2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——

第四讲排序算法

第一节　排序及其基本概念

一、基本概念

1．什么是排序

排序是数据处理中经常使用的一种重要运算。

设文件由 n 个记录{R_1，R_2，……，R_n}组成，n 个记录对应的关键字集合为{K_1，K_2，……，K_n}。所谓排序就是将这 n 个记录按关键字大小递增或递减重新排列。

2．稳定性

当待排序记录的关键字均不相同时，排序结果是惟一的，否则排序结果不唯一。

如果文件中关键字相同的记录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，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**。

3．排序的方式

由于文件大小不同使排序过程中涉及的存储器不同，可将排序分成内部排序和外部排序两类。整个排序过程都在内存进行的排序，称为内部排序；反之，若排序过程中要进行数据的内、外存交换，则称之为外部排序。

内排序适用于记录个数不是很多的小文件，而外排序则适用于记录个数太多，不能一次性放人内存的大文件。

内排序是排序的基础，本讲主要介绍各种内部排序的方法。

按策略划分内部排序方法可以分为五类：插入排序、选择排序、交换排序、归并排序和分配排序。

二、排序算法分析

1．排序算法的基本操作

几乎所有的排序都有两个基本的操作：

（1）关键字大小的比较。

（2）改变记录的位置。具体处理方式依赖于记录的存储形式，对于顺序型记录，一般移动记录本身，而链式存储的记录则通过改变指向记录的指针实现重定位。

为了简化描述，在下面的讲解中，我们只考虑记录的关键字，则其存储结构也简化为数组或链表。并约定排序结果为递增。

2．排序算法性能评价

排序的算法很多，不同的算法有不同的优缺点，没有哪种算法在任何情况下都是最好的。评价一种排序算法好坏的标准主要有两条：

（1）执行时间和所需的辅助空间，即时间复杂度和空间复杂度；

（2）算法本身的复杂程度，比如算法是否易读、是否易于实现。

第二节　插入排序

插入排序的基本思想是：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的记录集中，使记录依然有序，直到所有待排序记录全部插入完成。

一、直接插入排序

1．直接插入排序的思想

假设待排序数据存放在数组 A[1..n]中，则 A[1]可看作是一个有序序列，让 i 从 2 开始，依次将 A[i]插入到有序序列 A[1..i-1]中，A[n]插入完毕则整个过程结束，A[1..n]成为有序序列。

2．排序过程示例　（用【 】表示有序序列）

待排序数据：　【25】　54　8　54　21　1　97　2　73　15　　（n=10）

i=2：　【25　**54**】　8　54　21　1　97　2　73　15

i=3：　【**8**　25　54】　54　21　1　97　2　73　15

i=4：　【8　25　54　**54**】　21　1　97　2　73　15

i=5：　　【8　21　25　54　54】　1　97　2　73　15

i=6：　　【1　8　21　25　54　54】　97　2　73　15

i=7：　　【1　8　21　25　54　54　97】　2　73　15

i=8：　　【1　2　8　21　25　54　54　97】　73　15

i=9：　　【1　2　8　21　25　54　54　73　97】　15

i=10：　　【1　2　8　15　21　25　54　54　73　97】　　排序结束

3．算法实现

可在数组中增加元素 A[0]作为关键值存储器和循环控制开关。第 i 趟排序，即 A[i]的插入过程为：

① 保存 A[i]→A[0]

② $j=i-1$

③ 如果 A[j]<=A[0]（即待排序的 A[i]），则 A[0]→A[j+1]，完成插入；

否则，将 A[j]后移一个位置：A[j]→A[j+1]；$j=j-1$；继续执行③

对于上面的数据实例，i 从 2 依次变化到 10 的过程中，j 值分别为{1，0，3，1，0，6，1，7，3}

4．程序代码

```
procedure insertsort(n:integer);
  var i,j:integer;
  begin
      for i:=2 to n do
          begin
              a[0]:=a[i];
              j:=i-1;
              while a[j]>a[0] do                {决定运算次数和移动次数}
                  begin
                      a[j+1]:=a[j];
                      j:=j-1;
                  end;
              a[j+1]:=a[0];
          end;
  end;
```

5．算法分析

（1）稳定性：稳定

（2）时间复杂度：

①原始数据正序，总比较次数：n-1

②原始数据逆序，总比较次数：$\sum_{i=2}^{n} i=\frac{n^2+n-2}{2}=O(n^2)$

③原始数据无序，第 i 趟平均比较次数=$\sum_{j=1}^{i} j/i=\frac{i+1}{2}$，总次数为：$\frac{n^2+3n}{4}=O(n^2)$

