第10章 排序（参考答案）

一、选择题

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.D | 2.D | 3.D | 4.B | 5.B | 6.B | 7.C,E | 8.A | 9.C | 10.C,D,F | | 11.1D,C 11.2A,D,F | | |
| 11.3B 11.4(A,C,F)(B,D,E) | | | | 12.C,D | 13.A | 14.B,D | 15.D | 16.D | 17.C | 18.A | 19.A | 20.C | 21.C |
| 22.B | 23.C | 24.C | 25.A | 26.C | 27.D | 28.C | 29.B | 30.C,B | 31.D | 32.D | 33.A | 34.D | 35.A |
| 36.A | 37.A | 38.C | 39.B | 40.C | 41.C | 42.B | 43.A | 44.B | 45.A | 46.C | 47.B,D | 48.D | 49.D |
| 50.D | 51.C | 52.E,G | 53.B | 54.C | 55.C | 56.B | 57.B | 58.A | 59.1C 59.2A 59.3D 59.4B 59.5G | | | | |
| 60.1B 60.2C 60.3A | | | 61.1B 61.2D 61.3B 61.4C 61.5F | | | | | 62.A | 63.A | 64.B | 65.A | 66.A |  |

部分答案解释如下：

18. 对于后三种排序方法两趟排序后，序列的首部或尾部的两个元素应是有序的两个极值，而给定的序列并不满足。

20. 本题为步长为3的一趟希尔排序。 24.枢轴是73。

49. 小根堆中，关键字最大的记录只能在叶结点上，故不可能在小于等于n/2的结点上。

64. 因组与组之间已有序，故将n/k个组分别排序即可，基于比较的排序方法每组的时间下界为O(klog2k),全部时间下界为O(nlog2k)。

二、判断题

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1.√ | 2.× | 3.× | 4.× | 5.× | 6.× | 7.× | 8.× | 9.× | 10.× | 11.× | 12.× | 13.× |
| 14.√ | 15.√ | 16.× | 17.× | 18.× | 19.× | 20.× | 21.× | 22.× | 23.× | 24.× | 25.√ | 26.× |
| 27.√ | 28.× | 29.× | 30.× | 31.√ |  |  |  |  |  |  |  |  |

部分答案解释如下：

5. 错误。例如冒泡排序是稳定排序，将4,3,2,1按冒泡排序排成升序序列，第一趟变成3,2,1,4，此时3就朝向最终位置的相反方向移动。 12. 错误。堆是n个元素的序列，可以看作是完全二叉树，但相对于根并无左小右大的要求，故其既不是二叉排序树，更不会是平衡二叉树。

22. 错误。待排序序列为正序时，简单插入排序比归并排序快。

三、填空题

1. 比较,移动 2.生成有序归并段（顺串）,归并 3.希尔排序、简单选择排序、快速排序、堆排序等

4. 冒泡,快速 5. (1)简单选择排序 (2)直接插入排序（最小的元素在最后时）

6. 免去查找过程中每一步都要检测整个表是否查找完毕，提高了查找效率。 7. n(n-1)/2

8.题中p指向无序区第一个记录，q指向最小值结点，一趟排序结束，p和q所指结点值交换，同时向后移p指针。(1)!=null (2)p->next (3)r!=null (4)r->data<q->data (5)r->next (6)p->next

9. 题中为操作方便，先增加头结点（最后删除），p指向无序区的前一记录，r指向最小值结点的前驱，一趟排序结束，无序区第一个记录与r所指结点的后继交换指针。

(1)q->link!=NULL (2)r!=p (3)p->link (4)p->link=s (5)p=p->link

10.(1)i<n-i+1 (2)j<=n-i+1 (3)r[j].key<r[min].key (4)min!=i (5)max==i (6)r[max]<-->r[n-i+1]

11.(1)N (2)0 (3)N-1 (4)1 (5)R[P].KEY<R[I].KEY (6)R[P].LINK (7)(N+2)(N-1)/2

(8)N-1 (9)0 (10)O(1)(每个记录增加一个字段) (11)稳定 （请注意I的步长为-1）

12. 3,(10,7,-9,0,47,23,1,8,98,36) 13.快速 14.(4,1,3,2,6,5,7)

15.最好每次划分能得到两个长度相等的子文件。设文件长度n=2k-1，第一遍划分得到两个长度⎣n/2⎦的子文件，第二遍划分得到4个长度⎣n/4⎦的子文件，以此类推，总共进行k=log2(n+1)遍划分，各子文件长度均为1，排序结束。

16.O(n2) 17. O(nlog2n) 18.(1)2\*i (2)r[j].key>r[j+1].key (3)true (4)r[j] (5)2\*i

19.(1)2\*i (2)j<=r (3)j←j+1 (4)x.key>heap[j].key (5)i←j (6)j←2\*i (7)x

20.(1)j:=2\*i (2)finished:=false (3)(r[j].key>r[j+1].key) (4)r[i]:=r[j] (5)i:=j

(6) j:=2\*i (7)r[i]:=t; (8)s**if**t(r,i,n) (9)r[1]:=r[i] (10)s**if**t(r,1,i-1)

21. ④是堆 (1)选择 (2)筛选法 (3)O(nlog2n) (4)O(1)

22. (1) 选择 (2)完全二叉树 (3)O(Nlog2N) (4)O(1) (5)满足堆的性质

23.(1)finish:=false (2)h[i]:=h[j]; i:=j; j:=2\*j; (3)h[i]:=x (4)h,k,n (5)s**if**t(h,1,r-1)

24. {D,Q,F,X,A,P,B,N,M,Y,C,W}

25. (1)p[k]:=j (2)i:=i+1 (3)k=0 (4)m:=n (5)m<n (6)a[i]:=a[m] (7)a[m]:=t

26. 程序(a)(1)true (2)a[i]:=t (3)2 TO n step 2 (4)true (5)NOT flag

程序(b)(1)1 (2)a[i]=t (3)(i=2;i<=n;i+=2) (4)1 (5)flag

27.（Q,A,C,S,Q,D,F,X,R,H,M,Y）,(F,H,C,D,Q,A,M,Q,R,S,Y,X) 28. 初始归并段(顺串)

29. 初始归并段,初始归并段,减少外存信息读写次数（即减少归并趟数）,增加归并路数和减少初始归并段个数。 30. ⎡n/m⎤

31．(1)m,j-1 (2) m:=j+1 (3)j+1,n (4) n:=j-1 最大栈空间用量为O(logn)。

四、应用题

1. 假设含n个记录的序列为{ R1, R2, …，Rn }，其相应的关键字序列为{ K1, K2, …，Kn }，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Ks1≤Ks2≤…≤Ksn，按此固有关系将n个记录序列重新排列为{ Rs1, Rs2, …，Rsn }。若整个排序过程都在内存中完成，则称此类排序问题为内部排序。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 排序方法 | 平均时间 | 最坏情况 | 辅助空间 | 稳定性 | 不稳定排序举例 |
| 直接插入排序 | O（n2） | O（n2） | O（1） | 稳定 |  |
| 折半插入排序 | O（n2） | O（n2） | O（1） | 稳定 |  |
| 二路插入排序 | O（n2） | O（n2） | O（n） | 稳定 |  |
| 表插入排序 | O（n2） | O（n2） | O（1） | 稳定 |  |
| 起泡排序 | O（n2） | O（n2） | O（1） | 稳定 |  |
| 直接选择排序 | O（n2） | O（n2） | O（1） | 不稳定 | 2,2’,1 |
| 希尔排序 | O（n1.3） | O（n1.3） | O（1） | 不稳定 | 3,2,2’,1(d=2,d=1) |
| 快速排序 | O（nlog2n） | O（n2） | O（log2n） | 不稳定 | 2,2’,1 |
| 堆排序 | O（nlog2n） | O（nlog2n） | O（1） | 不稳定 | 2,1,1’(极大堆) |
| 2-路归并排序 | O（nlog2n） | O（nlog2n） | O（n） | 稳定 |  |
| 基数排序 | O ( d\*(rd+n) ) | O ( d\*(rd+n) ) | O (rd ) | 稳定 |  |

3. 这种说法不对。因为排序的不稳定性是指两个关键字值相同的元素的相对次序在排序前、后发生了变化，而题中叙述和排序中稳定性的定义无关，所以此说法不对。对4，3，2，1起泡排序就可否定本题结论。

4. 可以做到。取a与b进行比较，c与d进行比较。设a>b,c>d(a<b和c<d情况类似)，此时需2次比较，取b和d比较，若b>d，则有序a>b>d；若b<d时则有序c>d>b，此时已进行了3次比较。再把另外两个元素按折半插入排序方法，插入到上述某个序列中共需4次比较，从而共需7次比较。

