# 数据结构

## 定义

我们如何把显示中大量而复杂的问题以特定的数据类型和特定的存储结构保存到主存储器（内存）中，以及在此基础上为实现某个功能而执行的相应操作。这个相应操作也叫算法。

通俗地讲：

数据结构 = 个体（数据） + 个体（数据）的关系

算法 = 对存储数据的操作

注：参考高一凡实现严蔚敏的C/C++代码

## 特点

程序 = 数据存储 + 数据操作 + 可以被计算机执行的语言

# 算法

## 定义

解题的方法和步骤

## 衡量标准

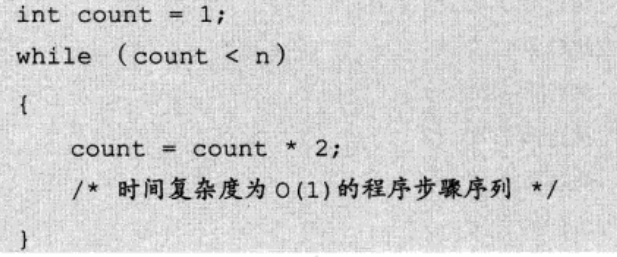
### 时间复杂度

大概程序执行的次数，而非执行的具体时间（因为具体机器时间不同）。

#### 常数阶

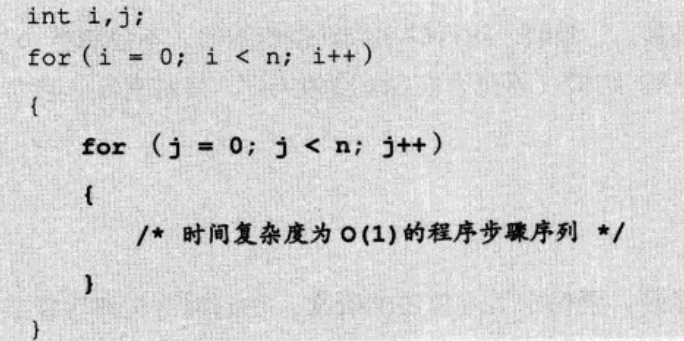
#### 线性阶

#### 对数阶



由于每次count乘以2之后，就距离n更近了一分。也就是说，有多少个2相乘后大于n，则会退出循环。由2x=n得到x=log2n。所以这个循环的时间复杂度为O(logn)。

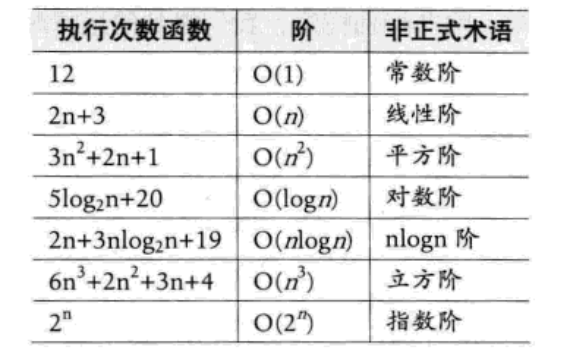
#### 平方阶



对于内循环，时间复杂度为O(n)。而对于外层的循环，不过是内部这个事件复杂度为O(n)的语句，再循环n次。所以这段代码的时间复杂度为O(n2)。

如果外循环的循环次数改为了m，时间复杂度变为O(m×n)。

常见的时间复杂度：



### 空间复杂度

算法执行过程中大概所占用的最大内存。

算法的空间复杂度通过计算算法所需的存储空间实现，算法空间复杂度的计算公式记作：S(n)=O(f(n))，其中，n为问题的规模，f(n)为语句关于n所占存储空间的函数。

一般情况下，一个程序在机器上执行时，除了需要存储程序本身的指令、常数、变量和输入数据外，还需要存储对数据操作的存储单元。若输入数据所占空间只取决于问题本身，和算法无关，这样只需要分析该算法在实现时所需的辅助单元即可。若算法执行时所需的辅助空间相对于输入数据量而言是个常数，则称此算法为原地工作，空间复杂度为O(1)。

通常，我们都使用“时间复杂度”来指运行时间的需求，使用“空间复杂度”指空间需求。当不用限定词地使用“复杂度”时，通常都是指时间复杂度。

### 难易程度

### 健壮性

注：主要标准是时间和空间复杂度。

# 基础知识

## 指针

内存的基本划分单位是字节。每个字节含有8位，字节和编号是一一对应的，每一个字节都有一个唯一确定的编号，一个编号对应一个字节，这个编号也叫地址。

一个软件分配到的空间中极有可能存在着以前其他软件使用过的残留数据，这些数据被称之为垃圾数据。所以通常情况下我们为一个变量，数组分配好内存空间后都要对该内存空间初始化。

