

线性穷举

双指针法

YouTube频道: [hwdong](https://www.youtube.com/hwdong)

博客: hwdong-net.github.io

双指针法

- 线性迭代有时可以采用“双指针”法避免不必要的穷举，提高算法的效率。
- 双指针法主要分为：
 - 左右指针（**夹逼指针/首尾指针、背向指针**）：相向而行或背向而行
 - **前后指针（快慢指针）**：同向而行。
 - **滑动窗口**：同向而行，但两指针之间距离动态伸缩

左右指针

夹逼指针/首尾指针

YouTube频道: [hwdong](https://www.youtube.com/channel/UCwv33333333333333333333)

博客: hwdong-net.github.io

查找4

1. 二分查找

1	3	4	7	8	10	13	16
---	---	---	---	---	----	----	----

1. 二分查找

查找4

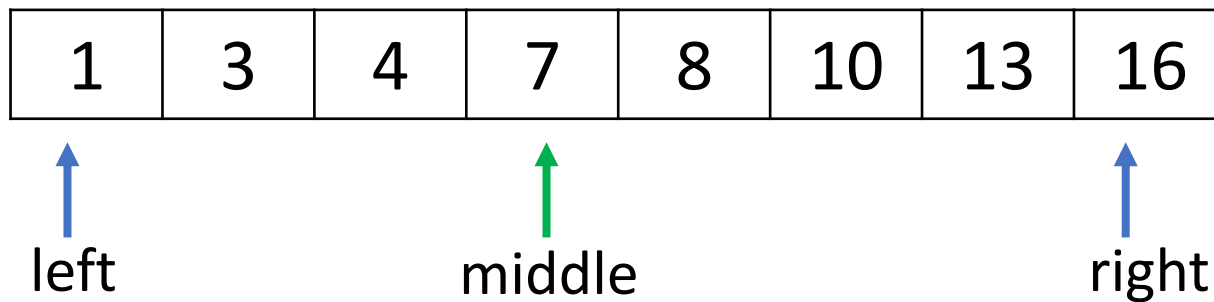
1	3	4	7	8	10	13	16
---	---	---	---	---	----	----	----

↑
left

↑
right

1. 二分查找

查找4



1. 二分查找

查找4

1	3	4	7	8	10	13	16
---	---	---	---	---	----	----	----

↑
left

↑ ↑
rightmiddle

查找4

1. 二分查找

1	3	4	7	8	10	13	16
---	---	---	---	---	----	----	----

↑ ↑ ↑
left middle right

查找4

1. 二分查找

1	3	4	7	8	10	13	16
---	---	---	---	---	----	----	----

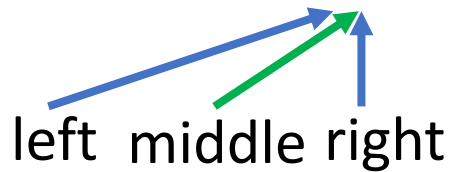
left middle right

查找4

1. 二分查找

1	3	4	7	8	10	13	16
---	---	---	---	---	----	----	----

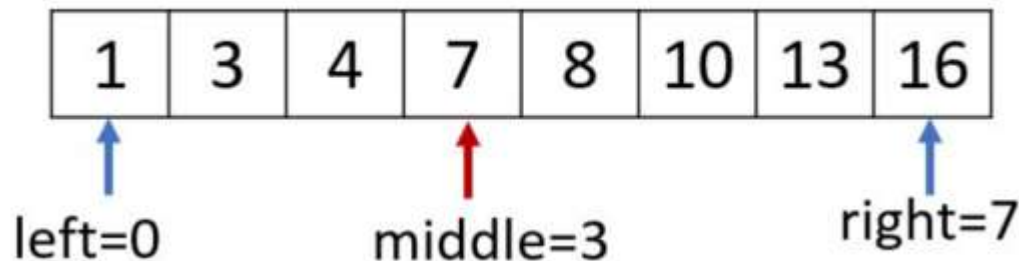
left middle right



1. 二分查找

```
int binary_search(a, left, right, x):  
    while left <= right:  
        mid = (left + right) / 2  
        if a[mid] < x:  
            left = mid + 1  
        elif a[mid] > x:  
            right = mid - 1  
        else:  
            return mid  
    return -1
```

查找4




2. 回文串的判别

- 回文串：正读和反读都一样的字符串。
- 如：aba、abcdcba都是回文串，而abc不是回文串。

夹逼指针判别回文串

a b c d c b a



↑ left ↑ right

夹逼指针判别回文串

a b c d c b a



left



right

夹逼指针判别回文串

a b c d c b a

↑ ↑
left right

夹逼指针判别回文串

a b c d c b a



用夹逼指针(首尾指针)指向字符串的开头和结尾，如果它们指向的字符相等，则继续相向各移一个位置，如果它们指向的字符不等，则不是回文串。当两个指针相遇时，则是回文串。


```
bool isPalindrome(string s) {  
    int left = 0, right = s.size() - 1;  
    while (left < right) {  
        if (s[left] != s[right]) {  
            return false;  
        }  
        left++;        right--;  
    }  
    return true;  
}
```

a b c d c b a
 ↑ ↑
 left right

3. 两数之和 II - 输入有序数组

- 返回有序递增的整数数组中和等于目标数的两个数的序号。
- 假设每个输入 只对应唯一的答案，而且你 不可以 重复使用相同的元素。

输入：numbers = [2,7,11,15], target = 9

输出：[1,2]

解释：2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2 。返回 [1, 2]

有序：左右指针（夹逼指针）法

[2 7 11 15]



left



right

target= 18

```
if(a[left]+a[right]> target)  
    right--;
```

有序：左右指针（夹逼指针）法

[2 7 11 15]



left



right

target= 18

```
if(a[left]+a[right]> target)
```

```
    right--;
```

```
else if(a[left]+a[right]< target)
```

```
    left++;
```

有序：左右指针（夹逼指针）法

[2 7 11 15]

↑
left

↑
right

target= 18

```
if(a[left]+a[right]> target)
    right--;
else if(a[left]+a[right]< target)
    left++;
```

有序：左右指针（夹逼指针）法

[2 7 11 15]

↑
left

↑
right

target= 18

```
if(a[left]+a[right]> target)
```

```
    right--;
```

```
else if(a[left]+a[right]< target)
```

```
    left++;
```

有序：左右指针（夹逼指针）法

[2 7 11 15]

↑ ↑
left right

target= 18

```
if(a[left]+a[right]> target)
    right--;
else if(a[left]+a[right]< target)
    left++;
```

有序：左右指针（夹逼指针）法

[2 7 11 15]

↑ ↑
left right

target= 18

```
if(a[left]+a[right]> target)
    right--;
else if(a[left]+a[right]< target)
    left++;
else if(a[left]+a[right]== target)
    return (left+1,right+1);
```



```
auto twoSum(vector<int> a, int target) {  
    int left = 0, right = a.size() - 1;  
    while (left < right) {  
        int sum = a[left] + a[right];  
        if (sum == target) {  
            return make_pair(left + 1, right + 1);  
        } else if (sum < target) {  
            最长回文子串  
            left++; // 让 sum 大一点  
        } else if (sum > target) {  
            right--; // 让 sum 小一点  
        }  
    }  
    return make_pair(-1, -1);  
}
```

左右指针

背向指针

YouTube频道: [hwdong](https://www.youtube.com/channel/UCwDong)

博客: hwdong-net.github.io

4. 最长回文子串

- 给你一个字符串 s ，找到 s 中最长的回文子串。(如果字符串的反序与原始字符串相同，则该字符串称为回文字符串。)
- 示例：
输入： $s = \text{"babad"}$
输出： "bab"
解释： "aba" 同样是符合题意的答案

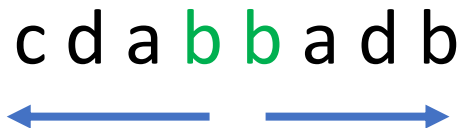
回文串的中心点

- 回文串的长度是奇数，则有一个中心点，否则有2个中心点。

c d a **b** a d b



c d a **b** **b** a d b



回文串的中心点

```
// 在 s 中寻找以 s[l] 和 s[r] 为中心的最长回文串
string palindrome(string s, int l, int r) {
    while l >= 0 && r < s.size()
        && s[l] == s[r]:
        l--; r++; // 双指针, 向两边展开
    }
    return s.substr(l + 1, r);
}
```

```
string longestPalindrome(string s) {  
    string res = "";  
    for (int i = 0; i < s.size(); i++) {  
        // 以 s[i] 为中心的最长回文子串  
        string s1 = palindrome(s, i, i);  
        // 以 s[i] 和 s[i+1] 为中心的最长回文子串  
        string s2 = palindrome(s, i, i + 1);  
        if s1.size() > s2.size():  
            if s1.size() > res.size(): res = s1 ;  
        else:  
            if s2.size() > res.size(): res = s2 ;  
    }  
    return res;  
}
```