5. 本题答案之一请参见第9章的“四、应用题”第70题，这里用分治法求解再给出另一参考答案。

对于两个数x和y，经一次比较可得到最大值和最小值；对于三个数x,y,z，最多经3次比较可得最大值和最小值；对于n(n>3)个数，将分成长为n-2和2的前后两部分A和B，分别找出最大者和最小者：MaxA、MinA、MaxB、MinB，最后Max={MaxA,MaxB}和Min={MinA,MinB}。对A 使用同样的方法求出最大值和最小值，直到元素个数不超过3。设C(n)是所需的最多比较次数，根据上述原则，当n>3时有如下关系式：

C(n)=

通过逐步递推，可以得到：C(n)=⎡3n/2⎤-2。显然，当n>=3时，2n-3>3n/2-2。事实上，⎡3n/2⎤-2是解决这一问题的比较次数的下限。

6. 假定待排序的记录有n个。由于含n个记录的序列可能出现的状态有n!个，则描述n个记录排序过程的判定树必须有n!个叶子结点。因为若少一个叶子，则说明尚有两种状态没有分辨出来。我们知道，若二叉树高度是h，则叶子结点个数最多为2h-1；反之，若有u个叶子结点，则二叉树的高度至少为⎡log2u⎤+1。这就是说，描述n个记录排序的判定树必定存在一条长度为⎡log2(n!)⎤的路径。即任何一个籍助“比较”进行排序的算法，在最坏情况下所需进行的比较次数至少是⎡log2(n!)⎤。根据斯特林公式，有⎡log2(n!)⎤ =O(nlog2n)。即籍助于“比较”进行排序的算法在最坏情况下能达到的最好时间复杂度为O(nlog2n)。证毕。

7. 答：拓扑排序，是有向图的顶点依照弧的走向，找出一个全序集的过程，主要是根据与顶点连接的弧来确定顶点序列；冒泡排序是借助交换思想通过比较相邻结点关键字大小进行排序的算法。

8. 直接插入排序的基本思想是基于插入，开始假定第一个记录有序，然后从第二个记录开始，依次插入到前面有序的子文件中。即将记录R[i](2<=i<=n)插入到有序子序列R[1..i-1]中，使记录的有序序列从R[1..i-1]变为R[1..i] ，最终使整个文件有序。共进行n-1趟插入。最坏时间复杂度是0(n2)，平均时间复杂度是0(n2)，空间复杂度是O(1)，是稳定排序。

简单选择排序的基本思想是基于选择，开始有序序列长度为零，第i(1<=i<n)趟简单选择排序是，从无序序列R[i..n]的n-i+1记录中选出关键字最小的记录，和第i个记录交换，使有序序列逐步扩大，最后整个文件有序。共进行n-1趟选择。最坏时间复杂度是0(n2)，平均时间复杂度是0(n2)，空间复杂度是O(1)，是不稳定排序。

二路并归排序的基本思想是基于归并，开始将具有n个待排序记录的序列看成是n个长度为1的有序序列，然后进行两两归并，得到⎡n/2⎤个长度为2的有序序列，再进行两两归并，得到⎡n/4⎤个长度为4的有序序列。如此重复，经过⎡log2n⎤趟归并，最终得到一个长度为n的有序序列。最坏时间复杂度和平均时间复杂度都是0(nlog2n)，空间复杂度是O(n)，是稳定排序。

9. 错误。快速排序，堆排序和希尔排序是时间性能较好的排序方法，但都是不稳定的排序方法。

10. 等概率（后插），插入位置0..n，则平均移动个数为n/2。

若不等概率，则平均移动个数为=

11. 从节省存储空间考虑：先选堆排序，再选快速排序，最后选择归并排序；

从排序结果的稳定性考虑：选择归并排序。堆排序和快速排序都是不稳定排序；

从平均情况下排序最快考虑：先选择快速排序。

12. (1)堆排序，快速排序，归并排序 (2)归并排序 (3)快速排序 (4)堆排序

13. 平均比较次数最少: 快速排序; 占用空间最多: 归并排序; 不稳定排序算法:快速排序、堆排序、希尔排序。

14. 求前k个最大元素选堆排序较好。因为在建含n个元素的堆时，总共进行的关键字的比较次数不超过4n ,调整建新堆时的比较次数不超过2log2n次。在n个元素中求前k个最大元素，在堆排序情况下比较次数最多不超过4n+2klog2n。

稳定分类是指，若排序序列中存在两个关键字值相同的记录Ri与Rj(Ki=Kj,i≠j)且Ri领先于Rj，若排序后Ri与Rj的相对次序保持不变,则称这类分类是稳定分类（排序），否则为不稳定分类。

A，C和E是稳定分类（排序），B和D是不稳定分类（排序）。

<

<

<

<

<

>

>

>

>

>

15.

16. (1)此为直接插入排序算法，该算法稳定。

(2)r[O]的作用是监视哨，免去每次检测文件是否到尾，提高了排序效率。

采用x.key<=r[j].key描述算法后，算法变为不稳定排序，但能正常工作。

17. (1) 横线内容:①m ②1 ③0 ④1

(2)flag起标志作用。若未发生交换，表明待排序列已有序，无需进行下趟排序。

(3)最大比较次数n(n-1)/2，最大移动次数3n(n-1)/2 (4)稳定

18. (1) ①499 ②A[j]>A[j+1] ③b:=true (2)冒泡排序 (3)499次比较,0次交换

(4) n(n-1)/2次比较，n(n-1)/2次交换（相当3n(n-1)/2次移动），本题中n=500,故有124750次比较和交换（相当373250次移动）。

19. 答：此排序为双向起泡排序：从前向后一趟排序下来得到一个最大值，若其中发生交换，则再从后向前一趟排序，得到一个最小值。 一趟：12,23,35,16,25,36,19,21,16,47

二趟：12,16,23,35,16,25,36,19,21,47 三趟：12,16,23,16,25,35,19,21,36,47

四趟：12,16,16,23,19,25,35,21,36,47 五趟：12,16,16,19,23,25,21,35,36,47

六趟：12,16,16,19,21,23,25,35,36,47 七趟：12,16,16,19,21,23,25,35,36,47

20. 对冒泡算法而言，初始序列为反序时交换次数最多。若要求从大到小排序，则表现为初始是上升序。

21. 证明：起泡排序思想是相邻两个记录的关键字比较，若反序则交换，一趟排序完成得到一个极值。由题假设知Rj在Ri之前且Kj>Ri，即说明Rj和Ri是反序；设对于Ri之前全部记录1—Ri-1(其中包括Kj)中关键字最大为Kmax，则Kmax≥Kj，故经过起泡排序前i-2次后，Ri-1的关键字一定为Kmax，又因Kmax≥Kj>Ri，故Ri-1和Ri为反序，由此可知Ri-1和Ri必定交换，证毕。

22.采用直接插入排序算法，因为记录序列已基本有序，直接插入排序比较次数少，且由于少量次序不对的记录与正确位置不远，使直接插入排序记录移动次数也相对较少，故选直接插入排序算法。

23. 各带标号语句的频度：(1)n (2)n-1 (3)(n+2)(n-1)/2 (4)n(n-1)/2 (5)n-1

时间复杂度O(n²), 属于直接选择排序。

24. 6, 13, 28, 39, 41, 72, 85, 20

i=1↑ m=4↑ r=7↑

20<39 i=1↑m=2↑r=3↑

20>13 i=3 r=3 m=3

20<28 r=2 i=3

将r+1(即第3个)后的元素向后移动，并将20放入r+1处，结果为6,13,20,28,39,41,72,85。

（1）使用二分法插入排序所要进行的比较次数，与待排序的记录的初始状态无关。不论待排序序列是否有序，已形成的部分子序列是有序的。二分法插入首先查找插入位置，插入位置是判定树查找失败的位置。失败位置只能在判定树的最下两层上。

（2）一些特殊情况下，二分法插入排序要比直接插入排序要执行更多的比较。例如，在待排序序列已有序的情况下就是如此。

25. (1)直接插入排序

第一趟 (3)[8,3],2,5,9,1,6 第二趟 (2)[8,3,2],5,9,1,6 第三趟 (5)[8,5,3,2],9,1,6

第四趟 (9)[9,8,5,3,2],1,6 第五趟 (1)[9,8,5,3,2,1],6 第六趟 (6)[9,8,6,5,3,2,1]

(2)直接选择排序（第六趟后仅剩一个元素，是最小的，直接选择排序结束）

第一趟 (9)[9],3,2,5,8,1,6 第二趟 (8)[9,8],2,5,3,1,6 第三趟 (6)[9,8,6],5,3,1,2

第四趟 (5)[9,8,6,5],3,1,2 第五趟 (3)[9,8,6,5,3],1,2 第六趟 (2)[9,8,6,5,3,2],1

(3)直接插入排序是稳定排序，直接选择排序是不稳定排序。

26.这种看法不对。本题的叙述与稳定性的定义无关，不能据此来判断排序中的稳定性。例如 5,4,1,2,3在第一趟冒泡排序后得到4,1,2,3,5。其中4 向前移动，与其最终位置相反。但冒泡排序是稳定排序。快速排序中无这种现象。

