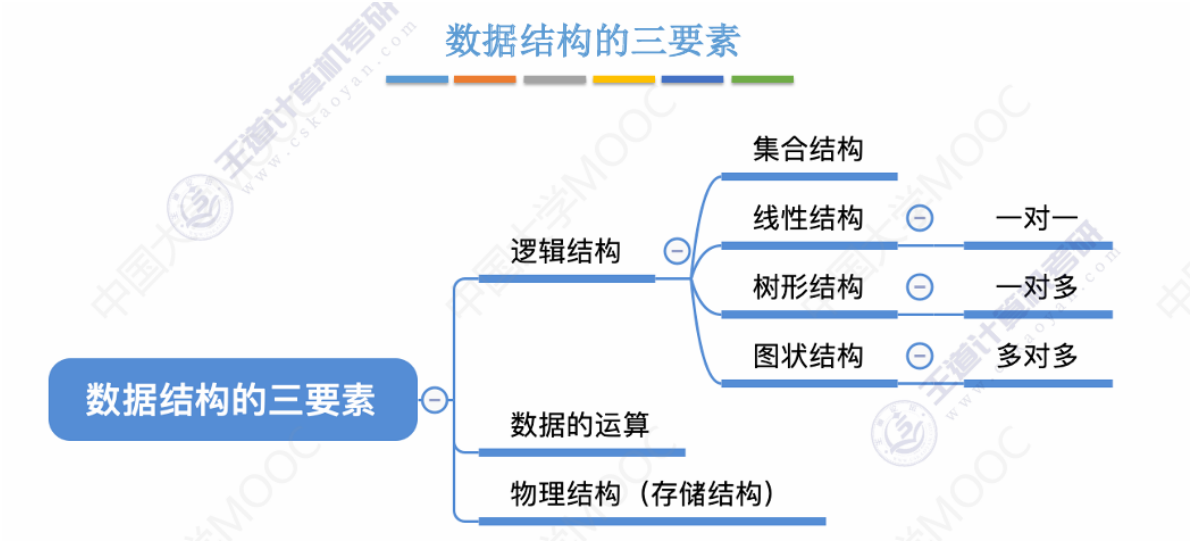
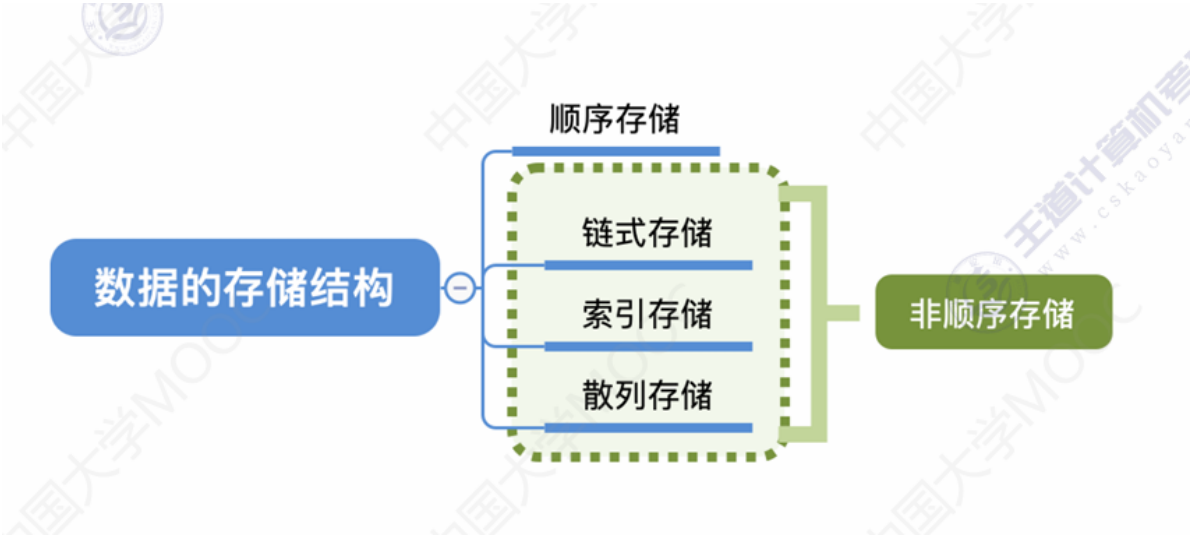
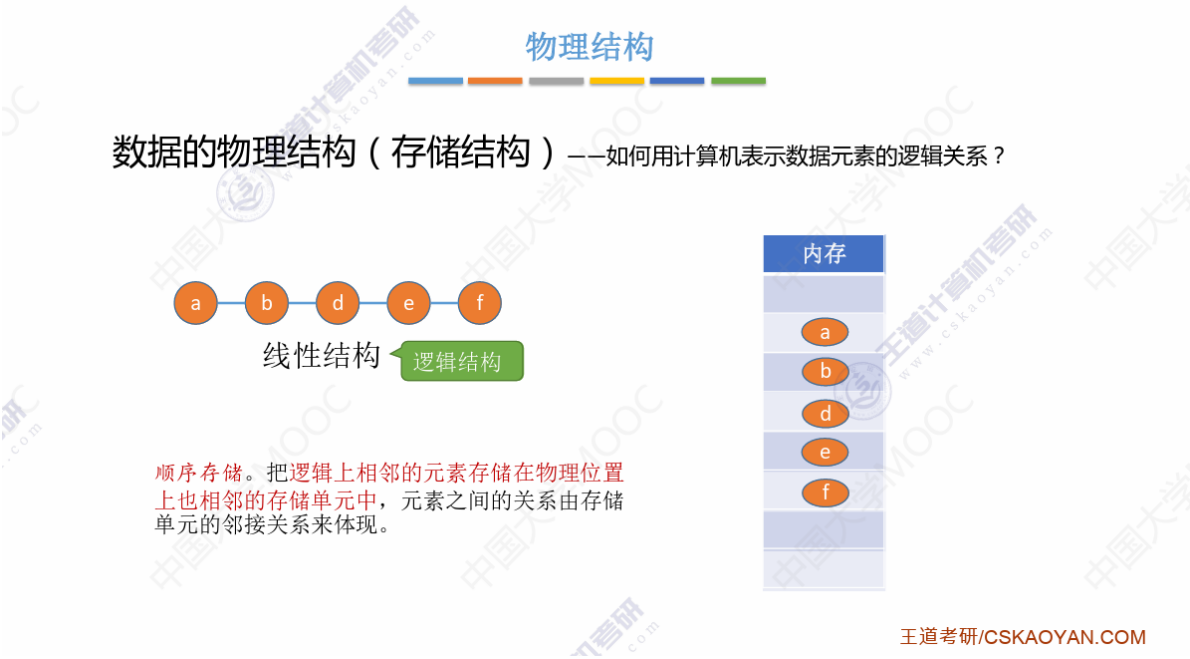
1.1 数据结构的基本概念





