排序：  
升序和降序：

稳定排序和非稳定排序

稳定排序，相等的数据排序完成之后，其顺序保持不变。

非稳定排序，相等的数据排序完成之后，其顺序可能发生改变。

区分稳定排序和非稳定排序的原因是：有两个排序关键字的时候，稳定排序可以让第一个关键字排序的结果服务于第二个关键字的排序。

例如，每次考试的排名结果，分数高的排名靠前，若两个分数相同可以按上一次考试的名次先后进行排序。

内排序和外排序：

三种基本内排算法：

1、插入排序；

2、选择排序；

3、交换排序；

上述三种基本排序算法的改良算法：

1、希尔排序；

2、堆排序；

3、快速排序；

4、归并排序；

**一、直接插入排序**

0 1 2 3 4 5 6 7 8 9 10

void insertSort(int \*data, int count){

int firstIndex;

int j, j;

int tmp;

for(firstIndex = 1; firstIndex < count; firstIndex++){

tmp = data[firstIndex];

for(i = 0; i < firstIndex; i++){

if(data[i] > tmp){

break;

}

}

for(j = firstIndex; j > i; j--){

data[j] = data[j-1];

}

data[i] = tmp;

}

}

4 8 2 0 9 5 7 1 5 6 3

|  |
| --- |
| (4) (8 2 0 9 5 7 1 5 6 3)  (4 8) (2 0 9 5 7 1 5 6 3)  (2 4 8) (0 9 5 7 1 5 6 3)  (0 2 4 8) (9 5 7 1 5 6 3)  (0 2 4 8 9) (5 7 1 5 6 3)  (0 2 4 5 8 9) (7 1 5 6 3)  (0 2 4 5 5 8 9) (7 1 6 3)  (0 2 4 5 5 7 8 9) (1 6 3)  (0 1 2 4 5 5 7 8 9) (6 3)  (0 1 2 4 5 5 6 7 8 9) (3)  (0 1 2 3 4 5 5 6 7 8 9) |

插入排序是一种稳定排序。

时间复杂度的基本运算如果按比较和移动相同的话，则，

时间复杂度固定为O（N\*N-1/2）→O(N^2).

最优情况和最差时间情况相同：

最优情况（完全顺序），则，比较次数最多，移动次数为0；

最差情况（完全逆序），则，比较次数最少，移动次数最多，

**二、选择排序**

4 7 2 5 9 1 0 5 8 6 3

算法思想：每次找到一个最小的数，与未排序列的第一个元素交换，总共需要查找count-1次，因为最后一个最小的一定是整个数组中最大的数。

void chooseSort(int \*data, int count){

int index;

int i;

int minIndex;

int tmp;

for(i = 0; i < count -1; i++){

for(minIndex=index=i; index < count; index++){

if(data[index] < data[minIndex]){

minIndex = index;

}

}

if(minIndex != i){

tmp = data[index];

data[index] = data[minIndex];

data[minIndex] = tmp;

}

}

}

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

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

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

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

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

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

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

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

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

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

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

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

选择排序是非稳定排序。

时间复杂度：O(n(n-1)/2) → O(n^2)

最优情况和最差情况（完全顺序和某种顺序）

但是，两者的时间复杂度仅仅在交换次数。

**三、直接交换排序（冒泡排序）**

算法思想：

每次从第一个数据开始进行比较，两两比较，若前者大于后者则交换，每一轮循环都会将当前最大的数交换到数组末尾，因此，不需要每次比较到最后一位数，只需要将前面乱序的数据进行比较即可，因此，内层循环条件是j < n–1-i。再者，如果内层循环发生交换则说明该序列还处于乱序状态，若内层循环一轮完成后并没有发生交换，则说明序列已经排序完成，则应该停止循环，因此，给出一个标志位表示是否发生交换，外层循环条件需要对内层循环是否发生交换作出判断，

SwapFlag = TRUE;

for(i = 0; i < n-1 && SwapFlag; i++){

for(SwapFlag = FALSE, j = 0; j < n – 1 – i; j++){

if(data[j] > data[j+1]){

tmp = data[j];

data[j] = data[j+1];

data[j+1] = tmp;

SwapFlag = TRUE;

