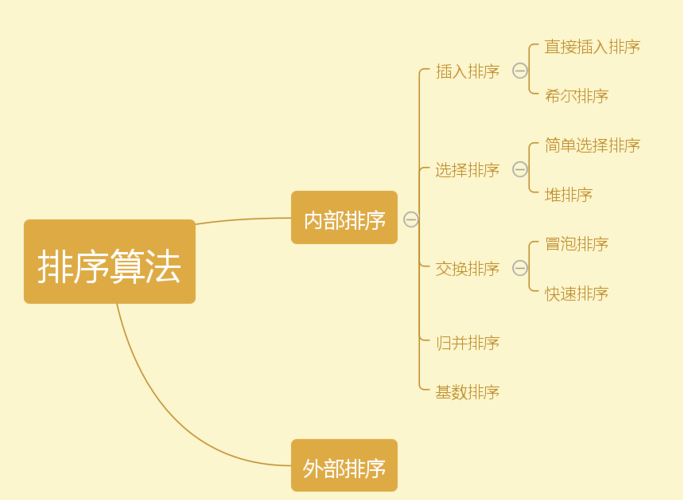
**八大排序总结**



常见的内部排序算法有：插入排序、希尔排序、选择排序、冒泡排序、归并排序、快速排序、堆排序、基数排序等。

**一、插入排序**

算法步骤：

1）将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

2）从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置。（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。）

****

public class InsertSort{

    public static <T extends Comparable<? super T>> void insertSort(T[] a){

        for(int p = 1; p < a.length; p++)

        {

            T tmp = a[p];//保存当前位置p的元素，其中[0,p-1]已经有序

            int j;

            for(j = p; j > 0 && tmp.compareTo(a[j-1]) < 0; j--)

            {

                    a[j] = a[j-1];//后移一位

            }

           a[j] = tmp;//插入到合适的位置

      }

    }

    //for test purpose

    public static void main(String[] args) {

       Integer[] arr = {34,8,64,51,32,21};

        insertSort(arr);

        for (Integer i : arr) {

           System.out.print(i + " ");

      }

    }

}

**算法二：希尔排序**

希尔排序，也称递减增量排序算法，是插入排序的一种更高效的改进版本。但希尔排序是非稳定排序算法。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

* 插入排序在对几乎已经排好序的数据操作时， 效率高， 即可以达到[线性排序](http://zh.wikipedia.org/w/index.php?title=%E7%BA%BF%E6%80%A7%E6%8E%92%E5%BA%8F&action=edit&redlink=1)的效率
* 但插入排序一般来说是低效的， 因为插入排序每次只能将数据移动一位

希尔排序的基本思想是：先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，待整个序列中的记录“基本有序”时，再对全体记录进行依次直接插入排序。

算法步骤：

1）选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

2）按增量序列个数k，对序列进行k 趟排序；

3）每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

public class ShellSort {

public static void shellSort(int[] data) {

int zjz;

if(data.length==1) {

return ;

}

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

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

for(int j=i;j>=h&&(data[j]<data[j-h]);j-=h) {

zjz=data[j];

data[j]=data[j-h];

data[j-h]=zjz;

}

}

}

}

public static void main(String[] args) {

int []data= {3,4,1,2,6,4,5,7,8,0,9,43,98,88,5,7,1001};

shellSort(data);

for(int i=0;i<data.length;++i) {

System.out.print(data[i]+" ");

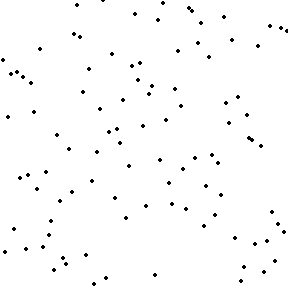
}

System.out.println();

}

}

**算法三：选择排序**

****

算法步骤：

1）首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置

2）再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。

3）重复第二步，直到所有元素均排序完毕。

//选择排序

public class SelectionSort {

   public static void main(String[] args) {

       int[] arr={1,3,2,45,65,33,12};

       System.out.println("交换之前：");

       for(int num:arr){

           System.out.print(num+" ");

      }

       //选择排序的优化

       for(int i = 0; i < arr.length - 1; i++) {// 做第i趟排序

           int k = i;

           for(int j = k + 1; j < arr.length; j++){// 选最小的记录

               if(arr[j] < arr[k]){

                   k = j; //记下目前找到的最小值所在的位置

              }

          }

           //在内层循环结束，也就是找到本轮循环的最小的数以后，再进行交换

           if(i != k){  //交换a[i]和a[k]

               int temp = arr[i];

               arr[i] = arr[k];

               arr[k] = temp;

          }

      }

       System.out.println();

       System.out.println("交换后：");

       for(int num:arr){

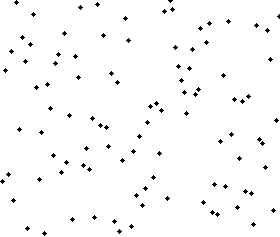
           System.out.print(num+" ");