前后指针

快慢指针

YouTube频道: [hwdong](https://www.youtube.com/hwdong)

博客: hwdong-net.github.io

5. 奇偶调序

- 重排数组元素使得所有的奇数位于所有偶数之前。

- 示例：

输入：nums = [1,2,3,4]

输出：[1,3,2,4]

注：[3,1,2,4] 也是正确的答案之一

奇偶调序-思路1：辅助数组

- 定义一个等长的数组，遍历两次数组，第一次存奇数，第二次存偶数，最后把临时数组的内存拷贝到原数组中。

[3 2 6 1 5 4]

[]

[3]

[3 1]

[3 1 5]

第1次迭代

奇偶调序-思路1：辅助数组

- 定义一个等长的数组，遍历两次数组，第一次存奇数，第二次存偶数，最后把临时数组的内存拷贝到原数组中。

[3 2 6 1 5 4]

[] [3 1 5 2]

[3] [3 1 5 2 6]

[3 1] [3 1 5 2 6 4]

[3 1 5]

第1次迭代

第2次迭代

```
void sortOddEven(int a[],int n) {  
    int *temp = new int[n];  
    int index = 0;    // 临时数组的当前下标  
    //第一遍迭代, 将原数组奇数放入临时数组  
    for(int i = 0; i < n; i++)  
        if((a[i] & 1) != 0)  
            temp[index++] = a[i];  
  
    //第一遍迭代, 将原数组奇数放入临时数组  
    for(int i = 0; i < n; i++)  
        if((a[i] & 1) == 0)  
            temp[index++] = a[i];  
  
    // 把临时数组拷贝会原来的数组  
    for(int i = 0; i < n; i++)  
        a[i] = temp[i];  
}
```

$$F(n) = 3n$$

空间复杂度为 $O(n)$

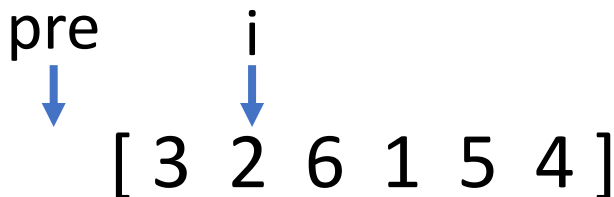
奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数

pre i
↓ ↓
[3 2 6 1 5 4]

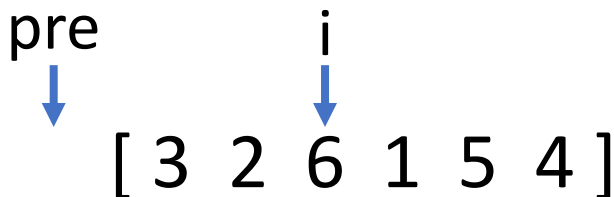
奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



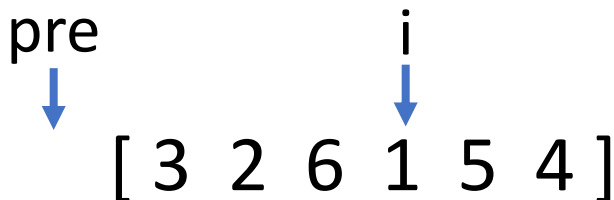
奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



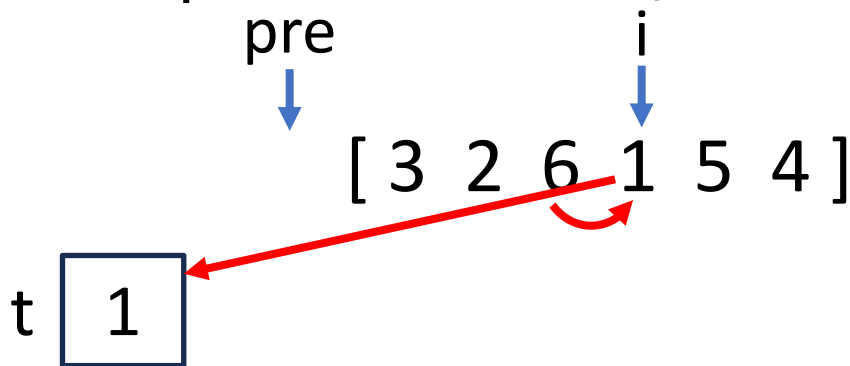
奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



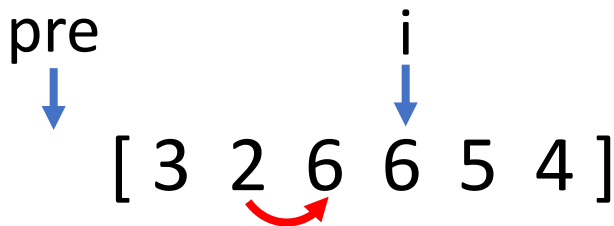
奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



t 1

奇偶调序-思路2：前后指针

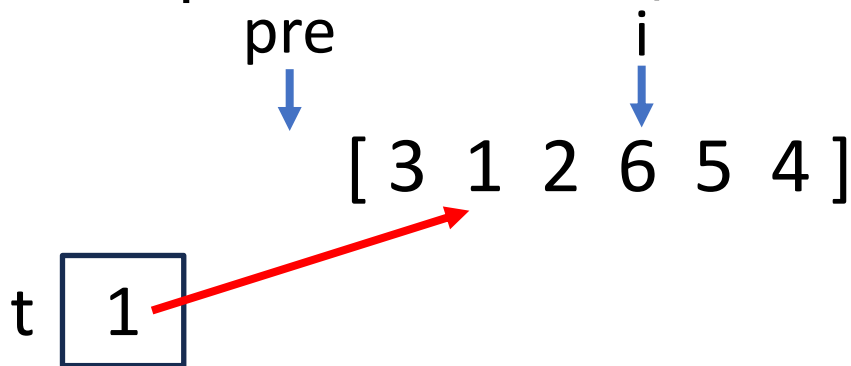
- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数

pre i
↓ ↓
[3 2 2 6 5 4]

t 1

奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



奇偶调序-思路2：前后指针

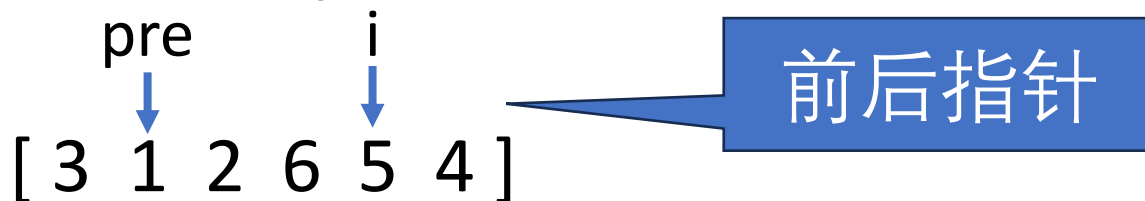
- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数

pre i
↓ ↓
[3 1 2 6 5 4]

t 1

i奇偶调序-思路2：前后指针

- 遍历数组，将当前奇数之前的偶数后移一位，把当前奇数移到这些偶数前。
- 用pre和i指向前一个和当前的奇数



t 1

```

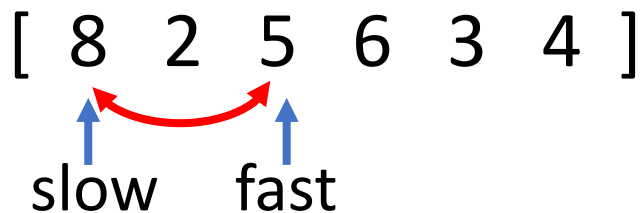
void sortOddEven(int a[],int n) {
    int pre = -1;           // 前一个奇数的位置
    for(int i = 0; i < n; i++){
        if((a[i] & 1) != 0){ // 当前整数是奇数
            int t = a[i];    //保存当前奇数到临时存储
            int j = i-1;
            //将 (i - 1) 到 pre+1 的偶数都后移一位
            for(; j>pre ;j--)
                a[j+1] = a[j];
            a[j+1] = t;       //当前奇数放在这些偶数前面
            pre = j+1;        // 更新前一个奇数的位置
        }
    }
}

