利用一个栈实现以下递归函数的非递归计算

$$P_n(x) \begin{cases} 1 & n=0 \\ 2x & n=1 \\ 2xP_{n-1}(x)-2(n-1)P_{n-2}(x) & n>1 \end{cases}$$

```
double p ( int n, double x ){
   struct stack{
      int no ;
      double val ;
   }st[Maxsize] ;
   int top=-1 ;
   double fv1=1, fv2=2*x ;
   for ( i=n; i>=2; i-- )
      top++ ;
      st[top] .no=i ;
   }
   while ( top>=0 ){
      st[top] .val=2*x*fv1-2*( st[top] .no-1 )*fv1 ;
      fv1=fv2 ;
      fv2=st[top].val ;
      top-- ;
}
   if ( n=0 )
      return fv1 ;
   return fv2 ;
}
```

计算二叉树中所有结点个数

法一：

```
int count ( BTNode *p ){
   int n1, n2 ;
   if ( p==NULL )
      return 0 ;
   else {
      n1=count ( p->lchild ) ;
      n2=count ( p->rchild ) ;
      return n1+n2+1 ;
   }
}
```

法二：

```
int n=0 ;
void count ( BTNode *p ){
   if ( p !=NULL ){
      ++n ;
      count ( p->lchild ) ;
      count ( p->rchild ) ;
   }
}
```

计算二叉树中所有叶子节点的个数

法一：

```
int count ( BTNode *p ){
   int n1, n2 ;
   if ( p==NULL )
      return 0 ;
  if( p->lchild==NULL && p->rchild==NULL)
      return 1 ;
  else {
      n1=count ( p->lchild ) ;
      n2=count ( p->rchild ) ;
      return n1+n2 ;
   }
}
```

法二：

```
int n=0 ;
void count ( BTNode *p ){
   if ( p !=NULL ){
    if( p->lchild==NULL&&p->rchild==NULL)
       ++n ;
    count ( p->lchild ) ;
    count ( p->rchild ) ;
   }
}
```

(a-(b+c))*(d/e)存储在二叉树，遍历求值

```
int comp ( BTNode *p ){
   int A, B ;
   if ( p !=NULL ){
    if(p->lchild !=NULL&&p->rchild !=NULL)
      {
         A=comp ( p->lchild ) ;
         B=comp ( p->rchild ) ;
         return op ( A, B, p->data ) ;
      }
      else
         return p->data-‘0’ ;
   }
   else
      return 0 ;
} //本道题目的图在天勤上
```

计算二叉树的深度

```
int getDepth ( BTNode *p ){
   int LD, RD ;
   if ( p==NULL )
      return 0 ;
   else {
      LD=getDepth ( p->lchild ) ;
      RD=getDepth ( p->rchild ) ;
      return ( LD>RD ? LD : RD )+1 ;
   }
}
```

判断两个二叉树是否相似(指都为空或者都只有一个根节点，或者左右子树都相似)

```
int fun ( BTNode *T1, BTNode *T2 ){
   int left, right ;
   if ( T1==NULL && T2==NULL )
      return 1 ;
   if ( T1==NULL || T2==NULL )
      return 0 ;
   else {
      right=fun ( T1->rchild, T2->rchild ) ;
      left=fun ( T1->lchild, T2->lchild ) ;
      return left && right ;
   }
}
```

把二叉树所有节点左右子树交换

```
void swap ( BTNode *p ){
   if ( p !=NULL ){
      swap ( p->lchild ) ;
      swap ( p->rchild ) ;
      temp=p->lchild ;
      p->lchild=p->rchild :
      p->rchild=temp ;
   }
}
```

查找二叉树中 data 域等于 key 的结点是否存在，若存在，将 q 指向它，否则 q 为空

```
void fun (BTNode *p, BTNode *&q, int key){
   if ( p !=NULL ){
      if ( p->data==key )
         q=p;
      else {
         fun (p->lchild, q, key ) ;
         fun (p->rchild, q, key ) ;
      }
   }
}
```

输出先序遍历第 k 个结点的值

```
int n=0 ;
void trave ( BTNode *p, int k ){
   if ( p !=NULL ){
      ++n ;
      if ( k==n ){
         printf ( “%d”, p->data ) ;
         return ;
      }
      trave ( p->lchild, k ) ;
      trave ( p->rchild, k ) ;
   }
}
```

