一、算法性能分析

算法及其性能分析与度量

***算法的事前估计

**时间复杂度的渐进表示法

*大 O 表示法及其加法与乘法规则

*数量级按增长率由小到大的排列顺序

例 1：程序段"i=1; while(i<=n) i=i*2;"的时间复杂度为 $O(\log_2 n)$。

i=i*2，即循环次数 k 满足 $2^k=n$，因此 $k=\log_2 n$。

例 2：有如下计算 n!的递归函数 Fact(n)，分析其时间复杂度。

```
Fact(int n)
{
    if(n<=1) return(1);
    else return (n*Fact(n-1));
```

设 Fact(n)的运行时间函数为 T(n)。该函数中语句 if(n<=1) return(1);的运行时间为 O(1)，递归调用 Fact(n-1)的时间是 T(n-1)，故 else return (n*Fact(n-1));的运行时间为 O(1)+T(n-1)。其中，设两数相乘和赋值操作的运行时间为 O(1)，则对某常数 C、D 有：$T(n)=\begin{cases} D & n\le 1 \\ C+T(n-1) & n>1 \end{cases}$。

现在，来求解该方程。设 n>2，利用上式对 T(n-1)展开，即在上式中用 n-1 替代 n 得到：T(n-1)=C+T(n-2)，并代入 T(n)=C+T(n-1)中，即当 n>2 时有：T(n)=2C+T(n-2)。同理，当 n>3 时有：T(n)=3C+T(n-3)。因此，当 n>i 时有：T(n)=iC+T(n-i)。

最后，当 i=n-1 时有：T(n)=(n-1)C+T(1)=(n-1)C+D。

即 T(n)=O(n)。

二、 数组、顺序表与字符串

数组

***数组的顺序存储方式

**一维、二维、三维（按照各种优先次序）

**特殊矩阵的存储（稀疏矩阵）

例 1：对称矩阵

由于对称矩阵中的元素关于对角线对称，因此存储时只需存储矩阵的上三角或下三角元素，使得对称元素共享一个存储空间。假如存储下三角的元素，则元素的总数为 n(n+1)/2；若按以行为主序存储在 A[1..n(n+1)/2]中，则 A[k]与的 a_{ij} 的对应关系为：

$$k=\begin{cases} i(i-1)/2+j & 当 i \geq j \\ j(j-1)/2+i & 当 i < j \end{cases}$$

例 2：三角矩阵

以主对角线划分，三角矩阵有上三角和下三角两种。上三角矩阵是指矩阵的下三角（不含对角线）中的元素均为常数 C 或零的 n 阶矩阵，下三角矩阵与之相反。在三角矩阵中值相同的元素可共享一个存储空间，若重复值为零则不分配空间，其元素共有 n(n+1)/2 个。

(1) 当下三角矩阵中重复元素为非零时，A[k]与 a_{ij} 的对应关系为：

$$k=\begin{cases} i(i-1)/2+j & 当 i \geq j \\ n(n+1)/2+1 & 当 i < j \end{cases}$$

(2) 当上三角矩阵中重复元素为非零时，A[k]与 a_{ij} 的对应关系为：

$$k=\begin{cases} (i-1)(2n-i+2)/2+(j-(i-1)) & 当 i \geq j \\ n(n+1)/2+1 & 当 i < j \end{cases}$$

例 3：三对角矩阵

所有非零元素都集中在以主对角线为中心的带状区域中，即除了主对角线和主对角邻近的上下方外，其余元素均为零。在三角矩阵中，除第一行和最后一行各有两个元素外，，其余元素均为零。

在三对角矩阵中，除了第一行和最后一行各有两个元素外，其余各行均有三个非零元素，所以共有 3n-2 个非零元素。

(1) 主对角线左下角的对角线上的元素下标具有关系式 i=j+1，而此时的 k 为：

k=3(i-1)　　（当 i=j+1 时）

(2) 主对角线上的元素下标具有关系式：i=j，而此时的 k 为：

k=3(i-1)+1　　（当 i=j 时）

(2) 主对角线右上角的对角线上的元素下标具有关系式：i=j-1，而此时的 k 为：

k=3(i-1)+2　　（当 i=j-1 时）

综合(1), (2), (3)得：k=2(i-2)+j

例 4：五对角矩阵

五对角矩阵中的 k 与 i、j 的关系如下：

$$k=\begin{cases} 4(i-1)+j & i=1 \\ 4(i-1)+j-1 & 1<i<n \\ 4(i-1)+j-2 & i=n \end{cases}$$

例 5：二维数组 A 的每个元素是由 6 个字符组成的串，其行下标 i=0, 1, …., 8，列下标 j=1, 2, …, 10。若 A 按行先存储，元素 A[8, 5]的起始地址与当 A 按列先存储时的元素的哪一个元素的起始地址相同。设每个字符占一个字节。

设二维数组 A[c_1..d_1, c_2..d_2]，每个数据元素占 L 个字节，则：

行优先存储地址计算公式：

$Loc(a_{ij})=Loc(a_{c1c2})+[(i-c_1)*(d_2-c_2+1)+(j-c_2)]*L$

列优先存储地址计算公式：

$Loc(a_{ij})=Loc(a_{c1c2})+[(j-c_2)*(d_1-c_1+1)+(i-c_1)]*L$

(8-0)*(10-1+1)+(5-1)=(j-1)*(8-0+1)+(i-0)

即 9(j-1)+(i-0)=84

推出 A[3, 10]。

*****顺序表*****

***查找、插入和删除运算及其性能评价

例 1：用线性表的顺序存储结构来描述一个城市的设计和规划是否合适？为什么？

不合适。因为一个城市的设计和规划涉及非常多的项目，比较复杂，需要经常改动、扩充和删除各种信息，这样才适应不断发展的需要，所以顺序表不能很好地适应其需要。

例 2：线性表的顺序存储结构具有三个弱点：其一，在作插入或删除操作时，需移动大量元素；其二，由于难以估计，必须预先分配较大的空间，往往使存储空间不能得到充分利用；其三，表的容量难以扩充。线性表的链式存储结构是否一定都能够克服上述三个弱点。试讨论之。

不一定。由于链式存储需要额外的空间来存储指针，所以要比顺序存储多占用空间。在空间允许的情况下，链式存储结构可以克服顺序存储结构的弱点，但空间不允许时，链式存储结构会出现新问题。

