数据结构与算法

排序算法

计算机学院

朱晨阳 副教授

zhuchenyang07@nudt.edu.cn

排序

- 5.1 基本概念
- 5.2 插入排序（直接插入，折半插入，shell排序*）
- 5.3 选择排序（直接选择，树形选择）
- 5.4 交换排序（冒泡排序，快速排序）
- 5.5 分配排序（基数排序*）
- 5.6 归并排序

基本概念

- 排序码是结点中的一个或多个字段，其值作为排序运算中的依据。
- 排序码可以是关键字或非关键字。不是关键字时，可能有多个结点的排序码具有相同的值，这时排序结果不唯一。
- 排序中将结点称为记录，每个记录有一个排序码，将一系列结点构成的线性表称为文件。
- 排序运算是将文件中的记录按排序码排成非递减(或非递增)序列。

基本概念

- 如果一个排序算法对于任意具有相同排序码的多个记录在排序之后，这些具有相同排序码的记录的相对次序一定能保持不变，则称该排序算法是稳定的；否则称该排序算法是不稳定的。
- 排序前：{"Mike, 90","John, 90","Tom, 80"}
- 排序后：
  - {"Tom, 80","Mike, 90","John, 90"}
  - {"Tom, 80","John, 90","Mike, 90"}

直接插入排序

- 序号 0 1 2 3 4 5 6 7
- 初始 [46] 58 15 45 90 18 10 62
- for i=1,···, n-1, A[i]插入A[0 : i-1]

i=1	[46	58]	15	45	90	18	10	62
i=2	[15	46	58]	45	90	18	10	62
i=3	[15	45	46	58]	90	18	10	62
i=4	[15	45	46	58	90]	18	10	62
i=5	[15	18	45	46	58	90]	10	62
i=6	[10	15	18	45	46	58	90]	62
i=7	[10	15	18	45	46	58	62	90]

直接插入排序

- 先将第一个记录看作是一个有序的记录序列，然后从第二个记录开始，依次将未排序的记录插入到这个有序的记录序列中去，直到整个文件中的全部记录排序完毕。
- i从1到n-1执行：
  - 将a[i] 插入到a[0..i-1]得到从小到大的排列a[0..i]（temp=a[i]； a[i-1]、a[i-2]、…依次与temp比较，若比temp大，则往后移动一个位置，最后一个腾出的位置放入temp值）

直接插入排序

- i=1,...,n-1, A[i]插入到A[0:i-1] 。
- 最好情况下，第i趟插入发生在A[i]处，这时总的比较次数为n-1次；最坏情况下，每一趟插入发生在A[0]处，这时总的比较次数为为n(n-1)/2。
- 第i趟平均需要i/2次比较，因而总的比较次数为1/2+2/2+...+(n-1)/2=n(n-1)/4，时间复杂度为O(n*n)。
- 直接插入排序是稳定的。

折半插入排序

- 折半排序算法：在已经排序的A[0 : i-1]里为A[i]寻找插入位置（可能的插入位置包括哪些？位置x是指将A[i]放入A[x]）。 包括0,1,2,…,i
- 将A[i]与A[m=(i-1)/2] 比较，若A[i]<A[m]则在A[0 : m-1]里寻找插入位置（可能的插入位置包括哪些？） 包括0,1,2,…,m
- 否则在A[m+1 : i-1]里寻找插入位置（可能的插入位置包括哪些？）。包括m+1,m+2,m+3,…,i
- 如此反复，直到确定插入位置。

折半插入排序

0	1	2	3	4	5	6	7
[15	18	45	46	58	90]	10	62

考虑在A[0 : 5]里插入A[6]

- A[6]<A[m=2]
- 在A[0 : 1]里插入A[6] （候选插入位置0,1,2）
- A[6]<A[m=0]
- 在A[0 : -1]里插入A[6] （候选插入位置0）
- 因为0>-1，所以插入点为0

折半插入排序

- 折半插入排序中需要的比较次数只与记录的个数有关，在插入A[i]时，如果i=2^j，则需要恰好j=log_2^i次比较才能确定应该插入的位置；如果2^j <=i<= 2^j +1, 则需要约j+1次比较。因此，n个记录采用折半插入排序总的比较次数约为

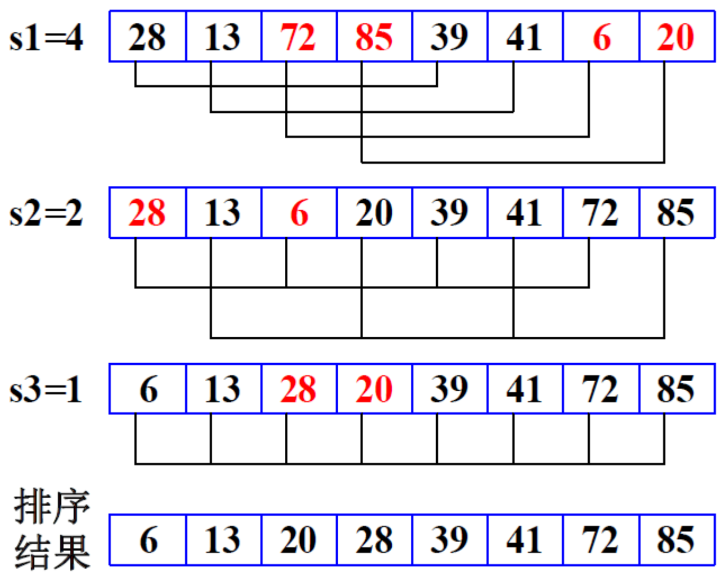
$$\sum_{i=1}^{n}\lceil \log_2 i \rceil = 0+1+2+2+3+3+3+3+\ldots+k+\ldots+k$$

$$=\sum_{j=1}^{k} j2^{j-1}$$

$$=k2^k-2^k+1=n\log_2 n-n+1$$

Shell排序

- 把待排序的n个记录分成si个组，距离为si的记录为一个组，对每一组进行排序。si逐渐变小。



Shell排序

- Shell排序的基本思想：先选定一个整数s1<n，例如s1=n/2，把待排序的n个记录分成s1个组，所有距离为s1的记录为一个组，对每一组进行排序。然后取s2<s1，重复上述分组和排序。当达到si=1时，所有记录排好序。
- 各组内通常用直接插入排序，开始时si值较大，各组记录数少，所以排序较快，si值增大时，由于已经按si-1排好序，排序速度也较快。

Shell排序

```
void ShellSort(ForSort A[], int n, int s)
{   int i, j, k;  ForSort temp;
    for(k=s; k>0; k>>=1)        // 组s逐渐减小
    { //从每组第2个元素开始进行组内插入排序
        for(i=k; i<n; i++)
        {  temp=A[i];  j=i-k;
           while(j>=0 && temp.key<A[j].key)
           {   A[j+k]=A[j];   j-=k; }
           A[j+k]=temp;
        }
    }
}
```

是否是稳定的?

