1720. Decode XORed Array

Easy Bit Manipulation Array

Problem Description

In the given problem, there is an array named `arr` that contains `n` non-negative integers which we can't see because it is *hidden*. This `arr` array has been transformed into another array named `encoded` with a length of `n - 1`. This transformation uses the bitwise XOR operation: each element of `encoded` is the result of XORing consecutive elements in `arr`, i.e., `encoded[i] = arr[i] XOR arr[i + 1]`. The XOR operation is a bitwise operation that outputs 1 only if the input bits are different, and 0 otherwise.

The challenge is to reconstruct the original `arr` array given the `encoded` array and the first element of `arr` (referred to as `first` or `arr[0]`). The description assures that there is a unique solution for `arr`.

Intuition

The XOR operation plays a crucial role here because it has a unique property: it's reversible if we know one of the operands. Meaning, if we have `a XOR b = c`, we can find `a` by doing `c XOR b`, and similarly, find `b` by doing `c XOR a`. This is because performing XOR twice with the same number cancels out the operation (e.g., `a XOR b XOR b` is equal to `a`).

This property makes it possible to recover the `arr` array starting from the provided `first` element (`arr[0]`). The intuition behind the solution is simply to iterate through the `encoded` array and apply the XOR operation between the last known value of `arr` and the current element in `encoded` to find the next value of `arr`. In essence, given `arr[i]` and `encoded[i]`, we can solve for `arr[i + 1]` by calculating `arr[i] XOR encoded[i]`.

Solution Approach

The implementation utilizes a simple `for` loop and the `append` method on lists in Python. For those unfamiliar with Python, appending to a list adds a new element to the end of the list. The algorithm works as follows:

- Initially, the known first element of `arr` (given as `first`) is appended to an empty list named `ans`.
- We then iterate over each element `e` in the `encoded` array.
- In each iteration, we XOR the last element of `ans` with `e`. In terms of the algorithm, if `ans[-1]` is the last element of the list `ans`, then `e` is XORed with `ans[-1]`, and the result is the next element in `arr`, which is then appended to `ans`.
- This method leverages the reversible property of XOR mentioned earlier: `encoded[i] = arr[i] XOR arr[i + 1]` implies `arr[i + 1] = encoded[i] XOR arr[i]`. Since `arr[i]` is the last known element (initially `first`), we can decode `arr[i + 1]` using the elements of `encoded`.
- The loop continues until we have reconstructed the entire `arr` array.

The `Solution` class and its `decode` method, provided in the reference solution, are examples of the use of Python's object-oriented programming paradigm. The `decode` method encapsulates the aforementioned algorithm. There are no specific data structures besides the list used for the result, and no complex patterns -- it's a straight application of XOR to decode each subsequent number in the sequence.

Here's an explicit breakdown of the steps in the `decode` function:

- Start by creating a result list `ans` with the first element `first`.
- Iterate over the `encoded` array using a `for` loop.
- In each iteration, XOR the last element of `ans` with the current element in `encoded` and append the result to `ans`.
- Continue until all elements in `encoded` have been used.
- The list `ans` is now the decoded `arr` array, which we return from the function.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following `encoded` array and `first` element:

- `encoded = [6, 1, 4]`
- `first = 5` (which is actually `arr[0]`)

We know that `encoded[i]` is derived from `arr[i] XOR arr[i + 1]`. So the original array `arr` starts with `first` and has `n` elements, with `n = len(encoded) + 1`.

Now we will use the given solution approach to decode the original array `arr`.

Step 1: Initialize the result list with `first`

- `ans = [5]`

Step 2: XOR `first` with the first element of `encoded`

- Calculate `5 XOR 6` which equals `3`
- Append `3` to `ans`
- `ans` becomes `[5, 3]`

Step 3: XOR the last element of `ans` (now `3`) with the next element of `encoded`

- Calculate `3 XOR 1` which equals `2`
- Append `2` to `ans`
- `ans` becomes `[5, 3, 2]`

Step 4: XOR the last element of `ans` (now `2`) with the next element of `encoded`

- Calculate `2 XOR 4` which equals `6`
- Append `6` to `ans`
- `ans` becomes `[5, 3, 2, 6]`

At this point, we've performed an XOR operation with all elements of the `encoded` array, and our result list now contains all elements of the original array `arr`.

