1356. Sort Integers by The Number of 1 Bits

Easy · Bit Manipulation · Array · Counting · Sorting

Problem Description

The problem presents a task where we are given an array of integers, `arr`, and we need to sort this array with a specific set of rules based on the binary representation of its elements. The primary sorting criterion is the number of `1`s in the binary representation of each integer. If two integers have the same number of `1`s, then they should be sorted in ascending order according to their integer values.

The goal is to return the array sorted first by the number of `1`s in their binary representation, and then by their value when there's a tie on the first criteria.

Intuition

To tackle the sorting problem, we need to decide upon a sorting strategy that complies with the rules provided:

1. Count the number of `1`s in the binary representation of each integer.
2. Sort the integers by the number of `1`s. In the event of a tie (when two numbers have the same number of `1`s), sort by the integer values themselves, in ascending order.

The built-in Python `sorted` function offers us a straightforward way to sort the elements of an array. We can customize the sorting order by providing a `key` argument that transforms each element before comparison during the sort. This transformation doesn't change the actual elements of the array, but it is used to guide the sort order.

Thus, we choose a `lambda` function as the `key`, that returns a tuple for each `x` in `arr`: `(x.bit_count(), x)`. The `bit_count` method returns the number of `1`s in the binary representation of `x`, which addresses our primary criterion. By creating a tuple with `x.bit_count()` as the first element and `x` as the second, we ensure that when two numbers have the same number of `1`s, the smaller number comes first, satisfying the secondary sorting criterion.

The sorted list according to these criteria will thus be the result we return.

Solution Approach

The provided Python solution makes use of Python's higher-level functionality to implement the sorting logic cleanly and efficiently. To understand the solution's implementation, let's break down the key components and the patterns leveraged in the code:

Lambda Functions

A lambda function is an anonymous function defined with the `lambda` keyword in Python. In the solution, the lambda function is used as a `key` argument to the `sorted` function. It defines the sorting behavior according to the specific problem constraints.

Tuple Sorting

The `lambda` function leverages a feature of Python's sorting algorithm, which can sort tuples lexicographically. That means the first elements of the tuples are compared first, and if those are equal, the second elements are compared, and so on.

`bit_count` Method

The `bit_count()` method returns the number of `1` bits in the binary representation of an integer (an important note here is that this method is only available in Python 3.10 or later). If you're using an earlier version of Python, you would need to use the `bin(x).count('1')` approach instead.

Sorted Function

Finally, the `sorted` function is a built-in Python function that returns a new list containing all items from the iterable in ascending order. A key feature of `sorted` is that it allows you to define a `key` function that is called on each element before making comparisons.

Now, let's piece everything together. The solution approach is carried out as follows:

1. **Define a Key Function:** The `lambda` function `(lambda x: (x.bit_count(), x))` generates a tuple with two items for each element `x` in the array `arr`. The first item is the count of `1`s in the binary representation of `x`, and the second item is `x` itself.
2. **Apply Sorting with Custom Key:** The `sorted` function then uses the tuples generated by the `lambda` function to sort the entire array. It prioritizes the count of `1` bits first, as it's the first element of the tuple. If two tuples have the same first element (meaning the elements have the same number of `1`s in their binary representations), the second element of the tuple (the element's value) is used as a tie-breaker.
3. **Return the Sorted Array:** The `sorted` function does not modify `arr` in place; instead, it returns a new list, which is the correctly sorted version of `arr` as per the problem's constraints.

By combining these Python features, the solution elegantly and efficiently sorts the array `arr` according to the problem's specifications.

Example Walkthrough

Let's consider an array of integers for demonstration: `arr = [3, 1, 2, 4]`.

First, we'll determine the binary representation of each number and count the number of `1`s:

- The binary representation of `3` is `11`, which has `2` ones.
- The binary representation of `1` is `1`, which has `1` one.
- The binary representation of `2` is `10`, which has `1` one.
- The binary representation of `4` is `100`, which has `1` one.

Following the primary sorting criterion (number of `1`s), we'd have an intermediate sort order of `[1, 2, 4]` (each with one `1`), and `[3]` (with two `1`s). But since `1`, `2`, and `4` all have the same number of `1`s in their binary representation, we must sort them by their value.

The custom `lambda` function used as the `key` in the `sorted` algorithm will generate the following tuples based on the binary count and the integer value:

- For `3`: `(2, 3)`
- For `1`: `(1, 1)`
- For `2`: `(1, 2)`
- For `4`: `(1, 4)`

When we pass these tuples to the `sorted` function, it will sort the numbers first by the number of `1`s in their binary representation, and then by their integer value in case of a tie.

Here is what the sorting stage looks like with these tuples:

1. `(1, 1)` comes before `(1, 2)` and `(1, 4)` because their first elements are equal and `1` is the smallest integer value among them.
2. `(1, 2)` comes before `(1, 4)` because they have the same number of `1`s and `2` is smaller than `4`.
3. `(2, 3)` comes after all `(1, x)` tuples because `2` is greater than `1`.

As a result, considering the second element of each tuple, we get the sorted array: `[1, 2, 4, 3]`.