Shell排序

- 我们来试一试

0	1	2	3	4	5	6	7
5	8	8	8	7	5	3	9

- 使用si=4，2，1进行排序结果是?
- 使用si=3，2，1进行排序结果是?

排序

- 5.1 基本概念
- 5.2 插入排序
- 5.3 选择排序

直接选择排序

- 直接选择排序：每次从A[i : n-1]中选出排序码最小的记录A[k]，放在已排序的记录A[0 : i-1]的后面(交换A[i]与A[k])。i从___0___到___n-1___执行该步骤。例

42 32 31 12 25 11 43 10 8

[8] 32 31 12 25 11 43 10 42

[8 10] 31 12 25 11 43 32 42

[8 10 11] 12 25 31 43 32 42

...

[8 10 11 12 25 31 32 42 43]

比较次数的复杂度？
是否是稳定的？

直接选择排序

- 直接选择排序的总比较次数为：

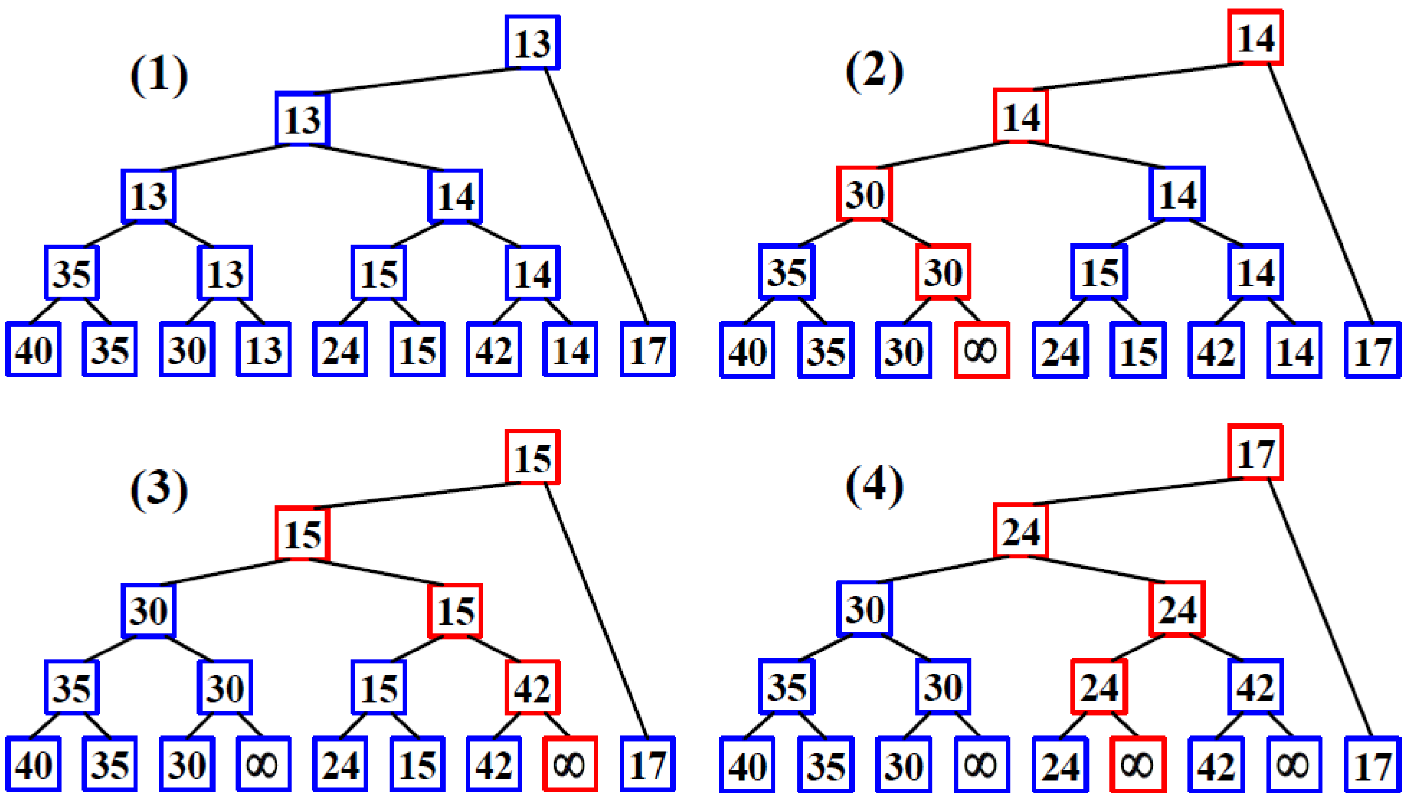
$$\sum_{i=1}^{n-1}(n-i)=n(n-1)/2$$

- 最好情况下，待排序记录已按非递减排好序，此时移动次数为0次； 最坏情况下，待排序记录已按非递增序排好，此时的移动次数为3(n-1)。
- 直接选择排序是不稳定的
  - 考虑5 8 5 2 9

