# 数据结构与算法分析

程序是由算法，数据结构，程序设计设计方法，语言环境和工具四个部分组成，算法是思想和灵魂，数据结构是骨架，两者都很重要。学一门编程语言不难，但是要精通一些算法和数据结构方面的东西，需要悟性。刚好在C语言的学习中看到排序算法那一章，就阻塞一下，准备看完《数据结构与算法分析》这本书，加油^< ^

## 前言

### 程序语言简述和特点

数据结构主要研究组织大量数据的方法；

而算法分析则是对算法运行时间的评估；

C代替Pascal的使用使得动态分配数组称为可能；

本书讨论的内容：

* 选择问题：设一组N个数而要确定其中第k个最大者；
* 流行字谜问题：
* 等等

### 数学知识复习

1. 指数；
2. 对数；
3. 级数：

1. 横运算（同余）：意味着A和B去除以N所得的余数相同

1. 证明方法：数学归纳法和反证法

### 递归(recursive)简论

定义：当一个函数可以用它自己去定义的时候称为递归；

C允许递归，但是C仅仅是提供递归思想的一种企图，不是所有的数学递归都可以有效的，或者说是正确的由C额递归模拟来实现。（我估计是考虑会有数值越界等情况的出现）

基本的写法都需要有一个条件作为退出的触发

int recursive(){

if(){

return 0 //结束递归，并提供初值

}else{

//调用递归函数

}

}

往往有这样的观点：能不用递归就不用递归，递归都可以用迭代来代替。

递归的四条法则：

* 基准情形
* 不断推进
* 设计法则

假设所有的递归调用都能运行

递归的主要问题是隐含的簿记开销，虽然这些开销几乎都是合理的，不仅简化了算法设计，而且简化了算法

* 合成效益法则

在求解同一个问题的同一个实例的时候，切勿在不同的递归调用中做重复性的事情。

### 我对递归的理解

首先把算法变成函数的递推关系式，从结果推导到初始。

一般写法：

int recrusive(int a){

while(持续条件){

if(初始条件1){

//初始结果1；

}else if(初始条件2){

//初始结果2；

……

}else{

//递归操作

if(递归条件1){

return recrusive(int a)

} else if(递归条件2){

return recrusive(int a)

}

……

return recrusive(int a) //其余条件

}

}

return 持续条件不满足时的结果

}

循环则是从初始推导到结果

While(){

If(结束条件){

Return;

}else(){

Break;

}

}

Return

## 算法分析

算法：为求解一个问题需要遵循的被清楚地制定的简单指令的集合

### 数学基础

* 定义1：如果存在正常数c和n0使得当N>n0时，，则记为；
* 定义2：如果存在正常数c和n0使得当N>n0时，T(N)>cg(N)，则记为；
* 定义3：当且仅当且；
* 定义4：如果且，则

以上概念目的是建立一种数量比较的相对的级别，比较两个数的大小其实并没有什么太大的意义，我们一般比较他们的相对增长率（relative rate of growth）。

比如，这种记法称为大O记法，人们常常不说“……级的”，而是说“大O……”。

* 法则1：如果而且，那么

* 法则2：如果是一个k次多项式，则
* 法则3：对于任意常数k，

典型的增长率比较：常数<对数级<对数平方级<线性级<线性级\*对数级<平方级<立方级<指数级

比较的时候可以使用工具——洛必达法则

* 结果为0，意味着
* 结果为c≠0，意味着
* 结果为∞，意味着
* 结果为极限摇摆，二者无关

### 模型和要分析的问题

模型假设

* 计算模型是一台标准的计算机
* 简单的指令系统：加法，乘法，比较和赋值
* 任何指令的执行都恰好花费一个时间单元
* 整数有固定的4个字节，或者32个比特
* 无限内存