27. 125，11，22，34，15，44，76，66，100，8，14，20，2，5，1

设D=7

1，11，8，14，15，2，5，66，100，22，34，20，44，76，125

D=3

1，11，2，5，15，8，14，34，20，22，66，100，44，76，125

D=1 1，2，5，8，11，14，15，20，22，34，44，66，76，100，125

28. 设待排序记录的个数为n，则快速排序的最小递归深度为⎣log2n⎦+1，最大递归深度n。

29. 平均性能最佳的排序方法是快速排序，该排序方法不稳定。

初始序列: 50，10，50，40，45，85，80

一趟排序: [45，10，50，40] 50 [85，80] 二趟排序: [40，10] 45 [50] 50 [80] 85

三趟排序: 10，40，45，50，50，80，85

30. 快速排序。

31. (1) 在最好情况下，假设每次划分能得到两个长度相等的子文件，文件的长度n=2k-1，那么第一遍划分得到两个长度均为⎣n/2⎦的子文件，第二遍划分得到4个长度均为⎣n/4⎦的子文件，以此类推，总共进行k=log2(n+1)遍划分，各子文件的长度均为1，排序完毕。当n=7时，k=3，在最好情况下，第一遍需比较6次，第二遍分别对两个子文件（长度均为3，k=2）进行排序，各需2次，共10次即可。

(2) 在最好情况下快速排序的原始序列实例：4,1,3,2,6,5,7。

(3) 在最坏情况下，若每次用来划分的记录的关键字具有最大值(或最小值)，那么只能得到左(或右)子文件，其长度比原长度少1。因此，若原文件中的记录按关键字递减次序排列，而要求排序后按递增次序排列时，快速排序的效率与冒泡排序相同，其时间复杂度为O(n2)。所以当n=7时，最坏情况下的比较次数为21次。

(4) 在最坏情况下快速排序的初始序列实例： 7,6,5,4,3,2,1，要求按递增排序。

32.该排序方法为快速排序。

33. 不是。因为当序列已有序时，快速排序将退化成冒泡排序，时间复杂度为O(n2)。当待排序序列无序，使每次划分完成后，枢轴两侧子文件长度相当，此时快速排序性能最好。

34. 初始序列：[28],07,39,10,65,14,61,17,50,21 21移动：21,07,39,10,65,14,61,17,50,[]

39移动：21,07,[],10,65,14,61,17,50,39 17移动：21,07,17,10,65,14,61,[],50,39

65移动：21,07,17,10,[],14,61,65,50,39 14移动：21,07,17,10,14,[28],61,65,50,39

***类似本题的另外叙述题***的解答**：**

（1）：快速排序思想：首先将待排序记录序列中的所有记录作为当前待排序区域，以第一个记录的关键字作为**枢轴**（或支点）（pivot），凡其关键字不大于枢轴的记录均移动至该记录之前，凡关键字不小于枢轴的记录均移动至该记录之后。致使一趟排序之后，记录的无序序列R[s..t]将分割成两部分：R[s..i-1]和R[i+1..t],且R[j].key≤R[i].key≤R[k].key(s≤j<i,i<k≤t),然后再递归地将R[s..i-1]和R[i+1..t]进行快速排序。快速排序在记录有序时蜕变为冒泡排序，可用“三者取中”法改善其性能，避免最坏情况的出现。具体排序实例不再解答。

35．（1）不可以，若m+1到n之间关键字都大于m的关键字时,<=k可将j定位到m上，若为<k则j将定位到m-1上，将出界线，会造成错误。

（2）不稳定，例如2，1，2´，（m=1,n=3）对2，1，2´排序，完成会变成1，2´，2。

（3）各次调用qsort1的结果如下：

一次调用m=1,n=10 11,3,16,4,18,22,58,60,40,30 j=6

二次调用m=7,n=10 11,3,16,4,18,22,40,30,58,60 j=9 (右部)

三次调用m=10,n=10 不变，返回 m=7,n=8 11,3,16,4,18,22,30,40,58,60 j=8

四次调用m=9,n=8 不变，返回 m=7,n=7 返回 m=1,n=5 4,3,11,16,18,22,30,40,58,60 j=3（左部）

五次调用m=1,n=2 3,4,11,16,18,22,30,40,58,60 j=2

六次调用m=1,n=1 不变，返回 m=3,n=2 返回 m=4,n=5 3,4,11,16,18,22,30,40,58,60 j=4

七次调用m=4,n=3 不变，返回 （注：一次划分后，左、右两部分调用算两次）

（4）最大栈空间用量为O(logn)。

36. 在具有n个元素的集合中找第k（1≤k≤n）个最小元素，应使用快速排序方法。其基本思想如下：设n个元素的集合用一维数组表示，其第一个元素的下标为1，最后一个元素下标为n。以第一个元素为“枢轴”，经过快速排序的一次划分，找到“枢轴”的位置i，若i=k，则该位置的元素即为所求；若i>k，则在1至i-1间继续进行快速排序的划分；若i<k，则在i+1至n间继续进行快速排序的划分。这种划分一直进行到i=k为止，第i位置上的元素就是第k（1≤k≤n）个最小元素。

37. 快速排序各次调用结果：

一次调用：18,36,77,42,23,65,84,10,59,37,61,98

二次调用：10,18,77,42,23,65,84,36,59,37,61,98

三次调用：10,18,61,42,23,65,37,36,59,77,84,98

归并排序各次调用结果：

一次调用36,98,42,77,23,65,10,84,37,59,18,61, (子文件长度为1，合并后子文件长度为2)

二次调用36,42,77,98,10,23,65,84,18,37,59,61 (子文件长度为2，合并后子文件长度为4)

三次调用10,23,36,42,65,77,84,98,18,37,59,61 (第一子文件长度8，第二子文件长度为4)

38. 建立堆结构:97,87,26,61,70,12,3,45 (2)70,61,26,3,45,12,87,97

(4)45,12,26,3,61,70,87,97 (6)12,3,26,45,61,70,87,97

39. (1)是大堆； (2)是大堆；（4）是小堆；

(3)不是堆，调成大堆 100，98，66，85，80，60，40，77，82，10，20

*类似叙述*(1):不是堆 调成大顶堆 92，86，56，70，33，33，48，65，12

(2)①是堆 ②不是堆 调成堆 100，90，80，25，85，75，60，20，10，70，65，50

(3)①是堆 ②不是堆 调成大堆21，9，18，8，4，17，5，6（图略） (4):略

40. 在内部排序方法中，一趟排序后只有简单选择排序和冒泡排序可以选出一个最大（或最小）元素，并加入到已有的有序子序列中，但要比较n-1次。选次大元素要再比较n-2次,其时间复杂度是O(n2)。从10000个元素中选10个元素不能使用这种方法。而快速排序、插入排序、归并排序、基数排序等时间性能好的排序，都要等到最后才能确定各元素位置。只有堆排序，在未结束全部排序前，可以有部分排序结果。建立堆后，堆顶元素就是最大（或最小，视大堆或小堆而定）元素，然后，调堆又选出次大（小）元素。凡要求在n个元素中选出k(k<<n,k>2)个最大（或最小）元素，一般均使用堆排序。因为堆排序建堆比较次数至多不超过4n,对深度为k的堆，在调堆算法中进行的关键字的比较次数至多为2(k-1)次，且辅助空间为O(1)。

41. (1)建小堆

(2) 求次小值

42. 用堆排序方法，详见第40题的分析。从序列{59，11，26，34，14，91，25}得到{11，17，25}共用14次比较。其中建堆输出11比较8次，调堆输出17和25各需要比较4次和2次。

***类似本题的另外叙述题***的解答**：**

（1）堆排序，分析同前，共20+5+4+5=34 次比较。

43. 对具体例子的手工堆排序略。

堆与败者树的区别：堆是n个元素的序列，在向量中存储，具有如下性质：

 （i=1,2,…,⎣n/2⎦）

由于堆的这个性质中下标i和2i、2i+1的关系，恰好和完全二叉树中第i个结点和其子树结点的序号间的关系一致，所以堆可以看作是n个结点的完全二叉树。而败者树是由参加比赛的n个元素作叶子结点而得到的完全二叉树。每个非叶（双亲）结点中存放的是两个子结点中的败者数据，而让胜者去参加更高一级的比赛。另外，还需增加一个结点，即结点0，存放比赛的全局获胜者。

