# 概述

双指针算法是指在遍历数组时，使用两个指针来协助解决问题的一种算法。这两个指针可以是数组的两端，也可以是同一侧移动的两个指针。双指针算法通常用于寻找数组中满足特定条件的子数组、链表等问题。

常见的双指针算法包括：

1、快慢指针：一种特殊的双指针算法，通常用于链表中判断是否有环、找到环的入口等问题。

2、左右指针：在数组中，左指针一般从起始位置开始，右指针一般从末尾位置开始，根据问题的要求移动指针，解决问题。

3、滑动窗口：通过维护一个区间（窗口），根据题目要求移动窗口的左右边界，来寻找符合条件的子数组或子串。

4、对撞指针：两个指针分别从数组的两端开始向中间移动，根据题目的要求移动指针，通常用于有序数组或链表的问题。

双指针算法的优点是简单高效，时间复杂度通常为 O(n)。它们常用于数组和链表等数据结构的问题，例如求两数之和、反转数组、移除重复元素等。

# 特点

双指针算法是一种高效的解决问题的方法，但也有其优缺点和适用场景。

## 优点

优点：

1、时间复杂度低：双指针算法通常能够在 O(n) 的时间复杂度内解决问题，效率较高。

2、空间复杂度低：双指针算法通常只需要常数级别的额外空间，不需要额外的数据结构存储中间结果。

3、实现简单：双指针算法的实现比较直观，容易理解和编写。

## 缺点

缺点：

1、适用场景有限：双指针算法通常适用于一些特定类型的问题，如数组或链表相关的问题，对于其他类型的问题可能不太适用。

2、不适合所有情况：有些问题并不容易用双指针算法解决，可能需要其他更复杂的算法来解决。

# 原理

# 适用场景

适用场景：

1、有序数组或链表的问题：对撞指针常用于有序数组或链表中的查找、求和等问题。

2、滑动窗口问题：滑动窗口算法通常使用双指针来维护窗口的边界，用于求解子数组或子串的问题。

3、快慢指针问题：快慢指针算法通常用于链表中判断是否有环、找到环的入口等问题。

总的来说，双指针算法在某些特定的问题场景下具有较高的效率和简洁性，但并不适用于所有类型的问题。在解决问题时，需要根据具体情况选择合适的算法策略。

# 应用

## 遍历

### 两个数组的交集

### 两个数组的交集II

### 轮转数组

注：LeetCode 189

### 验证回文串

给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

说明：本题中，我们将空字符串定义为有效的回文串。

注：LeetCode 125

分析：

采用双指针。

### 验证回文串 II

给你一个字符串 s，最多 可以从中删除一个字符。

请你判断 s 是否能成为回文字符串：如果能，返回 true ；否则，返回 false 。

注：LeetCode 680

分析：

### 回文链表

请判断一个链表是否为回文链表。

给你一个单链表的头节点head，请你判断该链表是否为回文链表。如果是，返回true；否则，返回false。

注：LeetCode 234

分析：

可以采用两种方式：

1、用快慢指针遍历的同时翻转前半部分，然后与后半部分比较即可。

class Solution {

public:

bool isPalindrome(ListNode\* head) {

if(!head || !head->next)

return 1;

ListNode \*fast = head, \*slow = head;

ListNode \*p, \*pre = NULL;

while(fast && fast->next){

p = slow;

slow = slow->next; //快慢遍历

fast = fast->next->next;

p->next = pre; //翻转

pre = p;

}

if(fast) //奇数个节点时跳过中间节点

slow = slow->next;

while(p){ //前半部分和后半部分比较

if(p->val != slow->val)

return 0;

p = p->next;

slow = slow->next;

}

return 1;

}

};

2、栈（涉及到回文的可以用栈）

### 环形链表

给定一个链表，判断链表中是否有环。

为了表示给定链表中的环，我们使用整数pos来表示链表尾连接到链表中的位置（索引从0开始）。 如果pos是-1，则在该链表中没有环。

注：LeetCode 141

分析：

可以采用两种方法：

1、哈希表：将节点存储起来，如果第二次遍历到相同节点，则表示环形

2、快慢指针

通过使用具有不同速度的快、慢两个指针遍历链表，空间复杂度可以被降低至O(1)。慢指针每次移动一步，而快指针每次移动两步。

如果列表中不存在环，最终快指针将会最先到达尾部，此时我们可以返回 false。

代码：

class Solution {

public:

bool hasCycle(ListNode \*head) {

if(nullptr == head || nullptr == head->next)

return false;

ListNode \*slow,\*fast;

slow = head;

fast = head->next;

while(slow != fast)

{

if(nullptr==fast || nullptr==fast->next)

return false;

slow = slow->next;

fast = fast->next->next;

}

return true;

}

};

### 环形链表 II

注：LeetCode 142

### 相交链表

编写一个程序，找到两个单链表相交的起始节点。

如下面的两个链表：



在节点 c1 开始相交。

注：LeetCode 160

分析：

对于这种判断是否存在相交的情况，都可采用两种方式实现：

1、哈希表

2、快慢指针

### 分割链表

注：LeetCode 86

### 旋转链表

注：LeetCode 61

### 重排链表

注：LeetCode 143

### 排序链表

注：LeetCode 148

## 查找

### 实现strstr/找出字符串中第一个匹配项的下标

实现strStr()函数。

给定一个haystack字符串和一个needle字符串，在haystack字符串中找出needle字符串出现的第一个位置 (从0开始)。如果不存在，则返回-1。

注：LeetCode 28

分析：

采用双指针实现，主要是处理不匹配的时候两个指针的重定位：

class Solution {

public:

int strStr(string haystack, string needle) {

int i = 0, j = 0;