假设的局限性：不是所有运算都恰好花费相同的时间，没有考虑数据读写的时间

要分析的问题

* 运行时间

这个运行时间一般取最恶劣情况下的时间，一方面对所有的输入提供了一个界限；另一方面平均情况下的时间比较难去计算

### 运行时间的计算

为了简化分析，约定：

* 不存在特定的时间单位，因此，我们抛弃一些常数系数和低阶项，计算大O运算时间。
* 变量声明不计算时间；
* 比较，自增和结果返回各占一个时间单位；

一般法则：

* For循环：一次for循环的运行时间至多是该for循环内语句运行时间乘以迭代的次数，；
* 嵌套的For循环：需要从外之内分析，嵌套在内部的语句的总的运行时间为该语句的运行时间乘以该组所有for循环的大小的乘积。
* 顺序语句
* IF/ELSE语句

If(Condition){

S1

}ELSE{

S2

}

一个if/else判断语句的运行时间从不超过判断再加上S1和S2中运行时间长者的总运行时间

* 函数迭代的运行时间也是迭代运算出来的;

函数迭代计算的时间计算：

如：

int func(int n){

if(n<2){ // 1

return 1; // 2

}else{

return func(n-1) + func(n-2) // 3

}

}

第N次计算的时间近似于斐波那契数列每一项小于，可见他是一个指数级的时间的，通过转换为for循环，能大大减少运行的时间。程序之所以缓慢，是由于第三行中的func(n-2)已经被func(n-1)计算过一遍，这不符合递归的第四条法则（合成效益法则）

基本策略是从内至外展开

验证一个程序是否是，一个常用的技巧是对N的某个范围（通常是2大的倍数隔开）计算比值，其中是凭经验观察到的时间，如果是运行是按的理想近似，那么所算出的值收敛于一个正常数，如果算出来的值收敛于0，则说明估计过大，否则过小。

## 表，栈和队列

本章讨论三种数据结构

### 抽象数据类型

模块化的优点

1. 调试简单
2. 分工容易
3. 在例程中修改简单（所谓历程，我猜测是将模块集中在一起的主程序）

抽象数据类型（abstract data type）是一些操作的集合，或者说对诸如表，集合，图和他们的操作一起看成是抽象数据类型

如对于集合ADT，可以有诸如并（union）和交（intersection），测量大小（size）以及取余（complement）等操作

### 表ADT

1. 表的大小

形如A1，A2，A3，A4，An的表，大小为N，N=0则称为空表；

对于除空表以外的所有表，Ai+1后继Ai，或者说Ai前驱Ai+1；

1. 链表

为了避免插入和删除的线性开销，需要允许表可以不连续存储，否则表的部分或者全部需要整体移动，这是链表出现的原因。

链表是一系列不必在内存中相连的结构体，每个结构体均包含有表元素和指向后继元结构的指针（Next 指针，最后一个结构的Next指针指向NULL）。

其具体定义和结构如下：

#ifndef \_list\_h

struct node;

typedef struct node \*Ptrtonode;

typedef Ptrtonode List;

typedef Ptrtonode Position;

//操作函数的定义

IsEmpty(Position p，List L)

isLast(Position p，List L)

find(ElementType X，List L)

findPrevious(ElementType X，List L)

delete (ElementType X，List L)

insert(ElementType X，List L, Position P)

deleteList(List L)

#endif

struct node{

ElementType Element;

Position Next;

}

1. 说明：List 和Position都是链表指针的数据类型，而L和P分别是其具体的指针变量，代表每个元素的首地址，L一般指的是带表头的表的表头header，而P指的是任意位置的链表元素
2. 空表判断IsEmpty(Position p，List L)

int isEmpty(Position p，List L){

return L->Next==NULL;

}

1. 链表结束判断isLast(Position p，List L)

int isLast(Position p，List L){

return p->Next==NULL;

}

1. 查找find(ElementType X，List L)

int find(Element X，List L){

Position P=L->Next;

While(P->Next!=NULL && P->Element!=X){

P=P->Next;

}

return p;

