Chapter 7 Sort

Content

7.1 Some basic concept

Sorting （排序）

一般情况下，假设含*n*个记录的序列为{ $R_1, R_2, \ldots, R_n$ }，其相应的关键字序列为 { $K_1, K_2, \ldots, K_n$ }，这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系：

$$K_{p1} \le K_{p2} \le \ldots \le K_{pn}$$

按此固有关系将上式记录序列重新排列为

$$\{ R_{p1}, R_{p2}, \ldots, R_{pn} \}$$

的操作称作排序。

Stable（稳定的）

A sorting algorithm is said to be **stable** if it does not change the relative ordering among duplicate keys

Input record sequence

A	C	D	T	F
10	20	3	10	15

Sorted sequence using Algorithm1

D	A	T	F	C
3	10	10	15	20

stable

Sorted sequence using Algorithm2

D	T	A	F	C
3	10	10	15	20

Un-stable

The efficiency of sorting

排序的时间开销:

衡量算法好坏的最重要的标志。

排序的时间开销可用算法执行中的关键字比较次数（KCN）与记录交换次数（RSN）来衡量。

一般按平均情况进行大略估算。对于那些受对象初始排列及对象个数影响较大的，需要按最好情况和最坏情况进行估算。

算法执行时所需的附加存储:

评价算法好坏的另一标准。

5

5

静态排序和动态排序:

静态排序:

排序的过程是对数据对象本身进行物理重排，经过比较和判断，将对象移到合适的位置。这时，数据对象一般都存放在一个顺序表（数组）中。

动态排序: -- 考虑对应存储结构

给每个对象增加一个链接指针，在排序的过程中不移动对象或传送数据，仅通过修改链接指针来改变对象之间的逻辑顺序，从而达到排序的目的。

6

6

内部排序和外部排序

若整个排序过程不需要访问外存便能完成，则称此类排序问题为内部排序;

反之，若参加排序的记录数量很大，整个序列的排序过程不可能在内存中完成，则称此类排序问题为外部排序。

7

7

7.2 three $\Theta(n^2)$ sorting Algorithms

1. Insertion Sort(插入排序)
2. Bubble Sort(冒泡排序)
3. Selection Sort(选择排序)

自学并讨论

8

8

自学并讨论时请思考并总结

	冒泡排序	选择 排序	插入 排序
主要思想			
Stable			
KCN&RSN (best case)			
KCN&RSN (worst case)			

假设原始序列为： 38 20 17 13 28 14 23 9， 判断下列各序列分别是以上那种排序第四趟的中间结果

(1) 9 13 14 17 38 20 23 28 (   )

(2) 13 17 20 28 38 14 23 9 (   )

(3) 9 13 14 17 28 20 23 38 (   )

9

9

1. Insertion Sort

插入排序

10

10

n： 记录个数

Insertion Sort(1)

下标		i=1	2	3	4	5	6	7
0	42	20	17	13	13	13	13	13
1	20	42	20	17	17	14	14	14
2	17	17	42	20	20	17	17	15
3	13	13	13	42	28	20	20	17
4	28	28	28	28	42	28	23	20
5	14	14	14	14	14	42	28	23
6	23	23	23	23	23	23	42	28
7	15	15	15	15	15	15	15	42
KCN	1	2	3	2	5	3	6	
RSN	1	2	3	1	4	2	5	

共进行 n-1趟　**有序的子序列中插一个**

第i (1～n-1)趟将第i个数插入到 已有序的子序列(包含i个元素)中。

此时，只有0~i 元素值会发生变化，而第i+1～n-1 元素值保持不变

11

11

Insertion Sort (2)

Stable?

```
template <class Elem>
void inssort(Elem A[], int n) {
  for (int i=1; i<n; i++)
    for (int j=i; (j>0) &&(A[j]< A[j-1]); j--)
        swap(A, j, j-1);
}
```

共进行 n-1趟

第i 趟需做(1~i)次比较，做(0~i)次交换

12

12

Insertion Sort time analysis(1)

