explain xor in details in context of cp/dsa
everything in detail
every algortihm
cases
use case
from basic to advance give all codes in cpp
from basic to advnace all agorithm and imp code should be covered
all concepts
give theory -> proof then code in cpp
in detail every possible algo should cover please

XOR in Competitive Programming & Data Structures

Table of Contents

---

1. Introduction to XOR

Definition

XOR (Exclusive OR) is a binary operation that returns true (1) when the number of true inputs is odd.

Truth Table:

A	B	A ⊕ B
0	0	0

A	B	A ⊕ B
0	1	1
1	0	1
1	1	0

Mathematical Representation

For bits: `a ⊕ b = (a + b) mod 2`

For integers: Bitwise XOR of binary representations

```cpp
// Basic XOR operation
int a = 5;    // 101
int b = 3;    // 011
int c = a ^ b; // 110 = 6
```

---

2. Properties & Proofs

2.1 Fundamental Properties

Property 1: Commutative

```text
a ⊕ b = b ⊕ a
```

Proof: Follows from definition - order doesn't affect odd count.

Property 2: Associative

```text
(a ⊕ b) ⊕ c = a ⊕ (b ⊕ c)
```

Proof: Both sides equal 1 when odd number of inputs are 1.

Property 3: Identity Element

```text
a ⊕ 0 = a
```

Proof: 0 doesn't change the odd/even count.

Property 4: Self-Inverse

```text
a ⊕ a = 0
```

Proof: Even number (2) of same bits → always 0.

Property 5: Inverse

```text
a ⊕ b = c  ⇔  a = b ⊕ c
```

Proof: XOR both sides by b: `a ⊕ b ⊕ b = c ⊕ b` → `a = c ⊕ b`

2.2 Important Corollaries

Corollary 1: Cancellation Law

```text
a ⊕ b = a ⊕ c  ⇒  b = c
```

Proof: XOR both sides with a.

Corollary 2: Even-Odd Property

XOR of all bits = 1 if odd number of 1's, 0 otherwise.

Corollary 3: Prefix XOR

For array arr, define prefix_xor[i] = arr[0] ⊕ arr[1] ⊕ ... ⊕ arr[i]
Then arr[i] ⊕ arr[i+1] ⊕ ... ⊕ arr[j] = prefix_xor[j] ⊕ prefix_xor[i-1]

3. Basic Operations

3.1 Swapping without temporary variable

cpp

```cpp
void swap(int &a, int &b) {
    a = a ^ b;
    b = a ^ b;  // b = (a ^ b) ^ b = a
    a = a ^ b;  // a = (a ^ b) ^ a = b
}
```

3.2 Checking if two numbers are equal

cpp

```cpp
bool areEqual(int a, int b) {
    return (a ^ b) == 0;
}
```

3.3 Toggling bits

cpp

```cpp
// Toggle k-th bit (0-indexed)
int toggleBit(int num, int k) {
    return num ^ (1 << k);
}
```

3.4 Finding missing number

cpp

```cpp
// Find missing number in 0..n
int findMissing(vector<int>& nums) {
    int n = nums.size();
    int xor_all = 0;
    for(int i = 0; i <= n; i++) xor_all ^= i;
    for(int num : nums) xor_all ^= num;
    return xor_all;
}
```

4. Common Algorithms

4.1 Find Non-Repeating Element

Problem: Find element that appears once, others appear twice

```cpp
int singleNumber(vector<int>& nums) {
    int result = 0;
    for(int num : nums) {
        result ^= num;
    }
    return result;
}
```

Proof:

- All pairs cancel out (a ⊕ a = 0)
- Remaining is the single element

4.2 Find Two Non-Repeating Elements

Problem: Array has two unique elements, others appear twice

```cpp
vector<int> singleNumberIII(vector<int>& nums) {
    // Step 1: Get XOR of two unique numbers
    int xor_all = 0;
    for(int num : nums) xor_all ^= num;

    // Step 2: Find rightmost set bit
    int rightmost_set_bit = xor_all & -xor_all;

    // Step 3: Partition and XOR
    int num1 = 0, num2 = 0;
    for(int num : nums) {
        if(num & rightmost_set_bit) {
            num1 ^= num;
        } else {
            num2 ^= num;
        }
    }

    return {num1, num2};
}
```

Proof:

- xor_all = x ⊕ y ≠ 0 (since x ≠ y)
- Rightmost set bit means x and y differ at that bit
- Partitioning separates x and y