44.（1）堆的存储是顺序的

（2）最大值元素一定是叶子结点，在最下两层上。

（3）在建含有n个元素、深度为h的堆时，其比较次数不超过4n，推导如下：

由于第i层上的结点数至多是2i-1,以它为根的二叉树的深度为h-i+1，则调用⎣n/2⎦次筛选算法时总共进行的关键字比较次数不超过下式之值：



45.(1)

冠军

(2)树形选择排序基本思想：首先对n个待排序记录的关键字进行两两比较，从中选出⎡n/2⎤个较小者再两两比较，直到选出关键字最小的记录为止，此为一趟排序。我们将一趟选出的关键字最小的记录称为“冠军”，而“亚军”是从与“冠军”比较失败的记录中找出，具体做法为：输出“冠军”后，将其叶子结点关键字改为“最大值”，然后从该叶子结点开始，和其左（或右）兄弟比较，修改从叶子结点到根结点路径上各结点的关键字，则根结点的关键字即为次小关键字。如此下去，可依次选出从小到大的全部关键字。

(3) 树形选择排序与直接选择排序相比较，其优点是从求第2个元素开始，从n-i+1个元素中不必进行n-i次比较，只比较⎣log2n⎦次，比较次数远小于直接选择排序；其缺点是辅助储存空间大。

(4) 堆排序基本思想是：堆是n个元素的序列，先建堆，即先选得一个关键字最大或最小的记录，然后与序列中最后一个记录交换，之后将序列中前n-1记录重新调整为堆（调堆的过程称为“筛选”），再将堆顶记录和当前堆序列的最后一个记录交换，如此反复直至排序结束。优点是在时间性能与树形选择排序属同一量级的同时，堆排序只需要一个记录大小供交换用的辅助空间，调堆时子女只和双亲比较。避免了过多的辅助存储空间及和最大值的比较，

46. K1到Kn是堆，在Kn+1加入后，将K1..Kn+1调成堆。设c=n+1,f=⎣c/2⎦,若Kf<=Kc，则调整完成。否则，Kf与Kc交换；之后，c=f,f=⎣c/2⎦，继续比较，直到Kf<=Kc，或f=0，即为根结点，调整结束。

47. (1)①child=child+1; ②child/2 (2)不能，调为大堆：92,86,56,70,33,33,48,65,12,24

48. (1)不需要。因为建堆后R[1]到R[n]是堆，将R[1]与R[n]交换后，R[2]到R[n-1]仍是堆，故对R[1]到R[n-1]只需从R[1]往下筛选即可。

(2) 堆是n个元素的序列，堆可以看作是n个结点的完全二叉树。而树型排序是n个元素作叶子结点的完全二叉树。因此堆占用的空间小。调堆时，利用堆本身就可以存放输出的有序数据，只需要一个记录大小供交换用的辅助空间。排序后，heap数组中的关键字序列与堆是大堆还是小堆有关，若利用大堆，则为升序；若利用小堆则为降序。

49. 最高位优先（MSD）法：先对最高位关键字K0进行排序，将序列分成若干子序列，每个子序列中的记录都具有相同的K0值，然后，分别就每个子序列对关键字K1进行排序，按K1值不同再分成若干更小的子序列，……，依次重复，直至最后对最低位关键字排序完成，将所有子序列依次连接在一起，成为一个有序子序列。

最低位优先（LSD）法：先对最低位关键字Kd-1进行排序，然后对高一级关键字Kd-2进行排序，依次重复，直至对最高位关键字K0排序后便成为一个有序序列。进行排序时，不必分成子序列，对每个关键字都是整个序列参加排序，但对Ki (0<=i<d-1)排序时，只能用稳定的排序方法。另一方面，按LSD进行排序时，可以不通过关键字比较实现排序，而是通过若干次“分配”和“收集”来实现排序。

50. (1)冒泡排序(H,C,Q,P,A,M,S,R,D,F,X,Y)

(2)初始步长为4的希尔排序(P,A,C,S,Q,D,F,X,R,H,M,Y)

(3)二路归并排序(H,Q,C,Y,A,P,M,S,D,R,F,X) (4)快速排序(F,H,C,D,P,A,M,Q,R,S,Y,X)

初始建堆:(A,D,C,R,F,Q,M,S,Y,P,H,X)

51. (1)一趟希尔排序: 12,2,10,20,6,18,4,16,30,8,28 （ D=5）

(2)一趟快速排序：6,2,10,4,8,12,28,30,20,16,18

(3)链式基数排序LSD [0][1][2][3][4][5][6][7][8][9]

↓ ↓ ↓ ↓ ↓

分配 30 12 4 16 8

↓ ↓ ↓ ↓

10 2 6 28

↓ ↓

20 18

收集：→30→10→20→12→2→4→16→6→8→28→18

52. (1)2路归并 第一趟:18,29,25,47,12,58,10,51；第二趟:18,25,29,47,10,12,51,58；

第三趟:10,12,18,25,29,47,51,58

(2)快速排序 第一趟:10,18,25,12,29,58,51,47；

第二趟:10,18,25,12,29,47,51,88；第三趟:10,12,18,25,29,47,51,88

(3)堆排序 建大堆:58,47,51,29,18,12,25,10；

①51,47,25,29,18,12,10,58；②47,29,25,10,18,12,51,58；

③29,18,25,10,12,47,51,58；④25,18,12,10,29,47,51,58；

⑤18,10,12,25,29,47,51,58；⑥12,10,18,25,29,47,51,58；⑦10,12,18,25,29,47,51,58

*类似叙述*：(1) ①设按3路归并 I/O次数=2\*wpl=202次 ②

|  |
| --- |
| 7 |

|  |
| --- |
| 18 |

|  |
| --- |
| 24 |

|  |
| --- |
| 52 |

③④⑤ 略。

53. (1)当至多进行n-1趟起泡排序，或一趟起泡排序中未发生交换（即已有序）时，结束排序。

(2)希尔排序是对直接插入排序算法的改进，它从“记录个数少”和“基本有序”出发，将待排序的记录划分成几组（缩小增量分组），从而减少参与直接插入排序的数据量，当经过几次分组排序后，记录的排列已经基本有序，这个时候再对所有的记录实施直接插入排序。

(3)13，24，33，65，70，56，48，92，80，112

(4)采用树型锦标赛排序选出最小关键字至少要15次比较。再选出次小关键字比较4次。（两两比较8次选出8个优胜，再两两比较4次选出4个优胜，半决赛两场，决赛一场，共比较了15次。将冠军的叶子结点改为最大值，均与兄弟比较，共4次选出亚军。）

54. (1)按LSD法 →321→156→57→46→28→7→331→33→34→63

分配 [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]

321 33 34 156 57 28

331 63 46 7

收集 →321→331→33→63→34→156→46→57→7→28

*类似叙述(1)* 略。

55. ①快速排序 ②冒泡排序 ③直接插入排序 ④堆排序

56. A.p[k]←j B.i←i+1 C.k=0 D.m←n E.m<n F.a[i]←a[m] G.a[m]←t

57. 一趟快速排序：22,19,13,6,24,38,43,32

初始大堆：43,38,32,22,24,6,13,19

二路并归：第一趟：19,24,32,43,6,38,13,22

第二趟：19,24,32,43,6,13,22,38

第三趟：6,13,19,22,24,32,38,43

堆排序辅助空间最少，最坏情况下快速排序时间复杂度最差。

58. (1)排序结束条件为没有交换为止

第一趟奇数：35,70,33,65,21,24,33 第二趟偶数：35,33,70,21,65,24,33

第三趟奇数：33,35,21,70,24,65,33 第四趟偶数：33,21,35,24,70,33,65

第五趟奇数：21,33,24,35,33,70,65 第六趟偶数：21,24,33,33,35,65,70

第七趟奇数：21,24,33,33,35,65,70(无交换) 第八趟偶数：21,24,33,33,35,65,70(无交换) 结束

59. 设归并路数为k，归并趟数为s，则s=⎡logk100⎤，因⎡logk100⎤=3 ，所以k=5,即最少5路归并。

60. 证明：由置换选择排序思想，第一个归并段中第一个元素是缓冲区中最小的元素，以后每选一个元素都不应小于前一个选出的元素，故当产生第一个归并段时（即初始归并段），缓冲区中m个元素中除最小元素之外，其他m-1个元素均大于第一个选出的元素，即当以后读入元素均小于输出元素时，初始归并段中也至少能有原有的m个元素。证毕。

|  |
| --- |
| 5 |

|  |
| --- |
| 14 |

|  |
| --- |
| 26 |

|  |
| --- |
| 58 |

61. 因(8-1)%(3-1)=1,故第一次归并

时加一个“虚段”。

WPL=5\*3+(4+5+7+9+10)\*2+18\*1

=103

*类似叙述*：

(1)加4-(12-1)%(4-1)-1=1个虚段

|  |
| --- |
| 17 |

|  |
| --- |
| 64 |

|  |
| --- |
| 196 |

|  |
| --- |
| 413 |

WPL=(3+6+8)\*3+(9+18+20+30+44+60+62)\*2+(68+85)\*1

=690 (2)(3)(4)略。

62. 加5-(31-1)%(5-1)-1=2个虚段。

|  |
| --- |
| 6 |

|  |
| --- |
| 261 |

|  |
| --- |
| 10 |

|  |
| --- |
| 52 |

|  |
| --- |
| 15 |

|  |
| --- |
| 19 |

|  |
| --- |
| 78 |

|  |
| --- |
| 25 |

总读写次数为2\*wpl=800次。

*类似叙述*(1)(2)(3)略。

63. 内部排序中的归并排序是在内存中进行的归并排序，辅助空间为O(n)。外部归并排序是将外存中的多个有序子文件合并成一个有序子文件，将每个子文件中记录读入内存后的排序方法可采用多种内排序方法。外部排序的效率主要取决于读写外存的次数，即归并的趟数。因为归并的趟数s=⎡logkm⎤,其中，m是归并段个数，k是归并路数。增大k和减少m都可减少归并趟数。应用中通过败者树进行多（k）路平衡归并和置换-选择排序减少m，来提高外部排序的效率。

