卡码笔记-Boreas

代码随想录

数组

> 《代码随想录》数组：二分查找

数组与二分查找详解

数组的基本特性

数组是相同类型的集合，存放在连续内存空间上。我们可以通过数组的下标索引访问数组中的元素，注意：

- 这个索引是从0开始的
- 内存空间的地址是连续的
- 在删除或者增添元素的时候，需要移动其他元素的地址
- 数组的元素不能直接删除，只能被覆盖
- 二维数组的内存空间是否连续：
  - 连续（C++等语言）
  - 不连续（Java等语言）

二分查找

适用条件

当题目中出现以下关键词时，就要考虑使用二分法：

- 数组有序
- 无重复元素

核心思想

> "大家写二分法经常写乱，主要是因为对区间的定义没有想清楚，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，就是在while寻找中每一次边界的处理都要坚持根据区间的定义来操作，这就是循环不变量规则。"

区间定义

区间的定义一般为两种：

1. 左闭右闭 [left, right]

2. 左闭右开 [left, right)

左闭右闭区间实现

```
int search(int* nums, int numsSize, int target) {
    int l = 0;
    int r = numsSize - 1;

    while (l <= r) {
        int m = l + ((r - l) / 2); // 防止溢出，等同于 (l + r)/2

        if (nums[m] > target) {
            r = m - 1; // 目标在左区间
        } else if (nums[m] < target) {
            l = m + 1; // 目标在右区间
        } else {
            return m;  // 找到目标
        }
    }
    return -1; // 未找到
}
```

边界处理逻辑分析

只要没进入 else 分支，就说明 nums[middle] != target 是确定无疑的！

☒ 场景一：nums[middle] > target

已知：

- 数组是升序的（二分查找的前提）
- nums[middle] > target

推理：

- nums[middle] 不可能等于 target（因为 >）
- 所有在 middle 右边的元素（nums[middle+1], nums[middle+2], ...）都 ⩾ nums[middle] → 都 > target

结论：

- 我们可以彻底丢弃 middle 及其右边的所有位置
- 新的搜索区间变为 [left, middle - 1]
- 所以：right = middle - 1

☒ 场景二：nums[middle] < target

已知：

- nums[middle] < target → nums[middle] ≠ target

推理：

- 所有在 middle 左边的元素都 ⩽ nums[middle] → 都 < target

- 所以 target 只可能出现在 middle 右边

结论：

- 丢弃 middle 及其左边
- 新区间：[middle + 1, right]
- 所以：left = middle + 1

《代码随想录》数组：移除元素

暴力搜索法

实现代码

```
int removeElement(int* nums, int numsSize, int val) {
    for (int i = 0; i < numsSize; i++) {
        if (nums[i] == val) {
            // 左移覆盖当前元素
            for (int j = i + 1; j < numsSize; j++) {
                nums[j - 1] = nums[j];
            }
            numsSize--; // 数组长度减1
            i--;      // 重新检查当前位置（因为后面元素前移了）
        }
    }
    return numsSize;
}
```

双指针法

实现代码

```
int removeElement(int* nums, int numsSize, int val) {
    int slowIndex = 0; // 慢指针：指向下一个"保留元素"应放置的位置

    for (int fastIndex = 0; fastIndex < numsSize; fastIndex++) {
        if (val != nums[fastIndex]) {
            // 当前元素不是要删除的值 → 保留它
            nums[slowIndex++] = nums[fastIndex];
            // 先赋值，再 slowIndex 自增（等价于：
            //   nums[slowIndex] = nums[fastIndex];
            //   slowIndex++;
            // ）
        }
        // 如果等于 val，fastIndex 继续前进，slowIndex 停留（跳过该元素）
    }
    return slowIndex; // slowIndex 正好是新数组的长度
}
```

双指针定义

- 快指针（fastIndex）：
  - 寻找新数组的元素
  - 新数组就是不含有目标元素 val 的数组
  - 遇到目标元素指针就跳过
- 慢指针（slowIndex）：
  - 指向更新新数组下标的位置
  - 指向被覆盖的那个索引

