**快速排序是一种划分交换排序方法，它是冒泡法的一种改进算法。**

**快速排序采用了分治策略**，即将原问题划分成为若干个规模更小但与原问题相似的子问题，然后用递归方法解决这些子问题，最后再将它们组合成为原问题的解。

**基本思想**

通过一趟排序将要排序的记录分割成为独立的两个部分，其中一部分的所有记录的关键字值都比另外一部分的所有记录关键字值小，然后再按此方法对这两部分记录分别进行快速排序，整个排序过程可以递归进行，以此达到整个记录序列变成有序。

假设待排序的记录序列为nums[low],nums[low+1],…,nums[high]，首先再该序列中任意选取一条记录（称为支点，通常选取nums[low]），然后将所有关键字比支点小的记录放到它的面前，所有关键字比支点大的记录都放在它的后边，由此可以将该支点记录最后落在的位置i作为分界线，将记录序列分为两个子序列{nums[low],nums[low+1],…,nums[i-1]},{nums[i+1],nums[i+1],…,nums[high]}，这个过程称为一趟快速排序或一次划分。通过一趟排序，支点记录就落在了最终排序结果的位置上。

**步骤**

1.设置两个变量i，j，初值分别为low和high，分别表示待排序列的起始下标和终止下标。

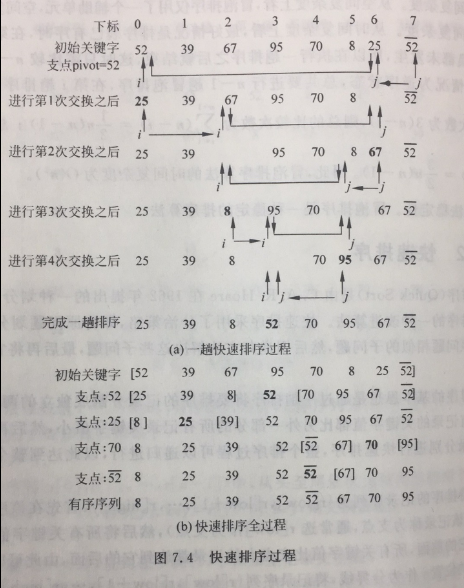
2.将第i个记录暂存再变量piovt中，即piovt=nums[i]

3.从下标为j的位置开始由后向前依次搜索，当找到第一个比piovt的关键字值小的记录时，则将该记录向前移动到下标为i的位置上，然后i=i+1.

4.从下标为i的位置开始由前向后依次搜索，当找到第1个比piovt的关键字值大的记录时，则该记录向后移动到下标为j的位置，然后j=j-1

5.重复3，4，直到i==j

6.nums[i]=pivot



**注意**，每次交换都是从后向前找到一个比支点小的关键字nums[j]移动到当前支点的位置i，直接移动覆盖，不是位置互换！！！因为i已经换过了，所以i++，然后从i开始从前先后遍历找到一个比支点大的关键字nums[i]移动到j，因为之前关键字nums[j]已经移动到了之前i的位置，所以直接移动覆盖就好了，不是互换位置！！！因为j位置已经换过了，所以j++。开始第二次交换。直到i等于j，整个序列遍历完毕。然后i==j的位置就是支点的正确位置，每趟进行的快速排序都能找到当前支点的正确位置，然后把整个序列分为一个整体关键字比支点小的待排序序列，一个整个关键字比支点大的待排序序列。然后在对子序列进行排序。

**代码**

public static int partition(int i, int j, int[] nums) {//一次的快速排序过程，每次的快速查找都能找到找当前支点的正确位置

int pivot = nums[i]; //用当前的low，也就是i值作为支点

while (i < j) { //当i等于j的时候退出，从两边向中间交换，先后再前，为当前的一次移动，当i=j的时候遍历完毕

//从后向前遍历，寻找比pivot小的关键字，前移放到pivot前边，移动到nums[i]

