jyang
@cs.ecnu.edu.cn

第三章
内部排序

1 插入排序

2 选择排序

3 冒泡排序

4 希尔排序

5 合并排序

6 快速排序

7 基数排序

第三章
内部排序

0 引言

对于由n结点组成的线性表，按照结点中某个字段的值的递增或递减次序，重新排列表中各个结点的过程，称为**排序**。

如果没有特别指明，排序总是按关键字的值的递增次序排列表中结点。

如果表中关键字的值相等的结点经过某种排序方法进行排序之后，仍能保持它们在排序之前的相对次序；也就是说，在排序前，如果$i<j$，且$k_i=k_j$，在经过排序之后，a_i仍在a_j之前，则称这种**排序方法是稳定**的。否则，称这种**排序方法是不稳定的**。

第三章
内部排序

0 引言

排序方法可以分成：内部排序，外部排序。

内部排序：排序期间全部结点都存放在内存，并在内存中调整等待排序的结点的存放位置。
只有两个基本操作：比较 ，交换

外部排序：排序期间大部分结点存放在外存中，排序过程借助内存，调整那些存放在外存、等待排序的结点的存放位置。

0 引言

简单排序算法

Bubble
Selection
insertion

各种排序方法的比较

排序算法	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性	算法思想
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	O（1）	稳定	假设前面的序列已排序，后面的结点插入到正确的位置。（已排序序列向后逐渐扩大。）
选择排序	$O(n^2)$	$O(n^2)$	$O(n)$	O（1）	不稳定	选当前序列中最小的结点，和序列中的第一个交换。（未排序序列向后逐渐缩小）
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	O（1）	稳定	两两比较，位置不正确则交换。直到某次比较不发生交换
希尔排序	取决于增量序列	取决于增量序列	$O(n)$	O（1）	不稳定	插入排序的扩展
合并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定	
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	
基数排序	O（d*n）	O（d*n）	O（d*n）	O（r*d）	稳定	

3 冒泡排序

假设待排序的n个结点的序列为 α_0，α_1，…，α_{n-1}，起始时排序范围是从 α_0 至 α_{n-1}。

冒泡排序是在当前排序范围内，自上而下对相邻的两个结点依次进行比较，让值较大的结点往下移（下沉），让值较小的结点往上移（上冒）。

当自上而下在当前排序范围内执行一遍比较之后，假设最后往下移的结点是 α_j，则下一遍的排序范围为从 α_0 至 α_j。

在整个排序过程中，最多执行（n-1）遍，但执行的遍数可能少于（n-1），这是因为在执行某一遍的各次比较没有出现结点交换时，就不用进行下遍的比较。

3 冒泡排序

```
void bubble_sort(int a[],int n)
{
        int i,j;
        int t;  //for swaps
        n--;
        while(n>0) //每次处理范围从a[0]到a[n]，n：上一轮循环中最后往下交换的结点的位置
        {       j=0;
                for(i=0;i<n;i++)
                        if(a[i]>a[i+1])
                        {       t=a[i];
                                a[i]=a[i+1];
                                a[i+1]=t;
                                j=i;
                        }
                n=j;    //如果某遍比较中没有发生过交换，则n将被置0，循环结束；
        }
}
```

//一次for循环，完成从0到n的一遍检查及交换

3 冒泡排序

假设待排序的结点有n个，那么冒泡排序最多执行（n-1）遍。
第一遍最多执行（n-1）次比较，
第2遍最多执行（n-2）次比较，……第n-1遍执行1次比较。

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

因此整个排序过程最多执行n(n-1)/2次比较。

最少：(如果待排序的n个结点已排序，那么只要执行一遍比较就可以结束排序，进行（n-1）次比较。

Time Complexity (Comparisons) $O(n^2)$

3 冒泡排序

假设待排序的结点有n个，冒泡排序最多执行多少次交换?

整个排序过程最多执行n(n-1)/2次交换。
Time Complexity – Swaps $O(n^2)$

冒泡排序是否稳定?

