# 快速排序 Quick Sort

同样可以使排序的时间复杂度简化为 $O(nlogn)$ 的算法不仅只有归并排序（Merge Sort），另一种排序方法快速排序也可以做到。

## 快速排序 Quick Sort

在实际应用中，当为存在大量变体的数组进行排序时，最快的排序算法就是快速排序。它的平均时间成本为 $O(NlogN)$，而最坏情况下的成本是 $O(N^2)$ 然而，最坏的情况几乎很少发生。快速算法并不稳定，它是另一种分而治之算法的变体。

### 原理

- 划分

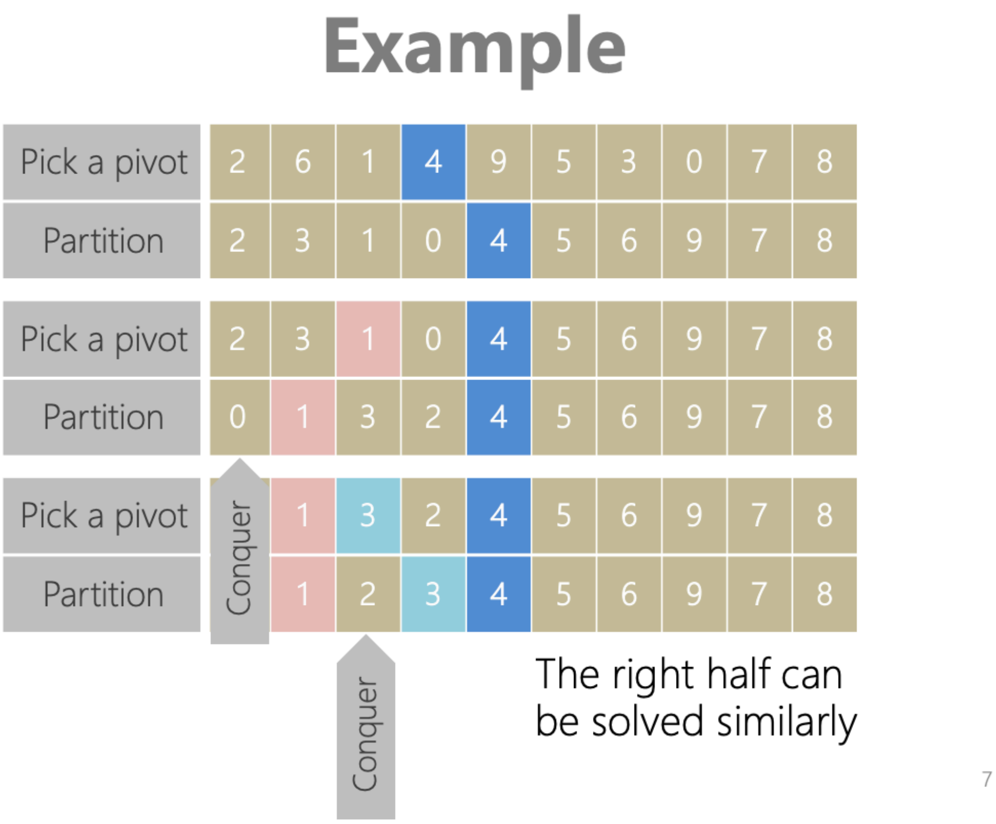
1. 在 $S$ 中选取一个元素 $v$，$v$ 被称为中心点（Pivot）
2. 将 $S-\{v\}$ 分成两个互不相交的组
   $S1=\{x \in S-v | x <= v\}$
   $S2=\{x \in S-v | x >= v\}$
3. 递归划分 S1 和 S2

- 排序

如果 $S$ 中的元素不超过 1 个，则直接返回

- 结合

无需任何操作。排序后的 $S1$ 在递归完成时紧接着 $v$，然后是递归完成时排序后的 $S2$，它们会一起组成一个排序后的新列表




快速排序

## 主函数实现

```
// 伪代码实现
QUICKSORT(A, left, right)
    IF left ⩾ right
        return
    q = PARTITION(A, left, right)
    //q is the position of the pivot
    QUICKSORT(A, left, q-1)
    QUICKSORT(A, q+1, right)
```

