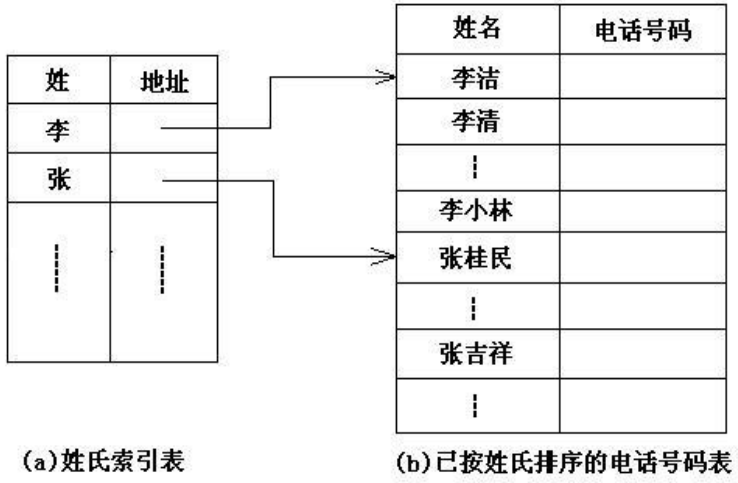
# 其他存储方式

## 散列存储

根据关键字key → H(key)，来确定存储地址

## 索引存储




(a)姓氏索引表　(b)已按姓氏排序的电话号码表

# 3、数据运算

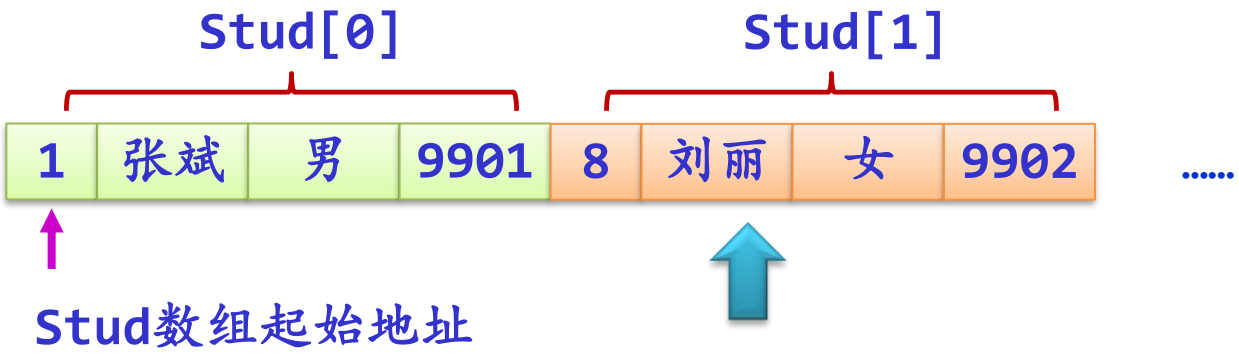
数据运算是对数据的操作。分为两个层次：运算描述和运算实现。

对于“学生表”这种数据结构，可以进行一系列的运算：

- 查找序号为2的学生姓名
- 增加一个学生记录；
- 删除一个学生记录；
- 查找性别为“女”的学生记录；
- 查找班号为“9902”的学生记录；
- ……

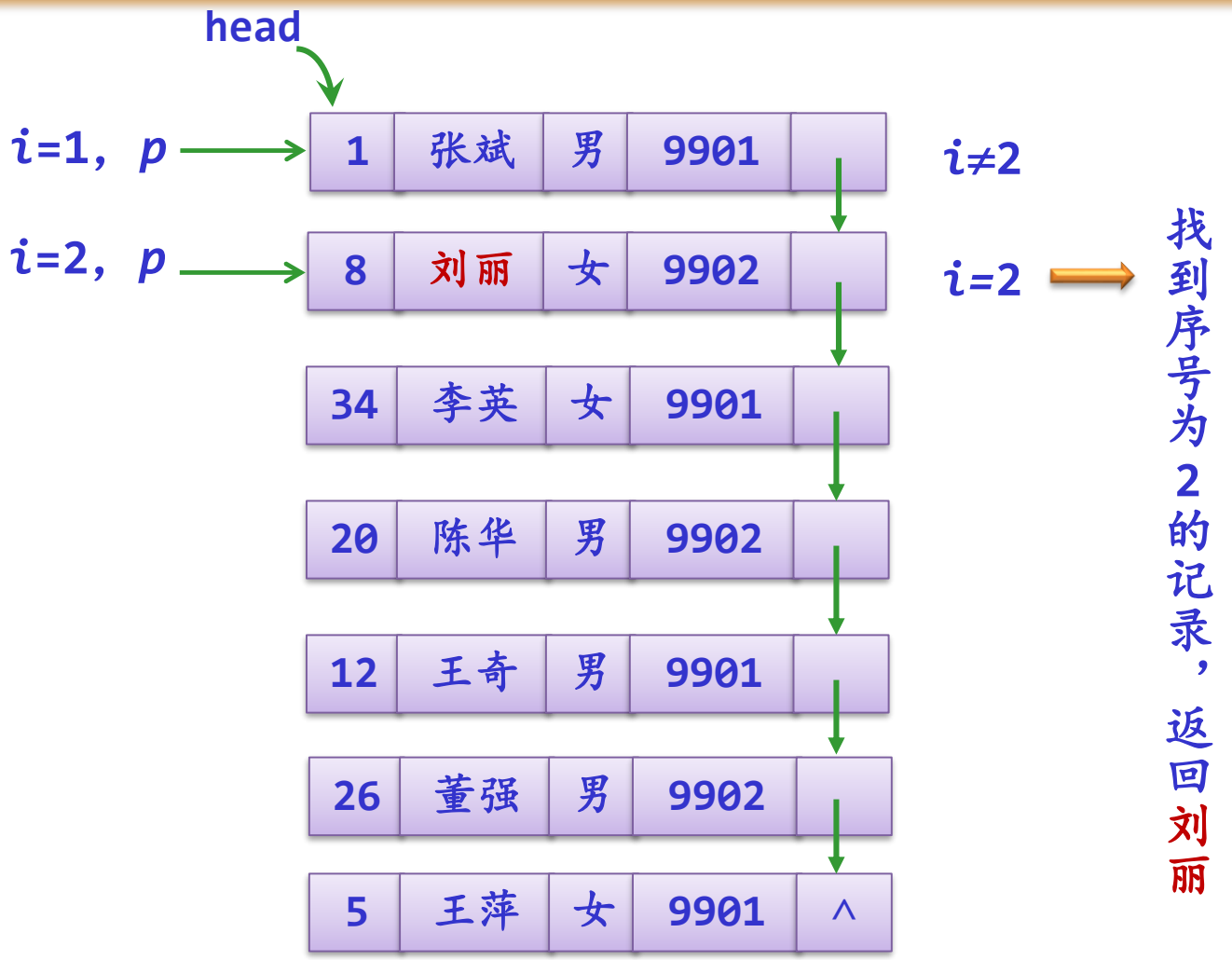
运算描述

## ❶ 顺序存储结构中实现“查找序号为2的学生姓名”




直接找到Stud[1] 记录，返回刘丽

## ❷ 链式存储结构中实现“查找序号为2的学生姓名”




- 同一逻辑结构可以对应多种存储结构。同样的运算，在不同的存储结构中，其实现过程是不同的。

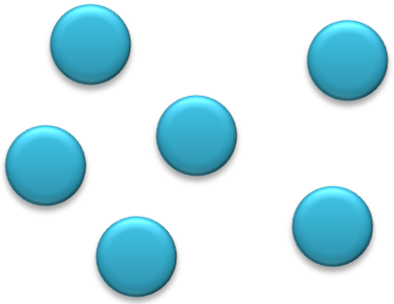
# 1.1.2 逻辑结构类型

各种各样的数据呈现出不同的逻辑结构，归纳为4种。

## 1、集合

元素之间关系：无。

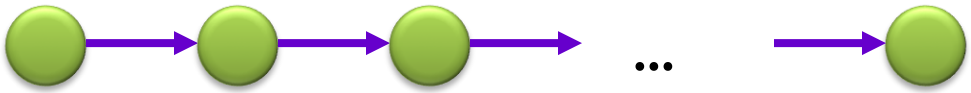
特点：数据元素之间除了“属于同一个集合”的关系外，别无其他逻辑关系。是最松散的，不受任何制约的关系。