冒泡排序是稳定的。

Time Complexity： 最坏$O(n^2)$，最好$O(n)$，平均$O(n^2)$
空间：O（1）

2 选择排序

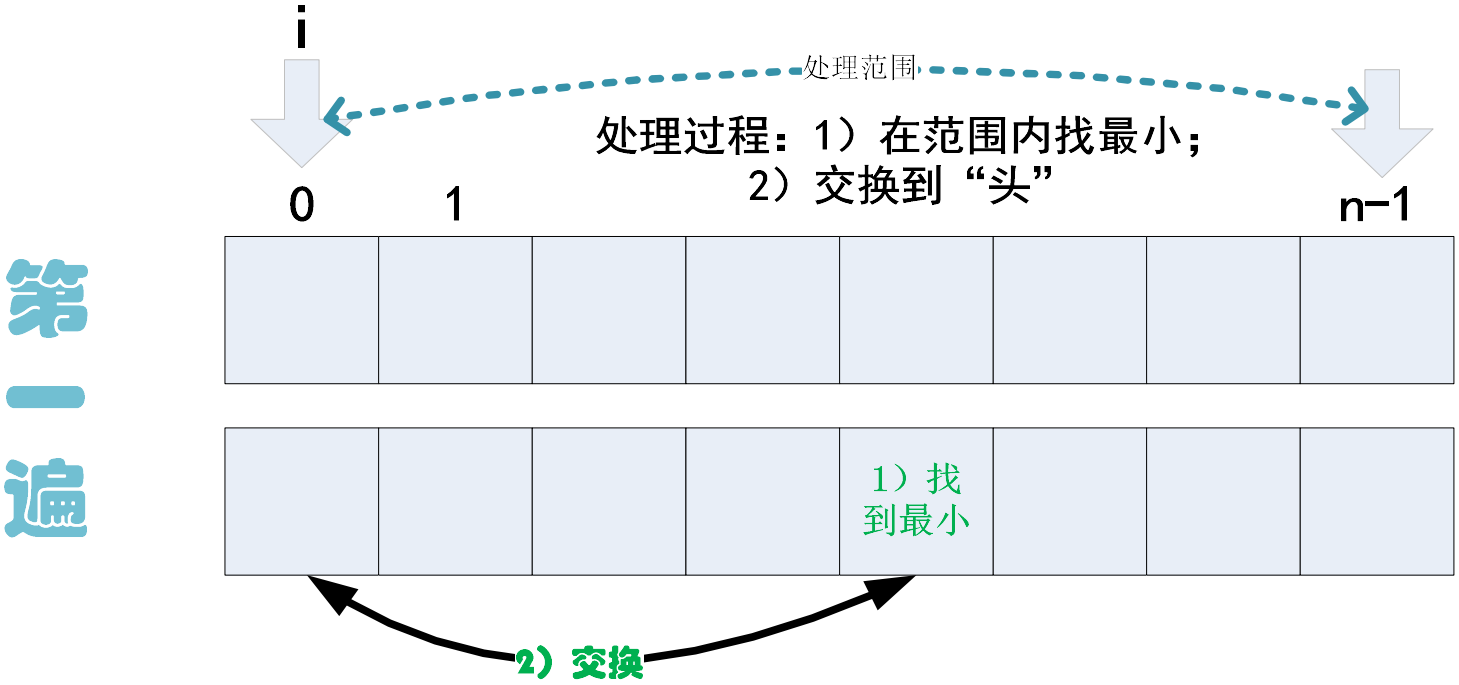
假设待排序的n个结点的序列为 α_0，α_1，…，α_{n-1}，
我们依次对i=0 ，1，…，n-2分别执行如下的选择步骤：

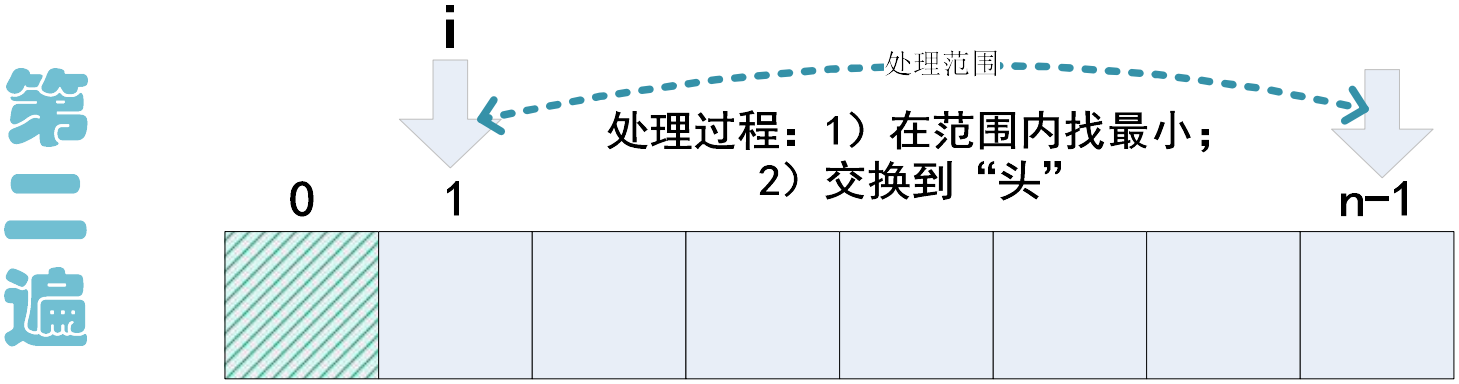
在 α_i，α_{i+1}，…，α_{n-1} 中选择一个最小的结点 α_k，
然后将 α_k 与 α_i 交换。

在执行上述的选择步骤（n-1）遍之后，完成排序。

选择：不断地选择剩余元素之中的最小者。

2 选择排序


i
处理范围
处理过程：1）在范围内找最小；
2）交换到“头”
0
1
n-1
第一遍
1）找
到最小
2）交换


i
处理范围
处理过程：1）在范围内找最小；
2）交换到“头”
0
1
n-1
第二遍

2 选择排序

```
Void selection_sort(int a[], int n)
        {       int i,j,k;
                int t;
                for(i=0;i<n-1;i++)
                {
                //在i到n-1的范围里面，找到最小的;和位置为i的结点交换
                k=i;
                for(j=i+1;j<n;j++)   if(a[k]>a[j])   k=j;

                SWAP a[i] a[k];
                }
        }
```

2 选择排序

```
Void selection_sort(int a[], int n)
        {       int i,j,k;
                int t;
                for(i=0;i<n-1;i++)
                {       k=i;  //用k记录此轮循环中最小结点的下标
                        for(j=i+1;j<n;j++)   if(a[k]>a[j])   k=j;
                        t=a[i];
                        a[i]=a[k];
                        a[k]=t;
                }
        }
```

2 选择排序

选择排序的比较次数：
当外循环i=0时，内循环要做（n-1）次比较。
当外循环i=1时，内循环要做（n-2）次比较。

当外循环i=n-2时，内循环要做1次比较。

因此，比较次数为$\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$　　$O(n^2)$

2 选择排序

选择排序的交换次数：

每发生一次交换，把一个结点放到了它最终的位置。总交换次数N。

最坏情况：O(N)