4.3 Find Missing and Duplicate

Problem: Array of size n contains 1..n with one missing, one duplicate

```cpp
pair<int, int> findMissingAndDuplicate(vector<int>& nums) {
    int n = nums.size();
    int xor_all = 0;

    // XOR of all numbers and indices
    for(int i = 0; i < n; i++) {
        xor_all ^= (i + 1);
        xor_all ^= nums[i];
    }

    // xor_all = missing ⊕ duplicate
    int rightmost_bit = xor_all & -xor_all;

    int bucket1 = 0, bucket2 = 0;

    // Partition numbers 1..n
    for(int i = 1; i <= n; i++) {
        if(i & rightmost_bit) bucket1 ^= i;
        else bucket2 ^= i;
    }

    // Partition array elements
    for(int num : nums) {
        if(num & rightmost_bit) bucket1 ^= num;
        else bucket2 ^= num;
    }

    // Determine which is missing and which is duplicate
    for(int num : nums) {
        if(num == bucket1) {
            return {bucket2, bucket1}; // missing, duplicate
        }
    }
    return {bucket1, bucket2};
}
```

4.4 XOR Linked List

```cpp
class XORNode {
public:
    int data;
    XORNode* npx; // XOR of next and prev pointers

    XORNode(int val) : data(val), npx(nullptr) {}
};

class XORLinkedList {
private:
    XORNode* head;

    XORNode* XOR(XORNode* a, XORNode* b) {
        return (XORNode*)((uintptr_t)a ^ (uintptr_t)b);
    }

public:
    XORLinkedList() : head(nullptr) {}

    void insert(int data) {
        XORNode* new_node = new XORNode(data);
        new_node->npx = head;

        if(head) {
            head->npx = XOR(new_node, head->npx);
        }
        head = new_node;
    }

    void traverse() {
        XORNode* curr = head;
        XORNode* prev = nullptr;
        XORNode* next;
        while(curr) {
            cout << curr->data << " ";
            next = XOR(prev, curr->npx);
            prev = curr;
            curr = next;
        }
        cout << endl;
    }
};
```

---

5. Advanced Concepts

5.1 XOR Basis (Gaussian Elimination for XOR)

Linear Basis for XOR Space

```cpp
class XORBasis {
private:
    vector<int> basis;
    int size;

public:
    XORBasis(int n = 32) : basis(n, 0), size(0) {}

    // Insert a number into basis
    void insert(int x) {
        for(int i = basis.size() - 1; i >= 0; i--) {
            if((x >> i) & 1) {
                if(!basis[i]) {
                    basis[i] = x;
                    size++;
                    return;
                }
                x ^= basis[i];
            }
        }
    }

    // Check if number can be represented
    bool canRepresent(int x) {
        for(int i = basis.size() - 1; i >= 0; i--) {
            if((x >> i) & 1) {
                if(!basis[i]) return false;
                x ^= basis[i];
            }
        }
        return true;
    }

    // Get maximum XOR subset
    int getMax() {
        int res = 0;
        for(int i = basis.size() - 1; i >= 0; i--) {
            if(!basis[i]) continue;
            if((res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }

    // Get k-th smallest XOR value
    int getKthSmallest(long long k) {
        // Reconstruct basis to reduced row echelon form
        vector<int> reduced_basis;
        for(int i = 0; i < basis.size(); i++) {
            if(basis[i]) {
                for(int j = i + 1; j < basis.size(); j++) {
                    if(basis[j] & (1 << i)) {
                        basis[j] ^= basis[i];
                    }
                }
                reduced_basis.push_back(basis[i]);
            }
        }

        int res = 0;
        for(int i = 0; i < reduced_basis.size(); i++) {
            if(k & (1LL << i)) {
                res ^= reduced_basis[i];
            }
        }
        return res;
    }

    int getSize() { return size; }
};
```

5.2 Nim Game and Grundy Numbers

Nim Game Solution

```cpp
bool canWinNim(vector<int>& piles) {
    int xor_sum = 0;
    for(int stones : piles) {
        xor_sum ^= stones;
    }
    return xor_sum != 0; // First player wins if XOR != 0
}
```

Proof (Bouton's Theorem):

- Terminal position: all piles empty (XOR = 0) → losing position
- From XOR ≠ 0, can move to XOR = 0
- From XOR = 0, any move leads to XOR ≠ 0