}

}

}

6 9 2 0 8 1 7 2 3 5 4

6 2 0 8 1 7 2 3 5 4 9

2 0 6 1 7 2 3 5 4 8 9

0 2 1 6 2 3 5 4 7 8 9

0 1 2 2 3 5 4 6 7 8 9

0 1 2 2 3 4 5 6 7 8 9

**四、希尔排序（直接插入排序的改良排序）**

第一次每两个一组，进行直接插入排序。

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

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

| |

2 | 2 |

1 | 8 |

1 5

…… 下标步长：10

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

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

| | | |

2 | 2 | 2 | 3 |

1 2 8 8

... ... 步长为5 = 10 / 2；

... …

直到步长为1，完成最后一次直接插入排序，排序完成。

在希尔排序的过程中，屡次使用直接插入排序算法，但是，又与以前的直接插入排序的环境有所不同。

void ShellInnerSort(int \*data, int count, int step, int start){

int first;

int i;

int j;

int t;

for(i = start + step; i < count; i+= step){

first = data[i];

for(j = start; j < i && data[j] <= first; j += step)

;

for(t = i; t > j; t -= step){

data[t] = data[t-step];

}

data[t] = first;

}

}

void ShellSort(int \*data, int count){

int step;

for(step = count/2; step > 0; step /= 2){

for(start = 0; start < step; start++){

ShellInnerSort(data, count, step, start);

}

}

}

Donald Shell提出了一种冲破二次时间屏障的算法Shellsort（希尔排序），在希尔排序中希尔给出了一组增量序列：ht = N / 2, h[k] = h[k+1] / 2，即{N/2, (N / 2)/2, ..., 1}，这个序列就叫做希尔增量。这个是编写希尔排序时最常用的序列，但却不是最好的。其余的增量序列还有Hibbard：{1, 3, ..., 2^k-1}，Sedgewick：{1, 5, 19, 41, 109...}该序列中的项或者是9\*4^i - 9\*2^i + 1或者是4^i - 3\*2^i + 1。使用不同的增量对希尔排序的时间复杂度的改进将不一样，甚至一点小的改变都将引起算法性能剧烈的改变。

Hibbard：{1, 3, ..., 2^k-1}；对于任意大于0的正整数n，如何计算得到t < n，且t为2^m – 1

20 => 0000 0000 0001 0100

void ShellSort(int \*data, int count){

int step = count;

int start = 0;

step |= step >> 1;

step |= step >> 2;

step |= step >> 4;

step |= step >> 8;

step |= step >> 16;

for(step = step >> 1; step > 0; step >>= 1){

for(start = 0; start < step; start++){

ShellInnerSort(data, count, step, start);

}

}

}

t = n;

t |= t >> 1;

t |= t >> 2;

t |= t >> 4;

t |= t >> 8;

t |= t >> 16;