关键码比较次数和记录移动次数与记录的初始排列有关。

- 最好情况下，初始时元素递增有序(正序)，每趟只需与前面的最后一个对象比较 1 次，总的比较次数为 n-1，交换次数为0。
- 最坏情况下，初始时元素递减有序(逆序)，第 i 趟时需比较并且与前面 i 个对象交换。则总的比较次数KCN和交换次数RSN分别为

$$KCN=\sum_{i=1}^{n-1} i = n(n-1)/2 \approx n^2/2,$$

$$RSN=\sum_{i=1}^{n-1} i = n(n-1)/2 \approx n^2/2$$

$\Theta(n^2)$

- Average Case: 初始元素随机无序排列 KCN: $n^2/4$ RSN: $n^2/4$

13

13

2. Bubble Sort

冒泡排序

14

14

Bubble Sort (1)

下标		i=0	1	2	3	4	5	6
0	42	13	13	13	13	13	13	13
1	20	42	14	14	14	14	14	14
2	17	20	42	15	15	15	15	15
3	13	17	20	42	17	17	17	17
4	28	14	17	20	42	20	20	20
5	14	28	15	17	20	42	23	23
6	23	15	28	23	23	23	42	28
7	15	23	23	28	28	28	28	42
KCN	7	6	5	4	3	2	1	
RSN	5	4	4	2	1	1	1	

共进行 n-1趟

有序的子序列末尾添一个

第i (0 ~ n-2)趟通过两两对比交换将第i小的数冒泡到下标为i的位置

此过程中，*i~n-1* 元素值可能发生变化，而*0~ i-1*元素值保持不变

15

15

Bubble Sort (2)

```
template <class Elem, class Comp>
void bubsort(Elem A[], int n) {
  for (int i=0; i<n-1; i++)
    for (int j=n-1; j>i; j--)
      if (A[j]<A[j-1])
        swap(A, j, j-1);
}
```

Stable?

共进行 n-1趟 (0 ~ n-2)
第i 趟需做(n-1-i)次比较，做(0 ~ n-1-i)次交换

16

16

Bubble Sort time analysis

–最好情况（正序）

- KCN：n*(n-1)/2
- RSN：0

KCN为固定值，与初始序列中元素值的顺序无关

–最坏情况（逆序）

- KCN： n*(n-1)/2
- RSN: n*(n-1)/2

平均情况 (无序)

- KCN $n^2/2$
- RSN： $n^2/4$

$\Theta(n^2)$

17

17

3. Selection Sort

选择排序

18

18

Selection Sort (1)

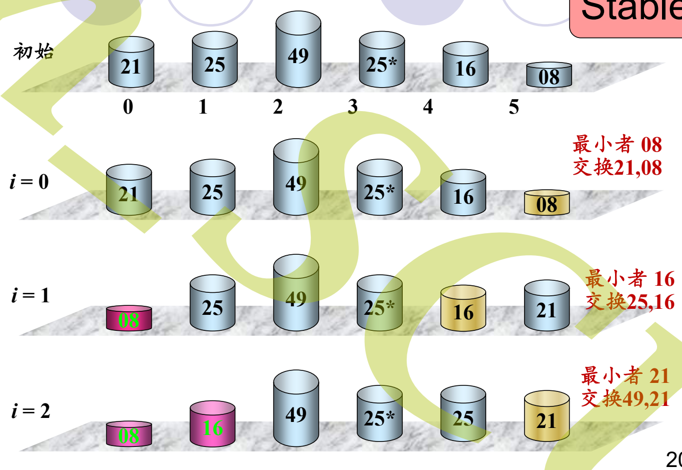
思路：

- ✓ 进行n-1趟（1～n-1）
- ✓ 每i趟从序列第i-1～n-1的n-i+1个记录中选择关键字最小那个，将其与第i-1个记录进行交换
- ✓ 此过程中，只有2个元素值（下标为i-1和选出的最小那个）可能发生变化，而其余元素值保持不变

19

Selection Sort (2)

