Siyao's Blog

【九章算法基础班】二分法

2017-12-07 | 算法，九章算法 | 浏览 次

5,543 | 28

二分查找

classical Binary Search

定义

给定一个排序数组和一个元素n，返回元素n的位置

index	0	1	2	3	4	5	6	7	8
num	2	3	5	8	13	21	34	55	89

查找元素5的位置

方法

初始化:

start = 0;end = 8;mid = 4

1. nums[mid] = 13;start = 0;end = 4,mid = 2
2. nums[mid] = 5;find it!

时间复杂度

```
数据规模为n:
T(n) = T(n/2)+O(1)
其中O(1)为比较的时间复杂度，T(n/2)为比较之后
时间复杂度是:O(logn)
```

实现方式

1. 递归:
   - 优点：代码简洁
   - 缺点：递归利用栈空间，递归层数过多会导致栈溢出
2. while循环
   - 优点：占用空间小
   - 缺点：代码可读性稍差，不够简洁

   面试的时候用什么?

   如果用递归的方式写会好理解很多，就用递归写，不然就不用递归，在工程上，递归很容易导致栈溢出。

这道题用最好用非递归的方式写，因为是很简短的

通用模板

- start + 1 < end
- mid = start +(end - start)/2;如果用(start + end)/2，如果start和end都很大相加就有可能溢出
- A[mid] = < > 三种情况讨论
- A[start] A[end]?target

```
public int findPosition(int[] nums, int target) {
        if (nums.length <= 0){
            return -1;
        }
        // write your code here
        int start = 0;
        int end = nums.length - 1;
        while (start + 1 < end) {
            //中值
            int mid = start + (end - start) / 2;
            //三种情况讨论
            if (nums[mid] == target) {
                end = mid;
            }
            if (nums[mid] < target) {
                start = mid;
            }
            if (nums[mid] > target) {
                end = mid;
            }
        }

        //结果
        if (nums[start] == target) {
            return start;
        }
        if (nums[end] == target) {
            return end;
        } else {
            return -1;
        }
    }
```

相关题目

Search for a Range

题目

> Given an array of integers sorted in ascending order, find the starting and ending position of a given target value.
>
> Your algorithm's runtime complexity must be in the order of O(log *n*).
>
> If the target is not found in the array, return `[-1, -1]` .
>
> For example,
> Given `[5, 7, 7, 8, 8, 10]` and target value 8,
> return `[3, 4]` .

分析

需要分别找到n**第一次出现的位置**和**最后一次出现的位置**，返回即可。

代码

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] result = {-1,-1};
        if(nums.length <=0){
            return result;
        }
```

```
        int start = 0;
        int end = nums.length -1;
        //寻找第一次出现的位置
        while (start + 1 < end){
            int mid = start +(end-start)/2;
            if(nums[mid] == target){
                //第一次出现的位置，在前半段，end前移
                end = mid;
            }
            if(nums[mid] < target){
                start = mid;
            }
            if(nums[mid] > target){
                end = mid;
            }
        }
        if(nums[start] == target){
            result[0] = start;
        }
        else if(nums[end] == target){
            result[0] = end;
        }
        start = 0;
        end = nums.length -1;
        //寻找最后一次出现的位置
        while (start + 1 < end){
            int mid = start +(end-start)/2;
            if(nums[mid] == target){
                //第一次出现的位置，在后半段，start后移
                start = mid;
            }
            if(nums[mid] < target){
                start = mid;
            }
            if(nums[mid] > target){
                end = mid;
            }
        }
        if(nums[end] == target){
            result[1] = end;
        }
        else if(nums[start] == target){
            result[1] = start;
        }
        return result;
    }
}
```

Search Insert Position

题目

> Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
>
> You may assume no duplicates in the array.
>
> **Example 1:**
>
> ```
> > Input: [1,3,5,6], 5
> > Output: 2
> >
> >
> ```
>
> **Example 2:**
>
> ```
> > Input: [1,3,5,6], 2
> > Output: 1
> >
> >
> ```
>

> **Example 3:**
>
> ```
> > Input: [1,3,5,6], 7
> > Output: 4
> >
> >
> ```
>

> **Example 1:**
>
> ```
> > Input: [1,3,5,6], 0
> > Output: 0
> >
> ```

分析

二分法的问题一般都是找满足条件的firstposition和lastposition.

这道题是需要找到firstposition >= targrt，第一个>=target的位置。

while结束后，需要找firstposition的话先判断start，找lastposition的话先判断end。

代码

