10.3 快速排序

一、起泡排序（Bubble sort）

假设在排序过程中，记录序列R[1..n]的状态为：

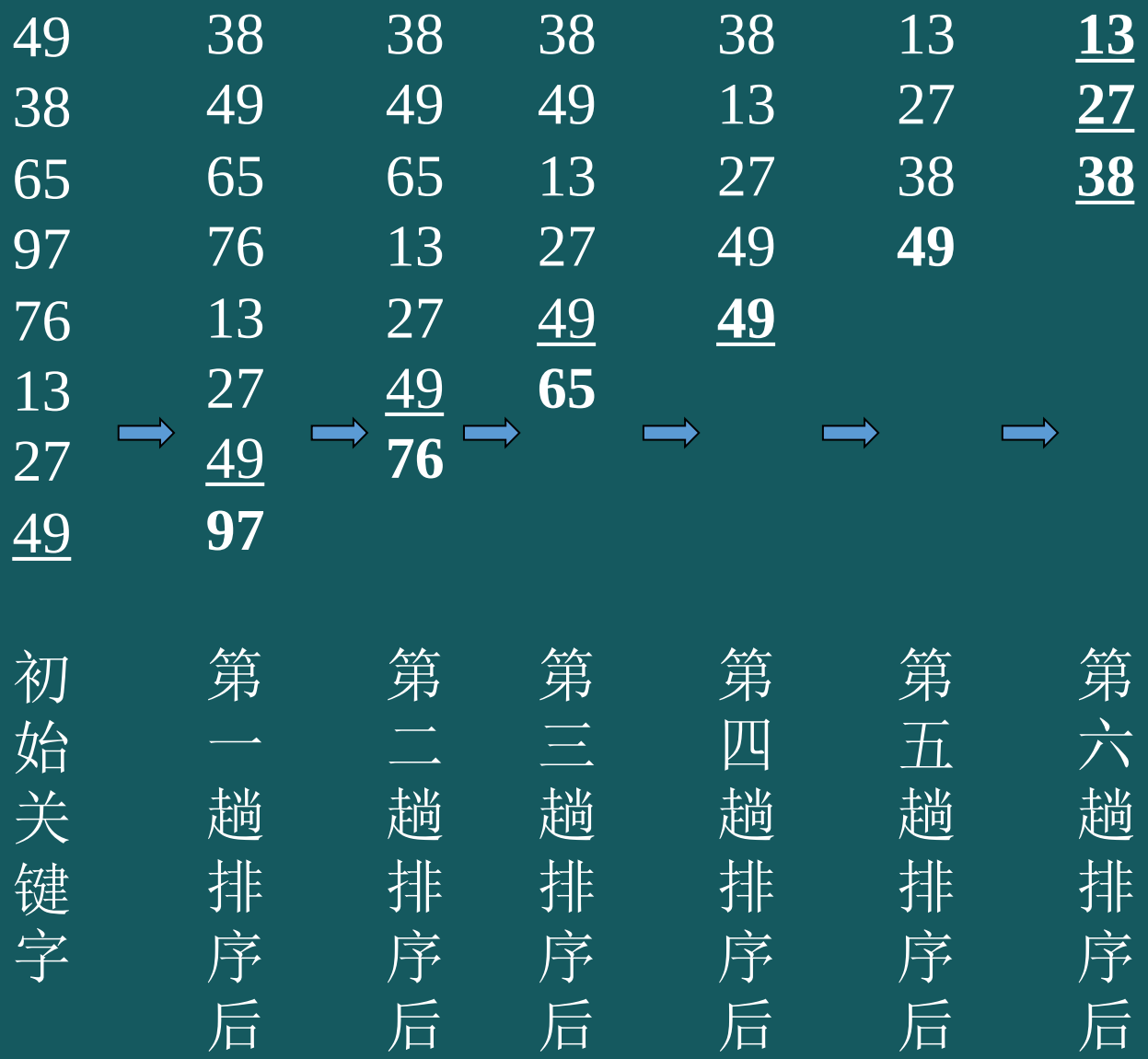

49
38
65
97
76
13
27
49
初始关键字
38
49
65
76
13
27
49
97
第一趟排序后
38
49
65
13
27
49
76
第二趟排序后
38
49
13
27
49
65
第三趟排序后
38
13
27
49
49
第四趟排序后
13
27
38
49
第五趟排序后
13
27
38
第六趟排序后