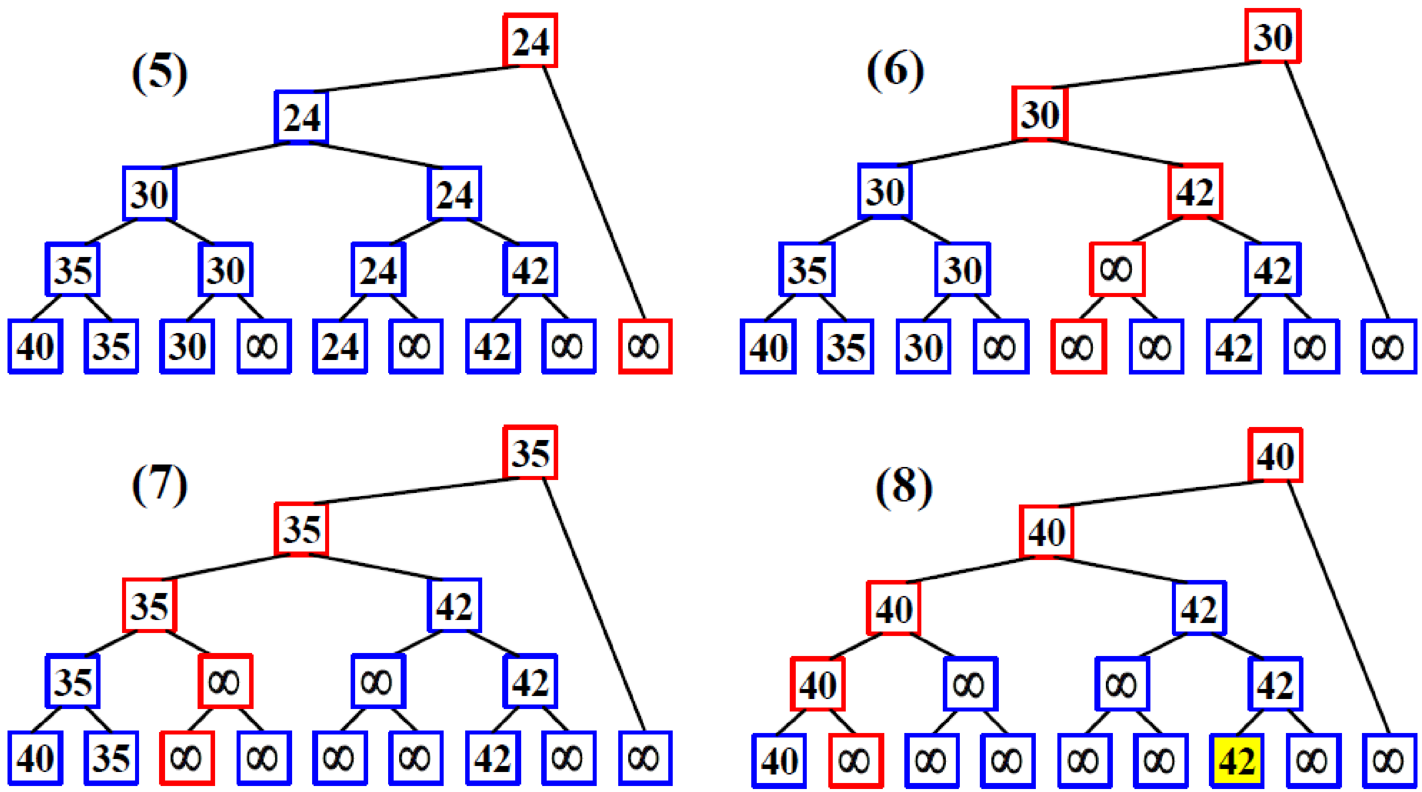
树形选择排序

- 树形选择排序的基本思想：把待排序的n个记录的排序码两两进行比较，取出 ┌n/2 ┐ 个较小的排序码作为作为结果保存下来，这 ┌n/2┐ 个排序码进一步两两进行比较，重复上述过程直到得到最小的排序码。

树形选择排序


(1)
13
13
13 14
35 13 15 14
40 35 30 13 24 15 42 14 17
(2)
14
14
30 14
35 30 15 14
40 35 30 ∞ 24 15 42 14 17
(3)
15
15
30 15
35 30 15 42
40 35 30 ∞ 24 15 42 ∞ 17
(4)
17
24
30 24
35 30 24 42
40 35 30 ∞ 24 ∞ 42 ∞ 17

树形选择排序


(5)
24
24
30
24
35
30
24
42
40
35
30
∞
24
∞
42
∞
∞
(6)
30
30
30
42
35
30
∞
42
40
35
30
∞
∞
∞
42
∞
∞
(7)
35
35
35
42
35
∞
∞
42
40
35
∞
∞
∞
∞
42
∞
∞
(8)
40
40
40
42
40
∞
∞
42
40
∞
∞
∞
∞
∞
42
∞
∞

树形选择排序

- 用数组保存树
  - 总共需要n-1次选择
  - 第1次选择进行n-1次比较
  - 第2次-第n-1次选择每次需要 $\lceil \log_2^n \rceil$ 次比较
- 总共需要

$$n-1+(n-2)\lceil \log_2 n \rceil \approx n \cdot \log_2 n$$

排序

- 5.1 基本概念
- 5.2 插入排序（直接插入，折半插入，shell排序*）
- 5.3 选择排序（直接选择，树形选择）
- 5.4 交换排序（冒泡排序，快速排序）

冒泡排序

- 对A[0:n-1]进行一趟冒泡:
  - for j=0, ..., n-2：若A[j]>A[j+1]则两者交换

```
20 30 10 45 25 22 55 50 //A[0]与A[1]比较，未交换
20 10 30 45 25 22 55 50 // A[1]与A[2]比较，交换
20 10 30 45 25 22 55 50 // A[2]与A[3]比较，未交换
20 10 30 25 45 22 55 50 // A[3]与A[4]比较，交换
20 10 30 25 22 45 55 50 // A[4]与A[5]比较，交换
20 10 30 25 22 45 55 50 // A[5]与A[6]比较，未交换
20 10 30 25 22 45 50 55 // A[6]与A[7]比较，交换
```

冒泡排序

- 对A[0:n-1]进行一趟冒泡：
  - for j=0, ..., n-2：若A[j]>A[j+1]则两者交换
  - 导致A[0:n-1]的最大者被交换到了A[n-1]处
- 冒泡排序
  - 对A[0:n-1]进行一趟冒泡
  - 对A[0:n-2]进行一趟冒泡
  - 对A[0:n-3]进行一趟冒泡
  - … …
  - 对A[0:1]进行一趟冒泡

冒泡排序

- 对A[0:n-1]进行冒泡排序：
  - for i=n-1, ..., 1 对A[0:i]进行一趟冒泡
- 20 30 10 45 25 22 55 50 的冒泡排序过程为：

```
20 10 30 25 22 45 50 [55]   //第1趟A[0:7]
10 20 25 22 30 45 [50 55]   //第2趟A[0:6]
10 20 22 25 30 [45 50 55]   //第3趟A[0:5]
10 20 22 25 [30 45 50 55]   //第4趟A[0:4]
-------------------------------------------
10 20 22 [25 30 45 50 55]   //第5趟A[0:3]
10 20 [22 25 30 45 50 55]   //第6趟A[0:2]
10 [20 22 25 30 45 50 55]   //第7趟A[0:1]
```

