2059. Minimum Operations to Convert Number

Medium | Breadth-First Search | Array | Leetcode Link

Problem Description

In this problem, we are provided with an array `nums` containing distinct integers and two other integers: `start` and `goal`. Our objective is to transform the value of a variable `x` from `start` to `goal`. The variable `x` can be modified by repeatedly performing the following operations:

- Adding any number `nums[i]` to `x` (`x + nums[i]`)
- Subtracting any number `nums[i]` from `x` (`x - nums[i]`)
- Performing a bitwise-XOR between `x` and any number `nums[i]` (`x ^ nums[i]`)

We are allowed to use each element in `nums` repeatedly in any order. The range of permissible values for `x` during the operations is from `0` to `1000`, inclusive. If an operation results in `x` falling outside this range, no further operations can be applied, although reaching an out-of-range number does not necessarily mean failing to reach the goal—it's possible that the goal, too, is out of range.

Our task is to find the minimum number of operations required to turn `x` from `start` to `goal` or to determine that such a transformation is impossible, in which case we would return `-1`.

Intuition

To find the minimum number of operations to reach the goal, we can use Breadth-First Search (BFS). BFS is a suitable approach since we want to find the shortest path—here, the smallest number of operations—to transform `start` into `goal`. BFS ensures that we explore all possibilities generated by the three operations (addition, subtraction, and XOR) level by level. By visiting each "level" sequentially, we guarantee that the first time we reach the `goal`, it is the minimum number of steps taken.

The BFS algorithm starts with `start`, applying all possible operations while maintaining a queue. Each element in the queue is a pair: the resulting number after applying an operation and the count of operations performed to reach that number. We mark each visited number within the range of `0` to `1000` to prevent re-visiting the same number multiple times and getting into infinite loops.

Whenever any operation results in the `goal`, we immediately return the current step count plus one. If the queue is exhausted without finding the `goal`, we conclude it is not possible to reach the `goal` with the given operations and return `-1`.

The algorithm effectively explores different pathways of reaching `goal` from `start` while avoiding duplicate work, thus efficiently finding the minimum steps required.

Solution Approach

The provided solution in Python makes use of the Breadth-First Search (BFS) algorithm. This solution first defines three operations as lambdas that take two arguments `x` and `y` and return `x + y`, `x - y`, and `x ^ y` respectively. These lambdas represent the three permissible operations on `x` as described in the problem statement.

The BFS algorithm is implemented as follows:

1. A boolean visited array `vis` of size `1001` is used to keep track of numbers in the range `0` to `1000` that have already been explored. Initially, all values are set to `False` since no numbers have been explored.

2. A queue `q` is initiated with a tuple containing the `start` number and step counter set to `0`. The Python deque (double-ended queue) is chosen for efficiency since we need to add and remove elements from both ends.

3. The BFS starts by popping elements from the queue one by one. For each number `x` dequeued along with its associated step count, we:

   - Iterate over each number `num` in `nums`.
   - For each number `num`, we apply the three operations using the previously defined lambdas. The result of each operation is stored in `nx`.
4. For each result `nx`, we check if `nx` equals the `goal`. If it does, we have found the shortest path to `goal` and return the current step count plus one, because we have made another operation to reach `goal`.

5. If `nx` is within the bounds (`0 <= nx <= 1000`) and has not been visited before, we add `(nx, step + 1)` to the queue and mark `nx` as visited.

6. If the queue is emptied and `goal` has not been found, this implies that it is not possible to convert `start` into `goal` with the provided operations, and we return `-1`.

The BFS continues to work level by level, guaranteeing that each number is reached in the minimum number of operations, and when `goal` is reached, it is done with the least number of transformations possible.

This algorithm makes efficient use of both time and space by avoiding unnecessary recomputation and by managing the search space with the visited array and the queue for pending computations.

Example Walkthrough

Let's illustrate the solution with a small example:

Assume `nums = [2, 3]`, `start = 0`, and `goal = 10`.

1. We begin by initiating the BFS process with the starting point. Our queue `q = [(0, 0)]`, where `0` is the initial value of `x` and `0` is the step counter. The visited array `vis` is set to `False` for all values.

2. The BFS starts by dequeuing the first element `(0, 0)` from `q`. We perform all three operations on `x` with each number in `nums`.