*****字符串*****

***基本概念（空串、空白串及子串）

***穷举模式匹配与改进的 KMP 算法（两者性能分析与比较）

例 1：给定一模式串 p=”abcabaa”与目标串 t=”abcaabbabcabaacbacba”，求解模式串的失效函数值，并给出利用 KMP 算法进行模式匹配的每一趟匹配过程的实现。

例 2：设 S 为一个长度为 n 的字符串，其中的字符各不相同，求解 S 中互异的非平凡子串（非空且不同于 S 本身）的个数。

除长度为 n 的子串外，长度为 n-1 的不同子串个数为 2，长度为 n-2 的不同子串个数为 3，依此类推，直至长度为 1 的不同子串个数为 n，即 S 的非平凡子串个数为：2+3+…+n=n(n+1)/2-1。

例 3：若串 S=”software”，求解其子串数目。

已知 S 串长度为 8，则其子串个数为 1+2+3+…+8=8/2(1+8)=36。

例 4：如果字符串的一个子串（其长度大于 1）的各个字符均相同，则称之为等值子串。试设计一个算法，输入字符串 S，以”!”作为结束标志。如果串 S 中不存在等值子串，则输出信息“无等值子串”，否则求出（输出）一个长度最大的等值子串。

先从键盘上接受字符串并送入字符串数组 S，然后扫描字符串数组 S。设变量 head 指向当前发现的最长等值子串的串头，max 记录此子串的长度。扫描过程中，若发现

等值子串则用 count 变量记录其长度，如果它的长度大于原最长等值子串的长度，则对 head 和 max 进行更新。重复上述过程直到 S 的末尾。最后，根据扫描所得的结果输出最长等值子串或输出等值子串不存在信息。

```
void Equstring (char s[ ])
{
    for(k=0; ; k++)
    {
        scanf((“%c”, &S[k]);
        if (S[k]==’!’)
            break;
    }
    for (i=0, j=1, head=0, max=1; S[i]!=’!’ && S[j]!=’!’; i=j, j++)
    {
        count=1;
        while (S[i]==S[j])
        {
            j++;
            count++;
        }
        if (count>max)
        {
            head=i;
            max=count;
        }
    }
    if (max>1)
        for (k=head; k<(head+max); k++)
            printf(“%c”, S[k]);
    else
        printf(“There is no equivalent substring in S!”);
}
```

三、 链表

单链表

***查找、插入和删除运算及其性能评价（无序链表和有序链表）

***静态链表

循环链表

***查找、插入和删除运算及其性能评价（无序链表和有序链表）

双向链表

***查找、插入和删除运算及其性能评价（无序链表和有序链表）

稀疏矩阵

例 1、(1) 对一个线性表分别进行遍历和逆置运算，求解其最好的渐进时间复杂度表示。

遍历时间的渐进时间复杂度表示为 O(n)；进行逆置运算时，顺序表的时间要少于单链表时间，其量级也为 O(n)。

(2) 求解求顺序表和单链表长度的渐进时间复杂度表示。

顺序表的长度可以直接从最后一个元素的位置获得，而单链表的长度则必须遍历整个表后获得，即 O(1)与 O(n)。

(3) 若给定 n 个元素的向量，则求解建立一个有序单链表的渐进时间复杂度表示。

单纯建立单链表的渐进时间复杂度为 O(n)，而建立一个有序单链表还涉及查找插入到其正确位置，故渐进时间复杂度表示为 $O(n^2)$。

例 2：设有头结点的单链表 L，编程对表中任一值只保留一个结点，删除其余值相同的结点。

(1) 首先用指针 p 指向链表中第一个数据结点，然后用指针 t 搜索整个链表以寻找值相同的结点直到链尾；在搜索中，指针 s 指向 t 所指结点前驱结点，当 t->data=p->data 时则删除 t 所指结点，即 s->link=t->link。

(2) 修改指针 p，使 p 指向链表的下一个结点（即 p=p->link），然后重复(1)的操作直至 p=NULL。

```
DelElem(lklist *L)
{
    Pointer *p, *t, *pre;
    p=L->link;
    t=p;
    while (p!=NULL)
    {
        pre=t;
        t=t->link;
        do
        {
            while ((t!=NULL) && (t->data!=p->data))
            {
                pre=t;
                t=t->link;
            }
            if (t!=NULL)
            {
                pre->link=t->link;
                free(t);
                t=pre->link;
            }
```

```
            }while (t!=NULL);
            p=p->link;
            t=p;
        }
    }
```

例 3：有一带头结点的单链表，编程将链表颠倒过来，要求不用另外的数组或结点完成。

在遍历原单链表每个结点的同时摘下该结点插入到新链表的表头。这样，先遍历到的结点先插入，后遍历到的结点后插入；由于插入是在表头进行，所以先插入的结点成为表尾，后插入的结点成为表头；也即实现了链表的逆置。

```
void reverse (lklist *h)
{
    lklist *p, *q;
    p=h->link;      /*p 指向原链表的第一个数据结点*/
    h->link=NULL;     /*新链表初始为空*/
    while (p)
    {
        q=p;       /*q 指向将摘下来插入到新链表的结点*/
        p=p->link;      /*p 指向下一个待摘下来的原链表结点*/
        q->link=h->link;          /*将*q 插入到新链表表头*/
        h->link=q;
    }
}
```

四、 栈和队列

栈

***栈的基本特点

***顺序栈（初始状态、栈空、栈满、进栈、出栈、取栈顶）

***链式栈（初始状态、栈空、进栈、出栈、取栈顶）

队列

***队列的基本特点

***顺序队列（初始状态、队空、队满、进队、出队、取队头）

**一般队列（伪溢出）

**循环队列

***链式队列（初始状态、队空、进队、出队、取队头）

优先级队列

***优先级队列的基本特点（出队）

*****表达式的计算*****

例 1：若用单链表来表示队列，则应该选用带尾指针的循环链表。
设尾指针为 tail，则通过 tail 可以访问队尾，通过 tail->next 可以访问队头。

例 2：设栈的输入序列为 1, 2, 3, …, n，输出序列为 $a_1, a_2, \ldots, a_n$，若存在 $1 \le k \le n$，使得 $a_k=n$，则当 $k \le i \le n$ 时，a_i 为不确定。
由于输出序列 $a_1, a_2, \ldots, a_n$ 并没有明确标示出具体的序列值，只是一个变量序列，即可能是根据输入序列所能形成的任何出栈序列；所以，当 $1 \le k \le n$ 且 $a_k=n$ 时，并不能确定 $k \le i \le n$ 时的 a_i。
设栈的输入序列为 1, 2, …, n，若输出序列的第一个元素为 n，则第 i 个输出元素为 n-i+1。
若栈输出的第一个元素为 n，则由栈定至栈底顺序存放的必然是 n, n-1, …, 2, 1，故第 i 个输出的元素为 n-i+1。