```

$F(n) = ?$

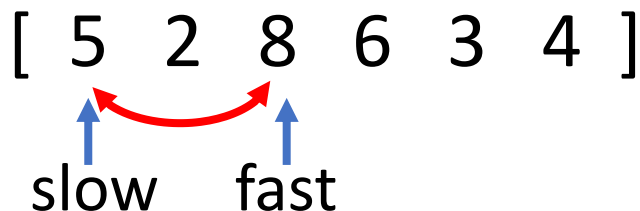
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。**



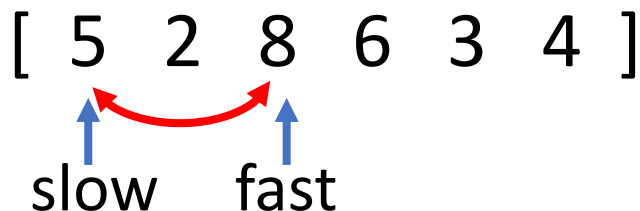
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。



奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后**slow++**，表示下个奇数的位置应该是下个位置。



奇偶调序-思路3：前后指针（快慢指针）

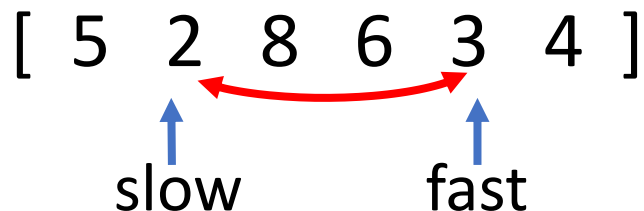
- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后**slow++**，表示下个奇数的位置应该是下个位置。

[5 2 8 6 3 4]


slowfast

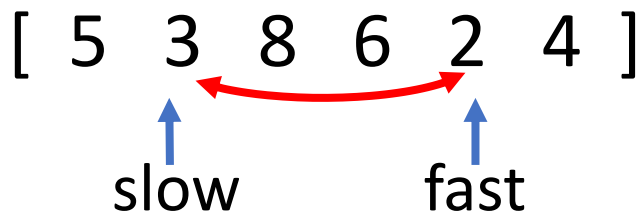
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。



奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。



奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。

[5 3 8 6 2 4]

↑ ↑
slow fast

奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。

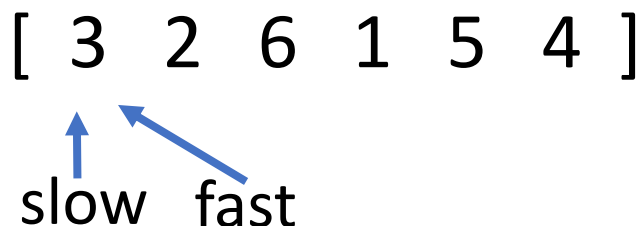
[5 3 8 6 2 4]

 ↑ ↑

 slow fast

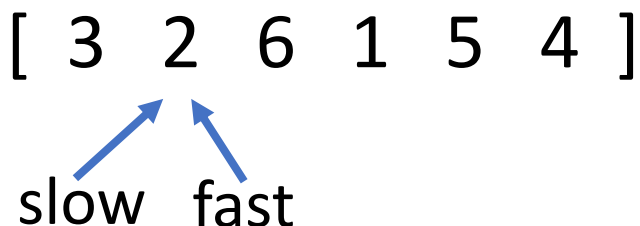
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



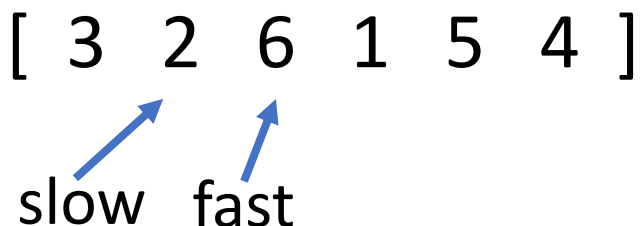
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



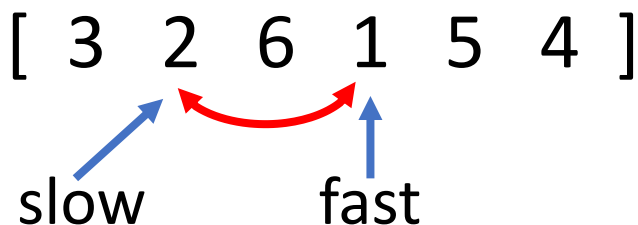
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



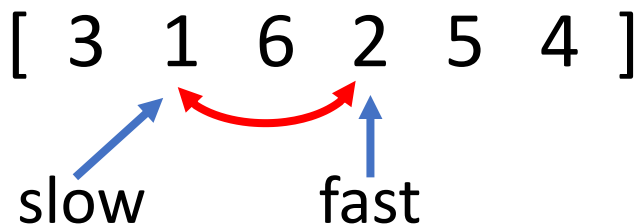
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



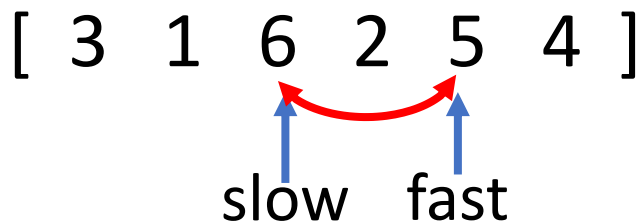
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



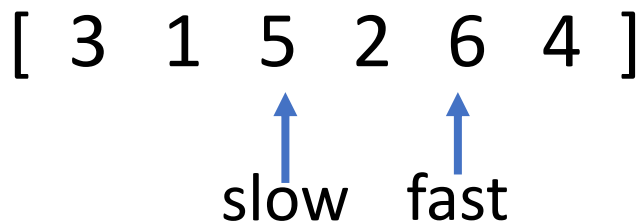
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



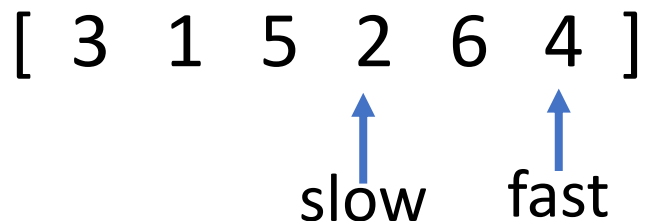
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



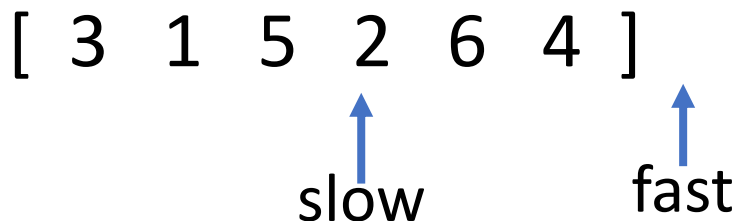
奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



奇偶调序-思路3：前后指针（快慢指针）

- 慢指针slow指向下个奇数应该存放的位置，快指针fast寻找下个奇数。
- fast找到奇数后，与slow对应的元素进行交换，随后slow++，表示下个奇数的位置应该是下个位置。
- 如果slow指向的位置刚好是奇数则自己和自己交换



```
void sortOddEven(int a[],int n) {  
    //fast指针一路前进  
    for(int low = 0, fast = 0, t = 0; fast < n; fast++)  
        if((a[fast] & 1) == 1){  
            t = a[fast]; a[fast] = a[low]; a[low] = t;  
            low++;          // 随后low++, 表示下个奇数的位置应该是当前奇数low的下个位置  
        }  
}
```


奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。

[3 2 6 1 5 4]

↑
left

奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。

[3 2 6 1 5 4]

↑
left

奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。

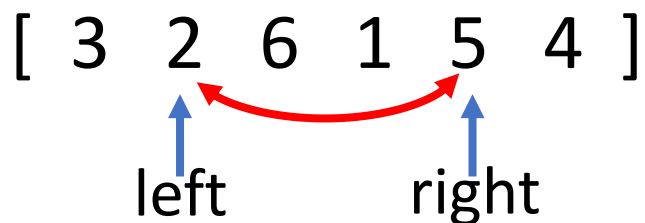
[3 2 6 1 5 4]

 ↑ ↑

 left right

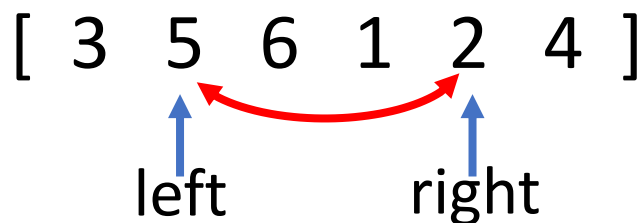
奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。



奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。



奇偶调序-思路4：首尾指针

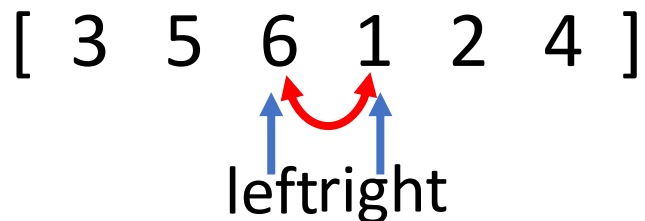
- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。

[3 5 6 1 2 4]

↑ ↑
left right

奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。



奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。

[3 5 1 6 2 4]



奇偶调序-思路4：首尾指针

- 双指针，一个指针从前往后，每当碰到偶数就停下，另一个指针从后往前，每当碰到奇数就停下，然后进行奇偶交换，类似快速排序的一次 partition 操作。

[3 5 1 6 2 4]



当right指针位于left指针的左边时，结束！

```
void sortOddEven(int a[],int n) {  
    for(int left = 0, right = n - 1; left < right; ){  
        //左指针从左向右，遇到偶数停下来  
        while(left<right && (a[left] & 1) != 0)  
            left++;  
        //右指针从右向左，遇到奇数停下来  
        while(left<right && (a[right] & 1) == 0)  
            right--;  
        auto t = a[left];  
        a[left] = a[right];  
        a[right] = t;  
    }  
}
```

$F(n) = n$

空间复杂度为 $O(1)$

思考题

- 调整链表顺序使奇数位于偶数前面。

删除有序数组中的重复项

- 给你一个升序排列的数组`nums`，请你原地删除重复出现的元素，使每个元素只出现一次，返回删除后数组的新长度。元素的相对顺序应该保持一致。

输入： `nums = [1,1,2]`

输出： `2, nums = [1,2,_]`

解释： 函数应该返回新的长度 `2`，并且原数组 `nums` 的前两个元素被修改为 `1, 2`。不需要考虑数组中超出新长度后面的元素。

删除有序数组中的重复项

- 给你一个升序排列的数组nums，请你原地删除重复出现的元素，使每个元素只出现一次，返回删除后数组的新长度。元素的相对顺序应该保持一致。

输入：nums = [0,0,1,1,1,2,2,3,3,4]

输出：5, nums = [0,1,2,3,4]

蛮力法：相邻元素重复则删除后面元素

- 蛮力法：迭代每个元素，如何后面的元素和该元素相同，则将后面的元素向前移动一个位置

[0, 0, 1, 1, 1, 2, 2, 3, 3, 4]

↑
i

蛮力法：相邻元素重复则删除后面元素

- 蛮力法：迭代每个元素，如何后面的元素和该元素相同，则将后面的元素向前移动一个位置

[0, 1, 1, 1, 2, 2, 3, 3, 4]

↑
i

蛮力法：相邻元素重复则删除后面元素

- 蛮力法：迭代每个元素，如何后面的元素和该元素相同，则将后面的元素向前移动一个位置

[0, 1, 1, 1, 2, 2, 3, 3, 4]

↑
i

蛮力法：相邻元素重复则删除后面元素

- 蛮力法：迭代每个元素，如何后面的元素和该元素相同，则将后面的元素向前移动一个位置

[0, 1, 1, 2, 2, 3, 3, 4]

↑
i

蛮力法：相邻元素重复则删除后面元素

- 蛮力法：迭代每个元素，如何后面的元素和该元素相同，则将后面的元素向前移动一个位置

[0, 1, 2, 2, 3, 3, 4]

↑
i

蛮力法：相邻元素重复则删除后面元素

- 蛮力法：迭代每个元素，如何后面的元素和该元素相同，则将后面的元素向前移动一个位置

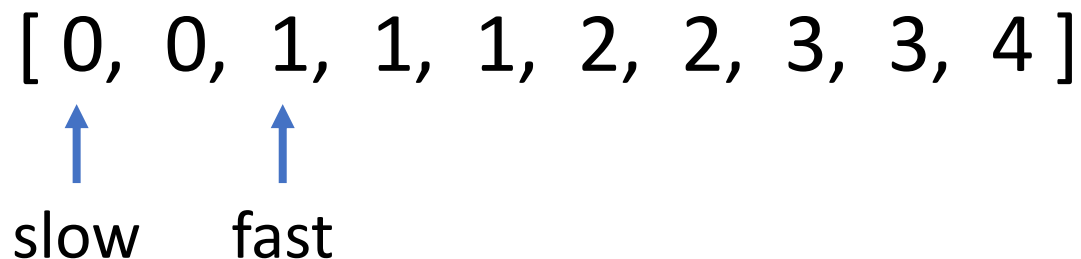
[0, 1, 2, 2, 3, 3, 4]

↑
i

$$T(n) = \Theta(n^2)$$

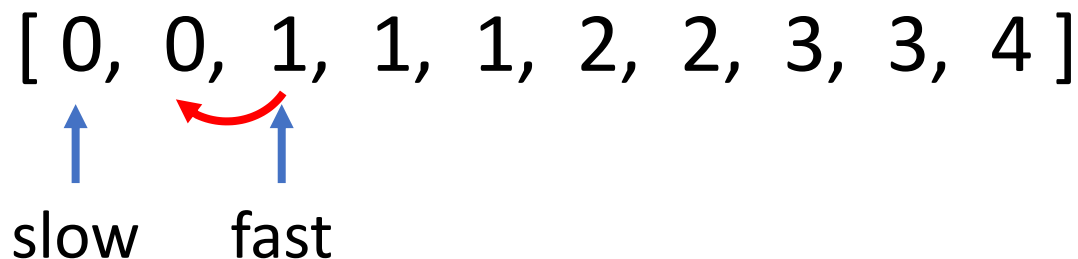
快慢指针：

- **slow**指针指向当前元素，**fast**指针指向其后的第1个不重复元素。



快慢指针：

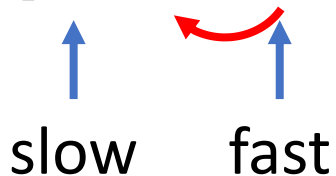
- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将**fast**元素放到**slow+1**的位置。



快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 1, 1, 1, 2, 2, 3, 3, 4]



快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 1, 1, 1, 2, 2, 3, 3, 4]

↑ ↑
slow fast

快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 1, 1, 1, 2, 2, 3, 3, 4]

↑ ↑
slow fast

快慢指针：

- `slow`指针指向当前元素，`fast`指针指向其后的第1个不重复元素。将`fast`元素放到`slow+1`的位置，`slow = slow+1`。

[0, 1, 1, 1, 1, 2, 2, 3, 3, 4]

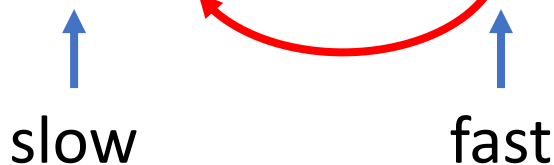
↑
`slow`

↑
`fast`

快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 1, 1, 1, 2, 2, 3, 3, 4]



快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 1, 1, 2, 2, 3, 3, 4]

↑
slow

↑
fast



快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 1, 1, 2, 2, 3, 3, 4]

↑ ↑
slow fast

快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 1, 1, 2, 2, 3, 3, 4]

↑
slow

↑
fast

快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 1, 1, 2, 2, 3, 3, 4]

↑
slow

↑
fast

快慢指针:

- slow指针指向当前元素, fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 1, 1, 2, 2, 3, 3, 4]



```
if( a[fast]!=a[slow] ){  
    slow++;  
    a[slow] = a[fast];  
}
```

快慢指针:

- slow指针指向当前元素, fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 3, 1, 2, 2, 3, 3, 4]



```
if( a[fast]!=a[slow] ){  
    slow++;  
    a[slow] = a[fast];  
}
```


快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 3, 1, 2, 2, 3, 3, 4]

↑
slow

↑
fast

快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 3, 1, 2, 2, 3, 3, 4]

↑
slow

↑
fast

快慢指针:

- slow指针指向当前元素, fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 3, 1, 2, 2, 3, 3, 4]

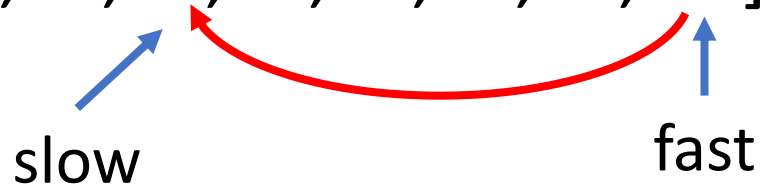


```
if( a[fast]!=a[slow] ){  
    slow++;  
    a[slow] = a[fast];  
}
```

快慢指针:

