快速排序算法

笔记本：	常用算法		
创建时间：	2019/12/30 13:15	**更新时间：**	2019/12/30 13:18
作者：	AlinaWang		
标签：	排序算法		
URL：	https://www.cnblogs.com/dongkuo/p/4827281.html		

1. 算法描述

快速排序(quick-sort)与前面介绍的归并排序(merge-sort)（见算法基础——算法导论(1)）一样，使用了**分治**思想。下面是对一个一般的子数组A[p~r]进行快速排序的分治步骤：

① **分解：**数组A[p~r]被划分为两个子数组A[p~q]和A[q+1~r]，使得A[q]大于等于A[p~q]中的每个元素，且小于等于A[q+1~r]中的每个元素。（需要说明的是，我们允许A[p~q]和A[q+1~r]为空）

② **解决：**对子数组A[p~q]和A[q+1~r]递归的调用快速排序。

③ **合并：**因为子数组都是**原址排序**的，所以不需要合并操作，此时的A数组已经是排好序的。

ps：所谓原址排序是指：我们在对组进行排序的过程中 只有常数个元素被存储到数组外面。

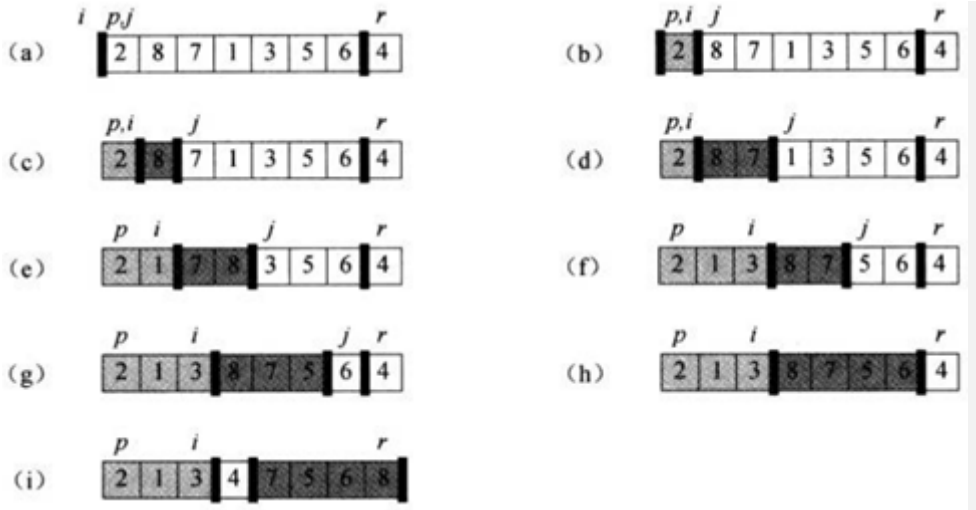
下面给出伪代码：

```
QUICKSORT(A, p, r)
1  if  p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q−1)
4       QUICKSORT(A, q+ 1, r)
```

可以看出，算法的关键是partiton方法的实现。下面给出它的算法实现：

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p−1
3  for j = p to r−1
4       if A[j]≤x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i+ 1] with A[r]
8  return i + 1
```

直接看可能觉得很晕，我们结合实例看看它是如何工作的：



上图(a~i)表示的是对子数组A[p~r] =[2,8,7,1,3,5,6,4]进行排序时，每次迭代之前数组元素和一些变量的值。

我们可以初步看出，在i和j移动的过程中，数组被分成了三个部分（分别用灰色，黑色，白色表示），其中i和j就是分割线，并且浅灰部分的元素均比A[r]小，黑色部分的元素均比A[r]大（(i)图除外，因为循环完毕之后执行了exchange A[i+1] with A[j]）。

我们再仔细分析一下具体细节：

① 首先看迭代之前的部分。它执行了x = A[r]，目的是把子数组A的最后一位作为一个“基准”，其他的所有元素都是和它进行比较。它在迭代过程中值一直都没改变。然后执行i = p –基准 1，此时i在子数组A的左端。

② 再看迭代部分。迭代时j从子数组A的开头逐步移至A的倒数第二位。每次迭代中，会比较当前j位置的值和“基准”的大小，如果小于或相等“基准”，就将灰色部分的长度增加1（ i=i+1 ），然后把j位置的值置换到灰色部分的末尾

（exchange A[i] with A[j]）。这样迭代下来，就能保证灰色部分的值都比“基准”小或相等，而黑色部分的值都比“基准”大。

③ 最后看迭代完成后的部分。就进行了一步 exchange A[i+1] with A[j]操作，就是把“基准”置换到灰色部分与黑色部分之间的位置。

这样所有的操作下来，就产生了一个“临界”位置q，使得A[q]大于等于A[p~q]中的每个元素，而小于等于A[q+1~r]中的每个元素。

更严格的，我们可以用以前介绍的循环不变式（见算法基础——算法导论(1)）来证明其正确性。但由于叙述起来比较麻烦，这里就不给出了。

下面我们给出快速排序(quick-sort)算法的Java实现代码：

```
public static void main(String[] args) {
int[] array = { 9, 2, 4, 0, 4, 1, 3, 5 };
quickSort(array, 0, array.length - 1);
printArray(array);
}

/**
* 快速排序
*
* @param array
* 待排序数组
* @param start
* 待排序子数组的起始索引
* @param end
* 待排序子数组的结束索引
*/public static void quickSort(int[] array, int start, int end) {
if (start < end) {
int position = partition(array, start, end);
quickSort(array, start, position - 1);
quickSort(array, position + 1, end);
}
}

/**
* 重排array，并找出“临界”位置的索引
*
* @param array
* 待重排数组
* @param start
* 待重排子数组的起始索引
* @param end
* 待重排子数组的结束索引
* @return
*/public static int partition(int[] array, int start, int end) {
int position = start - 1;
```

```
int base = array[end];
for (int i = start; i < end; i++) {
if (array[i] <= base) {
position++;
int temp = array[position];
array[position] = array[i];
array[i] = temp;
}
}
int temp = array[position + 1];
array[position + 1] = array[end];
array[end] = temp;
return position + 1;
}

/**
* 打印数组
*
* @param array
*/public static void printArray(int[] array) {
for (int i : array) {
System.out.print(i + "");
}
System.out.println();
}
```