利用结点的右孩子指针将一个二叉树的叶子节点从左向右连接成一个单链表（head 指向第一个，tail 指向最后一个）

```
void link ( Bt *p, Bt*&head, Bt *&tail ){
   if ( p !=NULL ){
      if ( ! p->lchild && ! p->rchild ){
         if ( head==NULL ){
            head=p ;
            tail=p ;
         }
         else {
            tail->rchild=p ;
            tail=p ;
         }
      }
      link ( p->lchild, head, tail ) ;
      link ( p->rchild, head, tail ) ;
   }
}
```

增加一个指向双亲节点的 parent 指针，并给指针赋值，并输出所有节点到根节点的路径

```
typedef Struct BTNode {
   char data ;
   Struct BTNode *parent, lchild, rchild ;
}BTNode ;
void fun ( BTNode *p, BTNode *q ){
   if ( p !=NULL ){
      p->parent=q ;
      q=p ;
      fun ( p->lchild, q ) ;
      fun ( p->rchild, q ) ;
   }
}
void printpath ( BTNode *p ){
   while ( p !=NULL ){
      printf ( "%c", p->data ) ;
      p=p->parent ;
   }
}
void allpath ( BTNode *p ){
   if ( p !=NULL ){
      printpath (p) ;
      allpath (p->lchild ) ;
      allpath (p->rchild ) ;
   }
}
```

求二叉树中值为 x 的层号

```
int L1=1 ;
void fun ( BTNode *p, int x ){
   if ( p !=NULL ){
      if ( p->data==x )
         printf ( "%d", L1 ) ;
      ++L1 ;
      fun ( p->lchild, x ) ;
      fun ( p->rchild, x ) ;
      --L1 ;
   }
}
```

输出根节点到每个叶子结点的路径

```
int i, top=0 ;
char pathstack [maxsize] ;
void allpath ( BTNode *p ){
   if ( p !=NULL ){
      pathstack[top]=p->data ;
      ++top ;
      if ( ! p->lchild && ! p->rchild )
         for ( i=0; i<top; ++i )
            printf ( " %c ", pathstack[i] ) ;
      allpath ( p->lchild ) ;
      allpath ( p->rchild ) ;
      --top ;
   }
}
```

已知满二叉树先序序列存在于数组中，设计算法将其变成后序序列

```
void change ( char pre[ ], int L1, int R1, char post[ ], int L2, int R2 ){
    if ( L1<=R1 ){
        post[ R2 ]=pre[ L1 ] ;
        change ( pre, L1+1, (L1+R1+1)/2, post, L2, (L2+R2-1)/2 ) ;
        change ( pre, (L1+R1+1)/2+1, R1, post, (L2+R2-1)/2+1, R2-1 ) ;
    }
}
```

假设先序遍历与中序遍历分别存在两个一维数组 A，B 中，编写算法建立二叉链表

```
BiTree creat ( int A[ ], int B[ ], int L1, int R1, int L2, int R2 ){
    root=(BTNode *) malloc ( sizeof (BTNode) ) ;
    root->data=A[L1] ;
    for ( i=L2; B[i] !=root->data; i++ ) ;
    if ( i >L2 )
        root->lchild=creat ( A, B, L1+1, i-L2+L1, L2, i-1 ) ;
    else
        root->lchild=NULL ;
    if ( i <R2 )
        root->rchild=creat ( A, B, i-L2+L1+1, R1, i+1, R2 ) ;
    else
        root->rchild=NULL ;
    return root ;
}
```

找出二叉树中最大值的点

```
void Max ( BTNode *p, int &max ){
    if ( p !=NULL ){
        if ( p->data>max )
            max=p->data ;
        Max ( p->lchild, max ) ;
        Max ( p->rchild, max ) ;
    }
}
```

一颗二叉树以顺序方式存在，数组 A 的 n 个元素中，设计算法以二叉链表表示

```
void creat ( BTNode *&t, char A[ ], int i, int j ){
    if ( i > j )
        t=NULL ;
    else {
        t=( BTNode *) malloc( sizeof (BTNode) ) ;
        t->data=A[i] ;
        create ( t->lchild, A, 2*i, n ) ;
        create ( t->rchild, A, 2*i+1, n ) ;
    }
}
```

先序非递归遍历二叉树

```
void preNonrecursion ( BTNode *bt ){
   if ( bt !=NULL ){
      BTNode *Stack[maxsize] ;
      int top=-1 ;
      BTNode *p ;
      Stack [++top]=bt ;
      while ( top !=-1 ){
         p=Stack[top--] ;
         visit (p) ;
         if ( p->rchild !=NULL )
            Stack[++top]=p->rchild ;
         if ( p->lchild !=NULL )
            Stack[++top]=p->lchild ;
      }
   }
}
```

