快速排序（QuickSort）和快速选择（QuickSelection）

https://nicodechal.github.io/2020/01/12/quick-sort-and-quick-selection/

快速排序算法（QuickSort）

快排是一个经典的排序算法，其时间复杂度为
O(nlogn)
O(nlogn), 其基本流程如下, 对于以个数组 nums：

1. 首先选定一个轴心值 p。
2. 将数组中小于 p 的值移到数组左端，其他移动到数组右端。
3. 递归分别处理数组中 p 左右两边的子数组。

大致代码如下：

```
function quickSort(nums, s, e) {
   let i = s, j = e;
   // move a random chosen value to index s.
   const rk = Math.floor(s + Math.random() * (e - s + 1));
   [nums[s], nums[rk]] = [nums[rk], nums[s]];
   const p = nums[s];
   // start moving i, j until i >= j, and after the loop, i will be filled
with p value.
   while (i < j) {
       while (i < j && nums[j] >= p) j--;
       nums[i] = nums[j];
       while (i < j && nums[i] <= p) i++;
       nums[j] = nums[i];
   }
   nums[i] = p;
   // recursion do quick sort.
   quickSort(nums, s, i - 1);
   quickSort(nums, i + 1, e);
}
```

代码实现使用 [i, j] 表示未调整的范围，首先，算法随机选择一个轴心值所在坐标 rk，并将该值换到数组开头。然后进入循环运行直到为调整范围长度为 0。
循环中，每次讲 j 移动到第一个小于 p 的坐标，然后将该值移动到左边（目前 i 指向的位置，此时 i 的值会被覆盖，不过这个值要么原本是 p，要么已经移到别的地方，所以可以直接覆盖），然后移动 i 直到第一个大于 p 的值，将该值移到右边（目前 j 指向的地方，这个值已经移到了 i 之前的位置）。
总的来说，循环就是在将小于等于 p 的值移到左边，大于等于 p 的值移到右边。
然后, i 所在的地方就是需要将 p 填入的位置（这个位置同时也将数组分为了左右两个子数组）。

最后，递归处理子数组。
快排是原地排序，所以最后无需返回，**nums** 已经有序。
快排理论复杂度为
O(nlogn)，实际操作时，需要结合各种优化才能快速运行，本例实现使用了两种优化：1. 随机选择轴心，2. 循环使用移位代替交换，来提高效率。

快速选择算法（QuickSelection）

快速排序算法基于快排的思想衍生，其可以使某些需要
O(nlogn) 时间复杂度的问题，在平均复杂度
O(n) 下完成。常见的例子是求数组的第 k 小的数，运用快速选择算法的基本流程如下：

1. 首先选定一个轴心值 p。
2. 将数组中小于 p 的值移到数组左端，其他移动到数组右端。
3. 计算轴心左端的数 (包括轴心自己) 有多少，记为 count
4. 如果 count 正好为 k，则返回此时轴心值，此值即为第 k 小的数。
5. 如果左端的数 count 大于 k，说明在左端，所以只递归左边即可。
6. 如果不在左端，只递归在右边寻找。

大致代码如下：

```
function quickSelection(nums, s, e, k) {
   let i = s, j = e;
   // move a random chosen value to index s.
   const rk = Math.floor(s + Math.random() * (e - s + 1));
   [nums[s], nums[rk]] = [nums[rk], nums[s]];
   const p = nums[s];
   // start moving i, j until i >= j, and after loop, i will be filled with
p value.
   while (i < j) {
       while (i < j && nums[j] >= p) j--;
       nums[i] = nums[j];
       while (i < j && nums[i] <= p) i++;
       nums[j] = nums[i];
   }
   nums[i] = p;
   const pk = i - s + 1;
   if (pk == k) return nums[i];
   if (pk > k) return quickSelection(nums, s, i - 1, k);
   else return quickSelection(nums, i + 1, e, k - pk);
}
```

与前面的快排几乎一样，只在后面有所不同，当完成分半以后，计算左边有多少个数记为 pk，如果其值正好为 k，则所求数已经找到，直接返回所求数，如果大于 k 则在左边寻找，否则在右边寻找。
算法可以的到平均复杂度 O(n)

一个例子

324. Wiggle Sort II (M)

```
Given an integer array nums, reorder it such that nums[0] < nums[1] >
nums[2] < nums[3]....
You may assume the input array always has a valid answer.

// Example 1:
Input: nums = [1,5,1,1,6,4]
Output: [1,6,1,5,1,4]
Explanation: [1,4,1,5,1,6] is also accepted.

// Example 2:
Input: nums = [1,3,2,2,3,1]
Output: [2,3,1,3,1,2]
```

