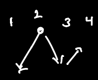
Input: array = [1, 2, 3, 3, 4, 0, 10, 6, 5, -1, -3, 2, 3]
Output: 6

Input: An array of integers
Output: The length of the longest peak in the array

A peak is defined as adjacent integers in the array that are strictly increasing until they reach a tip (the highest value in the peak) at which point they become strictly decreasing. At least three integers are required to form a peak

```javascript
// O(n) time | O(1) space
function longestPeak(array) {
  let longestPeak = 0;
  let i = 1;

  while (i < array.length - 1) {
    const peak = array[i] > array[i - 1] && array[i] > array[i + 1];
    if (!peak) {
      i++;
      continue;
    }

    let leftPointer = i - 2;
    while (leftPointer >= 0 &&
      array[leftPointer] < array[leftPointer + 1]
    ) {
      leftPointer--;
    }

    let rightPointer = i + 2;
    while (
      rightPointer < array.length &&
      array[rightPointer] < array[rightPointer - 1]
    ) {
      rightPointer++;
    }

    const currentPeakLength = rightPointer - leftPointer - 1;
    longestPeak = Math.max(currentPeakLength, longestPeak);
    i = rightPointer;
  }
  return longestPeak;
}
```

Time: O(n) (where n is the number of elements in the input array). This can be seen with the last variable statement: i = rightPointer. After we are done evaluating a peak, we assign the next value (after the last value of the peak) to continue our iteration. Therefore, we do not evaluate redundant elements



} we start new iteration @ 4

Space: O(1) since we are not using any more space as input size grows

Note: we go to array.length - 1 bc we are looking for a peak. The first and last elements cannot be a peak

Note: We do rightPointer - leftPointer - 1 because of the following:

[7, 1, 2, 3, 2, 1, 7]
↑ LP ↑ RP

Has a peak of length 5, using our algorithm, leftPointer is at index 0 and rightPointer is at index 6, we need to always subtract 1 to get the actual length.