快速排序算法-QuickSort

# DivideConquer分治法：

**快速排序**使用**分治法（Divide and conquer）(分治模式)**来把一个**串行（list）**分为**两个子串行（sub-lists）**。

# 快速排序QuickSort算法原理

**递归recursive和分治法。**整个快速排序算法由一个**递归方法**和一个**分治方法**组成。

分治方法用来不断的将串行数据分割成两部分，并返回中间索引值；递归方法是对每一段再进行分割，直至low>=high为止。

### 从数列中挑出一个元素low或high，称为 “基准”（pivot），

### 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），并将该基准值存到到数列的中间位置。这个操作称为分区（partition）操作，对应partition方法。(有多种实现)

### 递归（recursive）方法：再对小数列进行分割。

**递归方法：**两个关键点：终止条件和每次需要进行的操作。

**代码的具体编程思路：**

**一个quickSort递归方法，一个partition分治方法，一个swap交换方法。**

**三个方法：quickSort、partition、swap，三个方法参数一样int[] ,int，int**

**首先从quickSort开始：判断low<high，然后调用partition方法，返回mid索引，然后分别递归本身quickSort两次，从low到mid-1和从mid+1到high。而swap就是一个辅助方法，交换数组两个位置上的值。**

# 达到的目标：

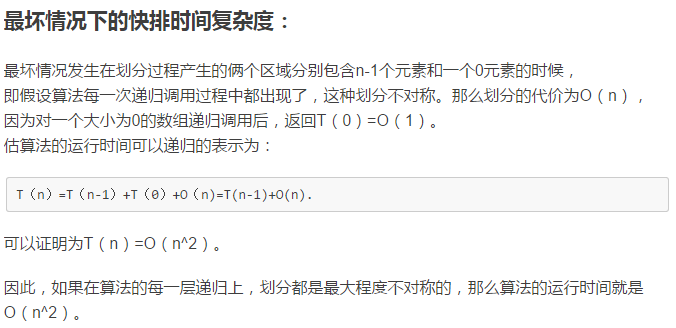
**能够利用C语言或Java语言，迅速的写出这个快速排序程序。**

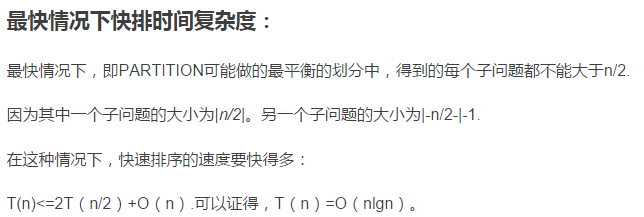
**理解记忆技巧：**

### 传入参数：数组名arr、最小索引low、最大索引high。quickSort方法和partition方法的参数都是这三个。不过quickSort没有返回值,partition方法需要返回分界的索引值(两部分的中间索引)。

### quickSort方法中就是递归调用(简单)。

# 算法复杂度：





## 空间复杂度：

在平均状况下，排序 n 个项目要**Ο(n log n)次**比较。在**最坏状况下则需要Ο(n2)次**比较，但这种状况并不常见。事实上，快速排序通常明显比其他**Ο(n log n) 算法**更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。

最坏的情况：

## 时间复杂度：

# C语言优秀代码：

#include <stdio.h>

**//找出基准值正确的索引值，并保证索引左边的小于基准值，索引右边的大于基准值**

**int partition(int \*arr,int low,int high)**

{

int pivot=arr[high];

int i=low-1;

int j,tmp;

for(j=low;j<high;++j)

if(arr[j]<pivot){

tmp=arr[++i];

arr[i]=arr[j];

arr[j]=tmp;

}

**//将大于或等于基准值的索引最小的数据与基准值交换**。

**tmp=arr[i+1];**

**arr[i+1] =arr[high];**

**arr[high]=tmp;**

return i+1;

}

**//采用递归实现**

**void quick\_sort(int \*arr,int low,int high)**

{

if(low<high){

int mid=**partition(arr,low,high);**

quick\_sort(arr,low,mid-1);

quick\_sort(arr,mid+1,high);

}

}

//test

int main()

