# 时间复杂度

**时间复杂度为O(n^2)的排序算法**

* 冒泡排序
* 选择排序
* 插入排序（基本有序情况下，n-1）
* 希尔排序（略优于O(n^2)，但又低于O(nlogn)，姑且归入本类）

**时间复杂度为O(nlogn)的排序算法**

* 快速排序, 快速排序法的时间复杂度**不一定**优于冒泡排序法
* 归并排序
* 堆排序，堆排序每一趟排序过程中，**不是都会有**一个元素被放置在最终位置上3

递归：o（n2）

**折半查找时间复杂度为O(log2n)**



**选堆快希不稳，选堆归基不变。**

# 简单排序

## 冒泡排序

原理：比较两个相邻的元素，将值大的元素交换至右端。

N个数字要排序完成，总共进行N-1趟排序，每i趟的排序次数为(N-i)次，所以可以用双重循环语句，外层控制循环多少趟，内层控制每一趟的循环次数。

待排序数据：7, 6, 9, 8, 5,1

第一轮排序过程：指针先指向7，7和6比较，6<7，交换6和7的位置，结果为：6,7,9,8,5,1

　　　　　　　　指针指向第二个元素7，7和9比较，9>7，不用交换位置，结果仍为：6,7,9,8,5,1

　　　　　　　　指针指向第三个元素9，比较9和8，8<9，交换8和9的位置，结果为：6,7,8,9,5,1

　　　　　　　　指针指向第四个元素9，比较9和5，5<9，交换5和9，结果为：6,7,8,5,9,1

　　　　　　　　指针指向第五个元素9，比较9和1，1<9，交换1和9的位置，结果为6,7,8,5,1,9

第一轮排序结束后，最大的数字9被移到了最右边。

进行第二轮排序，过程同上，只是由于最大的9已经放在最右边了，因此不用在比较9了，少了一次比较，第二轮结束的结果为：6,7,5,1,8,9

第三轮结果：6,5,1,7,8,9

第四轮比较结果：5,1,6,7,8,9

第五轮比较结果：1,5,6,7,8,9

最终排序结果为：1,5,6,7,8,9，由上可知N个数据排序，需要进行N-1轮排序；第i轮排序需要的比较次数为N-i次。

　　　　for(int i=0;i<a.length-1;i++){//外层循环控制排序趟数

　　　　　　for(int j=0;j<a.length**-1-i**;j++){//内层循环控制每一趟排序多少次

　　　　　　　　if(arr[j]>arr[j+1]){ //前一个大于后一个

　　　　　　　　　　int temp=arr[j];

　　　　　　　　　　arr[j]=arr[j+1];

　　　　　　　　　　arr[j+1]=temp;

　　　　　　　　}

　　　　　　}

　　　　}

## 选择排序（最小数）

每一趟排序获得最小数的方法：for循环进行比较，定义一个第三个变量temp，首先前两个数比较，把较小的数放在temp中，然后用temp再去跟剩下的数据比较，如果出现比temp小的数据，就用它代替temp中原有的数据。

待比较数据：7, 6, 9, 8, 5,1

　　第一轮：此时指针指向第一个元素7，找出所有数据中最小的元素，即1，交换7和1的位置，排序后的数据为：1,6,9,8,5,7

　　第二轮：第一个元素已经为最小的元素，此时指针指向第二个元素6，找到6,9,8,5,7中最小的元素，即5，交换5和6的位置，排序后的结果为：1,5,9,8,6,7

　　第三轮：前两个元素为排好序的元素，此时指针指向第三个元素9，找到9,8,6,7中最小的元素，即6，交换6和9的位置，排序后的结果为：1,5,6,8,9,7

　　第四轮：前三个元素为排好序的元素，此时指针指向第四个元素8，找到8,9,7中最小的元素，即7，交换8和7的位置，排序后的结果为：1,5,6,7,9,8

　　第五轮：前四个元素为排好序的元素，此时指针指向第五个元素9，找到9,8中最小的元素，即8，交换9和8的位置，排序后的结果为：1,5,6,7,8,9

　　到此，全部排序完成。

    public static void selectionSort(int[] a) {

  for (int i = 0; i < **a.length-1;** i++) {//趟数

            int minIndex = i;//当前的第一个值

            // 找出最小值的下标

            for (int j = i + 1; j < **a.length**; j++) {

                if (a[j] < a[minIndex]) {

                    minIndex = j;

                }

            }

            // 将最小值放到未排序记录的第一个位置

            if ( i！= minIndex) {

                int tmp = a[i];

                a[i] = a[minIndex];

                a[minIndex] = tmp;

            }

        }

    }

## 插入排序（第二个元素）

整理桥牌

待比较数据：7, 6, 9, 8, 5,1

　　第一轮：**指针指向第二个元素**6，假设6左面的元素为有序的，将6抽离出来，形成7,\_,9,8,5,1，从7开始，6和7比较，发现7>6。将7右移，形成\_,7,9,8,5,1，6插入到7前面的空位，结果：6,7,9,8,5,1

