3011. Find if Array Can Be Sorted

Created	@November 6, 2024 12:26 AM
Class	LeetCode

Problem Overview:

You are given an array `nums` of integers. The goal is to determine whether the array can be sorted if you are allowed to **group numbers based on the number of 1s in their binary representation** and only sort those within their groups. In other words:

- Numbers with the same number of 1s in their binary form must stay together in the same group.
- The order within the groups can be rearranged, but the groups themselves must remain sorted based on the value of the minimum number in each group.

Key Concepts:

1. **Binary Representation and Bit Count**:

   Each number can be represented in binary (base 2). For example:

   - `3` in binary is `11` (which has 2 ones).
   - `7` in binary is `111` (which has 3 ones).
     The "bit count" of a number refers to the number of 1s in its binary representation.

2. **Grouping Based on Bit Count**:

   The solution groups numbers by the number of 1s in their binary representation. The key observation here is that we can only rearrange numbers with the same bit count among themselves, and we must ensure the groups are in increasing order of their smallest number.

Code Walkthrough:

```
function canSortArray(nums: number[]): boolean {
    let previousMax = 0;
    let currentMin = Infinity;
    let currentMax = -Infinity;
    let previousBitCount = 0;
```

1. **Initialization**:
    - `previousMax`: This keeps track of the maximum number in the previous bit count group. It is initialized to `0`.
    - `currentMin` and `currentMax`: These track the minimum and maximum values in the current group of numbers that share the same number of 1s in their binary representation. They are initialized to `Infinity` and `Infinity` respectively.
    - `previousBitCount`: This tracks the bit count of the previous group. It is initialized to `0`.

```
    for (let num of nums) {
        const currentBitCount = num.toString(2).split('1').le
ngth - 1;
```

1. **Looping through each number**:
    - The loop iterates through each number in `nums`.
    - `currentBitCount` calculates how many 1s are in the binary representation of the current number (`num`):
        - `num.toString(2)` converts the number to its binary string representation.
        - `.split('1').length - 1` counts how many times `1` appears in the binary string. This is equivalent to `num.bit_count()` in Python (since JavaScript does not have a built-in method to count 1s in binary numbers).

```
        if (previousBitCount === currentBitCount) {
            currentMin = Math.min(currentMin, num);
            currentMax = Math.max(currentMax, num);
```

1. **When the bit counts are the same**:
    - If the bit count of the current number (`currentBitCount`) is equal to the bit count of the previous number (`previousBitCount`), the current number belongs to the same group as the previous one.
    - **Updating the current group**:
        - `currentMin`: Update to the smallest number in the current group.
        - `currentMax`: Update to the largest number in the current group.

```
        } else {
            if (currentMin < previousMax) {
                return false;
            }
            previousMax = currentMax;
            currentMin = currentMax = num;
            previousBitCount = currentBitCount;
        }
    }
```

1. **When the bit counts are different (indicating a new group)**:
    - If the bit count of the current number is different from the previous number's bit count, we are entering a new group.
    - **Check if the previous group was sorted**:
        - We need to ensure that the previous group was sorted by checking if the `currentMin` of the previous group is greater than or equal to the `previousMax` (the maximum value from the previous group).
        - If `currentMin < previousMax`, the numbers cannot be sorted in the desired way, so the function returns `false`.

- **Update group boundaries**:
  - `previousMax`: The maximum value of the previous group becomes the `currentMax` (largest number in the current group).
  - `currentMin` and `currentMax`: Both are set to the current number (`num`), as the new group starts with this number.
  - `previousBitCount`: Update to the `currentBitCount` for the new group.

```
    return currentMin >= previousMax;
}
```

1. **Final Check**:
   - After the loop finishes, we return `true` if the last group (the one that was processed last) is valid.
   - The last check ensures that the minimum of the last group is greater than or equal to the maximum of the previous group.

Key Insights:

- **Groups are formed by numbers having the same bit count**: This is the most important rule. The function groups numbers based on how many 1s their binary representations have.
- **Ordering between groups**: Groups must be sorted in increasing order of the smallest number in each group. This ensures that the groups can be rearranged into a sorted array while respecting the original numbers within each group.
- **Bit count comparison**: Each time we encounter a number with a different bit count than the previous one, we check whether the groups are already ordered in a valid way. If they are not, the function returns `false`.

Example:

Consider the input `nums = [3, 7, 1, 15, 5]`:

1. **Step 1**: Convert to binary:

- `3` → `11` (2 ones)
- `7` → `111` (3 ones)
- `1` → `1` (1 one)
- `15` → `1111` (4 ones)
- `5` → `101` (2 ones)

2. **Step 2**: Group based on bit count:
   - Group 1: Numbers with 1 one → `[1]`
   - Group 2: Numbers with 2 ones → `[3, 5]`
   - Group 3: Numbers with 3 ones → `[7]`
   - Group 4: Numbers with 4 ones → `[15]`
3. **Step 3**: Check if the groups can be sorted:
   - The smallest number in Group 1 is `1`, which is valid.
   - The smallest number in Group 2 is `3`, which is greater than `1` (previous group's max).
   - The smallest number in Group 3 is `7`, which is greater than `5` (previous group's max).
   - The smallest number in Group 4 is `15`, which is greater than `7` (previous group's max).

   Thus, the function will return `true` because the groups can be arranged in a valid way.

Time Complexity:

- The time complexity of the function is `O(n)`, where `n` is the length of the array `nums`. This is because we are iterating through each number once, and the bit count calculation (i.e., converting to binary and counting 1s) is constant time for each number (typically logarithmic in the number of bits, but still manageable within typical integer sizes).

Space Complexity:

- The space complexity is `O(1)`, since we are using only a few extra variables (no additional data structures that grow with the input size).