# 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。

写在前头：

1.更新left和right的值时，关于mid是否加1减1都可以用区间来看。

如果right = sizeof(nums) / sizeof(nums[0]) 即[left，right)左闭右开，则更新right都不需要mid – 1

2.同理：while的终止条件也如此，左闭右开直接while(left < right)

3.如果需要搜索边界时，nums[mid] == target 不要立即返回。

左边界，则继续向左压缩空间，right = mid

右边界，则继续向右压缩空间，left = left + 1.

4,搜索边界，有可能搜索不到结果都需要判断

Return nums[left] == target ? left : -1;由于右边界left + 1,因此

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

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

## 【二分查找之基本算法】

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;

}

## 【二分查找之寻找左边界】

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;

}

## 【二分查找之寻找右边界】

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;

}

## 【二分查找之应用到算法优化的题目】

二分查找不止应用到对一个有序数组进行查找，有时很隐秘。

在连续的空间线性搜索，也是可以应用二分查找的重要标志。一般也会和边界搜索一起用。

## 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;

}