选择排序的方法是否稳定？

选择排序的方法是不稳定的。

2 选择排序

选择排序算法的特点：

运行时间和输入无关。
数据的移动最少：N次交换，交换次数和数组的大小是线性关系。

1 插入排序

假设待排序的n个结点的序列为α_0，α_1，…，α_{n-1}，我们依次对i=1，2，…，n−1 分别执行下面的插入步骤：

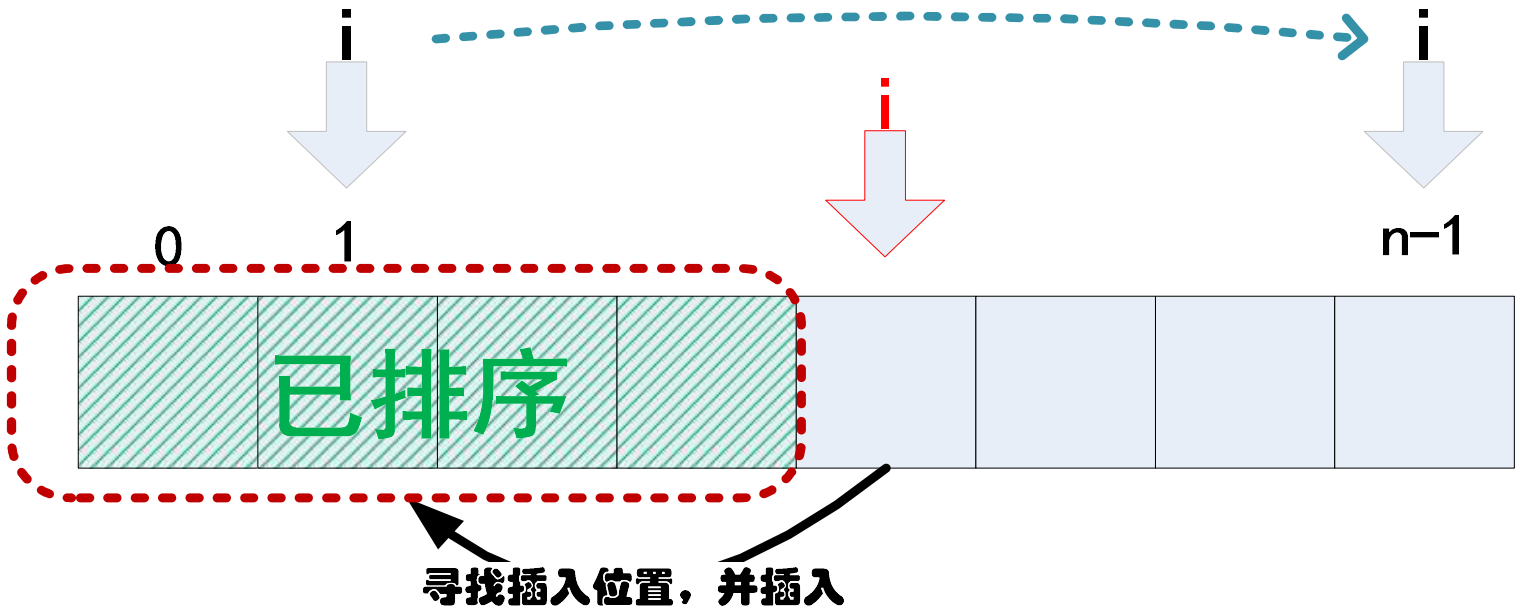
假设α_0，α_1，…，α_{i-1}，已排序，故有$\alpha_0 \leqslant \alpha_1 \leqslant \cdots \leqslant \alpha_{i-1}$。

首先，将α_i送t，然后将t依次与α_{i-1}，α_{i-2}，…进行比较，将比t大的结点依次右移一个位置，直到发现某个j（$0 \leqslant j \leqslant i-1$）成立$\alpha_j \leqslant t$，则把t送$\alpha_{j+1}$；如果这样的$\alpha_j$不存在，那么在比较过程中$\alpha_{i-1}$，$\alpha_{i-2}$，…，$\alpha_0$都依次右移一个位置，此时将t送$\alpha_0$。

经过执行上述插入步骤（n−1）遍后，α_0，…，α_{n-1}完成排序。

1 插入排序

插入排序的算法描述：



1 插入排序

插入排序的算法描述：

```
void insert_sort(int a[],int n)
{       int i,j;
        int t;
        for(i=1;i<n;i++)
                {       t=a[i];
                        for(j=i-1;j>=0&&t<a[j];j--)      a[j+1]=a[j];
                        a[j+1]=t;
                }
}
```

1 插入排序

插入排序的比较次数：
当待排序的结点序列已从小到大排了序时，对于j循环，每执行一次，只要进行一次结点比较，故整个排序过程只进行（n-1）次比较。

当待排序结点序列是从大到小排列，对于j循环，每执行一次，需进行j次比较，故整个排序过程需要进行：

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

比较次数：最少为（n-1），最大为n(n-1)/2　$O(n^2)$

1 插入排序

插入排序的交换次数：

因为需要移动节点，所以最坏情况下N*(N-1)/2　$O(n^2)$

插入排序是否稳定？

插入排序是稳定的。

插入排序所需的时间取决于输入中元素的初始顺序。当倒置的元素的数量很少时，插入排序效率更高。

4 希尔排序

在插入排序时，比较相邻的结点，一次比较最多把结点移动一个位置。如果对位置相隔较大距离的结点进行比较，使得结点在比较后能够一次跨过较大的距离。这样处理可以把值较小的结点尽快往前移动。

希尔排序（Diminishing increment sort 递减增量排序）

希尔排序：首先给定一组严格递减的正整数”增量”，$d_0,d_1,\ldots,d_{t-1}$，且取$d_{t-1}=1$.