//此时nums[i]数据选为支点，赋值到了pivot中，逻辑上nums[i]为空，不用担心覆盖数据

while (i < j && nums[j] >= pivot) //从后向前略过比pivot大的关键字

j++; //大就继续下一个关键字

//当前关键字nums[j]<pivot小，停下进行交换

if (i < j) { //判断当前位置是否被交换过，i<j的时候说明没有遍历过

nums[i] = nums[j]; //把nums[j]赋值到nums[i],直接覆盖掉数据，因为nums[i]的数据已经保存在pivot中

//相当于此时的nums[i]没有数据，同理覆盖了nums[i]的nums[j]

//数列中含有两个nums[j]的关键字，在j位置的没有用了，因为逻辑上移动到i位置，j位置为逻辑为空数据

i++;//开始从前往后，把大于pivot的关键字放到后边j的位置上

}

//从前先后遍历，寻找比pivot大的关键字，后移放到pivot后边，移动到nums[j]

//此时的nums[j]数据已经赋值到了nums[i]中，逻辑上nums[j]为空数据，不用担心覆盖数据

while (i < j && nums[i] <= pivot)//从前往后略过比pivot小的关键字

i++; //小就继续寻找下一个关键字，

//当关键字nums[i]>pivot大，就停下进行交换

if(i<j) { //判断位置是否有效

nums[j] = nums[i]; //把找到的比pivot小的关键字nums[i]换到nums[j]，所以i位置为逻辑上的空数据

j--;//开始从后向前，把小于pivot的关键字放到前边i的位置上

}

}

nums[i]=pivot;//当i等于j的时候，说明整个序列都遍历过了，i和j重合的位置正好是支点最后所在的正确位置

return i; //返回当前支点的位置

}

//一次的快速排序能找到当前支点的正确位置，同时把序列分为两个需要排序的子序列，然后需要对子序列排序

//直到子序列的low=high为止，当开始的快速排序开始之前low=high说明当前的待排序数列只有一个关键字

public static void quickSort(int low,int high,int[] nums){ //递归方法，实现整个序列的排序

if(low<high){ //当low=high的时候会退出递归，当low等于high的时候说明该序列只有一个关键字

int pivotloc = partition(low,high,nums); //找到当前的支点位置

quickSort(low,pivotloc-1,nums); //用快速排序排支点左边子序列

quickSort(pivotloc+1,high,nums); //用快速排序排当前支点右边子序列

}

}

public static void quickSortWithD(int low,int high,int[] nums){ //非递归方法

LinkedList<Integer> s = new LinkedList<>();

int pivotloc = partition(low,high,nums);

s.push(low);

s.push(pivotloc-1);

s.push(pivotloc+1);

s.push(high);

while(!s.isEmpty()){

int top = s.pop();

int bottom = s.pop();

if(top!=bottom){

pivotloc = partition(bottom,top,nums);

s.push(bottom);

s.push(pivotloc-1);

s.push(pivotloc+1);

s.push(top);

}

}

}

**算法性能分析**

1.空间复杂度

快速排序在系统内部需要一个栈来实现递归，每层递归调用时的指针和参数均需要用栈来存放。快速排序的递归过程可以用一颗二叉树来表示。若每次**划分比较均匀**，则其递归数的高度为O(),所以栈所需要的空间为**O()**。**最坏**情况下，递归树是一个单枝树，树的高度为O(n)，所需要的栈空间为**O(n)**

2.时间复杂度

**比较次数时间复杂度**，在含有n个记录的待排序序列中，一次划分需要约为n次的关键字比较，时间复杂度为**O(n)**。

**快速排序的时间复杂度**，最好情况，每次**均匀划分**为两个等长的子序列，时间复杂度为**O()**。**最坏**情况是当初始关键字序列有序或者基本有序的时候，每次划分只能得到一个子序列，这样的将退化称为冒泡排序，时间复杂度为**O(n2)**

**快速排序的最坏时间为O(n2)**.但是平均性能而言，他是**基于关键字比较的内部排序算法中速度最快的，平均的时间复杂度为O()**

**算法稳定性：不稳定**