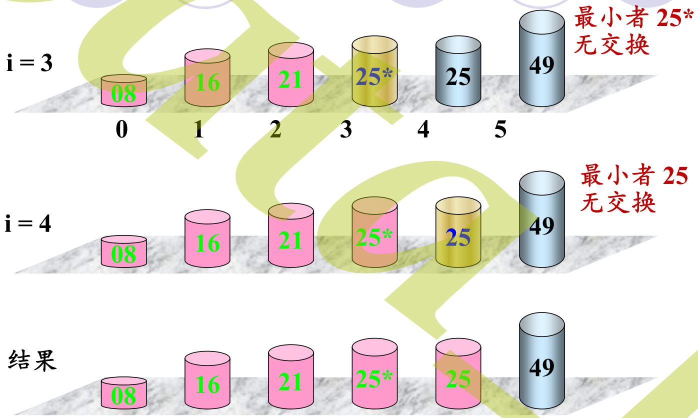
Stable？



20

20

Selection Sort (3)



21

21

Selection Sort (4)

```
template <class Elem>
void selsort(Elem A[], int n) {
  for (int i=0; i<n-1; i++) {
    int lowindex = i; // Remember its index
    for (int j=n-1; j>i; j--) // Find least
      if (A[j] < A[lowindex])
        lowindex = j; // Put it in place
    if(i != lowindex) swap(A, i, lowindex);
  }
}
```

22

22

Selection Sort time analysis

- 比较次数与序列初始排列无关。第i趟选择的比较次数总是 ***n-i-1*** 次。因此

$$KCN = \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2}$$

Θ(n²)

- 对象的交换次数与初始排列有关。
  - 最好情况（初始正序），$RSN = 0$
  - 最坏情况(初始逆序）是每一趟都要进行1次交换，总交换次数 $RSN = n\text{-}1$

23

23

Summary of above three sort

	Insertion	Bubble	Selection
Comparisons(KCN):			
Best Case(正序)	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case(逆序)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps(RSN）:			
Best Case(正序)	0	0	0
Worst Case(逆序)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

- 3种算法执行时均不需要附加存储，且都属于静态排序
- 插入和冒泡排序 stable， 选择排序 unstable

24

24

自学时请思考并总结

	冒泡排序	选择 排序	插入 排序
主要思想			
Stable	**Yes**	**No**	**Yes**
KCN & RSN (Avg)	$\Theta(n^2)$ & $\Theta(n^2)$	$\Theta(n^2)$ & $\Theta(n)$	$\Theta(n^2)$ & $\Theta(n^2)$
其他	均不需要额外空间开销		

假设原始序列为：38 20 17 13 28 14 23 9， 判断下列各序列分别是那种排序的中间结果

(1) 9 13 14 17 38 20 23 28 (Bubble)

(2) 13 17 20 38 28 14 23 9 (Insert)

(3) 9 13 14 17 28 20 23 38 (Select)