      }

  }

}

**算法四：冒泡排序**



算法步骤：

1）比较相邻的元素。如果第一个比第二个大，就交换他们两个。

2）对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

3）针对所有的元素重复以上的步骤，除了最后一个。

4）持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

/\*\*

    \* 冒泡排序

    \*/

   public static void bubbleSort(int[] arr) {

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

           boolean flag = true;//设定一个标记，若为true，则表示此次循环没有进行交换，也就是待排序列已经有序，排序已然完成。

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

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

                   swap(arr,j,j+1);

                   flag = false;

              }

          }

           if (flag) {

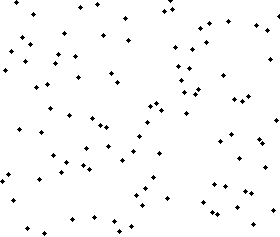
               break;

          }

      }

  }

**算法五：归并排序**



算法步骤：

1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列

2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置

3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置

4. 重复步骤3直到某一指针达到序列尾

5. 将另一序列剩下的所有元素直接复制到合并序列尾

//归并排序

public  class MergeSort {

static void metge(int []array,int p,int q,int r) {

  int a[]=new int[q-p+1];

  int cnt=0,zjz;

  int j=r+1,i=p;

  while(j<q+1||i<r+1) {

  if(j==q+1||(i<r+1&&array[i]<array[j])) {

  a[cnt++]=array[i++];

  }else {

  a[cnt++]=array[j++];

  }

  }

  for(i=0;i<a.length;i++) {

  array[i+p]=a[i];

  }

  }

static void metgeSort(int []array,int p,int q) {

  if(p<q) {

  int r=(p+q)/2;

  metgeSort(array,p,r);

  metgeSort(array,r+1,q);

  metge(array,p,q,r);

  }

}

public static void main(String[] args) {

int array[]= {3,4,6,1,2,7,9};

metgeSort(array,0,array.length-1);

for(int i:array) {

System.out.println(i);

}

}

}

**算法六：快速排序**



算法步骤：

1 从数列中挑出一个元素，称为 “基准”（pivot），

2 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。

3 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会退出，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

//快速排序一

public class QuickSore{

public static void quickSore(int[] data, int left, int right) {

int base;

if(right>left) {

base=sort(data,left,right);

quickSore(data,left,base-1);//小于基准

quickSore(data,base+1,right);//大于基准

}

}

public static int sort(int[] data, int left, int right) {

int zj=data[left];

int base=left,zjz;

data[left]=data[right];

data[right]=zj;

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

if(data[right]>data[i]) {

zjz=data[base];

data[base]=data[i];

data[i]=zjz;

base++;

}

}

zjz=data[base];

data[base]=data[right];

data[right]=zjz;

return base;

}

public static void main(String[] args) {

int []data= {3,4,1,2,6,4,5,7,8,0,9,43,98,88,5,7};

quickSore(data,0,data.length-1);

for(int i=0;i<data.length;++i) {

System.out.print(data[i]+" ");

}

System.out.println();

}

}

/\*

\* 可以用荷兰国旗问题来改进快速

\*/

public class ImproveQuickSort {

​

public static void main(String[] args) {

int arr[]={1,3,4,2,5,4,6};

quickSort(arr,0,arr.length-1);

for(int i=0;i<arr.length;++i) {

System.out.println(arr[i]);

}

}

​

public static void quickSort(int[] arr, int left, int right) {

​

if(right>left) {

int []p=sort(arr,left,right);

quickSort(arr,left,p[0]);

quickSort(arr,p[1],right);

}

}

​

private static int[] sort(int[] arr, int left, int right) {

int less=left,more=right-1;

int i=left;

while(more>=i) {

if(arr[i]>arr[right]) {

swap(arr,i,more--);

}

else {

if(arr[i]<arr[right]){

swap(arr,i,less++);

}

i++;

}

}

swap(arr,right,more+1);

return new int[]{less-1,more+1};

}

private static void swap(int[] arr, int i, int j) {

int zjz;

zjz=arr[j];

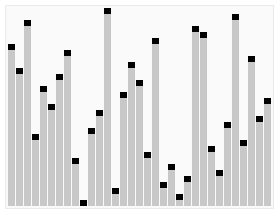
arr[j]=arr[i];

arr[i]=zjz;

}

}

**算法七：堆排序**



堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。堆排序的平均时间复杂度为Ο(nlogn) 。

算法步骤：

1）创建一个堆H[0..n-1]

2）把堆首（最大值）和堆尾互换

3）把堆的尺寸缩小1，并调用shift\_down(0),目的是把新的数组顶端数据调整到相应位置

4） 重复步骤2，直到堆的尺寸为1

/\*

\* 堆排序的细节和复杂度分析时间复杂度O(N\*logN)，额外空间复杂度O(logN)

\* 1，堆结构的heapInsert与heapify

\* 2堆结构的增大和减少

\*3，如果只是建立堆的过程，时间复杂度为O(N)

\*4优先级队列结构，就是堆

\*/

public class heapSort {

public static void main(String[] args) {

int arr[]={5555,9,7,4,2,196,6,8,7,5,4,3,343,234,222,333};

heapSort(arr);

for(int i=0;i<arr.length;++i) {

System.out.println(arr[i]);