- slow指针指向当前元素, fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 3, 4, 2, 2, 3, 3, 4]



```
if( a[fast]!=a[slow] ){  
    slow++;  
    a[slow] = a[fast];  
}
```

快慢指针：

- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置,slow = slow+1。

[0, 1, 2, 3, 4, 2, 2, 3, 3, 4]

↑
slow

↑
fast

快慢指针：

- `slow`指针指向当前元素，`fast`指针指向其后的第1个不重复元素。将`fast`元素放到`slow+1`的位置，`slow = slow+1`。

[0, 1, 2, 3, 4, 2, 2, 3, 3, 4]

↑
`slow`

↑
`fast`

快慢指针：

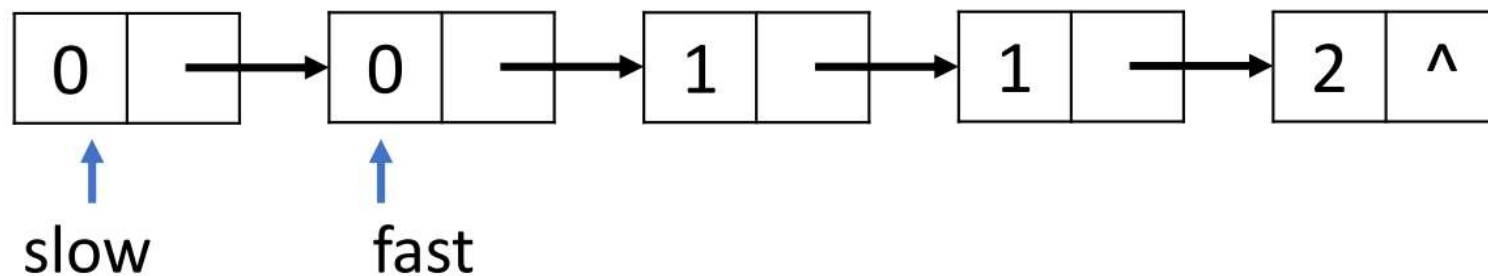
- slow指针指向当前元素，fast指针指向其后的第1个不重复元素。将fast元素放到slow+1的位置，slow = slow+1。

[0, 1, 2, 3, 4, 2, 2, 3, 3, 4]

slow fast

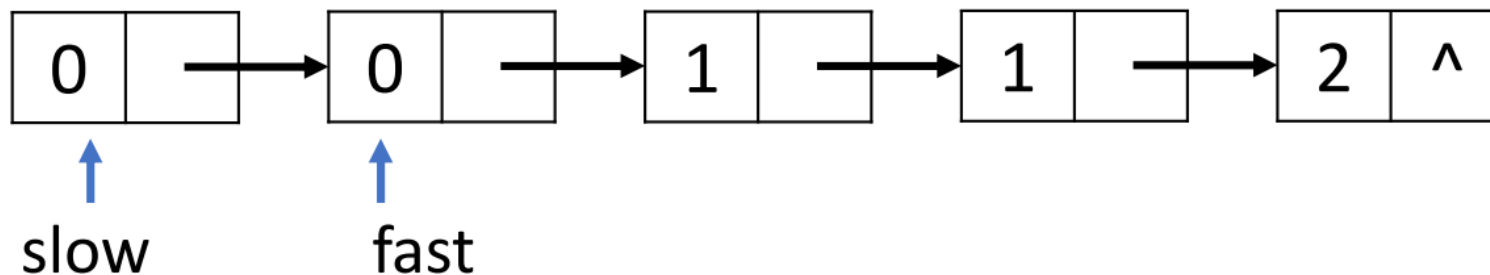
```
int slow=0, fast=0;
for(;fast<n;fast++)
    if( a[fast]!=a[slow] ){
        a[++slow] = a[fast];
    }
return slow+1;
```

- 对于有序的链表，算法过程和代码是类似的：



```
ListNode *slow,*fast = head->next;
```

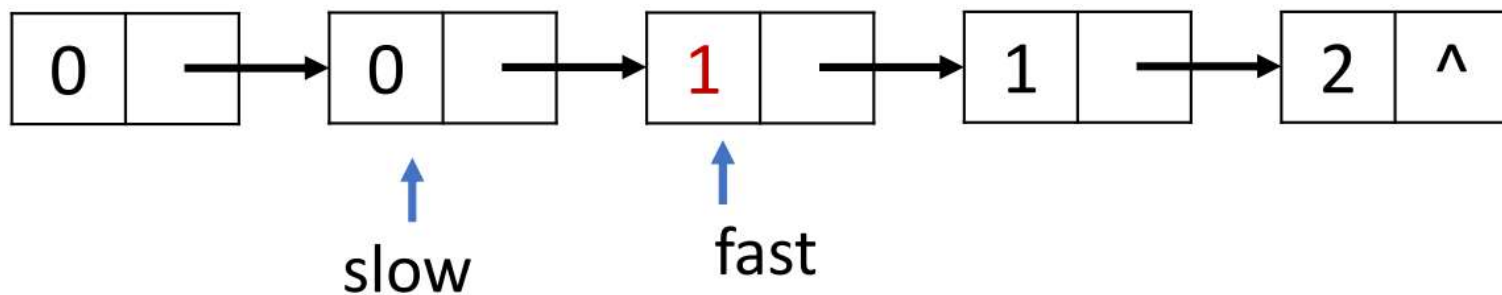

- 对于有序的链表，算法过程和代码是类似的：



```
ListNode *slow,*fast = head->next;  
while(fast!=nullptr){
```

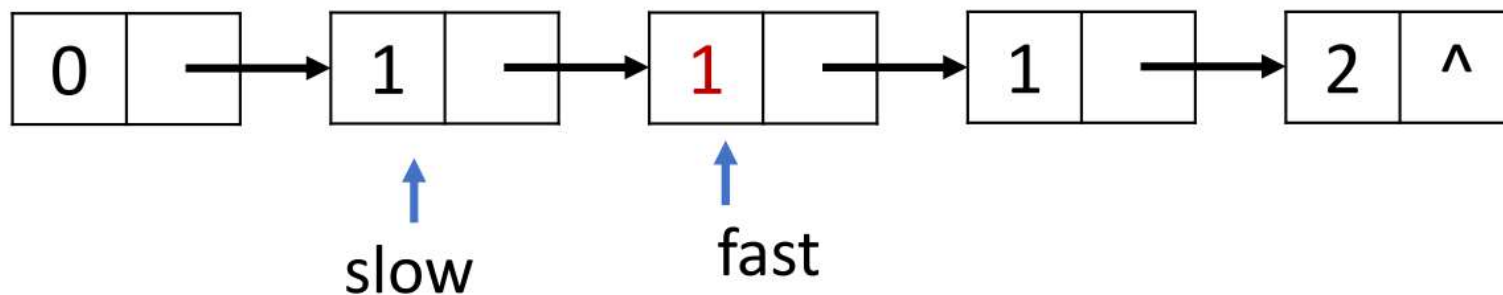
```
    fast = fast->next;  
}
```

- 对于有序的链表，算法过程和代码是类似的：



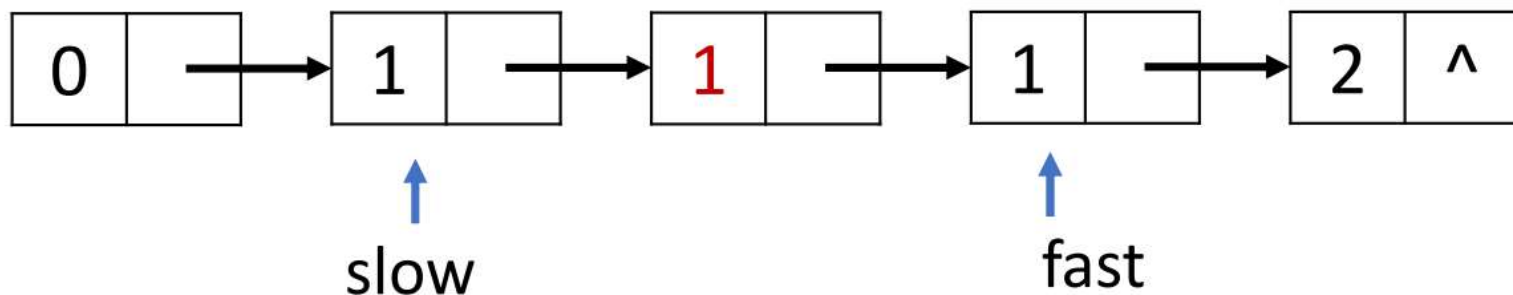
```
ListNode *slow,*fast = head->next;
while(fast!=nullptr){
    if(fast->val != slow->val){
        slow = slow->next;
    }
    fast = fast->next;
}
```