冒泡排序

- 最好情况下比较次数和移动次数…
- 最坏情况下比较次数和移动次数…
- 冒泡排序是稳定的吗？

冒泡排序

- 最好情况下，n个记录已经按非递减排好序，此时只需要一趟冒泡，比较次数为n-1次，移动次数为0次。
- 最坏情况下， n个记录已经按非递增排好序，此时需要n-1趟冒泡，比较次数和移动次数分别为

$$\sum_{i=1}^{n-1} i = n(n-1)/2, \quad 3\sum_{i=1}^{n-1} i = 3n(n-1)/2$$

- 冒泡算法是稳定的

快速排序

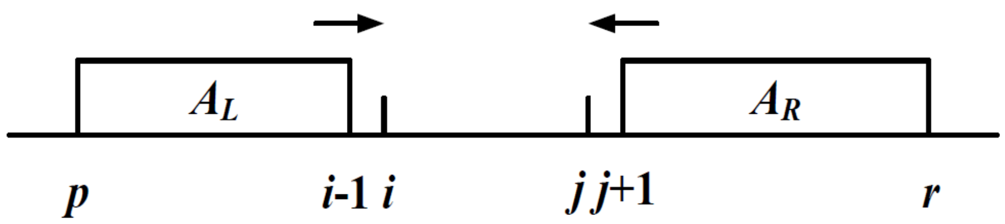
- 从待排序记录中任选一个记录，以这个记录的排序码作为中心值，将其它记录划分为两个部分，第一部分包含所有排序码小于等于中心值的记录，第二部分包含所有排序码大于中心值的记录。
- 对这两个部分采用同样的方法进行处理，直到每个部分为空或只含一个记录为止。

28	13	72	85	39	41	6	20

20	13	6	28	39	41	85	72

快速排序

- 用首元素 x 作为划分标准，将输入数组 A 划分成不超过 x 的元素构成的数组A_L, 大于 x 的元素构成的数组 A_R, 其中 A_L, A_R 从左到右存放在数组 A 中
- 递归地对子问题 A_L 和 A_R 进行排序，直到子问题规模为 1 时停止



快速排序

初始状态： 28 13 72 85 39 41 6 20

取出第1个记录： 13 72 85 39 41 6 20

第1次比较，在后端进行，28>20，交换

第1次交换： 20 13 72 85 39 41 6

第2次比较，在前端进行，13<28，不交换

第3次比较，继续在前端进行，72>28，交换

第2次交换： 20 13 85 39 41 6 72

第4次比较，在后端进行，28>6，交换

快速排序

第3次交换： 20 13 6 85 39 41 □ 72

第5次比较，在前端进行，85>28，交换

第4次交换： 20 13 6 □ 39 41 85 72

第6次比较，在后端进行，28<41，不交换

第7次比较，继续在后端进行，28<39，不交换

快速排序

- 快速排序QSort(A, low, high)

low high

28	13	72	85	39	41	6	20

i=low;j=high;temp=A[i]

28

	13	72	85	39	41	6	20

i j

快速排序

- 快速排序QSort(A, low, high)

low　　　　　　　　　　　　high

20	13	72	85	39	41	6	

i　　　　　　　　　　　　j

28

//找到第1个小于等于temp的A[j]
//并将其放入A[i], i++
while(___&&_______) _____;
if(i<j) {_______; _______;}

	13	72	85	39	41	6	20

i　　　　　　　　　　　　j

快速排序

- 快速排序QSort(A, low, high)

low high

20	13	72	85	39	41	6	

i j

28

```
//找到第1个小于等于temp的A[j]
//并将其放入A[i], i++
while(i<j&&A[j]>temp) j--;
if(i<j)  {A[i]=A[j]; i++;}
```

	13	72	85	39	41	6	20

i j

快速排序

- 快速排序QSort(A, low, high)

low high

20	13	72	85	39	41	6	

i j

28

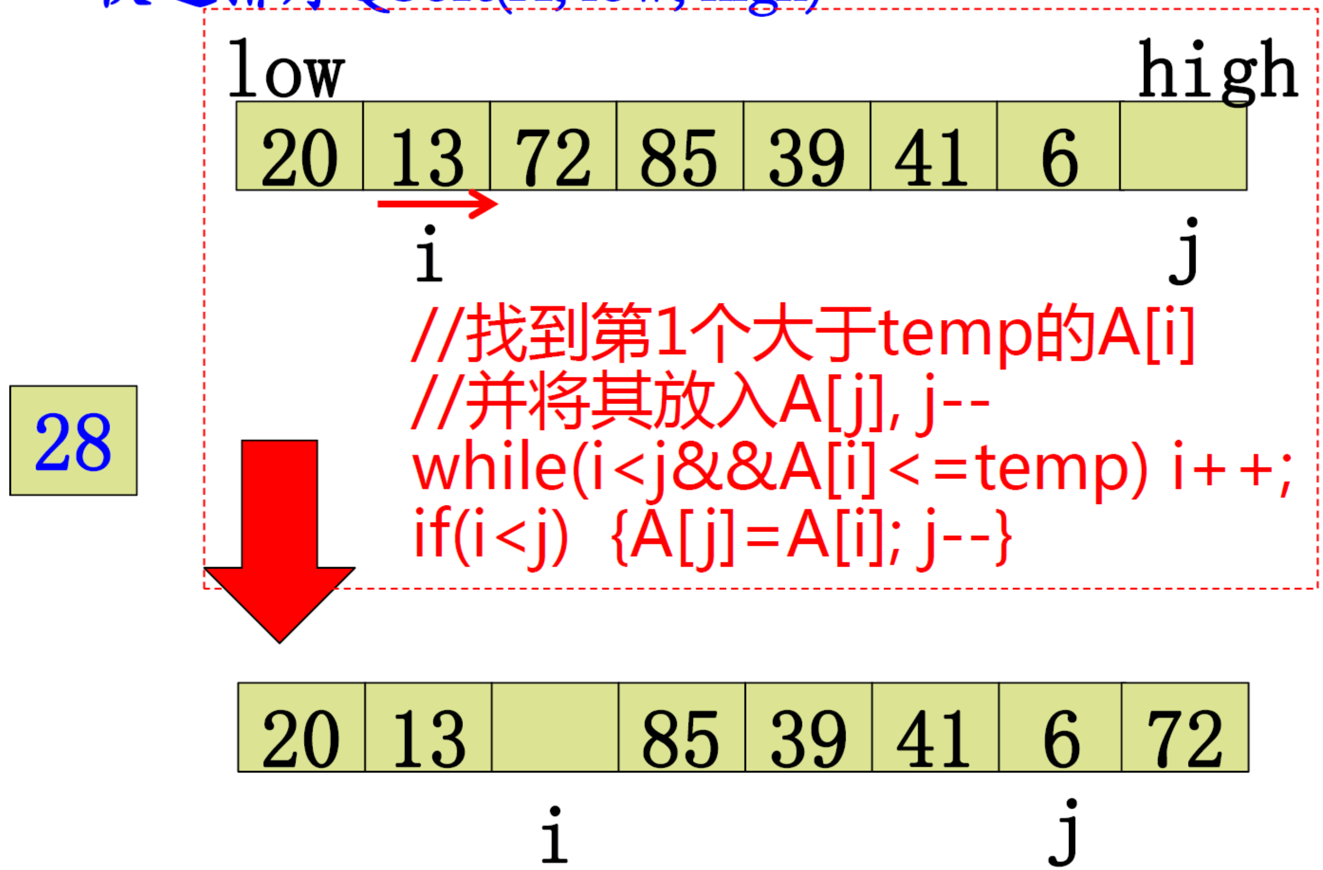
```
//找到第1个大于temp的A[i]
//并将其放入A[j], j--
while(____&&__________) ___;
if(i<j) {__________; ____}
```

