Squares of Sorted array

https://www.youtube.com/watch?v=z0InhrjK3es

Solving "Squares of a Sorted Array" (LeetCode 977) actually touches on a surprising number of fundamental ideas. Here's a breakdown of the main concepts you're exercising—and learning—by implementing the two-pointer approach:

1. **Understanding Sorted Arrays and the Squaring Effect**
   - **Why sorted matters:** The input array is already in non-decreasing order (e.g. [-7, -3, 2, 3, 11]). That means the smallest values (most negative) are on the left, and the largest positives on the right.
   - **Squaring changes the order:** Once you square negative numbers, they become positive—and can end up larger than some originally positive entries. For example, $(-7)^2 = 49$, which is bigger than $3^2 = 9$. So if you just squared everything in place, the new list is no longer guaranteed to be sorted.
2. **Absolute Value (`abs`)**
   - **Core idea:** To figure out which end of the array will produce a larger square, you compare the absolute values of the numbers at each end.
   - **Why it matters:** Without taking `abs`, you might mistakenly think "$-7 < 11$ so square 11," but actually $|-7| = 7$ makes it possible for $(-7)^2 = 49$ to outgrow small positive values later.
3. **Two-Pointer Technique**
   - **What it is:** You keep one pointer (`l`) at the start (index 0) and another pointer (`r`) at the end (index n−1). In each step, you decide which side (left or right) has the bigger |number|, square that, put it into a result list, and move that pointer inward.
   - **Why it's O(n):** Each of the n elements is "touched" exactly once—either by `l` or `r`—so you never do more than n iterations. You never have to sort again.
4. **Building the Result in Reverse Order + List `reverse()`**
   - **Why reverse?** Because you always pick the bigger square first, you're effectively constructing a descending list of squares (largest → smallest). Only at the very end do you call `result.reverse()` so that the final output is ascending.
   - **List operations:** Learning that appending to a Python list is O(1) amortized, and reversing a list is O(n). It's still O(n) in total because appending n times plus one reverse pass remains linear.
5. **Loop Invariants and Boundaries (`while l <= r`)**
   - **Loop condition:** `while l <= r` ensures you process each index exactly once. If `l` ever crosses `r`, you've already dealt with every element. Understanding how to translate "process until pointers cross" into correct boundary conditions is a key skill in two-pointer problems.
   - **Avoiding off-by-one errors:** Making sure you increment `l` or decrement `r` properly and don't read out of range or skip an element.
6. **Time Complexity Analysis (O(n) vs. O(n log n))**

- **"Square and sort" approach:** If you naively square every element and run a generic sort afterward, you get O(n) for squaring plus O(n log n) for sorting—total O(n log n).
- **Two-pointer approach:** By merging the two ends in one pass, you eliminate the O(n log n) sort and achieve O(n). Learning how to identify when you can replace a sort with linear merging logic is a crucial optimization mindset.

7. **Space Complexity Considerations (O(n) extra space)**
   - **Why we use extra space:** We build a new list `result` of length n rather than trying to overwrite `nums` in place. That's fine here—extra O(n) memory is acceptable.
   - **In-place variants:** Recognizing that, if asked, one could implement an in-place version (e.g. write squared values into `nums` itself from the back forward), but it's more error-prone. Right now, you learn the trade-off: clarity vs. in-place efficiency.
8. **Comparison Logic and Conditionals**
   - **Writing correct comparisons:** The `if abs(nums[l]) > abs(nums[r])` line teaches you to be careful about operator precedence—make sure you call `abs()` around the individual numbers, not around the Boolean result.
   - **Correct branching:** Understanding when to take from the left end vs. right end, and how that affects the pointers.
9. **Understanding "Greedy" Selection in a Merge-Like Process**
   - **Greedy choice:** At each step, pick whichever end yields the larger square—there's no need to look ahead. That local choice always leads to a globally sorted array of squares once you reverse at the end.
   - **Relation to merging sorted lists:** This is mathematically very similar to the "merge" step of merge sort, where you take the bigger (or smaller) of two front elements to build a new list in order.
10. **Practical Python List Methods**
    - **`append(...)`:** Learning that `result.append(value)` puts one element at the tail.
    - **`.reverse()`:** Learning that `reverse()` mutates the list in place (and runs in linear time).
11. **Edge Cases and Testing**
    - **All-negative or all-positive arrays:** If `nums = [-5,-3,-1]`, your code will always take from the left first until you exhaust negatives. If `nums = [1, 2, 3]`, you always take from the right first. Understanding how the same logic gracefully handles either extreme.
    - **Single-element arrays:** Seeing that if `nums = [-4]`, then `l=0`, `r=0`, you compare, square, append, reverse (trivial), and return `[16]`.
    - **Zeros in the middle:** For something like `[-2, 0, 3]`, understanding that at some point both pointers can land on zero and you still handle it correctly.
12. **Putting It All Together in a Function/Method**
    - **Definition syntax:** Knowing how to wrap your logic inside a `def sortedSquares(self, nums):` method of a `class Solution` so that an online judge (like LeetCode) can call it directly.
    - **Return value:** Remembering to return the final list, not just print it.
13. **Translating a Verbal Algorithm to Code**
    - **From idea to implementation:** You learn to take the high-level description ("use two pointers at each end, compare absolute values, build from largest square down, then reverse") and break it into exact lines of Python.