④可见，原始数据越趋向正序，比较次数和移动次数越少。

（3）空间复杂度：仅需一个单元 A[O]

二、希尔排序

1．基本思想：

任取一个小于 n 的整数 S_1 作为增量，把所有元素分成 S_1 个组。所有间距为 S_1 的元素放在同一个组中。

第一组：{A[1]，A[S_1+1]，A[2*S_1+1]，……}

第二组：{A[2]，A[S_1+2]，A[2*S_1+2]，……}

第三组：{A[3]，A[S_1+3]，A[2*S_1+3]，……}

……

第 s_1 组：{A[S_1]，A[2*S_1]，A[3*S_1]，……}

先在各组内进行直接插人排序；然后，取第二个增量 S_2（<S_1）重复上述的分组和排序，直至所取的增量 S_t=1（S_t<S_{t-1}<S_{t-2}<…<S_2<S_1），即所有记录放在同一组中进行直接插入排序为止。

2．排序过程示例

待排序数据：

序号		1	2	3	4	5	6	7	8	9	10
原始数据		12	89	57	32	96	37	54	5	79	57
S_1=5	组别	①	②	③	④	⑤	①	②	③	④	⑤
	排序结果	12	54	5	32	57	37	89	57	79	96
S_2=3	组别	①	②	③	①	②	③	①	②	③	①
	排序结果	12	54	5	32	57	37	89	57	79	96
S_3=2	组别	①	②	①	②	①	②	①	②	①	②
	排序结果	5	32	12	37	57	54	79	57	89	96
S_4=1	组别	①	①	①	①	①	①	①	①	①	①
	排序结果	5	12	32	37	54	57	57	79	89	96

3．算法实现

由于在分组内部使用的是直接插入排序，因此排序过程只要在直接插入排序的算法上对 j 的步长进行修改就可以了。

4．程序代码

```
procedure shell(n:integer);
  var s,k,i,j:integer;
  begin
      s:=n;
      repeat
          s:=round(s/2);                    {设置增量 S 递减}
          for k:=1 to s do                  {对 S 组数据分别排序}
            begin
              i:=k+s;                       {第二个待排序数据}
              while i<=n do                 {当 i>n 时，本组数据排序结束}
                begin
                  a[0]:=a[i];
```

```
                    j:=i-s;                    {约定步长为 S}
                    while (a[j]>a[0]) and (j>=k) do
                        begin
                            a[j+s]:=a[j];
                            j:=j-s;
                        end;
                    a[j+s]:=a[0];
                    i:=i+s;
                end;
            end;
        until s=1;
    end;
```

5．算法分析

（1）稳定性：不稳定

（2）时间复杂度：

①Shell 排序的执行时间依赖于增量序列。如实例所示，选择增量系列为 5-2-1 的比较次数<增量系列为 5-3-2-1 的比较次数。

②在直接插入排序中，数据越趋向于有序，比较和移动次数越少，Shell 排序的目的则是增加这种有序趋势。虽然看起来重复次数较多，需要多次选择增量，但开始时，增量较大，分组较多，但由于各组的数据个数少，则比较次数累计值也小，当增量趋向 1 时，组内数据增多，而所有数据已经基本接近有序状态，因此，Shell 的时间性能优于直接插入排序。

（3）空间复杂度：仅需一个单元 A[O]

第三节　交换排序

交换排序的基本思想是：两两比较待排序的数据，发现两个数据的次序相反则进行交换，直到没有反序的数据为止。

一、冒泡排序

1．冒泡排序的思想

最多进行 n-1 趟排序，每趟排序时，从底部向上扫描，一旦发现两个相邻的元素不符合规则，则交换。我们发现，第一趟排序后，最小值在 A[1]，第二趟排序后，较小值在 A[2]，第 n-1 趟排序完成后，所有元素完全按顺序排列。

2．排序过程示例

待排序数据：53　33　19　53　3　63　82　20　19　39

第一趟排序：3　53　33　19　53　19　63　82　20　39

第二趟排序：3　19　53　33　19　53　20　63　82　39

第三趟排序：3　19　19　53　33　20　53　39　63　82

第四趟排序：3　19　19　20　53　33　39　53　63　82

第五趟排序：没有反序数据，排序结束。

3．程序代码

```
procedure Bubble(n:integer);;
    var temp,i,j:integer;
```

```
        change:boolean;
    begin
        for i:=1 to n-1 do
            begin
                change:=false;
                for j:=n-1 downto i do
                    if a[j]>a[j+1] then
                        begin
                            change:=true;
                            temp:=a[j]; a[j]:=a[j+1]; a[j+1]:=temp;
                        end;
                if not(change) then exit;
            end;
    end;
```