```
class Solution {
        public int searchInsert(int[] nums, int target) {
        if(nums.length <=0){
            return -1;
        }
        int start = 0;
        int end = nums.length -1;
        //寻找第一次出现的位置
        while (start + 1 < end){
            int mid = start +(end-start)/2;
            //找到了
            if(nums[mid] == target){
                return mid;
            }
            if(nums[mid] < target){
                start = mid;
            }
            if(nums[mid] > target){
                end = mid;
            }
        }
        //没找到target,在start和end中找比target大的
        //start>=tartget
        if(nums[start] >= target){
            return start;
        }
        //end>=tartget
        else if (nums[end] >= target){
            return end;
        }
        //nums中所有元素都比target小
        else{
            return end+1;
        }
    }
}
```

Search a 2D Matrix

题目

> Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:
>
> - Integers in each row are sorted from left to right.
> - The first integer of each row is greater than the last integer of the previous row.

> For example,
>
> Consider the following matrix:
>
> ```
> [
>   [1,   3,  5,  7],
>   [10, 11, 16, 20],
>   [23, 30, 34, 50]
> ]
> ```

>

> Given **target** = `3`, return `true`.

分析

两种做法：

1. 先按每行的首个数字二分确定target所在的行，再在这行里二分。
2. 看成一维数组，二分查找，其中看成一维数组的序号n和二维数组中行号、列号对应关系为：

   r = n/columns;

   c = n%columns;

代码

```
public class SearchA2DMatrix {
    public boolean searchMatrix(int[][] matrix, int target) {
        //空矩阵判断
        int rows = matrix.length;
        if(rows <= 0){
            return  false;
        }
        int cols = matrix[0].length;
        if(cols <= 0){
            return  false;
        }
        int start = 0;
        int end = rows*cols -1;
        while(start+1 <end){
            int mid = start+(end-start)/2;
            int r = mid/cols;
            int c = mid%cols;
            if (matrix[r][c] == target){
                return true;
            }
            if(matrix[r][c] < target){
                start = mid;
            }
            if(matrix[r][c] > target){
                end = mid;
            }
        }
        if(matrix[start/cols][start%cols]==target){
            return true;
        }
        if(matrix[end/cols][end%cols]==target){
            return true;
        }
        return false;
    }
}
```

Search a 2D Matrix II

题目

> Write an efficient algorithm that searches for a value in an *m* x *n* matrix. This matrix has the following properties:
>
> - Integers in each row are sorted in ascending from left to right.
> - Integers in each column are sorted in ascending from top to bottom.
>
> For example,
>
> Consider the following matrix:
>
> ```
> [
>   [1,   4,  7, 11, 15],
>   [2,   5,  8, 12, 19],
>   [3,   6,  9, 16, 22],
>   [10, 13, 14, 17, 24],
>   [18, 21, 23, 26, 30]
> ]
> ```

>

> Given **target** = `5`, return `true`.
>
> Given **target** = `20`, return `false`.

分析

- 方法一：

从对角线开始二分找， 找到第一个>=target的位置，将大矩形分割成4个小矩形，分两种情况讨论：

1. 元素 = target，返回true
2. 元素 > target，左上角的矩阵中元素 < target，右下角矩阵中的元素都 > target，都不可能有符合条件的值了，所以只需继续在左下角和右上角的矩阵中继续寻找。

时间复杂度分析：

$T(n) = 2T(n/4) + O(log\sqrt{n})$，其中$\sqrt{n}$为对角线元素个数。

- 方法二：

  矩阵特点：每一行和每一列递增

  从左下角往右上角找，有如下三种情况：

1. 元素 = target，返回true
2. 元素 < target，下一步向右走，因为右边的元素都大于当前元素，上方元素小于当前元素
3. 元素 > target，下一步向上走，因为右边的元素都大于当前元素，上方元素小于当前元素

代码

```
public boolean searchMatrix(int[][] matrix, int target) {
        int rows = matrix.length;
        if(rows <= 0){
            return false;
        }
        int cols = matrix[0].length;
        if(cols <= 0){
            return false;
        }
        int r = rows-1;
        int c = 0;
        while(r >=0 && c <cols){
            if(matrix[r][c]==target){
                return  true;
            }
```