例 3：在给定进栈序列的前提下，判断合理的出栈序列以及计算出栈序列的不同种数。

例 4：设栈 S 和队列 Q 的初始状态为空，元素 1, 2, 3, 4, 5, 6 依次通过栈 S，一个元素出栈后即进入队列 Q，若 6 个元素出队的序列为 2, 4, 3, 6, 5, 1，则栈 S 的容量至少应该是 3。

例 5：利用两个栈来模拟一个队列，已知栈的三种运算定义为：PUSH(ST, x)、POP(ST, x)及 SEMPTY(ST)。利用栈的运算来实现该队列的三个运算：进队、出队及判队空。
由于队列是先进先出，而栈是先进后出；所以只有经过两个栈，即先在第一个栈里先进后出，再经过第二个栈后进先出来实现队列的先进先出。因此，用两个栈模拟一个队列运算就是用一个栈作为输入，而另一个栈作为输出。当进队列时，总是将数据进入到作为输入的栈中。在输出时，如果作为输出的栈已空，则从输入栈将已输入到输入栈的所有数据压入栈中，然后由输出栈输出数据；如果作为输出的栈不空，则就从输出栈输出数据。显然，只有在输入，输出栈均为空时队列才为空。一个栈用来插入元素，另一个栈用来删除元素，删除元素时应将前一栈中的所有元素读出，然后进入到第二个栈中。

```
void Enqueue(s1, x)
stack s1;
int x;
{
    if (s1->top= =n)     /*已达到队列最大值*/
        printf(‘队列上溢”);
    else
        push(s1, x);
}
```

```
void Dequeue(s1, s2, x)
stack s1, s2;
int x;
{
    s2->top=0;      /*将 s2 清空*/
    while (!Empty(s1))        /*将 s1 的所有元素退栈后压入 s2，此时 s2 为空*/
        push(s2, pop(s1));
    pop(s2, x);      /*弹出栈 s2 的栈顶元素（也即为队首元素）并赋给 x*/
    while (!Empty(s2))        /*将剩余元素重新压入栈 s1 恢复为原 s1 中的顺序*/
        push(s1, pop(s2));
}

int Queue_empty(s1)
stack s1;
{
    if Empty(s1)
        return (1);
    else return (0);
}
```

例 6：约瑟夫问题的解法

```
#include <stdio.h>
#include   <stdlib.h>
typedef int DataType ;
typedef struct qNode {
   DataType data;
   struct qNode *next;
} QueueNode;
typedef struct {
   QueueNode *rear;   //指向队尾辅助结点指针
}LinkQueue;
LinkQueue q;
void   Josephus (LinkQueue q,   int n, int m )
{   int i, j; QueueNode *p = q.rear, *t;
    printf(" \n ");
    for ( i = 0; i < n; i++ ) {   //执行 n 次
        for ( j = 0; j < m-1; j++ )
        p = p->next;
        printf("出列的人是: %d\n", p->next->data);
        t = p->next; p->next = t->next;
        free(t);
    }
    q.rear = NULL;
}
```

```
void main ( )
{   int n, m, i; QueueNode *p;
    printf("Enter n m ?   ");
    scanf("%d%d", &n, &m);
    q.rear = NULL;//  空队列
    for (i = 1; i <= n; i++ ) {   //形成约瑟夫环
          p = (QueueNode *)malloc(sizeof(QueueNode));
          p->data = i; printf("%d   ", i);
          if (q.rear == NULL) p->next = p;
          else {     p->next = q.rear->next; q.rear->next = p;      }
    q.rear = p;
    }
  Josephus (q, n, m);       //解决约瑟夫问题
}
```

例 7：以带头结点的循环链表表示队列，并且不设头指针，只设队尾指针，试编写相应的队初始化、判队空、入队和出队算法函数。

```
typedef struct qnode {
    dataType data;
    struct qnode *next;
} QNODE;
typedef struct {
    QNODE *rear;
} LinkQueue;
void initLoopQueue(LinkQueue *qpt)
{
    qpt->rear = (QNODE *) malloc (sizeof(QNNODE));
    qpt->rear->next = qpt->rear;
}
int isLoopQEmpty(LinkQueue *qpt)
{
  return qpt->rear = = qpt->rear->next;
}
void inLoopQueue(LinkQueue *qpt, dataType x)
{
     QNODE *p = (QNODE *)malloc(sizeof(QNODE));
     p->data = x; p->next = qpt->rear->next;
     qpt->rear = qpt->rear->next = p;
}
int deLoopQueue(LinkQueue *qpt, dataType *xpt)
{
    QNODE *p = qpt->rear->next->next;//队列首结点指针
    if(isLoopQEmpty(qpt))
       return 0;
```

```
    *xpt = p->data;
    qpt->rear->next->next = p->next;
    free(p);
    return 1;
}
int lookFront(LinkQueue *qpt, dataType *xpt)
{
    if(isLoopQEmpty(qpt))
        return 0;
    *xpt = qpt->rear->next->next->data;
    return 1;
}
```

五、 递归

递归

***递归算法的设计与实现

***递归到非递归的转换（引入栈）

广义表

***广义表的基本概念

***广义表的存储结构

***广义表的递归算法

例 1：若广义表满足 Head(A)=Tail(A)，则 A 为(())，其长度和深度为 1 和 2。

例 2：按以下计算规则求已知两个整数的最大公约数。

$$gcd(a, b) = \begin{cases} a\%b == 0, b; \\ b\%a == 0, a; \\ gcd(a\%b, b), a > b; \\ gcd(a, b\%a), a < b. \end{cases}$$

```
//函数以两个已知整数为形参，返回整型结果。
int gcd int a, int b)
{
    if(a % b == 0) return b;
    if(b % a == 0) return a;
    if(a > b) return gca(a%b, b);
    if(a < b) return gcd(a, b % a);
}
```