The return value of the `sorted` function with the custom `lambda` function as the provided key will give us this final sorted array which satisfies both the primary and secondary sorting criteria specified in the problem.

Solution Implementation

Python

```python
class Solution:
    def sort_by_bits(self, arr: List[int]) -> List[int]:
        # Sort the array based on the number of 1's in the binary representation
        # of each number ('x.bit_count()'). In the event of a tie, the numbers
        # are sorted based on their value ('x').
        return sorted(arr, key=lambda x: (bin(x).count('1'), x))
        # Note: The use of 'x.bit_count()' is available in Python 3.10 and later.
        # For versions before Python 3.10, we can use 'bin(x).count('1')' instead.

# Example usage:
# solution = Solution()
# result = solution.sort_by_bits([0,1,2,3,4,5,6,7,8])
```

Java

```java
import java.util.Arrays; // Importing Arrays class for sort function

class Solution {
    public int[] sortByBits(int[] arr) {
        int n = arr.length; // Store the length of the array

        // Add to each element in the array a value that represents
        // the bit count of the number multiplied by 100000 to ensure
        // it is prioritized in the sorting
        for (int i = 0; i < n; ++i) {
            int bitCount = Integer.bitCount(arr[i]); // Count number of 1-bits in arr[i]
            arr[i] += bitCount * 100000; // Add 100000 for each 1-bit to prioritize in sorting
        }

        Arrays.sort(arr); // Sort the array with modified values

        // After sorting retrieve the original values by taking modulo 100000
        for (int i = 0; i < n; ++i) {
            arr[i] %= 100000; // Reduce each element back to original value
        }

        return arr; // Return the sorted array by bits
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to sort the numbers based on the number of 1-bits they have.
    // In case of a tie, sort by the values themselves.
    vector<int> sortByBits(vector<int>& arr) {
        // Apply a transformation to each number in the array.
        // The transformation adds the number of 1-bits in the number times 100000
        // to the number itself. This is done to couple the number of 1-bits with the number.
        for (int& num : arr) {
            num += __builtin_popcount(num) * 100000;
        }

        // Sort the transformed array.
        // The numbers are now ordered first by the number of 1-bits, then by the number's value.
        sort(arr.begin(), arr.end());

        // Iterate through the array to revert the transformation and obtain
        // the original numbers, preserving the new order.
        for (int& num : arr) {
            num %= 100000; // Remove the added portion to get back the original number.
        }

        // Return the sorted array.
        return arr;
    }
};
```

TypeScript

```typescript
// Function to sort an array of numbers based on the number of 1-bits each number has.
// In the case of a tie, numbers are sorted by their value.
function sortByBits(arr: number[]): number[] {
    // Helper function to count the number of 1-bits in a binary representation of a number.
    const countBits = (num: number): number => {
        let count = 0;
        while (num) {
            // Remove the rightmost 1-bit from the number
            num &= num - 1;
            // Increment the count of 1-bits
            count++;
        }
        return count;
    };

    // Sorting the array based on the number of 1-bits each number has (asc order).
    // In the case of a tie, sort by numerical value (asc order).
    return arr.sort((a, b) => {
        // First, compare by the number of 1-bits
        const bitCountComparison = countBits(a) - countBits(b);
        if (bitCountComparison !== 0) {
            return bitCountComparison;
        }
        // If the number of 1-bits is the same, compare by the numbers themselves
        return a - b;
    });
}
```

```python
class Solution:
    def sort_by_bits(self, arr: List[int]) -> List[int]:
        # Sort the array based on the number of 1's in the binary representation
        # of each number ('x.bit_count()'). In the event of a tie, the numbers
        # are sorted based on their value ('x').
        return sorted(arr, key=lambda x: (bin(x).count('1'), x))
        # Note: The use of 'x.bit_count()' is available in Python 3.10 and later.
        # For versions before Python 3.10, we can use 'bin(x).count('1')' instead.

# Example usage:
# solution = Solution()
# result = solution.sort_by_bits([0,1,2,3,4,5,6,7,8])
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code primarily depends on the complexity of the sorting algorithm used by Python's `sorted` function. Python uses the TimSort algorithm, which has a time complexity of `O(n log n)` for the average and worst case, where `n` is the number of elements in the array to be sorted.

In this case, for each comparison, the sorting algorithm also calculates the bit count (number of 1s in the binary representation of the number), which is `O(1)` as Python's integer bit count implementation is efficient and not based on the value of the number but the number of set bits. However, this bit count operation will be performed multiple times per element during the sorting process.

Thus, assuming `k` is the number of comparisons performed by the sorting algorithm, the total time complexity considering the bit count operations for comparison purposes becomes `O(k)`. Since `k` can be as large as `n log n` comparisons, the total time complexity remains `O(n log n)`.

Space Complexity

The space complexity of this function is `O(n)`, as the `sorted` function returns a new list containing the sorted elements and does not sort the list in place. Hence, a new array of the same size as the input array is created.

Additionally, there is no significant extra space used during the sorting process, except for the temporary variables used in the lambda function during comparison, so the space complexity due to the lambda function remains constant, `O(1)`. Combining these, the overall space complexity remains `O(n)`.