# 基本理论

双指针可以分为2类，一个“快慢指针”，一个是“左右指针”。前者主要解决链表中的问题，后者主要解决字符串和数组的问题。

# 【快慢指针】

快慢指针，通常初始化时都指向链表头结点head,然后fast在前（fast->next->next）,慢指针slow->next,巧妙解决链表中的问题。

## 1. [141. 环形链表](https://leetcode-cn.com/problems/linked-list-cycle/)【fast = fast->next->next】



思路：慢指针每次走一步，快指针每次走2步，必定会相遇，就如赛跑。

bool hasCycle(struct ListNode \*head)

{

    struct ListNode \*slow, \*fast;

    if (head == NULL) {

        return false;

    }

    slow = head;

    fast = head;

    while (fast != NULL && fast->next != NULL) {

        fast = fast->next->next;

        slow = slow->next;

        if (fast == slow) {

            return true;

        }

    }

    return false;

}

## 2. [面试题22. 链表中倒数第k个节点](https://leetcode-cn.com/problems/lian-biao-zhong-dao-shu-di-kge-jie-dian-lcof/)

让快指针先行K步，然后慢指针开始从头开始，当快指针走到结尾的时候，此时慢指针恰好在N-K个即倒数第K个。

如果是求中点：其实就是同时走，快指针到达结尾的时候就可以了返回slow指针了。

struct ListNode\* getKthFromEnd(struct ListNode\* head, int k)

{

    struct ListNode \*slow, \*fast;

    if (head == NULL) {

        return NULL;

    }

    slow = head;

    fast = head;

    while(k >= 1 && fast != NULL) {

        fast = fast->next;

        k--;

    }

    while(fast != NULL) {

        slow = slow->next;

        fast = fast->next;

    }

    return slow;

}

## 3.判定有环并返回环的起点【环形链表的环起点】

思想：fast与slow都从head开始，fast以2步的速度前进，slow以1步的速度前进。当相遇的时候假设slow走了K步，那么fast必然是2K步，

可知环的长度其实是2K-K = K。

设相遇到的点距离环的起点有m步，，则相遇点沿着环方向再次到环的起点的距离为k-m。

那么此时slow要是从起点开始走k – m步必然可以到达环的起点。

如果第一次相遇后，fast慢下来那么就是相遇。

struct ListNode \*detectCycle(struct ListNode \*head)

{

    struct ListNode \*slow, \*fast;

    if (head == NULL) {

        return NULL;

    }

    fast = head;

    slow = head;

    while(fast != NULL && fast->next != NULL) {

        fast = fast->next->next;

        slow = slow->next;

        if (fast == slow) {

            break;

        }

    }

    // 无环

    if (fast == NULL || fast->next == NULL) {

        return NULL;

    }

    slow = head;

    while (fast != slow) {

        slow = slow->next;

        fast = fast->next;

    }

    return slow;

}

## 4.链表的中点

参考2

# 【左右指针】

一般就是指向2个索引。

# 【左右指针之二分查找】

二分查可以在有序的数组中搜索给定的目标值的索引。

但是如果有重复值的话怎么才能找到边界呢？除了上述应用外还有没有别的用处。

## 【二分查找编程的几个关键点】

1. 当需要使用nums[right]的时候，需要则将right初始化为numsSize – 1，即闭区间查找，但也可能会用到mid+1,因此即使闭区间找也可能使用“<”号，如下方的题3【寻找峰值】。

优选这种方式，难一点额都会用到num[right].

2.while的条件是left <= right还是left < right根据初始化的方式定，numsSize则对应left < right，当然闭区间找有时也会用<.

3.当缩小区间的时候，left = mid + 1 或者right = mid - 1要根据mid是否可以被排除及搜索区间的闭合性来决定。

如left <= right，即闭区间.

mid可以被排除，则缩小时，需要left + 1,或right - 1;

