# 911数据结构复习（代码—收藏basic

来源：<http://blog.csdn.net/davidluo001/article/details/50823122>

## 时间复杂度和空间

1、一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)，若有某个辅助函数f(n),使得当n趋近于无穷大时，T（n)/f(n)的极限值为不等于零的**常数**，则称f(n)是T(n)的同数量级函数。记作T(n)=Ｏ(f(n)),称Ｏ(f(n)) 为算法的渐进时间复杂度，简称时间复杂度

 按数量级递增排列，常见的时间复杂度有**：常数阶O(1),对数阶O(log2n),线性阶O(n), 线性对数阶O(nlog2n),平方阶O(n2)，立方阶O(n3),...， k次方阶O(nk),指数阶O(2n)。**随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

平均时间复杂度是指所有可能的输入实例均以等概率出现的情况下，算法的期望运行时间。

2、一个程序的**空间复杂度**是指运行完一个程序所需内存的大小。利用程序的空间复杂度，可以对程序的运行所需要的内存多少有个预先估计。

## ****数据的存储结构:****

**1>. 顺序(Sequential)存储方法:**  
    顺序存储方法就是把每个结点的数据, 按照某种顺序存放在一段连续的存储单元中。  
          
**2>. 链式(Linked)存储方法:**  
   把每个结点的数据, 零散地存放在一些存储单元中。  
          
**3>. 索引(Index)存储方法:**  
      用结点的索引号i来确定结点的存储单元地址, 把每一个结点的数据按一定规律顺序或链式存放在存储单元中。  
              
**4>. 散列(Hash)存储方法:**  
把每个结点的数据通过一个预设的散列函数, 来决定该结点的存储单元。

## 串

##### **普通模式匹配算法：BF**

因为i - j表示这一趟的起始匹配位置，i - j + 1则意为从这一趟起始比较位置的下一个位置继续进行比较。

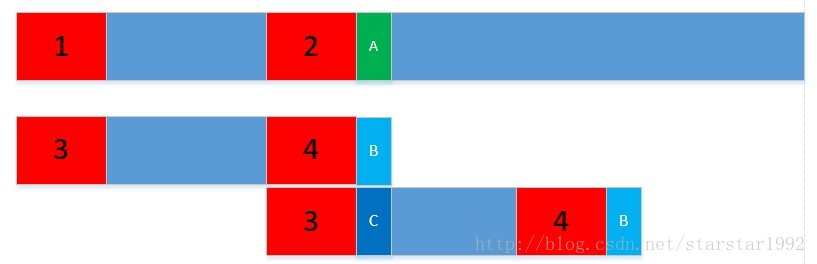
**[java]** [view plain](http://blog.csdn.net/fightlei/article/details/52712461" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/fightlei/article/details/52712461" \t "_blank" \o "copy)

1. /\*
2. \* BF匹配算法
3. \*/
4. public static int violentMatching(String s, String t) {
5. int i = 0;
6. int j = 0;
7. while (i < s.length() && j < t.length()) {
8. if (s.charAt(i) == t.charAt(j)) {
9. i++;
10. j++;
11. } else {
12. //i回溯到这一趟起始匹配位置的下一个位置
13. i = i - j + 1;
14. j = 0;
15. }
16. }
17. //当j==t.length()表示目标串S中的一个子串与模式串T完全匹配
18. if (j == t.length()) {
19. //返回这一趟起始匹配位置，即T在S中的位置
20. return i - j;
21. } else {
22. return -1;
23. }
24. }

##### 模式匹配算法Kmp **KMP算法：可以实现复杂度为O(m+n)**

特别容易理解：<http://blog.csdn.net/starstar1992/article/details/54913261>

**next数组的值，就是下次往前移动字符串ptr的移动距离。**next数组就是说一旦在某处不匹配时（下图绿色位置A和B），移动ptr字符串，使str的对应的最大后缀（红色2）和ptr对应的最大前缀（红色3）对齐，然后比较A和Ｃ。



这里就要引出KMP算法的关键所在next数组，next数组的作用就是当出现失配情况S[i] != T[j]时，next[j]就指示使用T中的以next[j]为下标的字符与S[i]进行比较（注意在KMP算法中，i是永远不会进行回溯的）。还需要说明的是当next[j] = -1时，就表示T中的任何字符都不与S[i]进行比较，下一轮比较从T[0]与S[i+1]开始进行。由此可见KMP算法在进行模式匹配之前需要先求出关于模式串T各个位置上的next函数值。即next[j]，j = 0,1,2,3,...n-1

**【求解next数组的算法实现】**

1. void cal\_next(char \*str, int \*next, int len)
2. {
3. next[0] = -1;//next[0]初始化为-1，-1表示不存在相同的最大前缀和最大后缀
4. int k = -1;//k初始化为-1
5. for (int q = 1; q <= len-1; q++)
6. {
7. while (k > -1 && str[k + 1] != str[q])//如果下一个不同，那么k就变成next[k]，注意next[k]是小于k的，无论k取任何值。
8. {
9. k = next[k];//往前回溯
10. }
11. if (str[k + 1] == str[q])//如果相同，k++
12. {
13. k = k + 1;
14. }
15. next[q] = k;//这个是把算的k的值（就是相同的最大前缀和最大后缀长）赋给next[q]
16. }
17. }

**[java]** [view plain](http://blog.csdn.net/fightlei/article/details/52712461" \t "_blank" \o "view plain) [copy](http://blog.csdn.net/fightlei/article/details/52712461" \t "_blank" \o "copy)

1. public static int[] getNext(String t) {
2. int[] next = new int[t.length()];
3. next[0] = -1;
4. int suffix = 0;  // 后缀
5. int prefix = -1;  // 前缀
6. while (suffix < t.length() - 1) {
7. //若前缀索引为-1或相等，则前缀后缀索引均+1
8. if (prefix == -1 || t.charAt(prefix) == t.charAt(suffix)) {
9. ++prefix;
10. ++suffix;
11. next[suffix] = prefix;  //1
12. } else {
13. prefix = next[prefix];  //2
14. }
15. }
16. return next;
17. }