20	13		85	39	41	6	72

i j

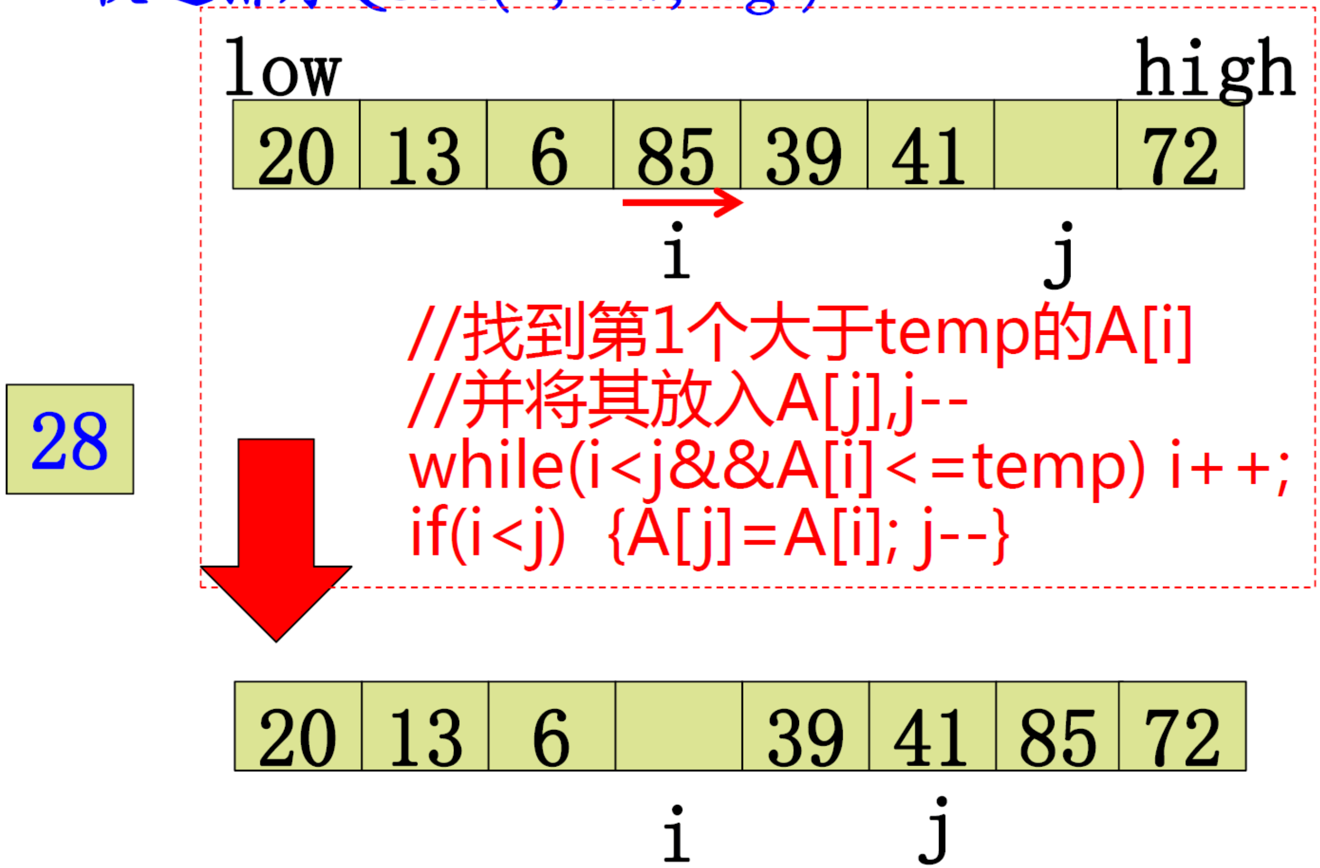
快速排序

- 快速排序QSort(A, low, high)



快速排序

- 快速排序QSort(A, low, high)



快速排序

- 快速排序QSort(A, low, high)