向左缩小时，设mid不能被排除，则right = mid可以被排除，则缩小时，需要left

如left < right，即左闭右开

mid可以被排除，向右缩小时，需要left = left + 1,因为左闭，如果不+1，被重复了。

向左缩小时，需要right = mid,因为右开的，直接写mid，mid也不会被包含

4. 二分法避免溢出 mid = left + (right - left) / 2

5.退出循环。如果用left < right 那么[left,left]其实是没有查的，或[right,right]其实也没做检查

## 【二分查找的基本应用】

1.有序数组搜索固定值

2.有序数组（含重复元素）寻找左/右边界 - leetcode-34

3.旋转数组（不含重复值搜索固定值） - leetcode-33

4.旋转数组（含重复值搜索固定值） - leetcode-81

5.旋转数组（不含重复值搜素最小值） - leetcode-153

6.旋转数组（含重复值搜索最小值） - leetcode-154 困难

7.优化算法，比如搜索线性空间，如可可吃香蕉- leetcode-875

## 【基本思想与难点】

要求数组有序，其实也就是单调性或者是不是可以确定排除另一边的元素。

用左元素还是右元素比较是非常关键的。

比如旋转数组中如果是寻找目标值，则寻找递增区间是关键；

比如旋转数组中如果是寻找最小值，则和右元素比较是很关键的。

## 【二分查找之基本算法-leetcode-704】

写法（1）

int MybSearch(int\* nums, int size, int target)

{

    int left = 0;

    int right = size;

    int mid;

    while (left < right) {

        // 避免溢出

        mid = left + (right - left) / 2;

        if (target == nums[mid]) {

            return mid;

        }

        if (target > nums[mid]) {

            left = mid + 1;

        }

        if (target < nums[mid]) {

            right = mid;

        }

    }

    return -1;

}

写法2：

int search(int\* nums, int numsSize, int target){

    int left = 0;

    int right = numsSize - 1;

    int mid;

    while (left <= right) {

        // 避免溢出

        mid = left + (right - left) / 2;

        if (target == nums[mid]) {

            return mid;

        }

        if (target > nums[mid]) {

            left = mid + 1;

        }

        if (target < nums[mid]) {

            right = mid - 1; // 因为mid可以被排除

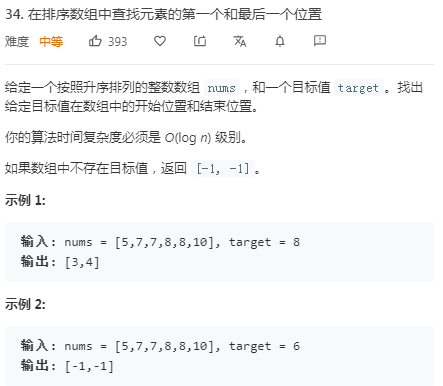
        }

    }

    return -1;

}

## 【二分查找之寻找左右边界-leetcode-34】



主要看写法2；

写法一：

int MybSearchLeftBound(int\* nums, int size, int target)

{

    int left = 0;

int right = size;

// 搜索区间是[left, fight),因此while循环的条件是left < right

    int mid;

    while (left < right) {

        mid = left + (right - left) / 2;

        // 相等的时候，缩小搜索区间

        if (target == nums[mid]) {

            right = mid;

        }

        if (target > nums[mid]) {

            left = mid + 1;

        }

        if (target < nums[mid]) {

            right = mid;

        }

    }

    return nums[left] == target ? left : -1;

}

写法二：

int FindBoudLeft(int\* nums, int numsSize, int target)

 {

     int left = 0;

     int right = numsSize - 1;

     int mid;

     if (numsSize == 0) {

        return -1;

     }

     while (left <= right) {

         mid = left + (right - left) / 2;

         if (nums[mid] == target) {

             right = mid - 1;

             continue;

         }

         if (nums[mid] > target) {

             right = mid - 1;

             continue;

         }

         if (nums[mid] < target) {

             left = mid + 1;

             continue;

         }

     }

// 因为是找左边界，相等的时候都是让right = mid -1。因此循环结束时其实主要目// 地是判定nums[left]是不是相等即可。特殊情况是找不到target，找不到的情况就是nums[left] ！= target 或者left > numsSize

     if (left >= numsSize) {

        return -1;

     }

// 一般情况因为是缩小

     return nums[left] == target ? left : -1;

 }

## 【二分查找之寻找右边界-leetcode-34】

写法一：

int MybSearchRightBound(int\* nums, int size, int target)

{

int left = 0;

int right = size;

int mid ;

while (left < right) {

mid = left + (right - left) / 2;

if (target == nums[mid]) {

left = left + 1;

}

if (target > nums[mid]) {

left = left + 1;

}

if (target < nums[mid]) {

right = mid;