图 10.6 起泡排序示例

```
void BubbleSort(Elem R[ ], int n) {
   i = n;
   while (i >1) {
      lastExchangeIndex = 1;
       for (j = 1;  j < i;  j++)
          if (R[j+1].key < R[j].key) {
                Swap(R[j], R[j+1]);
             lastExchangeIndex = j;  //记下进行交换的记录位置
           } //if
       i = lastExchangeIndex; // 本趟进行过交换的
                                  // 最后一个记录的位置
   } // while
} // BubbleSort
```

起泡排序算法

注意:

1. 起泡排序的结束条件为，**最后一趟没有进行“交换记录”。**

2. 一般情况下，每经过一趟“起泡”，“i 减1”，但并不是每趟都如此。

例如:


```
for (j = 1;  j < i;  j++)  if (R[j+1].key < R[j].key) …
```

时间分析:

最好的情况（关键字在记录序列中顺序有序）：　只需进行一趟起泡

“比较”的次数：　　“移动”的次数：

n-1　　0

最坏的情况（关键字在记录序列中逆序有序）：　需进行n-1趟起泡

“比较”的次数：　　“移动”的次数：

$$\sum_{i=n}^{2}(i-1)=\frac{n(n-1)}{2}$$

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

二、快速排序（Quick sort）

目标：找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。

经过**一趟排序**之后，记录的无序序列R[s..t]将**分割成两部分**：R[s..i-1]和R[i+1..t]，且R[j].key≤ R[i].key ≤ R[p].key

$(s \le j \le i-1)$ $(i+1 \le p \le t)$。

枢轴

例如:

设 R[s]=52 为枢轴

将 R[high].key 和 枢轴的关键字进行比较，要求R[high].key ≥ 枢轴的关键字

将 R[low].key 和 枢轴的关键字进行比较，要求R[low].key ≤ 枢轴的关键字

可见，经过“**一次划分**”，将关键字序列

52, 49, 80, 36, 14, 58, 61, 97, 23, 75

调整为: **23**, 49, **14**, 36, **(52)** 58, 61, 97, **80**, 75

在调整过程中，设立了两个指针： **low** 和**high**，它们的初值分别为： s 和 t。

之后逐渐减小 **high**，增加 **low**，并保证

R[**high**].key≥52，和 R[**low**].key≤52,否则进行记录的“交换”。

```
int Partition (RedType& R[], int low, int high) {
  pivotkey = R[low].key;  //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  && R[high].key>=pivotkey)
      --high;    //将比枢轴记录小的记录交换到低端
    R[low]←→R[high];
    while (low<high  && R[low].key<=pivotkey)
      ++low;   //将比枢轴记录大的记录交换到高端
    R[low]←→R[high];
  }
  return low;          // 返回枢轴所在位置
} // Partition
```

算法 10.6 (a)

对上面算法进行分析，每交换一对记录需进行三次记录移动（赋值）的操作。而实际上，在排序过程中对枢轴记录的赋值是多余的，因为只有在一趟排序结束时，即low=high的位置才是枢轴记录的最后位置。