25

25

- 能否对序列先做预处理，使得序列尽可能接近正序？然后再调用简单插入排序算法？
  - **Shell**排序即采用这种思想

26

26

7.3 Shell Sort
希尔排序

27

27

Shell Sort (1) 希尔排序

- 也叫：缩小增量排序
- 基本思想：对待排序列先作“宏观”调整，再作“微观”调整。
- 具体的：将序列分成若干子序列，分别对每个子序列进行插入排序。

如：将 n 个记录分成 d 个子序列：

{ R[0]，R[0+d]，R[0+2d]，…，R[0+kd] }

{ R[1]，R[1+d]，R[1+2d]，…，R[1+kd] }

…

其中，d称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为1。

28

28

Shell Sort (2)—example1

Stable？

例如：

12	25	12	30	47	11	23	36	9	18	31

设增量 d =5，，分为5个子序列，第一趟希尔排序结果：

11	23	12	9	18	12	25	36	30	47	31

设增量 d = 3，分为3个子序列，第二趟希尔排序结果：

9	18	12	11	23	12	25	31	30	47	36

设增量 d = 1，第三趟希尔排序结果：

9	11	12	12	18	23	25	30	31	36	47

29

29

Shell Sort (3)) — example2

d =8　59 20 17 13 28 14 23 83 36 98 11 70 65 41 42 15

d =4

d =2

增量d的取法：

- Shell： d =⌊n/2⌋，d =⌊d/2⌋。
- Knuth: d =⌊n/3⌋+1，d =⌊d/3⌋+1。

最后一个增量值必须为1。

d =1　11 13 17 14 23 15 28 20 36 41 42 70 59 83 65 98

11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

30

30

Shell Sort (3)

```
template <class Elem>
void ShellInsert (Elem A[ ], int n, int d )  {    //一趟希尔排序
  for ( int  k=0; k < d; k++)
      for( int  i=k+d, i<n; i=i+d)    // Sort sublists using insertion sort
          for (int j=i; (j>k) && (A[j]< A[j-d]);  j=j-d)
              swap(A, j, j-d);
}  // ShellInsert
```

```
template <class Elem>
void  ShellSort (Elem A[ ], int n, int Gap[],  int t)  {
// 增量为Gap[t]的希尔排序
     for (int k=0; k<t; ++k)
          ShellInsert  <Elem> (A[ ], n, Gap[k]);  // For each incr
} // ShellSort
```

31

31

Shell Sort Algorithm analysis

- 对特定的待排序序列，可以准确地估算KCN和RSN。
- 但要弄清KCN和RSN与增量选择之间的依赖关系，并给出完整的数学分析，目前还没有人能够做到。
- Knuth利用大量的实验统计资料得出，当 n 很大时，KCN和RSN大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内。

✓ shell排序算法执行时不需要附加存储

✓ shell排序属于静态排序

✓ shell排序是unstable

32

32

7.4 Merge Sort

合（归）并排序

Merge Sort

idea：将两个（或两个以上）有序子序列“归并”为一个有序序列。两路/多路归并

	有序子序列1				有序子序列2				
归并前	13	17	20	21	14	15	23	28	数组A
	13	17	20	21	14	15	23	28	临时数组T
	13	14	15	17	20	21	23	28	数组A

归并后

Merge Sort(1)---merge two ordered subseq

```
template <class Elem>
void TwoWayMerge (Elem A[], Elem temp[], int left, int mid,
    int right){
  for (int i=left; i<=right; i++)    temp[i] = A[i];    // Copy
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr < = right ; curr++)
    if (i1 > mid)    // Left exhausted
        A[curr] = temp[i2++];
    else if (i2 > right)    // Right exhausted
        A[curr] = temp[i1++];
    else if ( temp[i1] <= temp[i2])
        A[curr] = temp[i1++];
    else    A[curr] = temp[i2++];
}
```

$O(n)$

这段代码的作用？
若没有，会怎样？

left mid right

	13	17	20	21	14	15	23	28
temp[ ]	13	17	20	21	14	15	23	28
结果A[	13	14	15	17	20	21	23	28

Merge Sort 归并排序

待排序序列：36 20 17 13 28 14 23 15

36 20 17 13 28 14 23 15

2路归并 2路归并 2路归并 2路归并

20 36 | 13 17 | 14 28 | 15 23

2路归并 2路归并

13 17 20 36 | 14 15 23 28

2路归并

13 14 15 17 20 23 28 36

6趟+:次,5路归并， NFQ: 6-;

Merge Sort

归并排序的递归实现

Stable？

① 将原始序列A分为两子序列:

A[0]~ A[n/2-1] 和 A[n/2]~ A[n-1]

② 分别对两个子序列进行归并排序(递归调用)

③ 将两个排好序的子序列归并为一个序列

调用两路归并函数

37

37

Merge Sort(2) — 递归方法

```
template <class Elem>
void mergeSort(Elem A[], Elem temp[], int left, int right){
  if (left == right)  return;
  int mid = (left+right)/2;
  mergeSort<Elem>(A, temp, left, mid);
  mergeSort<Elem>(A, temp, mid+1, right);
  TwoWayMerge<Elem> (A, temp, left, mid, right);
}
```

$O(n\log_2 n)$

36	20	17	13	28	14	23	15
20	36	13	17	14	28	15	23
13	17	20	36	14	15	23	28
13	14	15	17	20	23	28	36

38

38

Mergesort Cost

MergeSort time cost: $O(n\log_2 n)$

Mergesort requires twice space.