Misère Nim

```cpp
bool canWinMisereNim(vector<int>& piles) {
    int xor_sum = 0;
    bool all_ones = true;

    for(int stones : piles) {
        xor_sum ^= stones;
        if(stones > 1) all_ones = false;
    }

    if(all_ones) {
        return (piles.size() % 2 == 0);
    }
    return xor_sum != 0;
}
```

5.3 Maximum XOR of Two Numbers

```cpp
```

```
class TrieNode {
public:
    TrieNode* children[2];
    TrieNode() {
        children[0] = children[1] = nullptr;
    }
};

class BinaryTrie {
private:
    TrieNode* root;

public:
    BinaryTrie() {
        root = new TrieNode();
    }

    void insert(int num) {
        TrieNode* node = root;
        for(int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if(!node->children[bit]) {
                node->children[bit] = new TrieNode();
            }
            node = node->children[bit];
        }
    }

    int findMaxXOR(int num) {
        TrieNode* node = root;
        int max_xor = 0;

        for(int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            int opposite_bit = 1 - bit;

            if(node->children[opposite_bit]) {
                max_xor |= (1 << i);
                node = node->children[opposite_bit];
            } else {
                node = node->children[bit];
            }
        }
        return max_xor;
    }
};
```

```cpp
int findMaximumXOR(vector<int>& nums) {
    BinaryTrie trie;
    int max_xor = 0;

    for(int num : nums) {
        trie.insert(num);
        max_xor = max(max_xor, trie.findMaxXOR(num));
    }

    return max_xor;
}
```

5.4 XOR Convolution (Walsh-Hadamard Transform)

cpp

```cpp
void walshHadamardTransform(vector<int>& a, bool inverse) {
    int n = a.size();
    for(int len = 1; 2 * len <= n; len <<= 1) {
        for(int i = 0; i < n; i += 2 * len) {
            for(int j = 0; j < len; j++) {
                int u = a[i + j];
                int v = a[i + j + len];
                a[i + j] = u + v;
                a[i + j + len] = u - v;
            }
        }
    }

    if(inverse) {
        for(int i = 0; i < n; i++) {
            a[i] /= n;
        }
    }
}

vector<int> xorConvolution(vector<int>& a, vector<int>& b) {
    int n = a.size();
    walshHadamardTransform(a, false);
    walshHadamardTransform(b, false);

    vector<int> c(n);
    for(int i = 0; i < n; i++) {
        c[i] = a[i] * b[i];
    }
    walshHadamardTransform(c, true);
    return c;
}
```

5.5 XOR Segment Tree

cpp

```cpp
class XORSegmentTree {
private:
    vector<int> tree;
    int n;

    void build(vector<int>& arr, int node, int start, int end) {
        if(start == end) {
            tree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            build(arr, 2*node, start, mid);
            build(arr, 2*node + 1, mid + 1, end);
            tree[node] = tree[2*node] ^ tree[2*node + 1];
        }
    }

    void update(int node, int start, int end, int idx, int val) {
        if(start == end) {
            tree[node] = val;
        } else {
            int mid = (start + end) / 2;
            if(idx <= mid) {
                update(2*node, start, mid, idx, val);
            } else {
                update(2*node + 1, mid + 1, end, idx, val);
            }
            tree[node] = tree[2*node] ^ tree[2*node + 1];
        }
    }

    int query(int node, int start, int end, int l, int r) {
        if(r < start || l > end) return 0;
        if(l <= start && end <= r) return tree[node];

        int mid = (start + end) / 2;
        int left_xor = query(2*node, start, mid, l, r);
        int right_xor = query(2*node + 1, mid + 1, end, l, r);
        return left_xor ^ right_xor;
    }

public:
    XORSegmentTree(vector<int>& arr) {
        n = arr.size();
        tree.resize(4 * n);
        build(arr, 1, 0, n-1);
    }

    void update(int idx, int val) {
        update(1, 0, n-1, idx, val);
    }

    int query(int l, int r) {
        return query(1, 0, n-1, l, r);
    }
};
```

