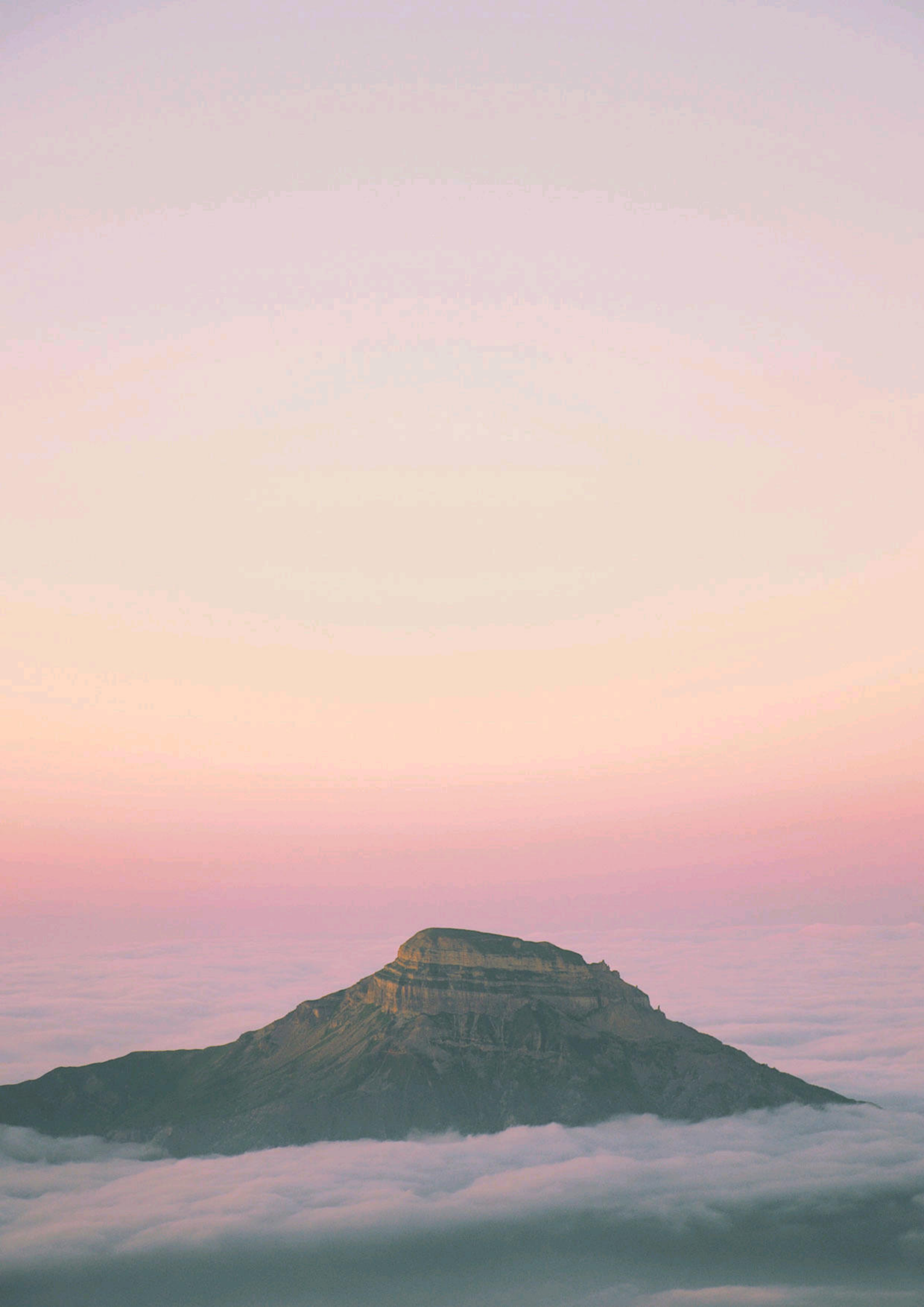

排序

一、排序的基本概念与分类.

1. 排序的稳定性

假设 $K_i=K_j$ $(1\le i\le n, 1\le j\le n, i\neq j)$ 且在排序前的序列中 r_i 领先于 r_j (即 $i<j$)。如果排序后 r_i 仍领先于 r_j，则称所用的排序方法是稳定的；反之，若可能使得排序后的序列中 r_j 领先于 r_i，则称其不稳定.