Final Result:

The decoded original array `arr` is `[5, 3, 2, 6]`.

Using the given solution approach, we are able to reconstruct `arr` from `encoded` and `first`. This illustrates how the reversible property of the XOR operation can be utilized to solve this type of decoding problem iteratively.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def decode(self, encoded: List[int], first: int) -> List[int]:
        # Initialize the result list with the first element
        decoded_list = [first]

        # Iterate over the encoded list and decode each element
        for encoded_element in encoded:
            # The next number is found by XORing the last number in the decoded list
            # with the current encoded element
            decoded_list.append(decoded_list[-1] ^ encoded_element)

        # Return the fully decoded list
        return decoded_list
```

Java

```java
class Solution {

    /**
     * Decodes an encoded array with the given first element value.
     *
     * @param encoded The array of integers to be decoded.
     * @param first The first element of the decoded array.
     * @return The decoded array of integers.
     */
    public int[] decode(int[] encoded, int first) {
        // The length of the decoded array is one more than the length of the encoded array.
        int n = encoded.length;
        int[] decodedArray = new int[n + 1];

        // Setting the first element of the decoded array.
        decodedArray[0] = first;

        // Iterating through the encoded array to decode it.
        for (int i = 0; i < n; ++i) {
            // The current element is obtained by XORing the previous element of the decoded
            // array with the current element of the encoded array.
            decodedArray[i + 1] = decodedArray[i] ^ encoded[i];
        }

        // Returning the decoded array.
        return decodedArray;
    }
}
```

C++

```cpp
#include <vector> // Include the vector header to use std::vector

class Solution {
public:
    // Decodes an encoded vector using the first element
    // @param encoded: the encoded vector of integers
    // @param first: the first element to start decoding
    // @return the decoded vector of integers
    std::vector<int> decode(std::vector<int>& encoded, int first) {
        std::vector<int> decoded; // Create an empty vector to store the decoded numbers
        decoded.push_back(first); // Add the first element to the decoded vector

        // Decode the rest of the encoded vector
        for (int i = 0; i < encoded.size(); ++i) {
            // The next number is found by XORing the current number with the encoded number
            decoded.push_back(decoded[i] ^ encoded[i]);
        }

        return decoded; // Return the fully decoded vector
    }
};
```

TypeScript

```typescript
// Import the Array type from TypeScript for type annotations
import { Array } from "typescript";

// Decode an encoded array using the first element
// @param encoded - the encoded array of numbers
// @param first - the first element to start decoding
// @return the decoded array of numbers
function decode(encoded: Array<number>, first: number): Array<number> {
    let decoded: Array<number> = []; // Create an empty array to store the decoded numbers
    decoded.push(first); // Add the first element to the decoded array

    // Decode the rest of the encoded array
    for (let i = 0; i < encoded.length; i++) {
        // The next number is found by XORing the current number with the encoded number
        decoded.push(decoded[i] ^ encoded[i]);
    }

    return decoded; // Return the fully decoded array
}
```

```python
from typing import List

class Solution:
    def decode(self, encoded: List[int], first: int) -> List[int]:
        # Initialize the result list with the first element
        decoded_list = [first]

        # Iterate over the encoded list and decode each element
        for encoded_element in encoded:
            # The next number is found by XORing the last number in the decoded list
            # with the current encoded element
            decoded_list.append(decoded_list[-1] ^ encoded_element)

        # Return the fully decoded list
        return decoded_list
```

Time and Space Complexity

Time Complexity

The given Python function `decode` consists of a single loop that iterates through the `encoded` list, which has `n` elements, where `n` is the length of the `encoded` list. Within the loop, there is a constant time operation which performs an XOR operation (`^`) and appends the result to the `ans` list. Therefore, since each operation in the loop takes `O(1)` time and the loop runs for `n` iterations, the overall time complexity is `O(n)`.

Space Complexity

The space complexity of the function `decode` is determined by the `ans` list which the function populates and returns. Since the `ans` list will contain exactly `n + 1` elements after processing an `encoded` list of length `n`, the space complexity is `O(n)`. The space required grows linearly with the input size, making the space complexity linear as well.