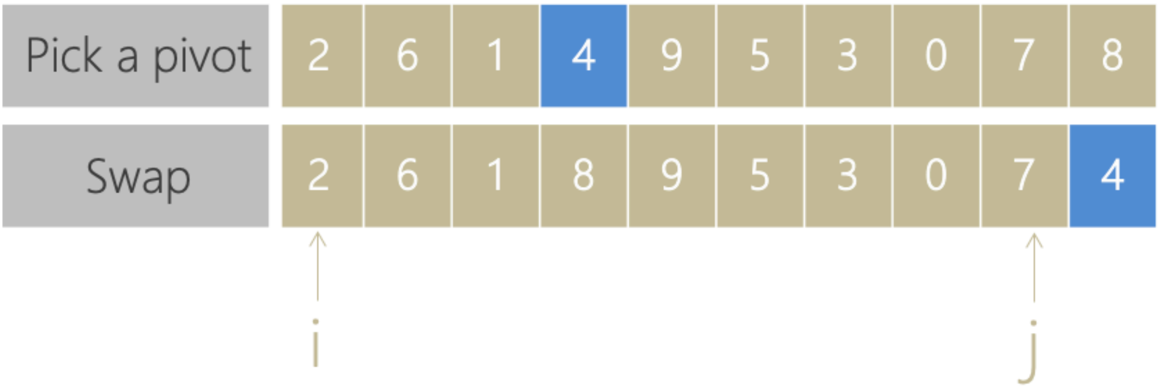
```java
// Java实现
public static void quickSort(int[] A) {
    quickSort(A, 0, A.length - 1);
}

private static void quickSort(int[] A, int left, int right) {
    // Base case: if the left index is greater than or equal to the right index,
    // return
    if (left ⩾ right) {
        return;
    }
    // Partition the array and get the index of the pivot
    int q = partition(A, left, right);
    // Recursively apply quickSort to the left and right partitions
    quickSort(A, left, q - 1);
    quickSort(A, q + 1, right);
}
```

## 分区

这是快速排序算法的关键步骤目标。在分区中，以选中的中心点作为标准，将剩余元素划分为两个较小的集合。实现分割的方法很多，即使是最微小的偏差也可能导致令人惊讶的糟糕结果。我们将在此学习一种简单高效的分区策略。

要分割数组 $A[left...right]$ 时，通过将中心点元素与最后一个元素对调，将其取出，即 swap 中心点和 $A[right]$。此时，让 $i$ 从第一个元素开始，$j$ 从倒数第二个元素开始，即 $i = left$，$j = right - 1$。