**KMP模式匹配算法实现】**

1. int KMP(char \*str, int slen, char \*ptr, int plen)
2. {
3. int \*next = new int[plen];
4. cal\_next(ptr, next, plen);//计算next数组
5. int k = -1;
6. for (int i = 0; i < slen; i++)
7. {
8. while (k >-1&& ptr[k + 1] != str[i])//ptr和str不匹配，且k>-1（表示ptr和str有部分匹配）
9. k = next[k];//往前回溯
10. if (ptr[k + 1] == str[i])
11. k = k + 1;
12. if (k == plen-1)//说明k移动到ptr的最末端
13. {
14. //cout << "在位置" << i-plen+1<< endl;
15. //k = -1;//重新初始化，寻找下一个
16. //i = i - plen + 2;//i定位到找到位置处的下一个位置（这里默认存在两个匹配字符串可以部分重叠）
17. return i-plen+1;//返回相应的位置
18. }
19. }
20. return -1;
21. }

## 循环队列

###### 3、 循环队列各个参数的含义

2个参数不同场合不同的含义?    建议初学者先记住，然后慢慢体会

1）队列初始化

front和rear的值都是零,初始化时队列就是空的。

2）队列非空

front代表队列的第一个元素

rear代表了最后一个有效元素的下一个元素

3）队列空

front和rear的值相等，但是不一定是零

###### 4、   循环队列入队伪算法讲解

需要判断r是否指向数组最后一个元素。两步完成：

1）将值存入r所代表的位置 pQ->PBase[pQ->rear]=val;

2）将r后移，正确写法是rear = (rear+1)%数组长度pQ->rear=(pQ->rear+1)%6;

###### 5、 循环队列出队伪算法讲解

 front = (front+1)% 数组长度：pQ->front=(pQ->front+1)%6;

###### 6、 如何判断循环队列是否为空

   如果front与rear的值相等，则队列一定为空：(pQ->front==pQ->rear)

###### 7、 如何判断循环队列是否已满。

所以为了判断循环队列是否已满,有一下两种方式:

1、多增加一个表标识的参数

2、少用一个队列中的元素（才一个，不影响的）如果r和f紧挨着(r的下一个位置是f)，则队列已满

用C语言描述： if((pQ->rear+1)% 数组长度==pQ->front)

## 栈

### 0栈（第三章）

### 基础

基本的抽象数据类型（ADT）是编写C程序必要的过程，这类ADT有链表、堆栈、队列和树

堆栈（stack）的显著特点是**后进先出**，其实现的方法有三种可选方案：

### （2）栈的实现方法

1静态**数组、**2**动态分配**的数组、3动态分配的**链式**结构。

**静态数组：**特点是要求结构的长度固定，而且长度在编译时候就得确定。其优点是结构简单，实现起来方便而不容易出错。而缺点就是不够灵活以及固定长度不容易控制，适用于知道明确长度的场合。

**动态数组：**特点是长度可以在运行时候才确定以及可以更改原来数组的长度。优点是灵活，缺点是由此会增加程序的复杂性。

**链式结构：**特点是无长度上线，需要的时候再申请分配内存空间，可最大程度上实现灵活性。缺点是链式结构的链接字段需要消耗一定的内存，在链式结构中访问一个特定元素的效率不如数组。

##### 1\*\*定义节点的结构体

\*/

typedef struct Node{

int data;//数据域

struct Node \* PNext;//指针域

} Node,\*PNext;

/\*\*

##### 2\*\*定义栈的结构体

\*/

typedef struct Stack {

PNext top;

PNext bottom;

}Stack,\* PStack;

##### 3 \*\*初始化栈

\*/

void init(PStack PStack )

{

//建立一个不存任何有限元素的头结点，作为栈底

PStack->bottom=malloc(sizeof(Node));

PStack->top=PStack->bottom;

PStack->top->PNext=NULL;

}

/\*\*

##### 4\*遍历栈

\*\*/

void traverse(PStack Ps )

{

if(Ps->bottom==Ps->top)

{

printf("栈为空\n");

return ;

}

PNext pt=Ps->top;

while(pt!=Ps->bottom)//不能把pt换成ps->top这样就修改了链表。尴尬。。

{

printf("%d ",pt->data);

pt=pt ->PNext;

}

printf("\n");

return ;

}

/\*\*

##### 5 \*\*入栈

\*/

void push(PStack Pstack,int val)

{

PNext Pnew=malloc(sizeof(Node));//生成一个新节点

Pnew->data=val;

Pnew->PNext=Pstack->top;

Pstack->top=Pnew;

}

/\*\*

##### 6\*\*出栈

\*/

void pop(PStack ps )

{

if(ps->top==ps->bottom)

{

printf("栈为空，无法完成出栈操作\n");

return;

}

PNext temp=ps->top;//引入辅助变量，用于释放内存

ps->top=ps->top->PNext;

free(temp);

temp=NULL;

}

/\*\*

##### 7 \*\*清空栈

\*/

void clear(PStack ps)

{

while(ps->top!=ps->bottom)

{

PNext temp=ps->top;

ps->top=ps->top->PNext;

free(temp);

}

}

#### 链表

* 链表是一种由节点（Node）组成的线性数据集合，每个节点通过指针指向下一个节点。它是一种由节点组成，并能用于表示序列的数据结构。

##### 类型:

* **单向链表**：每个节点仅指向下一个节点，最后一个节点指向空（null）。
* **双向链表**：每个节点有两个指针p，n。p指向前一个节点，n指向下一个节点；最后一个节点指向空。
* **循环链表**：每个节点指向下一个节点，最后一个节点指向第一个节点。

##### \*\*链表节点的定义

\*/

typedef struct Node{

int data;//数据域

struct Node \* PNext;//指针域，存放下一个节点的地址

} Node ,\* PNode ;

/\*\*

##### \*\*创建链表

\*/

PNode create\_list()

{

int len,i;

printf("请输入链表的长度：len=\n");

scanf("%d",&len);

PNode PHead=malloc(sizeof(Node));

PHead->PNext=NULL;

PNode PTail=PHead;//PTail是永远指向尾节点的指针

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

{

int val;

printf("请输入第 %d 个元素的值：", i+1);

scanf("%d",&val);

PNode PNew=malloc(sizeof(Node));

PNew->data=val;

PNew->PNext=NULL;

PTail->PNext=PNew;

PTail=PNew;

}

return PHead;

}

/\*\*

##### \*\*对链表进行遍历

\*/

void traverse(PNode pHead)

{

PNode p=pHead->PNext;

while(p!=NULL)

{

printf("%d ",p->data);

p=p->PNext;

}

printf("\n");

}

/\*\*

##### \*判断链表是否为空

\*/

bool isempty(PNode pHead)

{

if(NULL==pHead->PNext)

{

return true;

}else{

return false;

}

}

/\*\*

\*\*获取链表的长度

\*/

int list\_num (PNode pHead)

