Squaring a Sorted Array (easy)

We'll cover the following

Problem Statement

Given a sorted array, create a new array containing **squares of all the number of the input array** in the sorted order.

Example 1:

```
Input: [-2, -1, 0, 2, 3]
Output: [0, 1, 4, 4, 9]
```

Example 2:

```
Input: [-3, -1, 0, 1, 2]
Output: [0 1 1 4 9]
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
const make_squares = function(arr) {
  squares = []
  // TODO: Write your code here
  return squares;
};

```

TEST SAVE RESET

Solution

This is a straightforward question. The only trick is that we can have negative numbers in the input array, which will make it a bit difficult to generate the output array with squares in sorted order.

An easier approach could be to first find the index of the first non-negative number in the array. After that, we can use **Two Pointers** to iterate the array. One pointer will move forward to iterate the non-negative numbers and the other pointer will move backward to iterate the negative numbers. At any step, whichever number gives us a bigger square will be added to the output array. For the above-mentioned Example-1, we will do something like this:

Since the numbers at both the ends can give us the largest square, an alternate approach could be to use two pointers starting at both the ends of the input array. At any step, whichever pointer gives us the bigger square we add it to the result array and move to the next/previous number according to the pointer. For the above-mentioned Example-1, we will do something like this:

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function make_squares(arr) {
  const n = arr.length;
  squares = Array(n).fill(0);
  let highestSquareIdx = n - 1;
  let left = 0,
    right = n - 1;
  while (left <= right) {
    let leftSquare = arr[left] * arr[left],
      rightSquare = arr[right] * arr[right];
    if (leftSquare > rightSquare) {
      squares[highestSquareIdx] = leftSquare;
      left += 1;
    } else {
      squares[highestSquareIdx] = rightSquare;
      right -= 1;
    }
    highestSquareIdx -= 1;
  }
```

```
  return squares;
}


console.log(`Squares: ${make_squares([-2, -1, 0, 2, 3])}`);
console.log(`Squares: ${make_squares([-3, -1, 0, 1, 2])}`);
```

RUN SAVE RESET

Close

Output 3.077s

```
Squares: 0,1,4,4,9
Squares: 0,1,1,4,9
```

Time complexity

The time complexity of the above algorithm will be $O(N)$ as we are iterating the input array only once.

Space complexity

The space complexity of the above algorithm will also be $O(N)$; this space will be used for the output array.

MARK AS COMPLETED

← Back
Remove Duplicates (easy)

Next →
Triplet Sum to Zero (medium)

Report an Issue Ask a Question