## 2、线性结构

元素之间关系：一对一。

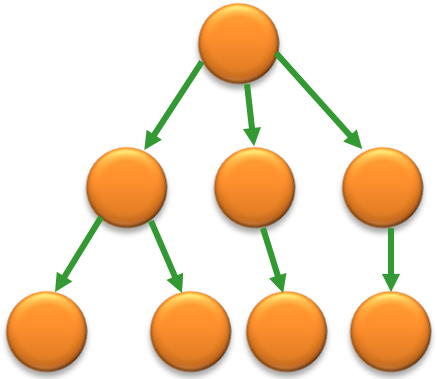
特点：开始元素和终端元素都是唯一的，除此之外，其余元素都有且仅有一个前驱元素和一个后继元素。



## 3、树形结构

元素之间关系：一对多。

特点：开始元素唯一，终端元素不唯一。除终端元素以外，每个元素有一个或多个后续元素；除开始元素外，每个元素有且仅有一个前驱元素。



【例1-3：p5】有一种数据结构B2=（*D*，*R*），其中

*D*={48，25，64，57，82，36，75}

*R*={$r_1$，$r_2$}

$r_1$={<25，36>，<36，48>，<48，57>，<57，64>，
<64，75>，<75，82>}

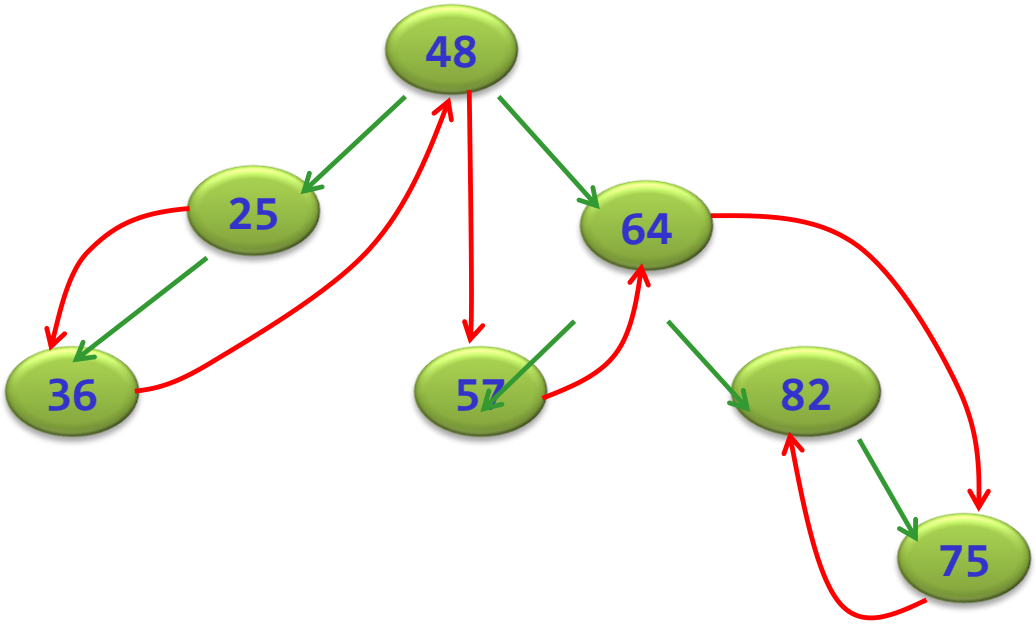
$r_2$={<48，25>，<48，64>，<64，57>，<64，82>，
<25，36>，<82，75>}

画出其逻辑结构表示，指出是什么类型？

解：B2的逻辑结构图如下。

$r_1$关系表示 → $r_1$为线性结构

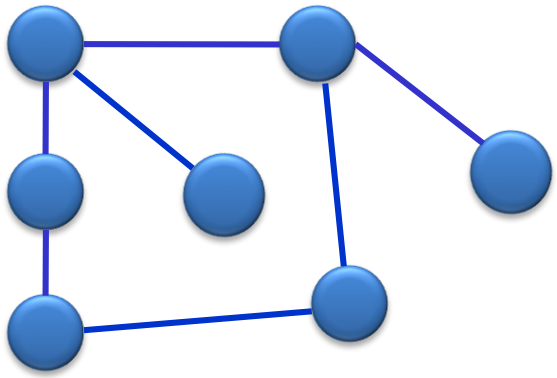
$r_2$关系表示 → $r_2$为树形结构



# 4、图形结构

元素之间关系：多对多。

特点：所有元素都可能有多个前驱元素和多个后继元素。



# 1.1.3 存储结构类型

在软件开发中，人们设计了各种存储结构。归纳为4种基本的存储结构。

- 顺序存储结构
- 链式存储结构
- 索引存储结构
- 哈希（散列）存储结构

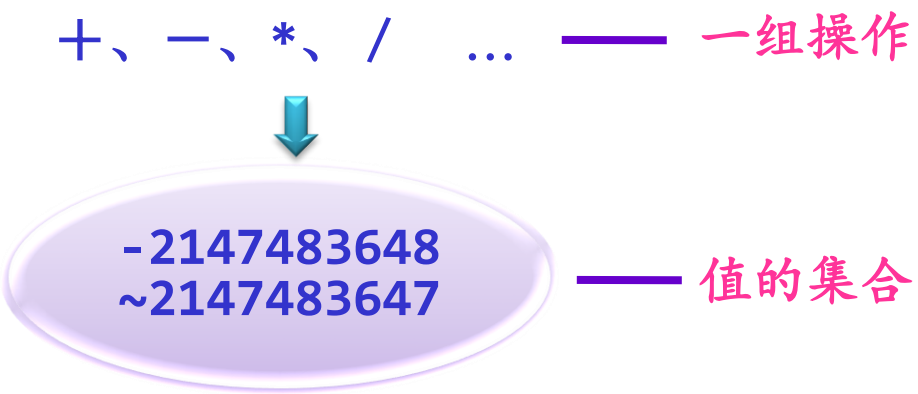
# 1.1.4 数据类型和抽象数据类型

## 1、数据类型

在高级程序语言中提供了多种数据类型。不同数据类型的变量，其所能取的值的范围不同，所能进行的操作不同。

**数据类型**是一个值的集合和定义在此集合上的一组操作的总称。

例如，C/C++中的int就是整型数据类型




数据元素在内存的空间占用（64位机、C语言）

- 整数类型（4字节）
- double类型（8字节）
- 布尔类型（1字节）
- 复合型

浮点类型（4字节）
字符类型（1字节）
指针类型（8字节）

```c
#include <stdio.h>

int main() {
    printf("sizeof( char): %zd\t", sizeof(char));
    printf("sizeof( float): %zd\n", sizeof(float));
    printf("sizeof( short int): %zd\t", sizeof(short int));
    printf("sizeof( int): %zd\n", sizeof(int));
    printf("sizeof( long int): %zd\t", sizeof(long int));
    printf("sizeof( long long int): %zd\n", sizeof(long long int));
    printf("sizeof( size_t): %zd\t", sizeof(size_t));
    printf("sizeof( void*): %zd\n\n", sizeof(void *));


    //printf("PRIu32 usage (see source): %"PRIu32"\n", (uint32_t)42);
    return 0;
}
```