附加空间

Mergesort is stable

39

39

Optimized MergeSort (1)

idea：将两个（或两个以上）有序子序列“归并”为一个有序序列。两路/多路归并

有序子序列1 有序子序列2

归并前	13	17	20	21	14	15	23	28	数组A
	13	17	20	21	14	15	23	28	临时数组T
	13	14	15	17	20	21	23	28	数组A

归并后

会出现其中一个子序列耗尽，而另一个子序列依然有剩余情况
因此需要判断是否到达子序列末端。

40

40

Optimized MergeSort (2)
---改进思路1

归并两个有序子序列为一个有序序列

有序子序列1　有序子序列2

归并前	13	17	20	21	14	15	23	28	数组A
	13	17	20	21	28	23	15	14	临时数组T
	13	14	15	17	20	21	23	28	数组A

归并后

因为比对是从两边往中间走，不会会出现子序列耗尽情况，因此不必判断是否到达子序列末端。

41

41

Optimized MergeSort (6)

```
template <class Elem>
void TwoWayMerge (Elem A[], Elem temp[], int left, int mid,
   int right){
  for ( int i1=mid;  i1>=left; i1--)    temp[i1] = A[i1];
  for ( i2=mid+1; i2 <= right; i2++)
       temp[right-i2+mid+1] = A[i2];
  i1 = left;  i2 = right;
  for (int curr=left; curr<=right; curr++)
    if (temp[i1] <= temp[i2])
       A[curr] = temp[i1++];
    else
       A[curr] = temp[i2--];
}
```

left　mid　right

	13	17	20	21	14	15	23	28
temp[ ]	13	17	20	21	28	23	15	14
结果A[ ]	13	14	15	17	20	21	23	28

42

42

Optimized MergeSort (4)
---改进思路2

待排序序列：36 20 17 13 28 14 23 15

归并排序

36 20 17 13　28 14 23 15

插入排序

13 17 20 36 | 14 15 23 28

2路归并

13 14 15 17 20 23 28 36

方案二：插入.5路归并（4趟4次）

Ehwhu

36 | 20 | 17 | 13 | 28 | 14 | 23 | 15

2路归并　2路归并　2路归并　2路归并

20 36 | 13 17 | 14 28 | 15 23

2路归并　2路归并

13 17 20 36 | 14 15 23 28

2路归并

13 14 15 17 20 23 28 36

方案一：完全5路归并（6趟:次）

So， we can use insertion sort to sort small sublists

43

43

Optimized MergeSort (5)

优化2路归并函数

1. have the two sublists run toward each other, so that their high ends meet in the middle. In this way, there is no need to test for end of sublistuse.
2. insertion sort to sort small sublists.

长度小于4或8

优化归并排序初始部分

44

44

Optimized MergeSort (7)

```
template <class Elem>
void mergesort(Elem A[], Elem temp[], int left, int right) {
  if ((right-left) <= THRESHOLD) {    //调用插入排序
        inssort<Elem>(&A[left],right-left+1);
        return;
  }
  int  mid = (left+right)/2;
  mergesort<Elem>(A, temp, left, mid);
  mergesort<Elem>(A, temp, mid+1, right);
 TwoWayMerge <Elem>(A, temp, left, mid, right);
}
```

45

45

7.5 Heap Sort

堆排序

46

46

Review Heap

Defination: n个元素组成的序列$\{k_0,k_1,\ldots,k_{n-1}\}$当且仅当满足下列关系之一时，称之为**堆**

1） $k_i \le k_{2i+1}$,且 $k_i \le k_{2i+2}$,（小堆）

2) $k_i \ge k_{2i+1}$,且 $k_i \ge k_{2i+2}$,（大堆）

堆排序：利用堆的特性对记录序列进行排序的一种排序方法。建大堆→重复removeFirst直到堆空

47

47

Heap Sort(1)

Stable？

example1：

初始 { 40, 55, 49, 73, 12, 55, 98, 81, 64, 36 }

↓ 建大堆(maxHeap)

{ 98, 81, 55, 73, 36, 40, 49, 55, 64, 12 }

↓ (removeFirst)

{ 81, 73, 55, 64, 36, 40, 49, 55, 12, 98 }

↓ continue removeFirst

……

结果 { 12, 36, 40, 49, 55, 55, 64, 73, 81, 98 }

48

48

Heap Sort(2)

```
template <class Elem>
void heapSort(Elem A[], int m) { // Heapsort
  Elem mval;
  maxHeap<Elem>  H(A, m, m);
  H.buildHeap();    // biuld max heap
  for (int i=0; i< m; i++)  {    //  sort
    H.removeFirst();      // Put max at end
    cout << H.heapSize( );  // for debug
  }
}
```