对于i=0,1,……，t-1,依次进行下面各遍的处理：
将当前序列中的结点，按当前增量d_i分组，每组中的结点的下标相差d_i，对每组中的结点用插入排序法排序。

4 希尔排序

例：递减增量序列如下： 增量 4，3，2，1

46 26 22 68 48 42 36 84 66。

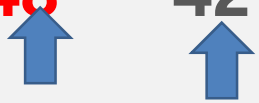
第一次，增量为4 ，记录在变量*h*中
按当前增量h分组，每组中的结点的下标相差h，对每组中的结点用插入排序法排序。

4 希尔排序

第一次，增量为4，记录在变量*h*中

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66



对 a[0],a[4]组成的子序列进行插入排序，a[0]被认为是已经排好序的。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

.....

对 a[1],a[5]组成的子序列进行插入排序，a[1]被认为是已经排好序的。

4 希尔排序

第一次，增量为4，记录在变量*h*中

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	**48**	42	36	84	**66**

对 a[0],a[4]，a[8]组成的子序列进行插入排序，a[0]a[4]被认为是已经排好序的

后端位置从a[4]逐一后移， 用程序语句可以表达为：

```
for(j= h;j<n; j++)  ……
```

4 希尔排序

第二次，增量为3，记录在变量h中

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

对 a[0],a[3]组成的子序列进行插入排序，a[0]被认为是已经排好序的。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

.....

4 希尔排序

第二次，增量为3，记录在变量stepLength中

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

对 a[0]a[3]a[6]组成的子序列进行插入排序，
a[0]a[3]在之前的程序执行中已排好序

```
插入排序：{    t=a[i];
               for(j=i-1;j>=0&&t<a[j]; j--)    a[j+1]=a[j];
               a[j+1]=t;
           }
```

4 希尔排序

第二次，增量为3，记录在变量h中

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
36	26	22	46	48	42	68	84	66

对 a[0] a[3]a[6]组成的子序列进行插入排序，
a[0]a[3]在之前的程序执行中已排好序

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
36	26	22	46	48	42	68	84	66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
36	26	22	46	48	42	68	84	66

4 希尔排序

第三次，增量为2，记录在变量h中

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
36	26	22	46	48	42	68	84	66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
36	26	22	46	48	42	68	84	66

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
22	26	36	46	48	42	68	84	66

jyang
@cs.ecnu.edu.cn

4 希尔排序

希尔排序：

```
for（delta=0；delta<s；delta++) //最外层循环取每次的增量
{       h=d[delta]; //增量即步长记录在h中
```

插入排序

```
        for(j=h;j<n;j++) {
                t=a[j];
//对于每个j，以h的步长向左，在前若干个结点中寻找a[j]的插入位置。
                for(k=j-h;k>=0&&t<a[k];k=k-h)
                        a[k+h]=a[k];
                a[k+h]=t;
        }
}
```

4 希尔排序

希尔排序是不稳定的。

希尔排序所需的比较次数：取决于“增量”序列。
The sequence suggested by Shell : N/2,
$h_{k+1}=h_k/2$，……，1

Values of h recommended by Knuth in 1969
1 4 13 40 121 364 1093 3280 9841 …
Start with 1, then multiply by 3 and add 1
Less than **$O(n^{3/2})$** comparisons