例 3、用递归函数实现已知十进制整数返回十进制的反向的整数。

例如，已知十进制数 1234，返回反向的十进制数是 4321。

一个数的反向过程是一个反复将数的个位作为对应的高位的转换过程：设一个数已被译出了一部分，例如，对于 1234，已译出部分高位 43，还有 12 还未被转换。问题是如何将 12 的个位数 2 转移到 43，使新的高位变成 432 呢？只要将 43 乘 10 后加上 2 即可。若用循环实现有以下函数：

```
int reverInt(int n)
{    int s = 0;
     while(n) {
      s = s*10 + n%10;
      n /= 10;
     }
     return s;
}
```

如果将上述计算改写成递归，需要将循环计算中高位部分的值作为参数。初始调用时，对应 s 为 0 值。在被转换的数是个位数情况下，函数的返回值是 s*10+n。如果 n 是多位数，则要继续转换的数变成 n/10，新的高位的部分值是 s*10+n%10。

```
int reverIntRe(int n, int s)
{
     if(n<10) return s*10+n;
     return reverIntRe(n/10, s*10 + n % 10);
}
```

例 4：汉诺塔问题的非递归解法。

分析递归解法可知：盘子移动的输出是在两种情况下进行的：其一是当递归到 n=1 时进行输出；其二是返回到上一层函数调用具有 n>=1 时进行输出。因此，可以使用一个 stack[]数组来实现汉诺塔问题的非递归解法。具体步骤如下：

(1) 将初值 n, x, y, z 送数组 stack[]；

(2) n 减 1 并交换 y, z 值后将 n, x, y, z 值送数组 stack[]，直到 n=1 时为止；

(3) 当 n=1 时输出 x 值指向 z 值；

(4) 回退到前一个数组元素，如果此时这个数组元素下标值大于等于 1，则输出 x 值指向 z 值（相当于递归函数 hanoi 返回到上一层函数调用时的输出）；当 i<=0 时程序结束。

(5) 将 n 减 1，如果 n>=1 则交换 x 和 y 值（相当于执行递归函数 hanoi 中的第二个 hanoi 调用语句参数替换时的情况）；并且：

A. 如果此时 n=1，则输出 x 值指向 z 值并继续退回到前一个数组元素处（相当于返回到递归函数 hanoi 的上一层）；如果 i>=1，则输出 x 值指向 z 值；

B. 如果此时 n>1，则转(2)处继续执行（相当于递归函数 hanoi 中的第二个 hanoi 调用语句继续调用的情况）。

```
#include <stdio.h>
struct hanoi
{
        char x, y, z;
```

```
        int id;
}stack[20];
void main( )
  {
        int i=1;
        int n;
        char s;
        printf("Input number of disks:\n");
        scanf("%d", &n);
        printf("The step to moving %d disks:\n", n);
        if (n==1)
              printf("a->c");
        else
              {
                    stack[1].id=n;
                    stack[1].x='a';
                    stack[1].y='b';
                    stack[1].z='c';
                    do
                    {
                          while (n>1)
                          {
                                n=n-1;
                                i=i+1;
                                stack[i].id=n;
                                stack[i].x=stack[i-1].x;
                                stack[i].y=stack[i-1].z;
                                stack[i].z=stack[i-1].y;
                          }
                          printf("%c", stack[i].x);
                          printf("(");
                          printf("%d", stack[i].id);
                          printf(")->");
                          printf("%c\n", stack[i].z);
                          i=i-1;
                          do
                          {
                                if (i>=1)
                                {
                                      printf("%c", stack[i].x);
                                      printf("(");
                                      printf("%d", stack[i].id);
                                      printf(")->");
                                      printf("%c\n", stack[i].z);
```

```
                    }
                    stack[i].id=stack[i].id-1;
                    n=stack[i].id;
                    if (n>=1)
                    {
                        s=stack[i].x;
                        stack[i].x=stack[i].y;
                        stack[i].y=s;
                    }
                    if (n==1)
                    {
                        printf("%c", stack[i].x);
                        printf("(");
                        printf("%d", stack[i].id);
                        printf(")->");
                        printf("%c\n", stack[i].z);
                        i=i-1;
                    }
                }while ((n<=1) && (i>0));
            }while (i>0);
        }
        getch( );
    }
Input number of disks:
3
The step to moving 3 disks:
a(1)->c
a(2)->b
c(1)->b
a(3)->c
b(1)->a
b(2)->c
a(1)->c
Press any key to continue
```

例 5：广义表一些操作的递归解法的实现。

六、 树

树

***树的存储结构（左孩子-右兄弟表示法）

***树的遍历（深度优先搜索与广度优先搜索）

*****二叉树*****

***二叉树的存储结构（顺序存储与链式存储）
***二叉树的遍历（递归解法与非递归解法及其应用）
***二叉树的计数
***哈夫曼树（哈夫曼编码）
***完全二叉树与满二叉树的基本概念及性质

*****线索化二叉树*****

***线索化二叉树的建立

*****堆*****

***堆的基本概念
***堆的操作（建立、插入和删除）
**堆的两个调整函数

*****森林*****

***森林的二叉树表示
***森林的遍历

例 1、一棵完全二叉树上由 1001 个结点，其中叶子结点的个数为 501。

由二叉树性质知：$n_0=n_2+1$，且完全二叉树的 $n_1=0$ 或 1；已知二叉树的总结点数 $n=n_0+n_1+n_2$，即有 $n=2n_0+n_1-1$；将总结点数 1001 代入得：$1001=2n_0+n_1-1$，因 1001 为奇数，故 $n_1=0$，得 $n_0=501$。

一棵有 124 个叶结点的完全二叉树，最多有 247 个结点。

由二叉树性质知：$n_0=n_2+1$，$n_2=123$，又因 n_0 为偶数，所以 $n_1=0$。

例 2：设深度为 k 的二叉树上只有度为 0 和度为 2 的结点，则这类二叉树上所含的结点总数为 2k-1。

设 n_0 为哈夫曼树的叶子结点数目，则该哈夫曼树共有 $2n_0-1$ 个结点。

由哈夫曼树构造方法知，对 n_0 个叶子结点构造哈夫曼树需要进行 n_0-1 次构造新结点操作，最终形成共有 $2n_0-1$ 个结点的哈夫曼树，即 $2n_0-1$。