{

int num=0;

PNode p=pHead->PNext;

while(p!=NULL)

{

num++;

p=p->PNext;

}

return num;

}

/\*\*

##### \*向链表中插入元素

\*/

bool insert\_list(PNode pHead,int val ,int pos){

//需要找到第pos个位置，并且需要判断这个位置pos是否合法

//i是p所指节点的位置，所以从一开始，为什么要pos-1呢，因为用的是while 当i=pos-1时跳出循环

int i=0;

PNode p=pHead;

while(NULL!=p&&i<pos-1)

{

i++;

p=p->PNext;

}

//如果插入位置过大，那么P=NULL,

//如果插入的位置是0或者负数，那么i>pos-1

if(i>pos-1||NULL==p)

{

printf("插入位置不合法\n");

return false;

}

PNode PNew=malloc(sizeof(PNode));

PNew->data=val;

PNode temp=p->PNext;

p->PNext=PNew;

PNew->PNext=temp;

return true;

}

/\*\*

\*\*在链表中删除节点

\*/

delete (PNode PHead,int pos , int \* pval)

{

int i=0;

PNode p=PHead;

//我们要删除p后面的节点，所以p不能指向最后一个节点 p->next!=NULL

while(p->PNext!=NULL&&i<pos-1){

p=p->PNext;

i++;

}

if(i>pos-1||p->PNext==NULL)

{

printf("删除位置不合法\n");

return false;

}

PNode temp=p->PNext;

p->PNext=temp->PNext;

free(temp);

}

#### 数组

struct Arr{

int len;//数组能存取的最大元素个数

int cnu;//数组中当前元素个数

int \* pBase;//存储指向数组的指针

};

/\*\*

##### \*初始化数组

pArray->pBase=(int \*)malloc(sizeof(int)\*len);//分配4\*len字节长度的内存

##### \*判断数组是否为空，

bool isempty(struct Arr \* pArray){

if(0==pArray->cnu)

##### \*\*判断数组是否满了

\*/

bool isfull(struct Arr \* pArray)

{

if(pArray->len==pArray->cnu)

##### \*\*向数组追加元素

\*/

bool append(struct Arr \* pArray,int val){

if(isfull(pArray))

{

printf("数组已经满了！\n");

return false;

}else{

pArray->pBase[pArray->cnu]=val;

pArray->cnu++;

}

}

/\*\*

##### \*\*向数组中插入元素

,pos为数组中第几个位置，pos=3就是向a[2]插入元素

\*/

bool insert(struct Arr \* pArray,int pos,int val)

{

if(pos<1||pos>pArray->len+1)//插入的位置不能小于1，同时不能比最后一个元素大二

{

printf("插入的位置输入的不合法\n");

return false;

}

if(isfull(pArray))

{

printf("数组已经满了,插入失败！\n");

return false;

}

int i;

//循环将pos位置开始的数组后移

for(i=pArray->cnu;i>=pos;i--)

//移动范围是从第pos个到底cnu个

{

pArray->pBase[i]=pArray->pBase[i-1];

/\*\*

若以i表示要移动元素的位置，从一开始的。右边都是i-1,若左移，左边是i-2,右移，左边是i

\*/

}

pArray->pBase[pos-1]=val;

pArray->cnu++;

pArray->len++;

return true;

}

/\*\*

##### \*\*删除数组中的第pos个元素，同时返回删除的元素的值

\*/

bool delete(struct Arr \* pArray,int pos,int \* val)

{

if(pos<1||pos>pArray->cnu)

{

printf("删除失败，位置不合法\n");

return false;

}

int i;

\*val=pArray->pBase[pos-1];

for(i=pos+1;i<=pArray->cnu;i++)

{

//移动单位是从第pos+1个到cnu

pArray->pBase[i-2]=pArray->pBase[i-1];

}

pArray->cnu--;

return true;

}

/\*\*

##### \*\*数组倒置

\*/

bool inverse(struct Arr \* pArray)

{

if(isempty(pArray))

{

printf("倒置失败，因数组为空");

return false;

}

int i=0;

int j=pArray->cnu-1;

int temp;

while(i<j)

{

temp=pArray->pBase[i];

pArray->pBase[i]= pArray->pBase[j];

pArray->pBase[j]=temp;

i++;

j--;

}

return true;

## 堆、

**堆**

* 堆是一种基于树的满足某些特性的数据结构：整个堆中的所有父子节点的键值都满足相同的排序条件。堆分为最大堆和最小堆。在最大堆中，父节点的键值永远大于等于所有子节点的键值，根节点的键值是最大的。最小堆中，父节点的键值永远小于等于所有子节点的键值，根节点的键值是最小的。

## 树和二叉树

* 二叉树定义，性质，存储结构
* 遍历二叉树和线索二叉树

##### **二叉树定义**

* 二叉树是一个树形数据结构，每个节点最多可以有两个子节点，称为左子节点和右子节点。
* **满二叉树（Full Tree）**：二叉树中的每个节点有 0 或者 2 个子节点。
* **完美二叉树（Perfect Binary）**：二叉树中的每个节点有两个子节点，并且所有的叶子节点的深度是一样的。
* **完全二叉树**：二叉树中除最后一层外其他各层的节点数均达到最大值，最后一层的节点都连续集中在最左边。

##### **二叉查找树**

* 二叉查找树（BST）是一种二叉树。其任何节点的值都大于等于左子树中的值，小于等于右子树中的值

**字典树**

* 字典树，又称为基数树或前缀树，是一种用于存储键值为字符串的动态集合或关联数组的查找树。树中的节点并不直接存储关联键值，而是该节点在树中的位置决定了其关联键值。一个节点的所有子节点都有相同的前缀，根节点则是空字符串

**树状数组**

* 树状数组，又称为二进制索引树（Binary Indexed Tree，BIT），其概念上是树，但以数组实现。数组中的下标代表树中的节点，每个节点的父节点或子节点的下标可以通过位运算获得。数组中的每个元素都包含了预计算的区间值之和，在整个树更新的过程中，这些计算的值也同样会被更新。