t >> 1 就是不大于n的最小2^m – 1

**五、堆排序（选择排序的改良排序）**

大根堆和小根堆：

这里的堆是一个二叉树，而且是完全二叉树。根据完全二叉树的性质可知，一个一维数组就是一颗完全二叉树。

而且，大根堆满足如下要求：

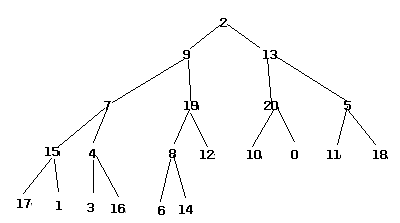
大根堆中的任意一个节点，若存在子节点，则，任何子节点的值，都小于其父节点的值。

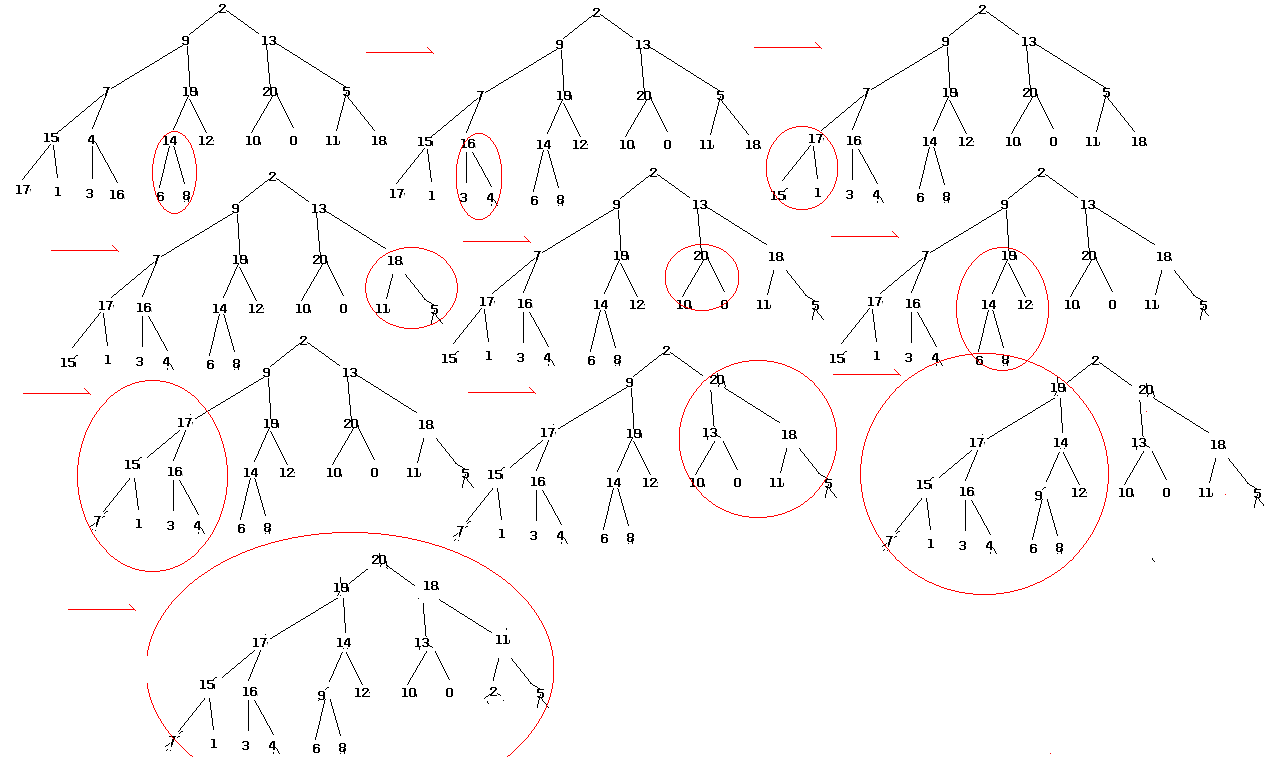
有如下数据：

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

2 9 13 7 19 20 5 15 4 8 12 10 0 11 18 17 1 3 16 6 14

生成一颗完全二叉树为：

大根堆的构建过程：

从最后一棵子树开始调整，每次使得子树成为大根堆。

结论：

有n个节点的堆，若父节点的下标为：i

则其左孩子的下标为：2\*i+1 右孩子的下标为：2\*i+2

最后一个非叶子节点的下标：n/2 – 1

生成大根堆之后，将最后一个叶子节点与根结点进行交换，之后进行只针对根结点的调整，调整完成后完成排序。

void adjustHeap(int \*data, int count, int root){

int maxIndex;//最大元素下标

int childMaxIndex;//孩子中最大者下标

int rightIndex;//右孩子下标

int leftIndex;//左孩子下标

int tmp;

while(root <= count/2-1){//判断该数是否为非叶子节点

leftIndex = 2\*root + 1;

rightIndex = leftIndex + 1;

//childMaxIndex = 是否只有左孩子节点 ？左孩子下标 ：（右孩子元素>左孩子元素 ？ 右孩子下标：左孩子下标）

childMaxIndex = rightIndex >= count ? leftIndex : (data[rightIndex] > data[leftIndex] ? rightIndex : leftIndex);

//求得孩子中最大者下标

//maxIndex = data[root] > 孩子中最大者下标 ？ root ：孩子中最大者下标

maxIndex = data[root] > data[childMaxIndex] ? root: childMaxIndex;//求得树中最大元素下标

if(maxIndex == root){

return ;

}

tmp = data[root];

data[root] = data[maxIndex];

data[maxIndex] = tmp;

root = maxIndex;

}

}

堆排序的时间复杂度分析：O(NlogN)

void heapSort(int \*data, int count){

int root;

int tmp;

for(root = count/2 - 1; root > 0; root--){//这里并没有完全生成大根堆，而是除了根结点未调整外都调整了，

adjustHeap(data, count, root);