顺序存储

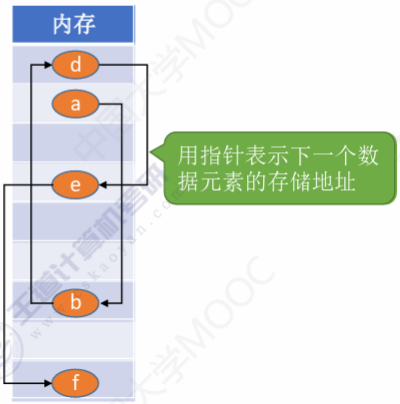


链式存储

物理结构

数据的物理结构（存储结构）——如何用计算机表示数据元素的逻辑关系？

线性结构 逻辑结构



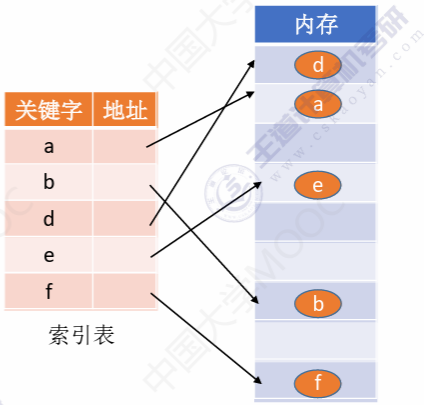
链式存储。逻辑上相邻的元素在物理位置上可以不相邻，借助指示元素存储地址的指针来表示元素之间的逻辑关系。

