Longest Increasing Subsequence Write-up

#Algorithm

1. Check if the input array `numbers` is null or empty. If so, return an empty array.

2. Initialize variables:

- `n` as the length of the `numbers` array.
- `lengths` as an integer array of size `n` to store the lengths of the increasing subsequences.
- `previousIndices` as an integer array of size `n` to store the previous indices of the numbers in the increasing subsequences.
- `maxLength` as 1 to store the maximum length of the increasing subsequence.
- `endIndex` as 0 to store the index of the last element of the longest increasing subsequence.

3. Iterate over the `numbers` array from index 0 to index `n - 1`.

- Initialize `lengths[i]` as 1.
- Initialize `previousIndices[i]` as -1.

4. Within the above iteration, iterate over the `numbers` array from index 0 to index `i - 1`.

- If `numbers[j]` is less than `numbers[i]` and `lengths[j] + 1` is greater than `lengths[i]`, update:
- `lengths[i]` to `lengths[j] + 1`.
- `previousIndices[i]` to `j`.

5. Within the above iteration, if `lengths[i]` is greater than `maxLength`, update:

- `maxLength` to `lengths[i]`.
- `endIndex` to `i`.

6. Create an integer array `longestIncreasingSubsequence` of size `maxLength`.

7. Initialize `index` as `maxLength - 1`.

8. Iterate while `endIndex` is greater than or equal to 0:

- Set `longestIncreasingSubsequence[index]` as `numbers[endIndex]`.
- Update `endIndex` to `previousIndices[endIndex]`.
- Decrement `index` by 1.

9. Return `longestIncreasingSubsequence`.

The code provided earlier in the previous response implements this algorithm to find the longest increasing subsequence.