## 2、抽象数据类型

抽象数据类型（ADT）指的是从求解问题的数学模型中抽象出来的数据逻辑结构和运算（抽象运算），而不考虑计算机的具体实现。

抽象数据类型 = 逻辑结构 + 抽象运算

例如，定义复数抽象数据类型Complex

一个复数的形式：$e_1+e_2i$或（$e_1$，$e_2$）

ADT Complex

{

数据对象：

$D=\{\ e_1, e_2\ \mid e_1, e_2$均为实数 $\}$

数据关系：

$R=\{<e_1, e_2>\ \mid\ e_1$是复数的实部，$e_2$ 是复数的虚部 $\}$

基本运算：

AssignComplex(&z, v1, v2)：构造复数Z。

DestroyComplex(&z)：复数z被销毁。

GetReal(z, &real)：返回复数z的实部值。

GetImag(z, &Imag)：返回复数z的虚部值。

Add(z1, z2, &sum)：返回两个复数z1、z2的和。

} ADT Complex

运算功能描述

Complex

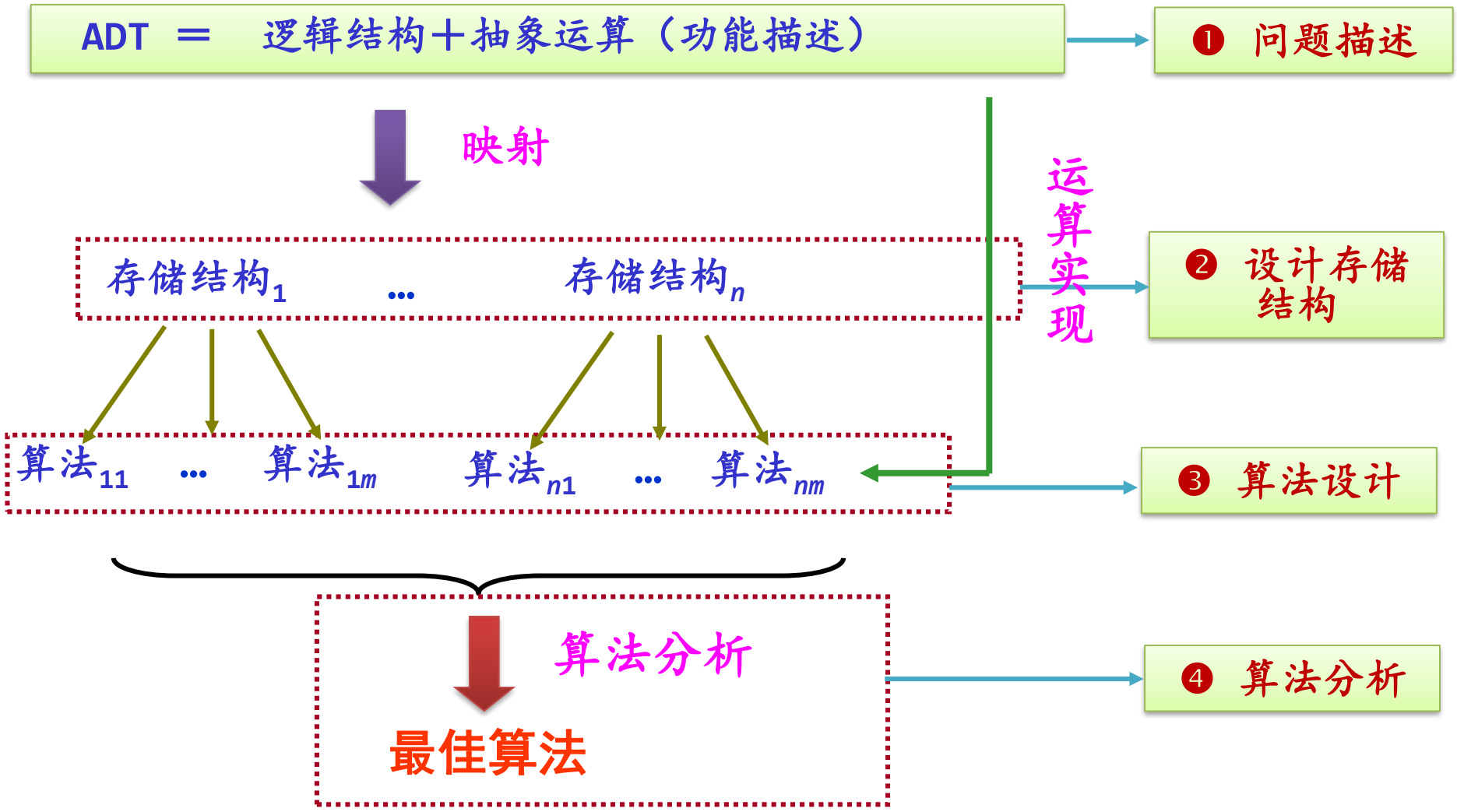
ADT


编程实现该数据结构

抽象数据类型实质上就是对一个求解问题的形式化描述（与计算机无关），程序员可以在理解基础上实现它。

**思考：** 采用C/C++语言如何实现复数抽象数据类型Complex?

# 1.1.5 数据结构求解问题的过程

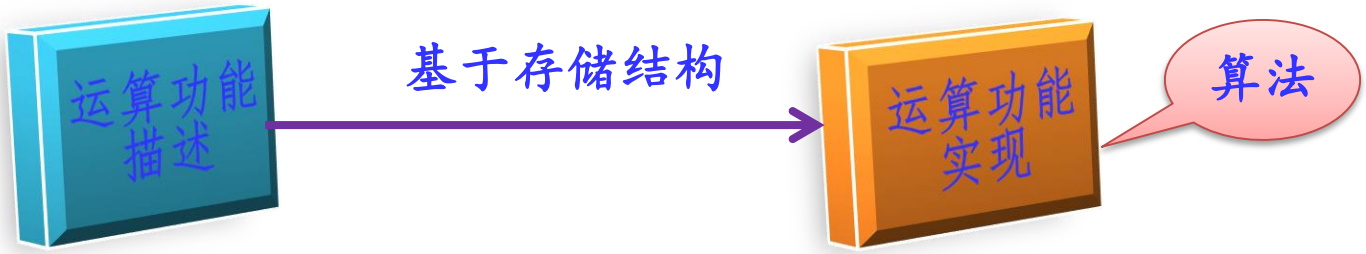


# 1.2 算法及其描述

## 1.2.1 什么是算法

数据元素之间的关系有逻辑关系和物理关系，对应的运算有基于逻辑结构的运算描述和基于存储结构的运算实现。

通常把基于存储结构的运算实现的步骤或过程称为算法。



## 算法的五个重要的特性

（1） 有穷性：在有穷步之后结束，算法能够停机。

（2） 确定性：无二义性。

（3） 可行性：可通过基本运算有限次执行来实现，
即算法中每一个动作能够被机械地执行。

（4） 有输入

（5） 有输出

} 表示存在数据处理

【例（补充）】考虑下列两段描述，这两段描述均不能满足算法的特性，试问它们违反了哪些特性？

## （1）描述一

```
void exam1()
{  int  n=2;
   while (n%2==0)
       n=n+2;
   printf("%d\n", n);
}
```

其中有一个死循环，违反了算法的有穷性特性。

## (2) 描述二

```
void exam2()
{   int x, y;
    y=0;
    x=5/y;
    printf("%d, %d\n", x, y);
}
```

其中包含除零错误，违反了算法的可行性特性

# 1.2.2 算法描述


算法描述的一般格式

```
返回值  算法对应的函数名(形参列表)
{   //临时变量的定义
//实现由输入参数到输出参数的操作
…
}
```

函数体

- 返回值：通常为bool类型，表示算法是否成功执行。
