最长交替子数组 题解

> 2024寒假十大必做——编程打卡（1月23日）
>
> 题目为[力扣 2765. 最长交替子数组]

题目

给你一个下标从 **0** 开始的整数数组 `nums` 。如果 `nums` 中长度为 `m` 的子数组 `s` 满足以下条件，我们称它是一个 **交替子数组** ：

- `m` 大于 `1` 。
- `s1 = s0 + 1` 。
- 下标从 **0** 开始的子数组 `s` 与数组 `[s0, s1, s0, s1,...,s(m-1) % 2]` 一样。也就是说，`s1 - s0 = 1` ，`s2 - s1 = -1` ，`s3 - s2 = 1` ，`s4 - s3 = -1` ，以此类推，直到 `s[m - 1] - s[m - 2] = (-1)m` 。

请你返回 `nums` 中所有 **交替** 子数组中，最长的长度，如果不存在交替子数组，请你返回 `-1` 。

子数组是一个数组中一段连续 **非空** 的元素序列。

示例 1：

```
输入：nums = [2,3,4,3,4]
输出：4
解释：交替子数组有 [3,4] ，[3,4,3] 和 [3,4,3,4] 。最长的子数组为 [3,4,3,4] ，长度为4 。
```

示例 2：

```
输入：nums = [4,5,6]
输出：2
解释：[4,5] 和 [5,6] 是仅有的两个交替子数组。它们长度都为 2 。
```

提示：

- `2 <= nums.length <= 100`
- `1 <= nums[i] <= 104`

题解

$O(n^2)$ 解法

枚举每个位置作为交替子数组的起始位置，向后判断由此位置可产生的最长交替子数组的长度并更新答案。

```
class Solution {
public:
    int alternatingSubarray(vector<int>& nums) {
        const int n = nums.size();
        int ans = 0;
        for (int i = 0; i <= n - 1; i++) {          // 枚举起始位置
            int len = 1, flag = 1;
            for (int j = i + 1; j <= n - 1; j++) {  // 向后扫描求最大长度
                if (nums[j] == nums[j - 1] + flag) {
                    len++;
                    flag *= -1;
                }
                else {
                    break;
                }
            }
            ans = max(ans, len);                    // 更新答案
        }
        return ans == 1 ? -1 : ans;
    }
};
```

$O(n)$ 解法

可以采用动态规划的思想，定义 $dp[i]$ 为以位置 i 结尾的最长交替子数组的长度，有 $dp[0] = 1$。

1. 若 `dp[i - 1] == 1`，即 $nums[i-1]$ 与之前的序列不构成交替子数组。此时若 `nums[i] == nums[i - 1] + 1`，则 $nums[i]$ 与 $nums[i-1]$ 构成一个长度为 2 的交替子数组，否则 $nums[i]$ 与之前的序列不构成交替子数组。
2. 若 `dp[i - 1] > 1`，即 $nums[i-1]$ 与之前的序列构成一个长为 $dp[i-1]$ 的交替子数组。此时若 `nums[i] == nums[i - 2]`，则 $nums[i]$ 与之前的序列构成一个长为 $dp[i-1]+1$ 的交替子数组，否则说明以 $nums[i-1]$ 结尾的交替子数组无法继续延伸下去，需判断 $nums[i]$ 与 $nums[i-1]$ 能否构成一个长度为 2 的交替子数组，同情况 1。

```
class Solution {
public:
    int dp[101] = {};
    int alternatingSubarray(vector<int>& nums) {
        const int n = nums.size();
        int ans = 0;
        dp[0] = 1;
        for (int i = 1; i <= n - 1; i++) {
            if (dp[i - 1] == 1) {
                dp[i] = (nums[i] == nums[i - 1] + 1) + 1;
```

```
            }
            else {
                if (nums[i] == nums[i - 2]) {
                    dp[i] = dp[i - 1] + 1;
                }
                else {
                    dp[i] = (nums[i] == nums[i - 1] + 1) + 1;
                }
            }
            ans = max(ans, dp[i]);
        }
        return ans == 1 ? -1 : ans;
    }
};
```

空间复杂度：$O(n)$

实际上由于扫描过程中只用到了上一个相邻的状态，空间复杂度可以变为常数。

```
class Solution {
public:
    int alternatingSubarray(vector<int>& nums) {
        const int n = nums.size();
        int ans = 0;
        int dp = 1;
        for (int i = 1; i <= n - 1; i++) {
            if (dp == 1) {
                dp = (nums[i] == nums[i - 1] + 1) + 1;
            }
            else {
                if (nums[i] == nums[i - 2]) {
                    dp++;
                }
                else {
                    dp = (nums[i] == nums[i - 1] + 1) + 1;
                }
            }
            ans = max(ans, dp);
        }
        return ans == 1 ? -1 : ans;
    }
};
```

空间复杂度：$O(1)$

其他解法

再简述两种时间复杂度为 $O(n)$ 的解法：

1. 对 $O(n^2)$ 解法进行优化：在更新 i 时不是一步一步的移动，而是利用 j 在移动时已经探明的信息来“跳跃地”更新 i，这样可以避免重复扫描。

2. 双指针（$left, right$）：不断向后移动 $right$ 遍历数组，当和 $left$ 的位置差为奇数时，需满足 `nums[right] == nums[left] + 1`，位置差为偶数时需满足 `nums[right] == nums[left]`，若不满足上述条件，则向后移动 $left$ 直到满足该条件。