- 对于有序的链表，算法过程和代码是类似的：



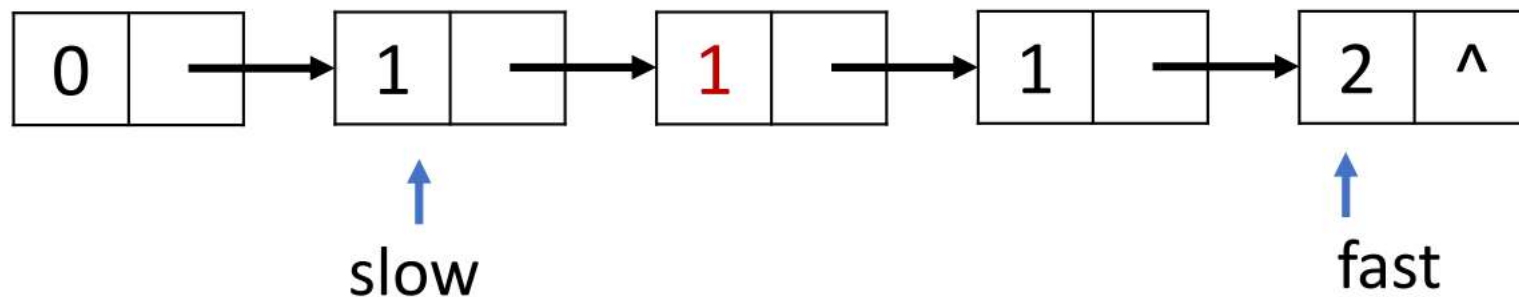
```
ListNode *slow,*fast = head->next;
while(fast!=nullptr){
    if(fast->val != slow->val){
        slow = slow->next;
        slow->val = fast->val;
    }
    fast = fast->next;
}
```

- 对于有序的链表，算法过程和代码是类似的：



```
ListNode *slow,*fast = head->next;
while(fast!=nullptr){
    if(fast->val != slow->val){
        slow = slow->next;
        slow->val = fast->val;
    }
    fast = fast->next;
}
```

- 对于有序的链表，算法过程和代码是类似的：

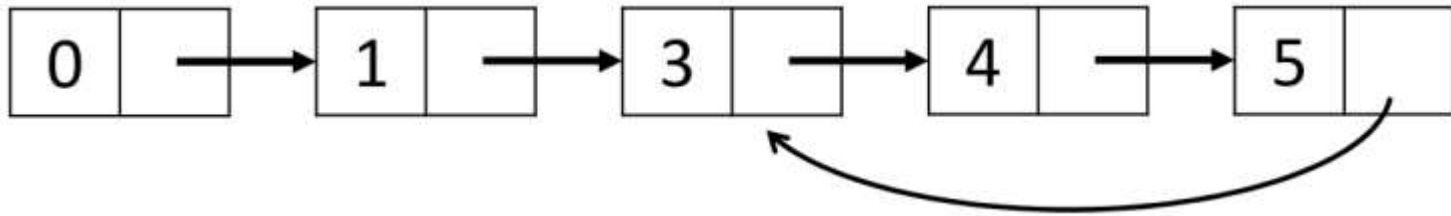


```
ListNode *slow,*fast = head->next;
while(fast!=nullptr){
    if(fast->val != slow->val){
        slow = slow->next;
        slow->val = fast->val;
    }
    fast = fast->next;
}
```

```
ListNode* removeDuplicates(ListNode *head) {  
    if (head == nullptr) return nullptr;  
    ListNode *slow = head, *fast = head->next;  
    while (fast != nullptr) {  
        if (fast->val != slow->val) {  
            slow = slow->next; // slow++;  
            slow->val = fast->val; // nums[slow] = nums[fast];  
        }  
        fast = fast->next; // fast++;  
    }  
    // 断开与后面重复元素的连接  
    slow->next = nullptr;  
    return head;  
}
```

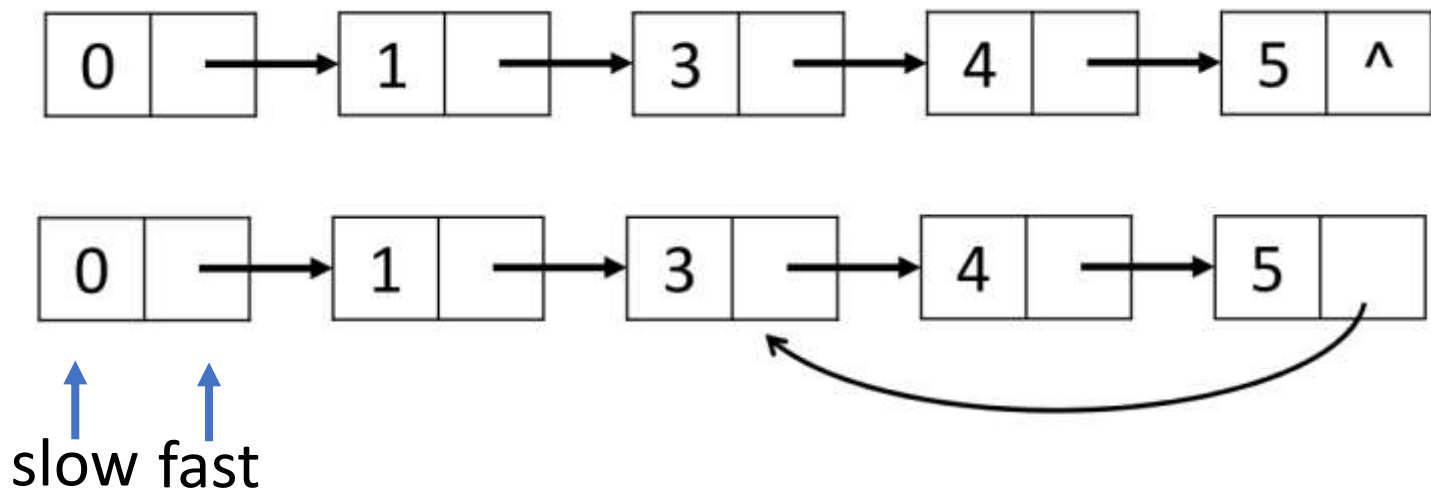
判断链表是否包含环

- 何判断链表中是否有环并找出环的入口位置。
(Leetcode 42)



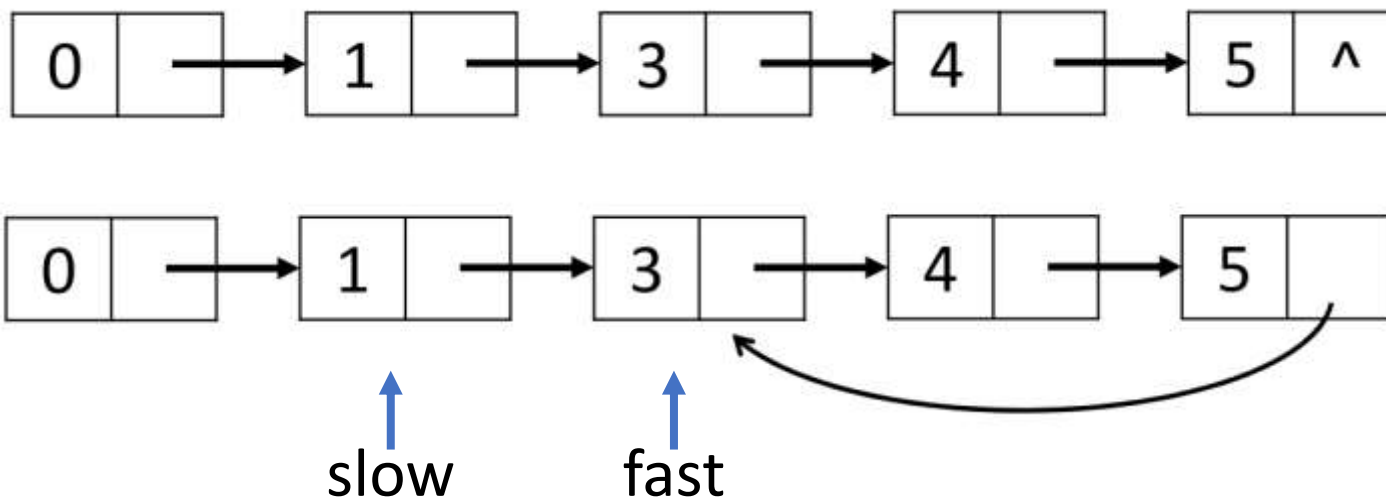
快慢指针

- **slow** 指针走一步，**fast** 指针走两步。如果可以在某一个点满足 **slow=fast**，那么就说明存在环，否则 **fast** 指针必然先到终点



