排序

所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。排序算法，就是如何使得记录按照要求排列的方法。排序算法在很多领域得到相当地重视，尤其是在大量数据的处理方面。一个优秀的算法可以节省大量的资源。在各个领域中考虑到数据的各种限制和规范，要得到一个符合实际的优秀算法，得经过大量的推理和分析。

常见的排序算法及复杂度的比较

排序算法的评价标准

（1）时间复杂度：即从序列的初始状态到经过排序算法的变换移位等操作变到最终排序好的结果状态的过程所花费的时间度量。

（2）空间复杂度：就是从序列的初始状态经过排序移位变换的过程一直到最终的状态所花费的空间开销。

（3）使用场景：排序算法有很多，不同种类的排序算法适合不同种类的情景，可能有时候需要节省空间对时间要求没那么多，反之，有时候则是希望多考虑一些时间，对空间要求没那么高，总之一般都会必须从某一方面做出抉择。

（4）稳定性：稳定性是不管考虑时间和空间必须要考虑的问题，往往也是非常重要的影响选择的因素。（**稳定的算法在排序的过程中不会改变元素彼此的位置的相对次序，反之不稳定的排序算法经常会改变这个次序**）

算法的代码实现

1、什么是冒泡排序？

冒泡排序的英文Bubble Sort，是一种最基础的交换排序。之所以叫做冒泡排序，因为每一个元素都可以像小气泡一样，根据自身大小一点一点向数组的一侧移动。

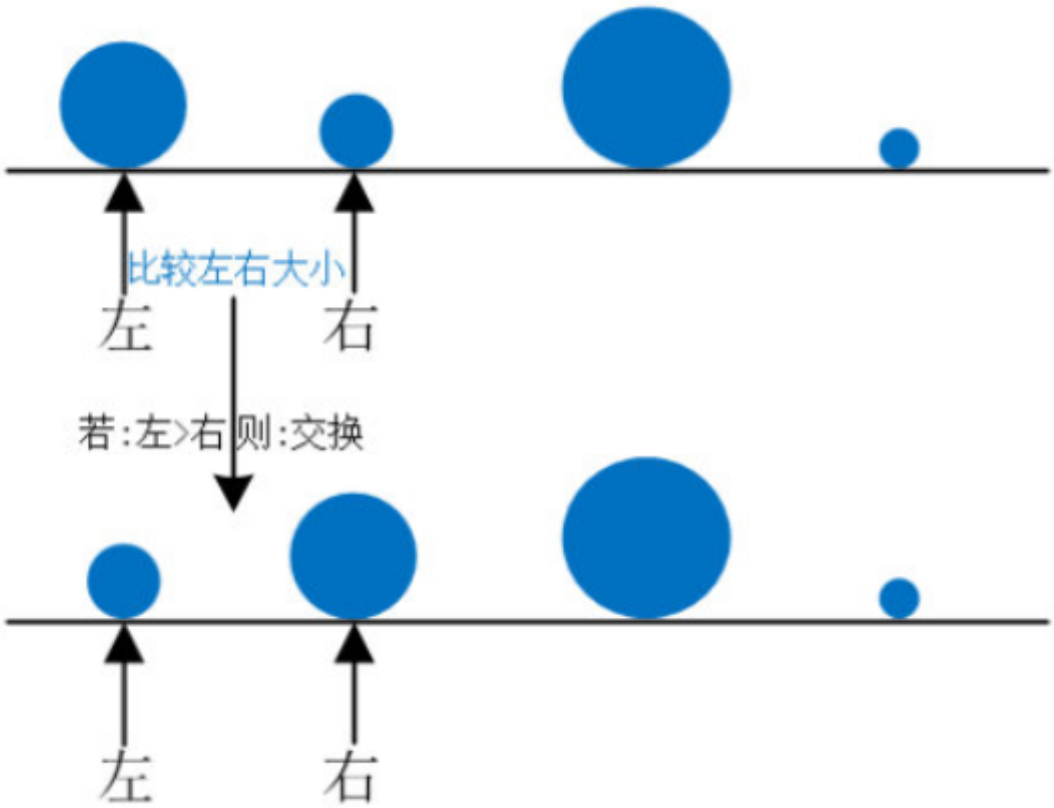
冒泡排序的原理：

每一趟只能确定将一个数归位。即第一趟只能确定将末位上的数归位，第二趟只能将倒数第 2 位上的数归位，依次类推下去。如果有 n 个数进行排序，只需将 n-1 个数归位，也就是要进行 n-1 趟操作。而 “每一趟 ” 都需要从第一位开始进行相邻的两个数的比较，将较大的数放后面，比较完毕之后向后挪一位继续比较下面两个相邻的两个数大小关系，重复此步骤，直到最后一个还没归位的数。