索引存储

物理结构

数据的物理结构（存储结构）——如何用计算机表示数据元素的逻辑关系？

线性结构 逻辑结构



索引存储。在存储元素信息的同时，还建立附加的索引表。索引表中的每项称为索引项，索引项的一般形式是（关键字，地址）

散列存储

物理结构

数据的物理结构（存储结构）——如何用计算机表示数据元素的逻辑关系？

线性结构 逻辑结构

这真是一门
让人头秃的科目

放轻松 你可以的

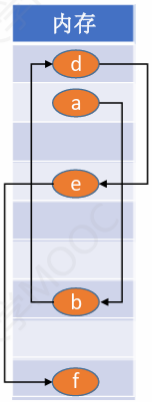
散列存储。根据元素的关键字直接计算出该元素的存储地址，又称哈希（Hash）存储

第六章，散列表

数据结构的三要素

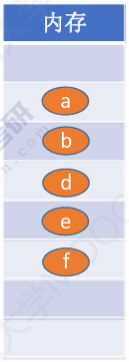
1. 若采用顺序存储，则各个数据元素在物理上必须是连续的；若采用非顺序存储，则各个数据元素在物理上可以是离散的。
2. 数据的存储结构会影响存储空间分配的方便程度
3. 数据的存储结构会影响对数据运算的速度

Eg：在b和d之间插入新元素c



链式存储

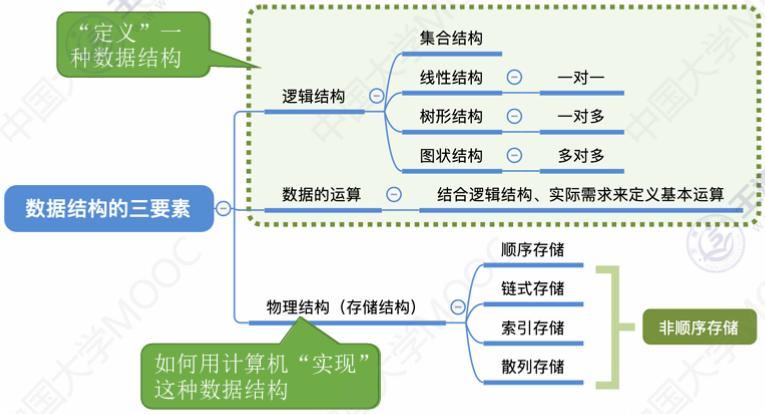
线性结构 逻辑结构



顺序存储

运算的定义是针对逻辑结构的，指出运算的功能；
运算的实现是针对存储结构的，指出运算的具体操作步骤。

数据结构的三要素



数据类型、抽象数据类型

数据类型、抽象数据类型：

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。
1）原子类型。其值不可再分的数据类型。
2）结构类型。其值可以再分解为若干成分（分量）的数据类型。

bool 类型

值的范围：true、false
可进行操作：与、或、非...

int 类型

值的范围：-2147483648 ~ 2147483647
可进行操作：加、减、乘、除、模运算...

```
struct Coordinate {
    int x;    //横坐标
    int y;    //纵坐标
};
```

定义一个具体的结构类型，表示一个坐标信息

x、y 各占 4B，用补码表示。
<x, y> 是有序对，不可互换。

可进行的操作：加、减、计算到原点的距离...