2. 内排序与外排序

内排序：在排序整个过程中，待排序的所有记录全部被放置在内存中。

外排序：由于排序的记录个数太多，不能同时放置在内存中，整个排序过程需要在内外存之间多次交换数据才能进行。

(1) 时间性能

内排序中：比较和移动

关键字比较次数；

记录移动次数.

(2) 辅助空间

(3) 算法的复杂性

内排序分为 插入排序、交换排序、选择排序、归并排序

3. 排序用到的结构和函数.

二、冒泡排序法

最基本的：

```
/* 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
            }
        }
    }
}
```

效率低

正常的冒泡排序

```
/* 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 */
            }
        }
    }
}
```

$O(n^2)$

改进：

```
/* 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则有数据交换，否则退出循环 */
    {
        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 */
            }
        }
    }
}
```

一趟排好则不用再排

三、简单选择排序

简单选择排序法（Simple Selection Sort）就是通过$n-i$次关键字间的比较，从$n-i+1$个记录中选出关键字最小的记录，并和第i（$1\leqslant i\leqslant n$）个记录交换。

我们来看代码：

```
/* 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 */
    }
}
```

比较：$\frac{n(n-1)}{2}$次

$O(n^2)$

四、直接插入排序

直接插入排序（Straight Insertion Sort）的基本操作是将一个记录插入到已经排好序的有序表中，从而得到一个新的、记录数增1的有序表。

顾名思义，从名称上也可以知道它是一种插入排序的方法。我们来看直接插入排序法的代码。

```
1  void InsertSort(SqList *L)          /* 对顺序表L作直接插入排序 */
2  {
3      int i,j;
4      for(i=2;i<=L->length;i++)
5      {
6          if (L->r[i]<L->r[i-1])      /* 需将L->r[i]插入有序子表 */
7          {
8              L->r[0]=L->r[i];        /* 设置哨兵 */
9              for(j=i-1;L->r[j]>L->r[0];j--)
10                 L->r[j+1]=L->r[j];  /* 记录后移 */
11             L->r[j+1]=L->r[0];      /* 插入到正确位置 */
12         }
13     }
14 }
```

$O(n^2)$

五、希尔排序

基本有序：小的关键字基本在前面，大的基本在后面。

希尔排序：将相距某个"增量"的记录组成一个子序列，

这样才能保障子序列内分别进行插入排序后得到的是基本有序

而不是局部有序

希尔排序算法

```
1  void ShellSort(SqList *L)
2  {                                                    /* 对顺序表L作希尔排序 */
3      int i,j,k=0;
4      int increment=L->length;
5      do
6      {
7          increment=increment/3+1;                     /* 增量序列 */
8          for(i=increment+1;i<=L->length;i++)
9          {
10             if (L->r[i]<L->r[i-increment])           /* 需将L->r[i]插入有序增量子表 */
11             {
12                 L->r[0]=L->r[i];                     /* 暂存在L->r[0] */
13                 for(j=i-increment;j>0 && L->r[0]<L->r[j];j-=increment)
14                     L->r[j+increment]=L->r[j];       /* 记录后移，查找插入位置 */
15                 L->r[j+increment]=L->r[0];           /* 插入 */
16             }
17         }
18     }
19     while(increment>1);
20 }
```

增量序列为 $dlta[k]=2^{t-k+1}-1 \quad 0\le k\le t\le \lfloor \log_2(n+1)\rfloor$ 时

时间复杂度为 $O(n^{\frac{3}{2}})$

△ 仍不稳定

六、堆排序

堆排序 的堆是完全二叉树 {
- 结点值 ≥ 左右孩子值 —— 大顶堆
- 结点值 < 左右孩子值 —— 小顶堆

堆排序代码

```
1  void HeapSort(SqList *L)          /* 对顺序表L进行堆排序 */
2  {
3      int i;
4      for(i=L->length/2;i>0;i--)    /* 把L中的r构建成一个大顶堆 */
5          HeapAdjust(L,i,L->length);
6      for(i=L->length;i>1;i--)
7      {
8          swap(L,1,i);              /* 将堆顶记录和当前未经排序子序列最后一记录交换 */
9          HeapAdjust(L,1,i-1);      /* 将L->r[1..i-1]重新调整为大顶堆 */
10     }
11 }
```