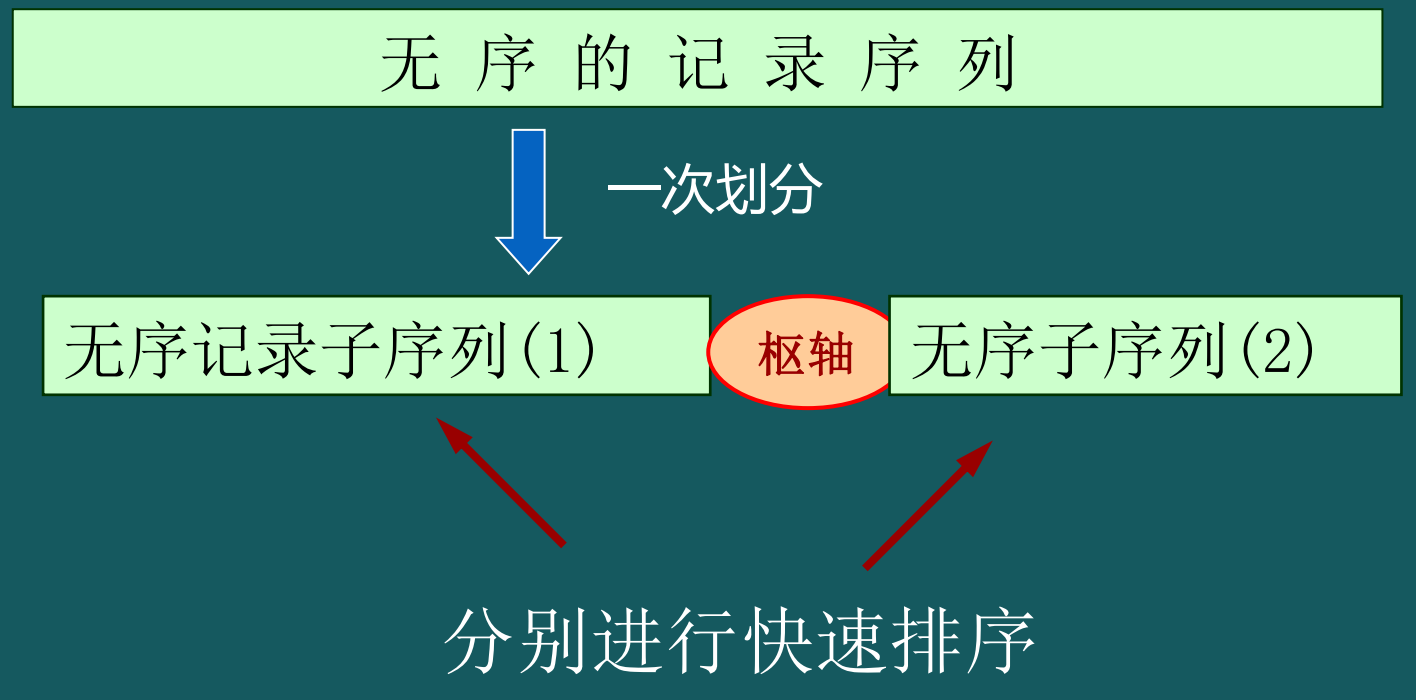
由此改写上述算法，先将枢轴记录暂存在R[0]的位置上，排序过程中只作 R[low]或R[high]的单向移动，直至一趟排序结束后再将枢轴记录移至正确位置上。如算法10.6（b）所示。

```
int Partition (RedType R[], int low, int high) {
   R[0] = R[low];  pivotkey = R[low].key;  // 枢轴
   while (low<high) {  //从表的两端交替地向中间扫描
         while(low<high&& R[high].key>=pivotkey)
              -- high;     //将比枢轴记录小的记录交换到低端
          R[low] = R[high];
         while (low<high &&R[low].key<=pivotkey)
              ++ low;     //将比枢轴记录大的记录交换到高端
          R[high] = R[low];
     }
     R[low] = R[0];     return low；//枢轴记录到位，返回
}// Partition                                  //枢轴位置
```

算法 10.6 (b)

一趟快速排序

首先对无序的记录序列进行“**一次划分**”，之后**分别**对分割所得两个子序列“**递归**”**进行快速排序。**



```
void QSort (RedType & R[],  int low,  int  high ) {
  // 对记录序列R[low..high]进行快速排序
  if (low < high) {              // 长度大于1
       pivotloc = Partition(R, low, high);
                          // 对 R[s..t] 进行一次划分
     QSort(R, low, pivotloc-1);
                 // 对低子序列递归排序，pivotloc是枢轴位置
        QSort(R, pivotloc+1, high); // 对高子序列递归排序
    }
} // QSort
```