例 3：在一棵二叉树的前序序列、中序序列和后序序列中，任意两种序列的组合可以唯一地确定这棵二叉树吗？若可以，有哪些组合，证明之；若不可以，有哪些组合，证明之。

前序序列和中序序列、后序序列和中序序列可以唯一确定一棵二叉树，而前序序列和后序序列不能。

例 4：假设二叉树用左右链表示，试编写一算法，判别给定二叉树是否为完全二叉树。

根据完全二叉树定义可知，对完全二叉树按照从上到下，从左到右的次序遍历应满足：

(1) 若某结点没有左孩子，则一定无右孩子；

(2) 若某结点缺（左或右）孩子，则其所有后继一定无孩子。

反之，可采用按层次序遍历二叉树的方法依次对每个结点进行判断。为此增加一个标志以表示所有已扫描过的结点均有左、右孩子，将局部判断结果存入 CM 中，CM 表示整个二叉树是否是完全二叉树，B 为 1 表示到目前为止所有结点均有左右孩子。

```
int cbit(Bitree *t)
{
        Initqueue(Q);        /*队列初始化*/
        B=1;
        CM=1;
        if (t!=NULL)
        {
            AddQueue(Q, t);
            while (!QueueEmpty(Q))      /*当队列非空时执行循环*/
            {
                p=DelQueue(Q);          /*出队*/
                if (p->lchild= =NULL)
                {
                    B=0;      /*B=0 表示缺少左、右孩子*/
                    if (p->rchild!=NULL)
                        CM=0;          /*CM=0 表示不是完全二叉树*/
                }
                else
                {
                    CM=B;
                    AddQueue(Q, p->lchild);          /*左孩子入队*/
                    if (p->rchild= =NULL)
                        B=0;
                    else
                        AddQueue(Q, p->rchild);      /*右孩子入队*/
                }
            }
        return (CM);
    }
}
```

例 5：设计算法：统计一棵二叉树中所有叶结点的数目。

可将此问题视为一种特殊的遍历问题，这种遍历中“访问一个结点”的内容为判断该结点是否为树叶，若是树叶则叶子数加 1。显然，可以采用任何遍历方法，在此采用前序遍历方法。下面算法中记录叶子数的 count 初值假定为 0。

```
void countleaf(Bitree *t, int *count)
{
```

```
    if (t!=NULL)
    {
        if ((t->lchild= =NULL) && (t-rchild= =NULL))
            *count++;       /*t 所指的结点是叶子，计数加 1*/
        countleaf(t->lchild, count);    /*累计左子树上的叶子数*/
        countleaf(t->rchild, count);    /*累计右子树上的叶子数*/
    }
}
```

例 6：假设二叉树用左右链表示，试编写一算法，求任意二叉树中第一条最长的路径，并输出此路径上各结点的值。

采用非递归后序遍历二叉树，当后序遍历访问到由 p 所指的树叶结点时，此时 stack 中所有结点均为 p 所指结点的祖先，由这些祖先便构成了一条从根结点到此树叶结点的路径。此外，另设一 longestpath 数组来保存二叉树中最长的路径结点值，m 为最长的路径长度。

```
void LongPath(Bitree *root)
{
    Bitree *stack[Maxsize], *s;
    char longestPath[Maxsize];
    int i, m, top;
    m=0;
    top=0;
    s=root;
    do
    {
        while (s!=NULL)         /*扫描左孩子，入栈*/
        {
            top++;
            stack[top]=s;
            tag[top]=0;         /*右孩子还未访问过*/
            s=s->lc;
        }
        if (top>0)
        {
            if (tag[top]= =1)    /*左、右孩子均已访问过，则访问该结点*/
            {
                if ((stack[top]->lc= =NULL) && (stack[top]->rc= =NULL) &&
                    (top>m))
                {
                    for (i=1; i<=top; i++)
                      longestPath[i]=stack[i]->data;
                      m=top;
                }
                top--;
```

```
                }
                else
                {
                    s=stack[top];
                    if (top>0)
                    {
                        s=s->rc;        /*扫描左孩子*/
                        tag[top]=1;     /*置当前结点右子树已访问过标志*/
                    }
                }
            }
        }while ((s!=NULL) || (top!=0));
        for (i=1; i<m; i++)
            printf("%c", longestPath[i]);
        printf("\n longest=");
        printf("%d", m);
    }
```

七、 集合与搜索

集合及其表示

***几种实现集合运算的存储结构

等价类和并查集

***确定等价类的几种方法（有序链表与并查集）

***并查集的操作（构造函数、查找与合并）

静态搜索结构

***顺序搜索

***折半搜索

二叉搜索树

***二叉搜索树的基本概念

***二叉搜索树的建立、插入和删除操作

AVL 树

**AVL 树的基本概念（平衡因子）

***AVL 树的建立、插入和删除操作（平衡化旋转——单旋与双旋）

例 1：从具有 n 个结点的二叉搜索树中查找一个元素时，最坏情况下的渐进时复杂度为 O(n)。
最坏情况下二叉搜索树变为单支树，渐进时间复杂度由 $O(\log_2 n)$变为 O(n)。
二叉搜索树的查找长度不仅与所包含的结点个数有关，而且也与二叉搜索树的生成过程有关。

例 2：试述顺序查找法、折半查找法对被查找的表中元素的要求，对长度为 n 的表来说，两种查找法在查找成功时的搜索长度各是多少？
两种方法对查找的要求如下：
(1) 顺序查找法：表中的元素可以任意存放，没有任何限制。
(2) 折半查找法：表中的元素必须以关键字的大小递增或递减的次序存放并且只能采用顺序存储方式存储。
两种方法的平均搜索长度分别如下：
(1) 顺序查找法：类似于顺序表查找成功时的评价方式 $ASL_{sq}=(n+1)/2$。
(2) 折半查找法：利用二叉搜索树评价性能 $ASL_{bin}=((n+1)/n)*\log_2(n+1)-1$。

例 3：试给出一棵树最少的关键字序列，使 AVL 树的四种调整平衡操作(LL, LR, RR, RL)至少执行一次，并画出其构造过程。
4, 5, 7, 2,1, 3, 6
4
4, 5
4, 5, 7　　LL
5, 4, 7, 2
5, 4, 7, 2, 1　　RR
5, 2, 7, 1, 4, 3　　LR
4, 2, 5, 1, 3,7, 6　　RL