“增量”的取法：增量序列中的值应该没有除1之外的公因子，并且最后一个增量值必须等于1。

5 合并排序

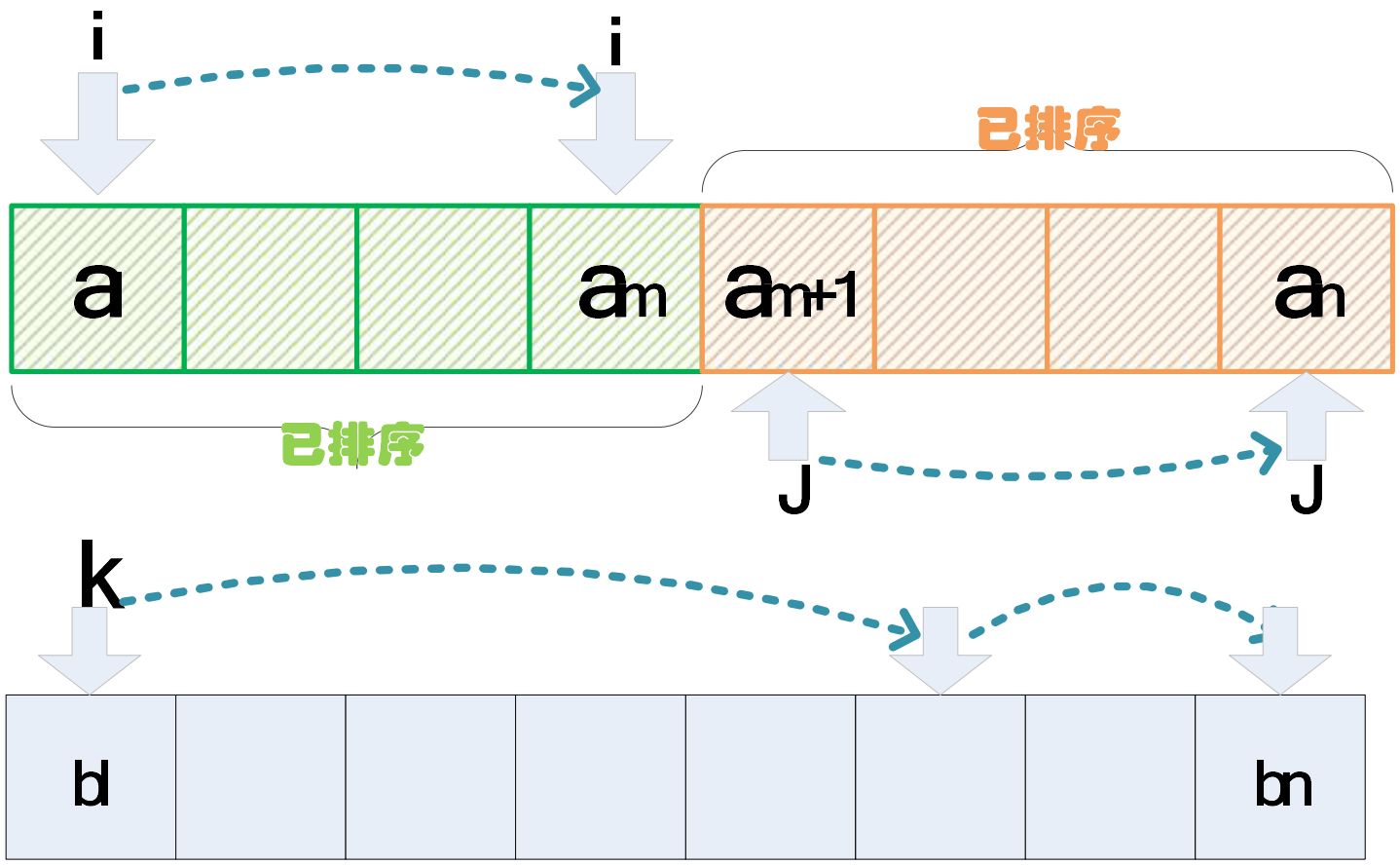
两路合并排序：通过对两个有序结点序列的合并来实现排序。

假设我们所要合并的两个有序结点序列分别存放在

$a_l, a_{l+1}, \cdots, a_m$ **和** $a_{m+1}, a_{m+2}, \cdots, a_n$

把合并后的新序列存放在 $b_l, b_{l+1}, \cdots, b_n$

5 合并排序



依次比较a[i]和a[j]，将其中小的一个存入b[k]，并移动处理过的序列的下标.

i或j有一个先到达边界，将另一个序列剩余的结点也依次存入b[k]

5 合并排序

(1) $l \to i, m+1 \to j, l \to k$

(2) 若$i \le m$，且$j \le n$，则

如果$a_i \le a_j$，那么$a_i \to b_k, i+1 \to i, k+1 \to k$，转（2）；

如果$a_i > a_j$，那么$a_j \to b_k, j+1 \to j, k+1 \to k$，转（2）；

(3) 若$i \le m$，则

$a_i \to b_k, i+1 \to i, k+1 \to k$，转（3）；

(4) 若$j \le n$，则

$a_j \to b_k, j+1 \to j, k+1 \to k$，转（4）；

(5) 算法结束

5 合并排序

```
void merge(int a[], int b[], int l, int m, int n)
  //  l:序列1的起始位置，  m:序列1的结束位置，  n：  序列2的结束位置
{    int i,j,k;
     i=l;        //序列1起始位置
     j=m+1; //序列2起始位置
     k=l;     //用于存放结果的b数组的开
     while(i<=m&&j<=n)
            if(a[i]<=a[j])
                   b[k++]=a[i++];
            else
                   b[k++]=a[j++];
      while(i<=m)
            b[k++]=a[i++];
      while(j<=n)
            b[k++]=a[j++];
}
```

执行时间分析：
前三个赋值语句的执行时间为O（1）

第一个循环：最多执行$(n-l+1)$次。
所以执行时间为O$(n-l+1)$

第二个循环：最多执行$(m-l+1)$次，故其执行时间为O$(m-l+1)$。

第三个循环：最多执行$(n-m)$次，故其执行时间为O$(n-m)$。

5 合并排序

```
void merge1(int a[],  int low, int mid, int hi)
  //  利用辅助数组将两个已经排好序的子序列归并，a[low..mid]  a[mid+1..hi]
{
        int b[MAXN];   // 定义一个辅助数组b
        int i=low;      //序列1开始位置
        int j = mid+1 ;      //序列2开始位置
        int k;
        for (k=low; k<=hi; k++)  b[k]=a[k];  //序列复制到辅助数组
        k=low;
        while(i<=mid&&j<=hi)
                if(b[i]<=b[j])
                        a[k++]=b[i++];
                else
                        a[k++]=b[j++];

        while(i<=m)                a[k++]=b[i++];
        while(j<=n)                a[k++]=b[j++];
}
```

5 合并排序

执行时间分析：
前三个赋值语句的执行时间为O（1）

第一个循环：最多执行$(n-l+1)$次。所以执行时间为$O(n-l+1)$

第二个循环：最多执行$(m-l+1)$次，故其执行时间为$O(m-l+1)$。

第三个循环：最多执行$(n-m)$次，故其执行时间为$O(n-m)$。

所以，整个函数的执行时间为O(n-1)

5 合并排序

两路合并排序：

利用有序结点序列的合并来实现的排序，称为合并排序。

可分为 自顶向下 和 自底向上 两种。

自顶向下：把整个序列分成2部分（通常尽量平分），独立进行合并排序，然后再将两个排好序的序列合并。

5 合并排序

两路合并排序： **自顶向下**

```
void mergeSort(T a[], int left, int right)
{
        if (left==right) return;
        int mid=(left+right)/2;
        mergeSort(a,left,mid);
        mergeSort(a,mid+1,right);
        merge1(a,left,mid,right);
        ……
}
```

