3.1 数组理论基础

数组：==存储在连续内存空间上的相同类型数据的集合==

- 数组的下标都是从0开始的。
- 数组的内存空间的地址是连续的。

二维数组在内存中的空间地址是连续的么?
==在C++中二维数组是连续分布的==

3.2 二分查找

力扣题目：704.二分查找
704. 二分查找 - 力扣（Leetcode）

给定一个 `n` 个元素有序的（升序）整型数组 `nums` 和一个目标值 `target` ，写一个函数搜索 `nums` 中的 `target`，如果目标值存在返回下标，否则返回 `-1`。

示例 1:

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

示例 2:

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不存在 nums 中因此返回 -1
```

提示:

你可以假设 `nums` 中的所有元素是不重复的。

`n` 将在 `[1, 10000]` 之间。

`nums` 的每个元素都将在 `[-9999, 9999]` 之间。

==区间的定义就是“不变量”==
要在二分查找的过程中保持“不变量”
在while循环中，每一次边界的处理都更具区间的定义来操作，这就是“循环不变量”规则。

左闭右闭 `[left,right]`

- `while (left <= right)` 要使用 <= ，因为left == right是有意义的，所以使用 <=
- `if (nums[middle] > target) right` 要赋值为 middle - 1，因为当前这个`nums[middle]`一定不是target，那么接下来要查找的左区间结束下标位置就是 middle - 1

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1; // 定义target在左闭右闭的区间里，[left, right]
        while (left <= right) { // 当left==right，区间[left, right]依然有效，所以用 <=
            int middle = left + (right - left) / 2;// 防止溢出 等同于(left + right)/2
            if (nums[middle] > target) {
                right = middle - 1; // target 在左区间，所以[left, middle - 1]
            } else if (nums[middle] < target) {
                left = middle + 1; // target 在右区间，所以[middle + 1, right]
            } else { // nums[middle] == target
                return middle; // 数组中找到目标值，直接返回下标
            }
        }
        // 未找到目标值
        return -1;
    }
};
```

左闭右开`[left,right)`

- `while (left < right)`，这里使用 < ,因为left == right在区间 `[left, right)`是没有意义的
- `if (nums[middle] > target) right` 更新为 middle，因为当前`nums[middle]`不等于target，去左区间继续寻找，而寻找区间是左闭右开区间，所以right更新为middle，即：下一个查询区间不会去比较`nums[middle]`

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size(); // 定义target在左闭右开的区间里，即：[left, right)
        while (left < right) { // 因为left == right的时候，在[left, right)是无效的空间，
所以使用 <
            int middle = left + ((right - left) >> 1);
            if (nums[middle] > target) {
                right = middle; // target 在左区间，在[left, middle)中
            } else if (nums[middle] < target) {
                left = middle + 1; // target 在右区间，在[middle + 1, right)中
            } else { // nums[middle] == target
                return middle; // 数组中找到目标值，直接返回下标
            }
        }
        // 未找到目标值
        return -1;
```

```
    }
};
```

左开右开(left,right)

- 本质上和前两种方式大同小异，其实主要还是因为除法会舍去右侧边界。

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size()-1;
        while (right - left > 1)//左右的距离必须大于1，保证中间有下标
        {
            int middle = left + ((right - left) >> 1);
            if (nums[middle] > target) {
                right = middle;
            } else if (nums[middle] < target) {
                left = middle;
            } else { // nums[middle] == target
                return middle; // 数组中找到目标值，直接返回下标
            }
        }
        if(nums[left]==target){return left;}
        if(nums[right]==target){return right;}
        // 未找到目标值
        return -1;
    }
};
```

3.3 移除元素

力扣题目：27.移除元素

[27. 移除元素 - 力扣（Leetcode）](#)

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。

说明:

为什么返回数值是整数，但输出的答案是数组呢?