**线段树**

* 线段树是用于存储区间和线段的树形数据结构。它允许查找一个节点在若干条线段中出现的次数

### 1二叉树（第七章）

### （1）问我二叉树的定义是什么？

**是节点的有限集合，**这个集合或者为空集，或者有一个根节点以及两颗不相交的、分别称为**这根的左字树和右子树的二叉树**组成

### （2）定义

一个节点拥有的孩子书为该节点的度，节点度的最大值为树的度，叶子节点，终端节点

节点的最大层数为深度或者高度

1. **几种特殊的二叉树：**

满二叉树：都有左右子树，子节点都在同一层

完全二叉树：只有最下面两层节点读书可以小于2，最下面一层节点都集中在左边

平衡二叉树——平衡二叉树又被称为AVL树（区别于AVL算法），它是一棵二叉排序树，且具有以下性质：**它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，**并且左右两个子树都是一棵平衡二叉树

### 二叉树的遍历（递归遍历）

则对一棵二叉树的遍历有三种情况：DLR（称为先根次序遍历），LDR（称为中根次序遍历），LRD （称为后根次序遍历）。

先序，中序，后序

首先访问根，再先序遍历左（右）子树，最后先序遍历右（左）子树，C语言代码如下

### （5）线索二叉树

线索二叉树(保留遍历时结点在任一序列的前驱和后继的信息)：若结点有左子树，则其lchild域指示其左孩子，否则令lchild域指示其前驱；

还需在结点结构中增加两个标志域LTag和RTag。LTag=0时，lchild域指示结点的左孩子，LTag=1时，lchild域指示结点的前驱；

以这种结点结构构成的二叉线索[链表](https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91/_blank)，链表作为二叉树的[存储结构](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E7%BB%93%E6%9E%84" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91/_blank)，叫做其中指向结点前驱和后继的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91/_blank)叫做线索，加上线索的二叉树称为线索二叉树。对二叉树以某种次序遍历使其变为线索二叉树的过程叫做线索化。

**非递归实现：借助堆栈转换成循环结构（p159）**

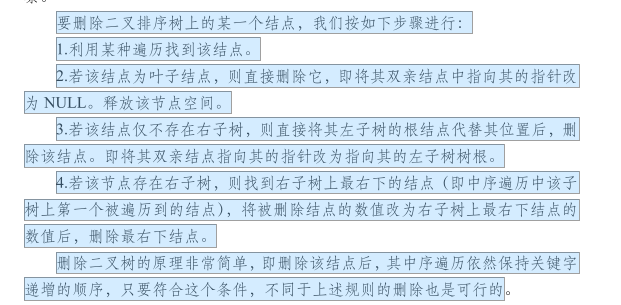
### （6）二叉树的应用

第一种应用就是用于**表达式树**，是编译器设计的核心数据结构，而实现编译器的技术又可用在计算器的实现上（你应想到有什么用了吧）。

第二种应用就是你所说的huffman编码，实现简单而相当有效的数据压缩算法。

第三种应用更多的体现在二叉查找树和优先队列上，如大型数据库快速搜索（B-树）。。。

（7）删除节点



看代码（王道论坛）

### 哈夫曼树

给定n个权值作为n个[叶子](https://baike.baidu.com/item/%E5%8F%B6%E5%AD%90/11999973" \t "https://baike.baidu.com/item/%E5%93%88%E5%A4%AB%E6%9B%BC%E6%A0%91/_blank)结点，构造一棵二叉树**，若带权路径长度达到最小，称这样的二叉树为最优二叉树**，**也称为哈夫曼树**(Huffman Tree)。哈夫曼树是带权路径长度最短的树，权值较大的结点离根较

(1) 将w1、w2、…，wn看成是有n 棵树的森林(每棵树仅有一个结点)；

(2) 在森林中选出两个根结点的权值最小的树合并，作为一棵新树的左、右子树，且新树的根结点权值为其左、右子树根结点权值之和；

(3)从森林中删除选取的两棵树，并将新树加入森林；

(4)重复(2)、(3)步，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

定义：

**1、路径和路径长度**

在一棵树中，从一个结点往下可以达到的孩子或孙子结点之间的通路，称为路径。通路中分支的数目称为路径长度。若规定根结点的层数为1，则从根结点到第L层结点的路径长L-1

**2、结点的权及带权路径长度**

若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积。

**3、树的带权路径长度**

树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为WP

### 应用：哈弗曼编码

使得编码总长最短

### 树和森林

**树的存储结构：**复制镇表示大，，孩子连表表示法，孩子兄弟表示法

**书和二叉树的相互转换：**

树-----二叉树:加线，抹线，旋转

二叉树--树：

森林-二叉树

树的遍历：先序后序中序

**决策树**

## 查找图

###### **图定义**

* 图是G =（V，E）的有序对，其包括顶点或节点的集合 V 以及边或弧的集合E，其中E包括了两个来自V的元素（即边与两个顶点相关联 ，并且该关联为这两个顶点的无序对）

#### 图（未复习。再看）

* 图的存储
* 图的遍历
* 图的连通性
* 有向无环图及其应用
* 最短路径

### 搜索：BFS和DFS，递归

#### BFS核心代码

/\*\*

 \* 广度优先搜索

 \* @param Vs 起点

 \* @param Vd 终点

 \*/

bool BFS(Node& Vs, Node& Vd){

**queue<node> Q;**

    Node Vn, Vw;

    int i;

    //用于标记颜色当visit[i][j]==true时，说明节点访问过，也就是黑色

    bool visit[MAXL][MAXL];

    //四个方向

    int dir[][2] = {    {0, 1}, {1, 0},     {0, -1}, {-1, 0}     };

    //初始状态将起点放进队列Q

    Q.push(Vs);

    visit[Vs.x][Vs.y] = true;//设置节点已经访问过了！

    while (!Q.empty()){//队列不为空，继续搜索！

      //取出队列的头Vn

        Vn = Q.front();

        Q.pop();

        for(i = 0; i < 4; ++i){

            Vw = Node(Vn.x+dir[i][0], Vn.y+dir[i][1]);//计算相邻节点

            if (Vw == Vd){//找到终点了！

                //把路径记录，这里没给出解法

                return true;//返回

            }

            if (isValid(Vw) && !visit[Vw.x][Vw.y]){

                //Vw是一个合法的节点并且为白色节点

                Q.push(Vw);//加入队列Q

                visit[Vw.x][Vw.y] = true;//设置节点颜色

            }

        }

    }

    return false;//无解

}

##### 初始化

int main(){

int s,n,m;

while(scanf("%d%d%d",&s,&n,&m)!=EOF){

if(s==0) break;//若s=0则n,m为0则退出

if(s%2==1) {

puts("NO");

continue;

}

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

for(int j=0;j<n;j++){

for(int k=0;k<s;k++){

mark[i][j][k]=false;