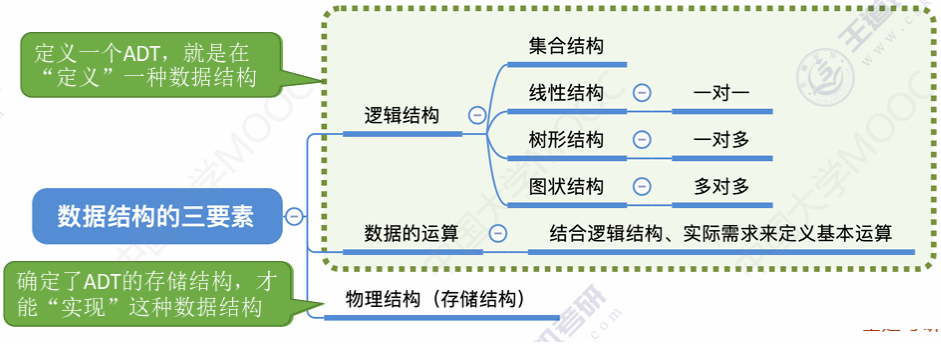
数据类型、抽象数据类型

数据类型、抽象数据类型：

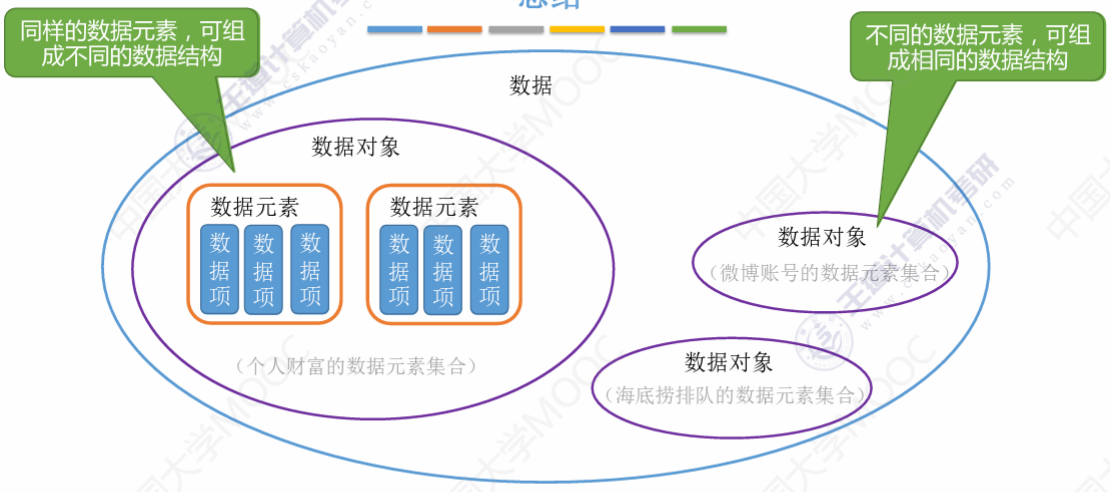
数据类型是一个值的集合和定义在此集合上的一组操作的总称。
1）原子类型。其值不可再分的数据类型。
2）结构类型。其值可以再分解为若干成分（分量）的数据类型。

