4 简单排序(2)

Simply Sorting

郝家胜

hao@uestc.edu.cn

自动化工程学院

内容提要

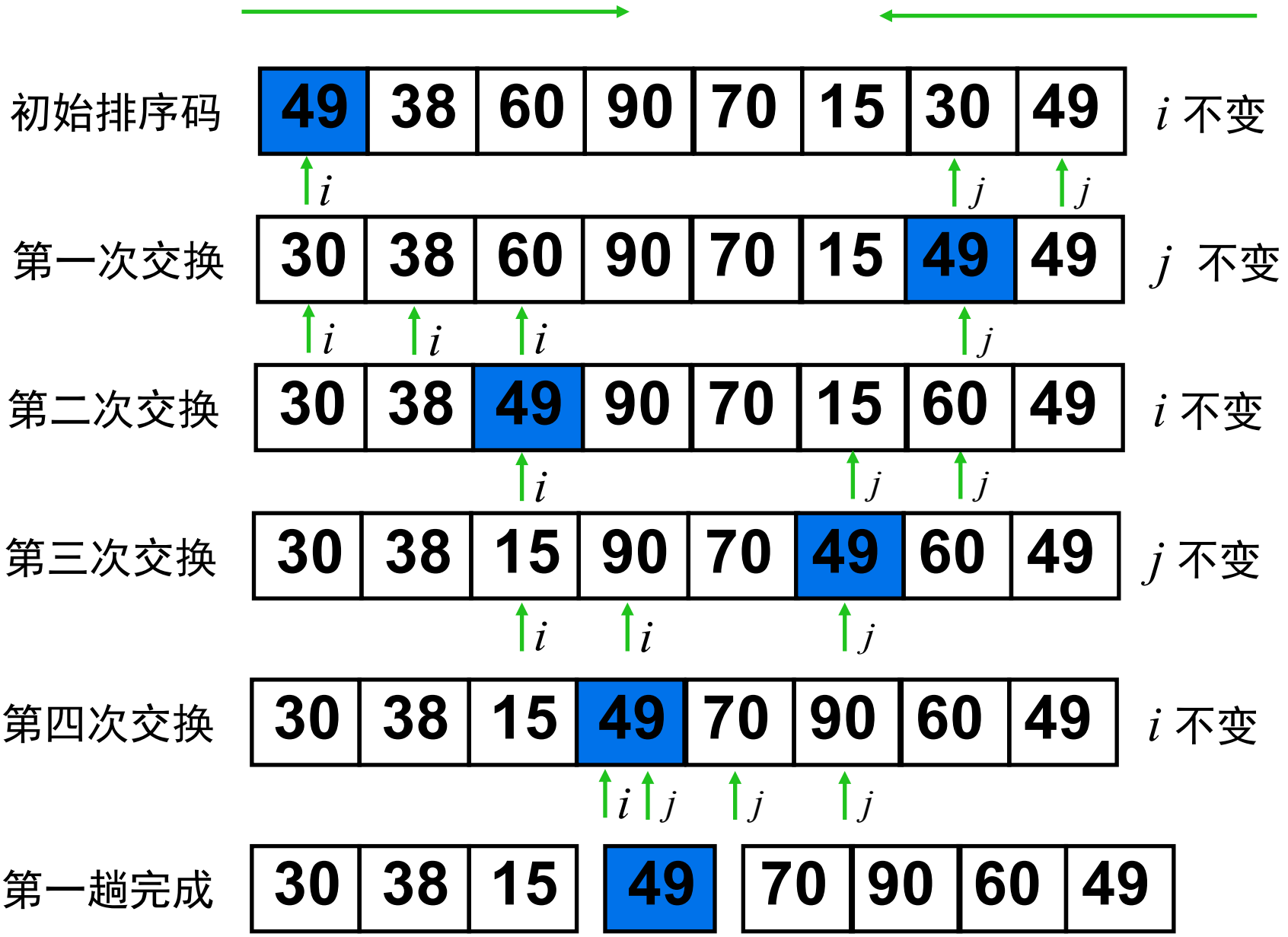
- 快速排序
- 归并排序
- 简单排序的比较

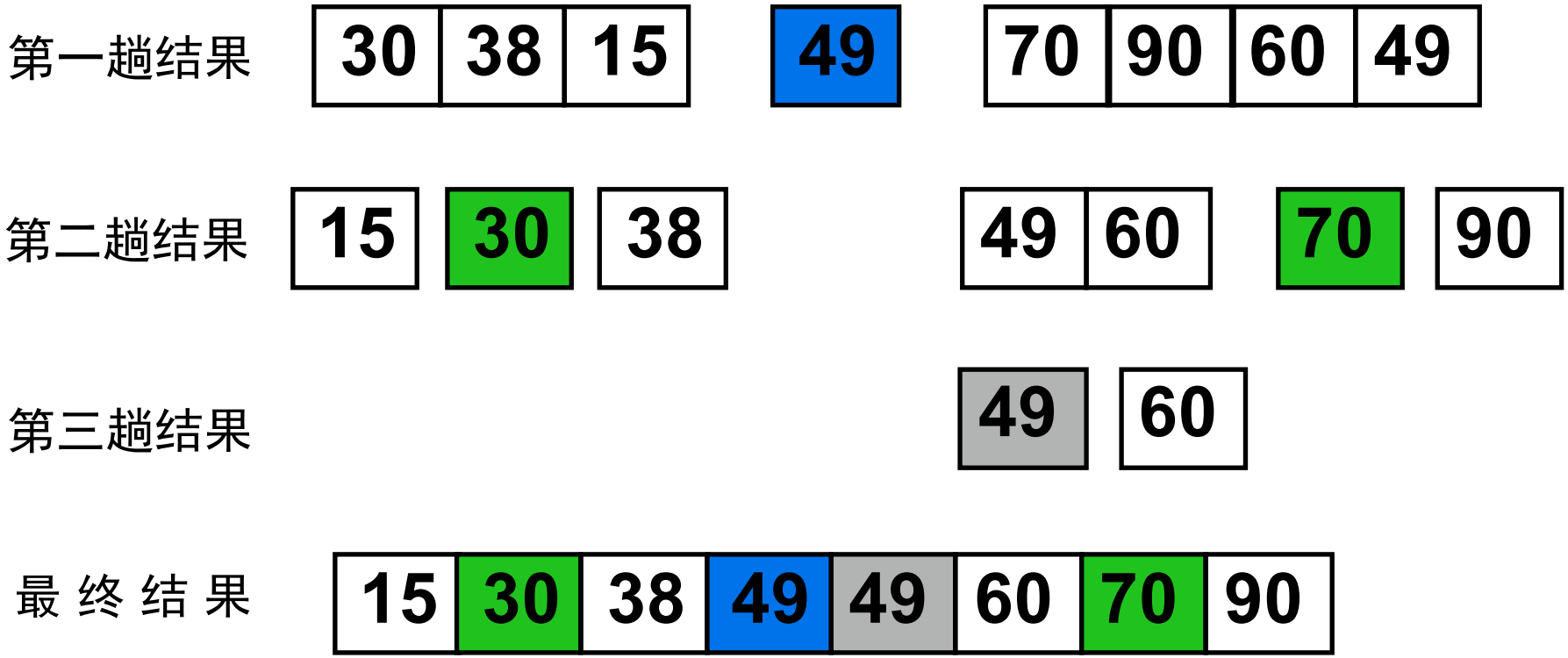