}

1. 前驱查找findPrevious(ElementType X，List L)

int find(Element X，List L){

Position P=L->Next;

While(P->Next!=NULL && P->Next->Element!=X){

P=P->Next;

}

return p;

}

1. 删除delete(ElementType X，List L)

void delete (Element X，List L){

Position P=findPrevious(X, L);

If(p->Next!=NULL){

P->= P->Next ->Next;

free(P->Next);

}

}

1. 插入insert(ElementType X，List L, Position P)

void insert (Element X，List L){

Position insertP=(Position)malloc(sizeof(struct Node));

If(insert==NULL){

Printf(“Out of Space !!”);

}else{

insertP->Element=X;

insertP->Next= P->Next;

P->Next= insertP;

}

}

1. 删除表deleteList(List L)

void deleteList(List L){

Position Tem,;

Position P=L->Next;

while(P!=NULL){

Tem= P->Next;

free(P);

P=Tem;

}

}

1. 常见错误：

Memory access violation/segmentation violation，原因一般是出现初始化变量失败或者出现野指针的情况；

调用malloc的次数应该应该等于表的大小，若有表头则加1，少一点不行，多一点浪费空间；

对链表的某一个元素进行删除操作的时候，应该先用一个临时变量存储被删除变量的Next指针，然后释放被删除元素堆空间，最后把被删除变量前驱元的Next指针指向临时变量

1. 进制
2. 八进制

前面添加0，0后数字0~7，不能出现超过7的数字

1. 十进制
2. 十六进制

前面添加0x，0x后数字0~9，字母a~f/A~F

如：printf("Print [8] 17=%o\n",17); //八进制打印

printf("Print [10] 17=%d\n",17); //十进制打印

printf("Print [16] 17=%x\n",17); //十六进制打印，字母以小写输出；

printf("Print [16] 17=%X\n",17); //十六进制打印，字母以大写输出；

结果：Print [8] 17=21

Print [10] 17=17

Print [16] 17=11

1. 原码反码与补码

正数的原码反码和补码相同

为什么不用原码进行储存？

* 0有两种存储方式，符号位一个为0，一个为1；
* 正数和负数相加，结果不正确（计算机只会加不会减）

符号位：最高位；

反码（为了算补码）

* 正数的原码和反码是一样的；
* 求原码；
* 其他位取反（0为1，1为0）；

为什么不用反码进行储存？

* 0有两种存储方式，符号位一个为0，一个为1；

补码

正数的补码是他的原码，负数的补码是他绝对值的二进制形式按位取反后再加1，最高位为符号位，0代表正数，1代表负数。相同绝对值的正数和负数想加，最高位1丢弃，结果为0000 0000，此时0只有一种存储方式；

如果你想从补码得知它的源码的话，最开始的方式是先减1，然后求反码。由于正数和负数是可逆的，所以你可以重复求补码的操作，也就是说两次原码求补码的操作得到的是源码；

综上所示，站在计算机的角度，数据储存用的是补码；

1. 实型常量后面可以用F/f或者L/l来修饰，其中F/f表示单精度类型，L/l表示双精度类型;
2. 字符常量vs字符串常量

字符常量只能是一个字符，用单引号作为分界符，字符串可以有多个字符，用双引号作为分界符，末尾系统自动添加\0，作为字符串的结束标志，这也就是“H”的长度为2的原因。