3. After applying the operations (add, subtract, XOR) with `num = 2`, we get three possibilities: `x + 2 = 2`, `x - 2 = -2`, and `x ^ 2 = 2`. The second result is out of bounds, so only 2 will be queued: `q = [(2, 1)]`.

   Repeating with `num = 3`, we get `x + 3 = 3`, `x - 3 = -3`, and `x ^ 3 = 3`. Again, "-3" is out of bounds, and since `3` is not yet visited, we add it to the queue. Now `q = [(2, 1), (3, 1)]`.

4. Next, we dequeue `(2, 1)` and repeat the operations with all `nums`:

   With `num = 2`, we get `2 + 2 = 4`, `2 - 2 = 0` (already visited), and `2 ^ 2 = 0` (already visited).

   With `num = 3`, we get `2 + 3 = 5`, `2 - 3 = -1` (out of bounds), and `2 ^ 3 = 1`. The new numbers `4` and `1` are added to the queue, while `0` is ignored. Now, `q = [(3, 1), (4, 2), (1, 2)]`.

5. This process continues as we dequeue from `q`, apply operations, and enqueue non-visited results until we either achieve the `goal` of `10` or empty the queue.

   After more iterations, we might eventually reach a state where `q` includes `(10, N)`, where `N` is the number of operations taken. At this point, we return `N + 1` because we have performed one more operation to reach our goal.

If we never reach the value of `10`, and we have no more elements left in our queue to explore, we will return `-1`, signifying it's not possible to reach the goal from the start given the operations.

Through this approach, we ensure that we always take the shortest path since BFS explores all possibilities one step at a time and does not revisit any number we've seen before.

Python Solution

```python
from collections import deque
from typing import List

class Solution:
    def minimum_operations(self, nums: List[int], start: int, goal: int) -> int:
        # Define the possible operations using operator lambdas
        add = lambda x, y: x + y
        subtract = lambda x, y: x - y
        xor = lambda x, y: x ^ y
        operations = [add, subtract, xor]  # Store operations in a list for easy iteration

        # Create an array to keep track of visited states
        visited = [False] * 1001

        # Initialize queue with a tuple containing the starting value and initial step count of 0
        queue = deque([(start, 0)])

        # Run a BFS to explore all possible states
        while queue:
            current_value, step_count = queue.popleft()  # Dequeue the next state to process

            # Iterate over each number in the given array
            for num in nums:
                # Try all three operations with the current number and value dequeued
                for operation in operations:
                    next_value = operation(current_value, num)
                    # Check if we have reached the goal
                    if next_value == goal:
                        return step_count + 1
                    # Enqueue the next state if it is valid and hasn't been visited
                    if 0 <= next_value <= 1000 and not visited[next_value]:
                        queue.append((next_value, step_count + 1))
                        visited[next_value] = True

        # Return -1 if the goal is not reachable with the given operations and constraints
        return -1

# Example usage:
# solution = Solution()
# result = solution.minimum_operations([1,3],[2],[4])
# print("Minimum operations to reach the goal:", result)
```