请注意，输入数组是以**「引用」**方式传递的，这意味着在函数里修改输入数组对于调用者是可见的。

你可以想象内部操作如下:

```
// nums 是以“引用”方式传递的。也就是说，不对实参作任何拷贝
int len = removeElement(nums, val);

// 在函数里修改输入数组对于调用者是可见的。
// 根据你的函数返回的长度，它会打印出数组中 该长度范围内 的所有元素。
for (int i = 0; i < len; i++) {
    print(nums[i]);
}
```

示例 1:

输入: nums = [3,2,2,3], val = 3
输出: 2, nums = [2,2]
解释: 函数应该返回新的长度 **2**, 并且 nums 中的前两个元素均为 **2**。你不需要考虑数组中超出新长度后面的元素。例如，函数返回的新长度为 2 ，而 nums = [2,2,3,3] 或 nums = [2,2,0,0]，也会被视作正确答案。

示例 2:

输入: nums = [0,1,2,2,3,0,4,2], val = 2
输出: 5, nums = [0,1,4,0,3]
解释: 函数应该返回新的长度 **5**, 并且 nums 中的前五个元素为 **0**, **1**, **3**, **0**, **4**。注意这五个元素可为任意顺序。你不需要考虑数组中超出新长度后面的元素。

提示:

- `0 <= nums.length <= 100`
- `0 <= nums[i] <= 50`

- 0 <= val <= 100

- 快慢指针
- 时间复杂度O(n),空间复杂度是O(1)

```
class Solution{
public:
    int removeElement(vector<int>&nums,int val){
        int fast = 0;
        int slow = 0;
        while(fast<nums.size())
        {
            if(nums[fast]!=val)
            {
                nums[slow] = nums[fast];
                ++slow;
            }
            ++fast;
        }
        return slow;
    }
};
```

3.4 长度最小的子数组

力扣题目：长度最小的子数组

[209. 长度最小的子数组 - 力扣（Leetcode）]

给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其和 ≥ target 的长度最小的 **连续子数组** $[num s_l, num s_{l+1}, ..., num s_{r-1}, num s_r]$ ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

示例 1：

输入： target = 7, nums = [2,3,1,2,4,3]
输出： 2
解释： 子数组 [4,3] 是该条件下的长度最小的子数组。

示例 2：

输入： target = 4, nums = [1,4,4]
输出： 1

示例 3：

输入： target = 11, nums = [1,1,1,1,1,1,1,1]
输出： 0

提示：

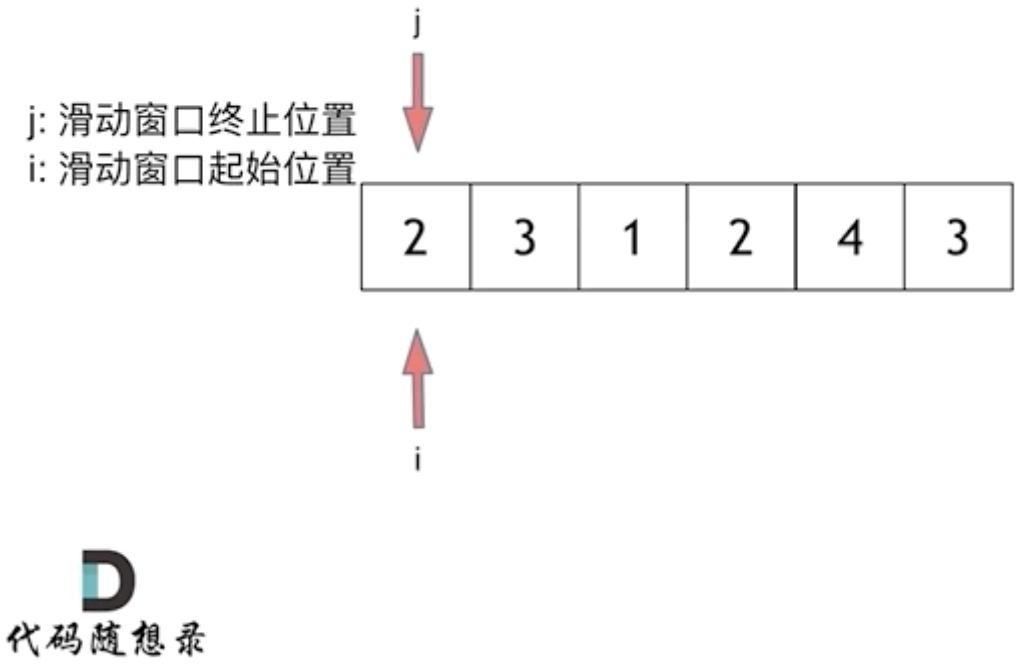
- $1 <= target <= 10^9$
- $1 <= nums.length <= 10^5$
- $1 <= nums[i] <= 10^5$