5 合并排序

两路合并排序：自底向上

首先把n个结点的序列看成n个长度为1的有序子序列，把这些有序的子序列成对地加以合并，得到$\lfloor \frac{n}{2} \rfloor$个长度为2的有序的子序列（若n为奇数，则有一个子序列没有参加合并，它的长度还是1）。

然后再将这$\lfloor \frac{n}{2} \rfloor$个有序子序列成对地合并。这样的合并过程一直继续。直到最后得到一个长度为n的有序子序列。

5 合并排序

例子：

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
46	26	22	68	48	42	36	84	66

l=1

26	46	22	68	42	48	36	84	66

l=2

22	26	46	68	36	42	48	84	66

l=4

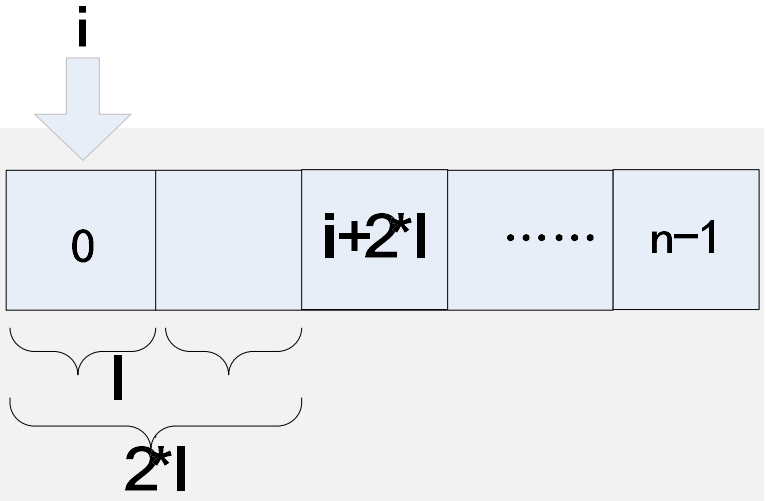
22	26	36	42	46	48	68	84	66

l=8

22	26	36	42	46	48	68	84	66

l=16 循环终止

5 合并排序



两路合并排序：自底向上

```
void mpass(int a[], int b[], int n, int l)
{ //把具有n个结点的序列a所包含的各个
有序子序列（每个长度为l，最后一个可能小于l）
成对合并，合并后的序列放在b
        int i,j;
        i=0;
        while(i+2*l<=n)
                {       merge(a,b,i,i+l-1,i+2*l-1);
                        i+=2*l;
                }
        if(i+l<n)   merge(a,b,i,i+l-1,n-1);
        else        for(j=i;j<n;j++)   b[j]=a[j];
}
```

i:第一段首
i+l-1:第一段尾
i+2*l-1:第二段尾

5 合并排序

两路合并排序：自底向上

```
void merge_sort(int a[], int n) //合并排序
        {
                int b[MAXN];
                int l;
                l=1； //自底向上
                while(l<n) //两个mpass，交替使用a和b数组保存结果
                        {       mpass(a,b,n,l);
                                l*=2;
                                mpass(b,a,n,l);
                                l*=2;
                        }
        }
```

5 合并排序

两路合并排序——执行时间

对具有n个结点的序列进行合并，需要$[\log_2 (n-1)] + 1$**遍合并。**
在每遍合并中，参加合并的结点个数不超过n。因此，每遍合并的执行时间为O（n）。

整个合并排序的执行时间为 O(n$\log_2 n$**)**

合并排序是稳定的。

6 快速排序

思想：分治。把序列分成两部分。C.A.R Hoare于1960年提出。

Charles Antony Richard Hoare (Tony Hoare or C.A.R. Hoare, born January 11, 1934) is a British computer scientist, and winner of the 1980 Turing Award.

He is best known for his fundamental contributions to the definition and design of programming languages (Algol 60), and for the development of Quicksort, the world's most widely used sorting algorithm.

6 快速排序

思想：分治。把序列分成两部分。

取待排序的结点序列中某个结点的值作为控制值，采用某种方法把这个控制值放到适当的位置，使得这个位置的左面的所有结点的值都小于等于这个控制值，右面的所有结点的值都大于这个控制值。接着再将这两部分独立排序。

如何为一个结点找到它合适的位置?

引人注目的特点:
1.只需要很少的辅助空间（原地排序）
2.将长度为N的序列排序所需要的时间和NlgN 成正比。

6 快速排序

$$a_0, a_1, \cdots, a_{n-1}$$

如果把a0作为控制值，那么根据a_0, 把原来的序列排成

$$a_1', a_2', \cdots, a_s', a_0, a_1'', a_2'', \cdots, a_t''$$ s+1+t=n

其中，$a_1', a_2', \ldots, a_s'$ 的值都小于等于a_0.

$a_1'', a_2'', \cdots, a_t''$ 的值都大于a_0 。然后，两个子序列再分别排序。

为方便起见，下面考虑的快速排序都以首结点的值作为控制值。

6 快速排序

快速排序 VS 自上而下的归并排序

6 快速排序

假设目前考虑的结点序列为a[low], a[low+1], ……, a[up]，且low<up。为了给首结点a[low]找到恰当的位置，可采用下面的算法：

使用新数组的方法：申请一个同样大小的数组。顺序扫描原数组，如果比a[low]小，则从新数组的左边开始放；否则，从新数组的右边开始放。最后将a[low]放到新数组唯一的空余空间中。（特点：实现简单，需要额外空间）

