# Question

Given an integer array nums sorted in **non-decreasing** order, return *an array of****the squares of each number****sorted in non-decreasing order*.

**Example 1:**

**Input:** nums = [-4,-1,0,3,10]

**Output:** [0,1,9,16,100]

**Explanation:** After squaring, the array becomes [16,1,0,9,100].

After sorting, it becomes [0,1,9,16,100].

**Example 2:**

**Input:** nums = [-7,-3,2,3,11]

**Output:** [4,9,9,49,121]

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums is sorted in **non-decreasing** order.

**Follow up:** Squaring each element and sorting the new array is very trivial, could you find an O(n) solution using a different approach?

# Solution

#### **Approach 1: Sort**

**Intuition and Algorithm**

Create an array of the squares of each element, and sort them.

|  |
| --- |
| class Solution {  public int[] sortedSquares(int[] A) {  int N = A.length;  int[] ans = new int[N];  for (int i = 0; i < N; ++i)  ans[i] = A[i] \* A[i];  Arrays.sort(ans);  return ans;  }  } |

**Complexity Analysis**

* Time Complexity: O(N \log N)*O*(*N*log*N*), where N*N* is the length of A.
* Space complexity : \mathcal{O}(N)O(*N*) or \mathcal{O}(\log{N})O(log*N*)
  + The space complexity of the sorting algorithm depends on the implementation of each program language.
  + For instance, the list.sort() function in Python is implemented with the [Timsort](https://en.wikipedia.org/wiki/Timsort) algorithm whose space complexity is \mathcal{O}(N)O(*N*).
  + In Java, the [Arrays.sort()](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html#sort-byte:A-) is implemented as a variant of quicksort algorithm whose space complexity is \mathcal{O}(\log{N})O(log*N*).

#### **Approach 2: Two Pointer**

**Intuition**

Since the array A is sorted, loosely speaking it has some negative elements with squares in decreasing order, then some non-negative elements with squares in increasing order.

For example, with [-3, -2, -1, 4, 5, 6], we have the negative part [-3, -2, -1] with squares [9, 4, 1], and the positive part [4, 5, 6] with squares [16, 25, 36]. Our strategy is to iterate over the negative part in reverse, and the positive part in the forward direction.

**Algorithm**

We can use two pointers to read the positive and negative parts of the array - one pointer j in the positive direction, and another i in the negative direction.

Now that we are reading two increasing arrays (the squares of the elements), we can merge these arrays together using a two-pointer technique.

|  |
| --- |
| class Solution {  public int[] sortedSquares(int[] nums) {  int n = nums.length;  int[] result = new int[n];  int left = 0;  int right = n - 1;  for (int i = n - 1; i >= 0; i--) {  int square;  if (Math.abs(nums[left]) < Math.abs(nums[right])) {  square = nums[right];  right--;  } else {  square = nums[left];  left++;  }  result[i] = square \* square;  }  return result;  }  } |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the length of A.
* Space Complexity: O(N)*O*(*N*) if you take output into account and O(1)*O*(1) otherwise.