4．算法分析

（1）稳定性：稳定

（2）时间复杂度：

①原始数据正序，需一趟排序，比较次数 n-1=O(n)

②原始数据反序，需 n-1 趟排序，比较次数 $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}=O(n^2)$

③一般情况下，虽然不一定需要 n-1 趟排序，但由于每次数据位置的改变需要 3 次移动操作，因此，总时间复杂度高于直接插入排序。

（3）空间复杂度：仅需一个中间变量

5．改进

可以在每趟排序时，记住最后一次发生交换发生的位置，则该位置之前的记录均已有序。下一趟排序扫描到该位置结束。利用这种方式，可以减少一部分比较次数。

二、快速排序

1．快速排序的思想

在 A[1..n]中任取一个数据元素作为比较的“基准”（不妨记为 X），将数据区划分为左右两个部分：A[1..i-1]和 A[i+1..n]，且 A[1..i-1]≤X≤A[i+1..n](1≤i≤n)，当 A[1..i-1]和 A[i+1..n]非空时，分别对它们进行上述的划分过程，直至所有数据元素均已排序为止。

2．算法实现

可以使用递归函数实现这一算法。假定待排序序列的下标范围为 low~high。

借用两个整型变量 i、j 作为指针，约定初值分别为 low、high。

排序过程：

① 选定基准 X（不妨用 A[low]）

② j 向前扫描，直到 A[j]<X，交换 A[i]与 A[j]，i+1。保证了 A[low..i-1]≤X

③ i 向后扫描，直到 A[i]>X，交换 A[i]与 A[j]，j-1。保证了 X≤A[j..high]

④ 继续执行②、③，直到 i=j。这时，X 恰好在 A[i]位置上。

⑤ 对序列 A[low..i-1]及 A[i+1..high]按照上述规律继续划分，直到序列为空。

仔细分析算法，我们发现，在排序中，我们总是用基准 X 与另一数据交换，因此，一趟排序结束后，X 就能确

切定位其最终位置。

3．排序过程示例

待排序数据：　67　67　14　52　29　9　90　54　87　71

X=67　　i　　j

扫描 j　　i　　j

交换　　54　67　14　52　29　9　90　67　87　71

扫描 i　　i　j

交换　　54　67　14　52　29　9　67　90　87　71

j=i，结束　　ij

第一趟排序后：54　67　14　52　29　9　[67]　90　87　71

第二趟排序后：　9　29　14　52　[54]　67 [67]　71　87　[90]

第三趟排序后：[9]　29　14　52　[54　67　67　71] 87 [90]

第四趟排序后：[9]　14 [29] 52　[54　67　67　71　87　90]

第五趟排序后：[9　14　29　52　54　67　67　71　87　90]

4．程序代码

```
procedure quicksort(low,high:integer);
  var temp,x,i,j:integer;
  begin
    if low<high then
      begin
        x:=a[low]; i:=low; j:=high;
        while i<j do
          begin
            while (j>i) and (a[j]>=x) do j:=j-1; {j 左移}
            temp:=a[i]; a[i]:=a[j]; a[j]:=temp;
            i:=i-1;
            while (j>i) and (a[i]<=x) do i:=i+1;   {i 右移}
            temp:=a[i]; a[i]:=a[j]; a[j]:=temp;
            j:=j-1;
          end;
        quicksort(low,i-1);
        quicksort(i+1,high);
      end;
  end;
```

5．算法分析

（1）稳定性：不稳定

（2）时间复杂度：

每趟排序所需的比较次数为待排序区间的长度-1，排序趟数越多，占用时间越多。

①最坏情况：

每次划分选取的基准恰好都是当前序列中的最小(或最大)值，划分的结果 A[low..i-1]为空区间或 A[i+1..high]是空区间，且非空区间长度达到最大值。

这种情况下，必须进行 n-1 趟快速排序，第 i 次趟去见长度为 n-i+1，总的比较次数达到最大值：$n(n-1)/2=O(n^2)$

②最好情况：

每次划分所取的基准都是当前序列中的“中值”，划分后的两个新区间长度大致相等。共需 lgn 趟快速排序，总的关键字比较次数：O(nlgn)