算法 10.7

第一次调用函数 Qsort 时，待排序记录序列的上、下界分别为 1 和 L.length。

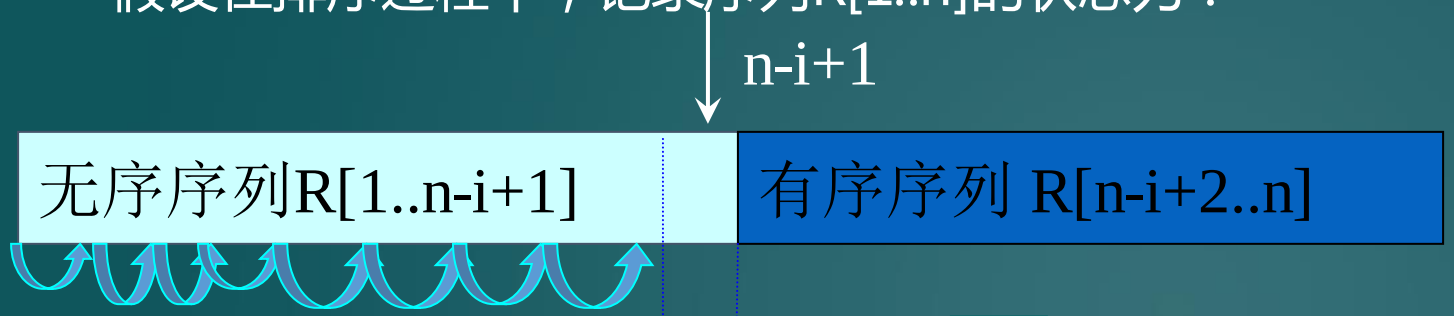
```
void QuickSort( SqList & L) {
  // 对顺序表进行快速排序
    QSort(L.r, 1, L.length);
} // QuickSort
```