4 快速排序

快速排序

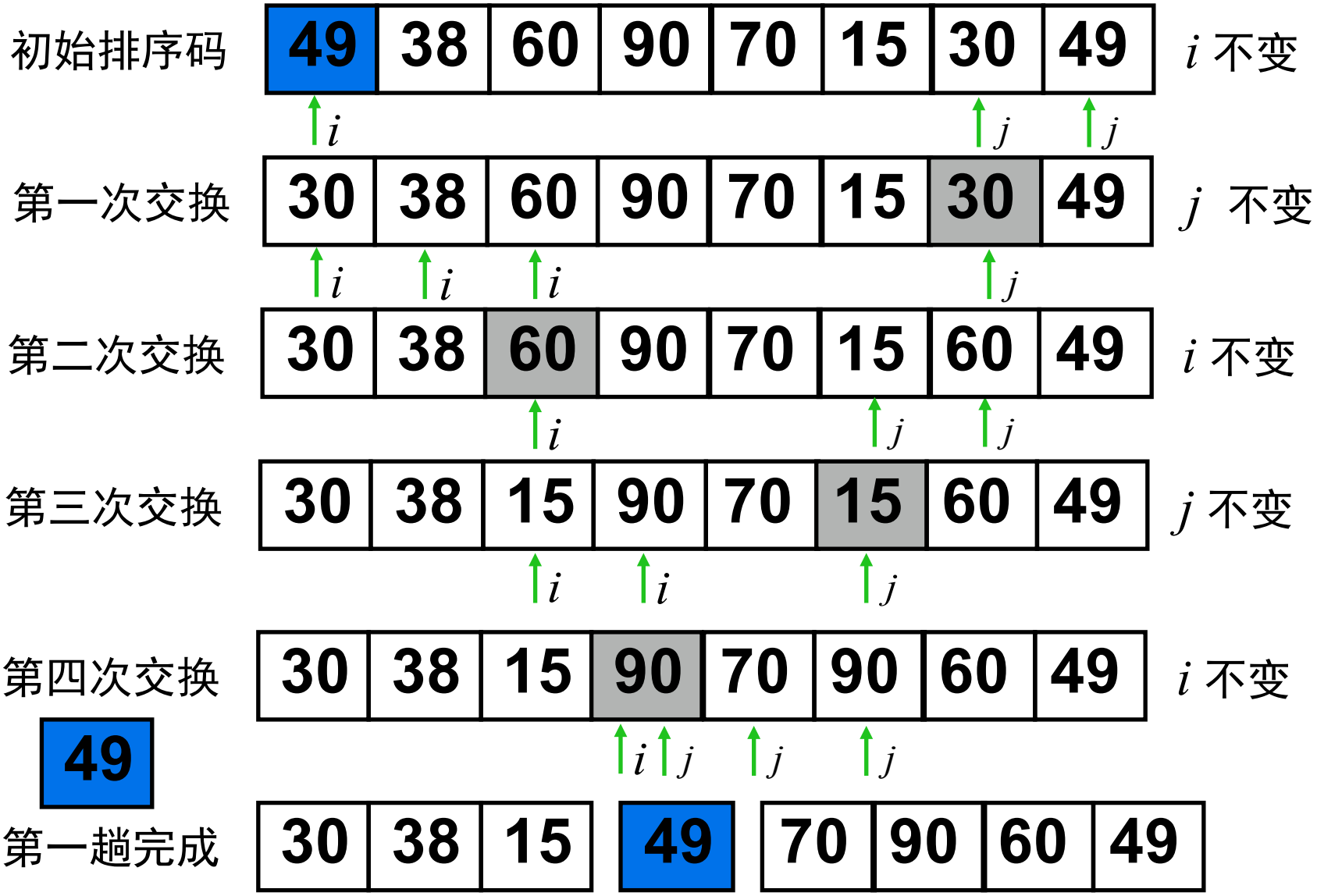
- 在待排序的n个记录中任取一个记录R(通常为第一个)，以该记录的排序码K为基准(pivot)，将所有剩下的n-1个记录划分为两个子序列
  - 第一个子序列中所有记录的排序码均小于或等于K
  - 第二个子序列中所有记录的排序码均大于K
- 然后将K所对应的记录R放在第一个子序列之后及第二个子序列之前，使得待排序记录序列成为<子序列1>，R，<子序列2>，完成快速排序的第一趟排序
- 然后分别对子序列1和子序列2重复上述划分，直到每个子序列中只有一个记录时为止