5.6 XOR Persistent Segment Tree

```cpp
class XorPersistentNode {
public:
    int value;
    XorPersistentNode* left;
    XorPersistentNode* right;

    XorPersistentNode(int val = 0) : value(val), left(nullptr), right(nullptr) {}
};

class XorPersistentSegmentTree {
private:
    vector<XorPersistentNode*> roots;
    int n;

    XorPersistentNode* build(int start, int end) {
        XorPersistentNode* node = new XorPersistentNode();
        if(start != end) {
            int mid = (start + end) / 2;
            node->left = build(start, mid);
            node->right = build(mid + 1, end);
        }
        return node;
    }
    XorPersistentNode* update(XorPersistentNode* prev, int start, int end, int idx,
int val) {
        XorPersistentNode* node = new XorPersistentNode();

        if(start == end) {
            node->value = prev->value ^ val;
        } else {
            int mid = (start + end) / 2;
            if(idx <= mid) {
                node->left = update(prev->left, start, mid, idx, val);
                node->right = prev->right;
            } else {
                node->left = prev->left;
                node->right = update(prev->right, mid + 1, end, idx, val);
            }
            node->value = node->left->value ^ node->right->value;
        }
        return node;
    }

    int query(XorPersistentNode* node, int start, int end, int l, int r) {
        if(!node || r < start || l > end) return 0;
        if(l <= start && end <= r) return node->value;

        int mid = (start + end) / 2;
        int left_xor = query(node->left, start, mid, l, r);
        int right_xor = query(node->right, mid + 1, end, l, r);
        return left_xor ^ right_xor;
    }

public:
    XorPersistentSegmentTree(int size) : n(size) {
        roots.push_back(build(0, n-1));
    }

    void update(int version, int idx, int val) {
        roots.push_back(update(roots[version], 0, n-1, idx, val));
    }

    int query(int version, int l, int r) {
        return query(roots[version], 0, n-1, l, r);
    }

    int getLatestVersion() {
        return roots.size() - 1;
    }
};
```

5.7 XOR Game Theory - Grundy Numbers

```cpp
// Calculate Grundy number for a pile
int calculateGrundy(int stones) {
    // For subtraction game {1, 3, 4}
    if(stones == 0) return 0;

    unordered_set<int> moves;
    moves.insert(calculateGrundy(stones - 1));
    if(stones >= 3) moves.insert(calculateGrundy(stones - 3));
    if(stones >= 4) moves.insert(calculateGrundy(stones - 4));

    // Mex (minimum excludant)
    int mex = 0;
    while(moves.count(mex)) mex++;
    return mex;
}

// XOR of Grundy numbers for multiple piles
bool canWinGame(vector<int>& piles) {
    int xor_sum = 0;
    for(int stones : piles) {
        xor_sum ^= calculateGrundy(stones);
    }
    return xor_sum != 0;
}
```

---

6. Problem Patterns

Pattern 1: Subarray XOR Problems

Count subarrays with XOR = k

```cpp
int countSubarraysWithXOR(vector<int>& nums, int k) {
    unordered_map<int, int> prefix_xor_count;
    prefix_xor_count[0] = 1;

    int xor_sum = 0, count = 0;
    for(int num : nums) {
        xor_sum ^= num;

        // We need: xor_sum ^ (some previous_xor) = k
        // => previous_xor = xor_sum ^ k
        if(prefix_xor_count.count(xor_sum ^ k)) {
            count += prefix_xor_count[xor_sum ^ k];
        }

        prefix_xor_count[xor_sum]++;
    }
    return count;
}
```

Maximum XOR subarray

```cpp
int maxSubarrayXOR(vector<int>& nums) {
    int n = nums.size();
    int max_xor = INT_MIN;

    // O(n^2) approach
    for(int i = 0; i < n; i++) {
        int curr_xor = 0;
        for(int j = i; j < n; j++) {
            curr_xor ^= nums[j];
            max_xor = max(max_xor, curr_xor);
        }
    }
    return max_xor;
}

// Optimized using Trie - O(n*log(max_value))
int maxSubarrayXOROptimized(vector<int>& nums) {
    BinaryTrie trie;
    trie.insert(0);

    int prefix_xor = 0, max_xor = 0;
    for(int num : nums) {
        prefix_xor ^= num;
        max_xor = max(max_xor, trie.findMaxXOR(prefix_xor));
        trie.insert(prefix_xor);
    }
    return max_xor;
}
```

Pattern 2: XOR in Bit Manipulation

Find position of rightmost set bit

```cpp
int rightmostSetBit(int n) {
    return n & -n;
}
```

Check if number is power of 2 using XOR

```cpp
bool isPowerOfTwo(int n) {
    if(n <= 0) return false;
    return (n & (n - 1)) == 0;
}
```

Swap adjacent bits

```cpp
int swapAdjacentBits(int n) {
    // Mask for even bits: 0xAAAAAAAA
    // Mask for odd bits: 0x55555555
    int even_bits = n & 0xAAAAAAAA;
    int odd_bits = n & 0x55555555;

    even_bits >>= 1;
    odd_bits <<= 1;

    return even_bits | odd_bits;
}
```