  - 形参列表：由输入型参数和输出型参数构成。

算法输入　　算法输出

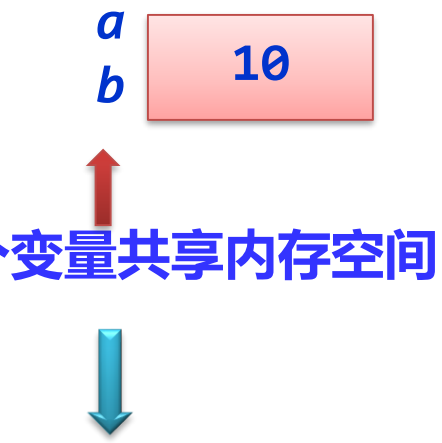
# 如何描述输出型参数?

C++语言中提供了一种引用运算符“&”用于描述输出型参数。

## 引用示例

```
int a=10;
int &b=a;
```

引用



**示例：**设计一个交换两个整数的算法。

编写一个函数swap1(*x*, *y*)：

```
void swap1(int x, int y)
{   int tmp;
    tmp=x; x=y; y=tmp;
}
```

交换形参*x*和*y*的值

当执行语句swap1(*a*, *b*)时，*a*和*b*实参值不会发生了交换。

分析：*x*、*y*既是输入型参数，也是输出型参数

**改正方法1**：采用指针的方式来回传形参的值，需将上述函数改为：

```
void swap2(int *x, int *y)
{   int tmp;
    tmp=*x;       //将x的值放在tmp中
    *x=*y;        //将x所指的值改为*y
    *y=tmp;       //将y所指的值改为tmp
}
```

交换形参x和y所指向的值

上述函数的调用改为swap2(&a，&b)

**改正方法2：** 采用引用型形参 ⇨ 将输出型形参改为引用类型。

```
void  swap(int &x, int &y)
//形参前的“&”符号不是指针运算符
{  int tmp=x;
   x=y; y=tmp;
}
```

交换形参x和y的值

当执行语句swap(a, b)时，形、实参的匹配相当于：

```
int  &x=a;      //a为x的引用
int  &y=b;      //b为y的引用
```

这样，a与x共享存储空间、b与y共享存储空间，因此执行函数后a和b的值发生了交换 ⇨ **简单明了**。

# 普通的参数传递

```
void fun1(int n)
{   int m=2;
    fun2( m );
    printf("%d\n", m);
}
```

实参

```
void fun2(int x)
{   x++;
    printf("%d\n", x);
}
```

普通形参

实参到形参单向值传递

fun1(m) → fun2(x)

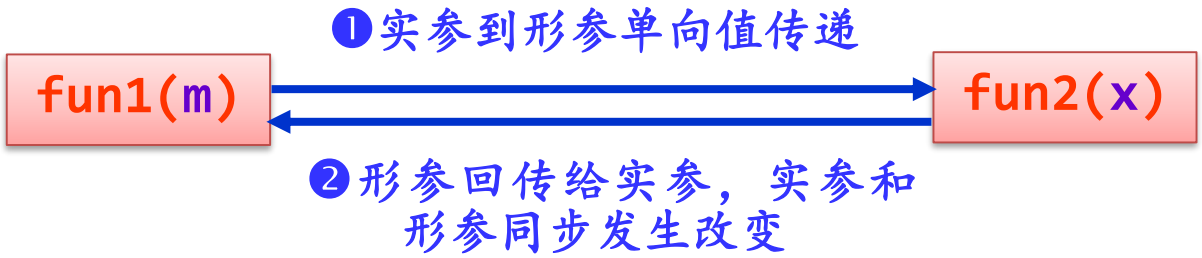
# 引用类型的参数传递

```
void fun1(int n)
{   int m=2;
    fun2( m );
    printf("%d\n", m);
}
```

实参

```
void fun2(int &x)
{   x++;
    printf("%d\n", x);
}
```

引用型形参



# 1.3 算法分析基础

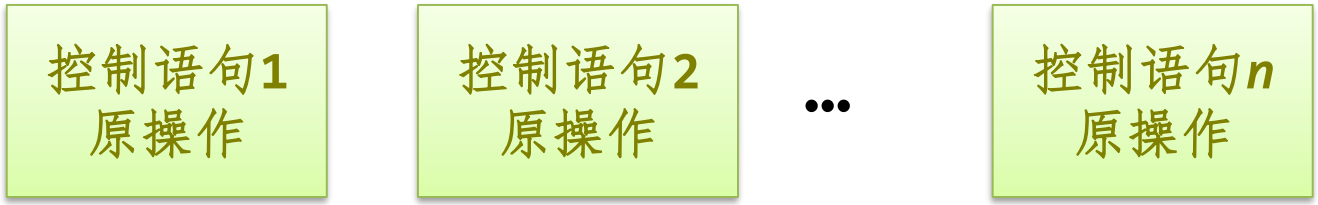
分析算法占用的资源
- CPU时间 → 时间性能分析
- 内存空间 → 空间性能分析

