排序Sorting

无隅

谈谈排序

- 排序是非常基础的算法
- 排序是处理数据很常见的一种做法

面试中的排序算法

- 一般排序算法（以元素比较为基础）
  - 插入排序，冒泡排序
  - 快速排序， 归并排序， 堆排序
- 特殊排序算法
  - 计数排序
  - 桶排序

最常考察的排序：快速排序， 归并排序，插入排序，冒泡排序

插入排序

核心想法：像排序一手扑克牌，把一张牌开始时，我们的左手为空并且桌子上的牌面向下。然后我们每次从桌子上拿走一张牌并将它插入左手中正确的位置。为了找到一张牌的正确位置，我们从右到左将它已在手中的每张牌进行比较

时间复杂度：

Best: O(n)

Average: O(n^2)

Worst: O(n^2)

插入排序 – 时间复杂度分析

- 最好情况：数组已经排好序
- 最坏情况：数组反向排序
- 平均情况：确定在什么位置插入元素num，平均地数组中有一半元素大于num，一半小于num

示例代码

```java
public void insertionSort(int[] nums) {
    for (int j = 1; j < length; j++) {
        int key = nums[j];
        int i = j - 1;
        while (i >= 0 && nums[i] > key) {
            nums[i + 1] = nums[i];
            i--;
        }

        nums[i + 1] = key;
    }
}
```

循环不变式

- 用来帮助我们理解算法的正确性
- 循环不变式过程：

初始化：循环的第一次迭代之前循环不变式为真

保持：如果循环的某次迭代之前它为真，那么下次迭代之前它仍为真（在从当前状态到下一个状态的过程中能得以保持）

终止：在循环终止时，不变式为我们提供了一个有用的性质，该性质有助于证明算法是正确的。如果程序可以在某种条件下终止，那么在终止的时候，就可以得到自己想要的正确结果

插入排序 - 循环不变式

在for循环(循环变量为j)的每次迭代的开始，包含元素nums[0…j - 1]的子数组是有序的

冒泡排序

核心想法：反复交换相邻的未按次序排列的元素

时间复杂度：

最好时间复杂度：$O(n)$

平均时间复杂度：$O(n^2)$

最坏时间复杂度：$O(n^2)$

示例代码

```java
public void bubbleSort(int[] nums) {
    int len = nums.length;

    for (int i = 0; i < len; i++) {
        for (int j = 1; j < (len - i); j++) {
            if (nums[j - 1] > nums[j]) {
                int temp = nums[j - 1];
                nums[j - 1] = nums[j];
                nums[j] = temp;
            }
        }
    }
}
```

归并排序

核心想法：递归， 分治法（Divide and Conquer)

时间复杂度：

最好时间复杂度：O(nlogn)

平均时间复杂度：O(nlogn)

最坏时间复杂度：O(nlogn)

关于分治法

分解（Divide）：将问题划分为一些子问题，子问题的形式与原问题一样，只是规模更小

解决（Conquer）：递归地求解出子问题。如果子问题规模足够小，则停止递归，直接求解

合并（Combine）：将子问题的解组合成原问题的解

归并排序 - 分治法

分解：将数组划分为两个规模为n/2的子数组

解决：递归地对两个子数组分别排序

合并：递归地合并两个子数组

归并排序 - 时间复杂度分析

对半分 $T(n) = 2T(n/2) + O(n) \rightarrow O(nlogn)$

归并排序 - 循环不变式

对于merge中的每一轮迭代开始时，子数组nums[0…index - 1]包含left[0…i]和right[0…j]中的index个最小元素

示例代码1

```java
public void mergeSort(int[] nums){
    mergeSort(nums, 0, nums.length-1);
}

private void mergeSort(int[] nums, int begin, int end) {
    if (begin < end) {
        int mid = (begin + end) / 2;
        mergeSort(nums, begin, mid);
        mergeSort(nums, mid+1, end);
        merge(nums, begin, mid, end);
    }
}
```

```java
private void merge(int[] nums, int start, int mid, int end) {
    int leftLen = mid - start + 1;
    int rightLen = end - mid;

    int[] left = new int[leftLen];
    int[] right = new int[rightLen];

    for (int i = 0; i < leftLen; i++) {
        left[i] = nums[start + i];
    }

    for (int j = 0; j < rightLen; j++) {
        right[i] = nums[mid + 1 + j];
    }

    int index = start;
    int i, j = 0;
    while (i < leftLen && j < rightLen) {
        if (left[i] <= right[j]) {
            nums[index++] = left[i++];
        } else {
            nums[index++] = rightLen[j++];
        }
    }

    while (i < leftLen) {
        nums[index++] = left[i++];
    }

    while (j < rightLen) {
        nums[index++] = right[j++];
    }
}
```

```java
public void mergeSort(int[] nums) {
    int[] temp = new int[nums.length];
    mergeSort(nums, 0, nums.length - 1, temp);
}

private void mergeSort(int[] nums, int start, int end, int[] temp) {
    if (start >= end) {
        return;
    }

    int left = start, right = end;
    int mid = (start + end) / 2;

    mergeSort(nums, start, mid, temp);
    mergeSort(nums, mid + 1, end, temp);
    merge(nums, start, mid, end, temp);
}
```