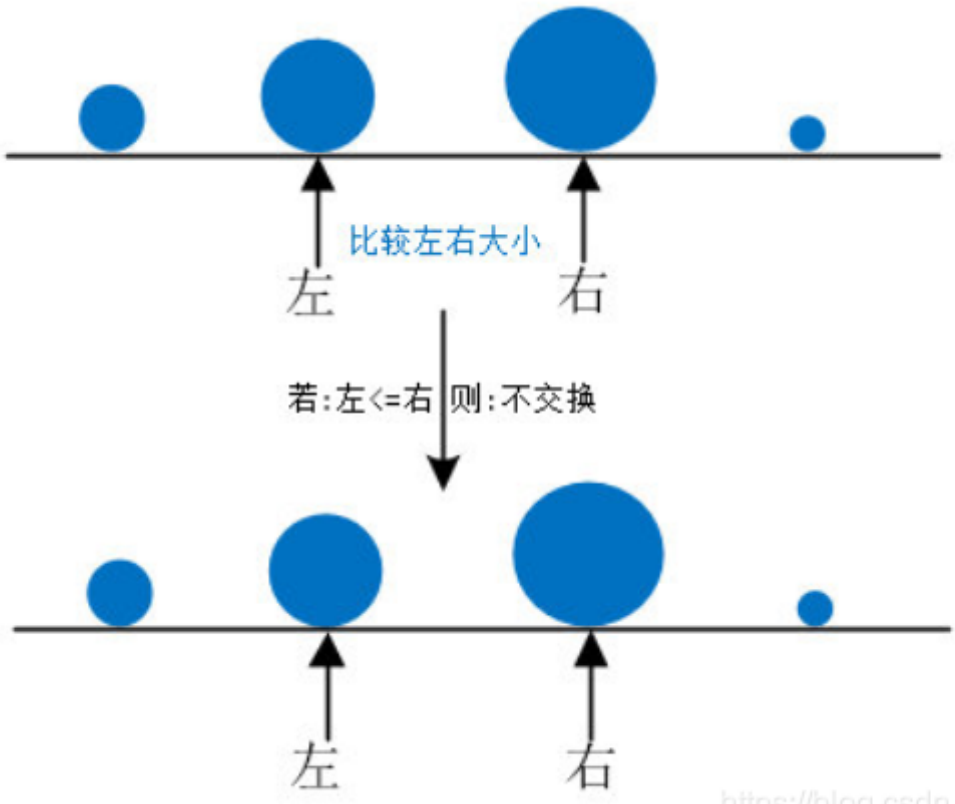
2、冒泡排序到底是如何排序的呢？

具体是如何移动的呢？这里参考了(帅地的冒泡排序)

