898. Bitwise ORs of Subarrays

Medium Bit Manipulation Array Dynamic Programming Leetcode Link

Problem Description

The problem presents us with a task: given an array `arr` of integers, we are required to find the total number of distinct bitwise OR results that can be obtained from all possible non-empty contiguous subarrays of `arr`.

To understand the problem better, let's clarify some concepts:

- **Bitwise OR** (`|`): A binary operation that takes two bit patterns of equal length and performs the logical inclusive OR operation on each pair of corresponding bits. The result in each position is `1` if at least one of the bits is `1`.
- **Subarray**: A sequence of elements within an array that is contiguous (elements are consecutive without gaps) and non-empty.
- **Distinct**: Each unique value should be counted only once in the final tally.

The problem, therefore, requires us to explore and assess every possible subarray that can be generated from the given array, perform the bitwise OR operation on its elements, and count the unique outcomes.

Intuition

The intuitive approach to solving this problem might involve a brute force strategy - calculating the bitwise OR for every subarray and then utilizing a set to count the distinct values. However, this approach would result in a high time complexity due to the multiple nested loops required (each subarray is recursively built starting from each element).

The solution code, however, uses a clever strategy that reduces the amount of redundant computation:

1. **Iterative Building**: It takes advantage of the fact that the bitwise OR of a subarray ending at position `i` depends upon the bitwise OR result of the subarray ending at `i-1`. The `prev` variable is used to accumulate the bitwise OR result till the current element, so that for the next element, we don't have to start from scratch.
2. **Early Breaking**: It recognizes that if a new subarray's bitwise OR equals the accumulated OR (`prev`), we can stop early because all subsequent subarrays will only give the same or greater OR results, which we would already have encountered due to the nature of the OR operation building on previous subarrays.
3. **Set for Uniqueness**: The set `s` is used to store unique bitwise OR results. This way, every time we add a new OR result, duplicate values are inherently avoided.

The main algorithm goes as follows:

- Initialize a set `s` to keep track of distinct bitwise OR results.
- Loop over each element `v` in the array:
  - Start a `prev` accumulator that holds the bitwise OR up to the current element.
  - Loop backwards from the current element to the beginning of the array:
    - Calculate the bitwise OR for the current subarray and add it to the set `s`.
    - Once the current subarray OR equals the accumulated `prev` OR, break early.
- Finally, return the length of set `s`, which represents the number of distinct bitwise ORs.

By using the set and early breaking, the solution immensely reduces the number of calculations compared to the brute force approach, and therefore, is much more efficient.

Solution Approach

The implementation of the reference solution makes use of sets and two pointers to efficiently compute the distinct bitwise ORs of all subarrays. Here's how the approach works in detail:

1. We initialize an empty set `s` that will hold the distinct bitwise OR results. Apart from this, we have a `prev` variable that is used to keep track of the bitwise OR up to the current element in the outer loop, which goes through each element `i` in the array.
2. As we iterate over each element `v` in the array using an index `i`, we first update the `prev` variable to hold the bitwise OR between itself and the current element (`prev |= v`). This new `prev` will be the bitwise OR of all elements from the beginning of the array up to the current element `i`.
3. In the inner loop, we start from the current element and go backwards through the array using another index `j`. We create a new variable `curr` to hold the result of bitwise ORs for the subarrays ending in the current element `i`.
4. During each iteration of the inner loop, we update `curr` to be the bitwise OR of itself and the current `j`th element (`curr |= arr[j]`).
5. After updating `curr`, we add it to the set `s`. This operation ensures that only distinct OR results are kept, as sets do not store duplicate elements.
6. The early breaking condition is checked (`if curr == prev`), which is based on the understanding that once the current subarray's OR matches the overall OR up to the current element (`prev`), all subsequent larger subarrays will yield the same OR value and have been considered before. When this condition is met, the inner loop breaks, avoiding unnecessary computation.
7. Lastly, once both loops are done, the length of set `s` will represent the number of distinct bitwise ORs that can be formed from all the subarrays of `arr`. This is the result that the function returns.