```
            else if(matrix[r][c] < target){
                c++;
            }
            else if(matrix[r][c] > target){
                r--;
            }
        }
        return false;
```

First Bad Version

题目

> You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.
>
> Suppose you have `n` versions `[1, 2, ..., n]` and you want to find out the first bad one, which causes all the following ones to be bad.
>
> You are given an API `bool isBadVersion(version)` which will return whether `version` is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

分析

有1到n个版本，找到第一个错误版本的位置。利用从第一个错误的开始之后后面的都是错的，用二分查找到第一个错误版本。

代码

```
public int firstBadVersion(int n) {
    int start = 1;
    int end = n;
    while(start + 1 < end){
        int mid = start+(end-start)/2;
        if(isBadVersion(mid)){
            end = mid;
        }
        else{
            start = mid;
        }
    }
    if(isBadVersion(start)){
        return start;
    }
    if(isBadVersion(end)){
        return end;
    }
    return -1;
}
```

Find Peak Element

题目

> A peak element is an element that is greater than its neighbors.
>
> Given an input array where `num[i] ≠ num[i+1]`, find a peak element and return its index.
>
> The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.
>
> You may imagine that `num[-1] = num[n] = -∞`.
>
> For example, in array `[1, 2, 3, 1]`, 3 is a peak element and your function should return the index number 2.

分析

给定一个数组，返回peak的元素（左边右边都比自己小），如果有多个返回任意一个即可，要求时间复杂度为O(logn)

遇到O(logn)要考虑到二分法：

二分法选取mid元素，其跟左右元素的关系有如下四中情况：

1. 两边元素都比mid小，mid就是peak，返回
2. 两边元素都比mid大，左右都有可能有peak，任选一边二分
3. 左边小又边大，右边一定有peak，继续对右边二分
4. 左边大右边小，左边一定有peak，继续对左边二分

需要注意的地方：

因为要将元素跟其左边和右边的元素比较，所以为了**避免越界，初始需要将start设为1，end设为length-1**；还需要把两个边界值设为负无穷，确保边界值也可以被选上。

代码

```
public int findPeakElement(int[] nums) {
    if(nums.length == 1){
        return 0;
    }
    long[] nums2 = new long[nums.length+2];
    nums2[0] = -Long.MAX_VALUE;
    nums2[nums.length+1] = -Long.MAX_VALUE;
    for(int i = 0;i < nums.length;i++){
        nums2[i+1] = nums[i];
    }
    //遇到这种需要判断元素左右的将start设为1，end设为len-2,放置越界
    int start = 1;
    int end = nums2.length-1;
    while(start+1<end){
        int mid = start + (end-start)/2;
        if((nums2[mid] > nums2[mid+1]) && (nums2[mid] > nums2[mid-1])){
            return mid-1;
        }
        else if((nums2[mid] < nums2[mid+1]) && (nums2[mid] < nums2[mid-1])){
            start = mid;
        }
        else if ((nums2[mid] > nums2[mid+1]) && (nums2[mid] < nums2[mid-1])){
            end = mid;
        }
        else {
            start = mid;
        }
    }
    if(nums2[start] < nums2[end]){
        return  end-1;
    }
    else
        return start-1;
}
```

Find Minimum in Rotated Sorted Array

题目

> Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
>
> (i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).
>
> Find the minimum element.
>
> You may assume no duplicate exists in the array.

分析

递增数组：

```
↗
旋转数组：
旋转数组，分成两段上升数组：
              ↗|
            ↗  |
          ↗    |
        ↗      |
      ↗        |
----------------|------------------
                |    ↗
                |  ↗
                |↗
      ↑     ↑        ↑
    start  mid      end
用二分法，判断mid与end的大小，确定mid位于两段上升数组的哪一段：
1. nums[mid] < nums[end]:
       mid在后段上升数组中，end = mid
2. nums[mid] > nums[end]:
       mid在前段上升数组中，start = mid
```

代码

```
public int findMin(int[] nums) {
  int start = 0;
  int end = nums.length-1;
  while (start+1<end){
    int mid = start+(end-start)/2;
    if(nums[mid] < nums[end]){
      end = mid;
    }
    if(nums[mid] > nums[end]){
      start = mid;
    }
  }
  if(nums[start] > nums[end]){
    return nums[end];
  }
  else return nums[start];
}
```

Find Minimum in Rotated Sorted Array II

题目

> *Follow up* for "Find Minimum in Rotated Sorted Array":
> What if *duplicates* are allowed?
>
> Would this affect the run-time complexity? How and why?

Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.

(i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).

Find the minimum element.

The array may contain duplicates.

延续上一题，有重复数字的旋转数组，找到最小的值。

分析

如果存在重复元素，无法使用二分查找使得时间复杂度为O(logn)，最坏时间复杂度只能是o(n)。

证明方法：**黑盒测试**

假设给定的数组中有一个1和n-1个2，此时mid=2，无法判断1在哪边。

代码