**算法分析目的：分析算法的时空效率以便改进算法性能。**

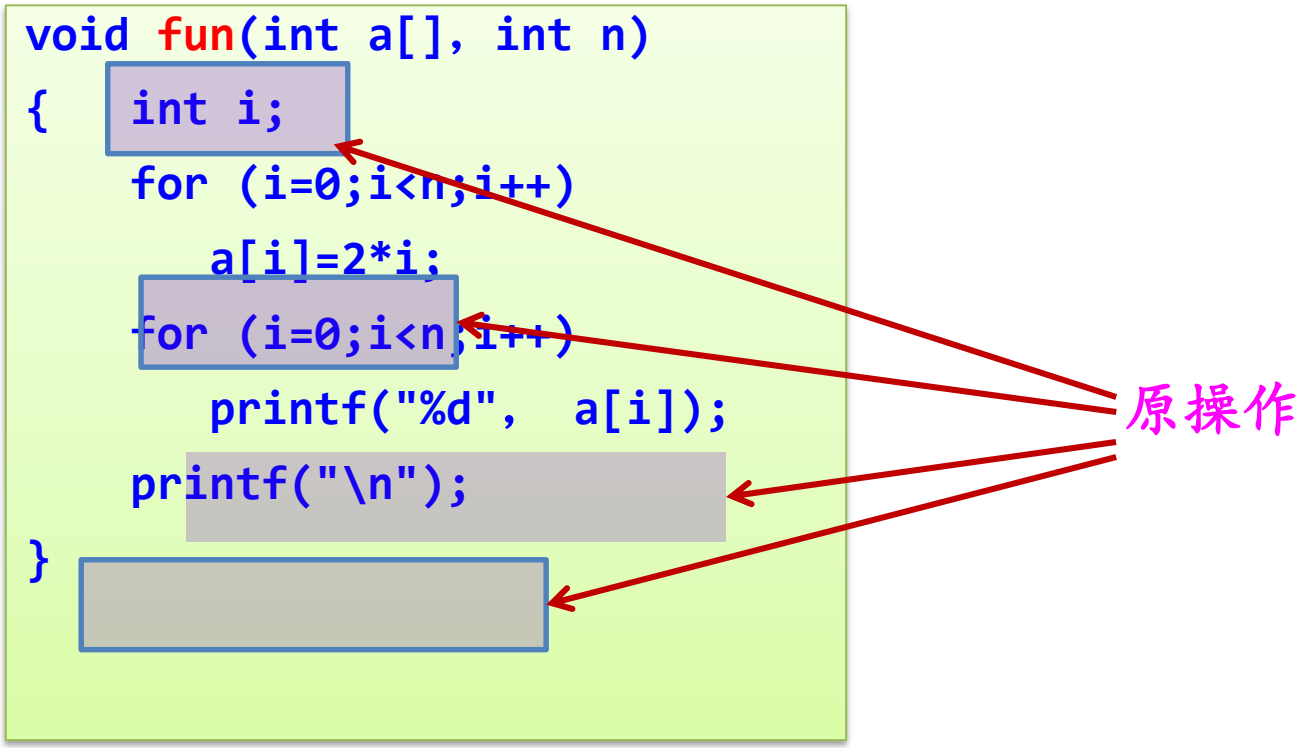
# 1.3.1 算法时间复杂度分析

一个算法是由控制结构（顺序、分支和循环三种）和原操作（指固有数据类型的操作，如+、-、*、/、++和--等）构成的。算法执行时间取决于两者的综合效果。

一个算法的基本构成：



例如：


void fun(int a[], int n)
{ int i;
for (i=0;i<n;i++)
a[i]=2*i;
for (i=0;i<n;i++)
printf("%d", a[i]);
printf("\n");
}
原操作

# 算法分析方式：

❶ 事后分析统计方法：编写算法对应程序，统计其执行时间。

- 编写程序的语言不同
- 执行程序的环境不同
- 其他因素

所以不能用绝对执行时间进行比较。

❷ 事前估算分析方法：撇开上述因素，认为算法的执行时间是问题规模$n$的函数。 ✓

## ❶ 分析算法的执行时间

- 求出算法所有原操作的执行次数（也称为频度），它是问题规模$n$的函数，用T($n$)表示。
- 算法执行时间大致 = 原操作所需的时间×T($n$)。所以T($n$)与算法的执行时间成正比。用T($n$)表示算法的执行时间。
- 比较不同算法的T($n$)大小得出算法执行时间的好坏。

用于表示求解问题大小的正整数，如$n$个记录排序

【例1-6：p19】求两个*n*阶方阵的相加*C=A+B*的算法如下，分析其时间复杂度。

```
#define MAX 20    //定义最大的方阶
void matrixadd(int n, int A[MAX][MAX], int B[MAX][MAX],
     int C[MAX][MAX])
  {  int i, j;
     for (i=0;i<n;i++)                          //①
        for (j=0;j<n;j++)                       //②
           C[i][j]=A[i][j]+B[i][j];             //③
  }
```

```
#define MAX   20    //定义最大的方阶
void matrixadd(int n, int A[MAX][MAX],
   int B[MAX][MAX],  int C[MAX][MAX])
{  int i, j;
   for (i=0;i<n;i++)                    //①
      for (j=0;j<n;j++)                 //②
         C[i][j]=A[i][j]+B[i][j]; //③
  }
```

解：除变量定义语句外，该算法包括3个可执行语句①、②和③。

①：频度为$n+1$，循环体执行$n$次

②：频度为$n(n+1)$

③：频度为$n^2$

所有语句频度之和为：

$$T(n) = n+1+n(n+1)+n^2$$

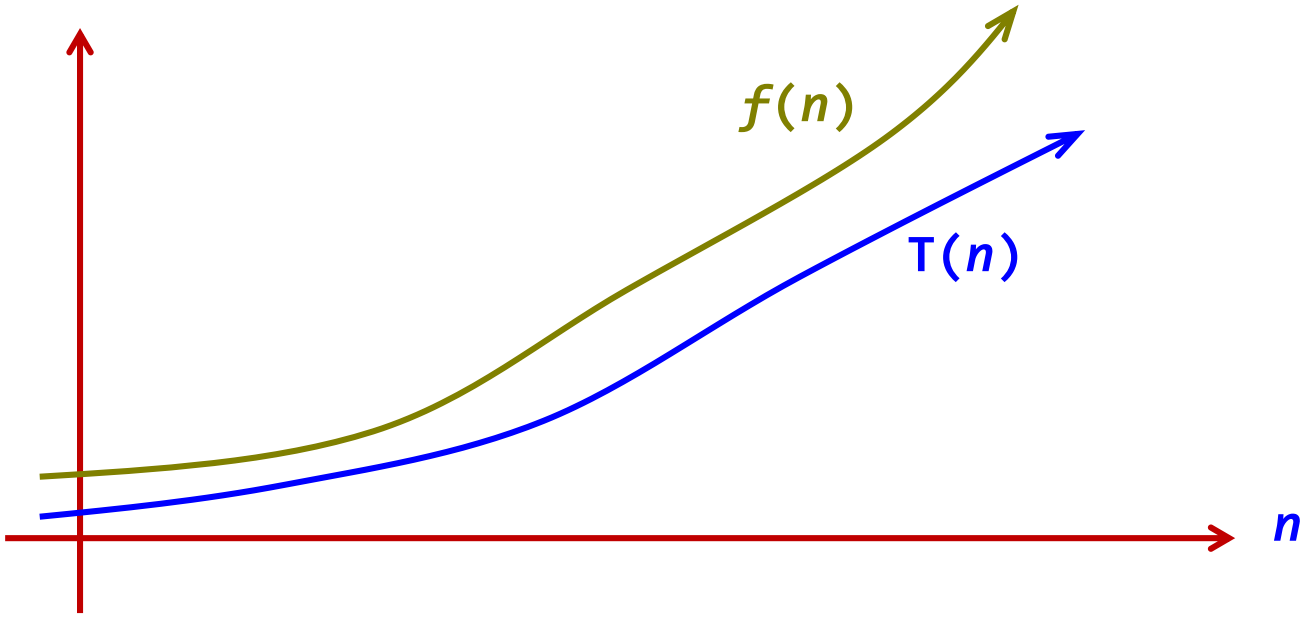
$$= 2n^2+2n+1$$

## ❷ 算法的执行时间用时间复杂度来表示

算法中执行时间T(*n*)是问题规模*n*的某个函数*f*(*n*)，记作：

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

记号“O”读作“大O”，它表示随问题规模*n*的增大算法执行时间的增长率和*f*(*n*)的增长率相同。 ⇨ 趋势分析



“O”的形式定义为：

$T(n) = O(f(n))$表示存在一个正的常数$M$，使得当$n \geq n_0$时都满足：

$$|T(n)| \leq M|f(n)|$$

$f(n)$是$T(n)$的上界

这种上界可能很多，通常取最接近的上界，即紧凑上界

大致情况：

$$\lim_{n \to \infty} \frac{T(n)}{f(n)} = M$$

本质上讲，是一种T(*n*)
最高数量级的比较

只求出T(*n*)的最高阶，忽略其低阶项和常系数 --》 既可简化T(*n*)的计算，又能比较客观地反映出当*n*很大时算法的时间性能。

例如 ： $T(n) = 2n^2+2n+1 = O(n^2)$

## 一般地：

- 一个没有循环的算法的执行时间与问题规模$n$无关，记作$O(1)$，也称作常数阶。
- 一个只有一重循环的算法的执行时间与问题规模$n$的增长呈线性增大关系，记作$O(n)$，也称线性阶。
- 其余常用的算法时间复杂度还有平方阶$O(n^2)$、立方阶$O(n^3)$、对数阶$O(\log_2 n)$、指数阶$O(2^n)$等。