64. 初始败者树

|  |
| --- |
| 71 |

|  |
| --- |
| 1 |

|  |
| --- |
| 3 |

|  |
| --- |
| 1 |

|  |
| --- |
| 61 |

|  |
| --- |
| 1 |

|  |
| --- |
| 19 |

|  |
| --- |
| 1 |

|  |
| --- |
| 51 |

|  |
| --- |
| 1 |

|  |
| --- |
| 101 |

|  |
| --- |
| 1 |

Ls[0]

Ls[1]

Ls[2]

Ls[3]

Wa[1]

Wa[0]

Ls[5]

Ls[4]

Wa[3]

Wa[4]

Wa[5]

Wa[2]

初始归并段： R1:3,19,31,51,61,71,100,101

R2:9,17,19,20,30,50,55,90

R3:6

*类似叙述*题略。

65. (1)4台磁带机：平衡归并只能用2路平衡归并，需归并⎡log2650⎤=10趟。多步归并进行3路归并，按3阶广义斐波那契序列。F11<650<F12=653,即应补3个虚段。 进行：12-2=10 趟归并。

t1=f11+f10+f9=149+81+44=274 t2= f11+f10=149+81=230 t3=f11=149

(2) 多步归并排序前五趟归并的情况如下：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |
| T1 | 1149+181+144 | 1步 | T1 | 181+144 | 2步 | T1 | 144 | 3步 | T1 |  | 4步 |
| T2 | 1149+181 |  | T2 | 181 |  | T2 |  |  | T2 | 944 |  |
| T3 | 1149 |  | T3 |  |  | T3 | 581 |  | T3 | 537 |  |
| T4 |  |  | T4 | 3149 |  | T4 | 368 |  | T4 | 324 |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
| T1 | 1724 | 5步 | T1 | 1711 | 注：mp表示p个长为m单位的归并段 |
| T2 | 920 |  | T2 | 97 | 1149表示149个长为1个单位的归并段。 |
| T3 | 513 |  | T3 |  |  |
| T4 |  |  | T4 | 3113 | *类似叙述*题略 |

66. (1) (2)

|  |
| --- |
| 13 |

|  |
| --- |
| 24 |

|  |
| --- |
| 52 |

*类似叙述*题略。

67.外排序用k-路归并(k>2)是因为k越小，归并趟数越多，读写外存次数越多，时间效率越低，故，一般应大于最少的2路归并。 若将k-路归并的败者树思想单纯用于内排序，因其由胜者树改进而来，且辅助空间大，完全可由堆排序取代，故将其用于内排序效率并不高。

68. R1:19,48,65,74,101 R2:3,17,20,21,21,33,53,99

五.算法设计题

1. **void** BubbleSort2(**int** a[],**int** n) //相邻两趟向相反方向起泡的冒泡排序算法

{ change=1;low=0;high=n-1; //冒泡的上下界

**while**(low<high && change)

{ change=0; //设不发生交换

**for**(i=low;i<high;i++) //从上向下起泡

**if**(a[i]>a[i+1]){a[i]<-->a[i+1];change=1;} //有交换，修改标志change

high--; //修改上界

**for**(i=high;i>low;i--) //从下向上起泡

**if**(a[i]<a[i-1]){a[i]<-->a[i-1];change=1;}

low++; //修改下界

}//**while**

}//BubbleSort2

[算法讨论]题目中“向上移”理解为向序列的右端，而“向下移”按向序列的左端来处理。

2. **typedef** **struct** node

{ ElemType data;

**struct** node \*prior,\*next;

}node，\*DLinkedList;

**void** TwoWayBubbleSort(DLinkedList la)

//对存储在带头结点的双向链表la中的元素进行双向起泡排序。

{**int** exchange=1; // 设标记

DLinkedList p,temp,tail;

head=la //双向链表头，算法过程中是向下起泡的开始结点

tail=null; //双向链表尾，算法过程中是向上起泡的开始结点

**while** (exchange)

{p=head->next; //p是工作指针，指向当前结点

exchange=0; //假定本趟无交换

**while** (p->next!=tail) // 向下（右）起泡，一趟有一最大元素沉底

**if** (p->data>p->next->data) //交换两结点指针，涉及6条链

{temp=p->next; exchange=1;//有交换

p->next=temp->next;temp->next->prior=p //先将结点从链表上摘下

temp->next=p; p->prior->next=temp; //将temp插到p结点前

temp->prior=p->prior; p->prior=temp;

}

**else** p=p->next; //无交换，指针后移

tail=p; //准备向上起泡

p=tail->prior;

**while** (exchange && p->prior!=head) //向上（左）起泡，一趟有一最小元素冒出

**if** (p->data<p->prior->data) //交换两结点指针，涉及6条链

{temp=p->prior; exchange=1; //有交换

p->prior=temp->prior;temp->prior->next=p； //先将temp结点从链表上摘下

temp->prior=p; p->next->prior=temp; //将temp插到p结点后（右）

temp->next=p->next; p->next=temp;

}

**else** p=p->prior; //无交换，指针前移

head=p; //准备向下起泡

}// **while** (exchange)

} //算法结束

3. **PROCEDURE** StraightInsertSort(**VAR** R:listtype;n:integer);

**VAR** i,j:integer;

**BEGIN**

**FOR** i:=2 **TO** n **DO** {假定第一个记录有序}

**BEGIN**

R[0]:=R[i]; j:=i-1; {将待排序记录放进监视哨}

**WHILE** R[0].key<R[j].key **DO** {从后向前查找插入位置，同时向后移动记录}

**BEGIN** R[j+1]:=R[j]; j:=j-1; **END**;

R[j+1]:=R[0] {将待排序记录放到合适位置}

**END** {**FOR**}

**END**；

4. **TYPE** pointer=↑node;

node=**RECORD** key:integer; link:pointer; **END**；

**PROCEDURE** LINSORT(L:pointer);

**VAR** t,p,q,s:pointer;

**BEGIN**

p:=L↑.link↑.link; {链表至少一个结点，p初始指向链表中第二结点（若存在）}

L↑.link↑.link=NIL; {初始假定第一个记录有序}

**WHILE** p<>NIL **DO**

**BEGIN** q:=p↑.link; {q指向p的后继结点}

s=L;

**WHILE** (s↑.link<>NIL AND s↑.link↑.key<p↑.key) **DO**

s:=s↑.link; {向后找插入位置}

p↑.link:=s↑.link; s↑.link=p;{插入结点}

p=q; {恢复p指向当前结点}

**END** {**WHILE**}

**END**; {LINSORT}

5. **typedef** **struct**

{ **int** num; **float** score; }RecType;

**void** SelectSort(RecType R[51]，**int** n)

{ **for**(i=1; i<n; i++)

{ //选择第i大的记录，并交换到位

k=i; //假定第i个元素的关键字最大

**for**(j=i+1;j<=n;j++) //找最大元素的下标

**if**(R[j].score>R[k].score) k=j;

**if**(i!=k) R[i] <-->R[k]; //与第i个记录交换

}//**for**

**for**(i=1; i<=n; i++) //输出成绩

{ printf("%d,%f",R[i].num,R[i].score); **if**(i%10==0) printf("\n");}

}//SelectSort

6. **typedef** **struct**

{ **int** key; datatype info}RecType

**void** CountSort(RecType a[],b[],**int** n) //计数排序算法，将a中记录排序放入b中

{ **for**(i=0;i<n;i++) //对每一个元素

{ **for**(j=0,cnt=0;j<n;j++)