手工排序示例

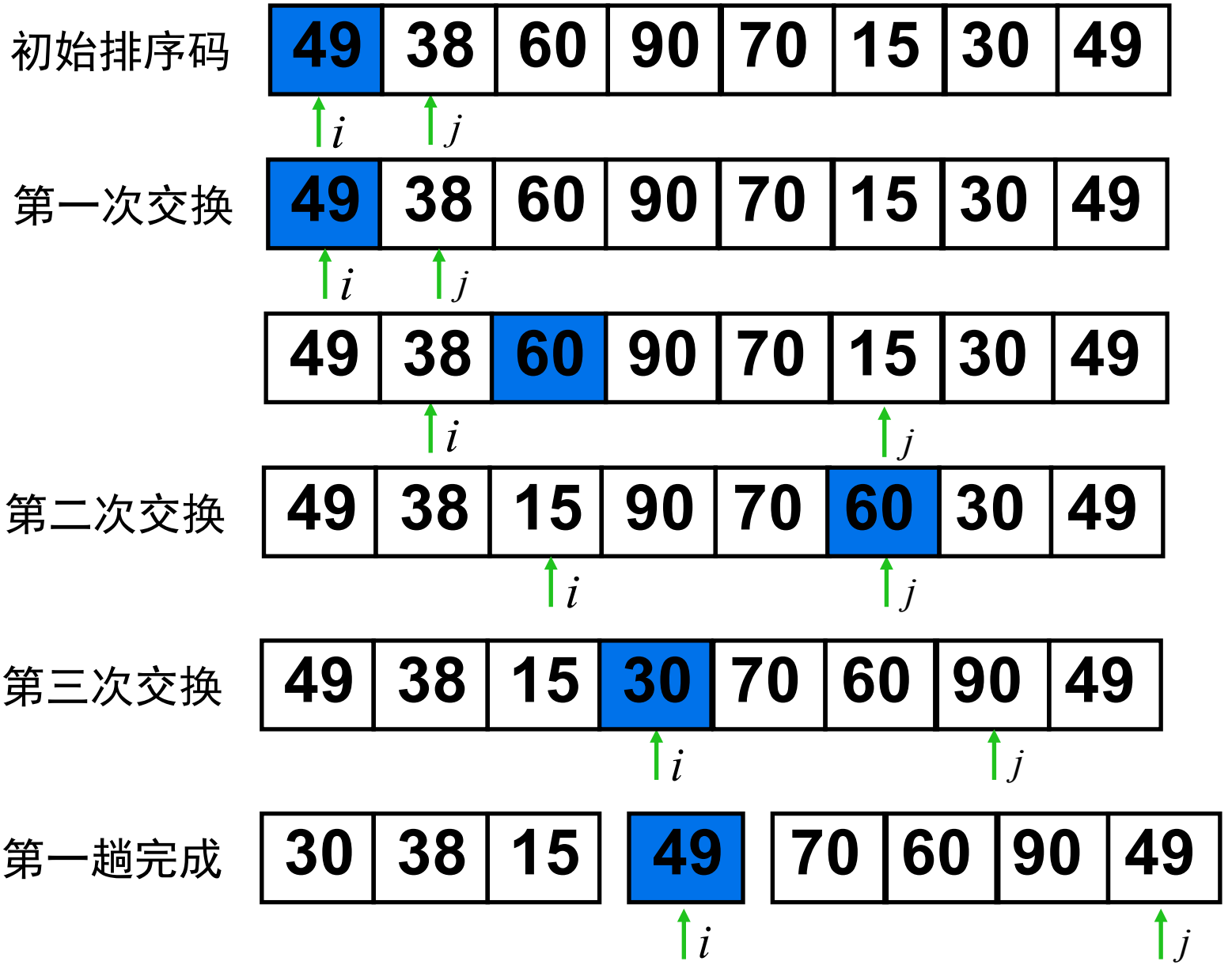

初始排序码
49 38 60 90 70 15 30 49
i 不变
i
j
j
第一次交换
30 38 60 90 70 15 49 49
j 不变
i i i
j
第二次交换
30 38 49 90 70 15 60 49
i 不变
i
j j
第三次交换
30 38 15 90 70 49 60 49
j 不变
i i
j
第四次交换
30 38 15 49 70 90 60 49
i 不变
i j
j
j
第一趟完成
30 38 15 49 70 90 60 49