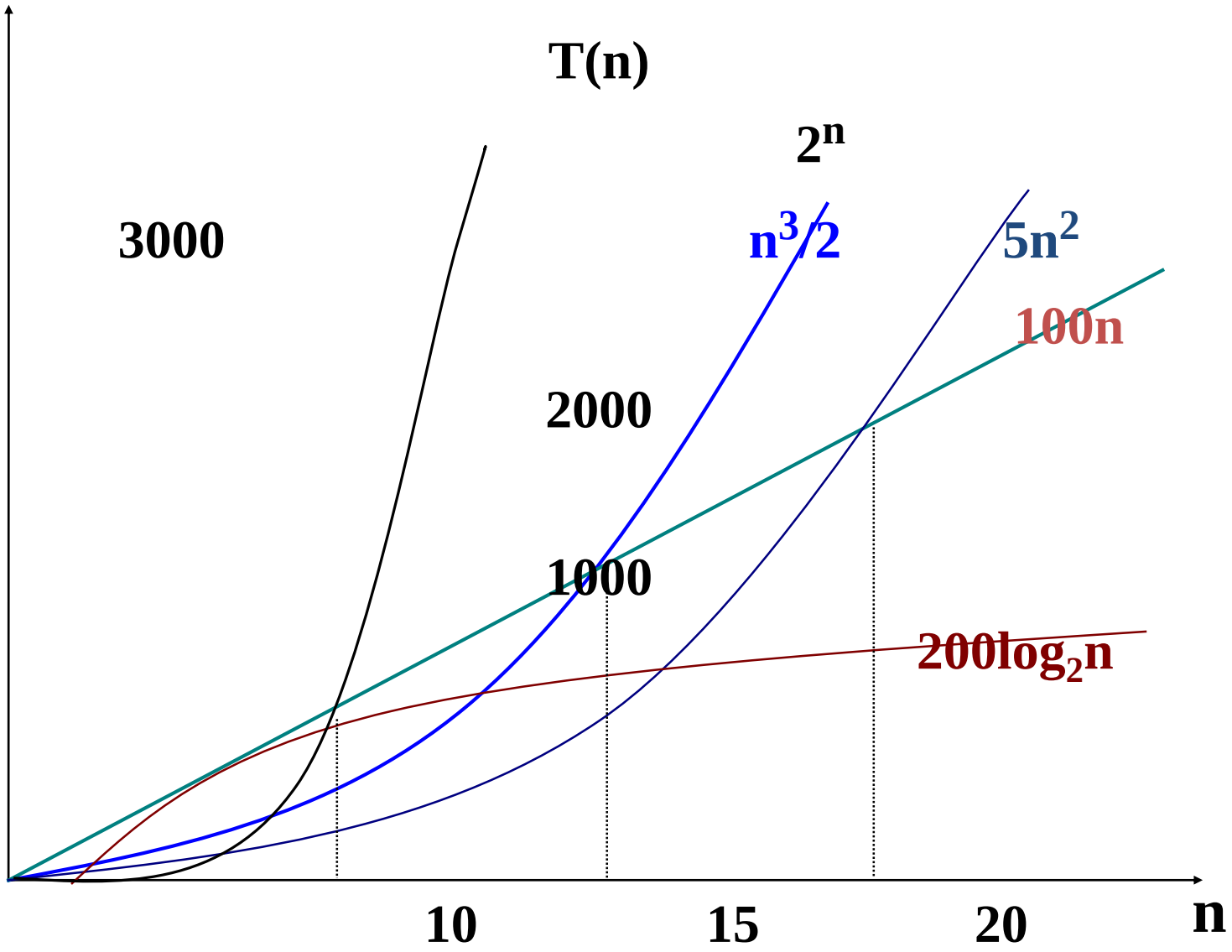
各种不同算法时间复杂度的比较关系如下：

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

多项式阶：P问题　　指数阶：NP问题

**算法时间性能比较**：假如求同一问题有两个算法：*A*和*B*，如果算法*A*的平均时间复杂度为$O(n)$，而算法*B*的平均时间复杂度为$O(n^2)$。

一般情况下，认为算法*A*的时间性能好比算法*B*。




关于问题规模n的常见函数f(n)增长率

## ❸ 简化的算法时间复杂度分析

算法中的基本操作一般是最深层循环内的原操作。

算法执行时间大致 = 基本操作所需的时间 ×其运算次数。

转化

在算法分析时，计算$T(n)$时仅仅考虑基本操作的运算次数。

【例1-6：p19】求两个*n*阶方阵的相加*C=A+B*的算法如下，分析其时间复杂度。

```
#define MAX   20     //定义最大的方阶
void matrixadd(int n, int A[MAX][MAX], int B[MAX][MAX],
      int C[MAX][MAX])
{  int i, j;
   for (i=0;i<n;i++)
      for (j=0;j<n;j++)
           C[i][j]=A[i][j]+B[i][j];
}
```

基本操作

解：该算法中的基本操作是两重循环中最深层的语句C[i][j]=A[i][j]+B[i][j]，分析它的频度，即：

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}1 = \sum_{i=0}^{n-1}n = n\sum_{i=0}^{n-1}1 = n*n = n^2$$

$$= O(n^2)$$

这种简化的时间复杂度分析方法得到的结果相同，但分析过程更简单。

# 思考题

下列程序段的时间复杂度是（C）。

```
count=0;
for(k=1;k<=n;k*=2)
        for(j=1;j<=n;j++)
                count++;
```

基本操作

A.$O(\log_2 n)$　B.$O(n)$　C.$O(n\log_2 n)$　D.$O(n^2)$

【例1-7：p20】分析以下算法的时间复杂度。

```
void func(int n)
{  int i=0, s=0;
   while (s<n)
   {  i++;
      s=s+i;
   }
}
```

基本操作

解：对于while循环语句，设执行的次数为$m$，变量$i$从0开始递增1，直到$m$为止，有：

循环结束：$s=m(m+1)/2 \geq n$，或者$m(m+1)/2+k=n$ 。

用于修正的常量

则：

$$m=\frac{-1+\sqrt{8n+1-8k}}{2}$$

$T(n)=m=O(\sqrt{n})$

所以，该算法的时间复杂度为$O(\sqrt{n})$。

# 1.3.2 算法空间复杂度分析

**空间复杂度**：用于量度一个算法在运行过程中临时占用的存储空间大小。

一般也作为问题规模$n$的函数，采用数量级形式描述，记作：

$$S(n) = O(g(n))$$

若一个算法的空间复杂度为O(1)，则称此算法为原地工作或就地工作算法。

【例（补充）】 分析如下算法的空间复杂度。

```
int fun(int n)
{  int i, j, k, s;
   s=0;
   for (i=0;i<=n;i++)
     for (j=0;j<=i;j++)
        for (k=0;k<=j;k++)
           s++;
   return(s);
}
```