```
public int findMin(int[] nums) {
  int min = Integer.MAX_VALUE;
  for(int i = 0; i < nums.length;i++){
    if(nums[i] < min){
      min = nums[i];
    }
  }
  return min;
}
```

Search in Rotated Sorted Array

题目

> Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand.
>
> (i.e., `0 1 2 4 5 6 7` might become `4 5 6 7 0 1 2`).
>
> You are given a target value to search. If found in the array return its index, otherwise return -1.
>
> You may assume no duplicate exists in the array.

给定一个旋转递增数组和一个数，返回数在数组中的位置。

分析

二分法

```
递增数组：
            ↗
          ↗
        ↗
      ↗
    ↗
  ↗
↗

旋转数组，分成两段上升数组：
              ↗|
            ↗  |
          ↗    |
        ↗      |
      ↗        |
----------------|------------------
                |    ↗
                |  ↗
                |↗
      ↑     ↑        ↑
    start  mid      end
二分法，mid和end比较：
1. nums[mid] > nums[end]:
       mid在第一段上升区间，跟target比较：
              1. 如果target在start和mid之间，end= mid
              2. 否则，start = mid,继续做二分查找，仍然是一个search in rotated sorted array问题
2. nums[mid] < nums[end]:
       mid在第二段上升区间，跟target比较：
              1. 如果target在mid和end之间，end= mid
              2. 否则，start = mid,继续做二分查找，仍然是一个search in rotated sorted array问题
```

代码

```
public class SearchInRotatedSortedArray {
    public int search(int[] nums, int target) {
        if(nums.length <= 0){
            return -1;
        }
        int start = 0;
        int end = nums.length-1;
        while(start+1 < end){
            int mid = start+(end-start)/2;
            if(target == nums[mid]){
                return mid;
```

```
            }
            //mid在第一个上升区间
            if(nums[mid] > nums[end]){
                if(target < nums[mid] && target >= nums[start]){
                    end = mid;
                }
                else{
                    start = mid;
                }
            }
            ///mid在第二个上升区间
            else{
                if(target > nums[mid] && target <= nums[end]){
                    start = mid;
                }
                else {
                    end = mid;
                }
            }
        }
        if(nums[start] == target){
            return start;
        }
        else if(nums[end] == target){
            return end;
        }
        else return -1;
    }
}
```

Search in Rotated Sorted Array II

有重复元素，无法二分查找，时间复杂度为O(n)，遍历查找即可。

Merge Sorted Array

题目

> Given two sorted integer arrays *nums1* and *nums2*, merge *nums2* into *nums1* as one sorted array.
>
> **Note:**
>
> You may assume that *nums1* has enough space (size that is greater or equal to *m* + *n*) to hold additional elements from *nums2*. The number of elements initialized in *nums1* and *nums2* are *m* and *n* respectively.

分析

合并连个有序数组，把S2并入S1，假设S1有足够大的空间。

考点：从后往前合并，因为后面的空间是空的，不会覆盖原有元素。

代码

```
public class MergeSortedArray {
    public void merge(int[] nums1, int m, int[] nums2, int n) {
        int end1 = m-1;
        int end2 = n-1;
        int end = m+n-1;

        while (end1 >=0 && end2>=0) {
            if(nums1[end1] < nums2[end2]){
                nums1[end] = nums2[end2];
                end2--;
            }
            else {
                nums1[end] = nums1[end1];
                end1--;
            }
            end--;
        }
        //nums1元素剩下了
        if (end1 >= 0) {
        }
```