LeetCode.324 要求在 O(n) 时间复杂度和 O(1) 空间复杂度将数组整理至：nums[0] < nums[1] > nums[2] < nums[3]... 的形式。

首先考虑数组有序的情况如何解决问题，如果数组已经有序 **(降序)** 则可以将数组分为两半 **[left, right]**，令左边的数都按顺序放在奇数位，右边的数按顺序放在偶数位即可。

此时每一个奇数位上的数一定大于它两边的数 **(** 两边的数都来自右边，一定小于它，偶数位同理 **)**。

上面没有说明有数相等的情况。由于题目确认一定有解，所以相同的数不会超过一半，因此，可以将左右两半根据是否等于中位数（只有中位数可能跨过左右的界限）再细分为两部分：[left(>mid, =mid), right(=mid, <mid)]。

此时，可以看到，由于相等的数不会超过一半，所以如果按顺序将左右的数字分别放入奇数偶数位，则不会出现一个数左右有和它相等的数：

```
right: [=mid, <mid] //相等的部分不会相交
left: [>mid, =mid]
```

举例来说：

```
before: 1 3 2 2 1 2
sorted: 3 2 2 2 1 1
  cut: 3 2 2 - 2 1 1
 fill:   3   2   2
       2   1   1
result: 2 3 1 2 1 2
```

即使刚好有一半的数相等，也不会出现冲突 (注意交界处，奇数位的数组中位数 2 会靠 “右”，偶数位的 2 会靠 “左”)。

当然，如果直接排序，时间复杂度为

O(nlogn)。考虑上面的分半，其实不需要使整个数组有序，只要得到一个数组，使得其可以分为三部分：[>mid, =mid, < mid]，此时，将数组平分，即可得到：

1 right: [=mid, <mid] //相等的部分不会相交
2 left: [>mid, =mid]

接着就可以按照之前的做法得到结果。
这里，使用前面提到的快速选择算法来在
O(n) 内完成该问题。使用快速选择来找到中位数，然后，依据得到的中位数，将数组划分为 [>mid, =mid, < mid] 的形式，这里使用三路合并的方法（和两路合并的区别在于还要单独考虑等于中位数的部分）。
得到 [>mid, =mid, < mid] 后，使用之前描述的算法得到最终的结果，下面给出大致实现:

```
var wiggleSort = function(nums) {
   const n = nums.length;
   const geti = i => (1 + 2 * i) % (n | 1);
   const mid = qs(nums, 0, n - 1, (n + 1) >> 1);
   let i = 0, j = 0, k = n - 1;
   while (j <= k) {
       const ii = geti(i), ji = geti(j), ki = geti(k);
       if (nums[ji] > mid) {
           [nums[ii], nums[ji]] = [nums[ji], nums[ii]];
           i++; j++;
       } else if (nums[ji] < mid) {
           [nums[ji], nums[ki]] = [nums[ki], nums[ji]];
           k--;
       } else {
           j++;
       }
   }
};
function qs(nums, s, e, k) {
   let i = s, j = e;
   const rk = Math.floor(s + Math.random() * (e - s + 1));
   [nums[s], nums[rk]] = [nums[rk], nums[s]];
   const p = nums[s];
   while (i < j) {
       while (i < j && nums[j] >= p) j--;
       nums[i] = nums[j];
       while (i < j && nums[i] <= p) i++;
       nums[j] = nums[i];
   }
   nums[i] = p;
   const pk = i - s + 1;
   if (pk == k) return nums[i];
   if (pk > k) return qs(nums, s, i - 1, k);
```

```
    else return qs(nums, i + 1, e, k - pk);
}
```

里为了实现 O(1) 的空间复杂度，使用了评论区学到的所谓 虚拟索引 的技术，即将数组的一个索引映射为实际数组中的两一个索引，

```
1    0 1 2 3 4 5 6 7 8 9
2    5 0 6 1 7 2 8 3 9 4
```

即对 0 索引操作时，实际上对数组的 5 索引数据进行操作，这样我们在虚拟空间将数组整理为 [>mid, =mid, < mid] 形式后，同时也将较小的数放在了偶数位，加大的数放在了奇数位。当然这不是本文的重点，这里不再展开。

小结

本文介绍了快速排序（QuickSort）算法及其衍生算法快速选择（QuickSelection）算法，并以 LeetCode.324 为例说明了快速选择算法的应用。使用快速选择算法，可以使一些需要 O(nlogn) 实现的算法可以在 avgO(n) 下实现，例如本文提到的，求第 k 小的数等问题。