# 1.2 Quick Sort 快速排序

## 问题简述

Input: 长度为n的无序数组 arr

Output: 元素相同并保持non-decreasing顺序的 arr

## 时间复杂度

Quick Sort 的时间复杂度为：$O(N^2)$

Randomized Quick Sort 的时间复杂度为: O(nlogn)

## PsuedoCode

**Algorithm:** QuickSort

**Input** : An array $A[1..n]$.
**Output:** $A$ sorted from 1 to $n$.
Choose a pivot $A[p]$
$r := \text{Partition}(A[1..n], p)$
$\text{QuickSort}(A[1..r-1])$
$\text{QuickSort}(A[r+1..n])$
**return** $A$

## 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；

## 算法思想

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序 n 个项目要 O(nlogn) 次比较。在最坏状况下则需要 O(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 O(nlogn) 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。

快速排序使用分治法（Divide and conquer）策略来把一个串行（list）分为两个子串行（sub-lists）。

快速排序又是一种分而治之思想在排序算法上的典型应用。本质上来看，快速排序应该算是在冒泡排序基础上的递归分治法。

快速排序的名字起的是简单粗暴，因为一听到这个名字你就知道它存在的意义，就是快，而且效率高！

虽然 Worst Case 的时间复杂度达到了 $O(n^2)$，但是它的平摊期望时间是 O(nlogn)，且 O(nlogn) 记号中隐含的常数因子很小，比复杂度稳定等于 O(nlogn) 的归并排序要小很多。所以，对绝大多数顺序性较弱的随机数列而言，快速排序总是优于归并排序。

另外，parition 算法是我们出现Worst Case的主要原因，我们可以通过randomly 选择我们的partition point来获取improvement，实现O(nlogn)的复杂度

## Quick Sort 代码 (java)

- **Swap**

```java
/**
     *
     * @param arr, an array
     * @param a, the first index
     * @param b, the second index
     *
     * Given an array, we will exchange the position in it
     */
    public static void swap(int[] arr, int a, int b){
        int num = arr[a];
        arr[a] = arr[b];
        arr[b] = num;
    }
```

## • Partition

```java
    /**
     *
     * @param input, an array
     * @param left, the left index that we are allowed to do operation
     * @param right, the right bound that we are allow to do operation, if the array start at 0, it is the length
     * @param pivot_index, the index of pivot we select
     * @return the index of pivot after partition, we will make values smaller or equal to it at left,
     *          values larger than the pivot at right
     */
    public static int partition(int[] input, int left, int right, int pivot_index){
        // Retrieve the value of the pivot from array
        int pivot_val = input[pivot_index];

        // Set up a pointer, point to the left of our array
        int p = left;

        // Set up the end cursor of the ⩽ region
        int small_equal_cur = left;

        while(p⩽right){
            // Case1: if the element ⩽ pivot value, exchage the value with the last element in the region
            if(input[p]⩽pivot_val){
                swap(input, small_equal_cur, p);
                small_equal_cur++;
            }
```

```
26                // Case2: the element is > pivot value: do nothing, keep while loop
27                p++;
28            }
29            return small_equal_cur-1;
30
31        }
```

## • Classic QuickSort

```
1         /**
2          *
3          * @param arr, an array
4          * @param start, the start index of an array, 0
5          * @param end, the end index of an array, length-1
6          * @return return a sorted int array
7          */
8
9         public static void quick_sort_classic(int[] arr, int start, int end){
10
11            //Base Case1: When the lenght of arr is 1, do nothing
12            if (end == start){
13                ;
14            }
15            // Base Case2: the input arr is sorted, do nothing
16            else if(end < start){
17                ;
18            }
19            //Recursive Case:
20            else{
21                // set the last number in the array to be pivot, here we use index of
    it
22                int pivot_index = end;
23                int pivot_pos = partition(arr, start, end, pivot_index);
24
25                //Recursive call left
26                quick_sort_classic(arr, start, pivot_pos-1);
27                quick_sort_classic(arr, pivot_pos+1, end);
28                //Recursive call right
29            }
30        }
```

- **Randimnized QuickSort**

```java
import java.util.Random;
    /**
     *
     * @param arr, an array
     * @param start, the start index of an array, 0
     * @param end, the end index of an array, length-1
     * @return return a sorted int array
     */

     public static void quick_sort_random(int[] arr, int start, int end){

        //Base Case1: When the lenght of arr is 1, do nothing
        if (end == start){
            ;
        }
        // Base Case2: the input arr is sorted, do nothing
        else if(end < start){
            ;
        }
        //Recursive Case:
        else{
            // Here we use random method to pick pivot

            // Create an instance of Random class
            Random random = new Random();

            // Set the pivot index
            int pivot_index = random.nextInt(end - start + 1) + start;
            // Partition and retrieve the index of pivot after partition
            int pivot_pos = partition(arr, start, end, pivot_index);

            //Recursive call left
            quick_sort_random(arr, start, pivot_pos-1);
            quick_sort_random(arr, pivot_pos+1, end);
            //Recursive call right
        }
    }
```