49

49

Heap Sort(2)

Cost of heapsort: $\Theta(n \log n)$ for all case

Cost of finding K largest elements: $\Theta(n+k \log n)$

✓ HeapSort算法执行时不需要附加存储

✓ HeapSort is unstable

50

50

7.6 Quick Sort

快速排序

51

51

Quick Sort (1)

--一次划分

选最后一个

目标：找一个记录，以它的关键字作为“枢轴”，

行为：凡关键字小于枢轴的记录均移动至该记录之前，反之，凡关键字大于等于枢轴的记录均移动至该记录之后。

KCN，RSN尽量少，尽量不需求额外空间

结果：处理之后，无序的记录序列R[s..t]将以R[i]为界分割成两部分：R[s..i-1]和R[i+1..t]，且

R[s..i-1].key < R[i].key ≤ R[i+1..t].key

快速排序的一次划分

52

52

Quick Sort (2)
----一次划分算法

输入：无序序列A[] ,起止索引 *l, h*

输出：已划分后的A[ ]

步骤： 16 76 52 139 2 7 95 46 60 85 40 57

S1: 确定枢轴， ***p=A[h]; t = h;***

S2: 从*l*处开始，往右寻找大于等于P的元素，找到或**碰到 h 就停**，更新*l*为当前元素下标。

S3: 从*h*处开始，往左寻找小于P的元素，找到或**碰到 l 就停**，更新*h*为当前元素下标。

S4: 交换***A[h]***，***A[l]*** 的值，

S5: 重复S2-S4, 直到 ***l == h***

S6： 交换A[h]，A[t] 的值

53

Quick Sort (3)—划分

```
templete <typename Elem>
int Partition (Elem& R[ ], int l, int h) {
    Elem  pivot = R[h];   int  t = h;    // 确定枢轴
    do {
        while((l<h) && R[l].key < pivot.key)
            l++;    // 从左向右搜索
        while ( (l<h) && (R[h].key>=pivot.key) )
            h--;    // 从右向左搜索
        swap( R, l, h);
    } while (l < h) ;
    swap(R, h, t);
    return l;   // return  pivot position
}
```

O(*n*)

54

Quick Sort (4)

进行“一次划分”之后，可分别对分割所得两个子序列进一步处理使之有序。

Divide & conquer

1 7 2 6 5 3 4

划分

1 3 2 | 4 | 5 7 6

划分

1 | 2 | 3 | 4 | 5 | 6 | 7

直到各子序列长度为1

55

Quick Sort（4）

快速排序的递归实现算法

① 将原始序列A做1次划分

无序子序列(1) 枢轴 无序子序列(2)

- ✓ 子序列(1)中的元素值小于枢轴
- ✓ 子序列(2)中的元素值大于等于枢轴

② 分别对两个子序列再进行快速排序(递归调用)

56

Quick Sort (5)

```
templete <typename Elem>
void QSort (Elem & R[ ], int i, int j ) {
 if (i < j) {          // 长度大于1
   int k = Partition(R, i, j);
                       // 对 R[i…j] 进行一次划分
   QSort(R, i, k-1);
     // 对低子序列递归排序, k是枢轴位置
   QSort(R, k+1, j);
        // 对高子序列递归排序
 }
} // QSort
```

16 76 52 139 2 7 95 46 60 85 40 57

57

Quick Sort algorithm analysis (1)

- 最理想的情况：每次划分后左侧与右侧子序列的长度相等。
- 已知对k个元素进行一次划分所需时间为 ck。若设 $t(n)$ 是对 n 个元素的序列进行快速排序所需的时间，则总的时间为：

$t(n) \le cn + 2\,t(n/2)$ // c是一个常数

$\le cn + 2\,(\,cn/2 + 2t(n/4)\,) = 2cn + 4t(n/4)$

………

$\le cn\log_2 n + nt(1) = O(n\log_2 n)$

- 已有文献证明，*quicksort*的平均计算时间也是$O(n\log_2 n)$。
- 大量实验结果表明：就平均计算时间而言，当n很大时快速排序是我们所讨论的所有内排序方法中最好的。

