3105. Longest Strictly Increasing or Strictly Decreasing Subarray

You are given an array of integers `nums`. Return *the length of the* ***longest*** subarray *of* `nums` *which is either* strictly increasing *or* strictly decreasing.

Example 1:

Input: nums = [1,4,3,3,2]

Output: 2

Explanation:

The strictly increasing subarrays of `nums` are `[1]`, `[2]`, `[3]`, `[3]`, `[4]`, and `[1,4]`.

The strictly decreasing subarrays of `nums` are `[1]`, `[2]`, `[3]`, `[3]`, `[4]`, `[3,2]`, and `[4,3]`.

Hence, we return `2`.

Example 2:

Input: nums = [3,3,3,3]

Output: 1

Explanation:

The strictly increasing subarrays of `nums` are `[3]`, `[3]`, `[3]`, and `[3]`.

The strictly decreasing subarrays of `nums` are `[3]`, `[3]`, `[3]`, and `[3]`.

Hence, we return `1`.

Example 3:

Input: nums = [3,2,1]

Output: 3

Explanation:

The strictly increasing subarrays of `nums` are `[3]`, `[2]`, and `[1]`.

The strictly decreasing subarrays of `nums` are `[3]`, `[2]`, `[1]`, `[3,2]`, `[2,1]`, and `[3,2,1]`.

Hence, we return `3`.

Constraints:

- `1 <= nums.length <= 50`
- `1 <= nums[i] <= 50`

3105. Longest Strictly Increasing or Strictly Decreasing Subarray

```
/*
   Brute force
   Time complexity:  O(2n^2)
   Space complexity: O(1)
*/
typedef std::vector<int> vi;
class Solution {
   public:
      int length_of_LIS_or_LDS(vi& nums,int flag){
         int n=nums.size();

         int ans=1; // an element has a length equal to 1
         // For each element at index i
         for(int i=0;i<n;++i){
            // Determine the LI/LD subarray starting from i
            int len=1; // The element at position i has a length equal to 1
            int j=i+1; // Start  from next position

            // While the previous element is less/greater than the next one,
            // increment the size of the LI/LD subarray
            while(j<n && flag*nums[j-1]<flag*nums[j]){
               len++;
               j++;
            }

            // Maximize the answer
            ans=std::max(ans,len);
         }
         return ans;
      }

      int longestMonotonicSubarray(vi& nums){
         // 1st call with a flag equal to 1 to compute LI
         int lis=length_of_LIS_or_LDS(nums,1);

         // 2nd call with a flag equal to -1 to compute LD
         int lds=length_of_LIS_or_LDS(nums,-1);

         return std::max(lis,lds);
      }
};
```

3105. Longest Strictly Increasing or Strictly Decreasing Subarray

```
/*
   Single pass: Sliding window, Four pointers
   Time complexity: O(n)
   Space complexity: O(1)
*/
typedef std::vector<int> vi;
class Solution {
   public:
      int longestMonotonicSubarray(vi& nums){
         int n=nums.size();
         int ans=1,len_li=1,len_ld=1; // an element has a length equal to 1
         // For each element at index:
         int i=0; // start of LI
         int k=0; // start of LD
         while(i<n){
            // Determine the LI/LD subarray starting from i/k
            int j=i+1,l=k+1; // Start  from next position
            // While the previous element is less than the next one,
            // increment the size of the LI subarray
            while(j<n && nums[j-1]<nums[j]) j++;
            // While the previous element is greater than the next one,
            // increment the size of the LD subarray
            while(l<n && nums[l-1]>nums[l]) l++;
            // Maximize the answer
            len_li=std::max(ans,j-i); // Size of LI
            len_ld=std::max(ans,l-k); // Size of LD
            ans=std::max({ans,len_li,len_ld});
            // The new subarray will start at the end of the previous one.
            i=j;
            k=l;
         }
         return ans;
      }
};
```

3105. Longest Strictly Increasing or Strictly Decreasing Subarray

```
/*
   Single pass, One pointer
   Time complexity: O(n)
   Space complexity: O(1)
*/
typedef std::vector<int> vi;
class Solution {
   public:
      int longestMonotonicSubarray(vi& nums){
         int n=nums.size();
         int ans=1,len_li=1,len_ld=1;

         // Iterate through array comparing adjacent elements
         for(int i=0;i<n-1;++i){
            // If next element is larger,
            if(nums[i]<nums[i+1]){
               len_li++;    // Extend increasing sequence
               len_ld=1;  // Reset decreasing sequence
            }
            // If next element is smaller,
            else if(nums[i]>nums[i+1]){
               len_ld++; // Extend decreasing sequence
               len_li=1; // Reset increasing sequence
            }
            // If they are equal
            else{
               // Reset increasing and decreasing sequence
               len_li=len_ld=1;
            }

            ans=std::max({ans,len_li,len_ld});
         }
         return ans;
      }
};
```