ADT 用数学化的语言定义数据的逻辑结构、定义运算。与具体的实现无关。

抽象数据类型（Abstract Data Type，**ADT**）是抽象数据组织及与之相关的操作。



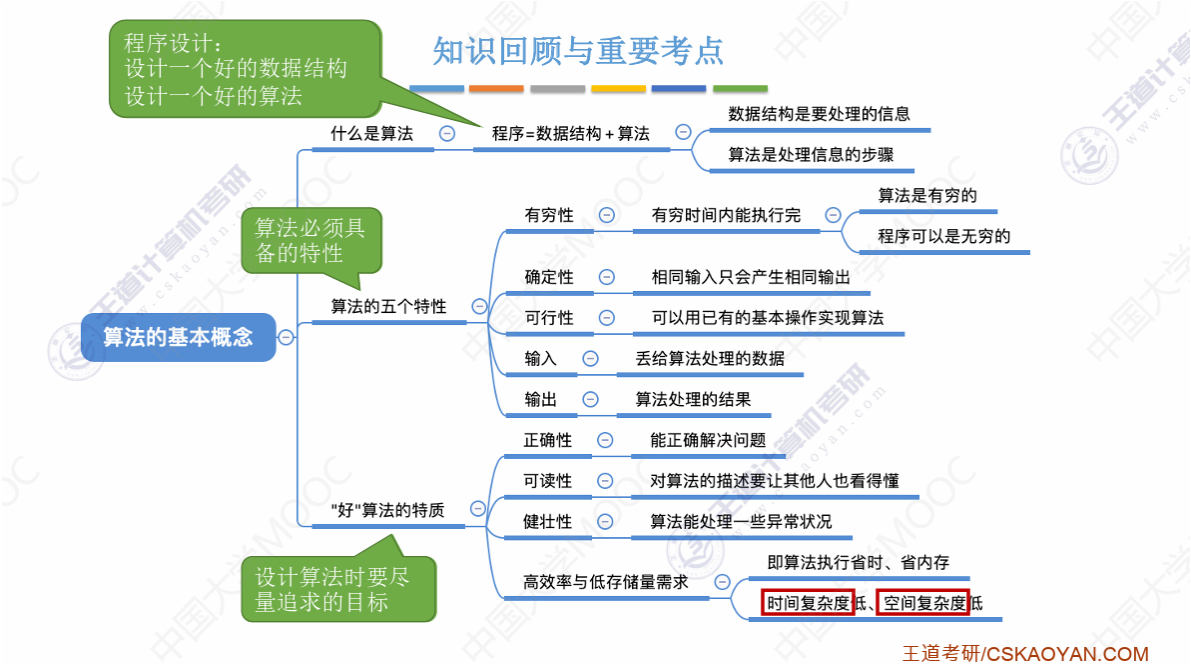
总结



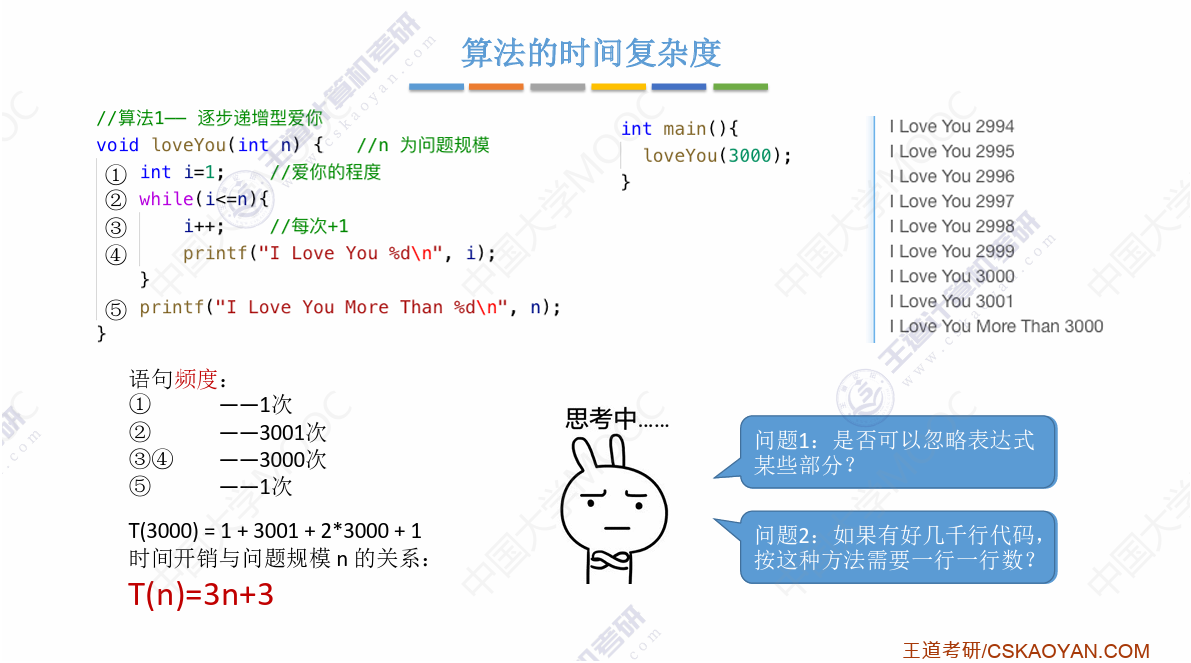
数据结构这门课着重关注的是数据元素之间的关系，和对这些数据元素的操作，而不关心具体的数据项内容