Java Solution

```java
import java.util.ArrayDeque;
import java.util.Queue;
import java.util.function.IntBinaryOperator;

class Solution {
    public int minimumOperations(int[] nums, int start, int goal) {
        // IntBinaryOperator is a functional interface representing an operation upon two int-valued operands
        // and returning an int-valued result. Here, we define three operations: addition, subtraction, and bitwise XOR.
        IntBinaryOperator add = (x, y) -> x + y;
        IntBinaryOperator subtract = (x, y) -> x - y;
        IntBinaryOperator bitwiseXor = (x, y) -> x ^ y;

        // Array of the operations
        IntBinaryOperator[] operations = {add, subtract, bitwiseXor};

        // A boolean array to keep track of visited values
        boolean[] visited = new boolean[1001];

        // Queue to manage the breadth-first search; each int array holds two elements: position and step count.
        Queue<int[]> queue = new ArrayDeque<>();
        queue.offer(new int[]{start, 0}); // Initial position with 0 steps taken

        // Main loop of breadth-first search
        while (!queue.isEmpty()) {
            int[] current = queue.poll();
            int position = current[0];
            int steps = current[1];

            // Apply each operation with each number in the given array 'nums'
            for (int num : nums) {
                for (IntBinaryOperator operation : operations) {
                    int nextPosition = operation.applyAsInt(position, num);

                    // If goal is reached, return the number of steps taken plus one for the current operation
                    if (nextPosition == goal) {
                        return steps + 1;
                    }

                    // If the next position is within the bounds and not visited, add it to the queue
                    if (nextPosition >= 0 && nextPosition <= 1000 && !visited[nextPosition]) {
                        queue.offer(new int[]{nextPosition, steps + 1}); // Increment step count by 1
                        visited[nextPosition] = true;
                    }
                }
            }
        }

        // If the goal cannot be reached, return -1
        return -1;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional>
#include <queue>

class Solution {
public:
    int minimumOperations(std::vector<int>& nums, int start, int goal) {
        // Define a pair structure for storing value and the current step count.
        using ValueStepPair = std::pair<int, int>;

        // Define the operations that can be applied.
        std::vector<std::function<int(int, int)>> operations{
            [](int x, int y) { return x + y; },
            [](int x, int y) { return x - y; },
            [](int x, int y) { return x ^ y; },
        };

        // Create a visited array to track the numbers that have already been checked.
        std::vector<bool> visited(1001, false);

        // Initialize a queue to perform Breadth-First Search (BFS).
        std::queue<ValueStepPair> queue;
        queue.push({start, 0});

        // Loop until the queue is empty.
        while (!queue.empty()) {
            // Get the front element of the queue.
            auto [currentValue, currentStep] = queue.front();
            queue.pop();

            // Apply each operation with each number in the input array.
            for (int num : nums) {
                for (auto operation : operations) {
                    int nextValue = operation(currentValue, num);

                    // Check if the operation result matches the goal.
                    if (nextValue == goal) {
                        return currentStep + 1;
                    }

                    // If the result is within bounds and not visited, add it to the queue.
                    if (nextValue >= 0 && nextValue <= 1000 && !visited[nextValue]) {
                        queue.push({nextValue, currentStep + 1});
                        visited[nextValue] = true;
                    }
                }
            }
        }

        // If the goal can't be reached, return -1.
        return -1;
    }
};
```

Typescript Solution

```typescript
function minimumOperations(nums: number[], start: number, goal: number): number {
    const numLength = nums.length;

    // Helper function to perform addition
    const add = (x: number, y: number): number => x + y;

    // Helper function to perform subtraction
    const subtract = (x: number, y: number): number => x - y;

    // Helper function to perform bitwise XOR
    const xor = (x: number, y: number): number => x ^ y;

    // Array of operations that can be performed
    const operations = [add, subtract, xor];

    // Use a boolean array to keep track of visited states
    let visited = new Array(1001).fill(false);

    // Initialize queue with the start value and step count of 0
    let queue: Array<[number, number]> = [[start, 0]];
    visited[start] = true;

    // Process the queue until empty
    while (queue.length) {
        // Pop the first element from the queue
        let [currentValue, currentStep] = queue.shift();

        // Iterate over all numbers that can be used for operations
        for (let i = 0; i < numLength; i++) {
            // Apply each operation with the current number
            for (let operation of operations) {
                const nextValue = operation(currentValue, nums[i]);

                // Check if we've reached the goal
                if (nextValue === goal) {
                    return currentStep + 1;
                }

                // Add the result of the operation to the queue if it's valid and not visited
                if (nextValue >= 0 && nextValue <= 1000 && !visited[nextValue]) {
                    visited[nextValue] = true;
                    queue.push([nextValue, currentStep + 1]);
                }
            }
        }
    }

    // -1 indicates that the goal cannot be reached
    return -1;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by the operations performed in the breadth-first search (BFS) algorithm implemented. On each iteration of the `while` loop, the algorithm processes each number in `nums` with each of the three operations (addition, subtraction, XOR).

Let `N` be the length of the `nums` list. At most `1001` different states can be visited since `vis` array is of that size, and any number out of that range is not considered.

For every state, we perform 3 operations for each number in `nums`. Hence, the total number of operations in the worst case would be at most `1001 * 3 * N`. Therefore, the time complexity is `O(N)`.

Space Complexity

The space complexity includes the storage for the queue `q`, the visited states `vis`, and some auxiliary space for function calls and variables.

- The `vis` list contains a fixed space of `1001` elements.
- The queue `q` can, in the worst case, contain all possible states that have been visited. Since a state will not be revisited once marked in the `vis` array, the maximum size of the queue would not exceed `1001`.

Considering both the fixed array and queue, the space complexity is also `O(1)` since these do not scale with the size of the input array `nums`.

Note: The above complexities assume that integer addition, subtraction, and XOR operations are `O(1)`.