算法 10.8

10.3 快速排序

一、起泡排序（Bubble sort）

假设在排序过程中，记录序列R[1..n]的状态为：



比较相邻记录，将**关键字最大的记录交换**到 **n-i+1** 的位置上

第 i 趟起泡排序

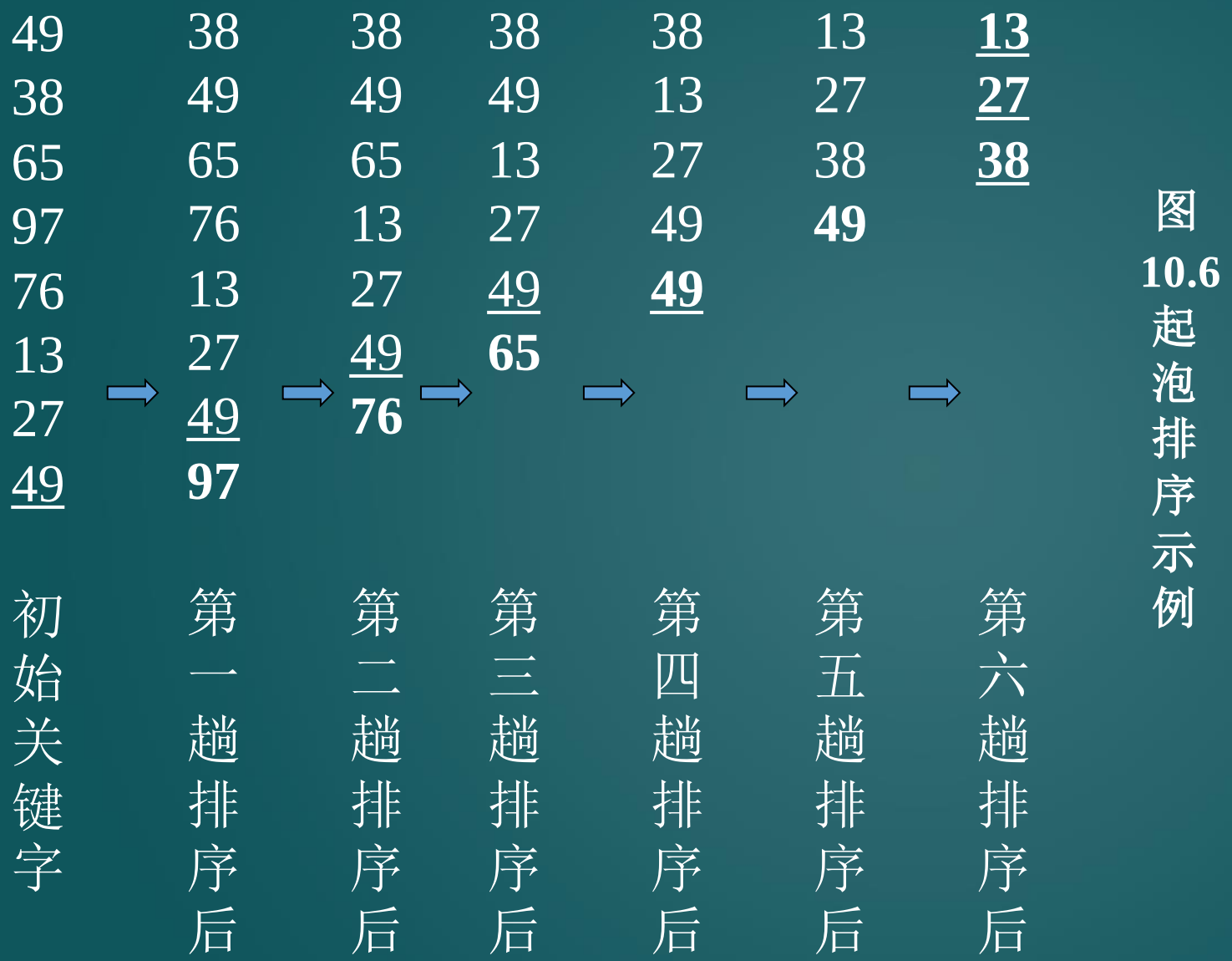

49
38
65
97
76
13
27
49
初始关键字
38
49
65
76
13
27
49
97
第一趟排序后
38
49
65
13
27
49
76
第二趟排序后
38
49
13
27
49
65
第三趟排序后
38
13
27
49
49
第四趟排序后
13
27
38
49
第五趟排序后
13
27
38
第六趟排序后

图 10.6 起泡排序示例

```
void BubbleSort(Elem R[ ], int n) {
   i = n;
   while (i >1) {
      lastExchangeIndex = 1;
       for (j = 1;  j < i;  j++)
          if (R[j+1].key < R[j].key) {
                Swap(R[j], R[j+1]);
             lastExchangeIndex = j;  //记下进行交换的记录位置
           } //if
       i = lastExchangeIndex; // 本趟进行过交换的
                                  // 最后一个记录的位置
   } // while
} // BubbleSort
```