③基准的选择决定了算法性能。经常采用选取 low 和 high 之间一个随机位置作为基准的方式改善性能。

（3）空间复杂度：快速排序在系统内部需要一个栈来实现递归，最坏情况下为 O(n)，最佳情况下为 O(lgn)。

第四节　选择排序

选择排序的基本思想是：每一趟从待排序的数据中选出最小元素，顺序放在已排好序的数据最后，直到全部数据排序完毕。

一、直接选择排序

1．过程模拟

待排序数据	92	28	62	84	62	16	56	87	33	66
第一趟排序	16	28	62	84	62	92	56	87	33	66
第二趟排序	16	28	62	84	62	92	56	87	33	66
第三趟排序	16	28	33	84	62	92	56	87	62	66
第四趟排序	16	28	33	56	62	92	84	87	62	66
第五趟排序	16	28	33	56	62	92	84	87	62	66
第六趟排序	16	28	33	56	62	62	84	87	92	66
第七趟排序	16	28	33	56	62	62	66	87	92	84
第八趟排序	16	28	33	56	62	62	66	84	92	87
第九趟排序	16	28	33	56	62	62	66	84	87	92

2．程序代码

```
procedure select(n:integer);
  var temp,k,i,j:integer;
  begin
    for i:=1 to n-1 do
        begin
            k:=i;
            for j:=i+1 to n do
                if a[j]<a[k] then k:=j;
                if k<>i then
                    begin
                      a[0]:=a[i]; a[i]:=a[k]; a[k]:=a[0];
                    end;
        end;
  end;
```

3．算法分析

（1）稳定性：不稳定

（2）时间复杂度：$O(n^2)$

（3）空间复杂度：仅需一个中间单元 A[0]

二、堆排序

1．堆排序思想

堆排序是一种树形选择排序，在排序过程中，将 A[1..n]看成是完全二叉树的顺序存储结构，利用完全二叉树中双亲结点和孩子结点之间的内在关系来选择最小的元素。

2．堆的定义：n 个元素的序列 K_1，K_2，K_3，…K_n 称为堆，当且仅当该序列满足特性：$K_i \le K_{2i}$ ， $K_i \le K_{2i+1}$（$1 \le i \le n/2$）

堆实质上是满足如下性质的完全二叉树：树中任一非叶子结点的关键字均大于等于其孩子结点的关键字。例如序列{1,35,14,60,61,45,15,81}就是一个堆，它对应的完全二叉树如下图 1 所示。这种堆中根结点（称为堆顶）的关键字最小，我们把它称为小根堆。反之，若完全二叉树中任一非叶子结点的关键字均大于等于其孩子的关键字，则称之为大根堆。