**if**(a[j].key<a[i].key) cnt++; //统计关键字比它小的元素个数

b[cnt]=a[i];

}

}//Count\_Sort

(3) 对于有n个记录的表，关键码比较n2次。

(4) 简单选择排序算法比本算法好。简单选择排序比较次数是n(n-1)/2,且只用一个交换记录的空间；而这种方法比较次数是n2，且需要另一数组空间。

[算法讨论]因题目要求“针对表中的每个记录，扫描待排序的表一趟”，所以比较次数是n2次。若限制“对任意两个记录之间应该只进行一次比较”，则可把以上算法中的比较语句改为：

**for**(i=0;i<n;i++) a[i].count=0;//各元素再增加一个计数域，初始化为0

**for**(i=0;i<n;i++)

**for**(j=i+1;j<n;j++)

**if**(a[i].key<a[j].key) a[j].count++; **else** a[i].count++;

7. [题目分析]保存划分的第一个元素。以平均值作为枢轴，进行普通的快速排序，最后枢轴的位置存入已保存的第一个元素，若此关键字小于平均值，则它属于左半部，否则属于右半部。

**int** partition (RecType r[],**int** l,h)

{ **int** i=l,j=h,avg=0;

**for**(;i<=h;i++) avg+=R[i].key;

i=l; avg=avg/(h-l+1);

**while** (i<j)

{ **while** (i<j &&R[j].key>=avg) j--;

**if** (i<j) R[i]=R[j];

**while** (i<j &&R[i].key<=avg) i++;

**if** (i<j) R[j]=R[i];

}

**if**(R[i].key<=avg) **return**  i; **else** **return**  i-1;

}

**void** quicksort (RecType R[],**int** S,T);

{**if** (S<T)

{k=partition (R,S,T);

quicksart (R,S,k);

quicksart (R,k+1,T);}

}

8. **int**  Partition(RecType R[]，**int** l，**int** h)

//一趟快速排序算法，枢轴记录到位，并返回其所在位置，

{ **int** i=l; j=h; R[0] = R[i]; x = R[i].key;

**while**(i<j)

{ **while**(i<j && R[j].key>=x) j--;

**if** (i<j) R[i] = R[j];

**while**(i<j && R[i].key<=x) i++;

**if** (i<j) R[j] = R[i];

}//**while**

R[i]=R[0];

**return**  i;

}//Partition

9. [题目分析]以Kn为枢轴的一趟快速排序。将上题算法改为以最后一个为枢轴先从前向后再从后向前。

**int**  Partition(RecType K[]，**int** l，**int** n)

{ //交换记录子序列K[l..n]中的记录，使枢轴记录到位，并返回其所在位置，

//此时，在它之前（后）的记录均不大（小）于它

**int** i=l; j=n; K[0] = K[j]; x = K[j].key;

**while**(i<j)

{ **while**(i<j && K[i].key<=x) i++;

**if** (i<j) K[j]=K[i];

**while**(i<j && K[j].key>=x) j--;

**if** (i<j) K[i]=K[j];

}//**while**

K[i]=K[0]; **return**  i;

}//Partition

10. [题目分析]把待查记录看作枢轴，先由后向前依次比较，若小于枢轴，则从前向后，直到查找成功返回其位置或失败返回0为止。

**int** index (RecType R[],**int** l,h,datatype key)

{ **int** i=l,j=h;

**while** (i<j)

{ **while** (i<=j && R[j].key>key) j--;

**if** (R[j].key==key) **return**  j;

**while** (i<=j && R[i].key<key) i++;

**if** (R[i].key==key) **return**  i;

}

printf(“Not find”) ; **return**  0;

}//index

11. (1) [题目分析]从第n个记录开始依次与其双亲（n/2）比较，若大于双亲则交换，继而与其双亲的双亲比较，以此类推直到根为止。

**void** s**if**t(RecType R[],**int** n)

{ //假设 R[1..n-1]是大堆，本算法把R[1..n]调成大堆

j=n; R[0]=R[j];

**for** (i=n/2;i>=1;i=i/2)

**if** (R[0].key>R[i].key){ R[j]=R[i];j=i;} **else** **break**;

R[j]=R[0];

}//s**if**t

(2)**void** HeapBuilder(RecType R[],**int** n)

{ **for** (i=2;i<=n;i++) s**if**t (R,i); }

12. **void** sort (RecType K[],**int** n)

{ **for** (i=1;i<=n;i++) T[i]=i;

**for** (i=1;i<n;i++)

**for** (j=1;j<=n-i;j++)

**if** (K[T[j]]>K[T[j+1]]) {t=T[j];T[j]=T[j+1];T[j+1]=t;}

}//sort

[算法讨论] 上述算法得到辅助地址表，T[i]的值是排序后K的第i个记录，要使序列K有序，则要按T再物理地重排K的各记录。算法如下：

**void** Rearrange(RecType K[],**int** T[],n)

//对有n个记录的序列K，按其辅助地址表T进行物理非递减排序

{**for**(i=1;i<=n;i++)

**if** (T[i]!=i)

{j=i; rc=K[i]; //暂存记录K[i]

**while** (T[j]!=i) //调整K[T[j]]到T[j]=i为止

{m=T[j]; K[j]=K[m]; T[j]=j; j=m;}

K[j]=rc; T[j]=j; //记录R[i]到位

}//**if**

}//Rearrange

13. (1) 堆排序是对树型选择排序的改进，克服了树型选择排序的缺点，其定义在前面已多次谈到，请参见上面“四、应用题”的43题和45题（4）。“筛选”是堆排序的基础算法。由于堆可以看作具有n个结点的完全二叉树，建堆过程是从待排序序列第一个非终端结点⎣n/2⎦开始，直到根结点，进行“筛选”的过程。堆建成后，即可选得一个关键字最大或最小的记录。然后堆顶元素与序列中最后一个元素交换，再将序列中前n-1记录重新调整为堆，可选得一个关键字次大或次小的记录。依次类推，则可得到元素的有序序列。

(2) **void** Sift(RecType R[]，**int** i，**int** m)

{ //假设R[i+1..m]中各元素满足堆的定义，本算法调整R[i]使序列R[i..m]中各元素满足堆的性质

R[0]=R[i];

**for**(j=2\*i; j<=m; j\*=2)

{ **if**(j<m && R[j].key<R[j+l].key) j++; //建大根堆

**if**(R[0].key<R[j].key) { R[i]=R[j]; i=j;} **else** **break**;

}//**for**

R[i]=R[0];

}//Sift

**void** HeapSort(RecType R[]，**int** n)

{ //对记录序列R[1..n]进行堆排序。

**for**(i=n/2;i>0;i--) S**if**t(R,i,n);

**for**(i=n;i>1;i--){ R[1]<-->R[i]; S**if**t(R,1,i-1);}//**for**

}//HeapSort

(3)堆排序的时间主要由建堆和筛选两部分时间开销构成。对深度为h的堆，“筛选”所需进行的关键字比较的次数至多为2(h－1)；对n个关键字，建成深度为h(=⎣log2n⎦+1)的堆，所需进行的关键字比较的次数至多C (n)，它满足下式：









调整“堆顶”n-1次，总共进行的关键字比较的次数不超过：2(⎣log2(n-1)⎦+ ⎣log2(n-2)⎦+ …+log22)<2n(⎣log2n⎦)因此，堆排序的时间复杂度为O（nlog2n）。

14. **PROC** LinkedListSelectSort( head: pointer);

//本算法一趟找出一个关键字最小的结点，其数据和当前结点进行交换;若要交换指针，则须记下

//当前结点和最小结点的前驱指针

p:=head↑.next;

**WHILE** p<>NIL **DO**

[q:=p↑.next; r:=p; //设r是指向关键字最小的结点的指针

**WHILE** <q<>NIL **DO**

[**IF** q↑.data<r↑.data **THEN** r:=q;

q:=q↑.next;

]

**IF** r<>p **THEN** r↑.data<-->p↑.data

p:=p↑.next;

]

**ENDP**;

15. **void** QuickSort(rectype r[n+1]; **int** n)

// 对r[1..n]进行快速排序的非递归算法

{**typedef** **struct**

{ **int** low,high; }node

node s[n+1]；//栈，容量足够大

**int** quickpass(rectype r[],int,int); // 函数声明

**int** top=1; s[top].low=1; s[top].high=n;

**while** (top>0)

{ss=s[top].low; tt=s[top].high; top--;

**if** (ss<tt)

{k=quickpass(r,ss,tt);

**if** (k-ss>1) {s[++top].low=ss; s[top].high=k-1;}

**if** (tt-k>1) {s[++top].low=k+1; s[top].high=tt;}

}

} // 算法结束

**int** quickpass(rectype r[];**int** s,t)

{i=s; j=t; rp=r[i]; x=r[i].key;

