1734. Decode XORed Permutation

Medium Bit Manipulation Array Leetcode Link

Problem Description

There is an array `perm` which is a permutation of the first `n` positive integers where `n` is an odd number. A permutation means that it contains all numbers from 1 to `n` exactly once in any order. This array `perm` was transformed into another array named `encoded` by taking the XOR (exclusive OR) of each pair of adjacent elements in `perm`. The array `encoded` has a length of `n - 1`. For instance, if we take `perm = [1,3,2]`, the resulting `encoded` array would be `[2,1]` because `1 XOR 3 = 2` and `3 XOR 2 = 1`. Given the `encoded` array, the task is to find out the original array `perm`. It is guaranteed that there is one unique solution for this problem.

Intuition

The intuition behind the solution starts by identifying properties of XOR operation. The XOR operation has an important property which is: if `a XOR b = c`, then it's also true that `a XOR c = b` and `b XOR c = a`. This property can be used to retrieve the original permutation from the `encoded` array.

The first step is to understand that since `n` is always odd, the XOR of all numbers from 1 to `n` will give us a single integer because XOR of a number with itself is 0 and the remaining number will be the one without a pair. This number combined with our XOR sequence can be used to deduce the original array.

1. Compute the XOR of numbers from 1 to `n` (inclusive), which will be referred to as `b`.
2. Since the sequence always starts with the first element of `perm` (call it `perm[0]`), we can compute the XOR of the elements at the even positions of `encoded`. Because `encoded[i] = perm[i] XOR perm[i + 1]`, when we take the XOR of all even `encoded[i]`, we're left with `perm[0] XOR perm[2] XOR ... XOR perm[n-1]`.
3. Now let's call the result from step 2 as `a`. Since `xor` of `a` includes all even positions of the original permutation, excluding all odd positions, and `b` includes all positions, `a XOR b` gives us `perm[0]`, because all even positions except the first position will be canceled out.
4. Knowing `perm[0]`, we can iterate backwards from the last element of `encoded` using another property of XOR: `perm[i] = encoded[i] XOR perm[i + 1]`. This allows us to recover each element of the permutation from `perm[n-1]` towards `perm[0]`.

By decoding the XOR in this way, we can find out the unique permutation `perm` that was encoded to give the `encoded` array.

Solution Approach

The provided solution follows the intuition and uses the XOR property effectively to decode the original permutation. Here's a step-by-step breakdown of the implementation referencing the python code provided:

1. First, the length `n` of the original permutation array `perm` is identified by adding 1 to the length of the `encoded` array, since `encoded` has `n-1` elements.
2. Two variables `a` and `b` are initialized to `0`. Variable `a` will hold the result of XOR of all elements at even indices of the `encoded` array. Variable `b` will hold the XOR of all integers from `1` to `n`.

```
for i in range(0, n - 1, 2):
    a ^= encoded[i]
for i in range(1, n + 1):
    b ^= i
```

3. A list `perm` of size `n` is created and initialized with zeros. The last element of `perm` is filled with `perm[0]` which is found out by taking `a XOR b`. As concluded in the intuition step, `a XOR b` gives the first element of the original permutation array since `b` is the XOR of the entire range and `a` contains XOR of elements at even positions in the original array (which leaves only the first element, since n is odd).

```
perm[-1] = a ^ b
```

4. Now that we have the first element, the rest of the permutation elements are retrieved by iterating backwards from the second last element to the first element of `perm` using the fact that `encoded[i] XOR perm[i + 1]` yields `perm[i]`.

```
for i in range(n - 2, -1, -1):
    perm[i] = encoded[i] ^ perm[i + 1]
```

5. Finally, the `perm` list is returned, which holds the decoded permutation.

In terms of data structures used, the solution uses a single list `perm` to hold the decoded permutation. The provided implementation efficiently employs the properties of XOR with simple iteration and list manipulation, avoiding the use of any complex data structures or algorithms. The space complexity is O(n) for storing the result and time complexity O(n) for the decoding, making the algorithm quite efficient.