临时占用的存储空间：函数体内分配的空间

解：算法中临时分配的变量个数与问题规模*n*无关，所以空间复杂度均为O(1)。

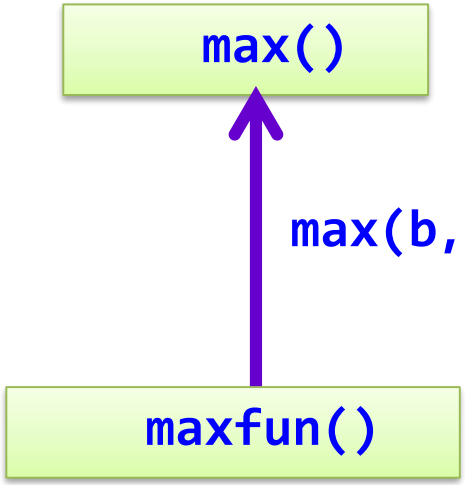
# 为什么空间复杂度分析只考虑临时占用的存储空间？

```
int max(int a[], int n)
{  int i,  maxi=0;
   for (i=1;i<=n;i++)
      if (a[i]>a[maxi])
         maxi=i;
   return a[maxi];
}
```

如果max函数中再考虑形参a的空间，
就重复累计了执行整个算法所需的空间。

max算法的空间复杂度为O(1)

```
void maxfun()
{  int b[]={1, 2, 3, 4, 5}, n=5;
   printf("Max=%d\n", max(b, n));
}
```




maxfun算法中为b数组分配了相应的内存空间，其空间复杂度为O(n)

# 1.4 其他情况的算法分析

## 1.4.1 最好、最坏和平均时间复杂度分析

**定义：**设一个算法的输入规模为$n$，$D_n$是所有输入的集合，任一输入$I \in D_n$，$P(I)$是$I$出现的概率，有$\sum_{I \in D_n} P(I) = 1$，$T(I)$是算法在输入$I$下的执行时间，则算法的**平均时间复杂度**为：

$$A(n) = \sum_{I \in D_n} P(I) \times T(I)$$

例如，10个1～10的整数序列递增排序：

$n=10$

$I_1=\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

$I_2=\{2, 1, 3, 4, 5, 6, 7, 8, 9, 10\}$

…

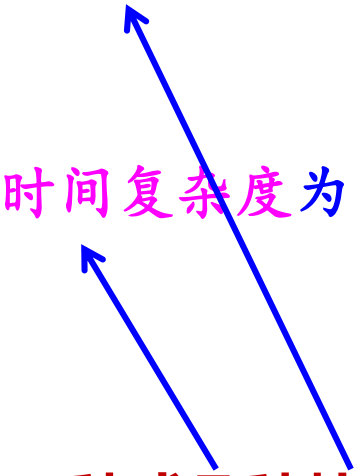
$I_m=\{10, 9, 8, 7, 6, 5, 4, 3, 2, 1\}$

构成$D_n$，$P(I)=1/m$

所有可能的初始序列有$m$个，$m=10!$

算法的最坏时间复杂度为：$W(n)=\max_{I\in D_n}\{T(I)\}$

算法的最好时间复杂度为：$B(n)=\min_{I\in D_n}\{T(I)\}$



一种或几种特殊情况

如果问题规模相同，时间代价与输入数据有关，则需要分析最好情况、最坏情况、平均情况。

| | |
|---|---|
| 最好情况 | 会被cheat、较频繁出现时才有意义 |
| 最差情况 | 性能评估的重要依据 |
| 平均情况 | 已知输入数据是如何分布的，通常假设等概率分布 |

【例1-8：p21】设计一个算法，求含$n$个整数元素的序列中前$i$（$1 \le i \le n$）个元素的最大值。并分析算法的平均时间复杂度。

解：整数序列用数组$a$表示，前$i$（$1 \le i \le n$）个元素为$a[0..i-1]$。

```
int fun(int a[], int n, int i)
{   int j, max=a[0];
    for (j=1;j<=i-1;j++)
        if (a[j]>max) max=a[j];
    return(max);
}
```

解：$i$的取值范围为1～$n$（共$n$种情况），对于求前$i$个元素的最大值时，需要元素比较$(i-1)-1+1=i-1$次。在等概率情况（每种情况的概率为$1/n$）：

$$T(n)=\sum_{i}^{n}\frac{1}{n}\times(i-1)=\frac{1}{n}\sum_{i}^{n}(i-1)=\frac{n-1}{2}$$

$= O(n)$ ⟵ 平均时间复杂度

该算法的最坏复杂度：$W(n)=O(n)$（当$i=n$时）

该算法的最好复杂度：$B(n)=O(1)$（当$i=1$时）

# 1.4.2 递归算法的时空复杂度分析

递归算法是指算法中出现调用自己的成分。

- 递归算法分析也称为变长时空分析。
- 非递归算法分析也称为定长时空分析。

# 1、递归算法的时间复杂度分析

【例1-9：p22】　有如下递归算法：

```
void fun(int a[], int n, int k) //数组a共有n个元素
{  int i;
   if (k==n-1)
     for (i=0;i<n;i++)
        printf("%d\n", a[i]); //执行n次
   else
   {  for (i=k;i<n;i++)
         a[i]=a[i]+i*i;       //执行n-k次
      fun(a, n, k+1);
   }
 }
```

调用上述算法的语句为fun(*a*, *n*, 0)，求其时间复杂度。

递归算法：

```
void fun(int a[], int n, int k) //数组a共有n个元素
{  int i;
   if (k==n-1)
     for (i=0;i<n;i++)
        printf("%d\n", a[i]); //执行n次
   else
   {  for (i=k;i<n;i++)
         a[i]=a[i]+i*i;       //执行n-k次
      fun(a, n, k+1);
   }
 }
```

含一重循环

fun(*a*, *n*, 0)的时间复杂度为O(*n*)。 ← 错误

解：设fun(*a*, *n*, 0)的执行时间为T(*n*)，fun(*a*, *n*, *k*)的执行时间为$T_1(n, k)$ ⇨ $T(n)=T_1(n, 0)$。

由fun()递归算法可知：

| | |
|---|---|
| $T_1(n, k) = n$ | 当$k=n-1$时 |
| $T_1(n, k) = (n-k)+T_1(n, k+1)$ | 其他情况 |

则

$$T(n) = T_1(n, 0) = n+T_1(n, 1) = n+(n-1)+T_1(n, 2)$$
$$= \cdots = n+(n-1)+\cdots+2+T_1(n, n-1)$$
$$= n+(n-1)+ \cdots+2+n$$
$$= O(n^2)$$

所以调用fun(*a*, *n*, 0)的时间复杂度为$O(n^2)$。

# 2、递归算法的空间复杂度分析

【例1-11：p23】有如下递归算法，分析调用fun(*a*, *n*, 0)的空间复杂度。

```
void fun(int a[], int n, int k) //数组a共有n个元素
{  int i;
   if (k==n-1)
     for (i=0;i<n;i++)
        printf("%d\n", a[i]); //执行n次
   else
   {  for (i=k;i<n;i++)
         a[i]=a[i]+i*i;       //执行n-k次
      fun(a, n, k+1);
   }
 }
```