Pattern 3: XOR in Matrix Problems

XOR of all elements in matrix sub-rectangle

```cpp
```

```
class XORMatrix {
private:
    vector<vector<int>> prefix;
    int rows, cols;

public:
    XORMatrix(vector<vector<int>>& matrix) {
        rows = matrix.size();
        cols = matrix[0].size();
        prefix.resize(rows + 1, vector<int>(cols + 1, 0));

        for(int i = 1; i <= rows; i++) {
            for(int j = 1; j <= cols; j++) {
                prefix[i][j] = matrix[i-1][j-1] ^
                               prefix[i-1][j] ^
                               prefix[i][j-1] ^
                               prefix[i-1][j-1];
            }
        }
    }

    int query(int r1, int c1, int r2, int c2) {
        return prefix[r2+1][c2+1] ^
               prefix[r1][c2+1] ^
               prefix[r2+1][c1] ^
               prefix[r1][c1];
    }
};
```

Pattern 4: XOR in Graphs

XOR Minimum Spanning Tree

```cpp
class DSU {
private:
    vector<int> parent, rank;

public:
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for(int i = 0; i < n; i++) parent[i] = i;
    }
    int find(int x) {
        if(parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if(rootX != rootY) {
            if(rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if(rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
};

int xorMST(vector<vector<pair<int, int>>>& graph) {
    int n = graph.size();
    vector<tuple<int, int, int>> edges;

    // Create edges with XOR weights
    for(int u = 0; u < n; u++) {
        for(auto& [v, w] : graph[u]) {
            if(u < v) {
                edges.emplace_back(w, u, v);
            }
        }
    }

    // Sort edges by weight
    sort(edges.begin(), edges.end());

    DSU dsu(n);
    int mst_weight = 0;

    for(auto& [w, u, v] : edges) {
        if(dsu.find(u) != dsu.find(v)) {
            dsu.unite(u, v);
            mst_weight ^= w;
        }
    }

    return mst_weight;
}
```

