2568. Minimum Impossible OR

Medium | Bit Manipulation | Brainteaser | Array

Leetcode Link

Problem Description

The problem provides us with an array `nums` that is 0-indexed. The goal is to determine the smallest positive non-zero integer that cannot be expressed as the bitwise OR of any subsequence of elements from `nums`.

To understand this, recall that a subsequence is a sequence that can be derived from the array by deleting some or no elements without changing the order of the remaining elements. The bitwise OR operation takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits. A bit is set (1) if one or both bits at that position is 1.

An integer `x` is **expressible** if `x` equals the result of the bitwise OR applied to a subsequence of `nums`. The problem asks us to find the smallest integer that is not expressible in such a way.

Intuition

Our intuition for solving this problem is based on the properties of the bitwise OR operation. Considering the nature of bitwise OR, any expressible number must be less than or equal to the sum of the max number in `nums` and all the largest bits set in the other numbers of `nums`.

Notice that if a bit is never set amongst all numbers of the array `nums`, then the number that has this bit as the least significant bit that is set (the smallest power of two greater than any of the numbers in `nums`) cannot be expressed using the given numbers. This is because you can only set a bit in the result of a bitwise OR operation if that bit is set in at least one number of the subsequence.

The solution takes advantage of this by iteratively checking, starting from the smallest power of two (1, 2, 4, 8, ...), whether each power of two is present in the set `s`, which contains all numbers from `nums`. When it finds a power of two that is not present, it returns that number as the smallest non-expressible number.

For example, if `nums` only contains the number 3 (`nums = [3]`), the smallest not expressible number would be 1 << 2, which is 4, since the numbers 1 (1 << 0) and 2 (1 << 1) are expressible from `nums` using the numbers 1 and 2 or 3, which when performed bitwise OR operation results in 1 and 2 respectively. However, since 4 is not in `nums`, no subsequence can produce a set bit at the position that corresponds to 4.

This code iterates through the powers of two until it finds one not present in the array `nums`—that power of two is the smallest number that can't be formed by a bitwise OR of elements of `nums`.

Solution Approach

In the provided solution, we see a straightforward approach using Python's set data structure and generator expression:

1. The set data structure `s` is created from `nums` to allow for constant time (`O(1)`) lookups when checking if a number is in the set. This operation is critical because the algorithm needs to repeatedly check for the presence of powers of two in `nums`.
2. The generator expression `(1 << i for i in range(32))` iteratively computes powers of two, from `1` (i.e., `1 << 0`) up to `2**31` (i.e., `1 << 31`). Since `nums` is an array of integers, and the largest integer in a 32-bit system is `2**31 - 1`, we don't need to check beyond `1 << 31`.
3. Using Python's `next` function, we find the first power of two not in set `s` by checking if `1 << i` is not in `s`. The `next` function will stop at the first occurrence where the condition matches, which is efficient because it avoids unnecessary iterations for higher powers of two that aren't needed.
4. The use of bitwise shift `1 << i` is a clever choice because it efficiently calculates powers of two, which are exactly the numbers we need to check. Each power of two has only one bit set, starting from the least significant bit (for `1 << 0`) to the 31st bit (for `1 << 31`).
5. Finally, the result of the `next` function is returned, which is the first power of two that cannot be formed by 'OR'ing any subsequence of `nums`. This value is essentially the answer to the problem.

This approach works because, as previously described, a number that's not in `nums` and has a bit set that isn't set in any of `nums` is by definition not expressible. Since powers of two have only one bit set, they serve as perfect candidates for finding the smallest such number. This algorithm runs in `O(N)` time where `N` is the number of elements in `nums` due to the set construction. The rest of the operation checking for the first missing power of two runs in constant time because there is a pre-defined limit of `32` iterations (bit sizes of standard integers).

Example Walkthrough

Let's use a small example to illustrate the solution approach.

Suppose our input `nums` array is `[1, 5, 7]`. We want to find the smallest positive non-zero integer that cannot be expressed as the bitwise OR of any subsequence of elements from `nums`.

