29 寻找两个有序数组的中位数

更新时间：2019-10-08 12:44:12

> 人生的旅途，前途很远，也很暗。然而不要怕，不怕的人的面前才有路。
>
> —— 鲁 迅

刷题内容

> **难度: Hard**
>
> 原题连接：https://leetcode-cn.com/problems/median-of-two-sorted-arrays/

内容描述

给定两个大小为 m 和 n 的有序数组 nums1 和 nums2。

请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

你可以假设 nums1 和 nums2 不会同时为空。

示例 1:

nums1 = [1, 3]
nums2 = [2]

则中位数是 2.0

示例 2:

nums1 = [1, 2]
nums2 = [3, 4]

则中位数是 (2 + 3)/2 = 2.5

题目详解

- 题目给我们两个数组，并且两个数组都是有序的，一般没说逆序就是顺序的，也就是升序的；
- 中位数有两种情况，如果数组长度为奇数的话，那中位数就是最中间的那一个数，否则为最中间的那两个数的平均数。

解题方案

思路 1:时间复杂度: O((m+n) * lg(m+n)) 空间复杂度: O(m+n)

首先最简单粗暴的方法，就是我们将两个数字列表合并起来，排好序，找到中间的 `median` 就ok了，但是千万要注意一点，如果 `median` 有两个，需要算平均。下面来看代码：

Python beats 99.95%

```python
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        nums = sorted(nums1 + nums2)
        if len(nums) % 2 == 1: # 如果数组长度为奇数的话，那中位数就是最中间的那一个数
            return nums[len(nums)//2]
        else: # 否则为最中间的那两个数的平均数
            return (nums[len(nums)//2-1] + nums[len(nums)//2]) / 2.0
```

Go beats 57.92%

```go
func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
    nums := append(nums1, nums2...)
    sort.Ints(nums)
    if len(nums) % 2 == 1 { // 如果数组长度为奇数的话，那中位数就是最中间的那一个数
        return float64(nums[len(nums)/2])
    } else { // 否则为最中间的那两个数的平均数
        return float64(nums[len(nums)/2-1] + nums[len(nums)/2]) / float64(2)
    }
}
```

Java beats 64.75%

```java
import java.util.Arrays;
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        // 创建新数组存储 nums1 和 nums2
        int[] nums = Arrays.copyOf(nums1, nums1.length + nums2.length);
        // 将 nums2 复制到新创建的数组 nums 中
        System.arraycopy(nums2,0,nums,nums1.length, nums2.length);
        // 数组排序
        Arrays.sort(nums);
        if (nums.length % 2 == 0) {
            // 新数组的长度如果是偶数，则需要找到中间的两个数进行取平均数
            int n1 = nums[nums.length / 2 - 1];
            int n2 = nums[nums.length / 2];
            return (double)(n1 + n2) / 2;
        } else {
            // 否则直接就是最中间的数
            return nums[nums.length / 2];
        }
    }
}
```

C++ beats 90.65%

```cpp
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        vector<int> data;
        for (int i = 0;i < nums1.size();i++){
            data.push_back(nums1[i]);
        }
        for (int i = 0;i < nums2.size();i++){
            data.push_back(nums2[i]);
        }
        sort(data.begin(), data.end());
        if (data.size() % 2 == 1) {
            return data[data.size() / 2];
        } else {
            return ((double)data[data.size() / 2 - 1] + data[data.size() / 2]) / 2;
        }
    }
};
```

思路 2：时间复杂度: O(lg(m+n)) 空间复杂度: O(1)

这时候我们观察到题目给的一个条件， `nums1` 和 `nums2` 本身也是有序的，放着这个条件不用反而用思路1是不是有点浪费了？换句话说我们没必要把它们整个排序，于是我们可以把它转化成经典的 **findKth问题**（不了解的同学可以去百度一下）。

