```c
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAX_LENGTH_INSERT_SORT 7 /* 用于快速排序时判断是否选用插入排序阈值 */

typedef int Status;


#define MAXSIZE 10000  /* 用于要排序数组个数最大值，可根据需要修改 */
typedef struct
{
    int r[MAXSIZE+1];   /* 用于存储要排序数组，r[0]用作哨兵或临时变量 */
    int length;         /* 用于记录顺序表的长度 */
}SqList;

/* 交换L中数组r的下标为i和j的值 */
void swap(SqList *L,int i,int j)
{
    int temp=L->r[i];
    L->r[i]=L->r[j];
    L->r[j]=temp;
}

void print(SqList L)
{
    int i;
    for(i=1;i<L.length;i++)
        printf("%d,",L.r[i]);
    printf("%d",L.r[i]);
```

```
    printf("\n");
}

/* 对顺序表L作交换排序（冒泡排序初级版） */
void BubbleSort0(SqList *L)
{
    int i,j;
    for(i=1;i<L->length;i++)
    {
        for(j=i+1;j<=L->length;j++)
        {
            if(L->r[i]>L->r[j])
            {
                swap(L,i,j);/* 交换L->r[i]与L->r[j]的值 */
            }
        }
    }
}

/* 对顺序表L作冒泡排序 */
void BubbleSort(SqList *L)
{
    int i,j;
    for(i=1;i<L->length;i++)
    {
        for(j=L->length-1;j>=i;j--)  /* 注意j是从后往前循环 */
        {
            if(L->r[j]>L->r[j+1]) /* 若前者大于后者（注意这里与上一算法的差异）*/
            {
                swap(L,j,j+1);/* 交换L->r[j]与L->r[j+1]的值 */
            }
        }
```

```
    }
}

/* 对顺序表L作改进冒泡算法 */
void BubbleSort2(SqList *L)
{
    int i,j;
    Status flag=TRUE;           /* flag用来作为标记 */
    for(i=1;i<L->length && flag;i++) /* 若flag为true
说明有过数据交换，否则停止循环 */
    {
        flag=FALSE;             /* 初始为False */
        for(j=L->length-1;j>=i;j--)
        {
            if(L->r[j]>L->r[j+1])
            {
                swap(L,j,j+1); /* 交换L->r[j]与L-
>r[j+1]的值 */
                flag=TRUE;     /* 如果有数据交换，则
flag为true */
            }
        }
    }
}


/* 对顺序表L作简单选择排序 */
void SelectSort(SqList *L)
{
    int i,j,min;
    for(i=1;i<L->length;i++)
    {
        min = i;                            /* 将当前下标
定义为最小值下标 */
        for (j = i+1;j<=L->length;j++)/* 循环之后的数据
*/
```

```
        {
            if (L->r[min]>L->r[j])  /* 如果有小于当前最小值的关键字 */
                min = j;                    /* 将此关键字的下标赋值给min */
        }
        if(i!=min)                          /* 若min不等于i，说明找到最小值，交换 */
            swap(L,i,min);                  /* 交换L->r[i]与L->r[min]的值 */
    }
}

/* 对顺序表L作直接插入排序 */
void InsertSort(SqList *L)
{
    int i,j;
    for(i=2;i<=L->length;i++)
    {
        if (L->r[i]<L->r[i-1]) /* 需将L->r[i]插入有序子表 */
        {
            L->r[0]=L->r[i]; /* 设置哨兵 */
            for(j=i-1;L->r[j]>L->r[0];j--)
                L->r[j+1]=L->r[j]; /* 记录后移 */
            L->r[j+1]=L->r[0]; /* 插入到正确位置 */
        }
    }
}

/* 对顺序表L作希尔排序 */
void ShellSort(SqList *L)
{
    int i,j,k=0;
    int increment=L->length;
    do
```

```
    {
        increment=increment/3+1;/* 增量序列 */
        for(i=increment+1;i<=L->length;i++)
        {
            if (L->r[i]<L->r[i-increment])/*  需将L->r[i]插入有序增量子表 */
            {
                L->r[0]=L->r[i]; /*  暂存在L->r[0] */
                for(j=i-increment;j>0 && L->r[0]<L->r[j];j-=increment)
                    L->r[j+increment]=L->r[j]; /*  记录后移，查找插入位置 */
                L->r[j+increment]=L->r[0]; /*  插入 */
            }
        }
        printf("    第%d趟排序结果: ",++k);
        print(*L);
    }
    while(increment>1);

}