}

}

}//初始化状态

N tmp;

tmp.a=s;

tmp.b=0;

tmp.c=0 ;

tmp.t=0//初始时状态

while（Q.empty()==false) Q.pop();//清空队列中的状态

Q.push(tmp) ;

mark[s][0][0]=true;//初始状态

int rec=BFS(s,n,m);//搜素

if(rec==-1) pus("no");

else printf("%d\n",rec);//否则输出答案

}

return 0；

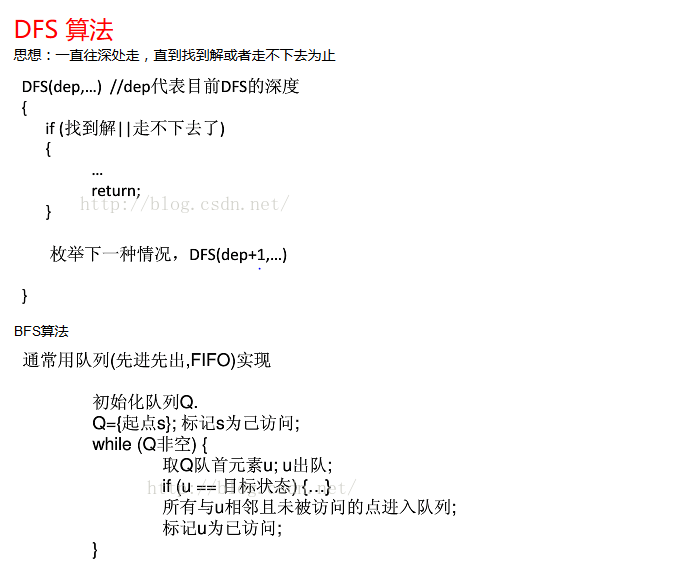
}

##### 经典问题走迷宫

1. int BFS(int a ,int b,int c){
2. //广度优先搜索，返回最小耗费时间
3. while(Q.empty()==false){
4. N now=Q.front();//得到队头的状态
5. Q.pop();//从队列中弹出队头状态
6. for(int i=0;i<6;i++){
7. int nx=now.x+go[i][0];
8. int ny=now.y+go[i][1];
9. int nz=now.z+go[i][2];//计算新坐标
10. if(nx<0||nx>=a||ny<0||ny>=b|nz<0||nz>=c) continue;
11. if(maze[nx][ny][nz]==1) continue;
12. if(mark[nx][ny][nz]==true) continue;
13. N tmp;
14. tmp.x=nx;
15. tmp.y=ny;
16. tmp.z=nz;
17. tmp.t=now.t+1;
18. Q.push(tmp);
19. mark[nx][ny][nz]=true;//标记该坐标
20. if(nx==a-1&ny==b-1&&nz==c-1) return tmp.t;// 该坐标就是终点。可以直接返回其耗时
22. }
23. }
24. return -1;
25. }

#### DFS

1. #include<stdio.h>
2. char maze[8][8];
3. int n,m,t;
4. bool success;
5. int go[][2]={1,0,-1,0,0,1,0,-1};
6. void DFS(int x,int y,int time){
7. for(int i=0;i<4;i++){
8. int nx=x+go[i][0];
9. int ny=y+go[i][1];//计算其坐标
10. if(nx<1||nx>n||ny<1||ny>m) continue;
11. if(maze[nx][ny]=='X') continue;
12. if(maze[nx][ny]='D') {
13. if(time+1==t){
14. success=true;
15. return;
16. }
17. else continue;
18. }
19. maze[nx][ny]='.';
20. DFS(nx,ny,time+1);//递归扩展
21. if(success) return;
22. }
23. }



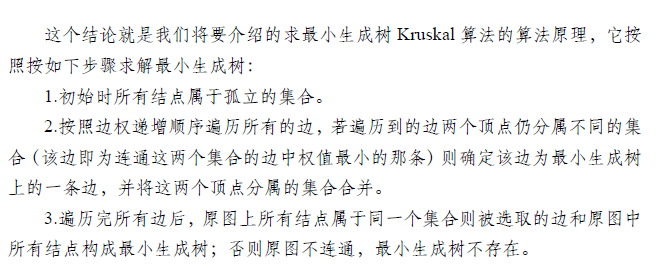
#### 递归DFS

### 4最小生成树（prim算法，kruskal算法）

无相连通图的生成树中权值总和最小的生成树，最小代价生成树

prim算法：选一个顶点u，然后选择最小权值的边uv，不断重复，直到顶点全都进来

Kruskal：按照权值大小依次选取，不形成回路为准则



### c的基础

### 5最短路径(dijkastra算法-狄克斯特拉)

选择路径最短的顶点加入到集合S，找到和已经存在的点距离最短的

### 6拓扑排序（还要看）

选择顶点，删除尾部，直到找不到入度为0的 顶点

**7Bellman-Ford算法**

* Bellman-Ford 是一种在带权图中查找单一源点到其他节点最短路径的算法

**8Floyd-Warshall 算法**

* Floyd-Warshall 算法是一种在无环带权图中寻找任意节点间最短路径的算法。

## 4动态规划的主要思想，具体的应用，

动态规划思想是[算法](http://lib.csdn.net/base/datastructure" \t "http://blog.csdn.net/zx64881926/article/details/_blank" \o "算法与数据结构知识库)设计中很重要的一个思想，所谓动态规划就是“边走边看”**，前面的知道了，后面的根据前面的也就可以推出来了。**和分治算法相似又不同，相同的是都需要去寻找最优子结构，重复子问题，边界条件。不同的是动态规划算法存储前面算得的每一个结果**，后面的结果由前面的结果推倒得出**

**特点：**

1.最优子结构：如果问题的最优解包含的子问题的解也是最优的，就称该问题具

3.状态转换方程：你找到的每一个子问题都可以认为是一个状态下的规划问题，**而状态与状态之间你需要找到关联方程，**这样我们才能**从最简单的状态循环求解出所需要求解的较复杂的状态**

4.边界条件：即子问题的初值，**你必须给出规模较小的一些子问题的初值来开启循环，否则无法循环求解**

**常见问题**——**串与序列（串必须连续，序列可以不连续）**

### 1最长公共子序列（LCS）：

状态：设数组dp[m][n]来表示以str1[m]结尾的str1和以str2[n]结尾的str2的最长公共子字符串长度，最长的长度为dp[m][n]中最大的元素值

状态转换方程：**if**(str1[m]==str2[n])   dp[m][n]=dp[m-1][n-1]+1;