中序非递归遍历二叉树

```
void inNonrecursion ( BTNode *bt ){
   if ( bt !=NULL ){
      BTNode *Stack[maxsize] ;
      int top=-1 ;
      BTNode *p=bt ;
      while ( top !=-1 || p !=NULL ){
         while ( p !=NULL ){
            Stack[++top]=p ;
            p=p->lchild ;
         }
         if ( top !=-1 ){
            p=Stack[top--] ;
            visit (p) ;
            p=p->rchild ;
         }
      }
   }
}
```

后序非递归遍历二叉树

```
void postNonrecursion (BTNode *bt ){
   InitStack (s) ;
   BTNode *p=bt, r=NULL ;
   while ( p!=NULL || ! IsEmpty (s) ){
      if ( p !=NULL ) {
         push (s, p) ;
         p=p->lchild ;
      }
      else {
         GetTop (s, p) ;   //取栈顶结点
         if ( p->rchild && p->rchild !=r ){
            p=p->rchild ;
            push (s, p) ;
            p=p->lchild ;
         }
         else {
            pop (s, p) ;
            visit ( p ) ;
            r=p ;
            p=NULL ;
         }
      }
   }    //while
}
```

后序非递归遍历二叉树（法二）

```
void postorderNonrecursion(BTNode *bt ){
   if ( bt !=NULL ){
      BTNode *Stack1[maxsize] ;
      BTNode *Stack2[maxsize] ;
      int top1=top2=-1 ;
      BTNode *p=NULL ;
      Stack1[++top1]=bt ;
      while ( top1 !=-1 ){
         p=Stack1[top1--] ;
         Stack2[++top2]=p ;
         if ( p->lchild !=NULL )
            Stack1[++top1]=p->lchild ;
         if ( p->rchild !=NULL )
            Stack1[++top1]=p->rchild ;
      }
      while ( top2 !=-1 ){
         p=Stack2[top2--] ;
         visit (p) ;
      }
   }
}
```

在二叉树中查找值为 x 的结点，打印出值为 x 的所有祖先

```
void Search (BTNode *bt, int x ){
   Stack Stack [ ] ;
   int top=0 ;
   while ( bt !=NULL || top>0 ){
      while ( bt !=NULL && bt->data !=x ){
         Stack [++top].t=bt ;
         Stack [top].tag=0 ;
         bt=bt->lchild ;
      }
      if ( bt->data==x ){
         for ( int i=1; i<=top; ++i )
            printf( “%d “, Stack[ i ].t->data ) ;
         exit (1) ;
      }
      while (top !=0 && Stack[top].tag==1 )
         top-- ;
      if ( top !=0 ){
         Stack[top].tag=1 ;
         bt=Stack[top].t->rchild ;
      }
   }
}
```

找到 p 和 q 最近公共祖先结点 r

```
typedef struct{
   BTNode *t ;
   int tag ;
} Stack ; //前一题也加上这一结构体
Stack s[ ], s1[ ] ;
BTNode fun(BT *Root, BT *p, BT *q){
   int top=0 ;
   BTNode *bt=Root ;
   while ( bt !=NULL || top>0 ){
      while ( bt !=NULL ){
         s[++top].t=bt ;
         s[top].tag=0 ;
         bt=bt->lchild ;
      }
      while ( top !=0 && S[top].tag==1 ){
         if ( s[top].t==p ){
            for ( i=1; i<=top; ++i )
               s1[i]=s[i] ;
               top1=top ;
         }
         if ( s[top].t==q )
            for ( i=top; i>0; --i )
               for ( j=top1; j>0; j-- )
                  if ( s1[j].t==s[i].t )
                     return s[i].t ;
         top-- ;
     }
       if ( top !=0 ){
          s[top] .tag=1 ;
          bt=s[top].t->rchild ;
       }
   }
   return NULL ;
}
```

层次遍历

```
void level ( BTNode * p ){
   int front=rear=0 ;
   BTNode *que[maxsize] ;
   BTNode *q ;
   if ( p !=NULL ){
      rear=(rear+1) %maxsize ;
      que[rear]=p ;
      while ( front !=rear ){
         front=(front+1) %maxsize ;
         q=que[front] ;
         visit (q) ;
         if ( q->lchild !=NULL ){
            rear=(rear+1) % maxsize ;
            que[rear]=q->rchild ;
         }
         if ( q->rchild !=NULL ){
            rear=(rear+1) % maxsize ;
            que[rear]=q->lchild ;
         }
      }
   }
}
```