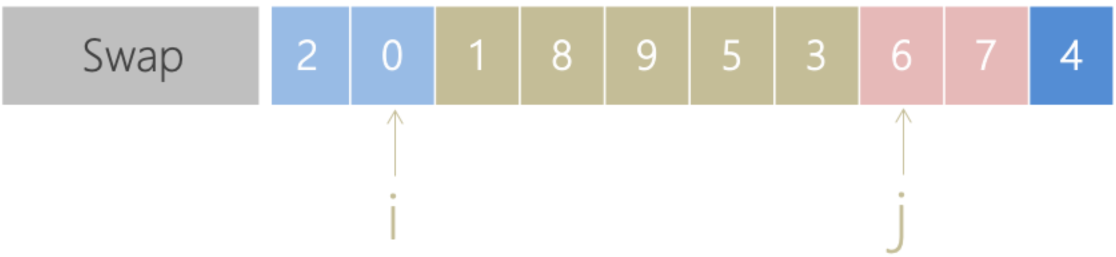



swap

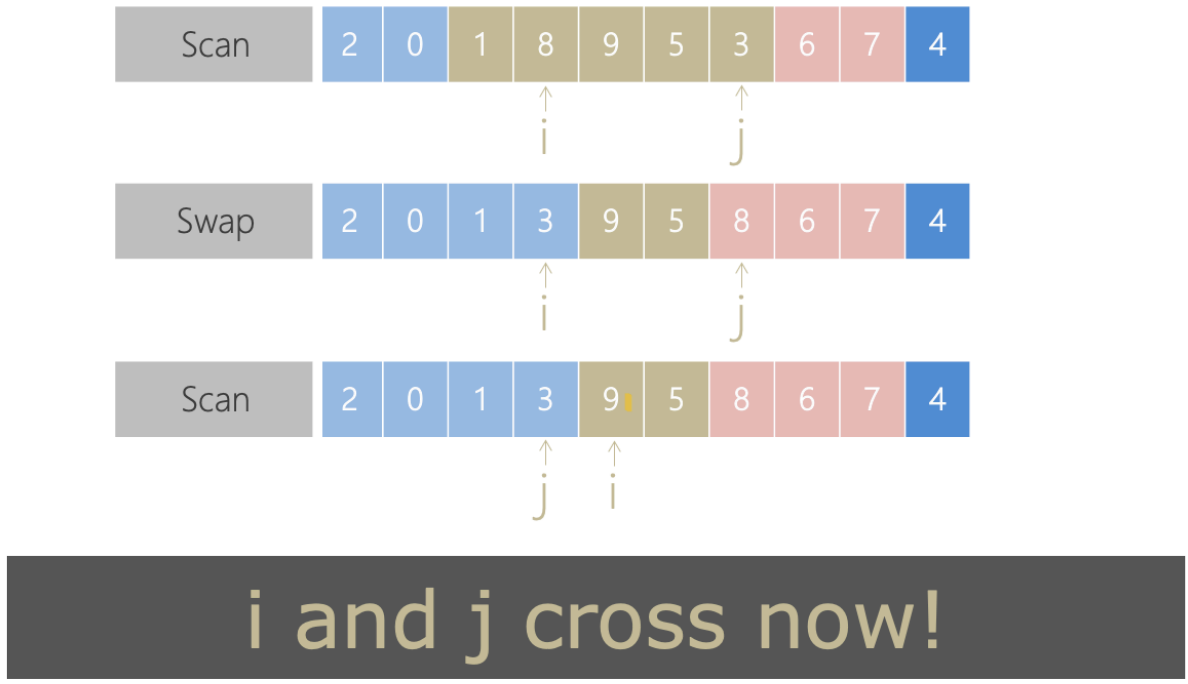
- 目标：
  > $A[left...i]$ 小于或等于中心点
  > $A[j...right]$ 大于或等于中心点
- 策略：
  > 当 $i < j$ 时
  > 向右移动 $i$，跳过小于中心点的元素
  > 向左移动 $j$，跳过大于中心点的元素
  > 当 $i$ 和 $j$ 都停止时
  > $A[i] >= pivot$
  > $A[j] <= pivot$（$A[i]$ 和 $A[j]$ 现在应该对调）


开始分区

- 当 $i$ 和 $j$ 停止且 $i$ 在 $j$ 的左边时（因此是合法的）
  1. 交换 $A[i]$ 和 $A[j]$
     然后两个元素都在 “正确 ”的一边
  2. 对调后
     $A[i] <= pivot$
     $A[j] >= pivot$
  3. 重复该过程，直到 $i$ 和 $j$ 相交为止




正在分区




分区结束

- 当 $i$ 和 $j$ 相交时

  交换 $A[i]$ 和枢轴

- 结果：

  $A[p] <= pivot$，对于 $p < i$

  $A[p] >= pivot$，对于 $p > j$

- **分区完成**

## 代码实现

```
// 伪代码实现
PARTITION(A, left, right)
    p = PIVOT(A, left, right)
    //p is the position of the pivot
    swap A[p] and A[right]
```

```
    i = left, j = right-1, pivot = A[right]
    WHILE true
        WHILE i<right AND A[i]<pivot
            i++
        WHILE j>=left AND A[j]>pivot
            j--
       IF i<j
           swap A[i] and A[j]
           i++, j--
       ELSE
           BREAK
   swap A[i] and A[right]
```

## 特殊情况

对于很小的数组，快速排序的性能不如插入排序好。小到什么程度取决于很多因素，如递归调用所花费的时间、编译器等。因此不要在小数组中递归使用快速排序，取而代之的是使用对小数组有效的排序算法，如插入排序。

## 主函数代码

```
QUICKSORT(A, left, right)
   IF left >= right - 10
      INSERTIONSORT(A, left, right)
      RETURN
   q = PARTITION(A, left, right)
   //q is the position of the pivot
   QUICKSORT(A, left, q-1)
   QUICKSORT(A, q+1, right)
```