```
        //nums2元素剩下了
        else if (end2 >=0){
            while (end2 >= 0){
                nums1[end] = nums2[end2];
                end--;
                end2--;
            }
        }
    }
}
```

Median of Two Sorted Arrays

题目

> There are two sorted arrays **nums1** and **nums2** of size m and n respectively.
>
> Find the median of the two sorted arrays. The overall run time complexity should be O(log (m+n)).
>
> **Example 1:**
>
> ```
> > nums1 = [1, 3]
> > nums2 = [2]
> >
> > The median is 2.0
> >
> >
> ```
>
> >
>
> **Example 2:**
>
> ```
> > nums1 = [1, 2]
> > nums2 = [3, 4]
> >
> > The median is (2 + 3)/2 = 2.5
> >
> ```

分析

令nums1.length = m;nums2.length = n;总长度m+n。先确定中位数m是多少:

1. (m+n)%2 == 0:

   m=(m+n)/2,(m+n)/2+1

2. (m+n)%2 == 1:

   m=(m+n)/2

问题转化为找**两个有序数组的第K大**的问题，如果用merge的方法获得merge之后的排序数组需要O(m+n)的时间复杂度，题目要求使用O(log(m+n))的时间复杂度，所以需要使用二分法：

如果A[k/2] <= B[k/2]：A的前k/2个数一定都在A、B合并后的前K个数中，去掉A的前k/2个元素，继续寻找A和B的第m-k/2个元素。

如果A[k/2] > B[k/2]：B的前k/2个数一定都在A、B合并后的前K个数中，去掉B的前k/2个元素，继续寻找A和B的第m-k/2个元素。

如果A[k/2]越界，A中剩余元素不足k/2个，则B中前k/2个元素一定在前K个中，去掉B的前k/2个元素，继续寻找A和B的第m-k/2个元素。

如果B[k/2]越界，B中剩余元素不足k/2个，则A中前k/2个元素一定在前K个中，去掉A的前k/2个元素，继续寻找A和B的第m-k/2个元素。

边界条件判断：

1. nums1中没有元素了,直接返回nums2中的第k个
2. nums2中没有元素了,直接返回nums1中的第k个

3. k == 1，递归出口，直接返回min(nums1[start1],nums2[start2])

代码

```
public class MedianOfTwoSortedArrays {
    //两个sorted array merge 寻找第k大的元素
    public int findKth(int[] nums1,int[] nums2,int k,int start1,int start2){
        int end1 = nums1.length - 1;
        int end2 = nums2.length - 1;

        if(k == 1){
            return Math.min(nums1[start1],nums2[start2]);
        }
        //nums1空了，
        if(start1 > end1){
            return nums2[start2 + k - 1];
        }
        //nums2空了
        if(start2 > end2){
            return nums1[start1 + k -1];
        }
        //nums1的前k/2个元素已经超过nums1中剩余的所有元素个数
        //那么nums2的前k/2个元素都包含在前k个元素中
        if(start1 + k/2 - 1 > end1){
            return findKth(nums1,nums2,k-k/2,start1,start2+k/2);
        }
        //nums2的前k/2个元素已经超过nums2中剩余的所有元素个数
        //那么nums1的前k/2个元素都包含在前k个元素中
        if(start2 + k/2 - 1 > end2){
            return findKth(nums1,nums2,k-k/2,start1+k/2,start2);
        }
        //nums和nums2的前k/2个元素都没有超过个字的末尾
        if(nums1[start1 + k/2 - 1] <nums2[start2+k/2-1]){
            return findKth(nums1,nums2,k-k/2,start1+k/2,start2);
        }
        else {
            return findKth(nums1,nums2,k-k/2,start1,start2+k/2);
        }
    }

    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int len1 = nums1.length;
        int len2 = nums2.length;
        int k = (len1+len2)/2;//中位数
        if((len1+len2)%2 == 0){
            int k1 = findKth(nums1,nums2,k,0,0);
            int k2 = findKth(nums1,nums2,k+1,0,0);
            return (double) (k1+k2)/2.0;
        }
        else{
            return findKth(nums1,nums2,k+1,0,0);
        }
    }

    public static void main(String[] args){
        MedianOfTwoSortedArrays test = new MedianOfTwoSortedArrays();
        int[] nums1 = {1,3};
        int[] nums2 = {2};
        double res = test.findMedianSortedArrays(nums1,nums2);
        System.out.println(res);
    }
}
```

Wood Cut

题目

> Given n pieces of wood with length `L[i]` (integer array). Cut them into small pieces to guarantee you could have equal or more than k pieces with the same length.
>
> What is the longest length you can get from the n pieces of wood? Given L & k, return the maximum length of the small pieces.
>
> **Notice**
>
> You couldn't cut wood into float length.

> If you couldn't get >= *k* pieces, return `0` .
>
> Example
>
> For `L=[232, 124, 456]` , `k=7` , return `114` .

分析

给定一些长度为L[i]的木料，将他们据成k段，返回满足条件的最大长度。

先找到木料中最长的那块，长度为m然后对m进行二分法，判断这些木料是否可以据成长度为mid，个数>=k个木料：

1. 如果可以，start = mid，进一步二分看是否可以据成更长的
2. 如果不可以，end = mid，进一步二分据成更短的小段

代码