求解二叉树的宽度

```
typedef struct {
   BTNode *p ;
   int lno ;
} St ;
int maxNode ( BTNode *boot ){
   St que[maxsize] ;
   int front=rear=0 ;
   int lno, i, j, n, max ;
   if ( boot !=NULL ){
      que[++rear].p=boot ;
      que[rear].lno=1 ;
      while ( front !=rear ){
         BTNode *q=que[++front] .p ;
         Lno=que[front].lno ;
         if ( q->lchild !=NULL ){
            que[++rear].p=q->lchild ;
            que[rear].lno=Lno+1 ;
         }
         if ( q->rchild !=NULL ){
            que[++rear].p=q->rchild ;
            que[rear].lno=Lno+1 ;
         }
      }
      max=0 ;
      for ( i=1; i<=lno; ++i ){
         n=0;
         for ( j=1; j<=rear; ++j )
            if ( que[j].lno==i )
               ++n ;
         if ( max<n )
            max=n ;
      } //for
      return max ;
   ) //if
   else return 0 ;   // if ( boot !=NULL )
}
```

试给出自下而上从右到左的层次遍历

```
void fun ( BTNode *p ){
   Stack s ;   Queue Q ;
   if ( p !=NULL ){
      InitStack (s) ;
      InitQueue (Q) ;
      EnQueue (Q, p) ;
      while ( IsEmpty (Q)==false ) {
         DeQueue (Q, p) ;
         Push (s, p) ;
         if ( p->lchild )
            EnQueue ( Q, p->lchild ) ;
         if ( p->rchild )
            EnQueue ( Q, p->rchild ) ;
      }
      while ( IsEmpty (s)==false ){
         Pop (s, p) ;
         visit ( p->data ) ;
      }
   }
}
```

用层次遍历求解二叉树的高度

```
int Btdepth ( BTNode *boot ){
   if ( boot==NULL )
      return 0 ;
   int front=-1, rear=-1 ;
   int last=0, level=0 ;
   BTNode *Q[maxsize] ;
   Q[++rear]=boot ;
   BTNode *p ;
   while ( front<rear ){
      p=Q[++front] ;
      if ( p->lchild )
         Q[++rear]=p->lchild ;
      if ( p->rchild )
         Q[++rear]=p->rchild ;
      if ( front==last ){
         level++ ;
         last=rear ;
      }
   }
   return level ;
}
```

判断二叉树是否为完全二叉树

```
bool fun ( BTNode *p ){
   Init Queue (Q) ;
   if ( p==NULL )
      return 1 ;
   EnQueue (Q, p) ;
   while ( ! IsEmpty (Q) ){
      DeQueue (Q, p) ;
      if (p !=NULL){
         EnQueue ( Q, p->lchild ) ;
         EnQueue ( Q, p->rchild ) ;
      }
      else
         while ( ! IsEmpty (Q) ){
            DeQueue (Q, p) ;
            if ( p !=NULL )
               return 0 ;
         }
   }
   return 1 ;
}
```

对树中每个元素为 x 的结点，删除以它为根的子树，并释放相应空间

```
void delete ( BTNode *bt ){
   if ( bt !=NULL ){
      delete ( bt->lchild ) ;
      delete ( bt->rchild ) ;
      free (bt) ;
   }
}
void Search ( BTNode *bt, int x ){
   BTNode *Q[ ] ;
   if ( bt !=NULL ){
      if ( bt->data==x ){
      delete (bt) ;
      exit (0) ;
      }
   InitQueue (Q) ;
   EnQueue (Q, p) ;
   while ( ! IsEmpty (Q) ){
      DeQueue (Q, p) ;
      if ( p->lchild !=NULL )
         if ( p->lchild->data==x ){
            delete ( p->lchild ) ;
            p->lchild=NULL ;
         else
            EnQueue ( Q, p->lchild ) ;
      if (右子树同上)
```

计算二叉树的带权路径长度（叶子节点）

```
int fun ( BTNode *root ){
   BTNode * que[maxsize] ;
   int front=rear=wp1=deep=0 ;
   BTNode *last, new ;
   last=root ;   new=NULL ;
   que[rear++]=root ;
   while ( front !=rear ){
      BTNode *t=que[front++] ;
      if( ! t->lchild && ! t->rchild )
         wp1+=deep*t->weigh ;
      if ( t->lchild !=NULL ){
         que[rear++]=t->lchild ;
         new=t->lchild ;
      }
      if ( t->rchild !=NULL ){
         que[rear++]=t->rchild ;
         new=t->rchild ;
      }
      if ( t==last ){
         last=new ;
         deep+=1 ;
      }
 }
   return wp1 ;
}
```

