2932. Maximum Strong Pair XOR I

Easy Bit Manipulation Trie Array Hash Table Sliding Window

Problem Description

You are provided with an array `nums` which contains integers. The task is to find two integers from this array that satisfy a specific condition defined as being a *strong* pair. The condition for a *strong* pair is that the absolute difference between `x` and `y` must be less than or equal to the smaller of the two numbers (`|x - y| <= min(x, y)`). The goal is to choose a pair that not only meets this condition but also has the highest possible bitwise XOR value compared to all other strong pairs that could be formed from the array elements. It's also worth noting that you can use the same integer from the array twice to create a pair. The output of the problem is the maximum XOR value obtained from all valid strong pairs in the array.

Intuition

The intuition behind the given solution is to employ a brute-force approach to solve the problem. Since the problem does not impose a strong restriction on the size of the input array, we can afford to use a double loop to enumerate all possible pairs of numbers from the array. For each pair `(x, y)`, we check if it meets the strong pair condition (`|x - y| <= min(x, y)`). If it does, we calculate the XOR of `x` and `y` (using the bitwise XOR operator `^`) and keep track of the maximum XOR value obtained. By the end of the iteration over all pairs, we will have the maximum XOR value for all strong pairs in the array.

In summary, the solution approach arrives by considering the following points:

1. A brute-force method can be used without significant concern for performance unless the input size is very large.
2. Checking the strong pair condition and calculating the XOR are both constant-time operations.
3. Keeping track of the maximum XOR value seen so far while iterating over pairs ensures we have the correct answer by the end of the iterations.

Solution Approach

The solution uses a straightforward enumeration algorithm to check every possible pair in the given integer array `nums`. This approach does not require any special data structures or advanced algorithms, relying on the Python list given as input and couple of nested loops to generate pairs.

Here's a step-by-step breakdown of the implementation:

1. A double `for` loop is constructed, where the first loop iterates over each element `x` in the array `nums`, and the second loop iterates over each element `y` in the array as well. This setup allows us to consider every possible pair `(x, y)` from `nums`.
2. For each pair `(x, y)`, we evaluate the condition `|x - y| <= min(x, y)`. To do this, we use Python's `abs()` function to find the absolute difference between `x` and `y`, and `min(x, y)` to find the smaller of the two numbers.
3. If the condition is satisfied, meaning the pair `(x, y)` is a *strong* pair, we calculate the bitwise XOR of `x` and `y` by using the `^` operator.
4. We employ Python's list comprehension combined with the `max()` function to iterate over all possible pairs, calculate their XOR values, and keep the maximum of these values. The `max()` function is called with a generator expression that yields the XOR of `x` and `y` for each strong pair.

The code snippet provided by the `Solution` class method, `maximumStrongPairXor(self, nums: List[int]) -> int`, returns the single highest XOR value discovered during the enumeration process.

The overall complexity of this approach is O(n^2), where n is the length of the array `nums`, since the enumeration involves checking each possible pair within the array.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the array `nums = [3, 10, 5, 25, 2, 8]`.

1. Initialize a variable named `max_xor` that will store the maximum XOR value found. Initially, this can be set to a very low value, such as zero.
2. Start with the first element in `nums`, which is `3`. We then check it against every other element including itself:
   - Check `3` with `3`: They are the same, so `|3 - 3| = 0` which is less than `min(3, 3)`. The XOR is `3 ^ 3 = 0`. `max_xor` remains `0`.
   - Check `3` with `10`: The difference `|3 - 10| = 7` which is not less than `min(3, 10) = 3`. This pair is not strong, so we move on.
   - Check `3` with `5`: The difference `|3 - 5| = 2` which is less than `min(3, 5) = 3`. The XOR is `3 ^ 5 = 6`. `max_xor` is updated to `6`.
   - Check `3` with `25`, `2`, and `8` the same way, updating `max_xor` if we find a higher XOR from a strong pair.
3. Move to the next element in `nums`, `10`, and repeat the process for each pair:
   - Check `10` with itself and then `5`, `25`, `2`, `8`. If we find any strong pairs, we evaluate the XOR and check if it's higher than `max_xor` and update accordingly.

   For example, when checking against `5`, since `|10 - 5| = 5` is equal to `min(10, 5) = 5`, the pair `(10, 5)` is considered strong. The XOR of `10` and `5` is `10 ^ 5 = 15`, which is higher than the current `max_xor` of `6`, so `max_xor` is updated to `15`.
4. Continue through all the elements of `nums`, comparing each with every other element, checking the strong pair condition, and keeping the maximum XOR value found.

At the end of the process, `max_xor` will hold the highest XOR value possible from all strong pairs. In this example, the maximum XOR value is found to be `28`, which is the XOR of the pair `(5, 25)` where `|5 - 25| = 20` is less than `min(5, 25) = 5`.

Thus, the `maximumStrongPairXor` function would return `28` for the input array `[3, 10, 5, 25, 2, 8]`.