In summary

By working through this problem, you gain or reinforce:

1. How to think about sorted data and why squaring can "disrupt" ordering.
2. The use of absolute values to compare magnitudes.
3. The two-pointer pattern for merging or selecting from both ends in one pass.
4. Careful loop boundaries and pointer updates to avoid skipping or double-using elements.
5. Building a result list in reverse order and then reversing it, instead of sorting post-hoc.
6. Analyzing why naive square-then-sort leads to O(n log n), while two-pointers achieves O(n).
7. Managing extra space (O(n)) vs. thinking about in-place variants.
8. Writing correct conditionals and avoiding common pitfalls (e.g. `abs((nums[l]) > abs(nums[r]))` vs. `abs(nums[l]) > abs(nums[r])`).
9. Handling edge cases cleanly without special branching.
10. Python list operations (`append`, `reverse`) and method definition/returning values.

All of these building blocks—absolute values, two-pointer merges, time/space complexity reasoning, careful loop logic—are widely useful whenever you work with sorted arrays or try to optimize beyond a straightforward sort.

Edge Cases and Testing

Let's look at those two "extreme" cases step by step. The goal is to see why the code always pulls from the left end when everything is negative, and always pulls from the right end when everything is positive.

Case 1: All negatives

Suppose

```
nums = [ -5,  -3,  -1 ]
```

Here's what happens inside the loop:

1. **Initial pointers:**
   - `l = 0` points at `nums[0] = -5`
   - `r = 2` points at `nums[2] = -1`
   - `result = []`
2. **First comparison:**
   - `abs(nums[l]) = abs(-5) = 5`
   - `abs(nums[r]) = abs(-1) = 1`
   - Since 5 > 1, we square `nums[l]` (which is `(-5)**2 = 25`) and append it to `result`.
   - Then we move the left pointer right by one: `l = 1`.

Now:

```
result = [25]
l = 1 (points at -3)
r = 2 (points at -1)
```

3. **Second comparison:**
   - `abs(nums[l]) = abs(-3) = 3`
   - `abs(nums[r]) = abs(-1) = 1`
   - 3 > 1, so we square `-3 → 9` and append it.
   - Move left pointer again: `l = 2`.

Now:

```
result = [25, 9]
l = 2 (points at -1)
r = 2 (points at -1)
```

4. **Third comparison (pointers meet):**
   - `abs(nums[l]) = abs(nums[r]) = abs(-1) = 1`
   - They're equal, so the code's `else` branch runs. We square `nums[r] → 1` and append it.
   - Then we move `r` left by one: `r = 1`.

Now:

```
result = [25, 9, 1]
l = 2, r = 1  (so l > r, loop stops)
```

5. **Reverse at the end:**
   - Right now, `result` is `[25, 9, 1]` (largest squares first).
   - After `result.reverse()`, it becomes `[1, 9, 25]`, which is the sorted list of squares.

Why always "take from the left" until it's done?

- In an all-negative array, the number furthest from zero is on the left (e.g. –5 is "bigger in absolute value" than –1).
- The code compares `abs(nums[l])` to `abs(nums[r])` and always finds the left side's absolute value is larger.
- So it repeatedly picks and removes squares from the left end until there are no more negatives—then finishes by squaring the last element.

Case 2: All positives

Now suppose

```
nums = [ 1,  2,  3 ]
```

Again we walk through:

1. **Initial pointers:**
   - `l = 0` (points at `1`)
   - `r = 2` (points at `3`)
   - `result = []`
2. **First comparison:**
   - `abs(nums[l]) = abs(1) = 1`
   - `abs(nums[r]) = abs(3) = 3`
   - Since 1 < 3, the `else` branch runs. We square `3 → 9` and append to `result`.
   - Move the right pointer left: `r = 1`.

   Now:

   ```
   result = [9]
   l = 0 (points at 1)
   r = 1 (points at 2)
   ```

3. **Second comparison:**
   - `abs(nums[l]) = 1`
   - `abs(nums[r]) = abs(2) = 2`
   - 1 < 2, so we square `2 → 4` and append.
   - Move the right pointer left: `r = 0`.

   Now:

   ```
   result = [9, 4]
   l = 0 (points at 1)
   r = 0 (points at 1)
   ```

4. **Third comparison (pointers meet):**
   - `abs(nums[l]) = abs(nums[r]) = abs(1) = 1`
   - They're equal, so `else` runs again. We square `1 → 1` and append.
   - Move `r` left: `r = -1`.

   Now:

   ```
   result = [9, 4, 1]
   l = 0, r = -1  (loop stops because l > r)
   ```

5. **Reverse at the end:**
   - Before reverse: `result = [9, 4, 1]`
   - After `result.reverse()`: `[1, 4, 9]`

Why always "take from the right" until it's done?

- In an all-positive array, the largest absolute value is at the right end (e.g. 3 is bigger than 1).
- Every iteration, the code sees `abs(nums[r]) > abs(nums[l])`, so it picks from the right, squares it, and moves the right pointer left, until only one element remains.

In short

- **All negatives (`[-5, -3, -1]`)**:
  - The biggest absolute value is always on the left side, so the code picks from the left (squares –5, then –3, then –1).
- **All positives (`[1, 2, 3]`)**:
  - The biggest absolute value is always on the right side, so it picks from the right (squares 3, then 2, then 1).

Either way, the "larger absolute value first" rule automatically handles these extremes without any special cases. When you reverse at the end, you end up with `[1, 9, 25]` in the negative case or `[1, 4, 9]` in the positive case—which is exactly the sorted squares.