}

}

return nums[left - 1] == target ? left - 1 : -1;

}

写法二：

 int FindBoudRight(int\* nums, int numsSize, int target)

 {

     int left = 0;

     int right = numsSize - 1;

     int mid;

     if (numsSize == 0) {

        return -1;

     }

     while (left <= right) {

        mid = left + (right - left) / 2;

        if (nums[mid] == target) {

            left = left + 1;

            continue;

        }

        if (nums[mid] > target) {

            right = mid - 1;

            continue;

        }

        if (nums[mid] < target) {

            left = mid + 1;

            continue;

        }

     }

// 找右边界，都是让left+1,因此正常下，只需要判断nums[left - 1]和target是否相等// 即可。特殊情况是找不到，即nums[left - 1] ！= target 或者right < 0

     if (right < 0) {

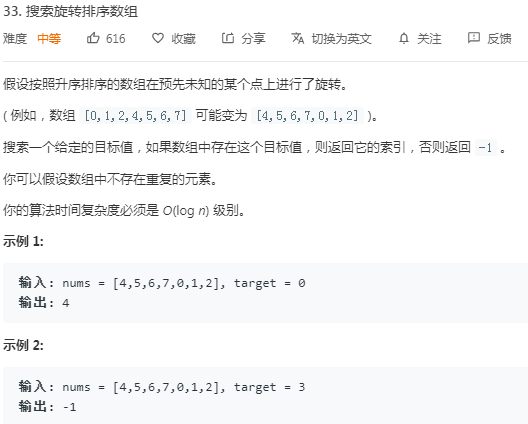
        return -1;

     }

     return nums[left - 1] == target ? left - 1 : -1;

 }

## 【寻找旋转数组固定值-无重复元素】【寻找递增区间+排除区间】



思路：

旋转数组中使用二分查找，无论是搜索特定值还是搜索最小值，都需要在左右2个区间中，找到连续递增的区间。

1. 判断区间递增只需要判断边界值。nums[left] <= nums[mid],则left-mid递增，反之mid-right递增。注意如果是有重复数值的时候，不适用，如【1,1,1,1,1,3,1】

**（2）**当找到递增区间后:

//对于左区间递增，如果nums[left] <= target && target <= nums[mid] 在左区间找：right = mid - 1;否则在右区间找:left = mid + 1;

//对于右区间递增，如果nums[mid] <= target && target <= nums[right],在右区间找：left = mid + 1;否则在左区间找：right = mid - 1;

int search(int\* nums, int numsSize, int target)

{

    int left = 0;

    int right = numsSize - 1;

    int mid;

    while (left <= right) {

        mid = left + (right - left) / 2;

        if (nums[mid] == target) {

            return mid;

        }

        // 左区间递增，则在左区间二分查找。

        if (nums[mid] >= nums[left]) {

// 如果符合，则肯定在这个范围

            if (nums[left] <= target && target <= nums[mid]) {

                right = mid;// 不能排除，而且是闭区间

            } else {

                left = mid + 1;// 因为可以排除

            }

        } else {

            if (nums[mid] <= target && target <= nums[right]) {

                left = mid; // 不能排除，而且是闭区间

            } else {

                right = mid - 1; // 因为可以排除

            }

        }

    }

    return -1;

}

## 【查找寻找旋转数组固定值-重复元素】【去重+寻找递增区间+排除区间】



思路：同81题，但是由于存在相同元素，以nums[mid] >= nums[left]就判定左半边是递增区间是有问题的。比如1，2，1，1，1.因此可以去重。其他基本类似

bool search(int\* nums, int numsSize, int target)

{

    int left = 0;

    int right = numsSize - 1;

    int mid;

    while (left <= right) {

        mid = left + (right - left) / 2;

        if (nums[mid] == target) {

            return true;

        }

// 去重，最坏情况退化成O(n)

        if (nums[mid] == nums[left]) {

            left++;

            continue;

        }

        if (nums[mid] >= nums[left]) {

            if (nums[left] <= target && target <= nums[mid]) {

                right = mid;

            } else {

                left = left + 1;

            }

        } else {

            if (nums[mid] <= target && target <= nums[right]) {

                left = mid;

            } else {

                right = mid - 1;