## 中心点的选择

在快速排序中我们需要利用中心点作为标准来分区，因此中心点的选择也是很重要的。

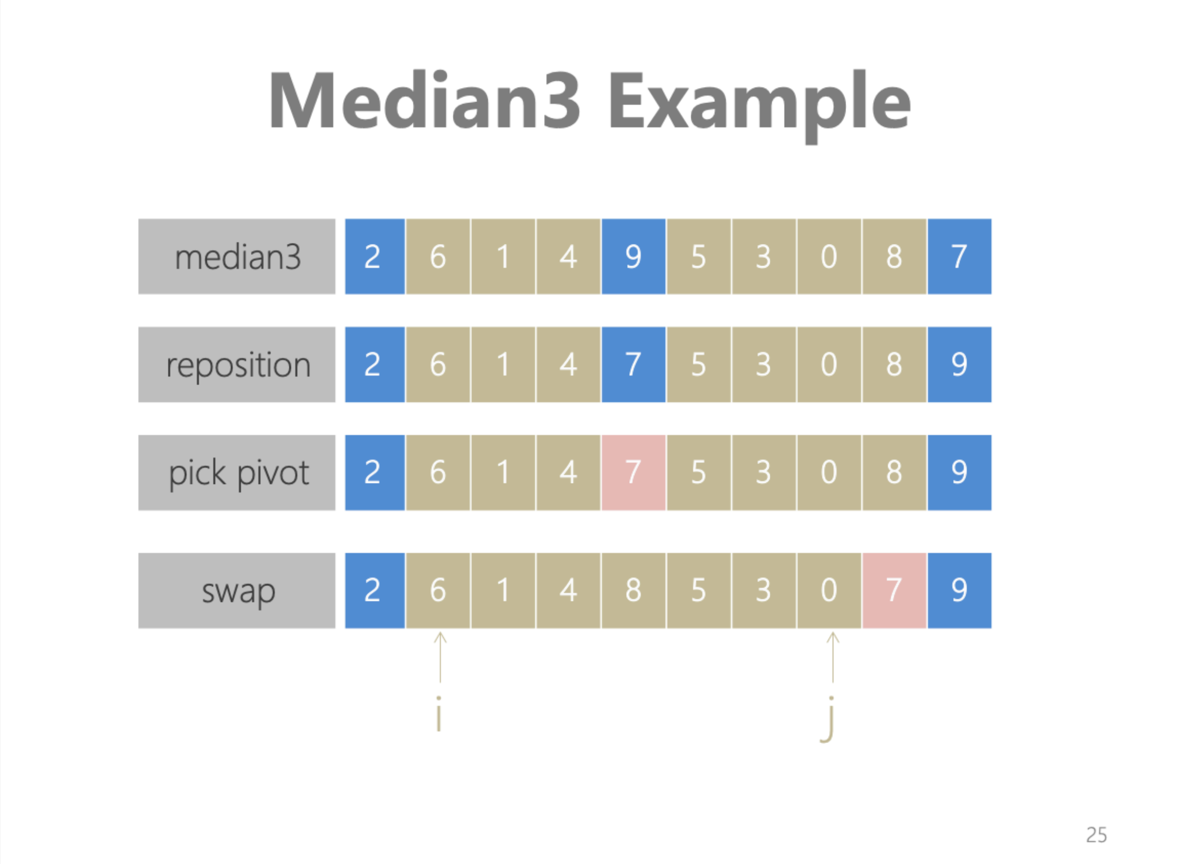
1. 使用第一个元素或最后一个作为中心点
   - 如果输入是随机的，OK没问题
   - 如果输入是预排序的（或顺序相反）
     则所有元素都进入 $S2$（或 $S1$）
     这种情况在递归调用中持续发生
     结果为 $O(n^2)$
2. 随机选择元素作为中心点
   - 大体上可能不错
   - 但随机选择元素的代价可能太大
3. （理论最佳方案）使用数组的中位数
   - 如果数组排序，中位数就是中间的元素。例如，如果数组中有 9 个元素，中位数就是第 5 个最大的元素

     分区将总是数组大致分成两半
   - 此时为最佳的快速排序，时间复杂度为 $O(NlogN)$
   - 不过，要找到精确的中位数，成本很高

     例如，对数组排序，选取中间值
4. （折中方案）使用三个元素的中位数 Median3
   - 只比较三个元素：最左边、最右边和中间元素，必要的时候还可以交换他们的位置，使得最终三个位置的元素大小如下所示
     $A[left] = min$

     $A[right] = max$

> $A[center] = center$
> 此时选取 $A[center]$ 作为中心点
>
> - 交换 $A[center]$ 和 $A[right-1]$，使枢轴位于倒数第二位，这样可以减小递归的范围