《代码随想录》数组：有序数组的平方

题目链接：https://leetcode.cn/problems/squares-of-a-sorted-array/description/

问题描述

给定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。

暴力解法

代码实现

c

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int cmp(const void* _a, const void* _b) {
    int a = *(int*)_a, b = *(int*)_b;
    return a - b;
}

int* sortedSquares(int* nums, int numsSize, int* returnSize) {
    *returnSize = numsSize;
    int* result = (int*)malloc(numsSize * sizeof(int));

    for (int i = 0; i < numsSize; i++) {
        result[i] = nums[i] * nums[i];
    }

    qsort(result, numsSize, sizeof(int), cmp);
    return result;
}
```

算法分析

1. 基本思路：
   - 先将每个数平方
   - 然后对平方后的数组进行排序
2. 时间复杂度：O(n log n)
   - 平方操作：O(n)
   - 排序操作：O(n log n)
3. 空间复杂度：O(n) - 需要额外的数组存储结果

双指针法

算法思想

> 数组其实是有序的，只不过负数平方之后可能成为最大数了。那么数组平方的最大值就在数组的两端，不是最左边就是最右边，不可能是中间。

解题思路

1. 使用双指针 i 指向起始位置，j 指向终止位置
2. 定义一个新数组 result，和原数组一样的大小
3. 让指针 k 指向 result 数组的终止位置
4. 比较 nums[i] * nums[i] 和 nums[j] * nums[j]：
   - 如果 nums[i] * nums[i] < nums[j] * nums[j]，那么 result[k--] = nums[j] * nums[j]
   - 如果 nums[i] * nums[i] >= nums[j] * nums[j]，那么 result[k--] = nums[i] * nums[i]

代码实现

c

```c
#include <stdio.h>
#include <stdlib.h>

/**
 * 将有序整数数组的每个元素平方后，返回一个按非递减顺序排序的新数组。
 *
 * @param nums: 输入的有序整数数组（升序，可含负数）
 * @param numsSize: 数组长度
 * @param returnSize: 输出参数，用于返回结果数组的实际长度
 * @return: 指向新分配的堆内存的指针，存储平方后的有序结果
 *
 * 注意：调用者必须使用 free() 释放返回的内存！
```

```
 */
int* sortedSquares(int* nums, int numsSize, int* returnSize) {
    // 设置返回数组的长度（与原数组相同）
    *returnSize = numsSize;

    // 动态分配结果数组内存（在堆上），大小为 numsSize 个 int
    int* result = (int*)malloc(numsSize * sizeof(int));

    // 双指针初始化：
    // i 指向数组开头（最小值，可能是绝对值最大的负数）
    // j 指向数组末尾（最大值，可能是绝对值最大的正数）
    // k 指向结果数组的最后一个位置（从后往前填充，先放最大平方值）
    int i = 0;
    int j = numsSize - 1;
    int k = numsSize - 1;

    // 当左指针不超过右指针时，继续比较
    while (i <= j) {
        // 计算左右两端元素的平方（避免重复计算）
        int leftSq = nums[i] * nums[i];
        int rightSq = nums[j] * nums[j];

        // 比较平方值大小：
        if (leftSq > rightSq) {
            // 左边平方更大 → 放入结果数组末尾
            result[k--] = leftSq;
            i++;  // 左指针右移，尝试下一个（绝对值更小的负数）
        } else {
            // 右边平方更大或相等 → 放入结果数组末尾
            result[k--] = rightSq;
            j--;  // 右指针左移，尝试下一个（更小的正数）
        }
    }

    // 返回指向堆内存的指针（调用者负责 free）
    return result;
}

/**
 * ACM 模式主函数：完整读取输入、调用函数、输出结果、释放内存
 */
int main() {
    int n;  // 数组长度
    scanf("%d", &n);

    // 动态分配输入数组内存
    int* nums = (int*)malloc(n * sizeof(int));

    // 读取 n 个整数到 nums 数组
    for (int i = 0; i < n; i++) {
```

```
        scanf("%d", &nums[i]);
    }

    // 声明变量接收返回数组的长度
    int returnSize;

    // 调用函数，获取结果数组指针
    int* result = sortedSquares(nums, n, &returnSize);

    // 输出结果：每个数字后跟一个空格（ACM/OJ 通常接受末尾空格）
    for (int i = 0; i < returnSize; i++) {
        printf("%d ", result[i]);
    }
    printf("\n"); // 输出换行符，符合多数 OJ 要求

    // 释放动态分配的内存，防止内存泄漏
    free(nums);   // 释放输入数组
    free(result); // 释放结果数组（由 sortedSquares 分配）

    return 0;
}
```