1.2 算法和算法评价

算法基本概念



时间复杂度



算法的时间复杂度

全称：渐进时间复杂度

问题1：是否可以忽略表达式某些部分？

当问题规模 n 足够大时…

大O表示“同阶”，同等数量级。即：当 n→∞时，二者之比为常数

$T_1(n) = O(n)$
$T_2(n) = O(n^2)$
$T_3(n) = O(n^3)$

简化

时间开销与问题规模 n 的关系：

$T_1(n)=3n+3$

$T_2(n)=n^2+3n+1000$

$T_3(n)=n^3 + n^2 +9999999$

若 n=3000，则

$3n = 9000$	V.S.	$T_1(n)= 9003$
$n^2 = 9,000,000$	V.S.	$T_2(n)= 9,010,000$
$n^3 = 27,000,000,000$	V.S.	$T_3(n)= 27,018,999,999$

结论1：可以只考虑阶数高的部分

当 n=3000 时，$9999n = 29,997,000$ 远小于 $n^3 = 27,018,999,999$
当 n=1000000时，$9999n = 9,999,000,000$ 远小于 $n^2 = 1,000,000,000,000$

结论2：问题规模足够大时，常数项系数也可以忽略

a）加法规则

$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

多项相加，只保留最高阶的项，且系数变为1

b）乘法规则

$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$

多项相乘，都保留

Eg：$T_3(n) = n^3 + n^2 \log_2 n$
$= O(n^3) + O(n^2 \log_2 n)$
$= ? \ ? \ ?$

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

语句频度：

①　——1次
②　——3001次
③④　——3000次
⑤　——1次

T(3000) = 1 + 3001 + 2*3000 + 1

时间开销与问题规模 n 的关系：

$T(n) = 3n+3 = O(n)$

问题1：是否可以忽略表达式某些部分？

只考虑阶数，用大O记法表示

问题2：如果有好几千行代码，按这种方法需要一行一行数？

算法的时间复杂度

```
//算法2—— 嵌套循环型爱你
void loveYou(int n) {   //n 为问题规模
    int i=1;    //爱你的程度
    while(i<=n){
        i++;    //每次+1
        printf("I Love You %d\n", i);
        for (int j=1; j<=n; j++){
            printf("I am Iron Man\n");
        }
    }
    printf("I Love You More Than %d\n", n);
}
```

外层循环执行n次

嵌套两层循环

内层循环共执行n^2次

时间开销与问题规模 n 的关系：

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

结论1：顺序执行的代码只会影响常数项，可以忽略

结论2：只需挑循环中的一个基本操作分析它的执行次数与 n 的关系即可

结论3：如果有多层嵌套循环，只需关注最深层循环循环了几次

机智如我

练习1

```
void loveYou(int n){
  int i = 1;
  while(i<=n){
    i=i*2;
    printf("I Love You  %d Times\n",i);
  }
  printf("I love you more than %d times\n",n);
}
```

O(log2n)

练习2

```
void loveYou(int flag[],int n){
    printf("I am Icon Man\n");
    int i;
    for(i=0;i<n;i++){
        if(flag[i]==n){
            printf("I love you");
            break;
        }
    }
}

//flag数组中乱序存放了1~n这些数字
```

最好情况：元素n在第一个位置 ——最好时间复杂度 T(n)=O(1)
最坏情况：元素n在最后一个位置 ——最坏时间复杂度 T(n)=O(n)
平均情况：假设元素n在任意一个位置的概率相同为 $\frac{1}{n}$ ——平均时间复杂度 T(n)=O(n)