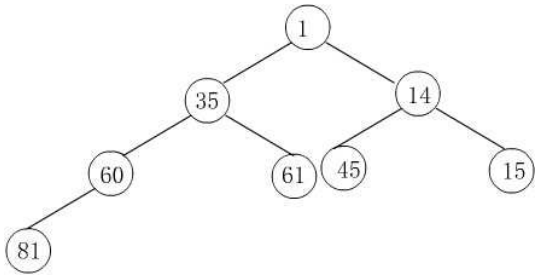

图 4_1 最小堆

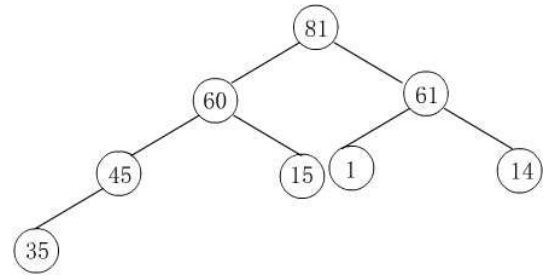

图 4_2 最大堆

存储结构：

1	35	14	60	61	45	15	81

81	60	61	45	15	1	14	35

3．堆排序过程（以最大堆为例）

（1）调整堆

假定待排序数组 A 为{20，12，35，15，10，80，30，17，2，1}（n=10），初始完全树状态为：

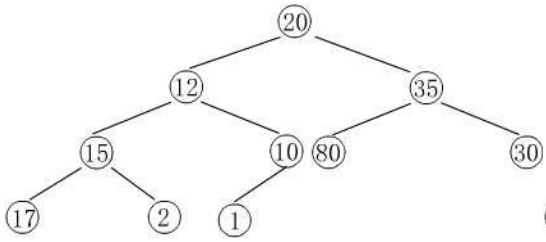

图 4_3

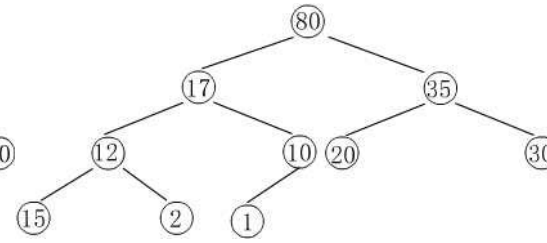

图 4_4

结点（20）、（35）、(12)等，值小于其孩子结点，因此，该树不属最大堆。

为了将该树转化为最大堆，从后往前查找，自第一个具有孩子的结点开始，根据完全二叉树性质，这个元素在数组中的位置为 i=[n/2]，如果以这个结点为根的子树已是最大堆，则此时不需调整，否则必须调整子树使之成为堆。随后，继续检查以 i-1、i-2 等结点为根的子树，直到检查到整个二叉树的根结点（i=1），并将其调整为堆为止。

调整方法：由于 A[i]的左、右子树均已是堆，因此 A[2i]和 A[2i+1]分别是各自子树中关键字最大的结点。若 A[i]不小于 A[2i]和 A[2i+1]，则 A[i]没有违反堆性质，那么以 A[i]为根的子树已是堆，无须调整；否则必须将 A[i]和 A[2i]与 A[2i+1]中较大者（不妨设为 A[j]）进行交换。交换后又可能使结点 A[j]违反堆性质，同样由于该结点的两棵子树仍然是堆，故可重复上述的调整过程，直到当前被调整的结点已满足堆性质，或者该结点已是叶子结点为止。

以上图为例，经过调整后，最大堆为：{80，17，35，12，10，20，30，15，2，1}。如图 4_4 所示。此堆作为排序的初始无序区。

（2）选择、交换、调整

①将建成的最大堆作为初始无序区。

②将堆顶元素（根）A[1]和 A[n]交换，由此得到新的无序区 A[1..n-1]和有序区 A[n]，且满足 A[1..n-1]≤A[n]

③将 A[1..n-1]调整为堆。

④再次将 A[1]和无序区最后一个数据 A[n-1]交换，由此得到新的无序区 A[1..n-2]和有序区 A[n-1..n]，且仍满足关系 A[1..n-2]≤A[n-1..n]，同样要将 A[1..n-2]调整为堆。直到无序区只有一个元素 A[1]为止。

说明：如果需要生成降序序列，则利用最小堆进行操作。

4．程序代码

```
procedure editheap(i:integer; s:integer);                    {堆的调整}
  var j:integer;
  begin
      if 2*i<=s then                    {如果该结点为叶子，则不再继续调整}
        begin
            a[0]:=a[i]; j:=i;
            if a[2*i]>a[0] then begin a[0]:=a[2*i]; j:=2*i; end;
            if (2*i+1<=s) and (a[2*i+1]>a[0]) then
              begin a[0]:=a[2*i+1]; j:=2*i+1; end; {获取最大值}
            if j<>i then
              begin
                a[j]:=a[i]; a[i]:=a[0]; {更新子树结点}
                editheap(j,s);          {调整左右孩子}
              end;
        end;
  end;
procedure buildheap;          {堆的建立}
  var i,j:integer;
  begin
      for i:=n div 2 downto 1 do          {从后往前，依次调整}
        begin
            a[0]:=a[i]; j:=i;
            if a[2*i]>a[0] then begin a[0]:=a[2*i]; j:=2*i; end;
            if (2*i+1<=n) and (a[2*i+1]>a[0]) then
              begin a[0]:=a[2*i+1]; j:=2*i+1; end;
            if j<>i then
              begin
                a[j]:=a[i]; a[i]:=a[0];
                editheap(j,n);
              end;
        end;
  end;
procedure heapsort(n:integer);          {堆排序}
  var k,i,j:integer;
  begin
```

```
    buildheap;
    for i:=n downto 2 do
        begin
            a[0]:=a[i];   a[i]:=a[1]; a[1]:=a[0];
            editheap(1,i-1);
        end;
end;
```

5．算法分析

（1）稳定性：不稳定。

假定数据序列为{1，1}，已是大堆，经过排序后，结果为{1，1}。

（2）时间复杂度：堆排序的时间，主要由建立初始堆和反复调整堆这两部分的时间开销构成，它们均是通过调用 editheap 函数实现的。

堆排序的最坏时间复杂度为 O(nlgn)。堆排序的平均性能较接近于最坏性能。由于建初始堆所需的比较次数较多，所以堆排序不适宜于记录数较少的文件。

（3）空间复杂度：O(1)