The above steps form an efficient algorithm as it reduces the number of subarrays we need to check, leveraging the properties of the bitwise OR operation and making use of sets to maintain distinct entries. Such an optimization is essential to pass all test cases on platforms like LeetCode, where the input size can be large and brute force methods would result in a timeout error.

Example Walkthrough

To illustrate the solution approach, let's take a small example:

Let's say we have an array `arr` = [1, 2, 3].

We will apply the solution approach step by step:

1. **Initialize a set and variables**:
   - Set `s` = {}
   - Variable `prev` is initialized.
2. **Outer loop - Iterate over each element in the array**:
   - For element 1 (`i=0`), set `prev = 1`.
   - For element 2 (`i=1`), `prev` becomes `prev | 2 = 1 | 2 = 3`.
   - For element 3 (`i=2`), `prev` becomes `prev | 3 = 3 | 3 = 3`.
3. **Inner loop - Backward iteration from current element**:
   - At `i=0`: Only one subarray `[1]`:
     - `curr = 1`
     - Set `s` becomes {1}
   - At `i=1`: Subarrays `[2]`, `[1, 2]`:
     - `curr = 2`, add to set `s` → {1, 2}
     - `curr = 2 | 1 = 3`, add to set `s` → {1, 2, 3}
     - Since `curr` now equals `prev`, inner loop breaks early.
   - At `i=2`: Subarrays `[3]`, `[2, 3]`, but we stop at `[2, 3]` because:
     - `curr = 3`, already in set `s`, no need to add.
     - Loop attempts to compute `curr = 3 | 2`, but since `prev` is already `3`, the inner loop breaks.
4. **Early Breaking**:
   - Utilized at each step in the inner loop when `curr` matches `prev`.
5. **Final Result**:
   - The length of set `s` is 3, representing the distinct bitwise OR results: {1, 2, 3}.

Each bitwise OR operation only computes new results, while duplicates are ignored due to the set data structure. By breaking early from the inner loop, we avoid unnecessary computation, optimizing the process. This compact example covers all the main elements present in the solution approach, demonstrating its efficiency and how it leads to the final answer.

Python Solution

```python
class Solution:
    def subarrayBitwiseORs(self, arr: List[int]) -> int:
        # Initialize a set to store unique bitwise OR results
        unique_or_results = set()
        # 'prev' will hold the cumulative OR result of the current iteration
        prev = 0
        # Iterate over the input array with both value and index
        for index, value in enumerate(arr):
            # Update 'prev' by taking the OR with the current value
            prev |= value
            # 'current' will hold the cumulative OR result starting from 'index' going back to the start
            current = 0
            # Iterate backwards from the current index to the start of the array
            for j in range(index, -1, -1):
                # Update 'current' by taking the OR with the value at j
                current |= arr[j]
                # Add 'current' to the set of unique OR results
                unique_or_results.add(current)
                # If 'current' equals 'prev', no new unique values can be found by continuing; break
                if current == prev:
                    break
        # Return the number of unique bitwise OR results found
        return len(unique_or_results)

```

Java Solution

```java
class Solution {
    public int subarrayBitwiseORs(int[] arr) {
        // We use a set to store unique values of bitwise ORs for all subarrays
        Set<Integer> uniqueBitwiseORs = new HashSet<>();

        // We iterate through each element in the array
        for (int i = 0; i < arr.length; ++i) {
            // 'aggregate' will hold the cumulative bitwise OR value up to the current element
            int aggregate = 0;

            // We iterate from the current element down to the start of the array
            for (int j = i; j >= 0; --j) {
                // We calculate the bitwise OR from the current element to the 'jth' element
                aggregate |= arr[j];

                // Add the current subarray's bitwise OR to the set
                uniqueBitwiseORs.add(aggregate);

                /* If the current aggregate value is the same as the previous
                   aggregate value, all future aggregates will also be the same
                   due to the properties of bitwise OR, so we break out early. */
                if (aggregate == (aggregate | arr[i])) {
                    break;
                }
            }
        }

        // Return the number of unique bitwise ORs found
        return uniqueBitwiseORs.size();
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to count the number of distinct bitwise OR values of all subarrays.
    int subarrayBitwiseORs(vector<int>& arr) {
        unordered_set<int> uniqueOrValues;  // To store unique OR values of subarrays.
        int currentOr = 0;                  // To store the running OR value of the current subarray.

        // Iterate over each element in the array.
        for (int i = 0; i < arr.size(); ++i) {
            currentOr |= arr[i];            // Update the running OR with the current element.
            int subarrayOr = 0;             // Used to calculate OR for each possible subarray ending at 'i'.

            // Iterate from the current element to the beginning of the array.
            for (int j = i; j >= 0; --j) {
                subarrayOr |= arr[j];        // Update the OR for the subarray ending at 'i' starting at 'j'.
                uniqueOrValues.insert(subarrayOr);  // Store the calculated OR in the set.

                // Break the loop if the subarray OR equals the currently calculated OR (all bits already set).
                if (subarrayOr == currentOr) break;
            }
        }

        // The size of the set represents the number of distinct OR values of all subarrays.
        return uniqueOrValues.size();
    }
};

