3011. Find if Array Can Be Sorted

Medium Bit Manipulation Array Sorting

Problem Description

In this problem, we are given an array of positive integers called `nums`, and our task is to determine if we can sort it using a specific operation. The operation allows us to swap any two adjacent elements, but only if those elements have the same number of 1s in their binary representations; this is known as the number of set bits. We can perform this operation as many times as needed, including not at all, if the array is already sorted.

In essence, we need to check if, by using the allowed swaps, we can arrange the array in non-decreasing order. If we find it's possible to achieve this, we should return `true`. Otherwise, we return `false` if it's impossible to sort the array under these constraints.

Intuition

The solution hinges on understanding that we cannot change the relative order of elements with different numbers of set bits since they can't be swapped directly based on the rule. Therefore, sorting the array essentially breaks it down into subarrays, each composed of elements with the same number of set bits, which could be ordered in any way since swaps among them are allowed.

With this in mind, we can iterate through the array and identify these subarrays. Within each subarray, we keep track of the minimum and maximum value encountered so far. A crucial insight is that if the minimum value in a subarray is less than the maximum value of the previous subarray, it indicates that sorting is impossible. This is because, in a sorted array, all elements in a previous (and thus lower) subarray should be less than or equal to all the elements in the following subarray.

To implement this, we use two pointers as we traverse the array. The first pointer, `i`, marks the start of a subarray, and the second pointer, `j`, explores the rest of the array to find the end of this subarray. The `while` loops ensure that we progress through the array, from one subarray to the next, assessing the minimum and maximum for each set bit count. If we successfully traverse the entire array without encountering a condition that makes sorting impossible, we return `true`. Otherwise, we return `false` as soon as such a condition is found.

Solution Approach

The implementation follows a straightforward two-pointer approach to tackle the problem:

1. Initialize a variable `pre_mx` to negative infinity. This will keep track of the maximum value of the previous subarray as we progress through `nums`.
2. Set up two pointers, `i` and `j`, starting from 0 (`i`) and 1 (`j`), respectively. Pointer `i` represents the beginning of the current subarray, while `j` will be used to find the end of this subarray.
3. Loop over `nums` using `i` to iterate through the array. For each position `i`, we: a. Find the number of set bits of `nums[i]`, which determines the current subarray we're evaluating. This can be done using the `.bit_count()` method in Python. b. Initialize `mi` and `mx` to the value of `nums[i]`. These will track the minimum and maximum values within the current subarray, respectively.
4. While inside the loop, initiate a nested loop that keeps running as long as `j` is within the bounds of the array and `nums[j]` shares the same number of set bits as `nums[i]`. Within this loop: a. Update `mi` and `mx` to be the minimum and maximum of the current subarray by comparing them to `nums[j]`. b. Increment `j` to check the next element.
5. After the nested loop terminates (meaning we reached the end of the current subarray with matching set bits), check if `pre_mx` is greater than `mi`. If it is, it means that the previous subarray had a value greater than the smallest value of the current subarray, which implies it is impossible to sort the array. Hence, return `False`.
6. Update `pre_mx` with the largest value in the current subarray, `mx`, because this now becomes the maximum of the last processed subarray for the next iteration.
7. Set pointer `i` to `j` to start the algorithm for the next subarray of elements with equal set bit count.
8. If the algorithm completes the iteration over the array without returning `False`, it means sorting the array is possible under the given rules, so it returns `True`.

This approach efficiently applies the rules of the problem to determine if sorting is feasible, relying on the fact that only elements with the same number of set bits can be adjacent in the final sorted array. It uses python's built-in `.bit_count()` method to count set bits and utilizes comparison operations to ensure order within and between identified subarrays.

Example Walkthrough

Let's illustrate the solution approach using a small example with the array `nums = [3, 1, 2, 4]`. The binary representations are `11`, `01`, `10`, and `100`, with set bit counts of `2`, `1`, `1`, and `1`, respectively.