            }

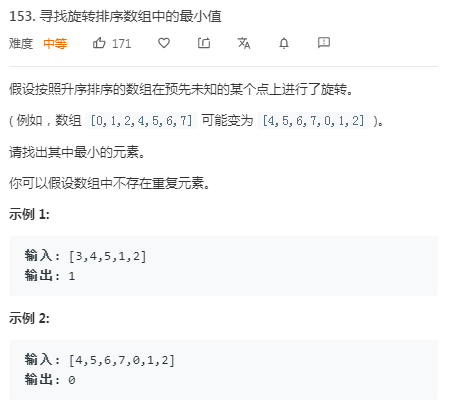
        }

    }

    return false;

}

## 【查找旋转数组最小值-无重复元素】【和尾部比排除区间】



思路1：

同样的寻找递增区间，但是如果有重复则不好做

int findMin(int\* nums, int numsSize)

{

    int left = 0;

    int right = numsSize - 1;

    int mid;

    while (left <= right) {

        mid = (right - left) / 2 + left;

        // 一旦变成nums[left] <= nums[right],则左侧肯定最小

        if (nums[left] <= nums[right]) {

            return nums[left];

        }

        // 递增区间，最小值肯定不在这,在右边,向右压缩

        if (nums[mid] >= nums[left]) {

            left = mid + 1;

        } else {

            right = mid;// 否则最小值肯定在左边

        }

    }

    return 0;

}

思路2：

和右边元素比较，如果nums[mid] > nums[right]，则最小值肯定在这。

代码：

int findMin(int\* nums, int numsSize)

{

    int left = 0;

    int right = numsSize - 1;

    int mid;

    while (left <= right) {

        mid = (right - left) / 2 + left;

        if (left == right) {

            return nums[left];

        }

        if (nums[mid] > nums[right]) {

            left = mid + 1;

        } else {

            right = mid;

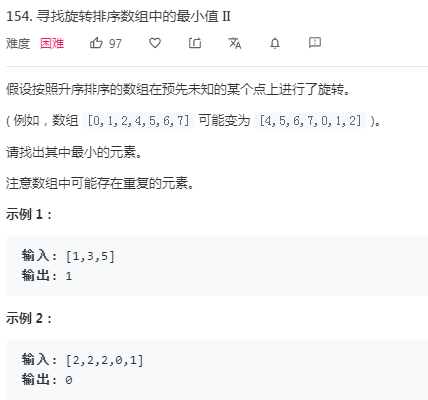
        }

    }

    return 0;

}

## 【查找旋转数组最小值-重复元素】【去重+和尾部比排除区间】



思路：

还是在无序的里面找，但是要注意如果因为有重复元素，因此nums[mid] = nums[right]，要对后面去重。

int findMin(int\* nums, int numsSize)

{

    int left = 0;

    int right = numsSize - 1;

    int mid;

    while (left <= right) {

        mid = (right - left) / 2 + left;

        if (left == right) {

            return nums[left];

        }

// 去重如3，3，1，3.

        if (nums[mid] == nums[right]) {

            right--;

            continue;

        }

        if (nums[mid] > nums[right]) {

            left = mid + 1;

        } else {

            right = mid;

        }

    }

    return 0;

}

## 1．875. 爱吃香蕉的珂珂【线性搜索空间】



思路：暴力解决的话其实也是从1到最大速度去搜索。这样的话其实可以用二分搜索中的边界搜索。

int IsOk(int\* piles, int pilesSize, int speed, int h)

{

    int i;

    int t = 0;

    int tmp = 0;

    for (i = 0; i < pilesSize; i++) {

        tmp = piles[i] / speed;

        t += (piles[i] % speed == 0) ? tmp : tmp + 1;

        if (t > h) {

            return 0;

        }

    }

    return 1;

}

int minEatingSpeed(int\* piles, int pilesSize, int H)

{

    int i = 0;

    int minSpeed = -1;

    int maxSpeed = -1;

    int speed, midSpeed;

    minSpeed = 1;

    maxSpeed = piles[0];

    for (i = 1; i < pilesSize; i++) {

        //minSpeed = (minSpeed < piles[i]) ? minSpeed : piles[i];

        maxSpeed = (maxSpeed < piles[i]) ? piles[i] : maxSpeed;

    }

    maxSpeed += 1;

    while(minSpeed < maxSpeed) {

        midSpeed = minSpeed + (maxSpeed - minSpeed) / 2;