第一趟结果
30 38 15 49 70 90 60 49
第二趟结果
15 30 38 49 60 70 90
第三趟结果
49 60
最 终 结 果
15 30 38 49 49 60 70 90

课本示例程序算法


初始排序码
49 38 60 90 70 15 30 49
i 不变
i
j
j
第一次交换
30 38 60 90 70 15 30 49
j 不变
i i i
j
第二次交换
30 38 60 90 70 15 60 49
i 不变
i
j j
第三次交换
30 38 15 90 70 15 60 49
j 不变
i i
j
第四次交换
30 38 15 90 70 90 60 49
i 不变
i j j j
49
第一趟完成
30 38 15 49 70 90 60 49

CLRS 算法


初始排序码
49 38 60 90 70 15 30 49
i j
第一次交换
49 38 60 90 70 15 30 49
i j
49 38 60 90 70 15 30 49
i j
第二次交换
49 38 15 90 70 60 30 49
i j
第三次交换
49 38 15 30 70 60 90 49
i j
第一趟完成
30 38 15 49 70 60 90 49
i j

算法伪代码

PARTITION(A, p, r)

▷ 教材算法

```
1  pivot ← A[p]
2  i ← p,  j ← r
3  while i < j  do
4      while (i < j and A[j] >= pivot)
5          j ← j - 1
6      if (i < j) A[i] ← A[j]
7      while (i < j and A[i] <= pivot)
8          i ← i + 1
9      if (i < j) A[j] ← A[i]
10 end
11 A[i] ← pivot
12 return i
```

算法伪代码

```
PARTITIONSED(A, p, r)
▷ R. Sedgewick 算法
1  pivot ← A[p]
2  i ← p,  j ← r
3  while i < j  do
4      while (i < j and A[j] >= pivot)
5          j ← j - 1
6      while (i < j and A[i] <= pivot)
7          i ← i + 1
8      if (i < j) A[i] ↔ A[j]
9  end
10 A[i] ↔ A[p]
11 return i
```

算法伪代码

```
QUICKSORT(A, p, r)
1 if p < r
2 then q ← PARTITION(A, p, r)
3     QUICKSORT(A, p, q-1)
4     QUICKSORT(A, q + 1, r)
5 end
```

```
PARTITIONCLRS(A, p, r)
▷ CLRS算法
1  pivot ← A[p]
2  i ← p
3  j ← p + 1
4  while j <= r
5      if A[j] < pivot then
6          i ← i + 1
7          A[i] ↔ A[j]
8      end
9      j ← j + 1
10 end
11 A[i] ↔ A[p]
12 return i
```

Divide-and-Conquer

- 分治策略
  - 对于输入的子序列L[p..r]，如果规模足够小则直接进行排序（一般选择直接插入排序），否则分三步处理
- 分解(Divide)：将待排序列L[p..r]划分为两个非空子序列L[p..q]和L[q+1..r]，使L[p..q]中任一元素的值不大于L[q+1..r]中任一元素的值
- 求解(Conquer)：通过递归调用快速排序算法，分别对子序列L[p..q]和L[q+1..r]进行排序
- 合并(Merge)：由于对分解出的两个子序列的排序是就地进行的，所以在L[p..q]和L[q+1..r]都排好序后不需要执行任何计算L[p..r]就已排好序

基准的选择

- 基准的好坏直接影响算法效率
- 理想的基准每次都将序列等分
- 一般做法
  - 取头
  - 取尾
  - 中位数
  - 随机
- 算法实现时的考虑
  - 首先交换到头或尾