进阶：

- 如果你已经实现 O(n) 时间复杂度的解法, 请尝试设计一个 O(n log(n)) 时间复杂度的解法。

- 滑动窗口
  不断地调整子数组的起始位置和终止位置，从而得出我们想要的结果。
  本题种有三点
- 窗口内的元素是什么?
  保持窗口内数值总和大于或者等于nums的长度最小的连续子数组。
- 如何移动窗口的起始位置?
  如果当前窗口的值大于nums,则窗口向前移动(也就是窗口该缩小了)。

- 如何移动窗口的终止位置?
  窗口的结束位置就是for循环遍历数组的指针。
  时间复杂度O(n),空间复杂度O(1)




```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        int result = INT32_MAX;
        int sum = 0; // 滑动窗口数值之和
        int start = 0; // 滑动窗口起始位置
        int subLength = 0; // 滑动窗口的长度
        for (int end = 0; end < nums.size(); ++end) {
            sum += nums[end];
            // 注意这里使用while，每次更新 i（起始位置），并不断比较子序列是否符合条件
            while (sum >= target) {
                subLength = (end - start + 1); // 取子序列的长度
                result = result < subLength ? result : subLength;
                sum -= nums[start++]; // 这里体现出滑动窗口的精髓之处，不断变更i（子序列的起
始位置）
            }
        }
        // 如果result没有被赋值的话，就返回0，说明没有符合条件的子序列
        return result == INT32_MAX ? 0 : result;
    }
};
```

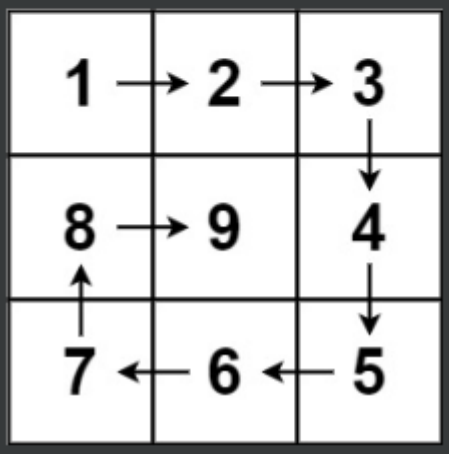
3.5 螺旋矩阵

力扣题目：59.螺旋矩阵II

[59. 螺旋矩阵 II - 力扣（Leetcode）]

给你一个正整数 `n` ，生成一个包含 `1` 到 n^2 所有元素，且元素按顺时针顺序螺旋排列的 `n x n` 正方形矩阵 `matrix` 。

示例 1：



输入： `n = 3`
输出： `[[1,2,3],[8,9,4],[7,6,5]]`

示例 2：

输入： `n = 1`
输出： `[[1]]`

提示：

- `1 <= n <= 20`

- 注意边界条件的变化

```
class Solution
{
public:
    vector< vector<int> > matrix(n,vector<int>(n));
    int Left = 0;//左边界
    int Up =0;//上边界
    int Right = n - 1;//右边界
    int Down = n - 1;//下边界
    int num = 0;//当前数字值
    int max = n * n;//最大数字值
    int i;//操作下标
    while(1)
```

```
    {
        for(i = Left; i <= Right; ++i)
        {matrix[Up][i] = ++num;}
        ++Up;
        if(num == max ){break;}

        for(i = Up; i <= Down; ++i)
        {matrix[i][Right] = ++num;}
        --Right;
        if(num == max ){break;}

        for(i = Right; i >= Left; --i)
        {matrix[Down][i] = ++num;}
        --Down;
        if(num == max ){break;}

        for(i = Down; i >= Up; --i)
        {matrix[i][Left] = ++num;}
        --Left;
        if(num == max ){break;}
    }
    return matrix;
};
```

3.6 有序数组的平方

力扣题目：977.有序数组的平方

[977. 有序数组的平方 - 力扣（Leetcode）]