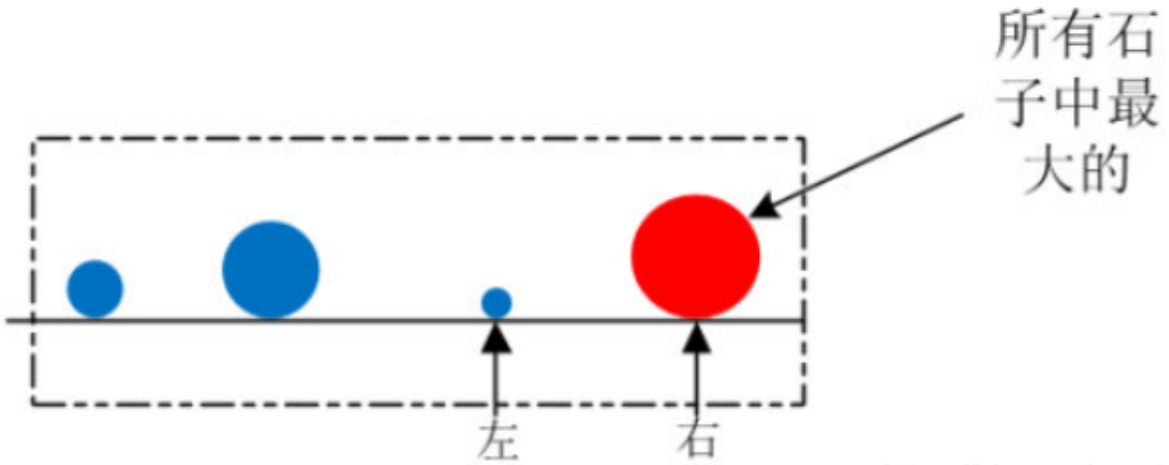
（1）起始时，左下标指向第一个石子，右下标指向第二个石子，然后比较



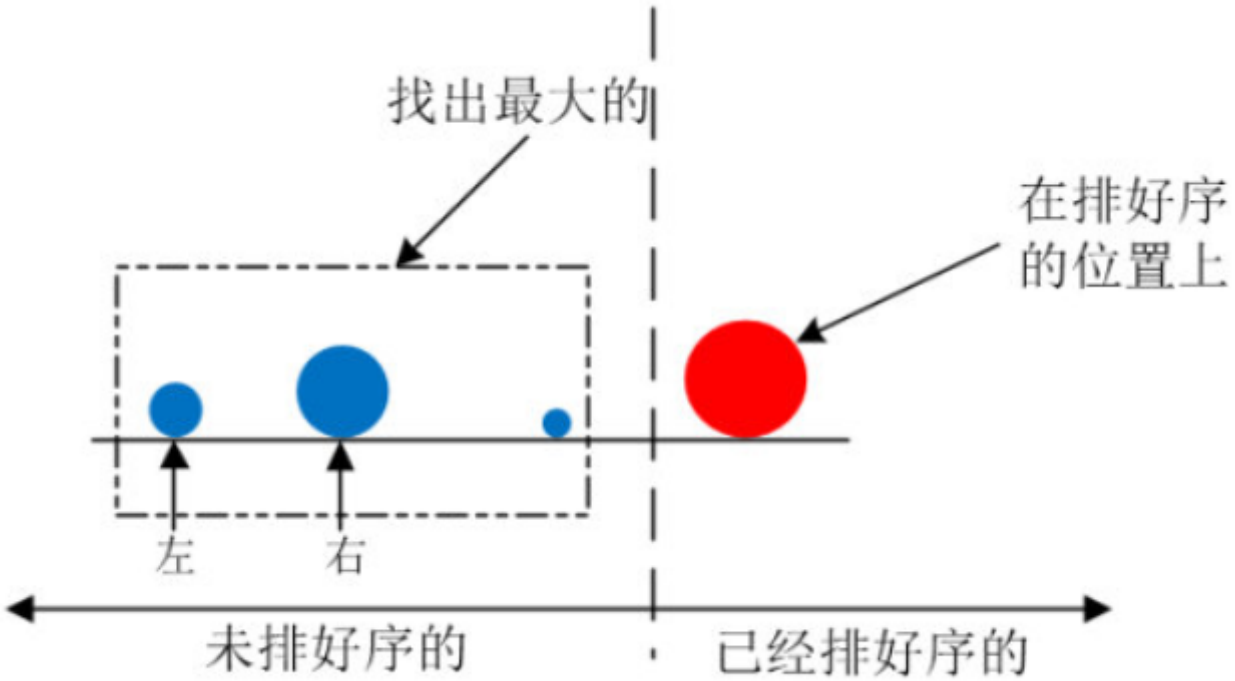
（2）然后左右下标同时向右移动，再次比较



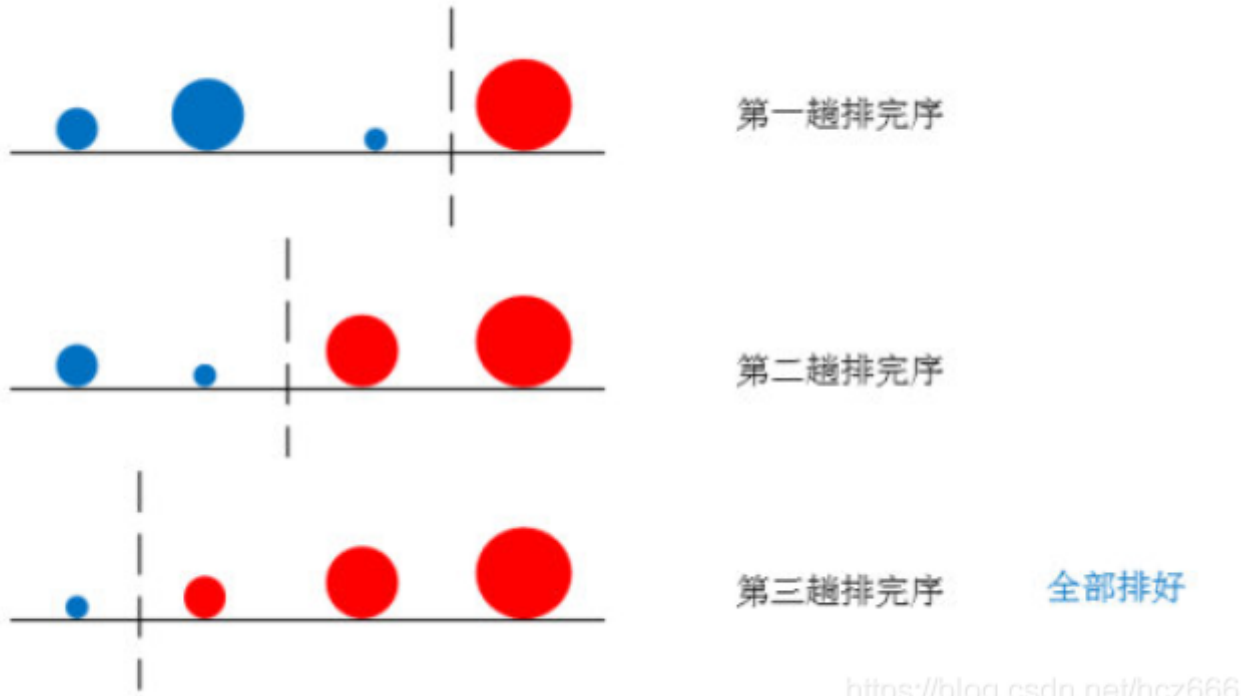
（3）第一趟结束之后，最大的石子就移动到了最右边



（4）接下来就从剩下 3 个没排好序的石子中继续选出最大的，规则和上面一样



（5）下面给出这 4 个石子完整的演示过程



3、时间复杂度

由上图可知，4 个石子的时候排完序需要 3 趟，第一趟需要比较3次，第二趟需要比较2次，第三趟需要比较1次，那一共比较了 3 + 2 + 1 次；

那如果有 n 个石子呢？

那就需要 (n-1) + (n-2) +…+2+1 次，这不就是一个等差数列吗，很显然：

根据复杂度的规则，去掉低阶项（也就是 n/2 ）,并去掉常数系数，那复杂度就是 **O(n^2)**了；

冒泡排序也是一种稳定排序，因为在两个数交换的时候，如果两个数相同，那么它们并不会因为算法中哪条语句相互交换位置。

4、冒泡排序代码原始版

//按照刚才那个动图进行对应

```java
// 优化第一版
public static void bubbleSort2(int[] arr, int len) {
    for (int i = 0; i < len; i++) {
        boolean isSorted = true;
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                isSorted = false;
            }
        }
        if (isSorted) {
            break;
        }
    }
}
```

疑问环节

小花：第一层循环是用来控制趟数，也就是 n 个数就要比较 n-1 趟；那么第二层循环能不能具体解答一下呢？

小明：第二层是控制第 i+1 趟（因为循环中 i 是从 0 开始的）的比较次数，那么 i+1 趟就是比较了 N-1-i 次

小花：还是不是很清楚，能不能用图形来跟我描述一下呀？

小明：当然可以呢，下面通过几张图来给你讲解一下。

小花：这个明白了，但是如果遇到下面这种情况呢？也就是执行到最后三轮的时候，发现整个数列已经是有序的了，可以按上面的代码执行的话算法还是仍然“兢兢业业”地继续执行第七轮、第八轮。这个可不可以优化一下呢？