例 4：设二叉搜索树利用二叉链表作为存储结构，其每一结点数据域为整数，现给出一个整数 x，请编写非递归程序，实现将 data 域之值小于 x 的结点全部删除掉。
在非递归中序遍历二叉搜索树过程中，若访问的结点其 data 值小于等于 x 时则删除此结点，但这种操作应仍然保持二叉搜索树中序遍历的有序特性。
难点在于，当删除某结点后，如何正确访问到所删除结点的后继结点。分析如下：若一个结点*p 将要被删除，则此结点*p 的左孩子必定为空，这是因为*p 的左孩子其关键字一定比*p 关键字小，所以必然在结点*p 之前删除。这样，如果结点*p 的右孩子为空，则可直接删除*p 结点，然后从栈中弹出栈顶元素继续进行中序遍历；如果结点*p 的右孩子不为空，则将 p 指向*p 的右孩子，然后删除原*p 结点，接下来再对 p 继续进行中序遍历。

```
void DeleteBST(Bitree *T, int x)
/*非递归中序遍历二叉搜索树，若所访问结点的 data 值小于等于 x 时则删除之*/
{
    Bitree *p, *q;
    initStack(S);
```

```
    p=T;
    q=NULL;
    while ((p!=NULL) || (!StackEmpty(S)))
    {
        if (p!=NULL)
        {
            push(S, p);
            p=p->lchild;
        }
        else
        {
            pop(S, p);
            if (p->data<=x)      /*若 p 所指结点<=x 时则应删除*/
            {
                if (p->rchild= =NULL)  /*p 的右孩子为空则直接删除*p 结点*/
                {
                    free(p);
                    pop(S, p);      /*弹出被删除结点*p 双亲结点由 p 指向*/
                    p->lchild=NULL;
                    /*原被删除结点*p 是现*p 结点的左孩子，故置空*/
                }
            }
            else        /*结点*p 的右孩子非空，修改 p 使其指向*p 的右孩子*/
            {
                q=p;
                p=p->rchild;
                free(q);
            }
        }
        p=p->rchild;
    }
}
```

例 5：判二叉树是否是一棵二叉搜索树。

算法思想：

采用递归算法。空二叉树 T 是一棵二叉搜索树；若 T 非空，首先是它的左、右两棵子树都是二叉搜索树，接着还要求左子树中的所有结点的键值都小于根结点的键值；并且右子树中的所有结点的键值都大于根结点的键值。为了能简单实现子树所有结点与根结点比较，对于左子树可用左子树中最大的键值；对于右子树可以用右子树中最小的键值。若左子树中的最大键值比根结点键值小；并且右子树中的最小键值比根结点键值大，就能得出这是一棵二叉搜索树的结论。

为此要设计的递归判定函数在判定一棵二叉树是二叉搜索树的过程中同时求得这棵二叉树的最大键值和最小键值。

```
#include <stdio.h>
```

```
#include <stdlib.h>
int a[] = {1, 2, 3, 4, 5, 6, 7};
typedef struct node {    /*二叉树结点类型*/
    int data;
    struct node *leftChild, *rightChild;
} NODE;
//已知数组段 a[low]..a[high]建一棵子树的函数
NODE *built(int a[], int low, int high)
{    NODE *p;
     int mid;
    if (low > high) return NULL;
    mid = (high+low)/2;
    p = (NODE *)malloc(sizeof(NODE));
    p->data = a[mid];
    p->leftChild = built(a, low, mid-1);
    p->rightChild = built(a, mid+1, high);
    return p;
   }
//二叉树是否是二叉查找树，并回答最小键值和最大键值
int isSearchtree(NODE *t, int *minp, int *maxp)
{
    int lmin, lmax, rmin, rmax;
    if (t == NULL) return 1; lmin = lmax = t->data;
    if (t->leftChild)    {
    if(isSearchtree(t->leftChild,&lmin,&lmax)==0)
            return 0;
    if (t->data <= lmax)    return 0;
    }
    rmin = rmax = t->data;
    if (t->rightChild)    {
    if(isSearchtree(t->rightChild,&rmin,&rmax)==0)
             return 0;
    if (t->data >= rmin)    return 0;
    }
    *minp = lmin;        *maxp = rmax;
    return 1;
}
void inTrave(NODE *t)
{
    if (t) {
      inTrave(t->leftChild);
      printf("%4d", t->data);
      inTrave(t->rightChild);
  }
```

```
}
void main()
{
    int i, min, max;
    NODE *r;
    for(i=0; i<7; i++)
        printf("%4d", a[i]);
    printf("\n");
    r = built(a, 0, 6);
    inTrave(r);
    printf("\n");
    i = isSearchtree(r, &min, &max);
    printf("\t%s\n", (i == 1) ? "OK" : "NO");
}
```

例 6：输入一篇英文短文，统计短文中不同单词的出现次数，并要求按单词的词典编辑顺序输出统计结果。

为了有效地实现问题的要求，可以用二叉搜索树数据结构作为单词表，将单词作为结点的键值，结点还包含单词出现次数的信息。程序从正文中读取下一个单词，接着查单词表，若单词表中已有该单词，则该单词的出现次数增 1；若单词表中还没有该单词，则生成该单词的新结点，并将它插入在单词表中。

待正文结束，中序遍历单词表，将遍历经过的二叉树上结点的信息输出。

设程序从正文文件 text.in 读入短文，统计该短文中不同单词和它的出现次数，并按词典编辑顺序将单词及它的出现次数输出到正文文件 word.out 中。

```
#include   <stdio.h>
#include   <malloc.h>
#include   <ctype.h>
#include   <string.h>
#define    INF   "TEXT.IN"
#define    OUTF   "WORD.OUT"
typedef   struct treenode {
    char *word;
    int count;
    struct treenode *left, *right;
} BNODE;
/*   从与 fpt 所对应的文件中读取单词置入 word，并返回单词中的字符数；若读单词遇文件尾，已无单词可读取时，则返回 0。*/
 int getword(FILE *fpt, char *word)
 {   int i, c;
     do c = fgetc(fpt);
     while (c != EOF && !isalpha(c));
     if (c == EOF) return 0;
     for(i = 0; isalpha(c); i++)   {
         word[i] = c;
```

```
            c = fgetc(fpt);
        }
        word[i] = '\0';
        return i;
  }
void bTree(BNODE *t, char *word)
{    BNODE *ptr, *p; int cmpres;
     p = NULL; ptr = t ;
     while (ptr)    {
       cmpres = strcmp(word, ptr->word );
       if (!cmpres) {ptr->count++ ; return; }
       else {p = ptr ; ptr = cmpres > 0 ? ptr->right : ptr->left;
       }
    }
    ptr = (BNODE *)malloc(sizeof(BNODE));
    ptr->right = ptr->left = NULL;
    ptr->word = (char *)malloc(strlen(word)+1);
    strcpy(ptr->word, word);      ptr->count = 1;
    if (p == NULL) t = ptr;
    else if (cmpres > 0) p->right = ptr;
    else p->left = ptr;
}
void midorder(FILE *fpt, BNODE *t)
{    if (t == NULL )    return;
     midorder(fpt, t->left);
     fprintf(fpt, "    %s      %d\n", t->word, t->count);
     midorder(fpt, t->right);
}
void main()
{ FILE *fpt; char word[40];
    BNODE *root = NULL;
    if ((fpt = fopen(INF, "r")) == NULL) {
         printf("Can't open file %s\n.", INF);
         return;
    }
    while (getword(fpt, word) == 1) bTree(&root, word);
    fclose(fpt);
    fpt = fopen(OUTF, "w");
    midorder(fpt, root);
    fclose(fpt);
  }
```