58

58

Quick Sort algorithm analysis (2)

- 在最坏的情况，即待排序序列已经有序时，每次划分只得到一个比上一次少一个对象的子序列。这样，必须经过 n-1 趟才能把所有对象定位，而且第 i 趟需要经过 n-i 次比较，总的比较次数将达到$n^2/2$

排序速度退化到简单排序的水平，比直接插入排序还慢。

➢ 对于 n 较大的平均情况而言，快速排序是“快速”的，但是当 n 很小时，这种排序方法往往比其它简单排序方法还要慢。

?

✓ quicksort算法执行时不需要附加存储

✓Quicksort is unstable

59

59

Optimizations for Quicksort

思路一

- Better Pivot（每次划分）
  - 选择基准对象，使得每次划分所得的两个子序列中的对象个数尽可能地接近，很难办到
  - 其他思路：选择中间位置元素，选择最左边元素
  - 一个简单实用思路：取每个待排序对象序列的第一个对象、最后一个对象和位置接近正中的3个对象，取其关键码居中者作为基准对象
- Better algorithm for small lists
  - use insertion sort to sort small sublists

思路二

60

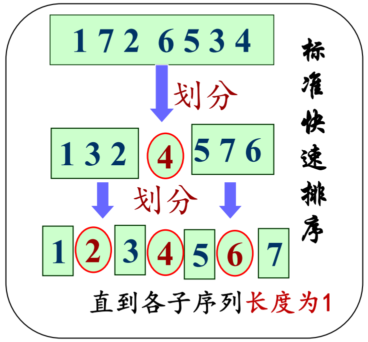
60

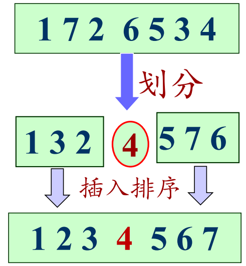
```
templete <typename Elem>
int findpivot (Elem & R[ ],  int i,  int  j ) {
//取中间位置记录作为pivot
   return  (i+j)/2;   }

templete <typename Elem>
int findpivot (Elem & R[ ],  int i,  int  j ) {
//取头.中,尾三个中关键值居中的记录作为pivot
   int k=(i+j)/2;
   if  (R[i].key >= R[k].key)  {
      if  (R[i].key<=R[j].key)       return i;
      else  if  R[k].key > R[j].key)  return k;
      else   return j;
}
```

61

61

快速排序改进思路二---引入插入排序





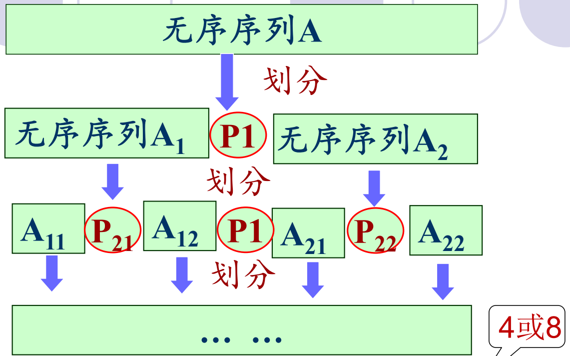
整个排序过程 只有划分

引入插入排序

62

62

快速排序改进思路二



直到各子序列长度 小于某一阈值

对长度小于阈值的子序列采用插入排序

63

Improved Quick Sort

```
templete <typename Elem>
void QSort (Elem & R[ ],  int i,  int  j ) {
  if ((j-i) <= THRESHOLD)   //长度较小时调用插入排序
   {  inssort<Elem>(R[i], j-i+1);      return;  }
  int pivotindex = findpivot(R, i, j);
  swap(R, pivotindex, j);   // Put pivot at most right
  int k = Partition(R, i, j); // 对 R[i..j] 进行一次划分
  QSort(R, i, k-1);   // 对低子序列递归排序
  QSort(R, k+1, j);  // 对高子序列递归排序
} // QSort
```

64

64