对有 孩子的结点 进行 操作

建堆函数：

```c
void HeapAdjust(SqList *L,int s,int m)
{ /* 本函数调整L->r[s]的关键字，使L->r[s..m]成为一个大顶堆 */
    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 */
}
```

调整位置.

```c
    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最后一记录交换 */
        HeapAdjust(L,1,i-1);  /* 将L->r[1..i-1]重新调整为大顶堆 */
    }
```

时间复杂度 $O(n\log n)$ { 建堆 n.
记录 n.
重建堆 $\log n$ >

不适合 数少的情况

七. 归并排序

1. 递归

2路归并，将初始序列看作 n 个有序的长度为1的子序列

两两归并，得到 $\lceil n/2 \rceil$ 个长度为2或1的有序子数列，...

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

```c
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] */
    }
}
```

→把TR2归并到TR1.

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

复杂度：

时间：

$O(n\log n)$

空间 $O(n+\log n)$

比较占用内存，但效率高且稳定。

2. 非递归

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

```c
void MergePass(int SR[],int TR[],int s,int n)
{/* 将SR[]中相邻长度为s的子序列两两归并到TR[] */
    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];
}
```

此空间复杂度为 $O(n)$，尽量用非递归算法

八. 快速排序

将待排记录分割成独立的两部分，其中一部分的关键字均比另一部分小，对两部分继续排序，以分使整个队列有序

1. 算法.

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

```c
/* 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)
    {
        /* 将L->r[low..high]一分为二，算出枢轴值pivot */
        pivot=Partition(L,low,high);
        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 */
    }
}
```

Partition() 函数 选取一个关键字 放在一个"合适的"位置，称为枢轴

```c
1  int Partition(SqList *L,int low,int high)
2  {/* 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)均不大(小)于它 *
3      int pivotkey;
4
5      pivotkey=L->r[low];      /* 用子表的第一个记录作枢轴记录 */
6      while(low<high)          /* 从表的两端交替地向中间扫描 */
7      {
8          while(low<high&&L->r[high]>=pivotkey)
9              high--;
10         swap(L,low,high);    /* 将比枢轴记录小的记录交换到低端 */
11         while(low<high&&L->r[low]<=pivotkey)
12             low++;
13         swap(L,low,high);    /* 将比枢轴记录大的记录交换到高端 */
14     }
15     return low;              /* 返回枢轴所在位置 */
16 }
```

时间复杂度 $O(n\log n)$

空间复杂度. $O(\log n)$

2.优化：　(1)优化选取枢轴

固定取第一个关键字为 pivotkey 不合理

采用 三数取中

```
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 */

/* 此时L.r[low]已经为整个序列左、中、右三个关键字的中间值 */

pivotkey=L->r[low];                      /* 用子表的第一个记录作枢轴记录 */
```

(2)优化不必要的交换

```
/* 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];       /* 采用替换而不是交换的方式进行操作 */
        while(low<high&&L->r[low]<=pivotkey)
            low++;
        L->r[high]=L->r[low];       /* 采用替换而不是交换的方式进行操作 */
    }
    L->r[low]=L->r[0];              /* 将枢轴数值替换回L.r[low] */
    return low;                     /* 返回枢轴所在位置 */
}
```

(3)优化小数组时的排序方案.

```c
#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 */
/* 对顺序表L中的子序列L.r[low..high]作快速排序 */
void QSort1(SqList *L,int low,int high)
{
    int pivot;
    if((high-low)>MAX_LENGTH_INSERT_SORT)
    {
        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 */
        QSort1(L,pivot+1,high);         /* 对高子表递归排序 */
    }
    else                                /* 当high-low小于等于常数时用直接插入排序 */
        InsertSort(L);
}
```

(4)优化递归操作

```c
/* 尾递归 */
void QSort2(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 */
            QSort2(L,low,pivot-1);        /* 对低子表递归排序 */
            low=pivot+1;                  /* 尾递归 */
        }
    }
    else
        InsertSort(L);                    /* 当high-low小于等于常数时用直接插入排序 */
}
```

九、总结

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n)\sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)\sim O(n)$	不稳定