low　　　　　　　　　　　　　　　　　　high

20	13	6		39	41	85	72

ij

28

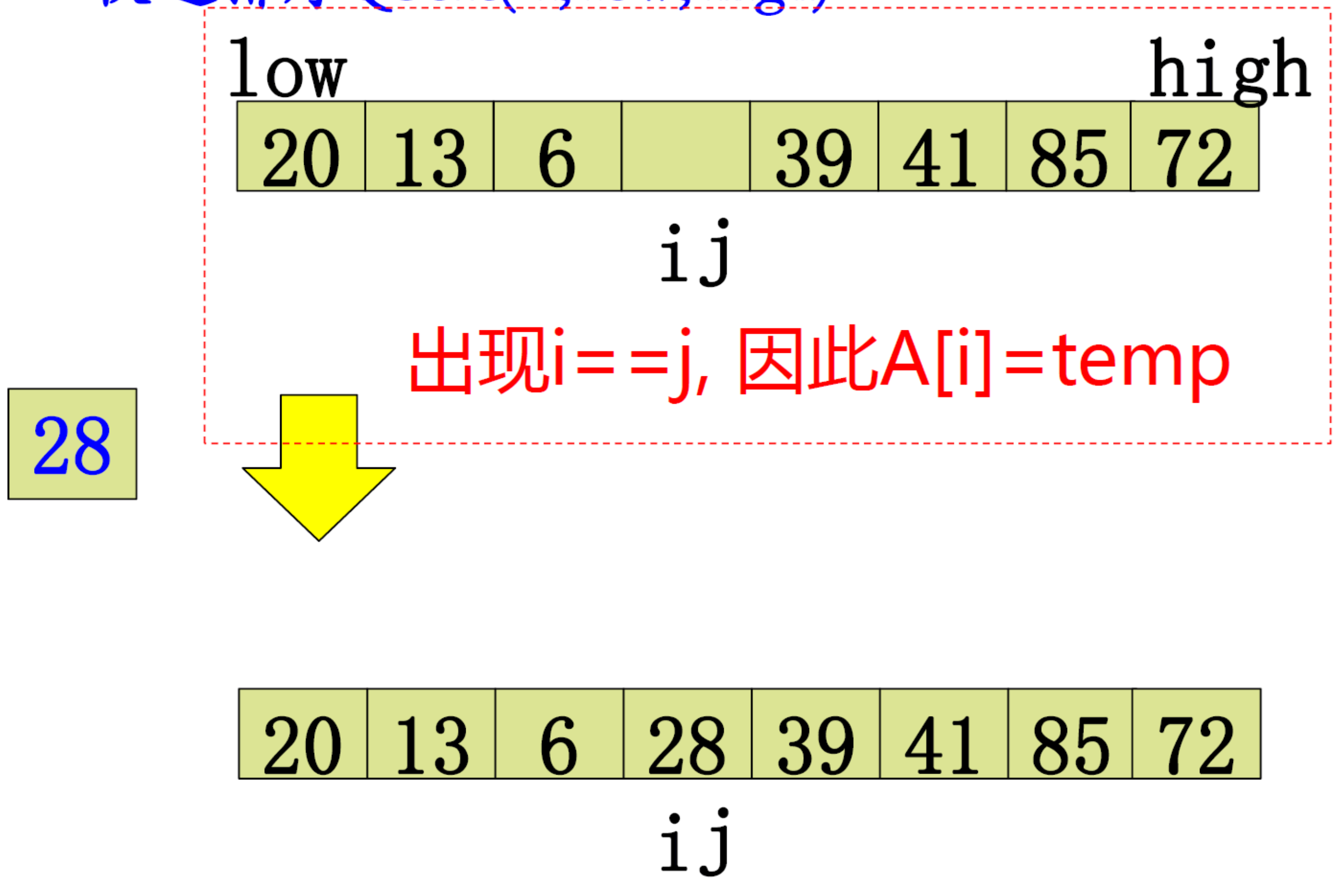
//找到第1个小于等于temp的A[j]
//并将其放入A[i],i++
while(i<j&&A[j]>temp) j--;
if(i<j) {A[i]=A[j]; i++;}

20	13	6		39	41	85	72

i　　　j

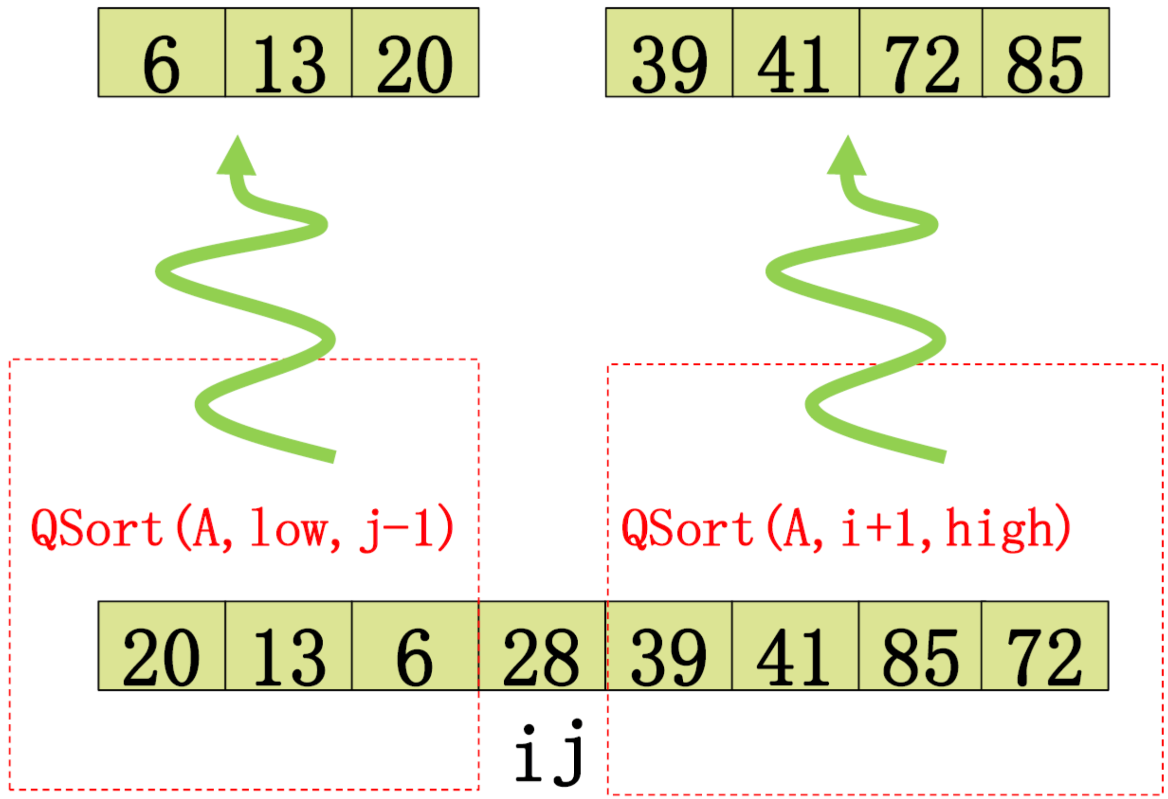
快速排序

- 快速排序QSort(A, low, high)



快速排序

- 快速排序QSort(A, low, high)



快速排序

```
void QSort(int A[], int low, int high)
{   int i,j;
    if(__________) return;
    i=_____; j=_____; temp=_____;
    while(_____)
    {   while(__________________) j--;
        if(i<j)  {A[i]=A[j]; i++;}
        while(__________________) i++;
        if(i<j)  {A[j]=A[i]; j--;}
    }
    _________; _______________; ___________________;
}
void QuickSort(int A[], int n)
{   _______________;  }
```

快速排序

```
void QSort(int A[], int low, int high)
{   int i,j;
    if(low>=high) return;
    i=low; j=high; temp=A[i];
    while(i<j)
    {   while(i<j&&A[j]>temp) j--;
        if(i<j)  {A[i]=A[j]; i++;}
        while(i<j&&A[i]<=temp) i++;
        if(i<j)  {A[j]=A[i]; j--;}
    }
    A[i]=temp; QSort(A, low, j-1); QSort(A, i+1, high);
}
void QuickSort(int A[], int n)
{   Qsort(A, 0, n-1);   }
```

快速排序

- 是否是稳定的?
- 什么情况下效率最高?
- 如何选择中心值?