{

int arr[10]={1,4,6,2,5,8,7,6,9,12};

quick\_sort(arr,0,9);

int i;

for(i=0;i<10;++i)

printf("%d ",arr[i]);

}

# 快速排序的Java代码

整个快速排序算法的实现主要由**quickSort递归方法**和**partition分治方法**组成；其中**quickSort递归方法是固定的，而partition分治方法主要有两种实现。**

## quickSort递归方法

对于递归quickSort方法都是一致的：

public void **quickSort**(int[] array,int low,int high){

**if(low >= high) return;//递归终止条件**

int mid = partition(array,low,high);

quickSort(array,low,mid-1);//递归前一半

quickSort(array,mid+1,high);//递归后一半

}

## partition分治方法

利用while只需要赋值就可以完成，不需要执行交换，因为让arr[high]或arr[low]作pivot，这样high或low位置就可以认为空出来了。

while实现原理：从两端开始遍历，具体从哪一端开始根据pivot为arr[high]还是arr[low]有关。若pivot=arr[high],则从左端开始，反之从右端开始。原因：左端遍历目的是找出大于pivot的元素，需要调到右端，因此需要空位置在右端。

无论从左向右还是从右向左，原理是一样的。以从左向右为例，利用while循环直到找到大于pivot的元素为止，然后将该元素赋值于空位置，同时该位置就变为空位置。从右向左同理，这样一直循环下去。直到i>=j。最后将pivot赋值于空位置即可。

### 利用while实现

原理：

纯代码：

public int partition(int[] arr,int low,int high) {

int pivot = arr[high];

int i = low,j = high;

while(i < j) {

**while(i < j && pivot >= arr[i]) i++;**

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

**while(i < j && pivot <= arr[j]) j--;**

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

}

arr[j] = pivot;

return j;

}

解释版本：

public int **partition**(int[] arr,int low,int high) {

//令末尾值为基准值;这样末尾值就保存到pivot中，可以认为high为空

int pivot = arr[high];

int i = low,j = high;

while(i < j) {

//pivot = arr[high]==>必须先从高到低遍历

//pivot = arr[low]==>必须先从低到高遍历

**while(i < j && pivot >= arr[i]) i++;**//从高到低:直到找到一个小于pivot的元素

**if(i < j) arr[j--] = arr[i];**//将找到的小于pivot的元素赋值于空位置并j减1；arr[i]可以认为为空

**while(i < j && pivot <= arr[j]) j--;**//从低到高:直到找到一个大于pivot的元素

**if(i < j) arr[i++] = arr[j];/**/将找到的大于pivot的元素赋值于空位置并i+1；arr[j]可以认为为空

}

arr[j] = pivot;//将pivot至于arr[j]空位置

return j;

}

**若利用arr[low]作pivot，则**

public int partition4(int[] arr,int low,int high) {

**int pivot = arr[low];**

int i = low, j = high;

while(i < j) {

while(i < j && pivot <= arr[j]) j--;

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

while(i < j && pivot >= arr[i]) i++;

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

}

**arr[i] = pivot;**

return i;

}

### 利用for循环实现

原理：以arr[high]作为基准值，用j作为遍历索引low—high-1，用i作为小于pivot的索引，从0开始，保证索引i之前的都是小于pivot；遍历过程中，如果arr[j]小于基准值，就将arr[j]与++i值互换位置，如果大于等于基准值，则i不增加，也不交换，继续遍历。

**说白了**：就是把小于基准值的数往前调，把大于基准值的数往后赶。

**如果令arr[high]作pivot，则**

private int partition(int[] array,int low,int high){

int pivot = array[high];//随意指定，一般另high处为基准

int i = low;

**for(int j = low;j < high;j++){//由于pivot为high值,因此不遍历high**

**if(array[j] < pivot){**

**swap(array,i++,j);**

**}**

**}**

swap(array,i,high);

return i;

}

**如果令arr[low]作pivot，则**

public int partition(int[] arr,int low,int high) {

int pivot = arr[low];

int i = high;

**for(int j = high;j > low;j--) {**

**if(arr[j] >= pivot) {**

**swap(arr,j,i--);**

**}**

**}**

swap(arr,i,low);

return i;