只使用一个额外单元的方法：

6 快速排序

假设目前考虑的结点序列为a[low], a[low+1], ……, a[up]，且low<up。为了给首结点a[low]找到恰当的位置，可采用下面的算法：

只使用一个额外单元的方法：首先，将a[low]放在变量t中，这样low的位置就“空” 出来了。接下去重复下列步骤：
（1）从右向左开始检查，如果a[up]>t，则该位置中的元素位置正确，up减一（继续往左）直到遇到一个小于等于t的元素；
（2）将这个元素放入low的位置（此时，up的位置又“空”出）。然后从low+1的位置开始从左向右检查，直到遇到一个大于t的元素；
（3）将这个大于t的元素放入up位置。重复（1），直到low和up重叠。将t放入此位置。

6 快速排序

注意：

别越界

如何终止循环

如何终止递归

6 快速排序

只使用一个额外的单元

（1） $low \to i, up \to j, a[low] \to t$ ，

（2）如果 $i \neq j$ ，则

（2.1）若 $a[j] > t$ ，那么 $j-1 \to j$ ，转（2.1）

否则， $a[j] \to a[i], i+1 \to i$ ，转（2.2）

（2.2）若 $a[i] \leq t$ ，那么 $i+1 \to i$ ，转（2.2）

否则， $a[i] \to a[j]$ ， $j-1 \to j$ ，转（2）

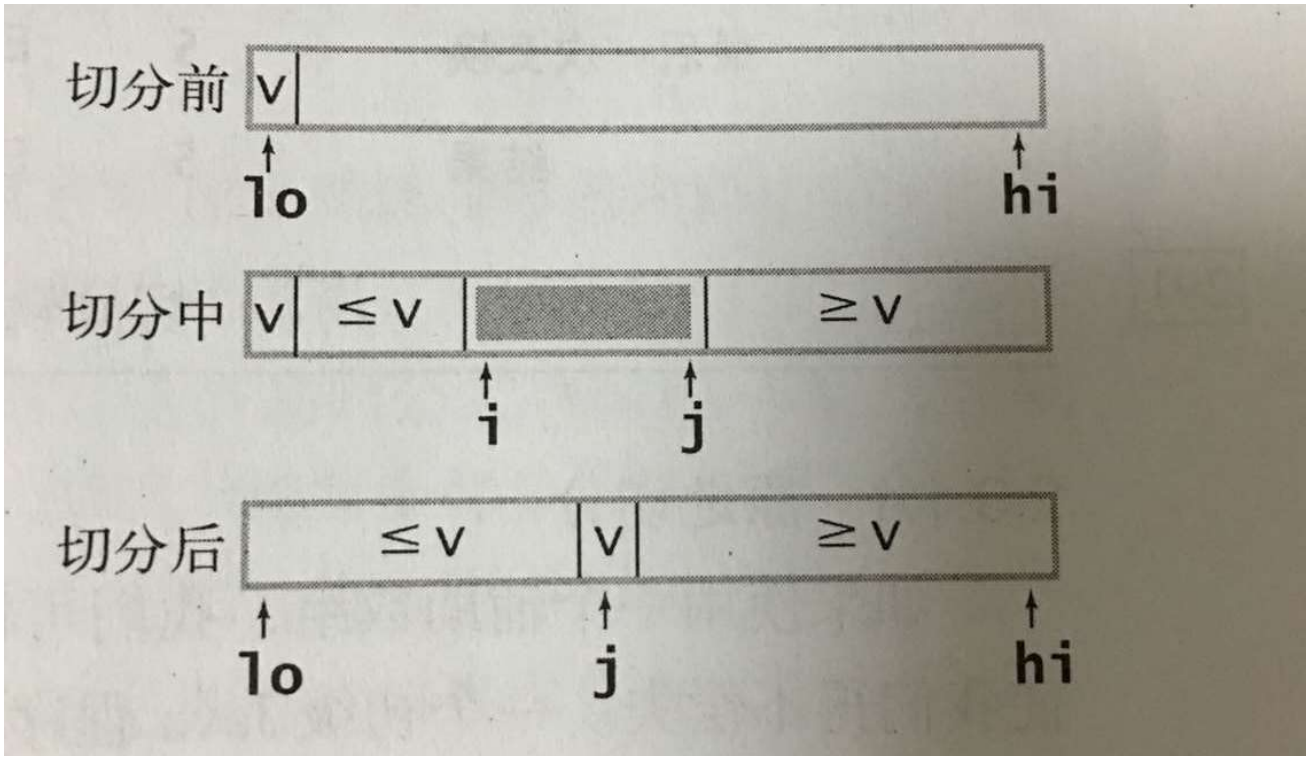
（3）如果 $i = j$ ，则 $t \to a[i]$ ，算法结束

6 快速排序

```
void quick_sort(int a[], int low, int up) //快速排序
{
        int i,j,t;
        if(low<up)                                          // 递归程序，low>=up时停止
        {       i=low;  j =up;  t=a[low];                   //t取首结点作为控制值
                while(i !=j)                                //当i=j完成一趟划分
                {
                        while(i<j&&a[j]>t)  j--; //从后往前找到一个不大于控制
值的结点，即小于等于
                        if(i<j) a[i++]=a[j];                //换到a[i],i后移
                        while(i<j&&a[i]<=t)  i++;   //找到一个大于t的结点
                        if(i<j) a[j--]=a[i];                //换到a[j]，j前移
                }
                a[i]=t;                                     //将控制值放到其最终位置
                quick_sort( (a,low,i-1);                    //递归排序左侧序列
                quick_sort(a,i+1,up);                       //递归排序右侧序列
                }
}
```