法二:

```
int fun ( BTNode *p, int deep ){
   int wp1=0 ;
   if ( ! p->lchild && ! p->rchild ){
      wp1+=deep*p->weigh ;
   if ( p->lchild !=NULL )
      fun ( p->lchild, deep+1 ) ;
   if ( p->rchild !=NULL )
      fun ( p->rchild, deep+1 ) ;
   return wp1 ;
}
```

中序遍历线索二叉树

```
TBTNode *First ( TBTNode *p){
   while ( p->ltag==0 )
      p=p->lchild ;
   return p ;
}
TBTNode *Next ( TBTNode *p){
   if ( p->rtag==0)
      return First ( p->rchild ) ;
   else
      return p->rchild :
}
```

通过中序遍历对二叉树线索化的递归算法

```
void InThread (TBT *p, TBT *&pre){
   if ( p != NULL ){
      InThread ( p->lchild, pre ) ;
      if ( p->lchild==NULL ){
         p->lchild=pre ;
         p->ltag=1 ;
      }
      if ( pre && pre->rchild==NULL ){
         pre->rchild=p ;
         pre->rtag=1 ;
      }
      pre=p ;
      InThread (p->rchild, pre) ;
   }
}
void creat ( TBTNode *root ){
   TBTNode *pre=NULL ;
   if ( root !=NULL ){
      InThread ( root, pre ) ;
      pre->rchild=NULL ;
      pre->rtag=1 ;
   }
}
```

编写算法，判断给定二叉树是否是二叉排序树

```
keyType pre=-32151；
int judgeBST ( BTNode *bt ){
   int b1, b2；
   if ( bt==NULL )
      return 1；
   else{
      b1=judgeBST ( bt->lchild )；
      if ( b1==0 || pre>=bt->data )
         return 0；
      pre=bt->data；
      b2=judgeBST ( bt->rchild )；
      return b2；
   }
}
```

设求出指定结点在给定二叉排序树的层次

```
int level ( BTNode *bt, BTNode *p ){
   int n=0；
   BTNode *t=bt；
   if ( bt !=NULL ){
      ++n；
      while ( t->data !=p->data ){
         if ( t->data<p->data )
            t=t->lchild；
         else
            t=t->rchild；
         ++n；
      }
   }
   return n；
}
```

利用二叉树遍历的思想判断一个二叉树是否为平衡二叉树

```
void fun (BTNode *bt, int &balance, int &h)
{
   int bl=br=hl=hr=0；
   if ( bt==NULL ){
      h=0；
      balance=1；
   }
   else if ( ! bt->lchild && ! bt->rchild )
      h=1；
      balance=1；
   }
   else{
      fun ( bt->lchild, bl, hl )；
      fun ( bt->rchild, br, hr )；
      h=( hl>hr ? hl:hr )+1；
      if ( abs(hl-hr)<2 )
         balance=bl&&br；
      else
         balance=0；
   }
}
```

直接插入排序

```
void InsertSort ( int R[ ], int n ){
   int i, j, temp ;
   for ( i=1; i<n; ++i ){
      temp=R[i] ;
      j=i-1 ;
      while ( j>=0 && temp<R[j] ){
         R[j+1]=R[j] ;
         --j ;
      }
      R[j+1]=temp ;
   }
}   //顺序存储
void Sort ( LNode *&L ){
   LNode *p=L->next ;
   LNode *r=p->next ;
   p->next=NULL ;
   p=r ;
   while ( p !=NULL ){
      r=p->next ;
      LNode *pre=L ;
      while ( pre->next!=NULL&&
               pre->next->data<p->data )
         pre=pre->next ;
      p->next=pre->next ;
      pre->next=p ;
      p=r ;   //链式存储
```

折半插入排序

```
void Insert ( int A[ ], int n ){
   int i, j ;
   int low, high, mid ;
   for ( i=2; i<=n; ++i ){
      A[0]=A[i] ;
      low=1 ;
      high=i-1 ;
      while ( low<=high ){
         mid=( low+high )/2 ;
         if ( A[mid] .key>A[0] .key )
            high=mid-1 ;
         else
            low=mid+1 ;
      }
      for ( j=i-1; j>=high+1; --j )
         A[j+1]=A[j] ;
      A[high+1]=A[0] ;
   }
}
```