}

for(; count > 0; count--){

adjustHeap(data, count, 0);//只针对根结点的调整

//将根结点与最后一个叶子节点进行交换

tmp = data[0];

data[0] = data[count-1];

data[count-1] = tmp;

}

}

**六、快速排序**

某一趟排序的最终目的：以第一个数据为标杆，将小于它的放在前面，大于他的放到后面。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | tmp |
| 5 | 8 | 1 | 2 | 9 | 0 | 9 | 3 | 6 | 4 | 7 |  |
| i |  |  |  |  |  |  |  |  |  | j |  |
|  | 8 | 1 | 2 | 9 | 0 | 9 | 3 | 6 | 4 | 7 | 5 |
| i |  |  |  |  |  |  |  |  |  | j |  |
|  | 8 | 1 | 2 | 9 | 0 | 9 | 3 | 6 | 4 | 7 | 5 |
| i |  |  |  |  |  |  |  |  | j |  |  |
| 4 | 8 | 1 | 2 | 9 | 0 | 9 | 3 | 6 |  | 7 | 5 |
| i |  |  |  |  |  |  |  |  | j |  |  |
| 4 |  | 1 | 2 | 9 | 0 | 9 | 3 | 6 | 8 | 7 | 5 |
|  | i |  |  |  |  |  |  |  | j |  |  |
| 4 |  | 1 | 2 | 9 | 0 | 9 | 3 | 6 | 8 | 7 | 5 |
|  | i |  |  |  |  |  |  | j |  |  |  |
| 4 |  | 1 | 2 | 9 | 0 | 9 | 3 | 6 | 8 | 7 | 5 |
|  | i |  |  |  |  |  | j |  |  |  |  |
| 4 | 3 | 1 | 2 | 9 | 0 | 9 |  | 6 | 8 | 7 | 5 |
|  |  | i |  |  |  |  | j |  |  |  |  |
| 4 | 3 | 1 | 2 | 9 | 0 | 9 |  | 6 | 8 | 7 | 5 |
|  |  |  | i |  |  |  | j |  |  |  |  |
| 4 | 3 | 1 | 2 | 9 | 0 | 9 |  | 6 | 8 | 7 | 5 |
|  |  |  |  | i |  |  | j |  |  |  |  |
| 4 | 3 | 1 | 2 |  | 0 | 9 | 9 | 6 | 8 | 7 | 5 |
|  |  |  |  | i |  | j |  |  |  |  |  |
| 4 | 3 | 1 | 2 |  | 0 | 9 | 9 | 6 | 8 | 7 | 5 |
|  |  |  |  | i | j |  |  |  |  |  |  |
| 4 | 3 | 1 | 2 | 0 | 5 | 9 | 9 | 6 | 8 | 7 | 5 |
|  |  |  |  |  | i j |  |  |  |  |  |  |

**七、归并排序**

int onceQuickSort(int \*data, int start, int end){

int tmp = data[start];

while(start < end){

while(start < end && data[end] >= tmp){

--end;

}

if(start < end){

data[start] = data[end];

}

while(start < end && data[start] <= tmp){

++start;

}

if(start < end){

data[end] = data[start];

}

}

data[start] = tmp;

return start;

}

void innerQuickSort(int \*data, int start, int end){

int middle;

if(start >= end){

return ;

}

middle = onceQuickSort(data, start, end);

innerQuickSort(data, start, middle);

innerQuickSort(data, middle, start);

}

void quickSort(int \*data, int count){

innerQuickSort(data, 0, count-1);

}

将两个已经排序的序列，归并成一个排序序列。

归并算法：

有两组数据，各自已经排序，且性质相同；将这两组数据归并成一组数据。

void merge(int \*data1, int count1, int \*data2, int count2, int result);

1: 2 6 9 10

while(i < count1 & j < count2){

if(data1[i] < data2[j]){

result[t++] = data1[i++];

}else{

result[t++] = data2[j++];

}

}

while(i < count1){

result[t++] = data1[i++];

}

while(j < count2){

result[t++] = data2[j++];

}

2: 3 7 19 20 22

2 3 6 7 9 10 19 20 22

1: 1 2 3 4 5

2: 13 15

1 2 3 4 5 13 15