**if**(str1[m]!=str2[n])    dp[m][n]=0;

初始条件：dp[i][0]=0  dp[0][j]=0   0<=i<=m      0<=j<=n

满足 最优子结构 以及 无后效性

### 3.最长递增子序列（LIS）：

LIS问题可以使用排序+LCS问题完成，还可以用动态规划，这些时间复杂度为n2

状态：设数组dp[m]为以a[m]结尾的序列中的最长递增序列的长度，最终的LIS长度为dp数组中的最大值

状态转换方程：dp[i]:     **for**(int j=1;j<i;j++){ **if**(a[j-1]<=a[i-1]&&dp[j]>max)   max=dp[j]; } dp[i]=max+1;

初始条件：dp[0]=0;

满足 最优子结构 以及 无后效性

**4.编辑距离问题：**

状态：设edit[m][n]为长度为m的str1和长度为n的str2的编辑距离

状态转换方程：**if**(j>i) edit[i][j]=edit[i][i]+j-i; **else if**(j<i) edit[i][j]=edit[j][j]+i-j; **else**{ **if**(str1.charAt(i-1)==str2.charAt(j-1)) edit[i][j]=edit[i-1][j-1]; **else**edit[i][j]=edit[i-1][j-1]+1;}

初始条件：**for**(int i=0;i<=m;i++) edit[i][0]=i;          **for**(int j=0;j<=n;j++)   edit[0][j]=j;

## 8大排序

#### 1总结

<http://blog.csdn.net/hguisu/article/details/7776068>

<http://www.jianshu.com/p/7d037c332a9d>





当n较大，则应采用时间复杂度为O(nlog2n)的排序方法：快速排序、堆排序或归并排序序。

   快速排序：是目前基于比较的内部排序中被认为是最好的方法，当待排序的关键字是随机分布时，快速排序的平均时间最短；

#### 1插入排序

##### 基本思想:

**将一个记录插入到已排序好的有序表中，从而得到一个新，记录数增1的有序表。即：先将序列的第1个记录看成是一个有序的子序列，然后从第2个记录逐个进行插入，直至整个序列有序为止。**

要点：设立哨兵，作为临时存储和判断数组边界之用。

如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，**所以插入排序是稳定的**

##### 代码算法的实现：

void InsertSort(int a[], int n)

{

for(int i= 1; i<n; i++){

if(a[i] < a[i-1]){ //若第i个元素大于i-1元素，直接插入。小于的话，移动有序表后插入

int j= i-1;

int x = a[i]; //复制为哨兵，即存储待排序元素

a[i] = a[i-1]; //先后移一个元素

while(x < a[j]){ //查找在有序表的插入位置

a[j+1] = a[j];

j--; //元素后移

}

a[j+1] = x; //插入到正确位置

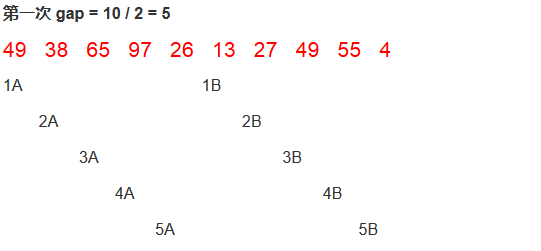
##### 效率：

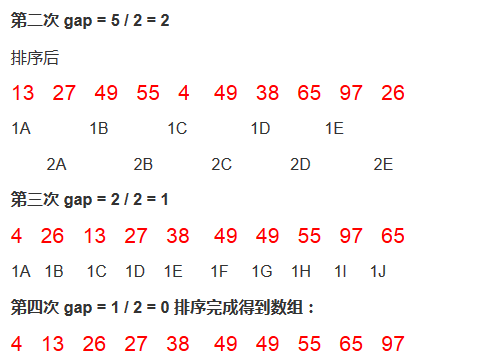
时间复杂度：O（n^2）.

其他的插入排序有二分插入排序，2-路插入排序。

#### 2. 插入排序—希尔排序（Shell`s Sort）

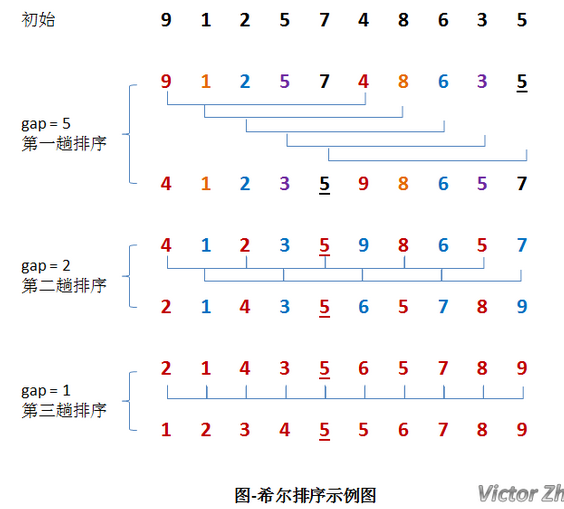
1介绍

以n=10的一个数组49, 38, 65, 97, 26, 13, 27, 49, 55, 4为例              



*1A,1B，2A,2B等为分组标记，数字相同的表示在同一组，大写字母表示是该组的第几个元素， 每次对同一组的数据进行直接插入排序。即分成了五组(49, 13) (38, 27) (65, 49)  (97, 55)  (26, 4)这样每组排序后就变成了(13, 49)  (27, 38)  (49, 65)  (55, 97)  (4, 26)，下同。*

2介绍2



2代码

void shellsort1(int a[], int n)

{

    int i, j, gap;

    for (gap = n / 2; gap > 0; gap /= 2) //步长

        for (i = 0; i < gap; i++)        //直接插入排序

        {

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

                if (a[j] < a[j - gap])

                {

                    int temp = a[j];

                    int k = j - gap;

                    while (k >= 0 && a[k] > temp)

                    {

                        a[k + gap] = a[k];

                        k -= gap;

                    }

                    a[k + gap] = temp;

#### 3简单选择排序

##### 1、基本思想：

在要排序的一组数中，选出最小（或者最大）的一个数与第1个位置的数交换；然后在剩下的数当中再找最小（或者最大）的与第2个位置的数交换，依次类推，直到第n-1个元素（倒数第二个数）和第n个元素（最后一个数）比较为止。