        // 能吃完的话，继续缩小范围

        if (IsOk(piles, pilesSize, midSpeed, H) == 1) {

            maxSpeed = midSpeed;

        } else {    // 不能吃完

            minSpeed = midSpeed + 1;

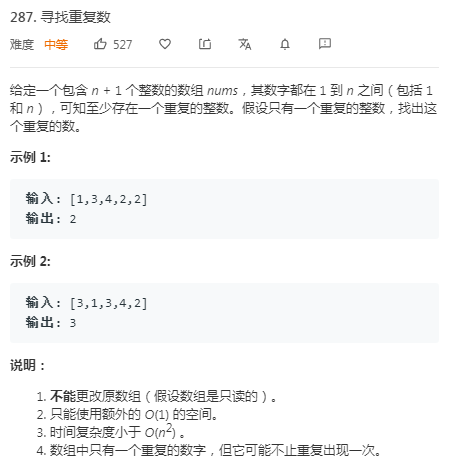
        }

    }

    return (IsOk(piles, pilesSize, minSpeed, H) == 1) ? minSpeed : -1;

}

## 2．287. 寻找重复数 【线性搜索空间】



思路：暴力是可以解答，但是复杂度会超。

其实题目中也就是求[1,n]中的1个数字，那么可以就这一段线性空间利用二分法倒是很像第875题。left =1,right = numsSize – 1;

由于是1+n个元素，值又是1-n,因此对于每一个mid，只需要对原数组进行计数对于<mid就对count累计，如果重复的值出现在左边，必然导致count>=mid；因此，对于count>=mid，对答案的结果向左缩小区间。

另外：由于元素组成的特殊性，必然导致会成环，可以利用链表的成环找起点的思路去做。

int findDuplicate(int\* nums, int numsSize)

{

    int left = 1;

    int right = numsSize - 1;

    int mid;

    int i;

    int count = 0;

    while (left <= right) {

        mid  = left + (right - left) / 2;

        count = 0;

        for (i = 0; i < numsSize; i++) {

            if (nums[i] < mid) {

                count++;

            }

        }

        if (count >= mid) {

            right = mid - 1;

        } else {

            left = mid + 1;

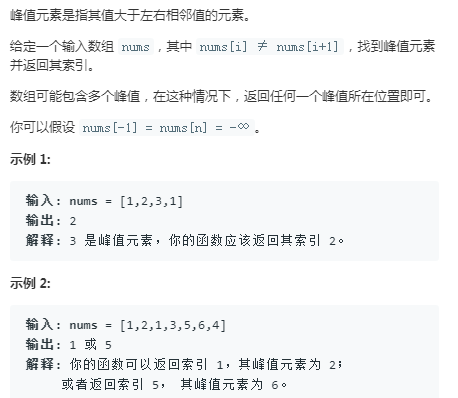
        }

    }

    return left - 1; //使用mid是不行的，使用right也可以

}

## 3. [162. 寻找峰值](https://leetcode-cn.com/problems/find-peak-element/)【排除区间】



思路：

由于两边是负无穷，因此可以利用如下特点。

假设num[mid] > nums[mid + 1]那么左边肯定存在封顶，反之右边肯定存在封顶。

代码：

int findPeakElement(int\* nums, int numsSize)

{

int left = 0;

    int right = numsSize - 1;

    int mid;

// 由于right是size – 1,本是闭区间查的，但是由于有mid+1,因此这里用小于号，// 边界不查.因此退出循环实际上

    while (left < right) {

        mid = left + (right - left) / 2;

        //printf("%d %d %d\n",left, right, mid);

        if (nums[mid] > nums[mid + 1]) {

            right = mid;

        } else if (nums[mid] < nums[mid + 1]) {

            left = mid + 1;

        }

    }

    return left;

}

## 4. [240. 搜索二维矩阵 II](https://leetcode-cn.com/problems/search-a-2d-matrix-ii/)【库函数或者自己实现-当成二叉树】



int MyCmp(const void\* a, const void\* b) {

    int\* pa = (int\*)a;

    int \*pb = (int\*)b;

    return \*pa - \*pb;

}

// @lc code=start

bool searchMatrix(int\*\* matrix, int matrixRowSize, int matrixColSize, int target)

