# 2433. Find The Original Array of Prefix Xor

Medium Bit Manipulation Array

## Problem Description

You are given an array of integers named `pref` that has a length of `n`. Your task is to find and return another integer array called `arr` that also has the size `n`. This `arr` array should meet a specific requirement: Each element at index `i` in the `pref` array must be equal to the cumulative bitwise XOR of elements from `0` to `i` in the `arr` array. The bitwise XOR operation is represented by the `^` symbol and combines bits where the result is `1` if only one of the bits compared is `1`, and `0` otherwise.

For example, if `pref[i]` equals `5`, it means that `arr[0] ^ arr[1] ^ ... ^ arr[i]` must also be equal to `5`.

It's important to note that there is only one unique solution to this problem. Your goal is to construct the `arr` array that will satisfy the given condition for each element in `pref`.

## Intuition

Now let's think about how to arrive at the solution. The key observation here is to recognize the properties of the XOR operation. One crucial property of XOR is that it is self-inverse, meaning that for any number `x`, we have `x ^ x = 0`. So, if we XOR a number with itself, we get 0.

Given the definition of `pref[i]` as `arr[0] ^ arr[1] ^ ... ^ arr[i]`, we can deduce `arr[i]` using `pref[i - 1]` and `pref[i]`. Because if we have `pref[i-1]` as `arr[0] ^ arr[1] ^ ... ^ arr[i-1]`, then `pref[i] = (arr[0] ^ arr[1] ^ ... ^ arr[i-1]) ^ arr[i]`. Now if we XOR `pref[i-1]` with `pref[i]`, we can isolate `arr[i]` because `(arr[0] ^ ... ^ arr[i-1]) ^ (arr[0] ^ ... ^ arr[i-1] ^ arr[i])` will cancel out all terms except `arr[i]`.

Hence, we start with `arr[0]` which is just `pref[0]` since `arr[0]` exclusively XORed with nothing is `arr[0]`. Then, for subsequent elements, we calculate `arr[i]` as `pref[i-1] ^ pref[i]`.

## Solution Approach

The solution involves a simple but clever use of properties of the XOR operation in combination with Python's built-in functions. Let's break down the implementation step-by-step to understand it better.

- First, is the initialization of the result array. The first element of `arr` can immediately be determined because `pref[0] = arr[0]`. Therefore, no calculation is needed for the first element.
- Then we use a list comprehension to build the rest of the `arr` array. List comprehensions are a concise way to create lists in Python, and in this case, it is used to iterate through pairs of elements from the input list `pref`.
- These pairs are generated using the `pairwise` utility, which groups every two adjacent items together. Since `pairwise` is not a default function in Python, it is assumed to be a hypothetical utility that would perform this action. A more accurate Python code to achieve the same without `pairwise` would look like this:

  ```
  [pref[i] ^ pref[i-1] for i in range(1, len(pref))]
  ```

  This code iterates over the `pref` array from the second element to the last, XORing each element `pref[i]` with its previous element `pref[i-1]`. The iteration starts from 1 because we are interested in pairs and the first element is already processed.
- Before executing the XOR operation, we prepare the list by adding a `0` to the beginning of the `pref` list. This is a critical step because it allows us to obtain the first element of `arr` (which is equal to `pref[0]`) without changing the algorithm. For subsequent elements, we use the XOR operation to derive them from the cumulative XOR values in `pref`.
- To demonstrate the XOR operation's effect, consider `pref` to be `[1, 2, 3]`. After prepending a `0`, it becomes `[0, 1, 2, 3]`. When we apply pairwise XOR as per the solution, we get `arr` as follows:

  ```
  arr[0] = 0 ^ 1 = 1
  arr[1] = 1 ^ 2 = 3
  arr[2] = 2 ^ 3 = 1
  ```

  Hence, `arr = [1, 3, 1]`.
- Finally, by iterating through these pairs, we XOR them to reconstruct each element of `arr` and derive the singular unique solution as proven by the properties of XOR.

In essence, the algorithm relies heavily on the understanding of how XOR functions as an associative and commutative operation, and on Python's ability to elegantly iterate through elements and perform operations using list comprehensions.

### Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose we are given the following `pref` array:

```
pref = [5, 1, 7]
```

Our goal is to construct an `arr` array such that `pref[i]` equals the cumulative XOR from `arr[0]` to `arr[i]`.

We start by setting `arr[0]` to `pref[0]` because there are no previous elements to XOR with, as described in the Intuition section. Thus:

```
arr[0] = pref[0] = 5
```

Next, we need to determine `arr[1]`. We use the fact that `pref[1]` is the XOR of `arr[0]` and `arr[1]`. Since `pref[1]` is `1` and `arr[0]` is `5`, we can isolate `arr[1]` as follows:

```
arr[1] = pref[0] ^ pref[1] = 5 ^ 1 = 4
```

We continue this process to find `arr[2]`:

```
arr[2] = pref[1] ^ pref[2] = 1 ^ 7 = 6
```

Now we have constructed the array `arr`:

```
arr = [5, 4, 6]
```

Let's verify that this `arr` satisfies the original problem conditions. To do this, we check the cumulative XOR for each index `i` in `arr`:

- For `i = 0`: `arr[0]` is `5`, which matches `pref[0]`.
- For `i = 1`: `arr[0] ^ arr[1]` equals `5 ^ 4`, which is `1`, matching `pref[1]`.
- For `i = 2`: `arr[0] ^ arr[1] ^ arr[2]` equals `5 ^ 4 ^ 6`, which is `7`, matching `pref[2]`.

As we can see, each `pref[i]` is equal to the cumulative XOR of elements from `0` to `i` in the constructed `arr`, validating our solution.