第五节　归并排序

归并排序是利用"归并"技术来进行排序。归并是指将若干个已排序的子文件合并成一个有序的文件。

一、两路归并算法

1、算法基本思路

设有两个有序（升序）序列存储在同一数组中相邻的位置上，不妨设为 A[l..m]，A[m+1..h]，将它们归并为一个有序数列，并存储在 A[l..h]。

为了减少数据移动次数，不妨采用一个临时工作数组 C，将中间排序结果暂时保存在 C 数组中，等归并结束后，再将 C 数组值复制给 A。

归并过程中，设置 p1，p2 和 p3 三个指针，其初值分别指向三个有序区的起始位置。归并时依次比较 A[p1]和 A[p2]的关键字，取关键字较小的记录复制到 C[p3]中，然后将被复制记录的指针 p1 或 p2 加 1，以及指向复制位置的指针 p3 加 1。

重复这一过程直至有一个已复制完毕，此时将另一序列中剩余数据依次复制到 C 中即可。

2．程序代码

```
procedure merge(l,m,h:integer);
  var p1,p2,p3,j:integer;
  begin
      if m<n then
          begin
              if h>n then h:=n;
              p1:=l; p2:=m+1;
              p3:=l;
              while (p1<=m) and (p2<=h) do
                  begin
                      if a[p1]<=a[p2] then
                          begin
```

```
                    c[p3]:=a[p1]; p1:=p1+1;
                end
            else begin
                    c[p3]:=a[p2]; p2:=p2+1;
                end;
            p3:=p3+1;
        end;
    if p1>m then
        for j:=p2 to h do begin c[p3]:=a[j]; p3:=p3+1; end;
    if p2>h then
        for j:=p1 to m do begin c[p3]:=a[j]; p3:=p3+1; end;
    for j:=l to h do a[j]:=c[j];
  end;
end;
```

二、归并排序

归并排序有两种实现方法：自底向上和自顶向下。

1．自底向上的方法

（1）自底向上的基本思想

自底向上的基本思想是：第 1 趟归并排序时，将数列 A[1..n]看作是 n 个长度为 1 的有序序列，将这些序列两两归并，若 n 为偶数，则得到[n/2]个长度为 2 的有序序列；若 n 为奇数，则最后一个子序列不参与归并。第 2 趟归并则是将第 1 趟归并所得到的有序序列两两归并。如此反复，直到最后得到一个长度为 n 的有序文件为止。

上述的每次归并操作，均是将两个有序序列合并成一个有序序列，故称其为“二路归并排序”。类似地有 k(k>2)路归并排序。

（2）程序代码

```
procedure mergesort;
    var i,s,k:integer;
    begin
        s:=1;
        while s<n do
            begin
                i:=1;
                repeat
                    merge(s*(i-1)+1,s*i,s*(i+1));
                    i:=i+2;
                until i>n;
                s:=s*2;
            end;
    end;
```

2．自顶向下的方法

采用分治法进行自顶向下的算法设计，形式更为简洁。

（1）分治法的三个步骤

设归并排序的当前区间是 A[l..h]，分治法的三个步骤是：

分解：将当前区间一分为二，即求分裂点 m=(l+h)/2

求解：递归地对两个子区间 A[l..m]和 A[m+1..h]进行归并排序；

组合：将已排序的两个子区间 A[l..m]和 A[m+1..h]归并为一个有序的区间 A[l..h]。

递归的终结条件：子区间长度为 1（一个数据组成的区间必然有序）。

（2）程序代码

```
procedure mergesort(l,h:integer);
    var m:integer;
    begin
        if h>l then
            begin
                m:=(h+l) div 2;
                mergesort(l,m); mergesort(m+1,h);
                merge(l,m,h);
            end;
    end;
```

3．算法分析

（1）稳定性

归并排序是一种稳定的排序。

（2）存储结构要求

用顺序存储结构。也易于在链表上实现。

（3）时间复杂度

长度为 n 的数列，需进行$[\log_2 n]$趟二路归并，每趟归并的时间为 O(n)，故其时间复杂度无论是在最好情况下还是在最坏情况下均是 $O(n\log_2 n)$。

4、空间复杂度

需要一个辅助数组来暂存两有序序列归并的结果，故其辅助空间复杂度为 O(n)。

作业：从以下角度对排序方法进行比较

一、影响排序效果的因素

二、不同条件下，排序方法的选择

三、排序与查找的关系