小明：当然可以优化，如果我们能判断出数列已经有序，并且做出标记，剩下的几轮排序就可以不必执行，提早结束工作。

小花：那到底又怎么样优化上面的代码才能得到这种效果呢？

小明：其实也很简单，就是将上面代码做一点点小小改动即可，也就是利用布尔变量 isSorted作为标记。如果在本轮排序中，元素有交换，则说明数列无序；如果没有元素交换，说明数列已然有序，直接跳出大循环。

5、冒泡排序代码优化版

```java
// 优化第一版
public static void bubbleSort2(int[] arr, int len) {
    for (int i = 0; i < len; i++) {
        boolean isSorted = true;
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                isSorted = false;
            }
        }
        if (isSorted) {
            break;
        }
    }
}
```

小花：懂了，我还有一个问题。那如果数列中前半部分是无序的，后半部分是有序的呢？比如（3，4，2，1，5，6，7，8）这个数组，其实后面的许多元素已经是有序的了，但是每一轮还是白白比较了许多次呢？例如：

第一轮：

元素 3 和元素 4 比较，3 大于 4，所以位置不变

接着元素 4 和元素 2 比较，4 大于 2，所以 4 和 2 交换
接着元素 4 和元素 1 交换
再接着就发现后面其实就是有序数列了，但是还是要每一次两两相比，这样就白白比较了很多次了
第二轮：
元素 3 和元素 2 比较，3 大于 2，所以 3 和 2 交换
元素 3 和 1 比较，发现 3 大于 1，所以 3 和 1 交换
再接着就发现后面其实又是有序数列了，但是还是要每一次两两相比，这样也白白比较了很多次了
小明：对于上面你提出的问题，关键在于对这数列有序区的界定。如果按冒泡排序代码原始版来分析的话，有序区的长度和排序的轮数是相等的。比如第一轮排序过后的有序区长度是1，第二轮排序过后的有序区长度是2 ……。
但是呢，实际数列真正的有序区可能会大于这个长度，也就是你上面这个例子，第二轮中后面 5 个实际上都已经属于有序区了。因此后面的比较是没有意义的了。
我们可以这样做来避免这种情况：在每一轮排序的最后，记录一下最后一次元素交换的位置，那个位置也就是无序数列的边界，再往后就是有序区了。
6、冒泡排序代码升级版

```
public static int[] bubbleSort(int[] arr) {
if (arr == null || arr.length < 2) {
return arr;
}
//记录最后一次交换的位置
int lastExchangeIndex = 0;
//无序数列的边界，每次比较只需要比到这里为止
int sortBorder = arr.length - 1;
for (int i = 0; i < arr.length - 1; i++) {
     boolean isSorted  = true;//有序标记，每一轮的初始是true
     for (int j = 0; j < sortBorder; j++) {
         if (arr[j + 1] < arr[j]) {
             isSorted  = false;//有元素交换，所以不是有序，标记变为false
             int t = arr[j];
             arr[j] = arr[j+1];
             arr[j+1] = t;
             lastExchangeIndex = j;
         }
     }
    sortBorder = lastExchangeIndex
     //一趟下来是否发生位置交换，如果没有交换直接跳出大循环
     if(isSorted )
          break;
```

```
 }
 return arr;

}
```

2、**归并排序**：就是把序列递归划分成为一个个短序列，以其中只有1个元素的直接序列或者只有2个元素的序列作为短序列的递归出口，再将全部有序的短序列按照一定的规则进行排序为长序列。

工作原理：

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

public class Msort {

```
public static void Msort (int []arr) {
        int []temp = new int[arr.length];//在排序前，先建好一个长度等于原数组长度的临时数组，避免递归中频繁开辟空间
        sort(arr, 0, arr.length-1, temp);

}

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

        if(left>=right) return;
        int mid=(left+right)/2;
        sort(arr, left, mid, temp);  //左边递归排序
        sort(arr, mid+1, right, temp);//右边归并排序，使得右子序列有序
        merge(arr,left,mid,right,temp);//将两个有序子数组合并操作

}

private  static void merge(int[] arr, int left, int mid, int right, int[] temp) {
        // TODO Auto-generated method stub
        int i=left;  //左序列指针
        int j=mid+1;   //右序列指针
        int t=0;   //临时数组指针
```

```
        while(i<=mid && j<=right) {
                if(arr[i]<=arr[j]) {
                        temp[t++]=arr[i++];
                }else {
                        temp[t++]=arr[j++];
                }
        }
        while(i<=mid){//将左边剩余元素填充进temp中
        temp[t++] = arr[i++];
    }
    while(j<=right){//将右序列剩余元素填充进temp中
        temp[t++] = arr[j++];
    }
    t = 0;
    //将temp中的元素全部拷贝到原数组中
    while(left <= right){
        arr[left++] = temp[t++];
    }
}