/* 堆排序********************************** */

/* 已知L->r[s..m]中记录的关键字除L->r[s]之外均满足堆的定义， */
/* 本函数调整L->r[s]的关键字,使L->r[s..m]成为一个大顶堆 */
void HeapAdjust(SqList *L,int s,int m)
{
    int temp,j;
    temp=L->r[s];
    for(j=2*s;j<=m;j*=2) /* 沿关键字较大的孩子结点向下筛选 */
    {
```

```
        if(j<m && L->r[j]<L->r[j+1])
            ++j; /* j为关键字中较大的记录的下标 */
        if(temp>=L->r[j])
            break; /* rc应插入在位置s上 */
        L->r[s]=L->r[j];
        s=j;
    }
    L->r[s]=temp; /* 插入 */
}

/*  对顺序表L进行堆排序 */
void HeapSort(SqList *L)
{
    int i;
    for(i=L->length/2;i>0;i--) /*  把L中的r构建成一个大根堆 */
         HeapAdjust(L,i,L->length);

    for(i=L->length;i>1;i--)
    {
         swap(L,1,i); /* 将堆顶记录和当前未经排序子序列的最后一个记录交换 */
         HeapAdjust(L,1,i-1); /*  将L->r[1..i-1]重新调整为大根堆 */
    }
}

/* ************************************** */


/* 归并排序********************************** */

/* 将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n] */
void Merge(int SR[],int TR[],int i,int m,int n)
{
    int j,k,l;
```

```
    for(j=m+1,k=i;i<=m && j<=n;k++) /* 将SR中记录由小到
大地并入TR */
    {
        if (SR[i]<SR[j])
            TR[k]=SR[i++];
        else
            TR[k]=SR[j++];
    }
    if(i<=m)
    {
        for(l=0;l<=m-i;l++)
            TR[k+l]=SR[i+l];         /* 将剩余的
SR[i..m]复制到TR */
    }
    if(j<=n)
    {
        for(l=0;l<=n-j;l++)
            TR[k+l]=SR[j+l];         /* 将剩余的
SR[j..n]复制到TR */
    }
}


/* 递归法 */
/* 将SR[s..t]归并排序为TR1[s..t] */
void MSort(int SR[],int TR1[],int s, int t)
{
    int m;
    int TR2[MAXSIZE+1];
    if(s==t)
        TR1[s]=SR[s];
    else
    {
        m=(s+t)/2;                  /* 将SR[s..t]平分为
SR[s..m]和SR[m+1..t] */
```

```
        MSort(SR,TR2,s,m);      /* 递归地将SR[s..m]归并为有序的TR2[s..m] */
        MSort(SR,TR2,m+1,t);    /* 递归地将SR[m+1..t]归并为有序的TR2[m+1..t] */
        Merge(TR2,TR1,s,m,t);   /* 将TR2[s..m]和TR2[m+1..t]归并到TR1[s..t] */
    }
}

/* 对顺序表L作归并排序 */
void MergeSort(SqList *L)
{
    MSort(L->r,L->r,1,L->length);
}

/* 非递归法 */
/* 将SR[]中相邻长度为s的子序列两两归并到TR[] */
void MergePass(int SR[],int TR[],int s,int n)
{
    int i=1;
    int j;
    while(i <= n-2*s+1)
    {/* 两两归并 */
        Merge(SR,TR,i,i+s-1,i+2*s-1);
        i=i+2*s;
    }
    if(i<n-s+1) /* 归并最后两个序列 */
        Merge(SR,TR,i,i+s-1,n);
    else /* 若最后只剩下单个子序列 */
        for(j =i;j <= n;j++)
            TR[j] = SR[j];
}

/* 对顺序表L作归并非递归排序 */
void MergeSort2(SqList *L)
{
```

```
    int* TR=(int*)malloc(L->length * sizeof(int));/*
申请额外空间 */
    int k=1;
    while(k<L->length)
    {
        MergePass(L->r,TR,k,L->length);
        k=2*k;/* 子序列长度加倍 */
        MergePass(TR,L->r,k,L->length);
        k=2*k;/* 子序列长度加倍 */
    }
}

/* ****************************************** */

/* 快速排序********************************** */

/* 交换顺序表L中子表的记录，使枢轴记录到位，并返回其所在位置
*/
/* 此时在它之前(后)的记录均不大(小)于它。 */
int Partition(SqList *L,int low,int high)
{
    int pivotkey;

    pivotkey=L->r[low]; /* 用子表的第一个记录作枢轴记录
*/
    while(low<high) /*  从表的两端交替地向中间扫描 */
    {
         while(low<high&&L->r[high]>=pivotkey)
            high--;
         swap(L,low,high);/* 将比枢轴记录小的记录交换到低
端 */
         while(low<high&&L->r[low]<=pivotkey)
            low++;
         swap(L,low,high);/* 将比枢轴记录大的记录交换到高
端 */
    }