1. First, we convert `nums` into a set `s` to benefit from constant time lookup. Our set `s` is `{1, 5, 7}`.
2. Next, we start checking for the smallest power of two that is not present in set `s` using the generator expression `(1 << i for i in range(32))`. This will produce `[1, 2, 4, 8, ..., 2**31]`.
3. The bitwise OR of any subsequence of `[1, 5, 7]` can give us the following results:
   - Using `1` from `nums`, bitwise OR gives us `1`.
   - Using `5` from `nums`, bitwise OR gives us `5`.
   - Using `7` from `nums`, bitwise OR gives us `7`.
   - By performing bitwise OR on `1` and `5`, we get `1 | 5 = 5`.
   - By performing bitwise OR on `1` and `7`, we get `1 | 7 = 7`.
   - By performing bitwise OR on `5` and `7`, we get `5 | 7 = 7`.
   - By using all elements `1, 5, 7`, we get `1 | 5 | 7 = 7`.
4. From the above bitwise OR operations, we know that `1` and `5` are expressible. Now we need to check for the smallest power of two not in `s`.
5. Going through the powers of two, we find:
   - `1` is in `s` (`1` is expressible).
   - `2` is not in `s`, but we can get `2` by expressing `1 | 1 = 2` (`2` is expressible).
   - `4` is not in `s`, and we cannot create it by using a bitwise OR on any of the available numbers since none of them has the third bit set.
6. Since `4` (which is `1 << 2`) cannot be expressed by any subsequence of `nums`, it is the smallest number that cannot be formed by a bitwise OR of elements from this array. Thus, `4` is the smallest expressible number for the given `nums`.

By following these steps using the provided solution approach, we conclude that for the input array `[1, 5, 7]`, the smallest positive non-zero integer that cannot be expressed as the bitwise OR of any subsequence of elements from `nums` is `4`.

Python Solution

```python
from typing import List

class Solution:
    def minImpossibleOR(self, nums: List[int]) -> int:
        # Create a set of unique values from the nums list for fast lookup
        unique_numbers = set(nums)

        # Iterate over the range of 32 bits, which is sufficient for all integers
        for i in range(32):
            # Calculate the current power of two
            power_of_two = 1 << i

            # Check if this power of two is not in the unique_numbers set
            if power_of_two not in unique_numbers:
                # Return the smallest power of two that is not in the set
                # This value cannot be formed by OR operations of the numbers in the set
                return power_of_two

# The next() function and generator expression were replaced with a for loop
# for improved readability and understanding.

```

Java Solution

```java
class Solution {
    public int minImpossibleOR(int[] nums) {
        // Initialize a hash set to store unique OR results
        Set<Integer> uniqueORResults = new HashSet<>();

        // Add all numbers from the input array to the set
        for (int number : nums) {
            uniqueORResults.add(number);
        }

        // Loop through each bit position starting from 0
        for (int i = 0; ; ++i) {
            // Calculate the current power of 2 (1 shifted i times to the left)
            int powerOfTwo = 1 << i;

            // If the set does not contain this power of two,
            // it means that we've found the minimum impossible OR result
            if (!uniqueORResults.contains(powerOfTwo)) {
                return powerOfTwo;
            }
        }
        // The loop above is infinite, as it does not have a breaking condition.
        // It is assumed that the function will always find a minimum impossible OR
        // and return from inside the loop.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>

class Solution {
public:
    // This function finds the minimum impossible OR-sum for a given set of numbers.
    int minImpossibleOR(vector<int>& nums) {
        // Create a set to store unique values from nums for efficient look-up.
        unordered_set<int> uniqueNums(nums.begin(), nums.end());

        // Iterate to find the smallest power of 2 that is not present as OR-sum in nums.
        for (int i = 0; ; ++i) {
            // Check if the current power of 2 is missing from the set.
            if (uniqueNums.count(1 << i) == 0) {
                // If missing, this is the minimum impossible OR-sum, so return it.
                return 1 << i;
            }
        }

        // The loop is guaranteed to break at the return statement,
        // hence there is no explicit return value outside of the loop.
        // This works under the assumption that at some i, `1 << i` won't be in `uniqueNums`.
    }
};

```

Typescript Solution

```typescript
function minImpossibleOR(nums: number[]): number {
    // Initialize a set to store unique elements from the input array
    const uniqueElements: Set<number> = new Set();

    // Iterate over the input array and add each number to the set
    for (const num of nums) {
        uniqueElements.add(num);
    }

    // Start checking for the smallest missing integer with 0
    let missingInteger = 0;

    // Iterate indefinitely as we will return from within the loop when the condition is met
    while (true) {
        // Check if the current power of 2 (1 shifted left by missingInteger places) is not in the set
        if (!uniqueElements.has(1 << missingInteger)) {
            // If it's not in the set, we found our smallest missing integer and return it
            return 1 << missingInteger;
        }
        // If it is in the set, increment missingInteger to check the next power of 2
        missingInteger++;
    }
}

```

Time and Space Complexity

The time complexity of the provided code is `O(1)`. This is because the number of iterations is bound by 32, which corresponds to the number of bits in an integer when using a standard 32-bit representation. The loop will run at most 32 times, regardless of the input size, because it checks for the presence of powers of two (using `1 << i`) in the set `s`.

The space complexity of the code is also `O(1)`. The set `s` is created with the elements of the input list `nums`, but this does not depend on the size of the input with respect to the range of numbers (0 to 31) we are checking. Hence, the space used by the set is constant with respect to the size of the input list. The only other space utilized is the space for the variable `i` and the output which is also constant.