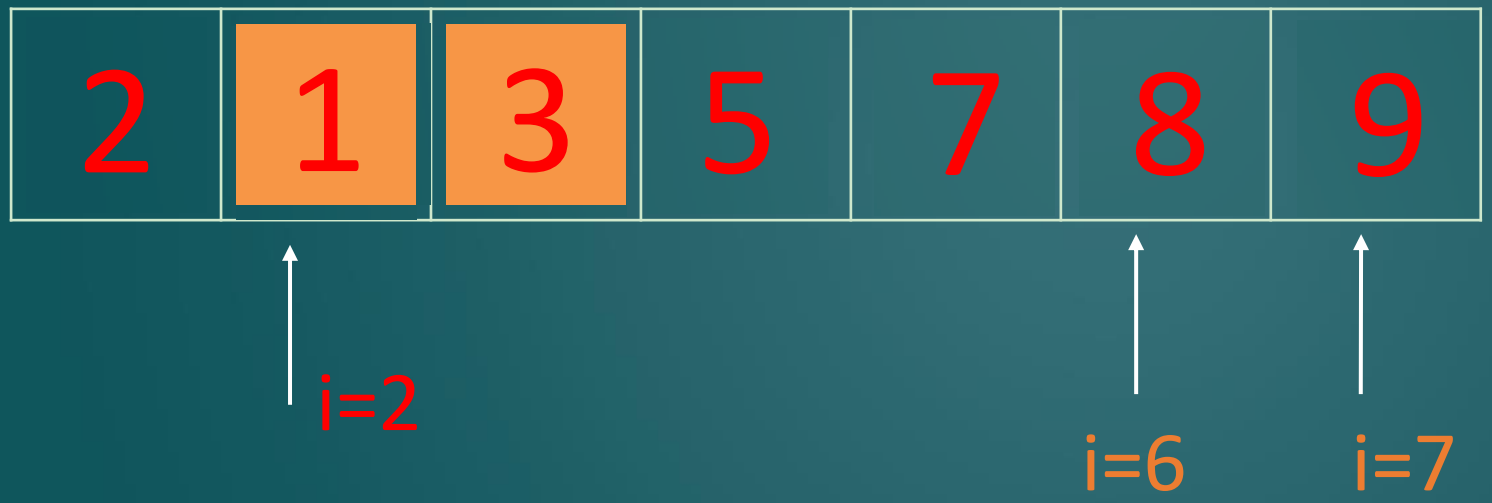
起泡排序算法

注意:

1. 起泡排序的结束条件为，**最后一趟没有进行“交换记录”。**

2. 一般情况下，每经过一趟“起泡”，“i 减1”，但并不是每趟都如此。

例如:




```
for (j = 1;  j < i;  j++)  if (R[j+1].key < R[j].key) …
```

时间分析:

最好的情况（关键字在记录序列中顺序有序）：　只需进行一趟起泡

“比较”的次数：　　**“移动”的次数：**

n-1　　**0**

最坏的情况（关键字在记录序列中逆序有序）：　需进行n-1趟起泡

“比较”的次数：　　**“移动”的次数：**

$$\sum_{i=n}^{2}(i-1)=\frac{n(n-1)}{2}$$

$$3\sum_{i=n}^{2}(i-1)=\frac{3n(n-1)}{2}$$

二、快速排序（Quick sort）

目标：找一个记录，以它的关键字作为“枢轴”，凡其关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于枢轴的记录均移动至该记录之后。

经过**一趟排序**之后，记录的无序序列R[s..t]将**分割成两部分**：R[s..i-1]和R[i+1..t]，且R[j].key≤ R[i].key ≤ R[p].key

$(s \le j \le i-1)$ $(i+1 \le p \le t)$。

枢轴

例如:

设 R[s]=52 为枢轴

将 R[high].key 和 枢轴的关键字进行比较，要求R[high].key ≥ 枢轴的关键字

将 R[low].key 和 枢轴的关键字进行比较，要求R[low].key ≤ 枢轴的关键字

可见，经过“**一次划分**”，将关键字序列

52, 49, 80, 36, 14, 58, 61, 97, 23, 75

调整为: **23**, 49, **14**, 36, **(52)** 58, 61, 97, **80**, 75

在调整过程中，设立了两个指针： **low** 和**high**，它们的初值分别为： s 和 t。

之后逐渐减小 **high**，增加 **low**，并保证

R[**high**].key≥52，和 R[**low**].key≤52,否则进行记录的“交换”。

```
int Partition (RedType& R[], int low, int high) {
  pivotkey = R[low].key;  //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  && R[high].key>=pivotkey)
      --high;    //将比枢轴记录小的记录交换到低端
    R[low]←→R[high];
    while (low<high  && R[low].key<=pivotkey)
      ++low;   //将比枢轴记录大的记录交换到高端
    R[low]←→R[high];
  }
  return low;          // 返回枢轴所在位置
} // Partition
```

算法 10.6 (a)

对上面算法进行分析，每交换一对记录需进行三次记录移动（赋值）的操作。而实际上，在排序过程中对枢轴记录的赋值是多余的，因为只有在一趟排序结束时，即low=high的位置才是枢轴记录的最后位置。