{

    int i;

    int\* ret = NULL;

    for (i = 0; i < matrixRowSize; i++) {

        ret = (int\*)bsearch((int\*)&target, (int\*)matrix[i], matrixColSize, sizeof(int), MyCmp);

        if (ret != NULL) {

            return true;

        }

    }

    return false;

}

/\*方法二：【也适用[**74. 搜索二维矩阵**](https://leetcode-cn.com/problems/search-a-2d-matrix/)】

选取起点右上角或左下角进行遍历。

当matrix[row][col] > target则向左走，换列；

当matrix[row][col] < target则向下走，换行；

也可以选择左下，为什么不选择左上和右下呢，因为这2个分别为最大最小值，当和target比较时路径不唯一既可以还行也可以换列

\*/

bool searchMatrix(int\*\* matrix, int matrixRowSize, int matrixColSize, int target)

{

int row = 0;

int col = matrixColSize - 1;

while (row < matrixRowSize && col >= 0) {

if (matrix[row][col] == target) {

return true;

}

if (matrix[row][col] > target) {

col--;

continue;

}

if (matrix[row][col] < target) {

row++;

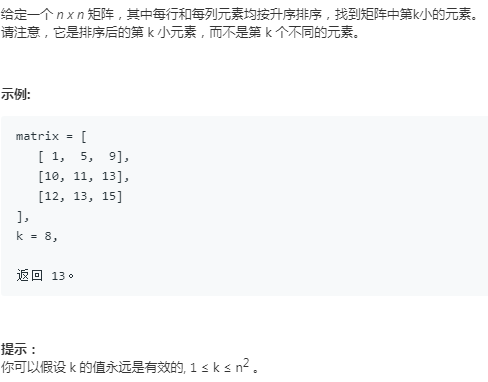
}

}

return false;

}

## 5. [378. 有序矩阵中第K小的元素](https://leetcode-cn.com/problems/kth-smallest-element-in-a-sorted-matrix/)【可二分，可一般方法】



这里主要为了练习二分，因此用二分，可以更一般，就是开一个辅助数组，拷贝，再用库函数排序，返回k-1.

当然也可以二分，在matrix[0][0] 到matrix[row -1][col - 1]的区间线性搜索，对每一个mid,遍历二维数组，对<=mid的进行计数，如果conut >= k 肯定在左区间，否则在右区间。类似第2题的重复数。

int Count(int\*\* matrix, int matrixSize, int colSize, int value)

{

    int row = 0;

    int col ;

    int count = 0;

    int j;

    for (row = 0; row < matrixSize; row++) {

        if (matrix[row][colSize - 1] <= value) {

            count += colSize;

            continue;

        }

        for (col = 0; col < colSize; col++) {

            if (matrix[row][col] <= value) {

                continue;

            }

            break;

        }

        count += col;

    }

    return count;

}

int kthSmallest(int\*\* matrix, int matrixSize, int\* matrixColSize, int k)

{

    int left = 0;

    int right = 0;

    int mid;

    int count;

    int colSize = matrixColSize[0];

    left = matrix[0][0];

    right = matrix[matrixSize - 1][matrixColSize[0] - 1];

    while (left <= right) {

        mid = left + (right - left) / 2;

        if (left ==right && left == mid) {

            return left;

        }

        count = Count(matrix, matrixSize, matrixColSize[0], mid);

        if (count >= k) {

            right = mid;

        } else {

            left = mid + 1;

        }

    }

    return left;

}

一般方法:

int MyCmp(const void \*a, const void\* b)

{

int \*pa = (int\*)a;

int \*pb = (int\*)b;

return \*pa - \*pb;

}

int kthSmallest(int\*\* matrix, int matrixSize, int\* matrixColSize, int k)

{

int i, j;

int ret;

int \*temp = (int\*)calloc(matrixColSize[0] \* matrixSize, sizeof(int));

for (i = 0; i < matrixSize; i++) {

memcpy(&temp[j] + i \* matrixColSize[0], matrix[i], matrixColSize[0] \* sizeof(int));

}

qsort(temp, matrixColSize[0] \* matrixSize, sizeof(int), MyCmp);