C 实现

```
1. int qsort(int A[], int left, int right)
2. {
3.     int p;
4.     if (left < right) {
5.         p = partition(A, left, right);
6.         qsort(A, left, p - 1);
7.         qsort(A, p + 1, right);
8.     }
9.     return 0;
10. }
```

```
1.  int partition_a(int A[], int left, int right)
2.  {
3.       int pivot = A[left];
4.       int i = left;
5.       int j = right;
6.
7.       while ( i < j) {
8.          while (i < j && A[j] >= pivot) j--;
9.              if (i < j) swap(A, j, i);
10.         while (i < j && A[i] <= pivot) i++;
11.             if (i < j) swap(A, i, j);
12.      }
13.
14.     return i;
15. }
```

```
1.  int partition_b(int A[], int left, int right)
2.  {
3.      int pivot = A[left];
4.      int i = left;
5.      int j = right;

6.      while ( i < j) {
7.          while (i < j && A[j] > pivot)  j--;
8.          while (i < j && A[i] <= pivot) i++;
9.          if (i < j) swap(A, j, i);
10.     }

11.     swap(A, i, left);

12.     return i;
13. }
```

R. Sedgewick

```
1.  int partition_c(int A[], int left, int right)
2.  {
3.      int pivot = A[left];
4.      int i, j;
5.
6.      i = left;
7.
8.      for ( j = left + 1; j <= right; j++) {
9.          if (A[j] < pivot)
10.         swap(A, ++i, j);
11.     }
12.
13.     swap(A, i, left);
14.
15.     return i;
16. }
```

CLRS

Scheme

```
(define (qsort ls)
  (if (null? ls) '()
      (let ((x (car ls))
            (xs (cdr ls)))
        (let ((lts (filter (lambda (y) (< y x)) xs))
              (gts (filter (lambda (y) (>= y x)) xs)))
    (append (qsort lts) (list x) (qsort gts))))))
```

Haskell

```
qsort [] = []
qsort (x:xs) = qsort lts ++ [x] ++ qsort gts
    where lts = [y | y <- xs, y < x]
          gts = [y | y <- xs, y >= x]
```

算法分析

- 不稳定
- 时间复杂度
  - 在n个记录的待排序列中，一次划分需要约O(n)次比较
  - 若每次划分的前后两部分数据元素个数基本相等，则需要经过log*n*次划分
  - 所以算法时间复杂度为O(*n*log*n*)，最坏情况下，时间复杂度为$O(n^2)$
- 空间复杂度
  - 快速排序是递归的，每层递归调用的指针和参数均要存储，递归调用次数与划分次数一次，在理想情况下为O(log*n*)；在最坏情况下，空间复杂度为为O(n)
- n较大时采用
  - 一般与简单（直接）插入排序法结合