选取中心点

## 代码实现

```
// 伪代码实现
PARTITION(A, left, right)
   MEDIAN3(A, left, right)
   // MEDIAN3 repositions the left, center
   // and the right elements
   i = left+1, j = right-2, pivot = A[right-1]
   WHILE true
```

```
        WHILE A[i]<pivot
            i++
        WHILE A[j]>pivot
            j--
        IF i<j
            swap A[i] and A[j]
            i++, j--
        ELSE
            BREAK
    Swap A[i] and A[right-1]
```

```java
// Java实现
private static int partition(int[] A, int left, int right) {
    // Use median-of-three method to choose the pivot
    median3(A, left, right);
    int i = left + 1; // Start from the first element after the left
    int j = right - 2; // Start from the last element before the right
    int pivot = A[right - 1]; // The pivot is now at right - 1
    int temp;

    // Check if indices are valid
    if (i > right || j < left)
        return i; // Return if out of bounds

    while (true) {
        // Move `i` to the right while the current element is less than the pivot
        while (i <= j && A[i] < pivot) {
            i++;
        }
        // Move `j` to the left while the current element
```

```
is greater than the pivot
        while (j >= i && A[j] > pivot) {
            j--;
        }
        // If indices haven't crossed, swap elements
        if (i < j) {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
            i++;
            j--;
        } else {
            break; // Break the loop if indices cross
        }
    }
    // Swap the pivot to its correct position
    temp = A[i];
    A[i] = A[right - 1];
    A[right - 1] = temp;
    return i; // Return the new index of the pivot
}
```

```
// java代码实现
private static void median3(int[] A, int left, int right) {
    int mid = (left + right) / 2; // Calculate the middle index
    int temp;

    // Sort the three elements: A[left], A[mid], A[right]
    if (A[left] > A[right]) {
        temp = A[left];
        A[left] = A[right];
        A[right] = temp;
    }
```

```
    if (A[left] > A[mid]) {
        temp = A[left];
        A[left] = A[mid];
        A[mid] = temp;
    }
    if (A[mid] > A[right]) {
        temp = A[mid];
        A[mid] = A[right];
        A[right] = temp;
    }

    // Move the median (middle value) to the position right - 1
    temp = A[mid];
    A[mid] = A[right - 1];
    A[right - 1] = temp;
}
```

# 分析

## 快速排序与归并排序

快速排序和归并排序的平均耗时都是$O(NlogN)$，为什么快速排序比归并排序快?

在内部递归中，归并排序由多个增减（以 1 为单位，速度很快）、测试和跳转组成。
**而快速排序中没有合并排序递归里的这些额外处理步骤。**

因此快速排序的速度对于大数组很快。

## 复杂度分析

假设中心点的选择是 Median3 方法，运行时间是 $T(n)$

- 分割

> 选择支点：$O(1)$
>
> 分区：$O(n)$
>
> 递归调用： T(i) + T(n-i-1)
> i：$S1$ 中的元素个数

- 排序与合并：$O(1)$

可得运行时间为：$T(n) = T(i) + T(n-i-1) + O(n)$

## 最差情况

中心点始终是最小的元素，或分区总是不平衡的，因此可得

$$
\begin{aligned}
T(N) &= T(N-1) + cN, \\
T(N-1) &= T(N-2) + c(N-1), \\
T(N-2) &= T(N-3) + c(N-2), \\
&\vdots \\
T(2) &= T(1) + c(2), \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2).
\end{aligned}
$$

在此时最差情况下，时间复杂度是 $O(n^2)$

## 最好情况

分区完全平衡，或是中心点始终位是数组的中位数，因此可得：

$T(n) = 2T(n/2) + n$

$= 2[2T(n/2^2) + n/2] + n$

$= 22T(n/2^2) + 2n$

$= 23T(n/2^3) + 3n$

$= 2iT(n/2^i) + i * n$

让 $i = log(n)$,

$= nT(n/n) + n * log(n)$

$= O(n * log(n))$

因此最好情况是 $O(n * log(n))$

## 总结

可以得到如下的表

| 情况 | 时间复杂度 |
| --- | --- |
| 最坏情况 | $O(n^2)$ |
| 最好情况 | $O(nlog(n))$ |
| 平均情况 | $O(nlog(n))$ |

由此可得，快速排序的平均情况下的时间复杂度是 $O(nlog(n))$