双指针算法

> **当遇到需要对一个数组进行重复遍历时，可以想到使用双指针法**

常见思想

对撞指针

> 适用于连续数组和字符串

指向最左侧的索引定义为`左指针(left)`，最右侧的定义为`右指针(right)`，然后从两头向中间进行数组遍历。

```
public void find (int[] list) {
  var left = 0;
  var right = list.length - 1;

  //遍历数组
  while (left <= right) {
    left++;
    // 一些条件判断 和处理
    ... ...
    right--;
  }
}
```

快慢指针

从同一侧开始遍历数组，将这两个指针分别定义为`快指针（fast）`和`慢指针（slow）`，两个指针以不同的策略移动，直到两个指针的值相等（或其他特殊条件）为止，如 fast 每次增长两个，slow 每次增长一个。

滑动窗口算法(TODO)

283. 移动零

题目

> 给定一个数组 `nums`，编写一个函数将所有 `0` 移动到数组的末尾，同时保持非零元素的相对顺序。
>
> **请注意**，必须在不复制数组的情况下原地对数组进行操作

示例 1:

```
输入: nums = [0,1,0,3,12]
输出: [1,3,12,0,0]
```

示例 2:

```
输入: nums = [0]
输出: [0]
```

提示:

- `1 <= nums.length <= 104`
- `-231 <= nums[i] <= 231 - 1`

进阶：你能尽量减少完成的操作次数吗?

解法

双指针

核心思想是将值为零的元素与非零元素进行交换，让所有值为零的元素浮到数组末尾。

定义两个指针`l`,`r`分别位于数组第一元素和第二元素，遍历数组，

当`nums[l] == 0 && nums[r] != 0`，交换`swap`值并且`l ++; r++`

当`nums[l] != 0`时, 没有`0`要移动, `l++; r ++`

其他情况, 说明`nums[l] == 0 && nums[r] == 0`需要移动`r`指针指向非零元素再进行交换。

```cpp
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        if (nums.size() < 2) return;
        for (int i = 0, j = 1; j < nums.size(); j ++) {
            if (nums[i] == 0 && nums[j] != 0) {
                swap(nums[i], nums[j]);
                i ++;
            } else if (nums[i] != 0) i ++;
        }
    }
};
```

复杂度分析:

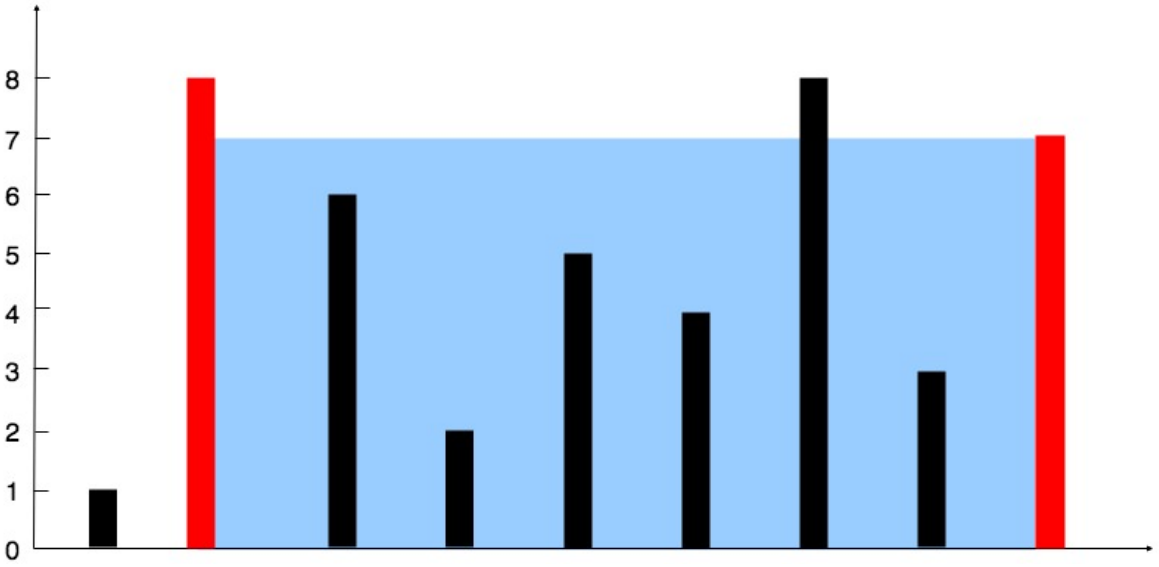
- 时间复杂度 O(n)需要遍历一次链表
- 空间复杂度 O(1)