}

for循环需要利用**swap方法**：交换数组中两个位置的值

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

int tmp = array[i];

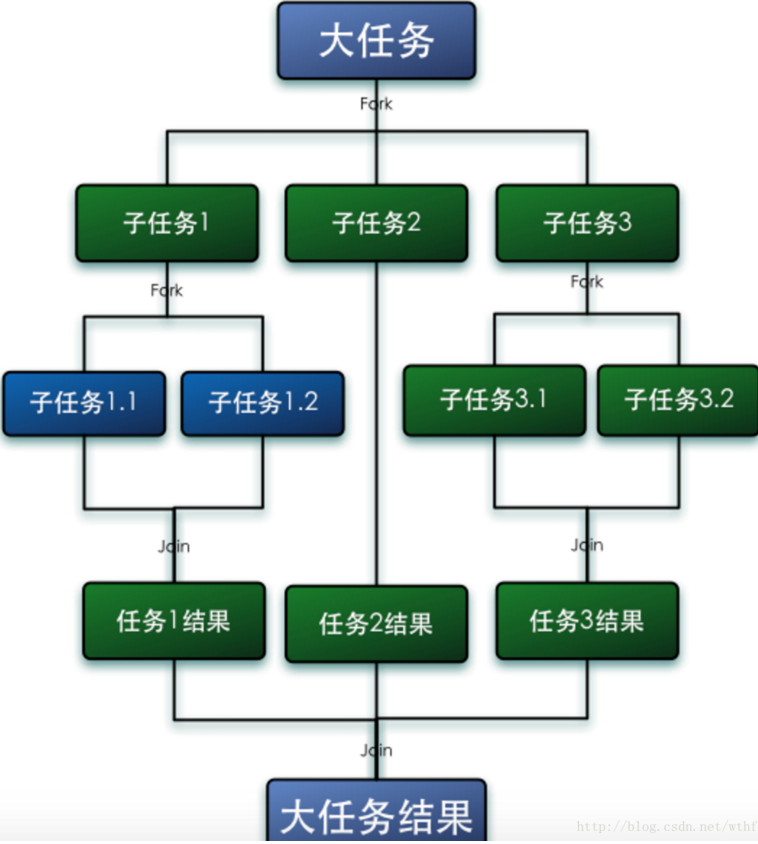
array[i] = array[j];

array[j] = tmp;

}

# ForkJoinTask与快速排序

Fork/Join框架是Java7提供了的一个用于并行执行任务的框架。它可以充分利用多核CPU的优势，将一个大任务切割成多个足够小的任务并行执行（fork）,等所有子任务执行完毕后合并结果（join）,流程图类似下面：



这样来看，fork/join的处理模式与quickSort算法。

倒很相似，都是不断分割任务以进行处理。而事实上，两者都是分治算法的实践者。我们可以将上述排序改成并发版的快速排序。

class **SortTask** extends **RecursiveAction**{

private int[] arr;

private int low;

private int high;

public SortTask(int[] arr,int low,int high) {

this.arr = arr;

this.high = high;

this.low = low;

}

@Override

protected void compute() {

if(low<high){

int i=low+1,j=high,base = arr[low];

while (i<j){

while (arr[j]>=base&&i<j){

j--;

}

while (arr[i] < base && i < j) {

i++;

}

swap(arr,i,j);

}

swap(arr,low,j);

SortTask leftTask =new SortTask(arr,low,j-1);

SortTask rightTask =new SortTask(arr,j+1,high);

//分割成子任务并执行，join()方法会再次调用compute()方法。

**leftTask.fork();**

**rightTask.fork();**

**leftTask.join();**

**leftTask.join();**

}

}

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

int tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

}

}

测试结果：

@Test

public void test()throws Exception {

ForkJoinPool forkJoinPool = new ForkJoinPool();

int[] arr = {4, 1, 8, 6, 7, 9, 3, 2};

SortTask sortTask = new SortTask(arr,0,arr.length-1);

ForkJoinTask<Void> task = forkJoinPool.submit(sortTask);

task.get();

Arrays.stream(arr).forEach(e->System.out.print(e+" "));

// 1 2 3 4 6 7 8 9

}