```

}

3、快速排序：

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列

排序流程：

快速排序算法通过多次比较和交换来实现排序，其排序流程如下：

(1)首先设定一个分界值，通过该分界值将数组分成左右两部分。

(2)将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于或等于分界值，而右边部分中各元素都大于或等于分界值。

(3)然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理。

(4)重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

一般步骤

方法一：

1）设置两个变量i、j，排序开始的时候：i=0，j=N-1； [1]

2）以第一个数组元素作为关键数据，赋值给key，即key=A[0]； [1]

3）从j开始向前搜索，即由后开始向前搜索(j--)，找到第一个小于key的值A[j]，将A[j]和A[i]的值交换；[1]

4）从i开始向后搜索，即由前开始向后搜索(i++)，找到第一个大于key的A[i]，将A[i]和A[j]的值交换；[1]

5）重复第3、4步，直到i=j；(3,4步中，没找到符合条件的值，即3中A[j]不小于key,4中A[i]不大于key的时候改变j、i的值，使得j=j-1，i=i+1，直至找到为止。找到符合条件的值，进行交换的时候i， j指针位置不变。另外，i==j这一过程一定正好是i+或j-完成的时候，此时令循环结束）。

6）循环结束后将KEY与 i=j相等时的A[i]交换；

7）递归调用

public class quicksort {

```
public static void quickSort1(int[] arr) {
        if(arr==null || arr.length<=1) return;
        sort2(arr, 0, arr.length-1);

}

 static void sort2(int[] arr,int left,int right) {
        // TODO Auto-generated method stub
        int i=left;
        int j=right;
        int t;
        int temp;   //基准数
        if(left>right) return;
        temp=arr[left];
        while(i!=j) {

                ///顺序很重要，要先从右边开始找，因为是以左边为基准数
                while(arr[j]>=temp && j>i) {  // j>i控制j的范围
                        j--;
                }

                if(i<j) {
                        arr[i++]=arr[j];

                }
                while(arr[j]<=temp && j>i) {  // j>i控制j的范围
                        i++;
                }
```

```
                if(i<j) {
                        arr[j--]=arr[i];

                }

        }
        arr[i]=temp;

        sort2(arr,left,i-1);
        sort2(arr,i+1,right);
}

```

}

方法二：

1、选择一个基准数，从后遍历找到第一个小于基准数的位置J停下；从左向后遍历，找到第一个大于基准数的地方i停下，交换I两者；

2、j继续向前，找到第一个大于基准数的地方i停下，i继续向前，找到第一个大于基准数的地方i停下，交换i ,j对应的值，重复步骤，直到i=j,最后将基准数于i对应的值交换。

public class quicksort {

```
public static void quickSort1(int[] arr) {
        if(arr==null || arr.length<=1) return;
        sort2(arr, 0, arr.length-1);

}


private static void sort(int[] arr,int left,int right) {
        // TODO Auto-generated method stub
        int i=left;
        int j=right;
        int t;
        int temp;   //基准数
        if(left>right) return;
        temp=arr[left];
        while(i!=j) {
                ///顺序很重要，要先从右边开始找，因为是以左边为基准数
                while(arr[j]>=temp && j>i) {  // j>i控制j的范围
                        j--;
```

```
                }
                while(arr[i]<=temp && j>i) {  // j>i控制j的范围
                        i++;
                }
                if(i<j) {
                        t=arr[i];
                    arr[i]=arr[j];
                    arr[j]=t;
                }
        }
        //将基准数交换到两者相遇的位置
        arr[left]=arr[i];
        arr[i]=temp;
        //

        sort(arr,left,i-1);
        sort(arr,i+1,right);
}

```

}
