2917. Find the K-or of an Array

Easy

Problem Description

In this problem, we're given an array of integers called `nums` and an integer `k`. We need to compute the K-or of `nums`. The K-or is a special value that is determined by looking at each bit position across all numbers in `nums`. For each bit position `i`, if at least `k` numbers in `nums` have a `1` in the `i`th bit position, then the `i`th bit of the K-or will also be `1`. Otherwise, it will be `0`. The K-or of `nums` is the combined result of checking this condition for each bit position from 0 to 31 (since we are dealing with 32-bit integers). This problem is essentially about understanding and manipulating binary representations of integers.

Intuition

To understand the solution, we need to grasp the concept of bitwise operations. Specifically, we will be using bitwise AND (`&`) and bitwise OR (`|`). A bitwise AND between two bits results in `1` if both bits are `1`, otherwise it's `0`. A bitwise OR between two bits results in `1` if at least one of the bits is `1`.

To find the K-or of `nums`, we follow these steps:

- We initialize our answer variable `ans` to `0`. This will hold the final K-or value.
- We iterate over each bit position `i` from 0 to 31 (since the problem states that `nums` is an array of non-negative integers, and non-negative integers are represented by 32-bit binary numbers).
- For each bit position `i`, we count how many numbers in `nums` have the `i`th bit set to `1`. We do this by right-shifting (`>>`) each number in `nums` by `i` positions, thus bringing the `i`th bit to the rightmost position. We then perform a bitwise AND with `1` (which has only the rightmost bit set to `1`). This gives us the value of the `i`th bit. Summing up these values across all numbers in `nums` gives us the total count of numbers with `i`th bit set, `cnt`.
- If `cnt` is greater than or equal to `k`, then we should have the `i`th bit of `ans` set to `1`. To achieve this, we use the bitwise OR operation to set the `i`th bit of `ans`. We create a value with only the `i`th bit set (`1 << i`) and OR it with `ans`.
- After checking all bit positions, `ans` will be the K-or of `nums`, which we return.

Solution Approach

The implementation of the solution follows a straightforward approach hinging on the concept of bitwise operations.

- The algorithm employs a **for loop** to iterate through 32 possible bit positions (from 0 to 31) since we are working with non-negative integers, which in this case are 32-bit.

  For each bit position `i`, the algorithm performs a key operation:

  - It counts the number of elements in `nums` (`cnt`) that have the `i`th bit set to `1`. This is achieved by iterating through each element `x` in `nums` and applying `x >> i & 1`.
    - `x >> i` shifts the bits of `x` right by `i` positions, moving the `i`th bit to the least significant position.
    - The `& 1` operation is a bitwise AND which masks all bits other than `x`'s least significant bit; effectively it extracts the value of the `i`th bit post-shift.
  - If the count `cnt` is greater than or equal to `k`, this indicates that at least `k` elements of `nums` have the `i`th bit set to `1`, and thus we should set the `i`th bit in our solution `ans`.
  - To set the `i`th bit in `ans`, we use the operation `ans |= 1 << i`. `1 << i` creates a number with only the `i`th bit set (`2^i`), and `|=` (bitwise OR assignment) combines this with the current `ans`.
    - If `ans` already has the `i`th bit set, `|=` will leave that bit set.
    - If `ans` does not have the `i`th bit set and `cnt` is greater than or equal to `k`, `|=` will set that bit to `1`.
- By the end of the loop, we have considered all bits from the least significant to the most significant and `ans` contains all the bits that met the condition (a count of at least `k`).
- The algorithm then returns the accumulated result stored in `ans` as the K-or of `nums`.

There is no need for any additional data structures for this approach, which keeps the space complexity to O(1) - only counters and the answer variable. The time complexity is O(N * M), with N being the length of the input list and M being the number of bits we are checking (in this case, 32). The simplicity comes with the direct counting and bitwise manipulation for each of the bit positions. The reference to `2^i` in the solution approach, wrapping it in "`", makes clear the relationship between shifting bits and powers of two which is central to understanding bitwise operations in this context.

Example Walkthrough

Let's consider an example where the `nums` array is `[3, 5, 12]` and `k` is set to `2`. The problem asks us to find the K-or of `nums`.

First, we examine the binary representations of each number in the array:

- `3` in binary is `0011`
- `5` in binary is `0101`
- `12` in binary is `1100`

We need to look through each bit position and count how many numbers have that bit set to `1`. Then we'll check if this count is at least `k` (2 in our example).

Let's step through a few iterations of the process for each bit position `i` from 0 to 31.

- For `i = 0` (the least significant bit or LSB), we have:
  - `3 >> 0 & 1 = 1`
  - `5 >> 0 & 1 = 1`
  - `12 >> 0 & 1 = 0` The count `cnt` of numbers with the least significant bit set to `1` is `2`, which is equal to `k`. Thus, we set the least significant bit of `ans` to `1`. Now, `ans = 0001`.
- For `i = 1` (the second least significant bit), we have:
  - `3 >> 1 & 1 = 1`
  - `5 >> 1 & 1 = 0`
  - `12 >> 1 & 1 = 0` The count `cnt` is `1`, which is less than `k`. We do not change `ans`. `ans` remains `0001`.
- For `i = 2`, we have:
  - `3 >> 2 & 1 = 0`
  - `5 >> 2 & 1 = 1`
  - `12 >> 2 & 1 = 1` The count `cnt` is `2`, equal to `k`. We set the third least significant bit (2^2 position) of `ans`. Now, `ans = 0101`.