递归算法：

```
void fun(int a[], int n, int k) //数组a共有n个元素
{  int i;
   if (k==n-1)
     for (i=0;i<n;i++)
        printf("%d\n", a[i]); //执行n次
   else
   {  for (i=k;i<n;i++)
         a[i]=a[i]+i*i;       //执行n-k次
      fun(a, n, k+1);
   }
 }
```

仅仅定义了一个临时变量i

fun(*a*, *n*, 0)的空间复杂度为O(1)。 ← 错误

解：设fun(*a*, *n*, 0)的空间为S(*n*)，fun(*a*, *n*, *k*)的空间为$S_1(n, k)$ ⇨ $S(n) = S_1(n, 0)$。

由fun()递归算法可知：

| | |
|---|---|
| $S_1(n, k) = 1$ | 当$k=n-1$时 |
| $S_1(k) = 1+S_1(n, k+1)$ | 其他情况 |

则：

$$S(n) = S_1(n, 0) = 1+S_1(n, 1) = 1+1+S_1(n, 2)$$
$$= \cdots = \underbrace{1 + 1 + \cdots + 1}_{n个1} = O(n)$$

所以调用fun(*a*, *n*, 0)的空间复杂度为O(*n*)。

## 思考题

递归算法和非递归算法在分析时间复杂度和空间复杂度上为什么不同？

# 有效算法的重要性

- 如果一台计算机 1 秒能处理1000个数据，那么如果处理n个数据所需执行的秒数如表中的函数所示。

| 时间函数 | n=20 | n=50 | n=100 | n=500 |
|---|---|---|---|---|
| n | .02s | .05s | .1s | .5s |
| nlogn | .09s | .3s | .6s | 4.5s |
| $n^2$ | .4s | 2.5s | 10s | 250s |
| $n^3$ | 8s | 2m | 17m | 35h |
| $2^n$ | 0.3h | 35702y | | |

# 可处理的数据量

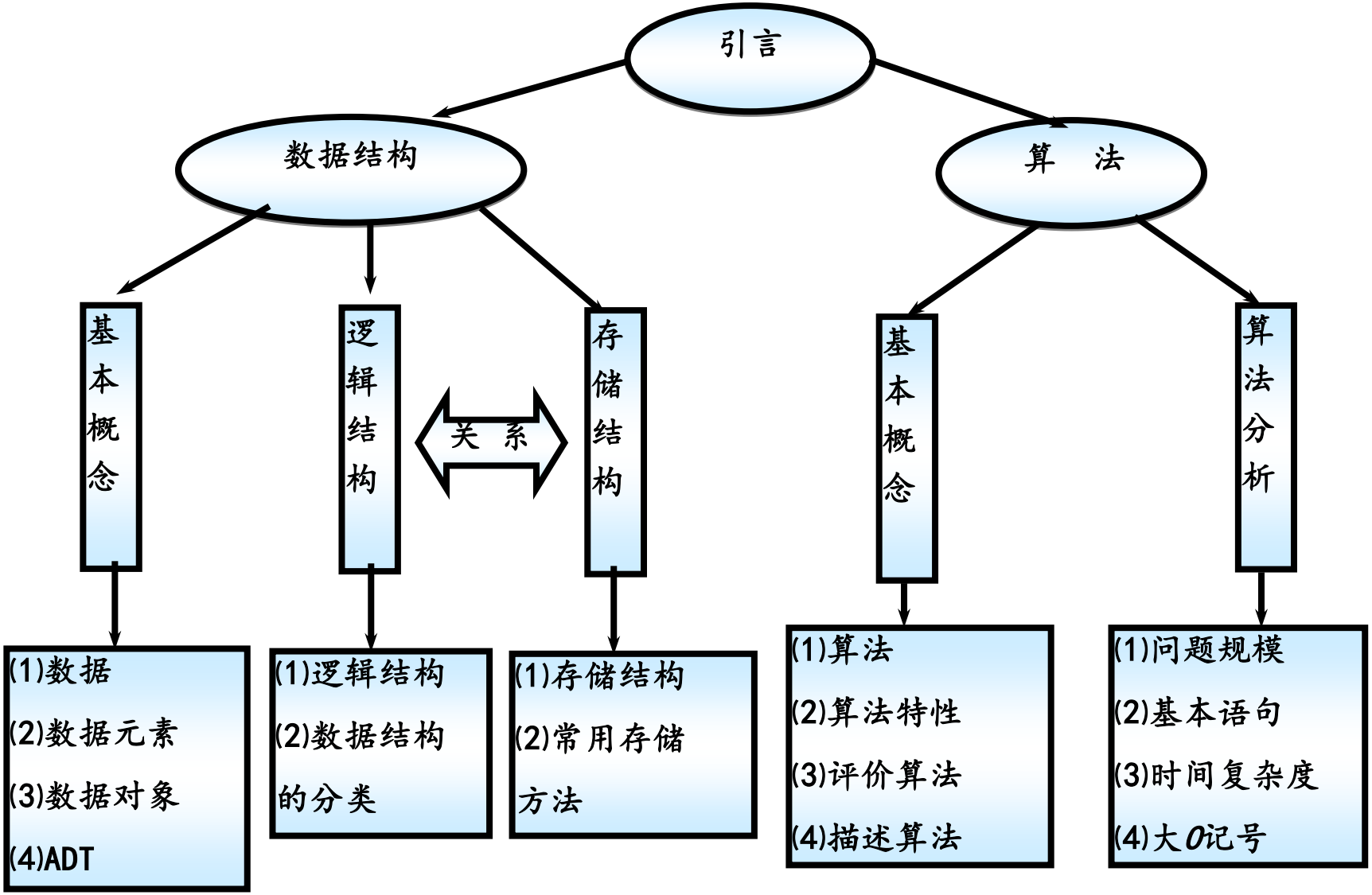
| 时间函数 | 在 1 秒内 | 在 1 分钟内 | 在 1 小时 |
| --- | --- | --- | --- |
| n | 1000 | $6 * 10^4$ | $3.6 * 10^6$ |
| nlogn | 140 | 4893 | $2 * 10^5$ |
| $n^2$ | 31 | 244 | 1897 |
| $n^3$ | 10 | 39 | 153 |
| $2^n$ | 10 | 15 | 21 |

# 有效算法的重要性

| 时间函数 | 提速10倍前的求解规模 | 提速10倍后的求解规模 |
| --- | --- | --- |
| n | S1 | 10S1 |
| nlogn | S2 | 10S2 |
| $n^2$ | S3 | 3.16S3 |
| $n^3$ | S4 | 2.15S4 |
| $2^n$ | S5 | S5 + 3.3 |

**关键**：提高算法的效率而不是提高机器的速度！

# 小 结


引言
数据结构
算 法
基本概念
逻辑结构
关 系
存储结构
基本概念
算法分析
(1)数据
(2)数据元素
(3)数据对象
(4)ADT
(1)逻辑结构
(2)数据结构的分类
(1)存储结构
(2)常用存储方法
(1)算法
(2)算法特性
(3)评价算法
(4)描述算法
(1)问题规模
(2)基本语句
(3)时间复杂度
(4)大O记号

# Data Structure

END