八、 图

*****图的基本概念*****

***基本概念（完全图、顶点的度、连通图与连通分量、强连通图与强连通分量）

***图的存储表示（邻接表、邻接矩阵）

*****图的遍历与连通性*****

***深度优先搜索（不同存储结构下的递归与非递归解法）

***广度优先搜索

***连通分量的获取（利用图的遍历）

*****最小生成树*****

***克鲁斯卡尔算法

***普里姆算法

*****最短路径*****

***迪克斯特拉算法

***贝尔曼与福特算法

***弗洛伊德算法

*****活动网络*****

***拓扑排序

***关键路径

例 1、如果含 n 个顶点的图形成一个环，则它有 n 棵生成树。

有 10 个顶点的无向图，边的总数最多为 45。

G 是一个非连通无向图，共有 28 条边，则该图至少有 9 个顶点。

图 G 是非连通无向图，至少有两个连通分量；一个连通分量最少的顶点数是由 28 条边组成的无向完全图，其顶点数 n 可由 $e \le n(n-1)/2$，解之得 $n \ge 8$；另外一个顶点自成一个连通分量，所以该图至少有 9 个顶点。

例 2：某乡有 A, B, C, D 四个村庄，图中边上的权值 W_{ij} 即为从 i 村庄到 j 村庄间的距离。现在要在乡里建立中心俱乐部，其选址应使得离中心最远的村庄离俱乐部最近。

(1) 请写出各村庄之间的最短距离矩阵；

(2) 写出该中心俱乐部应设在哪个村庄，以及各村庄到中心俱乐部的路径和路径长度。

弗洛伊德算法

例 3：试给出判定一个图是否存在回路的方法。

(1) 利用拓扑排序算法可以判定图 G 是否存在回路。即，在拓扑排序输出结束后所余下的顶点均有前驱，则说明只得到了部分顶点的拓扑有序序列，AOV 网中存在着有向回路。

(2) 设 G 是 n 个顶点的无向连通图，若 G 的边数 e>=n，则 G 中一定有回路存在。因此，只要计算出 G 的边数，就可判定图 G 中是否存在回路。

(3) 设 G 是 n 个顶点的无向连通图，若 G 的每个顶点的度大于或等于 2，则图中一定存在回路。

(4) 利用深度优先遍历算法可以判定图 G 中是否存在回路。对无向图来说，若深度优先遍历过程中遇到了回边则必定存在环；对有向图来说，这条回边可能是指向深度优先森林中另一棵生成树上顶点的弧；但是，如果从有向图上的某个顶点 v 出发进行深度优先遍历，若在 DFS(v)结束之前出现一条从顶点 u 到顶点 v 的回边，因 u 在生成树上是 v 的子孙，则有向图必定存在包含顶点 v 和顶点 u 的环。

例 4：设计算法，求出无向连通图中距离顶点 v_0 的最短路径长度（最短路径长度以边数为单位计算）为 k 的所有结点，要求尽可能地节省时间。

算法中必须用广度优先遍历的层次性特性来求解，也即要在以 v_0 为起点调用 BFS 算法输出第 k+1 层上的所有顶点。因此，在访问顶点时需要知道层数，而每个顶点的层数是由其前驱决定的（起点除外）。所以，可以从第一个顶点开始，每访问到一个顶点就根据其前驱的层次计算该顶点的层次，并将层数值与顶点编号一起入队、出队。实际上可增加一个队列来保存顶点的层数值，并且将层数的相关操作与对应顶点的操作保持同步，即一起置空、出队和入队。

```
Bfs_klevel(adjlist g, int v0, int k)
{
    int i;
    Setnull(Q);     /*置队列 Q 为空*/
    Setnull(Q1);    /*置队列 Q1 为空*/
    for (i=1; i<=n; i++)        /*对 n 个顶点的每一个顶点访问标志初始化*/
        visited[i]=0;
    visited[0]=1;           /*对起点 v0 置访问过标志*/
    level=1;                /*置 v0 的层次数为 1*/
    Enqueue(Q, v0);     /*v0 顶点进入队列 Q*/
    Enqueue(Q1, level);     /*v0 的层次数进入队列 Q1*/
    while ((!Empty(Q)) && (level<k+1))   /*当队列非空且层次数小于 k+1 时*/
    {
        v=Qutqueue(Q);     /*队头顶点 v 出队*/
        level=Qutqueue(Q1);     /*队头顶点 v 的层次数出队*/
        w=findadj(g, v);            /*寻找 v 的邻接顶点赋给 w*/
        while (w!=0)            /*当邻接顶点 w 存在*/
        {
            if (visited[w]= =0)         /*如果 w 为未访问过顶点*/
            {
                if (level= =k)          /*如果 w 的层次数为 k 则输出*/
                    printf("Node=%d", w);
```

```
                visited[w]=1;        /*置顶点 w 为访问过标志*/
                Enqueue(Q, w);      /*w 入队*/
                Enqueue(Q1, level+1);        /*w 的层次数入队*/
            }
            w=Findadj(g, v);        /*寻找 v 的下一个邻接点*/
        }
    }
}
```

九、 内排序

内排序

***插入排序

***交换排序

***选择排序

***归并排序

***基数排序

**比较各种方法的时间复杂度与稳定性，及其实现方式。

例 1：设有 5000 个无序的元素，希望用最快速度挑出其中前 10 个最大的元素。在快速排序、堆排序、归并排序、基数排序和希尔排序方法中，采用哪一种方法最好？为什么？

