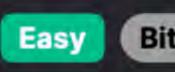
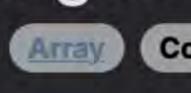
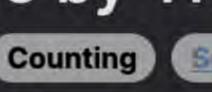
# 1356. Sort Integers by The Number of 1 Bits



Bit Manipulation

Problem Description









Leetcode Link

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 1s in the binary representation of each integer. If two integers have the same number of 1s, 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 1s in their binary representation, and then by their value when there's a

tie on the first criteria.

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

Intuition

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

values themselves, in ascending order.

change the actual elements of the array, but it is used to guide the sort 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

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 1s 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 1s, the smaller number comes first, satisfying the secondary sorting criterion. The sorted list according to these criteria will thus be the result we return.

The provided Python solution makes use of Python's higher-level functionality to implement the sorting logic cleanly and efficiently.

Lambda Functions

Solution Approach

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.

To understand the solution's implementation, let's break down the key components and the patterns leveraged in the code:

## 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.

Tuple Sorting

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 1s 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

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.

(meaning the elements have the same number of 1s in their binary representations), the second element of the tuple (the

- By combining these Python features, the solution elegantly and efficiently sorts the array arr according to the problem's specifications. Example Walkthrough
- First, we'll determine the binary representation of each number and count the number of 1s: The binary representation of 3 is 11, which has 2 ones.

## The binary representation of 4 is 100, which has 1 one.

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

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

The binary representation of 1 is 1, which has 1 one.

The binary representation of 2 is 10, which has 1 one.

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)

```
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 1s and 2 is smaller than 4.
```

**Python Solution** 

class Solution:

Java Solution

class Solution {

6

9

16

17

18

19

20

21

18

21

22

23

24

27

26 };

• For 2: (1, 2)

• For 4: (1, 4)

and then by their integer value in case of a tie.

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

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

When we pass these tuples to the sorted function, it will sort the numbers first by the number of 1s in their binary representation,

# 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. 10 # Example usage:

# 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

satisfies both the primary and secondary sorting criteria specified in the problem.

return sorted(arr, key=lambda x: (bin(x).count('1'), x))

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

int n = arr.length; // Store the length of the array

// the original numbers, preserving the new order.

2 // In the case of a tie, numbers are sorted by their value.

const countBits = (num: number): number => {

function sortByBits(arr: number[]): number[] {

for (int& num : arr) {

return arr;

Typescript Solution

// Return the sorted array.

// Add to each element in the array a value that represents

// the bit count of the number multiplied by 100000 to ensure

// After sorting retrieve the original values by taking modulo 100000

arr[i] %= 100000; // Reduce each element back to original value

public int[] sortByBits(int[] arr) {

for (int i = 0; i < n; ++i) {

for (int i = 0; i < n; ++i) {

// it is prioritized in the sorting

def sort\_by\_bits(self, arr: List[int]) -> List[int]:

# are sorted based on their value ('x').

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

## int bitCount = Integer.bitCount(arr[i]); // Count number of 1-bits in arr[i] 11 arr[i] += bitCount \* 100000; // Add 100000 for each 1-bit to prioritize in sorting 12 13 14 Arrays.sort(arr); // Sort the array with modified values 15

```
22
           return arr; // Return the sorted array by bits
23
24 }
25
C++ Solution
1 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;
10
11
12
           // 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());
15
16
           // Iterate through the array to revert the transformation and obtain
```

num %= 100000; // Remove the added portion to get back the original number.

1 // Function to sort an array of numbers based on the number of 1-bits each number has.

// Helper function to count the number of 1-bits in a binary representation of a number.

## let count = 0; while (num) {

```
// Remove the rightmost 1-bit from the number
              num \&= num - 1;
               // Increment the count of 1-bits
               count++;
12
           return count;
       };
14
15
       // Sorting the array based on the number of 1-bits each number has (asc order).
16
       // In the case of a tie, sort by numerical value (asc order).
17
       return arr.sort((a, b) => {
18
           // First, compare by the number of 1-bits
           const bitCountComparison = countBits(a) - countBits(b);
           if (bitCountComparison !== 0) {
               return bitCountComparison;
23
24
           // If the number of 1-bits is the same, compare by the numbers themselves
25
           return a - b;
       });
26
28
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.

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).

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 0(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.

Space Complexity

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, 0(1). Combining these, the overall space complexity remains O(n).

The space complexity of this function is O(n), as the sorted function returns a new list containing the sorted elements and does not