Solution Implementation

Python

```python
from typing import List  # We import List from typing to annotate the type of the nums parameter.

class Solution:
    def maximum_strong_pair_xor(self, nums: List[int]) -> int:
        # Initialize variable to store the maximum XOR value found.
        max_xor = 0

        # Iterate over each possible pair in the list.
        for i in range(len(nums)):
            for j in range(len(nums)):
                # Calculate the absolute difference between the two numbers.
                difference = abs(nums[i] - nums[j])

                # Calculate the minimum of the two numbers.
                minimum = min(nums[i], nums[j])

                # Check if the difference between the two numbers
                # is less than or equal to the minimum of the two.
                if difference <= minimum:
                    # Calculate XOR of the current pair and update the max_xor
                    # if it's greater than the current maximum.
                    possible_max_xor = nums[i] ^ nums[j]
                    max_xor = max(max_xor, possible_max_xor)

        # Return the maximum XOR value found among all the valid pairs.
        return max_xor
```

Java

```java
class Solution {
    public int maximumStrongPairXor(int[] nums) {
        // Initialize the variable to store the maximum XOR value found.
        // It is set to zero since XOR of any number with 0 is the number itself,
        // and we're trying to find the maximum of all XOR operations.
        int maxPairXor = 0;

        // Traverse all possible pairs in the array
        for (int i = 0; i < nums.length; i++) {     // Iterate through each element in nums with index i
            for (int j = 0; j < nums.length; j++) {  // Iterate through each element in nums with index j
                // Check the condition that the absolute difference between the two numbers
                // should be less than or equal to the smaller of the two numbers.
                if (Math.abs(nums[i] - nums[j]) <= Math.min(nums[i], nums[j])) {
                    // If the condition is met, calculate the XOR of the current pair
                    int currentXor = nums[i] ^ nums[j];
                    // Update the maximum XOR value if the current XOR is greater than the current maximum
                    maxPairXor = Math.max(maxPairXor, currentXor);
                }
            }
        }

        // Return the maximum XOR value found
        return maxPairXor;
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // for std::max

class Solution {
public:
    // Defines a function that calculates the maximum XOR value of any strong pair in the array.
    // A strong pair is defined as a pair of numbers (x, y) where abs(x - y) is less than or
    // equal to the minimum of x and y.
    int maximumStrongPairXor(std::vector<int>& nums) {
        int max_xor = 0; // Initialize the maximum XOR value to zero.

        // Iterate through all possible pairs of numbers within the nums array.
        for (int x : nums) {
            for (int y : nums) {
                // Check if x and y form a strong pair as per the given condition.
                if (abs(x - y) <= std::min(x, y)) {
                    // Update max_xor to hold the maximum value between the current max_xor
                    // and the XOR of x and y.
                    max_xor = std::max(max_xor, x ^ y);
                }
            }
        }

        // Return the calculated maximum XOR value.
        return max_xor;
    }
};
```

TypeScript

```typescript
/**
 * Computes the maximum XOR value of a strong pair from the array.
 * A strong pair (x, y) satisfies the condition: abs(x - y) <= min(x, y).
 * @param {number[]} nums - The array of numbers to evaluate.
 * @return {number} The maximum XOR value of any strong pair in the array.
 */
function maximumStrongPairXor(nums: number[]): number {
    let maximumXor = 0; // Holds the maximum XOR value found.

    // Iterate through each number in the array.
    for (const num1 of nums) {
        // Iterate through each number in the array to find all possible pairs.
        for (const num2 of nums) {
            // Check if the current pair (num1, num2) is a strong pair.
            if (Math.abs(num1 - num2) <= Math.min(num1, num2)) {
                // Update maximumXor if XOR of the current pair is greater than the current maximumXor.
                maximumXor = Math.max(maximumXor, num1 ^ num2);
            }
        }
    }

    // Return the maximum XOR value found among all strong pairs.
    return maximumXor;
}
```

```python
from typing import List  # We import List from typing to annotate the type of the nums parameter.

class Solution:
    def maximum_strong_pair_xor(self, nums: List[int]) -> int:
        # Initialize variable to store the maximum XOR value found.
        max_xor = 0

        # Iterate over each possible pair in the list.
        for i in range(len(nums)):
            for j in range(len(nums)):
                # Calculate the absolute difference between the two numbers.
                difference = abs(nums[i] - nums[j])

                # Calculate the minimum of the two numbers.
                minimum = min(nums[i], nums[j])

                # Check if the difference between the two numbers
                # is less than or equal to the minimum of the two.
                if difference <= minimum:
                    # Calculate XOR of the current pair and update the max_xor
                    # if it's greater than the current maximum.
                    possible_max_xor = nums[i] ^ nums[j]
                    max_xor = max(max_xor, possible_max_xor)

        # Return the maximum XOR value found among all the valid pairs.
        return max_xor
```

Time and Space Complexity

The time complexity of the provided code is `O(n^2)` where `n` is the length of the input array `nums`. This quadratic time complexity arises because there are two nested loops, with each element in `nums` being paired with every other element.

For every element `x` in `nums`, the code iterates through the entire `nums` array again to find another element `y`, and then it calculates `x ^ y` if the condition `abs(x - y) <= min(x, y)` is satisfied. There are `n` choices for `x` and for each `x`, there are `n` choices for `y`, resulting in `n * n` pairs to check, thus the `O(n^2)` complexity.

The space complexity of the provided code is `O(1)`. This constant space complexity is because the algorithm only uses a fixed amount of additional space that does not depend on the input size. All operations are performed in place and the max function keeps track of the current maximum without requiring additional space proportional to the size of the input `nums`.