Example Walkthrough

Let's illustrate the solution approach with a small example using an `encoded` array of `[3,6,1]`, which implies `n = 4`. Here's a step-by-step breakdown:

1. Determine the length `n` of the original permutation `perm`. Since the `encoded` array has 3 elements, `n` would be `3 + 1` which is `4`.
2. Initialize two variables `a` and `b` to `0`. `a` will store the XOR of `encoded` elements at even indices (0-based), and `b` will store the XOR of all integers from `1` to `n` (inclusive).
3. Compute the value of `a` by taking the XOR of `encoded` elements at even indices:

```
a = encoded[0] XOR encoded[2]
a = 3 XOR 1
a = 2
```

4. Compute the value of `b` by taking the XOR of all integers from `1` to `n`:

```
b = 1 XOR 2 XOR 3 XOR 4
b = 4
```

5. Create a list `perm` of size `n` with all zeros and calculate `perm[0]` using `a XOR b` because this will cancel out all values except for `perm[0]`:

```
perm[0] = a XOR b
perm[0] = 2 XOR 4
perm[0] = 6
```

6. Now with `perm[0]` known as `6`, backtrack to find the other values of original array `perm` using the property `perm[i] = encoded[i] XOR perm[i + 1]`:

```
perm[1] = encoded[0] XOR perm[0]
perm[1] = 3 XOR 6
perm[1] = 5

perm[2] = encoded[1] XOR perm[1]
perm[2] = 6 XOR 5
perm[2] = 3

perm[3] = encoded[2] XOR perm[2]
perm[3] = 1 XOR 3
perm[3] = 2
```

7. The resulting original permutation array `perm` is `[6,5,3,2]`.

After following the steps, the `perm` array obtained is the unique array that was transformed to `encoded`. The method uses simple XOR operations and leverages the properties of XOR to decode the array efficiently.

Python Solution

```python
class Solution:
    def decode(self, encoded):
        # Calculate the size of the original permutation
        size_of_permutation = len(encoded) + 1

        # Initialize variables to perform xor operations
        # `odd_xor` will hold the XOR of encoded elements at odd indices
        # `total_xor` will hold the XOR of all numbers from 1 to n
        odd_xor = total_xor = 0

        # XOR all encoded elements at odd indices (0-based)
        for index in range(0, size_of_permutation - 1, 2):
            odd_xor ^= encoded[index]

        # XOR all numbers from 1 to n to calculate the total_xor
        for num in range(1, size_of_permutation + 1):
            total_xor ^= num

        # Initialize the permutation list with zeros
        permutation = [0] * size_of_permutation

        # The last element of the permutation list can be found by XORing odd_xor and total_xor.
        # This is because the missing XOR'ed number is the first element of the original permutation.
        permutation[-1] = odd_xor ^ total_xor

        # Reconstruct the permutation starting from the end,
        # using the property that encoded[i] = permutation[i] XOR permutation[i+1]
        for index in range(size_of_permutation - 2, -1, -1):
            # To find permutation[i], we XOR encoded[i] with permutation[i+1]
            permutation[index] = encoded[index] ^ permutation[index + 1]

        # Return the resultant permutation list
        return permutation

