Here is the complete documentation for the **Bitwise Greedy Approach using HashSet** to find the **Maximum XOR of Two Numbers in an Array**.

Maximum XOR of Two Numbers in an Array

Table of Contents

1. Problem Statement

Given an integer array nums, return the **maximum result** of nums[i] XOR nums[j] where $0 \leq i < j < n$.

Constraints

- $1 \leq nums.length \leq 2 \times 10^5$
- $0 \leq nums[i] \leq 2^{31} - 1$

Example 1

Input

nums = [3,10,5,25,2,8]

Output

28

Explanation

- The maximum XOR is obtained from 5 XOR 25 = 28.

Example 2

Input

nums = [14,70,53,83,49,91,36,80,92,51,66,70]

Output

127

2. Intuition

- The XOR operation is **maximized when two numbers have the most different bits**.
- Instead of checking every pair (which is **$O(N^2)$**), we can efficiently find the best pair using a **bitwise greedy approach**.
- We utilize **hash sets** to store prefixes and iteratively construct the maximum XOR.

3. Key Observations

1. **Binary Representation:**
    - Each number is **at most 31 bits** (since $0 \leq nums[i] \leq 2^{31} - 1$).
2. **XOR Property:**
    - If $a \oplus b = c$, then $a = b \oplus c$ or $b = a \oplus c$.

3. **Prefix Matching:**
   - We maintain **prefixes of numbers up to i bits**.
   - If a **target XOR can be formed** using prefixes, update the result.

4. Approach

1. **Iterate from MSB to LSB:**
   - Start checking from the **31st bit to the 0th bit**.
2. **Store Prefixes:**
   - Keep track of the **prefixes of all numbers** (leftmost i bits).
3. **Check for Maximum XOR:**
   - Assume the **current bit is 1** in the maximum XOR.
   - Verify if any two prefixes can achieve this XOR.

5. Edge Cases

- **Single Element:** nums = [0] → Output 0 (XOR is undefined, return default).
- **All Elements are Same:** nums = [7,7,7,7] → Output 0.
- **Power of Two Elements:** nums = [1,2,4,8,16] → Should handle different bit positions.
- **Large Input:** nums = [random integers up to 2^{31} - 1] → Must run efficiently for 10^5 elements.

6. Complexity Analysis

Time Complexity

- **Outer Loop:** Iterates **32 times** (constant).
- **Inner Loop:** Iterates **O(N)** (to store prefixes).
- **Lookup in HashSet: O(1)** on average.
- **Total Complexity:** O(32×N)=O(N)O(32 \times N) = O(N)

Space Complexity

- **HashSet stores prefixes of N numbers: O(N)**
- **Total Space Complexity: O(N)**

7. Alternative Approaches

1. Brute Force ($O(N^2)$)

- Compute nums[i] XOR nums[j] for every pair.
- **Time Complexity: $O(N^2)$ → Too slow for large inputs.**

2. Trie-based Approach (O(N log M))

- Insert numbers as binary strings in a **Trie**.
- Query each number to find the best XOR pair.
- **Time Complexity: O(N log M)** (~O(N × 32) = O(N)).
- **Space Complexity: O(N log M)**.

Why is the HashSet Approach better?

✅ **Faster** than Trie-based (no tree traversal).

✅ **Lower space usage**.

✅ **Efficient HashSet lookups (O(1))**.

8. Test Cases

```
sol = Solution()
# Test Case 1
print(sol.findMaximumXOR([3,10,5,25,2,8]))  # Output: 28
# Test Case 2
print(sol.findMaximumXOR([14,70,53,83,49,91,36,80,92,51,66,70]))  # Output: 127
# Test Case 3 (Single Element)
print(sol.findMaximumXOR([0]))  # Output: 0
# Test Case 4 (All Identical)
print(sol.findMaximumXOR([7,7,7,7]))  # Output: 0
# Test Case 5 (Power of Two)
print(sol.findMaximumXOR([1,2,4,8,16]))  # Output: 24
```

9. Final Thoughts

- The **Bitwise Greedy + HashSet Approach** is the **most efficient**.
- **Avoids** Trie's extra space & complexity.
- **Scales well** for large inputs (10^5 elements).
- **Handles edge cases efficiently**.

🚀 **This approach ensures the best performance while keeping the code clean and optimized!** 🚀