给你一个按 **非递减顺序** 排序的整数数组 `nums`，返回 **每个数字的平方** 组成的新数组，要求也按 **非递减顺序** 排序。

示例 1：

输入： nums = [-4,-1,0,3,10]
输出： [0,1,9,16,100]
解释： 平方后，数组变为 [16,1,0,9,100]
排序后，数组变为 [0,1,9,16,100]

示例 2：

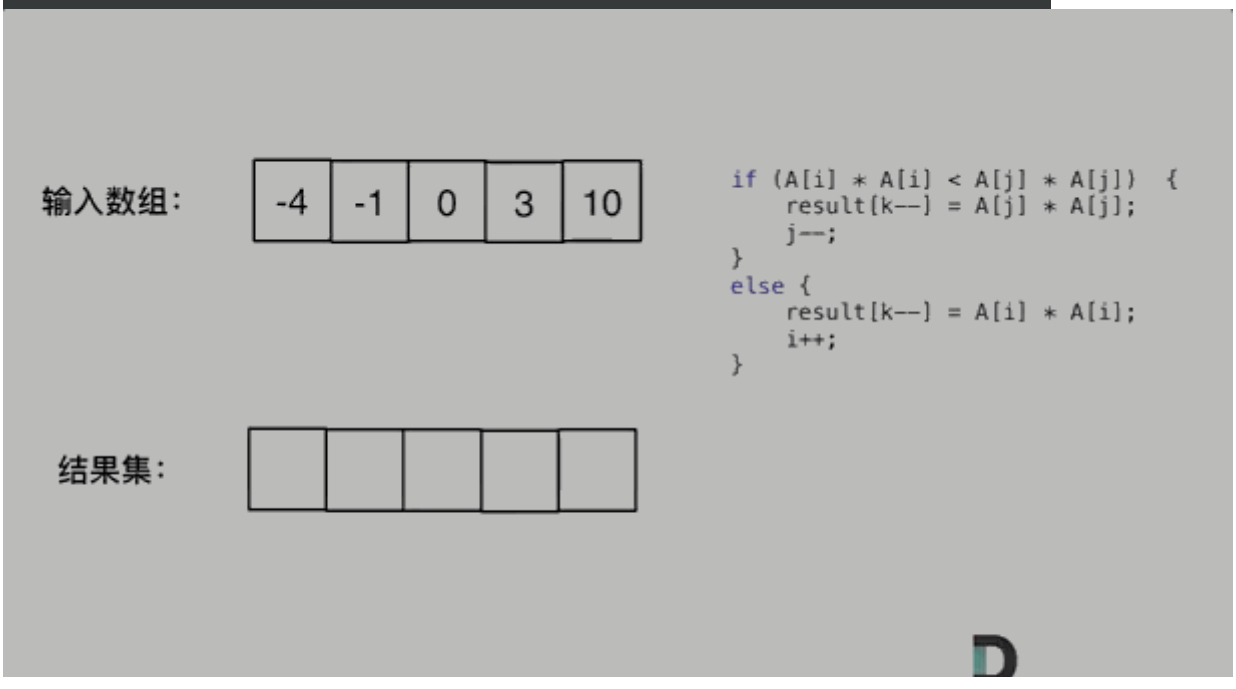
输入： nums = [-7,-3,2,3,11]
输出： [4,9,9,49,121]

提示：

- $1 <= nums.length <= 10^4$
- $-10^4 <= nums[i] <= 10^4$
- `nums` 已按 **非递减顺序** 排序

进阶：

- 请你设计时间复杂度为 `O(n)` 的算法解决本问题



- 双指针，因为正负数本身是有序的。
- 时间复杂度O(n)

```
class Solution{
public:
    vector<int> sortedSquares(vector<int>& nums)
    {
        int left = 0;
        int right = nums.size()-1;
        int pos = right;
        vector<int> result(right+1);

        while(left<=right)
        {
            int L = nums[left]*nums[left];
            int R = nums[right]*nums[right];
            if(L > R)
            {
                result[pos] = L;
                ++left;
            }else{
                result[pos] = R;
                --right;
            }
            --pos;
        }
        return result;
    }
}
```