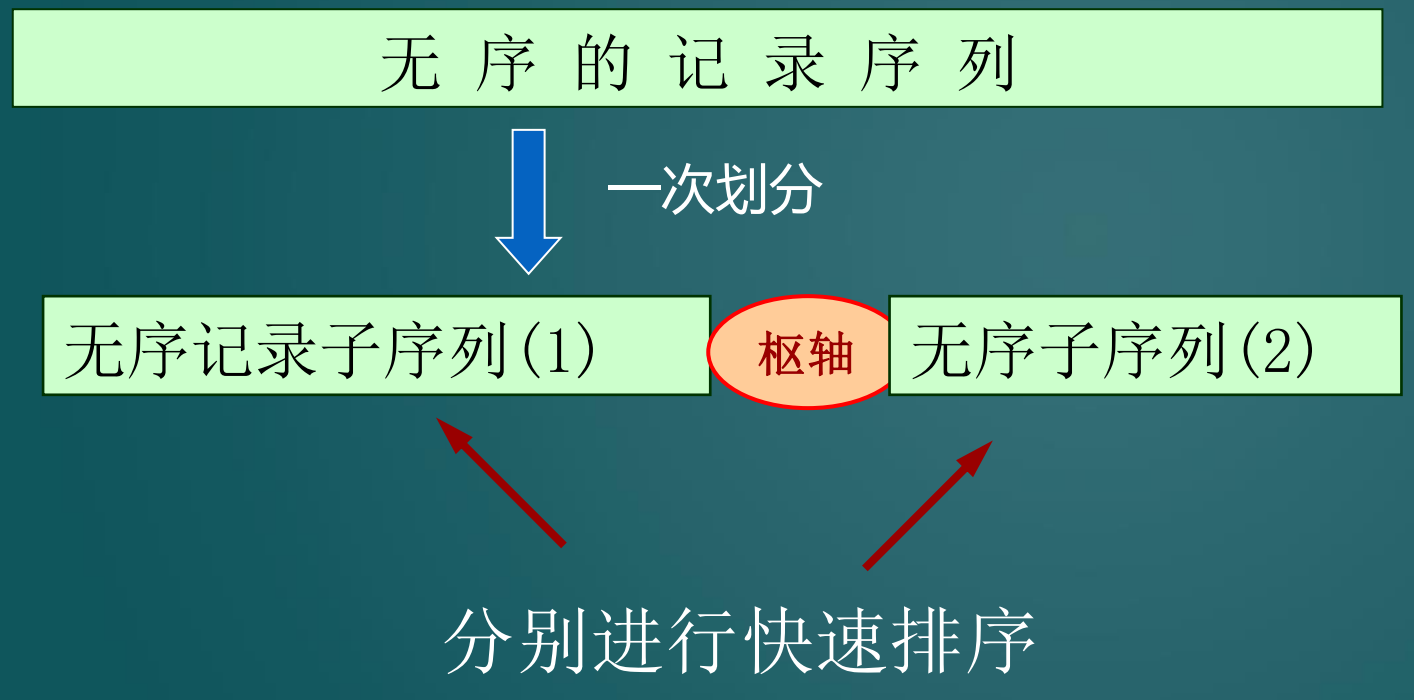
由此改写上述算法，先将枢轴记录暂存在R[0]的位置上，排序过程中只作 R[low]或R[high]的单向移动，直至一趟排序结束后再将枢轴记录移至正确位置上。如算法10.6（b）所示。

```
int Partition (RedType R[], int low, int high) {
   R[0] = R[low];  pivotkey = R[low].key;  // 枢轴
   while (low<high) {  //从表的两端交替地向中间扫描
         while(low<high&& R[high].key>=pivotkey)
              -- high;     //将比枢轴记录小的记录交换到低端
          R[low] = R[high];
         while (low<high &&R[low].key<=pivotkey)
              ++ low;     //将比枢轴记录大的记录交换到高端
          R[high] = R[low];
     }
     R[low] = R[0];     return low；//枢轴记录到位，返回
}// Partition                                  //枢轴位置
```

算法 10.6 (b)

一趟快速排序

首先对无序的记录序列进行“**一次划分**”，之后**分别**对分割所得两个子序列“**递归**”**进行快速排序。**



```
void QSort (RedType & R[],  int low,  int  high ) {
  // 对记录序列R[low..high]进行快速排序
  if (low < high) {              // 长度大于1
       pivotloc = Partition(R, low, high);
                          // 对 R[s..t] 进行一次划分
     QSort(R, low, pivotloc-1);
                 // 对低子序列递归排序，pivotloc是枢轴位置
       QSort(R, pivotloc+1, high); // 对高子序列递归排序
   }
} // QSort
```

算法 10.7

第一次调用函数 Qsort 时，待排序记录序列的上、下界分别为 1 和 L.length。

```
void QuickSort( SqList & L) {
  // 对顺序表进行快速排序
    QSort(L.r, 1, L.length);
} // QuickSort
```

算法 10.8