}

}

public static void heapSort(int[] arr) {

if(arr==null||arr.length<2) {

return;

}

for(int i=0;i<arr.length;++i) {

heapInsert(arr,i);

}

int i=arr.length;

swap(arr,0,--i);

while(i>0) {

heapIfy(arr,i);

swap(arr,0,--i);

}

}

private static void heapIfy(int[] arr, int index) {

int i=0;

int left=2\*i+1;

while(left<index) {

int max=left+1<index&&arr[left+1]>arr[left]?left+1:left;

max=arr[max]>arr[i]?max:i;

if(max==i) {

break;

}

swap(arr,max,i);

i=max;

left=2\*i+1;

}

}

private static void heapInsert(int[] arr, int index) {

while(arr[index]>arr[(index-1)/2]) {

swap(arr,index,(index-1)/2);

index=(index-1)>>1;

}

}

private static void swap(int[] arr, int i, int j) {

int zjz;

zjz=arr[j];

arr[j]=arr[i];

arr[i]=zjz;

}

}

**算法八：基数排序**

基数排序(radix sort)又称桶排序（bucket sort），相对于常见的比较排序，基数排序是一种分配式排序，即通过将所有数字分配到应在的位置最后再覆盖到原数组完成排序的过程。它是一种稳定的排序算法，但有一定的局限性：   
　　1、关键字可分解；   
　　2、记录的关键字位数较少，如果密集更好；   
　　3、如果是数字时，最好是无符号的，否则将增加相应的映射复杂度，可先将其正负分开排序。   
  初始化：构造一个10\*n的二维数组，一个长度为n的数组用于存储每次位排序时每个桶子里有多少个元素。   
  循环操作：从低位开始（我们采用LSD的方式），将所有元素对应该位的数字存到相应的桶子里去（对应二维数组的那一列）。然后将所有桶子里的元素按照桶子标号从小到大取出，对于同一个桶子里的元素，先放进去的先取出，后放进去的后取出（保证排序稳定性）。这样原数组就按该位排序完毕了，继续下一位操作，直到最高位排序完成。

public class RadixSort implements Sortable {

​

   @Override

   public int[] sort(int[] array) {

       if (array == null) {

           return null;

      }

​

       int maxLength = maxLength(array);

​

       return sortCore(array, 0, maxLength);

  }

​

   private int[] sortCore(int[] array, int digit, int maxLength) {

       if (digit >= maxLength) {

           return array;

      }

​

       final int radix = 10; // 基数

       int arrayLength = array.length;

       int[] count = new int[radix];

       int[] bucket = new int[arrayLength];

​

       // 统计将数组中的数字分配到桶中后，各个桶中的数字个数

       for (int i = 0; i < arrayLength; i++) {

           count[getDigit(array[i], digit)]++;

      }

​

       // 将各个桶中的数字个数，转化成各个桶中最后一个数字的下标索引

       for (int i = 1; i < radix; i++) {

           count[i] = count[i] + count[i - 1];

      }

​

       // 将原数组中的数字分配给辅助数组 bucket

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

           int number = array[i];

           int d = getDigit(number, digit);

           bucket[count[d] - 1] = number;

           count[d]--;

      }

​

       return sortCore(bucket, digit + 1, maxLength);

  }

​

   /\*

    \* 一个数组中最大数字的位数

    \*/

   private int maxLength(int[] array) {

       int maxLength = 0;

       int arrayLength = array.length;

       for (int i = 0; i < arrayLength; i++) {

           int currentLength = length(array[i]);

           if (maxLength < currentLength) {

               maxLength = currentLength;

          }

      }

​

       return maxLength;

  }

​

   /\*

    \* 计算一个数字共有多少位

    \*/

   private int length(int number) {

       return String.valueOf(number).length();

  }

​

   /\*

    \* 获取 x 这个数的 d 位数上的数字

    \* 比如获取 123 的 0 位数,结果返回 3

    \*/

   private int getDigit(int x, int d) {

       int a[] = { 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000, 1000000000 };

       return ((x / a[d]) % 10);

  }

}

**总结**



关于时间复杂度：

(1)平方阶(O(n2))排序  
各类简单排序:直接插入、直接选择和冒泡排序；

(2)线性对数阶(O(nlog2n))排序  
　　快速排序、堆排序和归并排序；  
(3)O(n1+§))排序,§是介于0和1之间的常数。

       希尔排序  
(4)线性阶(O(n))排序  
基数排序，此外还有桶、箱排序。

 关于空间复杂度：

(1)辅助存储O(1)排序

直接插入、直接选择、  希尔排序、冒泡排序和堆排序；

(2)辅助存储O(nlog**2**n)排序

快速排序

(3)辅助存储O(n)排序

归并排序

(4)辅助存储O(n+rd)排序

基数排序

关于稳定性：

稳定的排序算法：冒泡排序、插入排序、归并排序和基数排序

不是稳定的排序算法：选择排序、快速排序、希尔排序、堆排序