```

Java Solution

```java
class Solution {
    public int[] decode(int[] encoded) {
        // Calculate the size of the original permutation array
        int n = encoded.length + 1;

        // Initialize 'xorEven' to perform XOR on even-indexed elements
        int xorEven = 0;

        // Initialize 'xorAll' to store the XOR of all numbers from 1 to n
        int xorAll = 0;

        // XOR even-indexed elements in the encoded array
        for (int i = 0; i < n - 1; i += 2) {
            xorEven ^= encoded[i];
        }

        // XOR all numbers from 1 to n to find the XOR of the entire permutation
        for (int i = 1; i <= n; ++i) {
            xorAll ^= i;
        }

        // Initialize the permutation array to be returned
        int[] permutation = new int[n];

        // Find the last element of the permutation by XORing 'xorEven' with 'xorAll', because
        // the XOR of all elements except the last one has been accounted for in 'xorEven'
        permutation[n - 1] = xorEven ^ xorAll;

        // Work backwards to fill in the rest of the permutation array by using the property
        // that encoded[i] = permutation[i] XOR permutation[i + 1]
        for (int i = n - 2; i >= 0; --i) {
            permutation[i] = encoded[i] ^ permutation[i + 1];
        }

        // Return the decoded permutation array
        return permutation;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    vector<int> decode(vector<int>& encoded) {
        // Determine the size of the original permutation
        int size = encoded.size() + 1;

        // Initialize two variables to perform XOR operations
        int oddXor = 0;      // Variable to store the XOR of encoded elements at odd indices
        int totalXor = 0;    // Variable to store the XOR of all elements in the original permutation

        // Perform XOR on all odd indexed elements of the encoded array
        for (int i = 0; i < size - 1; i += 2) {
            oddXor ^= encoded[i];
        }

        // Perform XOR on all numbers from 1 to n (size of the original permutation)
        for (int i = 1; i <= size; ++i) {
            totalXor ^= i;
        }

        // Create a vector to hold the original permutation
        vector<int> permutation(size);

        // Last element of the permutation can be found by XORing oddXor and totalXor
        permutation[size - 1] = oddXor ^ totalXor;

        // Reverse-XOR the encoded array starting from the end to compute the original permutation
        for (int i = size - 2; i >= 0; --i) {
            permutation[i] = encoded[i] ^ permutation[i + 1];
        }

        // Return the original permutation
        return permutation;
    }
};

```

Typescript Solution

```typescript
// Define the decode function, which decodes an encoded array to find the original permutation
function decode(encoded: number[]): number[] {
    // Determine the size of the original permutation
    const size: number = encoded.length + 1;

    // Initialize variables to perform XOR operations
    let oddXor: number = 0;      // Variable to store the XOR of encoded elements at odd indices
    let totalXor: number = 0;    // Variable to store the XOR of all elements in the original permutation

    // Perform XOR on all odd indexed elements of the encoded array
    for (let i = 0; i < size - 1; i += 2) {
        oddXor ^= encoded[i];
    }

    // Perform XOR on all numbers from 1 to n (size of the original permutation)
    for (let i = 1; i <= size; ++i) {
        totalXor ^= i;
    }

    // Create an array to hold the original permutation
    const permutation: number[] = new Array(size);

    // The last element of the permutation can be found by XORing oddXor and totalXor
    permutation[size - 1] = oddXor ^ totalXor;

    // Reverse-XOR the encoded array starting from the end to compute the original permutation
    for (let i = size - 2; i >= 0; --i) {
        permutation[i] = encoded[i] ^ permutation[i + 1];
    }

    // Return the original permutation
    return permutation;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given algorithm involves iterating over the encoded list and then iterating over a range of numbers from 1 to `n` to compute the XOR of all elements and the original permutation's elements. Here's the breakdown:

1. The first for loop runs from 0 to `n-1` with a step of 2, resulting in approximately `n/2` iterations.
2. The second for loop runs from 1 to `n`, inclusive, resulting in `n` iterations.
3. The last for loop reverses the encoded array while XORing each element with the next element of the `perm` list, resulting in `n-1` iterations.

Since `n`, `n/2`, and `n-1` are all linearly proportional to the length of the encoded list, the overall time complexity is `O(n)`.

Space Complexity

The space complexity is determined by:

1. Variables `a` and `b`, which are constant space and thus `O(1)`.
2. The `perm` list that stores the result, with a length equal to `n`, and running `n` iterations for decoding the permutation.

Since no additional space is used that grows with the input size apart from the `perm` list, the space complexity is `O(n)` due to the output data structure.