首先转成求 `A` 和 `B` 数组中第 `k` 小的数的问题, 然后用 `k//2` 在 `A` 和 `B` 中分别进行查找。

比如 `k = 6`, 分别看 `A` 和 `B` 中的第 `3` 个数, 已知 `A1 <= A2 <= A3 <= A4 <= A5...` 和 `B1 <= B2 <= B3 <= B4 <= B5...`, 如果 `A3 <= B3`, 那么第 `6` 小的数肯定不会是 `A1, A2, A3`, 因为最多有两个数小于 `A1` (B1, B2), 三个数小于 `A2` (A1, B1, B2), 四个数小于 `A3` (A1, A2, B1, B2)。 关键点是从 `k//2` 开始来找。那就可以排除掉A1, A2, A3, 转成求 `A4, A5, ... B1, B2, B3, ...` 这些数中第 `3` 小的数的问题, `k` 就被减半了。

当 `k == 1` 或某一个数组空了, 这两种情况都是终止条件，下面我们来看代码：

Python beats 98.28%

```python
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        def findKth(A, pa, B, pb, k):
            res = 0
            m = 0
            while pa < len(A) and pb < len(B) and m < k:
                if A[pa] < B[pb]:
                    res = A[pa]
                    pa += 1
                else:
                    res = B[pb]
                    pb += 1
                m += 1

            while pa < len(A) and m < k:
                res = A[pa]
                pa += 1
                m += 1

            while pb < len(B) and m < k:
                res = B[pb]
                pb += 1
                m += 1

            return res

        n = len(nums1) + len(nums2)
        if n % 2 == 1:
            return findKth(nums1, 0, nums2, 0, n // 2 + 1)
        else:
            smaller = findKth(nums1, 0, nums2, 0, n // 2)
            bigger = findKth(nums1, 0, nums2, 0, n // 2 + 1)
            return (smaller + bigger) / 2.0
```

Go beats 100.00%

```go
func findKth(A []int, B []int, k int) int {
    if len(A) == 0 { // A 为空，第 k 小的数就是 B 中第k个数
        return B[k-1]
    }
    if len(B) == 0 { // B 为空，第 k 小的数就是 A 中第k个数
        return A[k-1]
    }
    if k == 1 { // k 为 1 就是求最小的数
        return int(math.Min(float64(A[0]), float64(B[0])))
    }
    var a, b int
    if len(A) >= k / 2 {
        a = A[k/2-1]
    } else {
        a = math.MaxInt32
    }
    if len(B) >= k / 2 {
        b = B[k/2-1]
    } else {
        b = math.MaxInt32
    }

    // 如果 A 总共都没有 k // 2 个数字，那么 B 中 前 k // 2 个数字肯定在前 k 个数字中（升序排序）
    // 令 i == k // 2，  n < k//2
    // 再令 B1 <= B2 <= B3 ... Bi

    //                A1 <= A2 <= A3 ... An,
    //         B1 -> Bi
    //           .... B1 -> Bi
    //                ..... B1 -> Bi
    //                      ..... B1 -> Bi
    //                            ..... B1 -> Bi
    // 我们发现 B1 -> Bi 窗口无论怎么滑动，他们永远在前 k 个数字中

    if a == math.MaxInt32 { // 这里要注意：因为 k//2 不一定等于 (k - k//2)
        return findKth(A, B[k/2:], k - k/2)
    }
    if b == math.MaxInt32 {
        return findKth(A[k/2:], B, k - k/2)
    }

    if a < b {
        return findKth(A[k/2:], B, k - k/2)
    } else {
        return findKth(A, B[k/2:], k - k/2)
    }
    if b == math.MaxInt32 || (a != math.MaxInt32 && a < b) {
        return findKth(A[k/2:], B, k - k/2)
    }
    return findKth(A, B[k/2:], k - k/2)
}

func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
    n := len(nums1) + len(nums2)
    if n % 2 == 1 {
        return float64(findKth(nums1, nums2, n / 2 + 1))
    } else {
        smallerMedian := findKth(nums1, nums2, n / 2)
        biggerMedian := findKth(nums1, nums2, n / 2 + 1)
        return float64(smallerMedian + biggerMedian) / float64(2)
    }
}
```

思路 3：时间复杂度: O(lg(m+n)) 空间复杂度: O(1)

上面那个思路，我们需要循环k次，每次求当前的最小值，跳过它，并修改下标。为什么每次只跳过一个呢？不能一次性跳过k-1个，再求下一个呢？

考虑一下，为了跳过 nums1 和 nums2 合并后的 p 个，我们拿 nums1 数组中的 a 个，和 nums2 数组中的 b 个出来比较，这里 a+b=p。