**while** （i<j）

{**while** (i<j && x<=r[j].key) j--;

**if** (i<j) r[i++]=r[j];

**while** (i<j && x>=r[j].key) i++;

**if** (i<j) r[j--]=r[i];;

]

r[i]=rp;

**return**  (i);

} // 一次划分算法结束

[算法讨论]可对以上算法进行两点改进：一是在一次划分后，先处理较短部分，较长的子序列进栈；二是用“三者取中法”改善快速排序在最坏情况下的性能。下面是部分语句片段：

**int** top=1; s[top].low=1; s[top].high=n;

ss=s[top].low; tt=s[top].high; top--; flag=true;

**while** (flag || top>0)

{k=quickpass(r,ss,tt);

**if** (k-ss>tt-k) // 一趟排序后分割成左右两部分

{**if** (k-ss>1) // 左部子序列长度大于右部，左部进栈

{s[++top].low=ss; s[top].high=k-1; }

**if** (tt-k>1) ss=k+1; // 右部短的直接处理

**else** flag=false; // 右部处理完，需退栈

}

**else** **if** (tt-k>1) //右部子序列长度大于左部，右部进栈

{s[++top].low=k+1; s[top].high=tt; }

**if** (k-ss>1) tt=k-1 // 左部短的直接处理

**else** flag=false // 左部处理完，需退栈

}

**if** (!flag && top>0)

{ss=s[top].low; tt=s[top].high; top--; flag=true;}

} // end of **while** (flag || top>0)

} // 算法结束

**int** quickpass(rectype r[];**int** s,t)

// 用“三者取中法”进行快速排序的一次划分

{ **int**  i=s, j=t, mid=(s+t)/2;

rectype tmp;

**if** (r[i].key>r[mid].key) {tmp=r[i];r[i]=r[mid];r[mid]=tmp }

**if** (r[mid].key>r[j].key)

{tmp=r[j];r[j]=r[mid];

**if** (tmp>r[i]) r[mid]=tmp; **else** {r[mid]=r[i];r[i]=tmp }

}

{tmp=r[i];r[i]=r[mid];r[mid]=tmp }

// 三者取中：最佳2次比较3次赋值；最差3次比较10次赋值

rp=r[i]; x=r[i].key;

**while** （i<j）

{**while** (i<j && x<=r[j].key) j--;

**if** (i<j) r[i++]=r[j];

**while** (i<j && x>=r[j].key) i++;

**if** (i<j) r[j--]=r[i];;

]

r[i]=rp;

**return**  (i);

} // 一次划分算法结束

|  |
| --- |
| 70 |

|  |
| --- |
| 45 |

|  |
| --- |
| 7 |

|  |
| --- |
| 50 |

|  |
| --- |
| 30 |

|  |
| --- |
| 20 |

|  |
| --- |
| 12 |

|  |
| --- |
| 30 |

|  |
| --- |
| 9 |

|  |
| --- |
| 10 |

|  |
| --- |
| 15 |

|  |
| --- |
| 80 |

|  |
| --- |
| 40 |

|  |
| --- |
| 70 |

|  |
| --- |
| 45 |

|  |
| --- |
| 5 |

|  |
| --- |
| 50 |

|  |
| --- |
| 30 |

|  |
| --- |
| 20 |

|  |
| --- |
| 12 |

|  |
| --- |
| 30 |

|  |
| --- |
| 9 |

|  |
| --- |
| 7 |

|  |
| --- |
| 15 |

|  |
| --- |
| 40 |

|  |
| --- |
| 10 |

16. （1） （2）

加入5后 加入80后

（3）[题目分析]从插入位置进行调整，调整过程由下到上。首先根据元素个数求出插入元素所在层次数，以确定其插入层是最大层还是最小层。若插入元素在最大层，则先比较插入元素是否比双亲小，如是，则先交换，之后，将小堆与祖先调堆，直到满足小堆定义或到达根结点；若插入元素不小于双亲，则调大堆，直到满足大堆定义。若插入结点在最小层，则先比较插入元素是否比双亲大，如是，则先交换，之后，将大堆与祖先调堆；若插入结点在最小层且小于双亲，则将小堆与祖先调堆，直到满足小堆定义或到达根结点。

（4）**void** MinMaxHeapIns(RecType R[],**int** n)

{ //假设R[1..n-1]是最小最大堆，插入第n个元素，把R[1..n]调成最小最大堆

j=n; R[0]=R[j];

h=⎣log2n⎦+1; //求高度

**if** (h%2==0) //插入元素在偶数层，是最大层

{i=n/2;

**if** (R[0].key<R[i].key) //插入元素小于双亲，先与双亲交换，然后调小堆

{R[j]=R[i];

j=i/4;

**while** (j>0 && R[j]>R[i]) //调小堆

{R[i]=R[j]; i=j; j=i/4; }

R[i]=R[0];

}

**else** //插入元素大于双亲，调大堆

{i=n; j=i/4;

**while** (j>0 && R[j]<R[i])

{R[i]=R[j]; i=j; j=i/4; }

R[i]=R[0];

}

}

**else** //插入元素在奇数层，是最小层

{i=n/2;

**if** (R[0].key>R[i].key) //插入元素大于双亲，先与双亲交换，然后调大堆

{R[j]=R[i];

j=i/4;

**while** (j>0 && R[j]<R[i]) //调大堆

{R[i]=R[j]; i=j; j=i/4; }

R[i]=R[0];

}

**else** //插入元素小于双亲，调小堆

{i=n; j=i/4;

**while** (j>0 && R[j]>R[i])

{R[i]=R[j]; i=j; j=i/4; }

R[i]=R[0];

}

}

}//MinMaxHeapIns

17.**void** BiInsertSort(RecType R[]，**int** n)

{//二路插入排序的算法

**int** d[n+1]; //辅助存储

 d[1]=R[1];first=1;final=1;

**for**(i=2;i<=n;i++)

  { **if**(R[i].key>=d[1].key) //插入后部

   {  low=1;high=final;

**while** (low<=high) //折半查找插入位置

{ m=(low+high)/2;

**if**(R[i].key< d[m].key) high=m-1; **else** low=m+1;

}//**while**

**for** (j=final;j>=high+1;j--) d[j+1] = d[j];//移动元素

       d[high+1]=R[i]; final++; //插入有序位置

     }

**else** //插入前部

   { **if**(first==1){ first=n; d[n]=R[i];}

**else**{ low=first;high=n;

**while** (low<=high)

{ m=(low+high)/2;

**if**(R[i].key< d[m].key) high=m-1; **else** low=m+1;

}//**while**

**for** (j=first;j<=high;j++) d[j-1] = d[j]; //移动元素

       d[high]=R[i]; first--;

     }//**if**

}//**if**

  }//**for**

R[1] =d[fisrt];

**for**(i=first%n+1,j=2;i!=fisrt;i=i%n+1,j++) R[j] =d[i]; //将序列复制回去

}//BiInsertSort

18. 手工模拟排序过程略。

#define n 待排序记录的个数

**typedef** **struct**

{ **int**  key[d]; //关键字由d个分量组成

**int**  next; //静态链域

AnyType other; //记录的其它数据域

} SLRecType;

SLRecType R[n+1]; //R[1..n]存放n个记录

**typedef** **struct**

{ **int**  f,e; //队列的头、尾指针

} SLQueue;

SLQueue B[m] //用队列表示桶，共m个

**int**  RadixSort(SLRecType R[], **int** n)

{ //对R[1..n]进行基数排序，返回收集用的链头指针

**for**(i=1;i<n;i++) R[i].next=i+1;//将R[1..n]链成一个静态链表

R[n].next=-1; p=1; //将初始链表的终端结点指针置空, p指向链表的第一个结点

**for**(j=d-1;j>=0;j--) //进行d趟排序

{**for**(i=0;i<m;i++) { B[i].f=-1;B[i].e=-1;}//初始化桶

**while**(p!=-1) //按关键字的第j个分量进行分配

{ k=R[p].key[j]; //k为桶的序号

**if**(B[k].f==-1) B[k].f=p; //将R[p]链到桶头

**else** R[B[k].e].next=p; //将R[p]链到桶尾

B[k].e=p; //修改桶的尾指针,

p=R[p].next; //扫描下一个记录

}//**while**

i=0;

**while**(B[i].f==-1) i++; //找第一个非空的桶

t=B[i].e; p=B[i].f //p为收集链表的头指针，t为尾指针

**while**(i<m-1)

{i++;

**if**(B[i].f!=-1){ R[t].next=B[i].f; t=B[i].e; } //连接非空桶

}//**while**

R[t].next=-1; //本趟收集完毕，将链表的终端结点指针置空

}//**for**

**return**  p;

}//RadixSort

19. [题目分析]本题是基数排序的特殊情况，关键字只含一位数字的整数。

**typedef** **struct**

{ **int** key;