5 归并排序

归并排序

归并排序就是利用归并操作把一个无序序列排列成一个有序序列的过程。

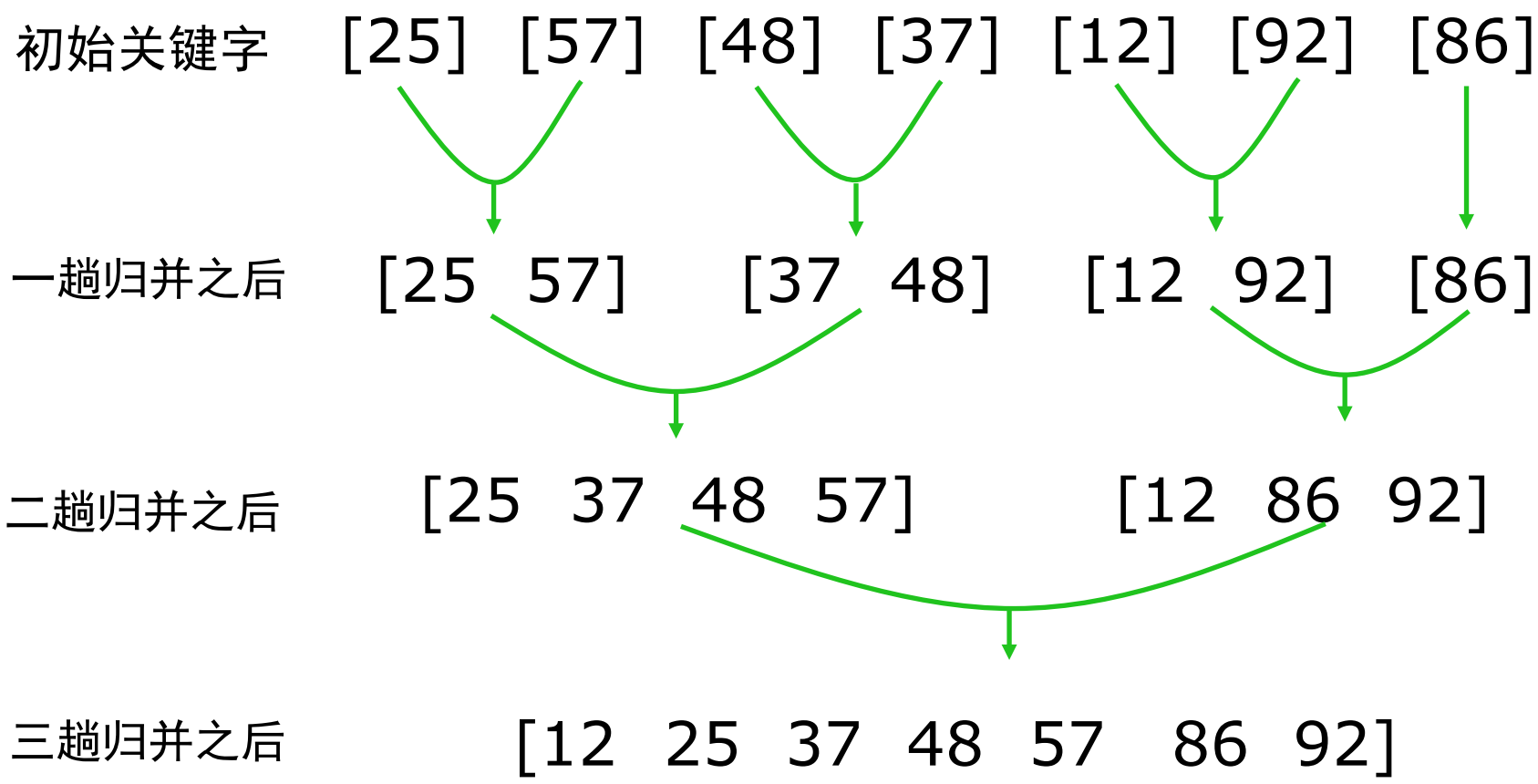
- 对于一个无序序列来说，归并排序把它看成是由n个只包含1个记录的有序序列组成的序列，然后进行两两归并或多个子序列归并，直至最后形成一个包含n个记录的有序序列

- 利用二路（两两）归并操作的排序过程称为二路归并排序

基本思想

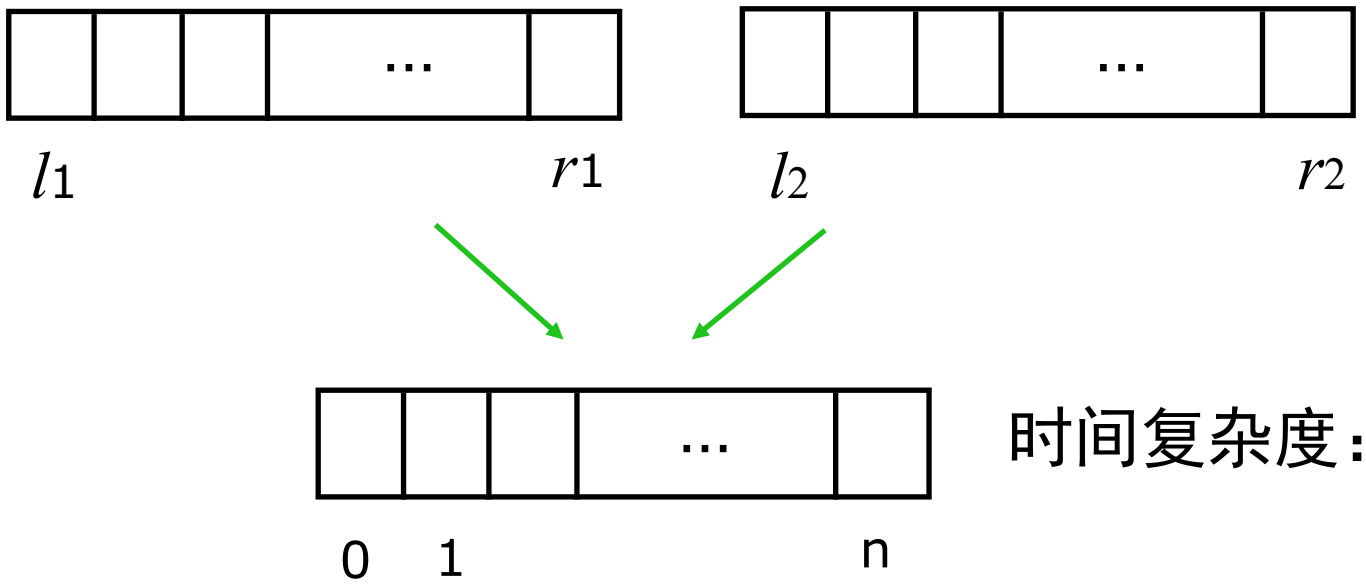
- 将有n个元素的序列看作n个有序子序列，每个子序列的长度为1
- 从第一个子序列开始，把相邻的子序列两两合并，得到 [n/2] 个长度为2或1的有序子序列（一次归并排序）
- 对新的子序列继续顺序成对归并
- 直至得到长为n的一个子序列