```

Typescript Solution

```typescript
// The TypeScript standard library already includes Set, so we don't need
// a separate import for unordered_set as we would in C++.

// Function to count the number of distinct bitwise OR values of all subsequences.
function subarrayBitwiseORs(arr: number[]): number {
    let uniqueOrValues: Set<number> = new Set(); // To store unique OR values of subarrays.
    let currentOr: number = 0;                   // To store the running OR value of the current subarray.

    // Iterate over each element in the array.
    for (let i = 0; i < arr.length; i++) {
        currentOr |= arr[i];                     // Update the running OR with the current element.
        let subarrayOr: number = 0;              // Used to calculate OR for each possible subarray ending at 'i'.

        // Iterate from the current element to the beginning of the array.
        for (let j = i; j >= 0; j--) {
            subarrayOr |= arr[j];               // Update the OR for the subarray ending at 'i' starting at 'j'.
            uniqueOrValues.add(subarrayOr);     // Store the calculated OR in the set.

            // Break the loop if the subarray OR equals the currently calculated OR (all bits already set).
            if (subarrayOr === currentOr) break;
        }
    }

    // The size of the set represents the number of distinct OR values of all subarrays.
    return uniqueOrValues.size;
}

export { subarrayBitwiseORs }; // Export the function to be available for import in other modules.

```

Time and Space Complexity

The given code aims to find the number of distinct subarray bitwise ORs. To do this, it iterates over the given array and computes the OR of elements from the current element to all previous elements by keeping a record of the previous OR in `prev` and the current progression in `curr`.

Time Complexity

The time complexity of this algorithm mainly depends on the number of iterations within the double-loop structure.

- The outer loop runs exactly `n` times where `n` is the number of elements in `arr`.
- The inner loop runs up to `i+1` times in the worst case (when `curr` never equals `prev` early).

However, due to the properties of the bitwise OR operation, repetitions are likely to occur much earlier, resulting in earlier breaks from the inner loop. Specifically, the sequence of ORs will eventually stablize into a set of values that does not grow with each additional OR operation. The actual number of unique elements in these OR sequences across all iterations is bounded by a factor much smaller than `n^2`.

While it's difficult to put a precise bound on this without specifics about the input distribution, let's denote the average unique sequence length as `k` (which is considerably smaller than `n` due to the saturation of OR operations). Therefore, the total number of operations is approximately `O(n*k)`.

However, it is important to note that `k` is not guaranteed to be a constant and its relation with `n` can depend heavily on the input, implying that in the worst case the time complexity could tend towards `O(n^2)`, but in practical scenarios, it is expected to perform significantly better.

Space Complexity

The space complexity is due to the set `s` that is used to store the unique subarray OR results.

- In the worst case, each subarray OR could be unique, which means the set could grow to the size of the sum of all subarray counts. As with the time complexity argument, this won't actually occur due to the saturation of bitwise ORs.

Let `m` represent the maximum possible unique OR values which can be much less than the total subarray count of roughly `n*(n+1)/2`. Therefore, the space complexity can be approximated as `O(m)`.

In conclusion, the time complexity of the code is approximately `O(n*k)` (with `k` being influenced by the input nature and much smaller than `n`) and space complexity is around `O(m)` for storing the unique OR results set, where `m` represents the maximum number of unique OR values across all subarrays.