**int** next;

} SLRecType;

SLRecType R[N+1];

**typedef** **struct**

{ int,f,e;

} SLQueue;

SLQueue B[10];

**int** Radixsort(SLRecType R[],**int** n) //设各关键字已输入到R数组中

{**for** (i=1;i<n;i++ ) R[i].next=i+1;

R[n].next=-1; p=1; //-1表示静态链表结束

**for** (i=0;i<=9;i++);{ B[i].f=-1; B[i].e=-1;} //设置队头队尾指针初值

**while** (p!=-1) //一趟分配

{k=R[p].key; //取关键字

**if**(B[k].f==-1)B[k].f=p; //修改队头指针

**else** R[B[k].e].next=p;

B[k].e=p;

p=R[p].next; //下一记录

}

i=0; //一趟收集

**while** (B[i].f==-1) i++;

t=B[i].e; p=B[i]f;

**while** (i<9)

{i++;

**if** (B[i].f!=-1)

{ R[t].next=B[i].f;t=B[i].e;} }

R[t].next=-1;

**return**  p; //返回第一个记录指针

}

[算法讨论]若关键字含d位，则要进行d趟分配和d趟收集。关键字最好放入字符数组，以便取关键字的某位。

20. **void** Adjust(**int** T[],**int** s)

{ //选得最小关键字记录后，从叶到根调整败者树，选下一个最小关键字

//沿从叶子结点R[s]到根结点T[0]的路径调整败者树

t=(s+k)/2; //T[t]是R[s]的双亲结点

**while**(t>0)

{ **if**(R[s].key>R[T[t]].key) s<-->T[t]; //s指示新的胜者

t=t/2;

}//**while**

T[0]=s;

}//Adjust

**void** CreateLoserTree(**int** T[])

{ //建立败者树，已知R[0]到R[k-1]为完全二叉树T的叶子结点，存有k个关键字，沿

//从叶子到根的k条路径将T调整为败者树

R[k].key=MINKEY; //MINKEY是最小关键字

**for**(i=0;i<k;i++) T[i]=k; //设置T中“败者”的初值

//依次从R[k-1],R[k-2],…,R[0]出发调整败者

**for**(i=k-1;k>=0;i--) Adjust(T,i);

}//CreateLoserTree

21、[题目分析]利用快速排序思想解决。由于要求“对每粒砾石的颜色只能看一次”，设3个指针i，j和k，分别指向红色、白色砾石的后一位置和待处理的当前元素。从k=n开始，从右向左搜索，若该元素是兰色，则元素不动，指针左移（即k-1）；若当前元素是红色砾石，分i>=j（这时尚没有白色砾石）和i<j两种情况。前一情况执行第i个元素和第k个元素交换，之后i+1；后一情况，i所指的元素已处理过（白色），j所指的元素尚未处理，应先将i和j所指元素交换，再将i和k所指元素交换。对当前元素是白色砾石的情况，也可类似处理。

为方便处理，将三种砾石的颜色用整数1、2和3表示。

**void** QkSort(rectype r[],**int** n)

// r为含有n个元素的线性表，元素是具有红、白和兰色的砾石，用顺序存储结构存储，

//本算法对其排序，使所有红色砾石在前，白色居中，兰色在最后。

{**int** i=1,j=1,k=n,temp;

**while** (k!=j)

{**while** (r[k].key==3) k--;// 当前元素是兰色砾石，指针左移

**if** (r[k].key==1) // 当前元素是红色砾石

**if** (i>=j){temp=r[k];r[k]=r[i];r[i]=temp; i++;}

//左侧只有红色砾石，交换r[k]和r[i]

**else** {temp=r[j];r[j]=r[i];r[i]=temp; j++;

//左侧已有红色和白色砾石，先交换白色砾石到位

temp=r[k];r[k]=r[i];r[i]=temp; i++;

//白色砾石（i所指）和待定砾石（j所指）

} //再交换r[k]和r[i]，使红色砾石入位。

**if** (r[k].key==2)

**if** (i<=j) { temp=r[k];r[k]=r[j];r[j]=temp; j++;}

// 左侧已有白色砾石，交换r[k]和r[j]

**else** { temp=r[k];r[k]=r[i];r[i]=temp; j=i+1;}

//i、j分别指向红、白色砾石的后一位置

}//**while**

**if** (r[k]==2) j++; /\* 处理最后一粒砾石

**else** **if** (r[k]==1) { temp=r[j];r[j]=r[i];r[i]=temp; i++; j++; }

//最后红、白、兰色砾石的个数分别为: i-1;j-i;n-j+1

}//结束QkSor算法

[算法讨论]若将j（上面指向白色）看作工作指针，将r[1..j-1]作为红色，r[j..k-1]为白色，r[k..n]为兰色。从j=1开始查看，若r[j]为白色，则j=j+1；若r[j]为红色，则交换r[j]与r[i]，且j=j+1，i=i+1；若r[j]为兰色，则交换r[j]与r[k];k=k-1。算法进行到j>k为止。

算法片段如下：

**int** i=1,j=1,k=n;

**while**(j<=k)

**if** (r[j]==1) //当前元素是红色

{temp=r[i]; r[i]=r[j]; r[j]=temp; i++;j++; }

**else** **if** (r[j]==2) j++; //当前元素是白色

**else** //(r[j]==3 当前元素是兰色

{temp=r[j]; r[j]=r[k]; r[k]=temp; k--; }

对比两种算法，可以看出，正确选择变量（指针）的重要性。

22、[题目分析]根据定义，DEAP是一棵完全二叉树，树根不包含元素，其左子树是一小堆（MINHEAP，下称小堆），其右子树是一大堆（MAXHEAP，下称大堆），故左右子树可分别用一维数组l[]和r[]存储，用m和n分别表示当前两完全二叉树的结点数，左右子树的高度差至多为1，且左子树的高度始终大于等于右子树的高度。

我们再分析插入情况：当均为空二叉树或满二叉树（m=2h-1）时，应在小堆插入；小堆满（二叉树）后在大堆插入。即当m>=n且m<>2h-1且⎣log2m⎦-⎣log2n⎦<=1在小堆插入，否则在大堆插入。

最后分析调堆情况：在小堆m处插入结点x后，若x的值不大于大堆的m/2结点的值，则在小堆调堆；否则，结点x与大堆的m/2结点交换，然后进行大堆调堆。在大堆n处插入结点x后，若x不小于小堆的n结点，则在大堆调堆；否则，结点x与小堆的n结点交换，然后进行小堆调堆。

1. 在DEAP中插入结点4后的结果如图：

|  |
| --- |
| 15 |

|  |
| --- |
|  |

|  |
| --- |
| 4 |

|  |
| --- |
| 10 |

|  |
| --- |
| 9 |

|  |
| --- |
| 30 |

|  |
| --- |
| 20 |

|  |
| --- |
| 19 |

|  |
| --- |
| 8 |

|  |
| --- |
| 25 |

|  |
| --- |
| 40 |

|  |
| --- |
| 45 |

|  |
| --- |
| 5 |

4先插入到大堆，因为4小于小堆中

对应位置的19，所以和19交换。交换后只需

调整小堆，从叶子到根结点。这时，大堆不需

调整，因为插入小堆19时，要求19必须小于

对应大堆双亲位置的25，否则，要进行交换。

**void** InsertDEAP(**int** l[],r[],m,n,x)

//在DEAP中插入元素x,l[]是小堆，r[]是大堆，m和n分别是两堆的元素个数，x是待插入元素。

{**if** (m>=n && m<>2h-1 && ⎣log2m⎦-⎣log2n⎦<=1)// 在小堆插入x,h是二叉树的高度

{m++; //m增1

**if** (x>r[m/2]) //若x大于大堆中相应结点的双亲，则进行交换

{l[m]=r[m/2];

c=m/2; f=c/2;

**while** (f>0 && r[f]<x) //调大堆

{r[c]=r[f]; c=f; f=c/2;}

r[c]=x;

} //结束调大堆

**else** //调小堆

{c=m; f=c/2;

**while** (f>0 && l[f]>x)

{l[c]=l[f]; c=f; f=c/2;}

l[c]=x;

}

**else** //在大堆插入x

{n++; //n增1

**if** (x<l[n]) //若x小于小堆中相应结点，则进行交换

{r[n]=l[n];

c=n; f=c/2;

**while** (f>0 && l[f]>x) //调小堆

{l[c]=l[f]; c=f; f=c/2;}

l[c]=x;

} //结束调小堆

**else** //调大堆

{c=n; f=c/2;

**while** (f>0 && r[f]<x)

{r[c]=r[f]; c=f; f=c/2;}

r[c]=x;

}

}//结束InsertDEAP算法