##### 2代码

　for(int i=0; i<v.size(); i++){  
 int min = v[i];   
 int temp;  
 int index = i;  
 for(int j=i+1;j<v.size();j++){  
 if(v[j] < min){   
 min = v[j];   
 index = j;  
 }   
 }   
   
 temp = v[i];   
 v[i] = min;  
 v[index]= temp;  
 }

##### 3简单二元选择排序

void SelectSort(int r[],int n) {

    int i ,j , min ,max, tmp;

    for (i=1 ;i <= n/2;i++) {

        // 做不超过n/2趟选择排序

        min = i; max = i ; //分别记录最大和最小关键字记录位置

        for (j= i+1; j<= n-i; j++) {

            if (r[j] > r[max]) {

                max = j ; continue ;

            }

            if (r[j]< r[min]) {

                min = j ;

            }

      }

      //该交换操作还可分情况讨论以提高效率

      tmp = r[i-1]; r[i-1] = r[min]; r[min] = tmp;

      tmp = r[n-i]; r[n-i] = r[max]; r[max] = tmp;

#### 4. 选择排序—堆排序（Heap Sort）--复杂

堆排序是一种树形选择排序，是对直接选择排序的有效改进。

初始时把要排序的n个数的序列看作是一棵**顺序存储的二叉树（一维数组存储二叉树）**，调整它们的存储序，使之成为一个堆，将堆顶元素输出，得到n 个元素中最小(或最大)的元素，这时堆的根节点的数最小（或者最大）。然后对前面(n-1)个元素重新调整使之成为堆，输出堆顶元素，得到n 个元素中次小(或次大)的元素。依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有n个节点的有序序列。称这个过程为**堆排序**。

因此，实现堆排序需解决两个问题：  
1. 如何将n 个待排序的数建成堆；  
2. 输出堆顶元素后，怎样调整剩余n-1 个元素，使其成为一个新堆。

#### 5. 交换排序—冒泡排序（Bubble Sort）

**1基本思想：**

在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

 void bubbleSort(int a[], int n){

     for(int i =0 ; i< n-1; ++i) {

         for(int j = 0; j < n-i-1; ++j) {

             if(a[j] > a[j+1])

    int tmp = a[j] ; a[j] = a[j+1] ;  a[j+1] = tmp;

#### 6. 交换排序—快速排序（Quick Sort）

##### 1基本思想：

1）选择一个基准元素,通常选择第一个元素或者最后一个元素,

2）通过一趟排序讲待排序的记录分割成独立的两部分，其中一部分记录的元素值均比基准元素值小。另一部分记录的 元素值比基准值大。

3）此时基准元素在其排好序后的正确位置

4）然后分别对这两部分记录用同样的方法继续进行排序，直到整个序列有序

1通过一趟排序将要排序的数据分割成独立的两部分，其中**一部分的所有数据都比另外一部分的所有数据都要小**

对这两部分数据分别进行**快速排序，整个排序过程可以递归进行**，以此达到整个数据变成有序序列。

复杂度分析：

  \* 1.平均时间复杂度：O(nlogn)

  \* 2.空间复杂度：O(logn)(最值元素存储空间)

快排的前身是归并，而正是因为归并存在不可忽视的缺点，才产生了快排。归并的最大问题是需要额外的存储空间，并且由于合并过程不确定，致使每个元素在序列中的最终位置上不可预知的。针对这一点，快速排序提出了新的思路：把更多的时间用在“分”上，而把较少的时间用在“治”上。从而解决了额外存储空间的问题，并提升了算法效率。

快速排序也并不是最好的，它的”快“是建立在综合考虑的基础上，具体情况则不一定

快速排序也不是万能的，例如当给定序列规模很小时，选择排序就要比快排好很多

##### 2代码

int partition(int a[], int low, int high)

{

int privotKey = a[low]; //基准元素

while(low < high){ //从表的两端交替地向中间扫描

while(low < high && a[high] >= privotKey) --high; //从high 所指位置向前搜索，至多到low+1 位置。将比基准元素小的交换到低端

swap(&a[low], &a[high]);

while(low < high && a[low] <= privotKey ) ++low;

swap(&a[low], &a[high]);

}

print(a,10);

return low;

}

void quickSort(int a[], int low, int high){

if(low < high){

int privotLoc = partition(a, low, high); //将表一分为二

quickSort(a, low, privotLoc -1); //递归对低子表递归排序

quickSort(a, privotLoc + 1, high); //递归对高子表递归排序

}

}

#### 7. 归并排序（Merge Sort）

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

代码

void mergearray(int a[], int first, int mid, int last, int temp[])

{

int i = first, j = mid + 1;

int m = mid, n = last;

int k = 0;

while (i <= m && j <= n)

{

if (a[i] <= a[j])

temp[k++] = a[i++];

else

temp[k++] = a[j++];

}

while (i <= m)

temp[k++] = a[i++];

while (j <= n)

temp[k++] = a[j++];

for (i = 0; i < k; i++)

a[first + i] = temp[i];

}

void mergesort(int a[], int first, int last, int temp[])

{

if (first < last)

{

int mid = (first + last) / 2;

mergesort(a, first, mid, temp); //左边有序

mergesort(a, mid + 1, last, temp); //右边有序

mergearray(a, first, mid, last, temp); //再将二个有序数列合并

}

}

#### 8. 桶排序/基数排序(Radix Sort)

##### 1代码

#include <stdio.h>

int main()

{

int book[1001],i,j,t,n;

for(i=0;i<=1000;i++)

book[i]=0;

scanf("%d",&n);//输入一个数n，表示接下来有n个数

for(i=1;i<=n;i++)//循环读入n个数，并进行桶排序

{

scanf("%d",&t); //把每一个数读到变量t中

book[t]++; //进行计数，对编号为t的桶放一个小旗子

}

for(i=1000;i>=0;i--) //依次判断编号1000~0的桶

for(j=1;j<=book[i];j++) //出现了几次就将桶的编号打印几次

printf("%d ",i);

getchar();getchar();

return 0;

##### 2原理：

<http://blog.jobbole.com/100361/>

##### **4基数排序**

* 基数排序类似于桶排序，将元素分发到一定数目的桶中。不同的是，基数排序在分割元素之后没有让每个桶单独进行排序，而是直接做了合并操作。

**4基数排序**

* 基数排序类似于桶排序，将元素分发到一定数目的桶中。不同的是，基数排序在分割元素之后没有让每个桶单独进行排序，而是直接做了合并操作。

### 2分治法的主要思想，应用

(1) 将求解的较大规模的问题分割成k个**更小规模的子问题。**

(2) 对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。

(3) 将求出的**小规模的问题的解合并为一个更大规模的问题的解**，自底向上逐步求出原来问题的解。

**经典应用问题**

（1）二分搜索[1]  （5）合并排序（6）快速排序

### 2 二分查找

       合并排序算法是用分治策略实现对n个元素进行排序的算法。其基本思想是：将待排序元素分成大小大致相同的2个子集合，分别对2个子集合进行排序，最终将排好序的子集合合并为所要求的排好序的集合。