1. 转义字符\

* \n换行回车
* \r光标位置回到句首
* \t制表位
* \a鸣铃

1. 字符型变量

实际上是以无符号整形的ASCII码值储存在内存单元中，使用命令man ascii可以查看ACSII表，小写字母比大写字母大32，小写转大写，小写字母-32；

1. 长整形vs基本整形

长整型是long 固定32位 sizeof(long)==4

基本整形是int 不固定，根据编译器位数来定

在TC这种16位编译器 sizeof(int)=2 也就是16位，像GCC，VS这样的32位编译器sizeof(int)==4也就是32位，跟long一样

如果是64位编译器，sizeof(int)==16

无论怎么样，int操作数度是最快的，因为系统相关

1. 无符号unsigned vs有符号signed的区别

无符号最高位不是符号位，而算作为数的一部分

无符号输出用%u

1. 数值越界

char a=127+2

* 如果是有符号输出，则

129转换为二进制码1000 0001，站在计算机的角度，这是补码

补码：1000 0001

反码：1111 1110

原码：1111 1111 最高位是符号位，得-127

* 如果是无符号输出

参数入栈时，不足int的，扩展为int后入栈

比如char(129)也即char(-127)扩展成int(-127)

也就是由 10000001 扩展成 11111111 11111111 11111111 10000001

然后你使用%u，也就是将 11111111 11111111 11111111 10000001 当成 unsigned int 来看，那它当然就是 4294967169 了

换句话说，要维持值不变，在高位补1还是补0取决于原来的符号位，这称为符号扩展（Sign Extension）。”

### 变量的存储类别

1. auto关键字

用于定义一个局部变量，意味着每次执行到定义该变量时，都会产生一个新的变量，并且对 其重新初始化；

auto int iInt = 1;

事实上，auto可以省略不写，因为局部变量的默认存储方式为自动的；

1. static关键字

静态变量

1. register变量

寄存器存储变量，对于一个具备可移植性的程序来说，register的作用并不大，不过不同于放在内存中，放在寄存器中效率更高，但是由于寄存器数量不多；

1. extern变量

外部存储变量，相当于public，在另外的c文件中可以调用本c文件中的变量

1. volatile变量

防止编译器优化代码，比如跑马灯的时候

int a;

a=1;

a=2;

a=3;

a=4;

跑马灯轮流亮，但是站在编译器的角度，最终结果是a=4，所以a=1~3的步骤会被编译器优化删除，此时跑马灯就跑不起来了，所以为例避免编译器作多余的优化，要加上此关键词。

### 类型转换

* 自动类型转换
* 强制类型转换

在变量前使用包含要转换类型的括号，如int j = (int)i ;

### 输入输出屏幕与打印格式

1. 输出：putchar() 输出字符；

不仅可以输出普通的字符，也可以输出转义字符，如putchar(‘\101’)等同于putchar(‘A’)，需要添加#include<stdio.h>

1. 输入：getchar() 输入字符；

要注意的是如果输出包含有换行\n，getchar会自动获取该换行符号

1. 输出：puts()，输出字符串，但是不能进行格式化操作；
2. 输入：gets()，输入字符串，但是不能进行格式化操作；
3. 输出：printf，输出字符串；
4. 输入：scanf(格式控制, 地址列表)；

//阻塞，等待用户输入内容，按回车结束，输入不需要加换行的符号\n

//编写程序时，在scanf函数的地址列表处，一定要使用变量的地址 &标识符，而不是变量的标识符，否则会报错。

* %hd 输入/输出short类型；
* %d 输入/输出int类型 可以使用%5d为输出，表示一共5个字符，默认是右对齐；%-5d中减号默认为右对齐；
* %l 输入/输出long类型
* %ll 输入/输出long long类型(8位)
* %hu 输入/输出unsigned short类型
* %u 输入/输出unsigned类型
* %lu 输入/输出unsigned long类型
* %llu 输入/输出unsigned long long类型
* %c 输入/输出字符类型
* %s 输入/输出字符串类型
* %p 输入/输出指针以16进制的形式
* %% 输入/输出一个%

### 运算符与表达式

1. 算术运算符

+ - \* / % ++ --

两个数相除，如果商想要获得小数，分子和分母必须是一个小数，否则结果只会取整；

1. 赋值运算符

= += -= \*= /= %/

1. 比较运算符

== != < > <= >=

1. 逻辑运算符

&& || ！

短路规则 ||，左边为真，右边不执行；

&&，左边为假，右边不执行；

优先级比赋值运算符要高；

1. 位运算符

**(1)&:按位操作符(“与”)**

将两个表达式的值按二进制位展开，对应的位（bit）按值进行”与”运算，结果保留在该位上。

eg1:17&18

对应的二进制就是00010010

&00010011

00010010

该位只要有一个值是0结果就是0，否则就是1.