循环次数 x = (1+2+3+...+n) $\frac{1}{n}$ = $(\frac{n(1+n)}{2})\frac{1}{n}$ = $\frac{1+n}{2}$ T(n)=O(x)=O(n)

小故事：算法的性能问题只有在 n 很大时才会暴露出来。

空间复杂度

程序运行时的内存需求

```
//算法1—— 逐步递增型爱你
void loveYou(int n) {   //n 为问题规模
    int i=1;    //爱你的程度
    while(i<=n){
        i++;    //每次+1
        printf("I Love You %d\n", i);
    }
    printf("I Love You More Than %d\n", n);
}
```

装入

无论问题规模怎么变，算法运行所需的内存空间都是固定的常量，算法空间复杂度为

$S(n) = O(1)$

注：S 表示 “Space”

算法原地工作——算法所需内存空间为常量

空间复杂度

```
void test(int n) {
    int flag[n];    //声明一个长度为n的数组
    int i;
    //.....此处省略很多代码
}
```

装入

假设一个 int 变量占 4B...
则所需内存空间 = 4 + 4n + 4 = 4n + 8

$S(n) = O(n)$

只需关注存储空间大小与问题规模相关的变量

空间复杂度

```
void test(int n) {
    int flag[n][n]; //声明 n*n 的二维数组
    int i;
    //.....此处省略很多代码
}
```

装入

$S(n) = O(n^2)$

空间复杂度

```
void test(int n) {
    int flag[n][n]; //声明 n*n 的二维数组
    int other[n];   //声明一个长度为n的数组
    int i;
    //.....此处省略很多代码
}
```

装入

$S(n) = O(n^2)+O(n)+O(1) = O(n^2)$

a）加法规则

$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

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

函数递归调用带来的内存开销

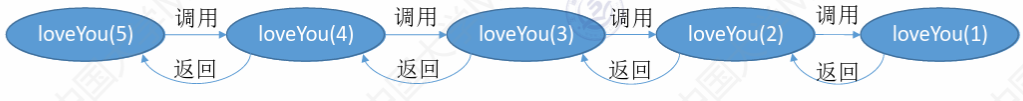
```
//算法5— 递归型爱你
void loveYou(int n) {    //n 为问题规模
    int a,b,c;  //声明一系列局部变量
    //...省略代码
    if (n > 1) {
        loveYou(n-1);
    }
    printf("I Love You %d\n", n);
}

int main(){
  loveYou(5);
}
```

```
I Love You 1
I Love You 2
I Love You 3
I Love You 4
I Love You 5
```

装入

$S(n) = O(n)$

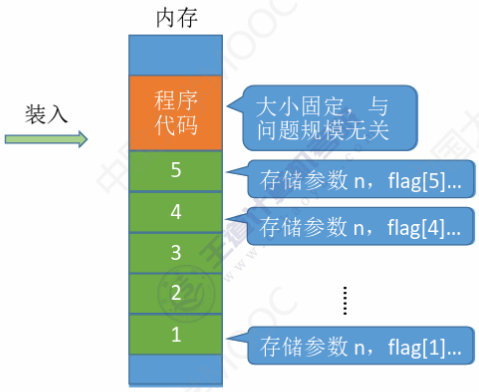
空间复杂度 = 递归调用的深度



函数递归调用带来的内存开销

```
//算法5—— 递归型爱你
void loveYou(int n) {    //n 为问题规模
    int flag[n];     //声明一个数组
    //...省略数组初始化代码
    if (n > 1) {
        loveYou(n-1);
    }
    printf("I Love You %d\n", n);
}

int main(){
  loveYou(5);
}
```

```
I Love You 1
I Love You 2
I Love You 3
I Love You 4
I Love You 5
```



$1+2+3+...+n = [n(1+n)]/2 = \frac{1}{2}n^2 + \frac{1}{2}n$

$S(n) = O(n^2)$

知识回顾与重要考点