If we continue this process up to `i = 31`, we would see that no more bits would meet the condition of having at least `k` counts. This is because our numbers are quite small and do not have `1`s in higher bit positions.

After completing this process for all bit positions, we conclude that the final `ans` equals `0101` in binary, which is `5` in decimal. This is the K-or of `nums [3, 5, 12]` with `k = 2`.

Solution Implementation

Python

```python
from typing import List  # Necessary for type hinting

class Solution:
    def findKOr(self, nums: List[int], k: int) -> int:
        # Initialize the answer to 0. The function will build up
        # the answer by setting bits where appropriate.
        answer = 0

        # Loop through each bit position from 0 to 31 (inclusive)
        # since an integer in Python has 32 bits
        for i in range(32):
            # Count number of integers in nums where the ith bit is set
            count = sum(1 for x in nums if x >> i & 1)

            # If the count is greater than or equal to k, it means at least k numbers
            # have a bit set at the ith position. Hence, set the ith bit in the answer.
            if count >= k:
                answer |= 1 << i

        # Return the computed answer where bits are set based on the count of set bits
        # at every bit position in the input numbers compared to k.
        return answer
```

Java

```java
class Solution {

    /**
     * Finds an integer which has at least k 1-bits in the respective bit positions,
     * when compared across all numbers in the given array.
     *
     * @param nums Array of integers.
     * @param k Number representing the minimum count of 1-bits.
     * @return The integer with at least k 1-bits in the corresponding bit positions.
     */
    public int findKOr(int[] nums, int k) {
        // Initialize the variable to store the answer
        int answer = 0;

        // Iterate through each bit position (0 to 31, assuming 32-bit integers)
        for (int i = 0; i < 32; ++i) {
            // Initialize a counter for the number of 1-bits in the current bit position
            int count = 0;
            // Iterate through all numbers in the array
            for (int number : nums) {
                // Shift the number i bits to the right and bitwise AND with 1
                // This extracts the i-th bit and increases the count if it's 1
                count += (number >> i) & 1;
            }
            // If the count is at least k, set the i-th bit of the answer to 1
            if (count >= k) {
                answer |= 1 << i;
            }
        }
        // Return the computed answer with the required bits set to 1
        return answer;
    }
}
```

C++

```cpp
class Solution {
public:
    // Method to find the number where each bit position has appeared at least k times across all numbers in the vector.
    int findKOr(vector<int>& nums, int k) {
        int result = 0; // Initializes the result which will be returned.

        // Iterate through each bit position (0 to 31 for a 32-bit integer).
        for (int i = 0; i < 32; ++i) {
            int count = 0; // Count the number of times the i-th bit is set across all numbers in nums.

            // Iterate through all numbers in the vector.
            for (int num : nums) {
                // If the i-th bit is set in num, increment the count.
                count += (num >> i) & 1;
            }

            // If the i-th bit is set at least k times across all numbers, set this bit in the result.
            if (count >= k) {
                result |= (1 << i);
            }
        }

        // Return the result, where each bit represents it appeared at least k times across all numbers in nums.
        return result;
    }
};
```

TypeScript

```typescript
function findKOr(nums: number[], k: number): number {
    let answer = 0; // Initialize the answer which will hold the result

    // Iterate over 32 bits because we're assuming numbers are within 32-bit integer range
    for (let bitPosition = 0; bitPosition < 32; ++bitPosition) {
        let count = 0; // Count the number of times a bit is set at the current position

        // Iterate over the array and count the bits set at the current bit position
        for (const num of nums) {
            count += (num >> bitPosition) & 1;
        }

        // If the count of set bits is greater than or equal to k, set the bit at this position in the answer
        if (count >= k) {
            answer |= 1 << bitPosition;
        }
    }

    // Return the final calculated answer
    return answer;
}
```

```python
from typing import List  # Necessary for type hinting

class Solution:
    def findKOr(self, nums: List[int], k: int) -> int:
        # Initialize the answer to 0. The function will build up
        # the answer by setting bits where appropriate.
        answer = 0

        # Loop through each bit position from 0 to 31 (inclusive)
        # since an integer in Python has 32 bits
        for i in range(32):
            # Count number of integers in nums where the ith bit is set
            count = sum(1 for x in nums if x >> i & 1)

            # If the count is greater than or equal to k, it means at least k numbers
            # have a bit set at the ith position. Hence, set the ith bit in the answer.
            if count >= k:
                answer |= 1 << i

        # Return the computed answer where bits are set based on the count of set bits
        # at every bit position in the input numbers compared to k.
        return answer
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n * 32)`, which simplifies to `O(n)` since 32 is a constant. Here, `n` is the length of the array `nums`. The time complexity assessment comes from the single loop running 32 times (once for each bit in a 32-bit integer) and the inner computation that performs a bitwise operation and a sum for all `n` elements in the array `nums`. This inner computation runs in `O(n)` time, and since it's nested inside a loop that runs a constant number of times, the overall time complexity remains linear with respect to the number of elements in `nums`.

Space Complexity