```
public class Solution {
    /*
     * @param L: Given n pieces of wood with length L[i]
     * @param k: An integer
     * @return: The maximum length of the small pieces
     */
public int count(int[] L,int len){
        int sum = 0;
        for(int i = 0;i <L.length;i++){
            sum += L[i]/len;
        }
        return sum;
    }
    public int woodCut(int[] L, int k) {
        // 找到最长的wood，
        int max = 0;
        for(int i = 0 ; i < L.length;i++){
            if(L[i] > max){
                max = L[i];
            }
        }
        //对要切割的wood长度做二分法
        int start = 1;
        int end = max;
        while (start + 1 < end){
            int mid = start + (end - start)/2;
            if(count(L,mid) >= k){
                start = mid;
            }
            else {
                end = mid;
            }
        }
        if(count(L,end) >= k){
            return end;
        }
        if(count(L,start) >= k){
            return start;
        }
        return 0;
    }
}
```

Count of Smaller Numbers After Self

Sqrt(x)

题目

> Implement `int sqrt(int x)` .
>
> Compute and return the square root of *x*.
>
> **x** is guaranteed to be a non-negative integer.

> **Example 1:**
>
> ```
> > Input: 4
> > Output: 2
> >
> >
> ```
>
> >

> **Example 2:**
>
> ```
> > Input: 8
> > Output: 2
> > Explanation: The square root of 8 is 2.82842..., and since we want to return an integer, the decimal part will be truncated.
> >
> ```

分析

输入数字x，返回根号x，只保留整数部分

思路就是找到最后一个number k，满足条件k^2 < x，则k就是结果

代码

```
class Solution {
    public int mySqrt(int x) {
        if(x==0){
            return 0;
        }
        int start = 1;
        int end = x;
        while (start + 1 < end){
            int mid = start+(end-start)/2;
            if(mid * mid <= x){
                start = mid;
            }
            else {
                end = mid;
            }
        }
        if(end*end <= x){
            return end;
        }
        else {
            return start;
        }
    }
}
```

Rotate Array

> Rotate an array of *n* elements to the right by *k* steps.
>
> For example, with *n* = 7 and *k* = 3, the array `[1,2,3,4,5,6,7]` is rotated to `[5,6,7,1,2,3,4]`.

分析

三步翻转法：

1. 翻转左半段

   4,3,2,1,5,6,7

2. 翻转右半段

   4,3,2,1,7,6,5

3. 翻转整体

   5,6,7,1,2,3,4

代码

```
class Solution {
public void rotate(int[] nums, int k) {
        k = k % nums.length;
        int start = 0;
        int end = nums.length - k - 1;
        while(start < end){
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }

        start = nums.length - k;
        end = nums.length - 1;
        while(start < end){
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
        start = 0;
        end = nums.length-1;
        while(start < end){
            int temp = nums[start];
            nums[start] = nums[end];
            nums[end] = temp;
            start++;
            end--;
        }
    }
}
```

Reverse Words in a String II

题目

> Given an input string, reverse the string word by word. A word is defined as a sequence of non-space characters.
>
> The input string does not contain leading or trailing spaces and the words are always separated by a single space.
>
> For example,
>
> Given s = " `the sky is blue` ",
>
> return " `blue is sky the` ".
>
> Could you do it *in-place* without allocating extra space?

分析

三步翻转法

代码

```
public void reverseWords(char[] str) {
    int start = 0;
    int end = 0;
    while(end < str.length){
        while(end < str.length && str[end] != ' '){
            end++;
        }
        int stop = end-1;
        while(start <= stop){
            char temp = str[start];
            str[start] = str[stop];
            str[stop] = temp;
            start++;
```