如 果 `nums1[i+a-1]` (也就是当前位置的第a个数)小于`nums2[j+b-1]`(也就是当前位置的第b个数)，那么`nums1[i+a-1]`最极端的情况，只可能是合并后的数组的第p-1个，所以这种情况，就需要 nums1 数组跳过a个数

那么a和b的取值应该是多少呢？运气不好的情况下，每次只会跳过min(a,b)个数，那么最大化min(a,b)就是我们需要的取值，即p=k//2

解法：

- 如果要跳过1个数，直接比较下一个数；
- 要跳过p个数，先让nums1和nums2尽量跳过p/2个数；
- 如果nums1剩下的不够p/2个数，则nums1剩下的整个数组跟nums2后面p-len(nums1)个数进行比较；
- 如果nums2剩下的不够p/2个数，则nums2剩下的整个数组跟nums1后面p-len(nums2)个数进行比较。

求第k个数就是跳过k-1个数，求下一个。下面我们来看代码：

Python beats 98.28%

```python
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        def findKth(A, B, k):
            if len(A) == 0: # A 为空，第 k 小的数就是 B 中第k个数
                return B[k-1]
            if len(B) == 0: # B 为空，第 k 小的数就是 A 中第k个数
                return A[k-1]
            if k == 1: # k 为 1 就是求最小的数
                return min(A[0], B[0])

            a = A[k // 2 - 1] if len(A) >= k // 2 else None
            b = B[k // 2 - 1] if len(B) >= k // 2 else None

            # 如果 A 总共都没有 k // 2 个数字，那么 B 中 前 k // 2 个数字肯定在前 k 个数字中（升序排序）
            # 令 i == k // 2， n < k//2
            # 再令 B1 <= B2 <= B3 ... Bi
            #
            #                   A1 <= A2 <= A3 ... An,
            #          B1 -> Bi
            #            .... B1 -> Bi
            #                  ..... B1 -> Bi
            #                              ..... B1 -> Bi
            #                                          ..... B1 -> Bi
            # 我们发现 B1 -> Bi 窗口无论怎么滑动，他们永远在前 k 个数字中

            if a is None:
                return findKth(A, B[k // 2:], k - k // 2) # 这里要注意：因为 k//2 不一定等于 (k - k//2)
            if b is None:
                return findKth(A[k // 2:], B, k - k // 2)
            if a < b:
                return findKth(A[k // 2:], B, k - k // 2)
            else:
                return findKth(A, B[k // 2:], k - k // 2)


            if b is None or (a is not None and a < b):
                return findKth(A[k // 2:], B, k - k // 2)
            return findKth(A, B[k // 2:], k - k // 2)

        n = len(nums1) + len(nums2)
        if n % 2 == 1:
            return findKth(nums1, nums2, n // 2 + 1)
        else:
            smaller_median = findKth(nums1, nums2, n // 2)
            bigger_median = findKth(nums1, nums2, n // 2 + 1)
            return (smaller_median + bigger_median) / 2.0
```

Go beats 100.00%

```go
func findKth(A []int, B []int, k int) int {
 if len(A) == 0 { // A 为空，第 k 小的数就是 B 中第k个数
  return B[k-1]
 }
 if len(B) == 0 { // B 为空，第 k 小的数就是 A 中第k个数
  return A[k-1]
 }
 if k == 1 { // k 为 1 就是求最小的数
  return int(math.Min(float64(A[0]), float64(B[0])))
 }
 var a, b int
 if len(A) >= k / 2 {
  a = A[k/2-1]
 } else {
  a = math.MaxInt32
 }
 if len(B) >= k / 2 {
  b = B[k/2-1]
 } else {
  b = math.MaxInt32
 }

 // 如果 A 总共都没有 k // 2 个数字，那么 B 中 前 k // 2 个数字肯定在前 k 个数字中（升序排序）
 // 令 i == k // 2，  n < k//2
 // 再令 B1 <= B2 <= B3 ... Bi

 //                 A1 <= A2 <= A3 ... An,
 //          B1 -> Bi
 //            .... B1 -> Bi
 //                  ..... B1 -> Bi
 //                        ..... B1 -> Bi
 //                              ..... B1 -> Bi
 // 我们发现 B1 -> Bi 窗口无论怎么滑动，他们永远在前 k 个数字中

 if a == math.MaxInt32 { // 这里要注意：因为 k//2 不一定等于 (k - k//2)
  return findKth(A, B[k/2:], k - k/2)
 }
 if b == math.MaxInt32 {
  return findKth(A[k/2:], B, k - k/2)
 }

 if a < b {
  return findKth(A[k/2:], B, k - k/2)
 } else {
  return findKth(A, B[k/2:], k - k/2)
 }
 if b == math.MaxInt32 || (a != math.MaxInt32 && a < b) {
  return findKth(A[k/2:], B, k - k/2)
 }
 return findKth(A, B[k/2:], k - k/2)
}

func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
 n := len(nums1) + len(nums2)
 if n % 2 == 1 {
  return float64(findKth(nums1, nums2, n / 2 + 1))
 } else {
  smallerMedian := findKth(nums1, nums2, n / 2)
  biggerMedian := findKth(nums1, nums2, n / 2 + 1)
  return float64(smallerMedian + biggerMedian) / float64(2)
 }
}
```