// 双指针

while (i < haystack.size() && j < needle.size()) {

if (haystack[i] == needle[j]) {

i++;

j++;

} else {

// 返回haystack的下一个位置:i-j+1

i = i - j + 1;

j = 0;

}

}

// 如果j为needle的大小则代表完全匹配了,返回haystack中的位置:i-j

if (j == needle.size()) {

return i - j;

}

return -1;

}

};

### 寻找重复数

注：LeetCode 287

## 删除

### 删除有序数组中的重复项

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用O(1)额外空间的条件下完成。

注：LeetCode 26

分析：

使用双指针/快慢指针实现：

class Solution {

public:

int removeDuplicates(vector<int>& nums) {

if (nums.empty()) return 0; // 处理空数组的情况

int i = 0; // 定义慢指针 i

for (int j = 1; j < nums.size(); ++j) { // 定义快指针 j，从第二个元素开始遍历（第一个肯定不会重复）

if (nums[i] != nums[j]) { // 如果快慢指针所指元素不相同

nums[++i] = nums[j]; // 将快指针指向的元素赋值给慢指针的下一个位置，并移动慢指针

}

}

return i + 1; // 返回不重复元素的个数（加上第一个元素nums[0]）

}

};

### 移除元素

给你一个数组 nums和一个值 val，你需要 原地 移除所有数值等于val的元素，并返回移除后数组的新长度。

不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并 原地 修改输入数组。

元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

注：LeetCode 27

分析：

使用双指针/快慢指针实现：

class Solution {

public:

int removeElement(vector<int>& nums, int val) {

int i = 0;

for(int j=0;j<nums.size();j++)

{

if(nums.at(j) != val)

{

nums[i] = nums[j];

i++;

}

}

return i;

}

};

说明：上述代码与LeetCode 26的不一样，26是删除重复的，因此第一个肯定不重复，只需要从第2个开始，这里是判断是否等于指定的val，因此需要从0开始遍历。

### 删除链表的倒数第 N 个结点

注：LeetCode 19

### 删除有序数组中的重复项 II

注：LeetCode 80

### 删除排序链表中的重复元素 II

注：LeetCode 82

### 删除最短的子数组使剩余数组有序

注：LeetCode 1574

### 删除链表的中间节点

注：LeetCode 2095

## 合并

### 合并两个有序数组

给你两个有序整数数组nums1 和 nums2，请你将 nums2 合并到nums1中，使 nums1 成为一个有序数组。

注：LeetCode 88

分析：

采用模拟或双指针实现：

class Solution {

public:

void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

int i = m - 1, j = n - 1, k = m + n - 1; // 初始化指针，分别指向 nums1 的末尾（有效元素的最后一个位置）、nums2 的末尾、合并后数组的末尾

while (i >= 0 && j >= 0) { // 从后往前比较 nums1 和 nums2 的元素

if (nums1[i] > nums2[j]) { // 如果 nums1 的当前元素大于 nums2 的当前元素

nums1[k--] = nums1[i--]; // 将 nums1 的当前元素放入合并后数组的末尾，并将对应指针向前移动一位

} else {

nums1[k--] = nums2[j--]; // 如果 nums2 的当前元素大于等于 nums1 的当前元素，将 nums2 的当前元素放入合并后数组的末尾，并将对应指针向前移动一位

}

}

while (j >= 0) { // 如果 nums2 还有剩余元素

nums1[k--] = nums2[j--]; // 将 nums2 剩余的元素依次填入合并后数组的前面

}

}

};

## 反转

### 翻转链表

### 反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[]的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用O(1)的额外空间解决这一问题。

你可以假设数组中的所有字符都是ASCII码表中的可打印字符。

注：LeetCode 344

分析：

class Solution {

public:

void reverseString(vector<char>& s) {

int len=s.size();

int i=0,j=len-1;

while(i<j)

{

char tmp;

tmp = s[i];

s[i] = s[j];

s[j] = tmp;

i++;

j--;

}

}

};

### 反转字符串中的元音字母

编写一个函数，以字符串作为输入，反转该字符串中的元音字母。

注：LeetCode 345

分析：

class Solution {

public:

string reverseVowels(string s) {

std::set<char> a;

a.insert('A');

a.insert('E');

a.insert('I');

a.insert('O');

a.insert('U');

a.insert('a');

a.insert('e');

a.insert('i');

a.insert('o');

a.insert('u');

int i=0,j=s.size()-1;

while(i<j)

{

while(!(a.find(s[i])!=a.end()) && i<j)

i++;

while(!(a.find(s[j])!=a.end()) && i<j)

j--;

swap(s[i++],s[j--]);

}

return s;

}

};

### 反转字符串 II

给定一个字符串s和一个整数k，从字符串开头算起，每计数至2k个字符，就反转这2k字符中的前k个字符。

如果剩余字符少于k个，则将剩余字符全部反转。

如果剩余字符小于2k但大于或等于k个，则反转前k个字符，其余字符保持原样。

注：LeetCode 541

分析：

你可以使用双指针来实现这个算法。具体步骤如下：

1、初始化变量 i 和 j 分别为 0 和 k-1，表示当前处理的区间为 [i, j]。

2、循环遍历字符串，每次处理一个区间 [i, j]：

- 反转区间 [i, j] 的字符。

- 更新 i 和 j 的值为 j + 1 和 min(j + k, n - 1)，继续下一轮循环。

3、返回处理后的字符串。

### 反转字符串中的单词 III

给定一个字符串s，你需要反转字符串中每个单词的字符顺序，同时仍保留空格和单词的初始顺序。

注：LeetCode 557

分析：

### 反转字符串中的单词

注：LeetCode 151

## 数学计算

### 两数之和 II - 输入有序数组

### 三数之和

### 最接近三数之和

### 四数之和

### 平方数之和

## 接雨水