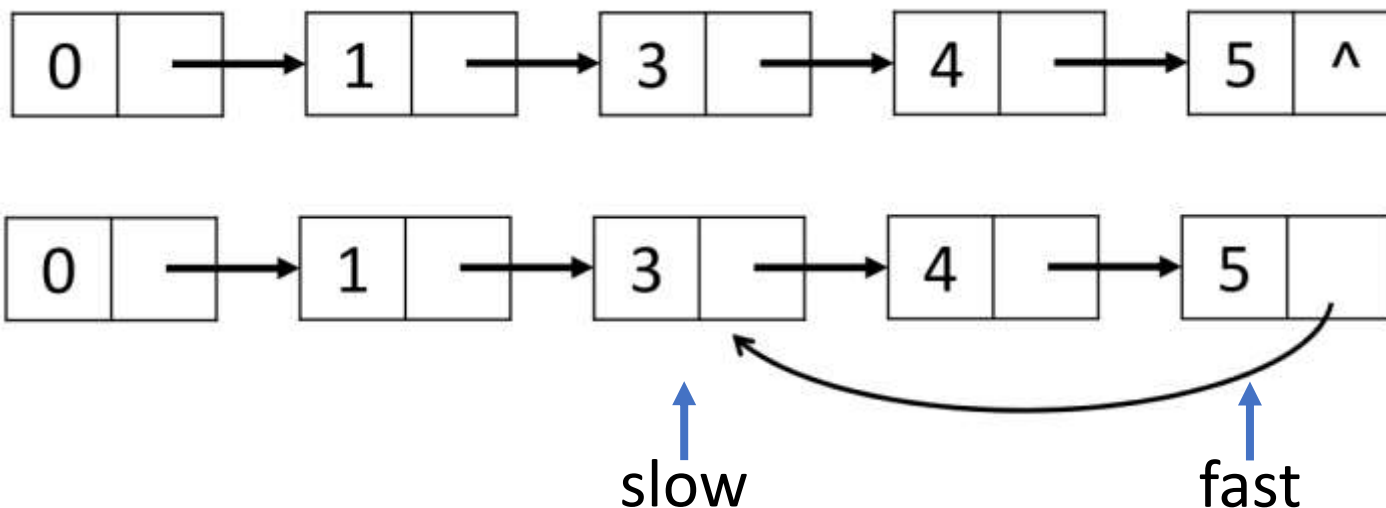
快慢指针

- **slow** 指针走一步，**fast** 指针走两步。如果可以在某一个点满足 **slow=fast**，那么就说明存在环，否则 **fast** 指针必然先到到终点



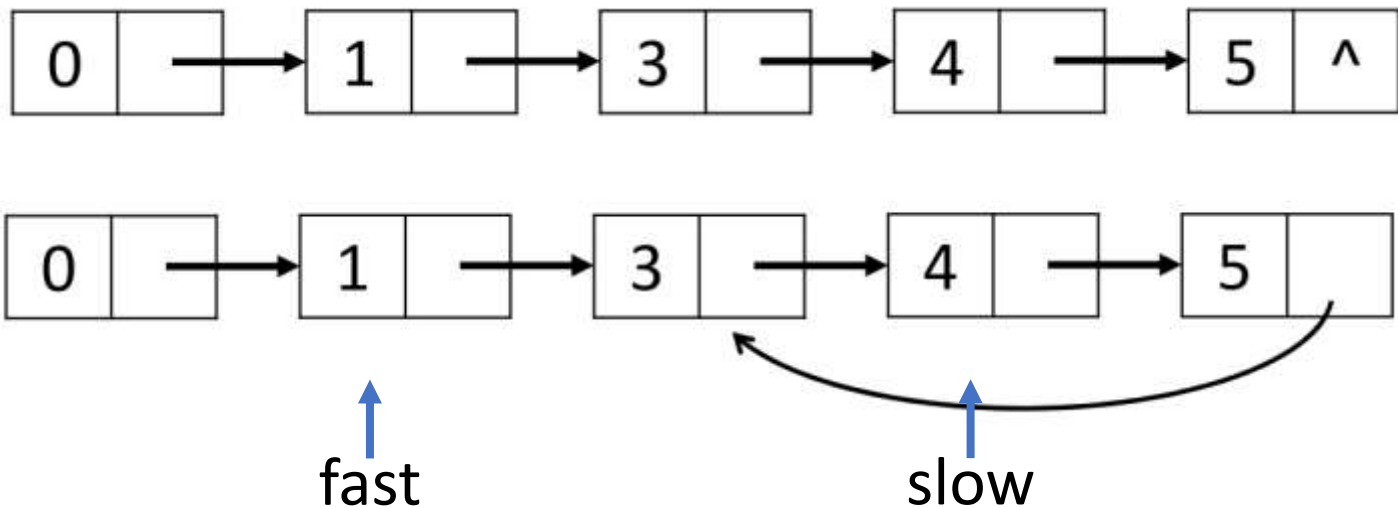
快慢指针

- **slow** 指针走一步，**fast** 指针走两步。如果可以在某一个点满足 **slow=fast**，那么就说明存在环，否则 **fast** 指针必然先到到终点



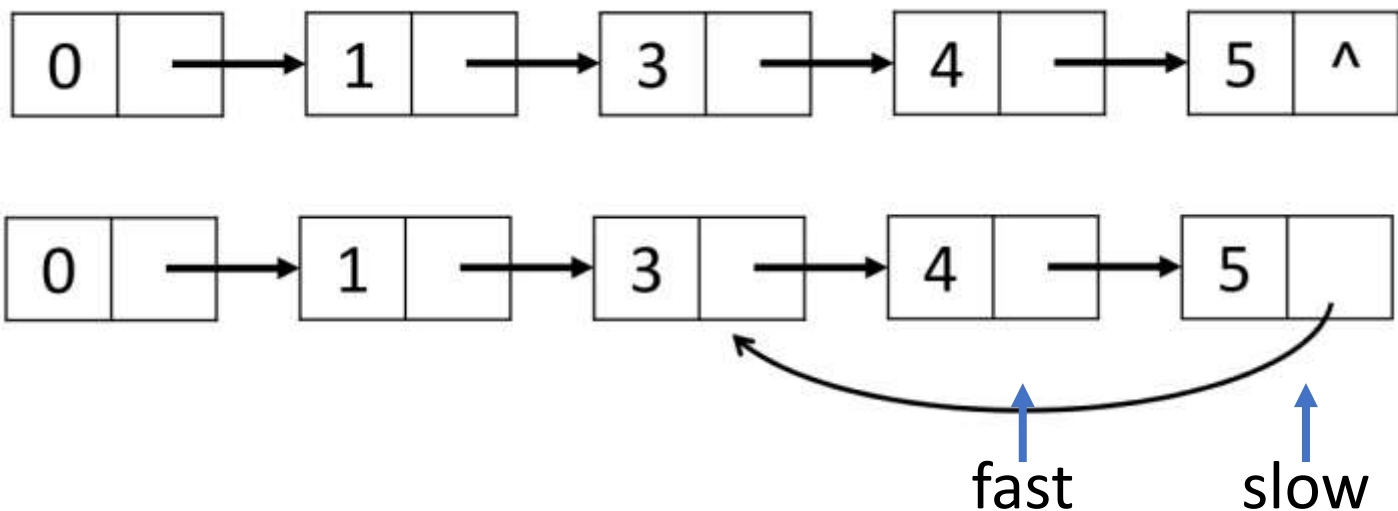
快慢指针

- **slow** 指针走一步，**fast** 指针走两步。如果可以在某一个点满足 **slow=fast**，那么就说明存在环，否则 **fast** 指针必然先到到终点