　　第二轮：指针指向第三个元素9，此时其左面的元素6,7为有序的，将9抽离出来，形成6,7,\_,8,5,1，从7开始，依次与9比较，发现9左侧的元素都比9小，于是无需移动，把9放到空位中，结果仍为：6,7,9,8,5,1

　　第三轮：指针指向第四个元素8，此时其左面的元素6,7,9为有序的，将8抽离出来，形成6,7,9,\_,5,1，从9开始，依次与8比较，发现8<9，将9向后移，形成6,7,\_,9,5,1，8插入到空位中，结果为：6,7,8,9,5,1

　　第四轮：指针指向第五个元素5，此时其左面的元素6,7,8,9为有序的，将5抽离出来，形成6,7,8,9,\_,1，从9开始依次与5比较，发现5比其左侧所有元素都小，**5左侧元素全部向右移动**，形成\_,6,7,8,9,1，将5放入空位，结果5,6,7,8,9,1。

　　第五轮：同上，1被移到最左面，最后结果：1,5,6,7,8,9。

/\*\* \* 通过交换进行插入排序，借鉴冒泡排序 \* \* @param a \*/

public static void sort(int[] a) {

for (int i = 0; i < a.length - 1; i++) {

for (int j = i + 1; j > 0; j--) {

if (a[j] < a[j - 1]) {

int temp = a[j];

a[j] = a[j - 1];

a[j - 1] = temp; } } } }

public void doInsertSort(){

for(int index = 1; index< array.length; index++){//外层向右的index，即作为**比较对象的数据的index**

int temp = array[index];//用作比较的数据

**int leftindex = index-1;**

while(leftindex>=0 && array[leftindex]>temp){//当比到最左边或者遇到比temp小的数据时，结束循环

array[leftindex+1] = array[leftindex];//往右移位

leftindex--;/**/到了比他小的，所以下面加1**

}

array[**leftindex+1**] = temp;//把temp放到空位上

}

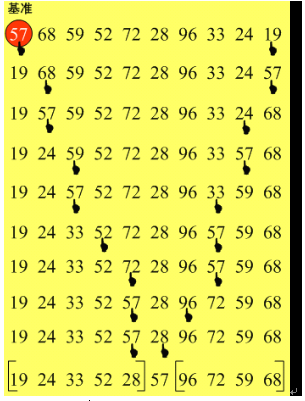
}

# 高级排序

## 快速排序

思路：选择一个基准元素,通常选择第一个元素或者最后一个元素,通过一趟扫描，将待排序列分成两部分,一部分比基准元素小,一部分大于等于基准元素,此时基准元素在其排好序后的正确位置,然后再用同样的**方法递归地排序划分**的两部分。

时间复杂度为n(第一步),然后递归的时间复杂度是logn(与遍历二叉树的操作类似最终快排的时间复杂度是nlogn(它是不稳定算法)。



public static void sort(int[] a, **int low, int hight)** {

//已经排完

if (low >= high) {**等于的时候结束**

return;

}

int left = low;

int right = high;

**int pivot** = a[left];// 核心;

while (left < right) {

//从后向前找到比基准小的元素

while (left < right && a[right] >= pivot){**//找到执行完**

right--;

}

a[left] = a[right];

//从前往后找到比基准大的元素

while (left < right && a[left] <= pivot){

left++;

}

a[right] = a[left];

}

// 放置基准值，准备**分治递归快排**

a[left] = pivot;

sort(a, low, left - 1);

sort(a, left + 1, high);

## 归并排序

基本思想——归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把带排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

public class MergeSort {

//两路归并算法，两个排好序的子序列合并为一个子序列

public void merge(int []a,int left,int mid,int right){

int []tmp=new int[a.length];**//辅助数组**

int p1=left,p2=mid+1,k=left;//p1、p2是检测指针，**k是存放指针**

while(p1<=mid && p2<=right){

if(a[p1]<=a[p2])

tmp[k++]=a[p1++];

else

tmp[k++]=a[p2++];

}

while(p1<=mid) tmp[k++]=a[p1++];//如果第一个序列未检测完，直接将后面所有元素加到合并的序列中

while(p2<=right) tmp[k++]=a[p2++];//同上

**//复制回原素组**

for (int i = left; i <=right; i++)

a[i]=tmp[i];