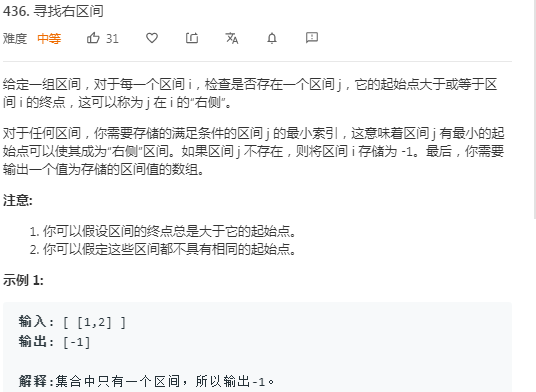
ret = temp[k -1];

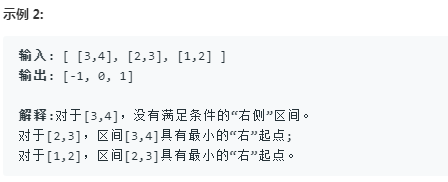
free(temp);

return ret;

}

## 6. [436. 寻找右区间](https://leetcode-cn.com/problems/find-right-interval/)【结构体捆绑排序+二分寻找边界】





题意：对于[1,2],如果有[3,4] [5,6],需要返回区间[3,4]所在的索引。

思路一：排序+二分

1.按区间的第一个元素进行排序，然后遍历每个区间的元素1，对每个区间的元素1 x，查找第一个>=x 的区间，再映射到其ndex

2.难点如何在排序之后能捆绑区间在原集合中的序号，这里用集合体捆绑

typedef struct {

    int nums[2];

    int index;

}ELEMENT\_T;

int MyCmp(const void\* a, const void\* b)

{

    ELEMENT\_T\* pa = (ELEMENT\_T\*)a;

    ELEMENT\_T\* pb = (ELEMENT\_T\*)b;

    return pa->nums[0] - pb->nums[0];

}

// 寻找第一个>=target的元素，其实是寻找左边界的模板套用

int SearchFirstEle(ELEMENT\_T\* table, int size, int target)

{

    int left = 0;

    int right = size - 1;

    int mid;

    while (left <= right) {

        mid = left + (right - left) / 2;

        if (table[mid].nums[0] >= target) {

            right = mid - 1;

        } else {

            left = mid + 1;

        }

    }

    if (left >= size) {

        return -1;

    }

    return table[left].nums[0] >= target ? table[left].index : -1;

}

int\* findRightInterval(int\*\* intervals, int intervalsSize, int\* intervalsColSize, int\* returnSize)

{

    int i;

    int temp;

    ELEMENT\_T\* table = (ELEMENT\_T\*)calloc(intervalsSize, sizeof(ELEMENT\_T));

    int\* ans = (int\*)calloc(intervalsSize, sizeof(int));

    // 将区间与在原集合中的序号捆绑

for (i = 0; i < intervalsSize; i++) {

        memcpy(table[i].nums, intervals[i], 2 \* sizeof(int));

        table[i].index = i;

    }

    qsort(table, intervalsSize, sizeof(ELEMENT\_T), MyCmp);

// 对每个结合，去排序数组中寻找第一个比他大或者相等的元素的序号

for (i= 0; i < intervalsSize; i++) {

        temp = SearchFirstEle(table, intervalsSize, table[i].nums[1]);

        ans[table[i].index] = temp;

    }

    free(table);

    table = NULL;

    \*returnSize = intervalsSize;

    return ans;

}

思路二：暴力求解。

对于区间集合的每一个区间intervals[i]，都遍历的去找最小右区间索引。一路上记住该最小值和索引，及时同步更新，但是最后会超时。

#define GETMIN(x, y)  ((x) >= (y)) ? y :x;

int\* findRightInterval(int\*\* intervals, int intervalsSize, int\* intervalsColSize, int\* returnSize)

{

    int \* ans = NULL;

    int i, j;

    int minStart = 0xfffffff;

    int index;

    ans = (int\*)calloc(intervalsSize, sizeof(int));

    for (i = 0; i < intervalsSize; i++) {

        index = -1;

        minStart = 0xfffffff;

        for (j = 0; j < intervalsSize; j++) {

            if (i == j) {

                continue;

            }

// 一旦发现比他大的，和历史值比较，看下是否更新index

            if (intervals[i][1] <= intervals[j][0]) {

                minStart = GETMIN(minStart, intervals[j][0]);

                index = (minStart == intervals[j][0]) ? j : index;

            }

        }

        ans[i] = index;

    }

    \*returnSize = intervalsSize;

    return ans;

}