快速排序

- 快速排序是不稳定的
- 最好情况下每次选取的中心值恰好将其它记录分成大小相等(或相差一个记录)的两个部分
  - 第1遍时，经过大约n次(实际上为n-1次)比较，产生两个大小约为n/2的子文件；
  - 第2遍对每个子文件经过约n/2次比较产生4个大小约为n/4的子文件，比较次数约为2*(n/2)次；
  - …
  - $n/1+2(n/2)+4(n/4)+\ldots+n(n/n)=n\log n$
- 最坏情况下，待排序文件已经排好序

$$(n-1)+(n-2)+(n-3)+\ldots+1=n(n-1)/2$$

快速排序

- 平均时间复杂度 设输入数组首元素排好序后的正确位置处在 1,2,...,n 各种情况的概率都为 1/n
  - 首元素在位置 1： $T(0)$, $T(n-1)$
  - 首元素在位置 2： $T(1)$, $T(n-2)$
  - ...
  - 首元素在位置 n： $T(n-1)$, $T(0)$
- 子问题工作量 2[$T(1)+T(2)+\ldots+T(n-1)$]
- 划分工作量 n-1

快速排序

- 平均时间复杂度

$$T(n)=\frac{1}{n}\{\sum_{k=1}^{n-1}[T(k)+T(n-k)]+n(n-1)\}$$

$$\Rightarrow \boxed{\begin{aligned} &T(n)=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+n-1 \\ &T(1)=0 \end{aligned}}$$

快速排序

- 差消化简

$$nT(n) = \boxed{2\sum_{k=1}^{n-1} T(k)} + n(n-1)$$

$$(n-1)T(n-1) = \boxed{2\sum_{k=1}^{n-2} T(k)} + (n-1)(n-2)$$

$\Rightarrow$ $\boldsymbol{nT(n)-(n-1)T(n-1)=2T(n-1)+n(n-1)-(n-1)(n-2)}$

$\boldsymbol{nT(n) = (n+1)T(n-1)+2(n-1)}$

$\Rightarrow$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}$$

快速排序

$$\frac{T(n)}{n+1}=\frac{T(n-1)}{n}+\frac{4}{n+1}-\frac{2}{n}=\ldots$$

$$=4\left(\frac{1}{n+1}+\frac{1}{n}\ldots+\frac{1}{3}\right)-2\left(\frac{1}{n}+\frac{1}{n-1}\ldots+\frac{1}{2}\right)+\frac{T(1)}{2}$$

$$=\Theta(\log n)$$

$$\Rightarrow \quad \boldsymbol{T(n)=\Theta(n\log n)}$$

快速排序

- 是否是稳定的？不稳定
- 什么情况下效率最高？每次拆分均衡
- 如何选择中心值？

基数排序

- 初始：114 179 572 835 309 141 646 520

第一次按个位数分配	第二次按十位数分配
第一组：520	第一组：309
第二组：141	第二组：114
第三组：572	第三组：520
第四组：	第四组：835
第五组：114	第五组：141，646
第六组：835	第六组：
第七组：646	第七组：
第八组：	第八组：572，179
第九组：	第九组：
第十组：179 309	第十组：

第一次收集：520，141，572，114，835，646，179，309。

第二次收集：309，114，520，835，141，646，572，179。

基数排序

第三次按百位数分配

第一组：

第二组：114，141，179

第三组：

第四组：309

第五组：

第六组：520，572

第七组：646

第八组：

第九组：835

第三次收集114，141，179，309，520，572，646，835

如果按百位数、十位数、个位数的次序，会得到什么结果？

基数排序

- 将排序码k_i看作是一个d元组 $(k_i^0, k_i^1, \ldots, k_i^{d-1})$其中每个分量有r种可能的取值：$c_0,c_1,\ldots,c_{r-1}$， r称为基数，当排序码为十进制数时，r=10，当排序码是大写英文字符串时，r=26。
- 排序时，先按最低位k_i^{d-1}的值从小到大将待排序的记录分配到r个队列中，队列中的记录按分配进来的先后排放，然后依次收集这些记录。再将收集起来的序列按k_i^{d-2}进行分配和收集，如此反复，直到按k_i^0进行分配后，收集起来的序列就是排序后的有序序列。

基数排序

- 最低位优先:从低位往高位进行分配和收集
- 最高位优先:从高位往低位进行分配和收集

基数排序

- 执行基数排序算法时，可采用顺序存储结构，用一个数组存放待排序的n个记录，用r个数组存放分配时所需的r个队列，每个队列最大需要n个记录的空间。每分配一次，需要移动___个记录，每收集一次需要移动___个记录，d趟分配和收集需要移动_____个记录，且需要____个附加的记录空间。

基数排序

- 执行基数排序算法时，可采用顺序存储结构，用一个数组存放待排序的n个记录，用r个数组存放分配时所需的r个队列，每个队列最大需要n个记录的空间。每分配一次，需要移动n个记录，每收集一次也需要移动n个记录，d趟分配和收集需要移动2.d.n个记录，且需要r.n个附加的记录空间。

基数排序

- 采用链式存储结构，将移动记录改为修改指针，则可克服时间和空间消耗问题
- 复杂性分析

基数排序的时间复杂性取决于基数和排序码的长度。每执行一次分配和收集，队列初始化需要______的时间，分配工作需要_____的时间，收集工作需要_____的时间，即每趟需要________的时间，总共执行d趟，共需________的时间。

需要的附加存储空间：每个记录增加一个指针共需要O(n)的空间，以及需要一个分配队列占用O(r)的空间，总的附加空间为O(n+r)。

基数排序

- 采用链式存储结构，将移动记录改为修改指针，则可克服时间和空间消耗问题
- 复杂性分析

基数排序的时间复杂性取决于基数和排序码的长度。每执行一次分配和收集，队列初始化需要O(r)的时间，分配工作需要O(n)的时间，收集工作需要O(r)的时间，即每趟需要O(n+2r)的时间，总共执行d趟，共需O(d(n+2r))的时间。

需要的附加存储空间：每个记录增加一个指针共需要O(n)的空间，以及需要一个分配队列占用O(r)的空间，总的附加空间为O(n+r)。

归并排序

- 将两个或者多个已有序的序列，合并成一个有序序列的复杂度？ O(n)

13 28 72 85　　6 20 39 41

归并排序

- 例：28，13，72，85，39，41，6，20

初始： [28] [13] [72] [85] [39] [41] [6] [20]

第1趟归并： [13 28] [72 85] [39 41] [6 20]

第2趟归并： [13 28 72 85] [6 20 39 41]