1. Initialize `pre_mx` to negative infinity. Start with `i = 0` and `j = 1`.
2. At `i = 0`, `nums[i] = 3`. Number of set bits in `3` (`11` in binary) is `2`. Initialize min (`mi`) and max (`mx`) of the current subarray to the value of `nums[i]` which is `3`.
3. Move to `j` which is `1`. Number of set bits in `nums[j]` (`1` in binary) is `1`, which is different from the `2` set bits we have for `nums[i]`. So we close this subarray here because we cannot form a subarray with different set bit counts. Since we cannot include `nums[j]` in the current subarray, we skip updating `mi` and `mx` and proceed to step 5.
4. We check if `pre_mx > mi` which means if `-inf > 3`. This is not true, so we move on and update `pre_mx` to `mx` which is `3`.
5. We set `i` to `j`, therefore `i` is now `1` and increment `j` to `2` to start evaluating the next subarray.
6. At `i = 1`, `nums[i]` is `1`. The number of set bits is `1`. Initialize `mi` and `mx` to `1`.
7. `j = 2`, `nums[j] = 2` which also has `1` set bit. We include `nums[j]` in the current subarray and update `mi` to `min(mi, nums[j])` (which remains `1`) and `mx` to `max(mx, nums[j])` (which becomes `2`). Increment `j` to `3`.
8. `j = 3`, `nums[j] = 4` which also has `1` set bit. We include `nums[j]` in the current subarray and update `mi` to `min(mi, nums[j])` (which is still `1`) and `mx` to `max(mx, nums[j])` (which becomes `4`). After this, `j` is out of bounds (since `nums` has only four elements), so we exit this nested loop.
9. We check if `pre_mx > mi` which means if `3 > 1`. This is not true, so we update `pre_mx` to `mx` which is now `4`.
10. Since we've reached the end of the array, we've verified that each subarray with the same set bit count is such that its minimum value is not less than the maximum value of the previous subarray.

The result here is that the array can be sorted using the allowed operations: swap `1` and `2` to get `[3, 2, 1, 4]`, which is now in the non-decreasing order. Our algorithm returns `True`.

Solution Implementation

Python

```python
from math import inf

class Solution:
    def canSortArray(self, nums: List[int]) -> bool:
        # Initialize previous maximum to negative infinity.
        previous_max = -inf
        # Initialize index and get the length of the list.
        i, n = 0, len(nums)

        # Iterate through the list of numbers.
        while i < n:
            # Initialize the next index and get the bit count of the current number.
            j = i + 1
            bit_count = bin(nums[i]).count('1')
            # Keep track of the minimum and maximum for the current bit count.
            current_min = current_max = nums[i]

            # Slide through the array to find consecutive numbers with the same bit count.
            while j < n and bin(nums[j]).count('1') == bit_count:
                current_min = min(current_min, nums[j])
                current_max = max(current_max, nums[j])
                j += 1

            # If the previous maximum number is greater than the current minimum,
            # the array cannot be sorted based on the conditions, hence return false.
            if previous_max > current_min:
                return False
            # Update previous maximum for the next iteration.
            previous_max = current_max
            # Move to the next segment with a different bit count.
            i = j

        # If we've reached this point, the array can be sorted, therefore return True.
        return True
```