```

```
    return low; /* 返回枢轴所在位置 */
}

/* 对顺序表L中的子序列L->r[low..high]作快速排序 */
void QSort(SqList *L,int low,int high)
{
    int pivot;
    if(low<high)
    {
            pivot=Partition(L,low,high); /*  将L->r[low..high]一分为二，算出枢轴值pivot */
            QSort(L,low,pivot-1);        /*  对低子表递归排序 */
            QSort(L,pivot+1,high);       /*  对高子表递归排序 */
    }
}

/* 对顺序表L作快速排序 */
void QuickSort(SqList *L)
{
    QSort(L,1,L->length);
}

/* ************************************** */

/* 改进后快速排序******************************** */

/* 快速排序优化算法 */
int Partition1(SqList *L,int low,int high)
{
    int pivotkey;

    int m = low + (high - low) / 2; /* 计算数组中间的元素的下标 */
    if (L->r[low]>L->r[high])
```

```
        swap(L,low,high);   /* 交换左端与右端数据，保证左端较小 */
    if (L->r[m]>L->r[high])
        swap(L,high,m);     /* 交换中间与右端数据，保证中间较小 */
    if (L->r[m]>L->r[low])
        swap(L,m,low);      /* 交换中间与左端数据，保证左端较小 */

    pivotkey=L->r[low]; /* 用子表的第一个记录作枢轴记录 */
    L->r[0]=pivotkey;  /* 将枢轴关键字备份到L->r[0] */
    while(low<high) /*  从表的两端交替地向中间扫描 */
    {
         while(low<high&&L->r[high]>=pivotkey)
            high--;
         L->r[low]=L->r[high];
         while(low<high&&L->r[low]<=pivotkey)
            low++;
         L->r[high]=L->r[low];
    }
    L->r[low]=L->r[0];
    return low; /* 返回枢轴所在位置 */
}

void QSort1(SqList *L,int low,int high)
{
    int pivot;
    if((high-low)>MAX_LENGTH_INSERT_SORT)
    {
        while(low<high)
        {
            pivot=Partition1(L,low,high); /*  将L->r[low..high]一分为二，算出枢轴值pivot */
            QSort1(L,low,pivot-1);      /*  对低子表递归排序 */
```

```
            /* QSort(L,pivot+1,high);       /*  对高子表递归排序 */
            low=pivot+1;    /* 尾递归 */
        }
    }
    else
        InsertSort(L);
}

/* 对顺序表L作快速排序 */
void QuickSort1(SqList *L)
{
    QSort1(L,1,L->length);
}

/* ************************************** */
#define N 9
int main()
{
   int i;

   /* int d[N]={9,1,5,8,3,7,4,6,2}; */
   int d[N]={50,10,90,30,70,40,80,60,20};
   /* int d[N]={9,8,7,6,5,4,3,2,1}; */

   SqList l0,l1,l2,l3,l4,l5,l6,l7,l8,l9,l10;

   for(i=0;i<N;i++)
     l0.r[i+1]=d[i];
   l0.length=N;
   l1=l2=l3=l4=l5=l6=l7=l8=l9=l10=l0;
   printf("排序前:\n");
   print(l0);

   printf("初级冒泡排序:\n");
   BubbleSort0(&l0);
```

```
    print(l0);

    printf("冒泡排序:\n");
    BubbleSort(&l1);
    print(l1);

    printf("改进冒泡排序:\n");
    BubbleSort2(&l2);
    print(l2);

    printf("选择排序:\n");
    SelectSort(&l3);
    print(l3);

    printf("直接插入排序:\n");
    InsertSort(&l4);
    print(l4);

    printf("希尔排序:\n");
    ShellSort(&l5);
    print(l5);

    printf("堆排序:\n");
    HeapSort(&l6);
    print(l6);

    printf("归并排序（递归）:\n");
    MergeSort(&l7);
    print(l7);

    printf("归并排序（非递归）:\n");
    MergeSort2(&l8);
    print(l8);

    printf("快速排序:\n");
    QuickSort(&l9);
```

```
    print(l9);

    printf("改进快速排序:\n");
    QuickSort1(&l10);
    print(l10);


     /*大数据排序*/
     /*
     srand(time(0));
     int Max=10000;
     int d[10000];
     int i;
     SqList l0;
     for(i=0;i<Max;i++)
         d[i]=rand()%Max+1;
     for(i=0;i<Max;i++)
         l0.r[i+1]=d[i];
     l0.length=Max;
     MergeSort(l0);
     print(l0);
     */
     return 0;
}
```