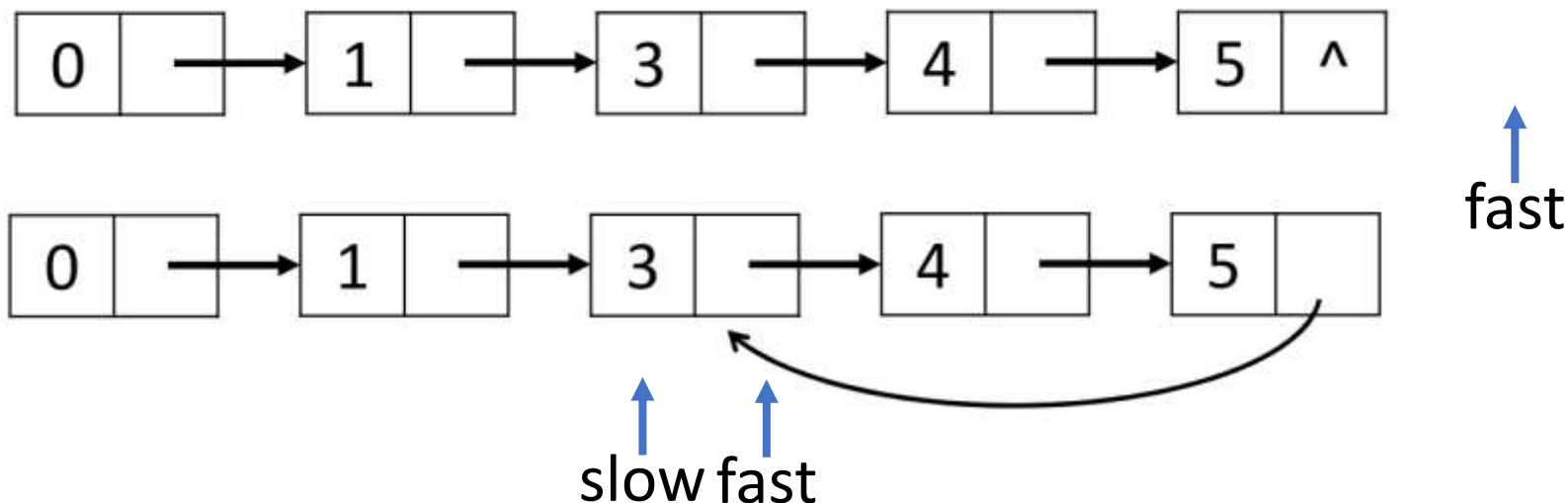
快慢指针

- **slow** 指针走一步，**fast** 指针走两步。如果可以在某一个点满足 **slow=fast**，那么就说明存在环，否则 **fast** 指针必然先到到终点



快慢指针

- **slow** 指针走一步，**fast** 指针走两步。如果可以在某一个点满足 **slow=fast**，那么就说明存在环，否则 **fast** 指针必然先到到终点



```
#include <iostream>
using namespace std;

class ListNode{
public:
    int val;
    ListNode *next;
    ListNode(){next = nullptr;}
    ListNode(int val=0){this->val = val; next = nullptr;}
};
```

```
bool hasCycle(ListNode *head){  
    if ( !head || !head->next){  
        return false;  
    }  
    ListNode *slow = head,*fast = head;  
    while(fast && fast->next){//注意这个条件, 要防止空指针  
        slow = slow->next;//slow 指针一次一步  
        fast = fast->next->next;//fast指针一次两步  
        if (slow == fast){  
            return true;  
        }  
    }  
    return false;  
}
```

```
int main(){
    ListNode *head = new ListNode(0);
    ListNode *p= head;
    p->next = new ListNode(1); p = p->next;
    p->next = new ListNode(2); p = p->next;
    p->next = new ListNode(3); p = p->next;
    p->next = new ListNode(4); p = p->next;
    p->next = new ListNode(5); p = p->next;

    cout<<hasCycle(head)<<endl;
    p->next = head;
    cout<<hasCycle(head)<<endl;
}
```

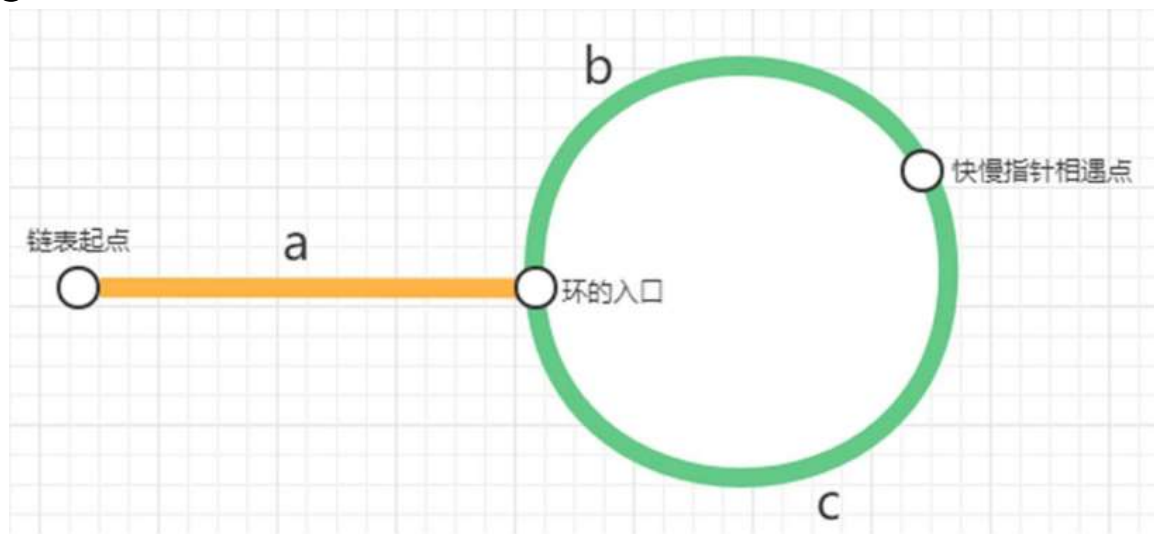

环的入口在哪里？

- 假设快慢指针第一次相遇时，快指针走了 n 圈。则：

$$a+n(b+c)+b = 2(a+b)$$

$$n(b+c) = a+b$$

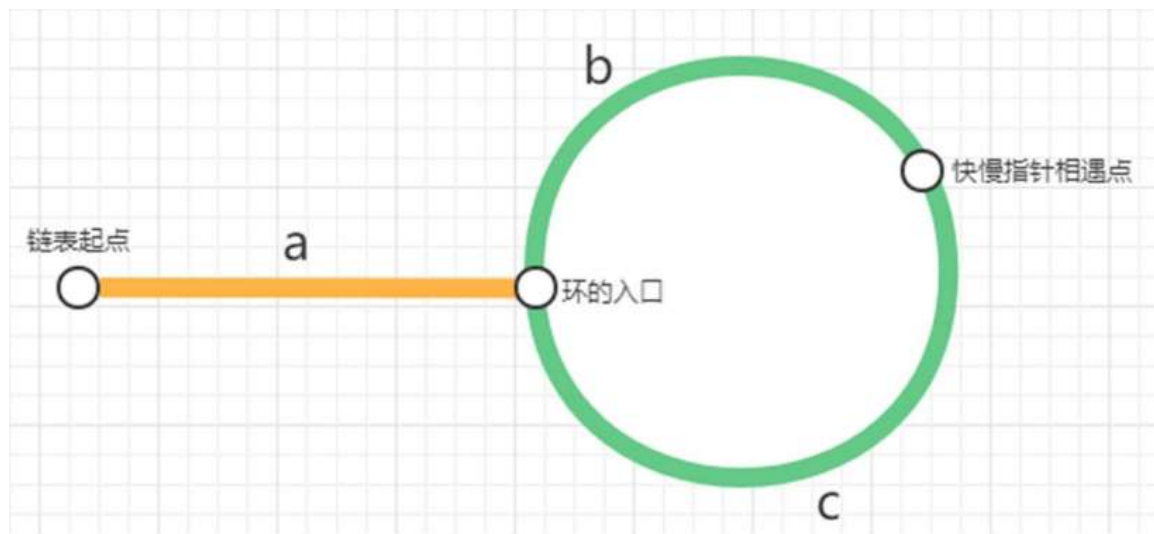
$$a = (n-1)(b+c)+c$$



环的入口在哪里？

$$a = (n-1)(b+c) + c$$

- 如果再让一个指针p（一次一步）从开头行走，而慢指针slow从相遇处继续行走。当p达到环入口时，走了a步，而slow走了 $(n-1)(b+c) + c$ 。正好也到达入口



```
ListNode* hasCycle(ListNode *head){
    if ( !head || !head->next){
        return nullptr;
    }
    ListNode *slow = head,*fast = head;
    while(fast && fast->next){//注意这个条件，要防止空指针
        slow = slow->next;//slow 指针一次一步
        fast = fast->next->next;//fast指针一次两步
        if (slow == fast){
            ListNode *p = head;//定义一个新指针
            while (p != slow){
                p = p->next;
                slow = slow->next;
            }
            return p;//返回环的入口位置
        }
    }
    return nullptr;
}
```

练习

- 练习编写有序数组的二分查找算法。
- 删除有序链表中的重复元素。
- 编写一个函数，其作用是将输入的字符串反转过来。
如：“hwdong”变成“gnodwh”

关注我

博客: hwdong-net.github.io

Youtube频道



hwdong

@hwdong · 5.01K subscribers · 558 videos

博客: <https://hwdong-net.github.io> >

youtube.com/c/4kRealSound and 4 more links