```
            stop--;
        }
        start = end+1;
        end = end+1;
    }
    start = 0;
    end = str.length-1;
    while(start <= end){
        char temp = str[start];
        str[start] = str[end];
        str[end] = temp;
        start++;
        end--;
    }
}
```

Find the Duplicate Number

题目

> Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.
>
> **Note:**
>
> 1. You **must not** modify the array (assume the array is read only).
> 2. You must use only constant, O(1) extra space.
> 3. Your runtime complexity should be less than `O(n2)`.
> 4. There is only one duplicate number in the array, but it could be repeated more than once.

分析

方法一：

题目说有个n+1长的数组，里面的元素的范围在1~n，所以必定会有重复的元素，要求我们找出重复的元素。

1. baseline：

   遍历，时间复杂度$O(n^2)$，超时

2. 二分法：

   start = 1;end = n

   mid = (1+n)/2;

   遍历数组，记录值<=mid的元素个数sum，分两种情况讨论：

   a. sum <= mid，在1~mid之间的数组少于mid，说明重复数字在mid+1~len-1里，故令left = mid+1

   b. sum > mid，在1~mid之间的数组多于mid，说明重复数字在1~mid里，故令right = mid

方法二:

```
idx: 0  1       2       3
val: 1  2       3       2
实质：用下标idx的正负表示该数字idx是否出现过，如果出现过idx对应的数字为负数，否则为正数
从左向右遍历，把遇到的数字在数组中的下标标记为负值，当遇到下标对应的值已经为负数时，返回该下标：
i = 0 , val = 1,将nums[1]标记为负数
idx: 0  1       2       3
val: 1  -2      3       2
i = 1 , val = abs(-2) = 2 , 将nums[2]标记为负数
idx: 0  1       2       3
val: 1  -2      -3      2
i = 2 , val = abs(-3) = 3 , 将nums[3]标记为负数
idx: 0  1       2       3
val: 1  -2      -3      -2
```



```
i = 3 , val = abs(-2) = 2 , 此时nums[2] < 0, 则返回2
idx: 0  1       2       3
val: 1  -2      -3      -2
```

代码

```
public int findDuplicate(int[] nums) {
  int left = 1;
  int right = nums.length-1;
  while(left <right){
    int mid = left+(right-left)/2;
    int sum = 0;//记录在1~mid之间的数字有多少个
    for(int i:nums){
      if(i <= mid){
        sum++;
      }
    }
    //在1~mid之间的数组少于mid，说明重复数字在mid+1~len-1里
    if(sum <= mid){
      left = mid + 1;
    }
    //在1~mid之间的数组多于mid，说明重复数字在1~mid里
    else{
      right = mid;
    }
  }
  return left;
}
```

Find K Closest Elements

> Given a sorted array, two integers `k` and `x`, find the `k` closest elements to `x` in the array. The result should also be sorted in ascending order. If there is a tie, the smaller elements are always preferred.
>
> **Example 1:**
>
> ```
> > Input: [1,2,3,4,5], k=4, x=3
> > Output: [1,2,3,4]
> >
> >
> ```
>
> >
>
> **Example 2:**
>
> ```
> > Input: [1,2,3,4,5], k=4, x=-1
> > Output: [1,2,3,4]
> >
> ```

给定一个递增数组，两个整数k和x，返回数组中里最接近x的k个数

思路：

先用二分法找到数组中>=x的第一个数字，然后向左向右寻找，两个指针向左右移动，直至k次结束，此时将两个指针中间的元素加入结果集即是最终答案。

代码：

```
class Solution {
    public List<Integer> findClosestElements(int[] arr, int k, int x) {
        List<Integer> res = new ArrayList<>();
        int start = 0;
        int end = arr.length-1;
        while(start < end){
            int mid = (start+end)/2;
            if(arr[mid] >= x){
                end = mid;
            }
            else{
                start = mid+1;
            }
```

```
        }
        //end是第一个>=x的数
        int left = end-1;
        int right = end;
        int num = 0;
        while(num < k){
            int l_delta = Integer.MAX_VALUE;
            int r_delta = Integer.MAX_VALUE;
            if(left >= 0){
                l_delta = Math.abs(x-arr[left]);
            }
            if(right < arr.length){
                r_delta = Math.abs(x-arr[right]);
            }
            if(l_delta <= r_delta){
                left--;
            }
            else{
                right++;
            }
            num++;
        }
        for (int i = left+1;i < right;i++){
            res.add(arr[i]);
        }
        return res;
    }
}
```

Binary Search

WAND算法 DeepFM论文笔记