二路归并排序示例


初始关键字
[25] [57] [48] [37] [12] [92] [86]
一趟归并之后
[25 57] [37 48] [12 92] [86]
二趟归并之后
[25 37 48 57] [12 86 92]
三趟归并之后
[12 25 37 48 57 86 92]

一次归并

把若干个长度为 k的相邻有序子序列，从前向后两两进行合并，得到若干个长为2k的相邻有序子序列



时间复杂度：$O(m+n)$

特殊情况

- 剩余个数超过一个子序列长度，但不够两个子序列长度
  - 计算好后一个子序列的长度，按正常方式合并处理
- 剩余个数不够两个子序列长度
  - 直接放进新序列的尾部
- 举例说明
  - 例15

算法分析

- 分而治之
- 稳定
- 时间复杂度
  - O(*n*log*n*)
- 空间复杂度
  - 归并排序需要与所排序序列相同大小的辅助存储空间，空间复杂度为O(n)
- 在一般情况下，很少利用归并排序法进行内部排序
  - 外部排序

排序方法的比较

	平均时间复杂度	最坏情况	原地排序	稳定	空间复杂度
简单插入	$O(n^2)$	$O(n^2)$	√	√	O(1)
选择排序	$O(n^2)$	$O(n^2)$	√		O(1)
冒泡排序	$O(n^2)$	$O(n^2)$	√	√	O(1)
快速排序	O(n log(n))	$O(n^2)$	√		O(log(n))
归并排序	O(n log(n))	O(n log(n))		√	O(n)

几个结论

- 综合性能以快速排序最佳, 但最坏情况下不如归并排序
- 直接插入排序最简单, 当序列“基本有序”或 n 较小时, 它是最佳排序方法, 通常用它与“先进的排序方法”结合使用
- 从稳定性看, 直接插入排序、冒泡排序和归并排序是稳定的；而快速排序， 希尔排序和选择排序都是不稳定的
- 稳定性由方法本身决定，不稳定的方法总能举出使其不稳定的实例
- 一般来说，排序过程中比较是在两个相邻元素之间进行的排序方法是稳定的

- 插入、冒泡排序的速度较慢，但参加排序的序列局部或整体有序时，这种排序能达到较快的速度。
- 反而在这种情况下，快速排序反而慢了。
- 当n较小时，对稳定性不作要求时宜用选择排序，对稳 定性有要求时宜用插入或冒泡排序。
- 若待排序的记录的关键字在一个明显有限范围内时,且 空间允许是用桶排序。
- 当n较大时，关键字 元素比较随机，对稳定性没要求宜用快速排序。
- 当n较大时，关键字 元素可能出现本身是有序的，对稳定性有要求时，空间允许的情况下，宜用归并排序。
- 当n较大时，关键字 元素可能出现本身是有序的，对稳定性没有要求时宜用堆排序。