所列几种排序方法的速度都很快，但快速排序、归并排序、基数排序和希尔排序都是在排序结束之后才能确定数据元素的全部顺序，而无法知道排序过程中部分元素的有序性。但堆排序则每次输出一个最小（或最大）的元素，然后对堆进行调整，保证堆顶的元素是未排序元素中的最小（或最大）的。因此，选取前 10 个最大元素采用堆排序方法最好。

例 2：对由 n 个元素组成的线性表进行快速排序时，所需进行的比较次数与这 n 个元素的初始排列有关。

(1) 当 n=7 时，在最好情况下需进行多少次比较？

(2) 当 n=7 时，给出一个最好情况的初始排列的实例。

(3) 当 n=7 时，在最坏情况下需进行多少次比较？

(4) 当 n=7 时，给出一个最坏情况的初始排列的实例。

(1) 在最好情况下，每一趟快速排序后均能划分出左、右两个相等的区间，即设线性表的长度 n=2k-1，则第一趟快速排序后划分得到两个长度均为 n/2 的子表，第二快速排序后划分得到 4 个长度为 n/4 的 子表，以此类推，总共需进行 $k=\log_2(n+1)$ 遍划分，即子表长度为 1 时排序完毕。因此，当 n=7 时 k=3，也即在最好情况下第一个元素由两头向中间扫描到正中位置，即需与其余 6 个元素都进行比较后找到最终存储位置，因此需要比较 6 次。第二趟分别对左、由两个子表（长度均为 3，即 k=2）进行排序，与第一趟类似，需与子表中其余 2 个元素进行比较后找到

其最终存储位置，也即两个子表共需比较 4 次，并且继续划分出的每个子表长度均为 1 ($\lfloor n/4 \rfloor = \lfloor 7/4 \rfloor = 1$)，即排序完毕。故总共需比较 10 次。

(2) 由(1)所知，每趟排序都应使第一个元素存储于表的正中位置，因此最好的初始排列的例子为：4, 7, 5, 6, 3, 1, 2。

(3) 快速排序最坏的情况是，每趟用来划分的基准元素总是定位于表的第一位置或最后一个位置，这样划分的左、右子表一个长度为 0，另一个仅是原表长减 1。这样，快速排序的效率蜕化为冒泡排序，其时间复杂度为 $O(n^2)$，即比较次数为 6+5+4+3+2+1=21 次。

(4) 由(3)可知，快速排序最坏的情况是初始序列有序。所以，当 n=7 时，最坏情况的初始排列的例子为：1, 2, 3, 4, 5, 6, 7 或者 7, 6, 5, 4, 3, 2, 1。

例 3：某个待排序的序列是一个可变长度的字符串序列，这些字符串一个接一个地存储于唯一的字符数组中。请改写快速排序算法，对这个字符串序列进行排序。

```
void Quicksort(String A[ ],int low, int high)
{
    int i, j;
    String x;
    if (low<high)
    {
        i=low;
        j=high;
        x=A[1];
        do
        {
            while ((i<j) && StringComp(A[i], x))
                j--;
            if (i<j)    /*A[j]左移到 A[i]*/
            {
                A[i]=A[j];
                i++;
            }
            while ((i<j) && StringComp(A[i], x))
                i++;
            if (i<j)    /*A[i]右移到 A[j]*/
            {
                A[j]=A[i];
                j--;
            }
        }while (i!=j);        /*划分结束*/
        A[i]=x;        /*将存放在 x 中的串移到 A[i]中*/
        Quicksort(A, low, i-1);        /*对左区间 A[low..i-1]进行快速排序*/
        Quicksort(A, i+1, high);       /*对右区间 A[i+1..high]进行快速排序*/
    }
```

}

十、 散列

散列

***散列函数

**几种散列函数的定义方式及实现

***处理溢出的闭散列方法

**线性探测、二次探查法、双散列法

***处理溢出的开散列方法

**链地址法

***散列表分析

**装载因子

例 1：使用散列函数 hash f(x)=x%11，把一个整数值转换成散列表下标，现要把数据 1, 13, 12, 34, 38, 33, 27, 22 插入到散列表中。

(1) 使用线性探查再散列法来构造散列表。

(2) 使用链地址法来构造散列表。

(3) 针对这种情况，确定其装填因子，查找成功所需的平均查找次数以及查找不成功所需的平均探查次数。

(1) 首先计算出每个数据的 hash 地址如下：

f(1)=1, f(13)=2, f(12)=1, f(34)=1, f(38)=5, f(33)=0, f(27)=5, f(22)=0

使用线性探查再散列法构造的散列表：

地址	0	1	2	3	4	5	6	7	8	9	10
数据	33	1	13	12	34	38	27	22			
探查成功次数	1	1	1	3	4	1	2	8			
探查不成功次数	9	8	7	6	5	4	3	2	1	1	1

(2) hash 表长度为 m，填入表中的数据个数为 n，则装填因子 α=n/m；因此有：α=8/11。

在等概率情况下，

使用线性探查再散列法查找成功情况时所需的平均查找次数为：

$ASL_{成功}=(1+1/(1-\alpha))/2=(1+1(1-8/11))/2=7/3$

使用线性探查再散列法查找不成功情况时所需的平均查找次数为：

$ASL_{不成功}=(1+1/(1-\alpha)^2)/2=(1+1(1-8/11)^2)/2=65/9$

使用链地址法查找成功情况时所需的平均查找次数为：

$ASL_{成功}=1+\alpha/2=1+8/22=15/11$

使用链地址法查找不成功情况时所需的平均查找次数为：

$ASL_{不成功}=\alpha+e^{-\alpha}=8/11+e^{-8/11}$

***注意，利用装填因子所计算的平均查找长度为近似值。

也可直接计算如下：

线性探查再散列法——

ASL $_{成功}$=1/8*(1+1+1+3+4+1+2+8)=21/8

ASL $_{不成功}$=1/11(9+8+7+6+5+4+3+2+1+1+1)=47/11

链地址法查找成功时为链表中找到该结点时的比较次数，查找不成功时必须比较到链尾的空指针处，因此有：

ASL $_{成功}$=1/8*(1+2+1+3+1+1+2)=13/8

ASL $_{不成功}$=1/11(3+4+2+1+1+3+1+1+1+1+1)=19/11