To summarize the steps:

1. Initialize the `arr` array by setting `arr[0]` to `pref[0]`.
2. Iterate through the `pref` array from index `1` to `n-1`, where `n` is the number of elements in `pref`.
3. Calculate `arr[i]` using the formula `arr[i] = pref[i-1] ^ pref[i]` for each `i`.
4. The resultant `arr` array will have the same size as `pref` and will meet the required condition.

By following this approach, we have successfully solved the problem by understanding the behavior of the XOR operation and applying it systematically.

## Solution Implementation

### Python

```python
# The following import is necessary to use the pairwise utility.
# It creates an iterator that returns consecutive pairs of elements from the input.
from itertools import pairwise

class Solution:
    def find_array(self, pref):
        # Initialize an array with a leading zero that will be used
        # for computing the original array based on the prefix XOR array.
        orig_array_with_zero = [0] + pref

        # Compute the original array by XORing each consecutive pair of elements.
        # This reverses the prefix XOR operation since a^a=0 and a^0=a.
        # The resulting array is the original array that was used to compute
        # the prefix XOR array.
        orig_array = [a ^ b for a, b in pairwise(orig_array_with_zero)]

        return orig_array

# An example usage:
# If pref = [1, 3, 5]
# The Solution().find_array(pref) will return the original array [1, 2, 7].
```

### Java

```java
class Solution {

    // Method to find the original array from its prefix XOR array
    public int[] findArray(int[] prefixXorArray) {
        // Number of elements in the prefix XOR array
        int length = prefixXorArray.length;

        // Initialize the array to store the original array elements
        int[] originalArray = new int[length];

        // The first element of the original array is the same as the first element of the prefix XOR array
        originalArray[0] = prefixXorArray[0];

        // Iterate through the prefix XOR array starting from the second element
        for (int i = 1; i < length; ++i) {
            // Each element of the original array is obtained by XORing the current and previous elements of the prefix XOR array
            originalArray[i] = prefixXorArray[i - 1] ^ prefixXorArray[i];
        }

        // Return the original array
        return originalArray;
    }
}
```

### C++

```cpp
#include <vector> // Required for using the std::vector

class Solution {
public:
    // Function to find the original array from its prefix XOR array.
    std::vector<int> findArray(std::vector<int>& prefixXor) {
        // The size of the prefix XOR array.
        int size = prefixXor.size();

        // Initialize the resultant array with the first element from prefixXor.
        // since the first element of both the original and prefix XOR arrays would be the same.
        std::vector<int> originalArray = {prefixXor[0]};

        // Iterating over the prefixXor array starting from the second element.
        for (int i = 1; i < size; ++i) {
            // The current original array element is the XOR of the previous and current elements
            // in the prefixXor array because the prefixXor[i] represents the XOR of all elements
            // in originalArray from 0 to i, so "undoing" the previous XOR (prefixXor[i-1]) will
            // give us the original value.
            originalArray.push_back(prefixXor[i - 1] ^ prefixXor[i]);
        }

        // Return the fully populated originalArray.
        return originalArray;
    }
};
```

### TypeScript

```typescript
function findArray(prefixArray: number[]): number[] {
    // Create a copy of the prefixArray to hold the answer.
    let answerArray = prefixArray.slice();

    // Iterate over the prefixArray starting from index 1, as the first element
    // doesn't change (ans[0] = pref[0] since XOR with 0 is a no-op).
    for (let i = 1; i < prefixArray.length; i++) {
        // XOR the current element with the previous element of prefixArray
        // and store the result in the answerArray, effectively computing the
        // original array before the prefix sums.
        answerArray[i] = prefixArray[i - 1] ^ prefixArray[i];
    }

    // Return the resultant array after reversing the prefix sum operation.
    return answerArray;
}

// Example usage:
// const result = findArray([1,3,2,3,3]); // Should return [1,2,0,1]
```

```python
# The following import is necessary to use the pairwise utility.
# It creates an iterator that returns consecutive pairs of elements from the input.
from itertools import pairwise

class Solution:
    def find_array(self, pref):
        # Initialize an array with a leading zero that will be used
        # for computing the original array based on the prefix XOR array.
        orig_array_with_zero = [0] + pref

        # Compute the original array by XORing each consecutive pair of elements.
        # This reverses the prefix XOR operation since a^a=0 and a^0=a.
        # The resulting array is the original array that was used to compute
        # the prefix XOR array.
        orig_array = [a ^ b for a, b in pairwise(orig_array_with_zero)]

        return orig_array

# An example usage:
# If pref = [1, 3, 5]
# The Solution().find_array(pref) will return the original array [1, 2, 7].
```

## Time and Space Complexity

The given code snippet implements a function to find an array from its prefix XOR array. The time and space complexities of the code can be analyzed as follows:

### Time Complexity

The time complexity is `O(n)`, where `n` is the length of the input `pref` array. This is because the function executes a single loop through the `pref` array plus an extra element (`0` appended to the front), performing a constant-time XOR operation for each pair of elements.

### Space Complexity

The space complexity of the function is `O(n)` as well, because it creates a new list with the same number of elements as the input list. This new list is populated with the results of the XOR operations between adjacent elements of the extended list `[0] + pref`.

In the case of the `pairwise` function, which is usually imported from the `itertools` module (not explicitly shown in the code), if it is implemented in such a way that it yields pairs of elements on-the-fly without creating a separate list or collection for them, the additional space overhead is `O(1)` (constant space). However, the resulting list still retains the time complexity of `O(n)` and space complexity of `O(n)` due to the reasons mentioned above.