地址：内存单元的编号（CPU通过地址线，控制线，数据线实现内存的操作，地址线寻址，控制线控制读写操作，数据线传输数据）。

范围：0~0XFFFFFFFF（4G-1）

指针：指针就是地址，地址就是指针；指针变量：存放内存地址单元的变量（本质上是一个操作受限的非负整数）。

所有指针变量只占4个字节。

指针变量不能相加，不能想成，不能相除。如果两指针变量同属于一个数组，则可以相减。指针变量可以加减一整数（指针p）：

p+i的值是p+i\*（p所指向的变量所占的字节数）

p-i的值是p-i\*（p所指向的变量所占的字节数）

P++ 🡸🡪p+1

如何通过被调函数修改主调函数中普通变量的值：

1. 实参为相关变量的地址；
2. 形参为以该变量的类型为类型的指针变量；
3. 在被调函数中通过“\*形参变量名”的方式就可以修改主调函数中普通变量的值。

## 结构体

结构体是用户根据实际需要自己定义的复合数据类型。

使用结构体的两种方式：

struct Student st = {1,’xx’,29};

struct Student \*pst = &st;

赋值：st.sid;

pst->sid

注意：

1、结构体变量不能加减乘除，但可以相互赋值。

2、普通结构体变量和结构体指针变量作为函数传参，不建议使用普通结构体变量，拷贝结构体耗费内存。

## 动态内存

内存分配：

int \*pArr = (int \*)malloc(sizeof(int) \* len);

\*pArr = 1; //相当于pArr[0]=4

pArr[1]=10; //相当于Arr[1]=10

# 线性存储结构

把所有的节点用一个直线穿起来。

## 顺序表

一张顺序表具有以下特征：

1、有一个唯一的表名来标识该顺序表；

2、内存单元连续存储，也就是说，一张顺序表要占据一块连续的内存空间；

3、数据顺序存放，元素之间由先后顺序。

### 定义

有两种定义顺序表的方法：一是静态地定义一张顺序表，二是动态地生成一张顺序表。

静态地定义一张顺序表的方法与定义一个数组的方法类似，如下：

#define MaxSize 100

ElemType SqlList[MaxSize];

int len;

动态地生成一张顺序表的方法如下：

#define MaxSize 100

typedef struct {

ElemType \*elem;

int length;

int listsize;

}SqlList;

void initSqlList(SqlList \*L)

{

L->elem = (int\*)malloc(MaxSize\*sizeof(ElemType));

if(!L->elem) exit(0);

L->length = 0;

L->listsize = MaxSize;

}

### 插入

静态顺序表插入：

void InsertElem(ElemType SqlList[], int &n, int i, ElemType item)

{

//向顺序表SqlList中第i个位置插入元素item,该顺序表长度为n

int t;

if(n==MaxSize || i<1 || i>n+1)

exit(0);

for(t=n-1;t>i-1;t--)

SqlList[t+1] = SqlList[t]; //将i-1以后的元素顺序向后移一个元素位置

SqlList[i-1] = item;

n = n + 1;

}

动态顺序表插入：

void InsertElem(SqlList \*L, int i, ElemType item)

{

//向顺序表L中第i个位置插入元素item

ElemType \*base, \*insertPtr, \*p;

if(i<1 || i>L->listSize+1)

exit(0);

if(L->length>=L->listSize)

{

base = (ElemType\*)realloc(L->elem,(L->listSize+10)\*sizeof(ElemType));

L->elem = base; //更新内存基地址

L->listSize += 100;

}

insertPtr = &(L->elem[i-1]); //insertPtr为插入位置

for(p=&(L->elem[l->length-1);p>=insertPtr;p--)

\*(p+1) = \*p; //将i-1以后的元素顺序向后移一个元素位置

\*insertPtr = item; //在第i个位置上插入元素item

L->length++;

}

### 删除

**静态顺序表删除：**

void DelElem(ElemType SqlList[], int &n, int i)

{

int j;

if(i<1 || i>n)

exit(0);

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

SqlList[j-1] = SqlList[j]; //将第i位置以后的元素依次前移（与插入操作相反）

n--;

}

**动态顺序表删除：**

void DelElem(SqlList \*L, int i)

{

ElemType \*delItem,\*q;

if(i<1 || i>L->length)

exit(0);

delItem = &(L->elem[i-1]); //delItem指向第i个元素

q = L->elem + L->length-1; //q指向队尾

for(++delItem; delItem<=q; delItem)

\*(delItem-1) = \*delItem; //将第i位置以后的元素依次前移

L->length--;

}

## 连续存储

### 数组

## 离散存储

### 链表

## 应用

### 栈

### 队列

# 非线性存储结构

## 树

## 图

# 查找

## 折半查找

# 排序

## 冒泡排序

## 快速排序

## 插入排序

## 选择排序

## 归并排序