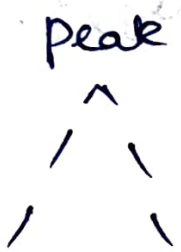
845. Longest Mountain in array

I/p: arr = [2, 1, 4, 7, 3, 2, 5]

O/p: 5

Explanation: the largest mountain is [1, 4, 7, 3, 2] which has length 5.



* First, numbers strictly increase
* then, numbers strictly decrease
* minimum length = 3 (because you need at least 1 up + 1 down + the peak)

we want longest mountain in array.

arr = [2, 1, 4, 7, 3, 2, 5]

peak = 7

Left side increasing : 4 → 7

Right side decreasing : 7 → 3 → 2

Length = 5.

Brute force :

* For each index i, check if it can be a peak (larger than neighbors)
* If yes:
  * count left side going up
  * count right side going down
* calculate mountain length = left + right + 1
* update maximum length.

```
class Solution {
    public int longestMountain (int [] arr) {
        int n = arr.length;
        int maxLen = 0;
      for (int i = 1 ; i < n ; i++) {
        if (arr[i] > arr [i-1] && arr [i] > arr [i+1]) {
            int left = i-1;
            int right = i+1;
          while (left > 0 && arr [left] > arr [left -1]) {
            left --;
          }
```

```
        while (right < n-1 && arr[right] > arr[right+1]) {
            right++;
        }
        int len = right - left + 1;
        maxLen = Math.max(maxLen, len);
    }
}
return maxLen;
}
}
```

Time → $O(n^2)$

Optimized approach:

Idea:

* start from left = 1
* If we see uphill (arr[i] > arr[i-1]) → keep climbing
* If we see peak (arr[i] > arr[i-1] & arr[i] > arr[i+1]) → start descending
* count the mountain length while descending
* update maximum
* move pointer to end of mountain → avoid double counting.

```
class Solution {
    public int longestMountain(int[] arr){
        int n = arr.length;
        int maxLen = 0;
        int i = 1;
```

```
while (i < n-1){
  if (arr[i] > arr[i-1] && arr[i] > arr[i+1]){
      int left = i-1;
      int right = i+1;
  while (left > 0 && arr[left] > arr[left-1]) left--;
  while (right < n-1 && arr[right] > arr[right+1]
                                    right++;
    maxLen = Math.max(maxLen, right - left+1);
    i = right;
  } else {
      i++;
    }
  }
  return maxLen;
 }
}
```

time → O(n) space → O(1).