示例代码2

```java
private void merge(int[] nums, int start, int mid, int end, int[] temp) {
    int left = start;
    int right = mid + 1;
    int index = start;

    while (left <= mid && right <= end) {
        if (nums[left] < nums[right]) {
            temp[index++] = nums[left++];
        } else {
            temp[index++] = nums[right++];
        }
    }

    while (left <= mid) {
        temp[index++] = nums[left++];
    }
    while (right <= end) {
        temp[index++] = nums[right++];
    }

    for (index = start; index <= end; index++) {
        nums[index] = temp[index];
    }
}
```

快速排序

核心想法：递归， 分治法（Divide and Conquer)

时间复杂度：

最好时间复杂度: O(nlogn)

平均时间复杂度: O(nlogn)

最坏时间复杂度: O(nlogn)

快速排序 - 分治法

分解：将数组划分为两个（可能为空）子数组，使得前一个子数组中的每个元素都小于或等于nums[pivot]，后一个都大于nums[pivot]

解决：递归地对两个子数组分别排序

合并：由于子数组都是原地排序不需要合并

快速排序 – 时间复杂度分析

- 最坏情况：划分的两个数组分别有n – 1个元素与0个元素，
  划分操作的时间复杂度为$O(n)$
  $T(n) = T(n - 1) + T(0) + O(n) \rightarrow O(n^2)$
- 最好情况：对半分
  $T(n) = 2T(n/2) + O(n) \rightarrow O(nlogn)$
- 平均情况：平衡划分
  任何以常数比例的划分都会产生深度为O(lgn)
  的递归树，其中每一层的时间代价都是O(n)
  结果: $O(nlogn)$

快速排序 - 循环不变式

对于每一轮迭代开始时对于任意数组下标i都有：

- 若start <= i <= pivot; 则nums[i] <= nums[pivot]
- 若pivot + 1 <= i <= end; 则nums[i] > nums[pivot]
- 若i = pivot; 则nums[i] == nums[pivot]

```java
public void quicksort(int[] nums, int begin, int end) {
    if (begin >= end) {
        return;
    }
    int pivotPostion = partition(nums, begin, end);
    quicksort(nums, begin, pivotPostion - 1);
    quicksort(nums, pivotPostion + 1, end);
}
```

示例代码

```java
public int partition(int[] nums, int begin, int end) {
    int pivot = nums[begin];
    while (begin < end) {
        while (begin < end && nums[end] >= pivot) {
            end--;
        }
        nums[begin] = nums[end];
        while (begin < end && nums[begin] <= pivot) {
            begin++;
        }
        nums[end] = nums[begin];
    }
    nums[begin] = pivot;
    return begin;
}
```

```java
public int partition(int[] nums, int begin, int end) {
    int key = nums[end];
    int j = begin - 1;

    for (int i = 0; i < nums.length; i++) {
        if (nums[i] <= key) {
            j = j + 1;
            if (i != j) {
                swap(nums, i, j);
            }
        }
    }
    swap(nums, j + 1, end);
    return j + 1;
}
```

扩展1：快速排序随机化版本

```
public int randomizedPartition(nums, begin, end) {
    int i = random(begin, end);
    swap(nums, i, end);
    return partition(nums, begin, end)
}
```

扩展2：期望为线性时间的选择算法(quick select)

求数组nums中第k小的元素

```java
public int quickSelect(int start, int end, int k, int[] nums) {
    if (start == end) {
        return nums[start];
    }
    int index = partition(nums, start, end);
    int num = index - start + 1;
    if (k == num) {
        return nums[index];
    }
    else if (num > k) {
        return quickSelect(start, index - 1, k, nums);
    }
    else {
        return quickSelect(index + 1, end, k - num, nums);
    }
}
```

选择排序运行过程

- 检查递归基本情况
- 将数组划分为两个（可能为空）子数组，使得前一个子数组中的每个元素都小于或等于nums[index]，后一个都大于nums[index]
- 检查nums[index]是否为第k小的元素，

  1. 如果是则返回nums[index]
  2. 否则要确定第k小的元素落在哪一个子数组

  （1）如果num > k，则落入划分低区

  （2）反之num < k 则落下划分高区，而且我们已经知道有num个 元素比nums[index]小