Java

```java
class Solution {
    public boolean canSortArray(int[] nums) {
        // Initialize the previous maximum value to the lowest possible value.
        int prevMax = Integer.MIN_VALUE;

        // Index 'i' will track the start of each segment with equal bit count.
        int i = 0;
        // 'n' holds the length of the input array.
        int n = nums.length;

        // Loop through the elements of the array.
        while (i < n) {
            // 'j' will track the end of the current segment.
            int j = i + 1;
            // 'bitCount' stores the number of 1-bits in current array element.
            int bitCount = Integer.bitCount(nums[i]);
            // 'min' and 'max' track the minimum and maximum of the current segment.
            int min = nums[i], max = nums[i];

            // Continue to next elements if they have the same bit count.
            while (j < n && Integer.bitCount(nums[j]) == bitCount) {
                min = Math.min(min, nums[j]);
                max = Math.max(max, nums[j]);
                j++;
            }

            // If the max value of the previous segment is greater than the min of the current, it can't be sorted.
            if (prevMax > min) {
                return false;
            }
            // Update the prevMax to the max value of the current segment.
            prevMax = max;
            // Move 'i' to the start of the next segment.
            i = j;
        }
        // If all segments are in the correct order, return true.
        return true;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    bool canSortArray(vector<int>& nums) {
        int previousMax = -200;  // Initial value considering the constraints for possible integer values
        int currentIndex = 0, numSize = nums.size();

        // Loop through all elements in the array
        while (currentIndex < numSize) {
            int nextIndex = currentIndex + 1;
            int currentPopCount = __builtin_popcount(nums[currentIndex]);
            int currentMin = nums[currentIndex], currentMax = nums[currentIndex];

            // Find subsequence where all elements have the same number of set bits (1s)
            while (nextIndex < numSize &&  __builtin_popcount(nums[nextIndex]) == currentPopCount) {
                currentMin = min(currentMin, nums[nextIndex]);
                currentMax = max(currentMax, nums[nextIndex]);
                nextIndex++;
            }

            // If the maximum value from previous segment is greater than minimum of the current,
            // then the array can't be sorted based on the rules given
            if (previousMax > currentMin) {
                return false;
            }
            previousMax = currentMax; // Update previousMax to the max of the current segment
            currentIndex = nextIndex; // Move to the next segment
        }

        // If the loop completes without returning false, it means the array can be sorted
        return true;
    }
};
```

TypeScript

```typescript
function canSortArray(nums: number[]): boolean {
    let previousMax = -200; // Initiate the previous maximum to a value lower than any element in nums
    const length = nums.length;

    // Iterate over the array
    for (let i = 0; i < length; ) {
        let j = i + 1; // Start from the next element
        const bitCountOfCurrent = bitCount(nums[i]); // Get the bit count of the current element

        // Find min and max element within the same bit count group
        let minElement = nums[i];
        let maxElement = nums[i];

        // Keep updating min/max in the group where the bit count is the same
        while (j < length && bitCount(nums[j]) === bitCountOfCurrent) {
            minElement = Math.min(minElement, nums[j]);
            maxElement = Math.max(maxElement, nums[j]);
            j++;
        }

        // If the max of the previous group is greater than the min of the current group, return false
        if (previousMax > minElement) {
            return false;
        }

        // Update the previousMax to the max of the current group
        previousMax = maxElement;
        i = j; // Move to the next group
    }

    // If the entire array can be separate into groups with incremental mins, return true
    return true;
}

function bitCount(i: number): number {
    // Apply bit manipulation tricks to count the bits set to 1
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f; // Return the count of bits set to 1
}
```

```python
from math import inf

class Solution:
    def canSortArray(self, nums: List[int]) -> bool:
        # Initialize previous maximum to negative infinity.
        previous_max = -inf
        # Initialize index and get the length of the list.
        i, n = 0, len(nums)

        # Iterate through the list of numbers.
        while i < n:
            # Initialize the next index and get the bit count of the current number.
            j = i + 1
            bit_count = bin(nums[i]).count('1')
            # Keep track of the minimum and maximum for the current bit count.
            current_min = current_max = nums[i]

            # Slide through the array to find consecutive numbers with the same bit count.
            while j < n and bin(nums[j]).count('1') == bit_count:
                current_min = min(current_min, nums[j])
                current_max = max(current_max, nums[j])
                j += 1

            # If the previous maximum number is greater than the current minimum,
            # the array cannot be sorted based on the conditions, hence return false.
            if previous_max > current_min:
                return False
            # Update previous maximum for the next iteration.
            previous_max = current_max
            # Move to the next segment with a different bit count.
            i = j

        # If we've reached this point, the array can be sorted, therefore return True.
        return True
```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)` because there is a single while loop that iterates through each element of the array `nums` exactly once. Inside the loop, the `.bit_count()` method is called, which is expected to run in constant time, and basic arithmetic operations and comparisons are performed, which also take constant time per iteration. Even though there is a nested loop, it does not increase the overall number of iterations; it just continues from where the outer loop left off.

Space Complexity

The space complexity of the code is `O(1)` since it uses a fixed amount of extra space: variables `pre_mx`, `i`, `n`, `j`, `cnt`, `mi`, and `mx`. There is no use of any data structures that grow with the size of the input array `nums`, which keeps the space complexity constant.