C++ beats 99.68%

```cpp
class Solution {
public:
```

```
// i: nums1数组目前的位置
// j: nums2数组目前的位置
// k: 跳过最小的k个
//
// 跳过最小的k个，i和j表示跳过k个后，目前nums1和nums2的位置
void filterK(vector<int>& nums1, vector<int>& nums2, int &i, int &j, int k) {
    while (k > 0) {
        if (i == nums1.size()) {
            // 当i已经过了nums1的最后元素，说明只剩下nums2的元素，直接走nums2，k个元素
            j += k;
            k = 0;
        } else if (j == nums2.size()) {
            // 当j已经过了nums2的最后元素，说明只剩下nums1的元素，直接走nums1，k个元素
            i += k;
            k = 0;
        } else if (k == 1) {
            // 当k=1，说明跳过下一个数，则比较下一个数的大小，小的跳过
            if (nums1[i] < nums2[j]) {
                i++;
            } else {
                j++;
            }
            k = 0;
        } else {
            //a表示nums1的当前步长，b表示nums2的当前步长
            //为了跳过尽量多的数，我们初始化步长为k/2
            //如果有哪个数组没有剩下这么多数，则步长设定为当前的长度，另一个数组同步修改
            //对比两个数组最后一个数，小的那一组按照它的步长走
            int a = k / 2, b = k - k / 2;
            if (i + a - 1 >= nums1.size()) {
                a = nums1.size() - i;
                b = k - a;
            }
            if (j + b - 1 >= nums2.size()) {
                b = nums2.size() - j;
                a = k - b;
            }
            if (nums1[i + a - 1] < nums2[j + b - 1]) {
                i += a;
                k -= a;
            } else {
                j += b;
                k -= b;
            }
        }
    }
}
// i: nums1数组目前的位置
// j: nums2数组目前的位置
//
// 跳过一个数，并返回这个被跳过的数的值
double next(vector<int>& nums1, vector<int>& nums2, int &i, int &j) {
    if (i == nums1.size()) {
        return nums2[j++];
    } else if (j == nums2.size()) {
        return nums1[i++];
    } else if (nums1[i] < nums2[j]) {
        return nums1[i++];
    } else {
        return nums2[j++];
    }
}
double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    int n = nums1.size() + nums2.size();
    int i = 0 , j = 0;
    if (n & 1) {
        //n为奇数，需要求位置为n/2+1的数(下标从0开始)
        //跳过n/2个，返回下一个即可
        filterK(nums1, nums2, i, j, n / 2);
```

```
            return next(nums1, nums2, i, j);
        } else {
            //n为偶数，需要求位置为n/2的数和位置为n/2+1的数(下标从0开始)
            //跳过n/2-1个，返回下两个的中位数即可
            filterK(nums1, nums2, i, j, n / 2 - 1);
            return (next(nums1, nums2, i, j) + next(nums1, nums2, i, j)) / 2;
        }
    }
};
```

总结

时刻要注意问题的分解，我们可以把这道题转化成一些我们已经知道解法的经典问题上面去。

对于不同的实现方式，我们都可以去尝试一下。

}

28 摆动排序

30 正则表达式匹配