希尔排序

```
void shellsort ( int arr[ ], int n ){
   int temp ;
   for ( int gap=n/2, gap>0; gap/=2 ){
      for ( int i=gap; i<n; ++i ){
         temp=arr[i] ;
         int j=i ;
         while (j>=gap&&arr[j-gap]>temp){
            arr[j]=arr[j-gap] ;
            j-=gap ;
         }
         arr[j]=temp ;
      }
   }
}
```

冒泡排序

```
void Bubblesort ( int R[ ], int n ){
   int i, j, flag, temp ;
   for ( i=n-1; i>=1; --i ){
      flag=0 ;
      for ( j=1; j<=1; ++j )
         if ( R[j-1]>R[j] ){
            temp=R[j] ;
            R[j]=R[j-1] ;
            R[j-1]=temp ;
            flag=1 ;
         }
      if ( flag==0 )
         return ;
   }
}
```

快速排序

```
void Quicksort ( int A[ ], int low, int high ){
   if ( low<high ){
      int pivotpos=part ( A, low, high ) ;
      Quicksort ( A, low, pivotpos-1 ) ;
      Quicksort ( A, pivotpos+1, high ) ;
   }
}
int part ( int A[ ], int low, int high ){
   int pivot=A[low] ;
   while ( low<high ){
      while ( low<high && A[high]>=pivot )
         --high ;
      A[low]=A[high] ;
      while ( low< high && A[low]<=pivot )
         ++low ;
      A[high]=A[low] ;
   }
   A[low]=pivot ;
   return low ;
}
```

选择排序

```
void SelectSort (int A[ ], int n ){
   int i, j, k, temp ;
   for ( i=0; i<n; ++i ){
      k=i ;
      for ( j=i+1; j<=i; ++j )
         if ( R[k]>R[j] )
            k=j ;
      temp=R[i] ;
      R[i]=R[k] ;
      R[k]=temp ;
   }
}
```

堆排序

```
void Sift (int arr[ ], int low, int high ){
   int i=low, j=2*i+1 ;
   int temp=arr[i] ;
   while ( j<=high ){
      if ( j<high && arr[j]<arr[j+1] )
      ++j ;
      if ( temp < arr[j] ){
         arr[i]=arr[j] ;
         i=j ;
         j=2*i+1 ;
      }
      else
         break ;
   }
   arr[i]=temp ;
}
void heapSort ( int arr[ ], int n ){
   for ( int i=n/2-1; i>=0; --i )
      sift (arr, i, n-1 ) ;
   for ( i=n-1; i>0; --i ){
      int temp=arr[0] ;
      arr[0]=arr[i] ;
      arr[i]=temp ;
      sift ( arr, 0, i-1 ) ;
 }
}
```

归并排序

```
int *B=(int *) malloc( (n+1)*sizeof (int) ) ;
void fun (int A[ ], int low, int mid, int high){
   int i=low, j=mid+1 ;
   for ( int k=low; k<=high; ++k )
       B[k]=A[k] ;
   for( int k=i; i<=mid && j<=high; ++k){
      if ( B[i] <=B[j] )
         A[k]=B[i++] ;
      else
         A[k]=B[j++] ;
   }
   while ( j<=high )
      A[k++]=B[j++] ;
   while ( i<=mid )
      A[k++]=B[i++] ;
}
void MergeSort ( int A[ ], int low, int high ){
   if ( low<high ){
      int mid =( low+high )/2 ;
      MergeSort ( A, low, mid ) ;
      MergeSort ( A, mid+1, high ) ;
      fun ( A, low, mid, high ) ;
   }
}
```

KMP 算法

```
void getnext ( str substr, int next[ ] ){
   int i=1, j=0 ;
   next[1]=0 ;
   while ( i<substr.length ){
   if ( j==0||substr.ch[i]==substr.ch[j] ){
         ++i, ++j ;
         next[i]=j ;
      }
      else
         j=next[j] ;
   }
}
int KMP ( Str str, Str substr, int next[ ] ){
   int i=j=1 ;
   while(i<=str.length&&j<=substr.length ){
      if ( j==0 || str .ch[i]==substr.ch[j] )
         ++i, ++j ;
      else
         j=next[j] ;
      if ( j >substr.length )
         return i-substr.length ;
      else
         return 0 ;
   }
}
```

(祝大家考研顺利)