11. 盛最多水的容器

题目

> 给定一个长度为 `n` 的整数数组 `height` 。有 `n` 条垂线，第 `i` 条线的两个端点是 `(i, 0)` 和 `(i, height[i])` 。
>
> 找出其中的两条线，使得它们与 `x` 轴共同构成的容器可以容纳最多的水。
>
> 返回容器可以储存的最大水量。
>
> **说明**：你不能倾斜容器。

示例 1：




```
输入：[1,8,6,2,5,4,8,3,7]
输出：49
解释：图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。
```

示例 2:

```
输入：height = [1,1]
输出：1
```

提示:

- `n == height.length`
- `2 <= n <= 105`
- `0 <= height[i] <= 104`

解法

暴力迭代（超时）

二维循环遍历`height`数组，维护result = `max(result, (j - i)*min(height[i], height[j]))`

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int result = 0;
        for (int i = 0; i < height.size(); i ++) {
            for (int j = i + 1; j < height.size(); j ++) {
                result = max(result, (j - i)*min(height[i], height[j]));
            }
        }
        return result;
    }
};
```

复杂度分析:

- 时间复杂度 O(n^2^)
- 空间复杂度 O(1)

双指针(对撞指针)

面积的大小决定因素:

- 底边的长度
- 围成容器的高度的`短板`

定义`left`指针和`right`指针分别位于数组的左右两侧

对于`left`指针,

- 当`height[left + 1] > height[left]`时，如果短板是right，那left增大不会增大面积；如果短板是left，left增大可能会增大面积

对于`right`指针, 同理

因此，只有向内移动短板，面积才可能增大。

1. 初始化： 双指针 left , right 分列水槽左右两端；
2. 循环收窄： 直至双指针相遇时跳出；

   a. 更新面积最大值 res ；
   b. 选定两板高度中的短板，向中间收窄一格；
3. 返回值： 返回面积最大值 res即可；

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int left = 0, right = height.size() - 1;
        int result = 0;
        while(left < right) {
            result = max(result, (right - left) * min(height[left],
height[right]));
            if (height[left] < height[right]) left ++;
            else right --;
        }
        return result;
    }
};
```

复杂度分析:

- 时间复杂度 O(n)
- 空间复杂度 O(1)

15. 三数之和

题目

> 给你一个整数数组`nums`，判断是否存在三元组`[nums[i], nums[j], nums[k]]`满足`i != j`、`i != k`且`j != k`，同时还满足`nums[i] + nums[j] + nums[k] == 0`。请
>
> 你返回所有和为`0`且不重复的三元组。
>
> **注意：**答案中不可以包含重复的三元组。

示例 1：

```
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
解释：
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0 。
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0 。
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0 。
不同的三元组是 [-1,0,1] 和 [-1,-1,2] 。
注意，输出的顺序和三元组的顺序并不重要。
```

示例 2：

```
输入：nums = [0,1,1]
输出：[]
解释：唯一可能的三元组和不为 0 。
```

示例 3：

```
输入：nums = [0,0,0]
输出：[[0,0,0]]
解释：唯一可能的三元组和为 0 。
```

提示：

- `3 <= nums.length <= 3000`
- `-105 <= nums[i] <= 105`

解法：

一次迭代+双指针

如果是求两数之和可以直接用对撞指针来一次遍历，但是本题是求三数之和，可以先通过一次迭代确定第一个元素，然后再剩下的数组范围内使用对撞指针求解。

细节：

- 对撞指针需要数组有序。所以需要先对数组进行排序
- 本题求解的三元组要求不重复，可以在第一个元素和第二个元素的迭代时进行去重。

  如果第一个元素满足条件`i > 0 && nums[i] == nums[i - 1]`时说明第一个元素没有变化，那么求得的三元组也没有变化

  如果第二个元素满足条件`l > i + 1 && nums[l] == nums[l - 1]`时说明对于同一个第一个元素，第二个元素没有变化，那么如果三数之和等于0，第三个元素也没有变化，这就是个重复的三元组了，需要去重。

```cpp
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        vector<vector<int>> result;
        for (int i = 0; i < nums.size() - 2; i ++) {
            // 转为两数之和
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            int l = i + 1, r = nums.size() - 1;
            while(l < r) {
                if(l > i + 1 && nums[l] == nums[l - 1]) {
```

```
                    l ++;
                    continue;
                }
                int sum = nums[i] + nums[l] + nums[r];
                if (sum < 0) l ++;
                else if (sum > 0) r --;
                else {
                    result.push_back({nums[i], nums[l], nums[r]});
                    l ++;
                    r --;
                }
            }
        }
        return result;
    }
};
```

复杂度分析:

- 时间复杂度 O(n^2^)
- 空间复杂度