如果两数位数不同，则较短数高位补零，再运算。

不具备短路功能。

**(2)按位或运算符(|)**

按位或运算将两个运算分量的对应位按位遵照以下规则进行计算：

0 | 0 = 0, 0 | 1 = 1, 1 | 0 = 1, 1 | 1 = 1

即只要有1个是1的位，结果为1，否则为0。

例如，023 | 035 结果为037。

按位或运算的典型用法是将一个位串信息的某几位置成1。如将要获得最右4为1，其他位与变量j的其他位相同，可用逻辑或运算017|j。若要把这结果赋给变量j，可写成：

j = 017|j

**(3)按位异或运算符(^)**

按位异或运算将两个运算分量的对应位按位遵照以下规则进行计算：

0 ^ 0 = 0, 0 ^ 1 = 1, 1 ^ 0 = 1, 1 ^ 1 = 0

即相应位的值相同的，结果为 0，不相同的结果为 1。

例如，013^035结果为026。

异或运算的意思是求两个运算分量相应位值是否相异，相异的为1，相同的为0。按位异或运算的典型用法是求一个位串信息的某几位信息的反。如欲求整型变量j的最右4位信息的反，用逻辑异或运算017^j，就能求得j最右4位的信息的反,即原来为1的位，结果是0,原来为0的位，结果是1。

**(4)按位取反运算符(~)**

按位取反运算是单目运算，用来求一个位串信息按位的反，即哪些为0的位，结果是1，而哪些为1的位，结果是0。例如, ~7的结果为0xfff8。

1. 移位运算

移位运算用来将整型或字符型数据作为二进位信息串作整体移动。有两个运算符：

<< (左移) 和 >> (右移)

移位运算是双目运算，有两个运算分量,左分量为移位数据对象，右分量的值为移位位数。移位运算将左运算分量视作由二进位组成的位串信息,对其作向左或向右移位，得到新的位串信息。

移位运算符的优先级低于算术运算符，高于关系运算符，它们的结合方向是自左至右。

(1)左移运算符(<<)

左移运算将一个位串信息向左移指定的位，右端空出的位用0补充。例如014<<2,结果为060,即48。

左移时，空出的右端用0补充，左端移出的位的信息就被丢弃。在二进制数运算中，在信息没有因移动而丢失的情况下，每左移1位相当于乘2。如4 << 2，结果为16。

(2)右移运算符(>>)

右移运算将一个位串信息向右移指定的位，右端移出的位的信息被丢弃。例如12>>2,结果为3。与左移相反，对于小整数，每右移1位，相当于除以2。在右移时，需要注意符号位问题。对无符号数据，右移时，左端空出的位用0补充。对于带符号的数据，如果移位前符号位为0(正数)，则左端也是用0补充；如果移位前符号位为1(负数)，则左端用0或用1补充，取决于计算机系统。对于负数右移，称用0 补充的系统为“逻辑右移”，用1补充的系统为“算术右移”。以下代码能说明读者上机的系统所采用的右移方法：

1. sizeof运算符
2. 运算符优先级

优先级1（从左往右）：【】数组下标 （） 圆括号 .成员变量 ->成员选择

优先级2（从右往左）：单目运算符 - ~ ++ -- \* &取地址 ！ （类型）强制类型转换 sizeof长度运算符

## 选择结构程序设计



### 选择结构程序设计

1. If
2. 三目运算符
3. switch

switch语句检验的表达式必须是一个整形表达式，意味着可以使用函数调用和运算符；case检验的值必须是整形常量；但是其效率没有if高，所以当检验的条件较少，如3~4个的时候建议用if，否则用switch。

switch(表达式){

case 情况1:

语句块1;

case 情况2:

语句块2;

case 情况3:

语句块3;

default:

语句块4;

}

## 循环控制

1. while(表达式)语句

其实就是相当于if语句，当条件满足的时候就进入循环，条件为假的时候就跳出循环。变量的初始化在循环之前

1. do…while

不管条件是否为真，循环体语句至少执行一次；

do

循环体语句

while(表达式)；

1. for(循环变量赋初值;循环条件;循环变量){}
2. 若省略“循环变量赋初值”，则必须在循环体之前添加初值，且其后的分号不能省略；
3. 若省略“循环条件”，则默认表达式2始终为真，则循环会无终止地进行下去；
4. 若省略“循环变量”，则程序员应该设法保证程序能结束；
5. 若三个都省略，for（；；），那就相当于while(1){循环体语句}程序将无休止地进行下去。

如：打印ASCII表

printf("Table ASCII\n");

char cChar;

for(cChar=65; cChar<91; cChar++){

printf("%c%c\t", cChar, cChar+32);

if(4 == cChar % 6){

printf("\n");

}

}

printf("\n");

注意：

使用gcc编译代码是报出

error: 'for' loop initial declarations are only allowed in C99 mode

note: use option -std=c99 or -std=gnu99 to compile your code

错误，这是因为在gcc中直接在for循环中初始化了增量：

for(int i=0; i<len; i++) {

}

这语法在gcc中是错误的，必须先先定义i变量：

int i;

for(i=0;i<len;i++){

}

这是因为gcc基于c89标准，换成C99标准就可以在for循环内定义i变量了：

gcc src.c -std=c99 -o src

1. 区别
2. for语句功能更强，凡是用while循环能完成的功能，用for循环都能实现；
3. while和do while，循环变量的初始化需要在循环前完成；
4. break可以跳出循环，continue语句结束本次循环；
5. goto语句

goto 标识符；

标识符：

语句

1. 随机数函数

在C语言中，生成一个随机数，一般利用两个函数，即void srand ( unsigned int seed ) 和 int rand ( void )。

srand() 函数用于生成一个随机数种子，种子的值等于参数seed，这个参数由我们指定一个数、式子或者函数值。而 rand() 函数根据这个随机数种子进行运算生成一个[ 0 , RAND\_MAX (int或 unsigned int最大值) ]范围内的随机数。

在一个程序中只需运行一次srand函数即可，若参数seed是一个固定值，则同一程序中多次调用rand函数生成的数是随机数，但多次调用这一程序时，由于初始的随机数种子相同，所以生成的随机数列完全相同，因此一般使用系统时间函数time(NULL)（在time.h文件中，返回从1970年1月1日0点到现在的秒数的值）作为参数seed，使每次调用程序时的种子或者随机数列也具有一定的随机性。

例如给int x 赋值为 [ 0 , 100 ) 的随机整数，先调用 srand(time(NULL))，再利用表达式x = rand()%100 即可。

如果在一个程序在中，循环调用 srand(time(NULL)) 和 rand() 函数，由于现在计算机运行快速，两次循环的时间差不到1秒，使每次的随机数种子相同，从而生成的随机数也相同。即使在循环中加延时函数，由于时差不大，生成的随机数也相差不大，而且使程序运行时间大大增加。因此，一定要在循环外使用srand函数，一个程序中使用一次即可。

如果对随机性要求较高，要避免采用取模操作%，这是为了避免在某些情况下，某些伪随机数生成器产生的数，低位不够随机的问题，此时应采用 x = (int) (100.0 \* rand() / (RAND\_MAX + 1.0)) ，生成一个 [ 0 , 100 ) 的随机浮点数，再将其转换为 int 型，从而得到一个 0 ~ 99 的随机整数。

## 数组

1. 数组的声明和定义

类型说明符 数组标识符[常量表达式]

如：int array[5];

1. ；
2. ；
3. ；