}

public void **mergeSort**(int [] a,int start,int end){

if(start<end){//当子序列中只有一个元素时结束递归

int mid=(start+end)/2;//划分子序列

mergeSort(a, start, mid);//对左侧子序列进行递归排序，不断进行拆分

mergeSort(a, mid+1, end);//对右侧子序列进行递归排序，不断进行拆分

merge(a, start, mid, end);//合并

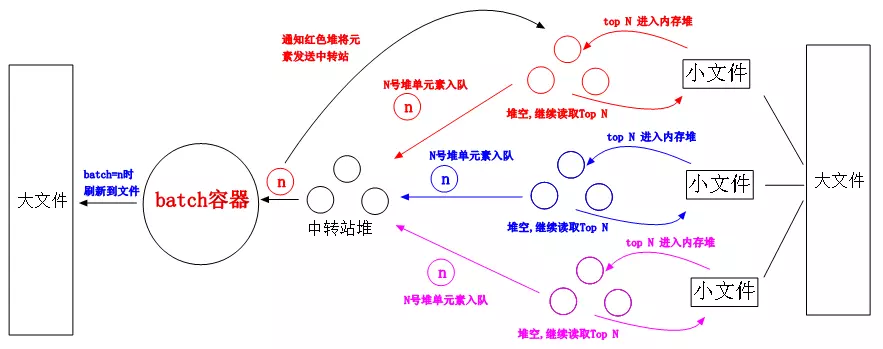
}

}

## 外排序

**我们可以将这个“二”扩大到M。**

将一个大文件分成M个小文件，每个小文件是有序的，然后对应在内存中我们开M个优先队列，每个队列从对应编号的文件中读取TopN条记录，然后我们从M路队列中各取一个数字进入中转站队列，并将该数字打上队列编号标记，当从中转站出来的最小数字就是我们最后要排序的数字之一，因为该数字打上了队列编号，所以方便我们通知对应的编号队列继续出数字进入中转站队列，可以看出中转站一直保存了M个记录，当中转站中的所有数字都出队完毕，则外排序结束。这考验的是我们的架构能力。



## 堆排序

堆树的定义如下：

（1）堆树是一颗**完全二叉树**；

（2）堆树中某个节点的值总是不大于其孩子节点的值；

（3）堆树中**每个节点的子树都是堆树。**

主要问题在于它使用了一个附加的数组，因此存储需求增加一倍。

**大顶堆：每个结点的值都大于或等于其左右孩子结点的值**

**大顶堆：arr[i] >= arr[2i+1] && arr[i] >= arr[2i+2]**

**普通树占用的内存空间比它们存储的数据要多。你必须为节点对象以及左/右子节点指针分配额外的内存。堆仅仅使用数组，且不使用指针**

再简单总结下堆排序的基本思路：

**a.将无需序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆;**

**b.将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端;**

**c.重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。**

public class HeapSort {

public static void main(String[] args) {

int[] array = new int[] { 2, 1, 4, 3, 6, 5, 8, 7 };

// 接下来就是排序的主体逻辑

sort(array);

System.out.println(**Arrays.toString**(array));

}}

public static void sort(int[] array) {

//**从最后一个非叶子节点开始，对于整棵树进行大根堆的调整,从下至上，从右至左调整结构**

for (int i = array.length / 2 - 1; **i >= 0**; i--) {

adjustHeap(array, i, array.length);

}  
 *// 上述逻辑，建堆结束*

// 下面，开始排序逻辑

for (int j = array.length - 1; **j > 0;** j--) {

swap(array, 0, j);//首个元素，和最后一个交换。

adjustHeap(array, 0, j);//*最后一个元素无需再考虑*

}

}

public static void adjustHeap(int[] array, int i, int length) {

// 先把当前元素取出来，因为当前元素可能要一直移动

int temp = array[i];

for (int k = 2 \* i + 1; k < length; k = 2 \* k + 1) {

//从i结点的**左子结点**开始，也就是2i+1处开始

if (k + 1 < length && array[k] < array[k + 1]) {

k++; //如果左子结点小于右子结点，k指向右子结点，找到最大的值

}

if (array[k] > temp) {

swap(array, i, k); // 发现子节点更大，则进行值的交换

i = k;

} else {

break;*// 如果不用交换，那么，就直接终止循环了*

}

}

**array[i] = temp;//将temp值放到最终的位置**

}

public static void swap(int[] arr, int a, int b) {

int temp = arr[a];

arr[a] = arr[b];

arr[b] = temp;

}

}

## 希尔算法

/\*\***希尔排序是加强版的插入排序**

**\*它通过比较相距一定间隔的元素来工作，各趟比较所用的距离随着算法的进行而减小，知道只比较相邻元素的最后一趟排序为止，也叫做缩减增量排序。**

\* 拿数组5, 2, 8, 9, 1, 3，4来说，数组长度为7，当increment为3时，数组分为两个序列

\* 5，2，8和9，1，3，4，第一次排序，9和5比较，1和2比较，3和8比较，4和比其**下标值小increment**的数组值相比较

\* 此例子是按照从大到小排列，所以大的会排在前面，第一次排序后数组为9, 2, 8, 5, 1, 3，4

\* 第一次后increment的值变为3/2=1,此时对数组**进行插入排序，**

\*实现数组从大到小排

\*/

public static void shellSort(int[] data) {

//每次将步长缩短为原来的一半，**取增量**

for (int increment = data.length / 2; increment > 0; increment /= 2){

for (int i = increment; i < data.length; i++) {

int temp = data[i];

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

if(temp > data[j - increment])//如想从小到大排只需修改这里{

data[j] = data[j - increment];}

else{

break;}}

data[j] = temp;} }}