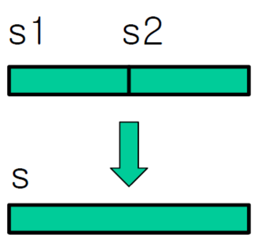
第3趟归并： [6 13 20 28 39 41 72 85]

归并排序

- 设计代码的自顶向下逻辑
  - 待排序的数组为ForSort A[]，长度为n
  - 两个有序子序列归并为一个有序序列TwoWayMerge(???)
  - 一趟归并： 每两个相邻有序子序列归并OnePassMerge(???)
  - MergeSort(ForSort A[], int n)归并排序

归并排序

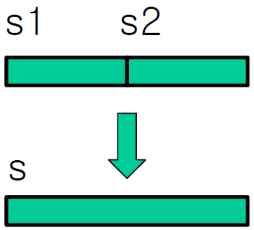
```
void TwoWayMerge(ForSort Dst[], ForSort Src[],
                               int s, int e1, int e2)
//两个子文件归并为一个子文件:
//源数组中[s:e1]与[e1+1:e2]归并到目的数组中[s:e2]
{   int s1, s2;
```




```
}
```

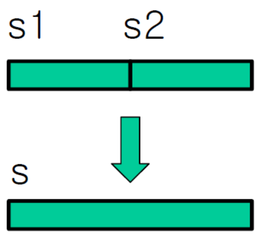
归并排序

```
void TwoWayMerge(ForSort Dst[], ForSort Src[],
                                    int s, int e1, int e2)
//两个子文件归并为一个子文件:
//源数组中[s:e1]与[e1+1:e2]归并到目的数组中[s:e2]
{   int s1, s2;
    for(s1=s, s2=e1+1; s1<=e1&&s2<=e2; )
        if(________________________)
            Dst[s++]=Src[s1++];
        else
            Dst[s++]=Src[s2++];
    if(___________)
        memcpy(&Dst[s], &Src[s1], (e1-s1+1)*sizeof(ForSort);
    else
        memcpy(&Dst[s], &Src[s2], (e2-s2+1)*sizeof(ForSort);
}
```



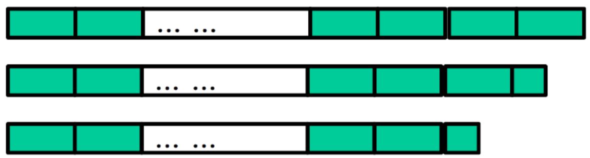
归并排序

```
void TwoWayMerge(ForSort Dst[], ForSort Src[],
                                int s, int e1, int e2)
//两个子文件归并为一个子文件:
//源数组中[s:e1]与[e1+1:e2]归并到目的数组中[s:e2]
{   int s1, s2;
    for(s1=s, s2=e1+1; s1<=e1&&s2<=e2; )
        if(Src[s1].key<=Src[s2].key)
            Dst[s++]=Src[s1++];
        else
            Dst[s++]=Src[s2++];
    if(s1<=e1)
        memcpy(&Dst[s], &Src[s1], (e1-s1+1)*sizeof(ForSort);
    else
        memcpy(&Dst[s], &Src[s2], (e2-s2+1)*sizeof(ForSort);
}
```



归并排序

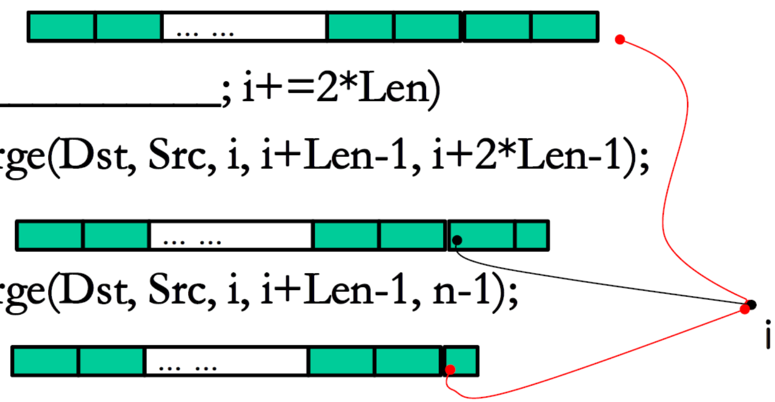
```
void OnePassMerge(ForSort Dst[], ForSort Src[], int Len, int n)
//一趟归并： 每两个相邻子文件归并, 子文件长度为Len
{   int i;
```




```
}
```

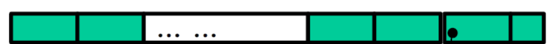
归并排序

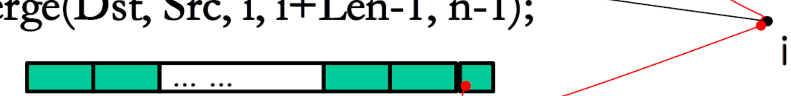
```
void OnePassMerge(ForSort Dst[], ForSort Src[], int Len, int n)
//一趟归并：每两个相邻子文件归并，子文件长度为Len
{   int i;
    for(______; _______________; i+=2*Len)
        TwoWayMerge(Dst, Src, i, i+Len-1, i+2*Len-1);
    if(___________)
        TwoWayMerge(Dst, Src, i, i+Len-1, n-1);
    else
        memcpy(&Dst[i], &Src[i], (n-i)*sizeof(ForSort));
}
```



归并排序

```
void OnePassMerge(ForSort Dst[], ForSort Src[], int Len, i
   nt n)
//一趟归并： 每两个相邻子文件归并, 子文件长度为Len
{   int i;
    for(i=0; i+2*Len<=n; i+=2*Len)
        TwoWayMerge(Dst, Src, i, i+Len-1, i+2*Len-1);
    if(i<n-Len)
        TwoWayMerge(Dst, Src, i, i+Len-1, n-1);
    else
        memcpy(&Dst[i], &Src[i], (n-i)*sizeof(ForSort));
}
```





归并排序

```
void MergeSort(ForSort A[], int n)//归并排序
{   int k=1;
    ForSort* B=(ForSort*)malloc(sizeof(ForSort)*n);
    while(k<n)
    {   OnePassMerge(B, A, k, n);
        k<<=1;
       if(k>=n)   memcpy(A,B,n*sizeof(ForSort));
       else  { OnePassMerge(A,B,k,n);  k<<=1; }
    }
}
```

归并排序

- 复杂性分析
  - 对n个记录进行归并排序，需要调用函数OnePassMerge约log2n次，OnePassMerge的时间复杂性为O(n)，最后可能执行n次移动，总的时间复杂性为O(n*log2n)。需要n个附加存储空间。
  - 可以并行