6 快速排序

V最后参与交换，（当两个指针相遇时，将切分元素a[lo]和左子数组最右侧的元素a[j]交换）



6 快速排序

```
void quick_sort1(int a[], int low, int up)
        {       int i,j,t;
                if (up<=low)  return;
                i=low;   j =up+1;  t=a[low];
                while(true)
                {
                        while(a[++i]<t )   if (i==up)  break;
                        while(t<a[--j] )   if (j==low)  break;
                        if (i>=j) break;
                        exch(a,i,j);
                }
                exch(a,low,j)

                quick_sort1(a,low,i-1);
                quick_sort1(a,i+1,up);
                }
        }
```

6 快速排序

排序n个结点的序列所需要的结点平均比较次数为：

O（$n\log_2 n$），可证明不超过：$1.4\log_2 n$

最坏情况：对于几乎是排好序的结点序列，快排的效率最差，执行时间为O（n^2）

快速排序是不稳定的。

改进：取样切分（例如，三取样切分，取三个元素，用大小居中的元素切分）。
当数组比较小时，不用快排而使用其他简单排序（比如用插入排序）。
含有大量重复元素的数组：三向切分的快速排序。

7 基数排序

前面所介绍的排序方法都是根据结点的关键字的整个值的大小进行“比较+移动”来实现的。在本节中，我们讨论按组成关键字的各位的值进行排序的方法。

实现基数排序，通常的两种方法：

（1）**最高位优先排序**（Most Significant Digit First，缩写为MSD）

（2）**最低位优先排序**（Least Significant Digit First，缩写为LSD）

7 基数排序

（1）最高位优先排序（MSD）：首先按关键词最高位排序，结果得到若干子序列。接着对每个子序列按关键字次高位排序。重复上述过程。

（2）最低位优先排序（LSD）：首先按关键词最低位的值的大小把结点序列分成若干个子序列，再从小到大依次把各个子序列收集起来，产生一个新序列。对新的结点序列按次最低位，接着对每个推按关键字次高位排序。重复上述过程，最后可以得到排好序的结点序列。

LSD，从头结点到尾结点进行若干次分配和收集。执行的次数取决于构成关键字的位数。
MSD处理各堆与子堆的独立排序问题，可以是一个递归的过程。

7 基数排序

（2）最低位优先排序（LSD）： 首先按关键词最低位的值的大小把结点序列分成若干个子序列，再从小到大依次把各个子序列收集起来，产生一个新序列。对新的结点序列按次最低位的值又分成若干个子序列，然后从小到大收集。重复上述过程，最后可以得到排好序的结点序列。

子序列的个数：r（以r为基的排序）
需要重复的次数：d（关键词有d位）

每个子序列的大小较难估计，经常要进行子序列的合并，所以使用链接存储更为合适。

```
for（i=最低位；i<=最高位；i升高一位）
{  //一次循环中完成一次分配+收集，循环中i值不改变；

//（1）准备若干（r个）空队列，用于分配；
//（2）分配：按当前的i位的值分配到相应的队列；
//（3）收集：把子序列按序连接，生成新的队列；
}
```

7 基数排序

```
NODE * head[MAX_R], *tail[MAX_R];


for（i=最低位；i<=最高位；i升高一位）
{  //一次循环中完成一次分配+收集，循环中i值不改变；

//（1）准备若干（r个）空队列，用于分配；
   for(j=0;j<r;j++）head[j]=NULL;

//（2）分配：按当前的i位的值分配到相应的队列；
//（3）收集：把子序列按序连接，生成新的队列；


}
```

7 基数排序

```
for（i=最低位；i<=最高位；i升高一位)
{  //一次循环中完成一次分配+收集，循环中i值不改变；
// （1）准备若干（r个）空队列，用于分配；
   for(j=0;j<r; j++）head[j]=NULL;

// （2）分配：按当前的i位的值分配到相应的队列；
          while(p!=NULL )
                    {
                    //k=p->data 的第i位，将p指向的结点插入相应的head[k]队尾
                    if (head[k]==NULL) head[k]=p; else head[k]=p;
                    tail[k]=p;
                    //p=p->link;
                    }
// （3）收集：把r个子队列按序连接，生成新的队列；连接方法：把小的队尾指向大
的队首
          p=NULL;
          for (j=r-1;j>=0;j--)
               if (head[j]!=NULL )
               {  tail[j]->link =p; //把j队列的队尾结点的指针指向当前的队首结点，
                   p=head[j]; //当前队首改变为j队列的队首;
                }
```

7 基数排序

对线性链表的所有n个结点进行进行d遍分配和收集，每遍运行的时间为O(n+r)。

因此，总的运行时间为O(d（n+r）)

基数排序是稳定的。

7 基数排序

多关键字排序

可以使用上述的基数排序的方法来解决。

例如：扑克牌的排序。两个关键词：花色和面值。
定义：花色优先级高
花色：梅花<方块<红心<黑桃
面值：2<3.....<A

MSD：按花色分4堆，每堆按面值排序。
LSD: 按面值分成13堆，从小到大放；再按花色分四堆，从小到大收集。

各种排序方法的比较

排序算法	平均时间复杂度	最坏时间复杂度	最好时间复杂度	空间复杂度	稳定性	算法实现
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	O（1）	稳定	前面的序列已排序，后面的结点插入到正确的位置。序列向后逐渐扩大。
选择排序	$O(n^2)$	$O(n^2)$	$O(n)$	O（1）	不稳定	选当前序列中最小的结点，和序列中的第一个交换。序列向后逐渐缩小。
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	O（1）	稳定	两两比较，位置不正确则交换。直到某次比较不发生交换
希尔排序	取决于增量序列	取决于增量序列	$O(n)$	O（1）	不稳定	插入排序的扩展。
合并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定	
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定	
基数排序	O（d*n）	O（d*n）	O（d*n）	O（r*d）	稳定	