# 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。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

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

## 【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】

就是如何更新左右指针，每次需要知道要排查那边的元素。

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;

}

## 【二分查找之寻找左边界-leetcode-34】

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;

while (left <= right) {

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

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

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

}

if (nums[mid] == target) {

right = mid; // right不能跳过如6，7，8 找7

continue;

}

if (nums[mid] > target) {

right = mid - 1;

continue;

}

if (nums[mid] < target) {

left = mid + 1;

continue;

}

}

return -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;

     while (left <= right) {

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

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

           // [1,4]找2

 if (mid == 0) {

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

            }

// [8,8,10]找8与[8,8]找8

            return nums[mid] == target ? mid : ((nums[mid - 1] == target) ? mid - 1 : -1);

        }

        if (nums[mid] ==  target) {

            left = left + 1;

            continue;

        }

        if (nums[mid] > target) {

            right = mid - 1;

            continue;

        }

        if (nums[mid] < target) {

            left = mid + 1;

            continue;

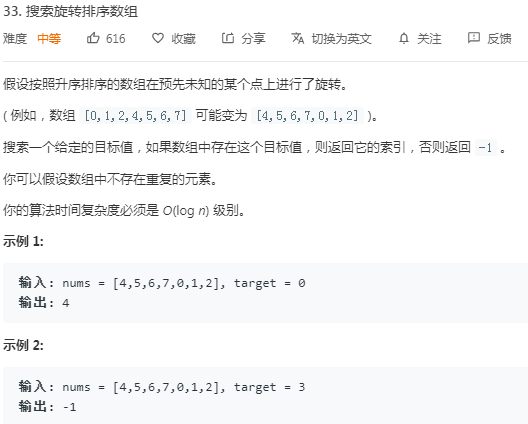
        }

     }

     return -1;

 }

## 【二分查找之寻找无重复元素旋转数组固定值-leetcode-33】



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;

}

## 【二分查找寻找有重复元素旋转数组固定值-leetcode-81】



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;

            }

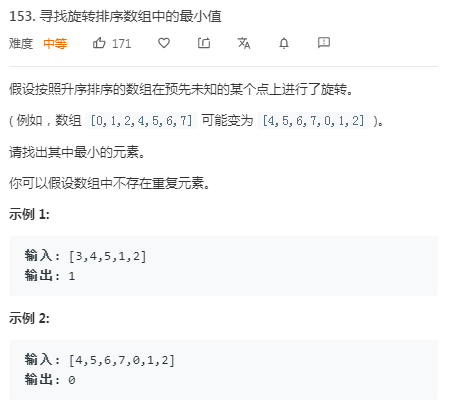
        }

    }

    return false;

}

## 【二分法之查找旋转数组无重复元素最小值-leetcode-153】



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;

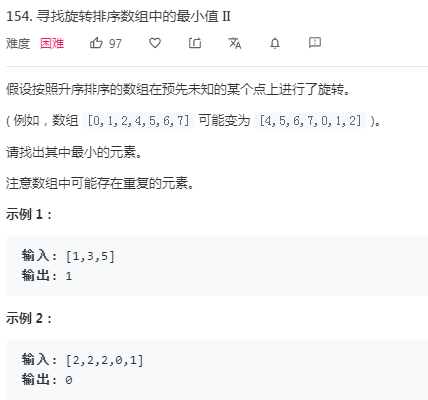
        }

    }

    return 0;

}

## 【二分法之查找旋转数组有重复元素最小值-leetcode-154】



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. [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】



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;

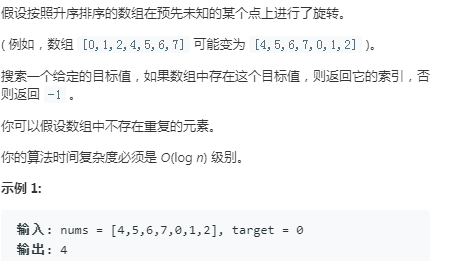
        }

    }

    return false;

}

## 3. [33. 搜索旋转排序数组](https://leetcode-cn.com/problems/search-in-rotated-sorted-array/)【先二分查找寻找下降的边界】



思路：找到下降的index，然后分2次二分法。

由于logn因此肯定寻找边界的下降的地方也需要用二分法。

修改一般的二分法即可。

判断条件改为：

当nums[mid] > nums[left] 向右压缩，left = mid

当nums[mid] < nums[right]，向左压缩，right = mid

当numss[mid] == nums[left],可以返回mid

int FindBond(int\* nums, int numsSize)

{

    int left = 0;

    int right = numsSize;

    int mid;

    while (left < right) {

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

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

            return mid;

        }

// 向右压缩

        if (nums[mid] > nums[left]) {

            left = mid;

            continue;

        }

        if (nums[mid] < nums[left]) {

            right = mid;

            continue;

        }

    }

    return -1;

}

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

{

    int \*pa = (int\*)a;

    int \*pb = (int\*)b;

    return \*pa - \*pb;

}

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

{

    int index;

    int\* ret = NULL;

    index = FindBond(nums, numsSize);

    if (index == -1) {

        return -1;

    }

    ret = bsearch(&target, (int\*)nums, index + 1, sizeof(int), MyCmp);

    if (ret != NULL) {

        return (ret - nums); // 利用指针相减来获取

    }

    ret = bsearch(&target, (int\*)(&nums[index + 1]), numsSize - index - 1, sizeof(int), MyCmp);

    if (ret != NULL) {

        return (ret - nums);

    }

    return -1;

}