 算法名称：归并排序

  \* 算法描述：

  \* 1.将数组分为n等份(算法中为2)，对各子数组递归调用**归并排序**

  \* 2.等分为2份时为2路归并，最后子数组排序结束后，将元素合并起来，复制回原数组

  \*

  \* 复杂度分析：

  \* 1.平均时间复杂度：O(nlogn)

  \* 2.空间复杂度：O(n)(临时数据储存空间)

零碎问题

## 1 循环链表的好处

循环链表的特点是**无须增加存储量**，仅对表的链接方式稍作改变，即可使得表处理更加**方便灵活。**

①循环链表中没有NULL指针。涉及遍历操作时，其终止条件就不再是像非循环链表那样判别p或p-＞next是否为空，而是**判别它们是否等于某一指定指针，**如头指针或尾指针等。

②在单链表中，从一已知结点出发，只能访问到该结点及其后续结点，无法找到该结点之前的其它结点。而在单循环链表中**，从任一结点出发都可访问到表中所有结点，**这一优点使某些运算在单循环链表上易于实现

2指针是什么？空指针是什么？

指针就是一个存放地址的变量

指针是通过地址来访问变量的数据类型。简单的说就是指向的东西

### 1、C++的new和malloc有什么区别，

new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。

### 2、堆和栈有什么区别

一、堆栈空间分配区别：  
　　1、栈（[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3njRkuHNhnWu9rjRvuHT30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD" \t "_blank)）：由[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3njRkuHNhnWu9rjRvuHT30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD" \t "_blank)自动分配释放 　　2、堆（[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3njRkuHNhnWu9rjRvuHT30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD" \t "_blank)）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。  
二、堆栈缓存方式区别：  
　　1、栈使用的是[一级缓存](https://www.baidu.com/s?wd=%E4%B8%80%E7%BA%A7%E7%BC%93%E5%AD%98&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3njRkuHNhnWu9rjRvuHT30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD" \t "_blank)，调用完毕立即释放；  
　　2、堆是存放在二级缓存中，生命周期由虚拟机的[垃圾回收](https://www.baidu.com/s?wd=%E5%9E%83%E5%9C%BE%E5%9B%9E%E6%94%B6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3njRkuHNhnWu9rjRvuHT30ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1m4nH6YPjD" \t "_blank)算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。、

三、堆栈数据结构区别：  
　　堆（数据结构）：堆可以被看成是一棵树，如：堆排序；  
　　栈（数据结构）：一种先进后出的数据结构

### 3什么是随机变量

随机变量表示随机现象各种结果的变量

### 3.1怎么计算时间复杂度和空间复杂度

### 3[求无序数组中位数的方法](http://www.cnblogs.com/xiaoZQ/articles/5244816.html)，并说出时间复杂度，

简单的想了下：  
思路1） 把无序数组排好序，取出中间的元素  
            时间复杂度 采用普通的比较排序法 O(N\*logN)  
            如果采用非比较的计数排序等方法， 时间复杂度 O(N), 空间复杂度也是O(N).  
  
思路2）   
          2.1）将前(n+1)/2个元素调整为一个小顶堆，  
          2.2）对后续的每一个元素，和堆顶比较，如果小于等于堆顶，丢弃之，取下一个元素。 如果大于堆顶，用该元素取代堆顶，调整堆，取下一元素。重复2.2步             
          2.3）  当遍历完所有元素之后，堆顶即是中位数。  
  
思路3） 熟话说，想让算法跑的更快，用分治！  
            快速排序之所以得名"快排",绝非浪得虚名！因为快排就是一种分治排序法！  
            同样，找中位数也可以用快排分治的思想。具体如下：  
            任意挑一个元素，以改元素为支点，划分集合为两部分，如果左侧集合长度恰为 (n-1)/2，那么支点恰为中位数。如果左侧长度<(n-1)/2, 那么中位点在右侧，反之，中位数在左侧。 进入相应的一侧继续寻找中位点。  
            这种方法很快，但是在最坏的情况下时间复杂度为O(N^2), 不过平均时间复杂度好像是O(N)。

如何反转一个单链表，如何用HMM做分词......

听说你用SVM做过文本主题分类，那你详细说一下它的原理

# 921

## [C++中的栈内存和堆内存的区别](http://www.cnblogs.com/ChenZhongzhou/p/5685537.html)

* 数据结构中的堆与栈：

1栈：是一种**连续**储存的数据结构，具有**先进后出**的性质。通常的操作有入栈（圧栈）、出栈和栈顶元素。想要读取栈中的某个元素，就要将其之前的所有元素出栈才能完成。类比现实中的箱子一样。

堆：是一种**非连续**的树形储存数据结构，每个节点有一个值，整棵树是**经过排序**的。特点是根结点的值最小（或最大），且根结点的两个子树也是一个堆。常用来实现优先队列，存取随意

2栈内存：由程序自动向操作系统申请分配以及回收，速度快，使用方便，但程序员无法控制。

**堆内存**：程序员向操作系统申请一块内存，当系统收到程序的申请时，会遍历一个记录空闲内存地址的链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。分配的速度较慢，地址不连续，容易碎片化。此外，由程序员申请，同时也必须由程序员负责销毁，否则则导致**内存泄露**。

## C++交换不用第三个数

**第一类方法也是常用的方法，通过多次的数值计算来完成交换，到现在知道的有下面三种：**

**（1）加减法。**

**a = a + b;**

**b = a - b;**

**a = a - b;**

还有乘除法，亦或法

不足：会溢出

）使用xchg指令（2**）使用额外的栈**(3)使用mov指令

## Const和define的区别

const：有数据类型，编译进行安全检查，可调试   
define:宏，不考虑数据类型，没有安检，不能调试

const定义的只读变量在程序运行过程中只有一份拷贝(因为它是全局的只读变量，存放在静态区)，而#define定义的宏常量在内存中有**若干个拷贝**。

## Java和c的区别

，因为JAVA没有指针概念，都是通过java虚拟机完成了内存回收，因此对于编程者来说这个更易用。而C++需要编程人员自己掌握内存的申请和释放，如果掌握不好，就容易出现程序崩溃等问题。