Solution Review: Problem Challenge 1

We'll cover the following

Search Bitonic Array (medium)

Given a Bitonic array, find if a given 'key' is present in it. An array is considered bitonic if it is monotonically increasing and then monotonically decreasing. Monotonically increasing or decreasing means that for any index `i` in the array `arr[i] != arr[i+1]`.

Write a function to return the index of the 'key'. If the 'key' is not present, return -1.

Example 1:

```
Input: [1, 3, 8, 4, 3], key=4
Output: 3
```

Example 2:

```
Input: [3, 8, 3, 1], key=8
Output: 1
```

Example 3:

```
Input: [1, 3, 8, 12], key=12
Output: 3
```

Example 4:

```
Input: [10, 9, 8], key=10
Output: 0
```

Solution

The problem follows the **Binary Search** pattern. Since Binary Search helps us efficiently find a number in a sorted array we can use a modified version of the Binary Search to find the 'key' in the bitonic array.

Here is how we can search in a bitonic array:

1. First, we can find the index of the maximum value of the bitonic array, similar to Bitonic Array Maximum. Let's call the index of the maximum number `maxIndex`.
2. Now, we can break the array into two sub-arrays:
   - Array from index '0' to `maxIndex`, sorted in ascending order.
   - Array from index `maxIndex+1` to `array_length-1`, sorted in descending order.
3. We can then call **Binary Search** separately in these two arrays to search the 'key'. We can use the same Order-agnostic Binary Search for searching.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
function search_bitonic_array(arr, key) {
  const maxIndex = find_max(arr);
  const keyIndex = binary_search(arr, key, 0, maxIndex);
  if (keyIndex !== -1) {
    return keyIndex;
  }
  return binary_search(arr, key, maxIndex + 1, arr.length - 1);
}

// find index of the maximum value in a bitonic array
function find_max(arr) {
  let start = 0,
    end = arr.length - 1;
  while (start < end) {
    const mid = Math.floor(start + (end - start) / 2);
    if (arr[mid] > arr[mid + 1]) {
      end = mid;
    } else {
      start = mid + 1;
    }
  }
  // at the end of the while loop, 'start === end'
```

```
23   return start;
24 }
25
26
27 // order-agnostic binary search
28 function binary_search(arr, key, start, end) {
```

RUN SAVE RESET

Close

Output 2.343s

```
3
1
3
0
```

Time complexity

Since we are reducing the search range by half at every step, this means that the time complexity of our algorithm will be $O(logN)$ where 'N' is the total elements in the given array.

Space complexity

The algorithm runs in constant space $O(1)$.