7. Complete Code Library

7.1 Header File with All XOR Functions

```cpp
#ifndef XOR_UTILS_H
#define XOR_UTILS_H

#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>
#include <climits>
#include <iostream>
#include <memory>

using namespace std;

namespace XORUtils {

    // Basic operations
    int swapNumbers(int &a, int &b) {
        a = a ^ b;
        b = a ^ b;
        a = a ^ b;
    }

    bool isPowerOfTwo(int n) {
        return n > 0 && (n & (n - 1)) == 0;
    }

    int toggleBit(int num, int pos) {
        return num ^ (1 << pos);
    }
```

```
// Single number problems
int singleNumber(vector<int>& nums) {
    int result = 0;
    for(int num : nums) result ^= num;
    return result;
}

vector<int> twoSingleNumbers(vector<int>& nums) {
    int xor_all = 0;
    for(int num : nums) xor_all ^= num;

    int rightmost_bit = xor_all & -xor_all;
    int num1 = 0, num2 = 0;

    for(int num : nums) {
        if(num & rightmost_bit) num1 ^= num;
        else num2 ^= num;
    }

    return {num1, num2};
}

// XOR Basis
class XORBasis {
private:
    vector<int> basis;
    int size;

public:
    XORBasis(int bits = 32) : basis(bits, 0), size(0) {}

    void insert(int x) {
        for(int i = basis.size() - 1; i >= 0; i--) {
            if((x >> i) & 1) {
                if(!basis[i]) {
                    basis[i] = x;
                    size++;
                    return;
                }
                x ^= basis[i];
            }
        }
    }

    bool canRepresent(int x) {
        for(int i = basis.size() - 1; i >= 0; i--) {
            if((x >> i) & 1) {
                if(!basis[i]) return false;
                x ^= basis[i];
            }
        }
        return true;
    }

    int getMax() {
        int res = 0;
        for(int i = basis.size() - 1; i >= 0; i--) {
            if(basis[i] && (res ^ basis[i]) > res) {
                res ^= basis[i];
            }
        }
        return res;
    }

    int getSize() { return size; }
};

// Trie for maximum XOR
class MaxXORTrie {
private:
    struct TrieNode {
        TrieNode* children[2];
        TrieNode() {
            children[0] = children[1] = nullptr;
        }
    };

    TrieNode* root;

public:
    MaxXORTrie() {
        root = new TrieNode();
    }

    void insert(int num) {
        TrieNode* node = root;
        for(int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if(!node->children[bit]) {
                node->children[bit] = new TrieNode();
            }
            node = node->children[bit];
        }
    }
    int findMaxXOR(int num) {
        TrieNode* node = root;
        int max_xor = 0;

        for(int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            int opp_bit = 1 - bit;

            if(node->children[opp_bit]) {
                max_xor |= (1 << i);
                node = node->children[opp_bit];
            } else {
                node = node->children[bit];
            }
        }
        return max_xor;
    }
};

// Subarray XOR operations
int countSubarraysWithXOR(vector<int>& nums, int k) {
    unordered_map<int, int> prefix_count;
    prefix_count[0] = 1;

    int xor_sum = 0, count = 0;
    for(int num : nums) {
        xor_sum ^= num;
        count += prefix_count[xor_sum ^ k];
        prefix_count[xor_sum]++;
    }
    return count;
}

int maxSubarrayXOR(vector<int>& nums) {
    MaxXORTrie trie;
    trie.insert(0);

    int prefix_xor = 0, max_xor = 0;
    for(int num : nums) {
        prefix_xor ^= num;
        max_xor = max(max_xor, trie.findMaxXOR(prefix_xor));
        trie.insert(prefix_xor);
    }
    return max_xor;
}
```

```
    // Game Theory
    bool nimGame(vector<int>& piles) {
        int xor_sum = 0;
        for(int stones : piles) xor_sum ^= stones;
        return xor_sum != 0;
    }

    // Matrix XOR operations
    vector<vector<int>> buildXORPrefix(vector<vector<int>>& matrix) {
        int rows = matrix.size(), cols = matrix[0].size();
        vector<vector<int>> prefix(rows + 1, vector<int>(cols + 1, 0));

        for(int i = 1; i <= rows; i++) {
            for(int j = 1; j <= cols; j++) {
                prefix[i][j] = matrix[i-1][j-1] ^
                              prefix[i-1][j] ^
                              prefix[i][j-1] ^
                              prefix[i-1][j-1];
            }
        }
        return prefix;
    }

    int queryXORRectangle(vector<vector<int>>& prefix,
                         int r1, int c1, int r2, int c2) {
        return prefix[r2+1][c2+1] ^
               prefix[r1][c2+1] ^
               prefix[r2+1][c1] ^
               prefix[r1][c1];
    }

} // namespace XORUtils

#endif // XOR_UTILS_H
```

7.2 Test Cases and Examples

```cpp
#include "xor_utils.h"
#include <iostream>
#include <cassert>

void testBasicOperations() {
    cout << "Testing Basic Operations:\n";
    int a = 5, b = 3;
    XORUtils::swapNumbers(a, b);
    assert(a == 3 && b == 5);
    cout << "Swap test passed\n";

    assert(XORUtils::isPowerOfTwo(16) == true);
    assert(XORUtils::isPowerOfTwo(15) == false);
    cout << "Power of two test passed\n";
}

void testSingleNumber() {
    cout << "\nTesting Single Number:\n";

    vector<int> nums1 = {1, 2, 3, 2, 1};
    assert(XORUtils::singleNumber(nums1) == 3);

    vector<int> nums2 = {4, 1, 2, 1, 2};
    assert(XORUtils::singleNumber(nums2) == 4);
    cout << "Single number test passed\n";
}

void testTwoSingleNumbers() {
    cout << "\nTesting Two Single Numbers:\n";

    vector<int> nums = {1, 2, 3, 2, 1, 4};
    vector<int> result = XORUtils::twoSingleNumbers(nums);
    sort(result.begin(), result.end());
    assert(result[0] == 3 && result[1] == 4);
    cout << "Two single numbers test passed\n";
}

void testXORBasis() {
    cout << "\nTesting XOR Basis:\n";

    XORUtils::XORBasis basis;
    basis.insert(5);  // 101
    basis.insert(6);  // 110
    basis.insert(3);  // 011

    assert(basis.canRepresent(7) == true);   // 111 = 101 ^ 110
    assert(basis.canRepresent(1) == true);   // 001 = 101 ^ 110 ^ 011
    assert(basis.getSize() == 3);
    cout << "XOR basis test passed\n";
}

void testMaxSubarrayXOR() {
    cout << "\nTesting Max Subarray XOR:\n";
    vector<int> nums = {8, 1, 2, 12};
    int result = XORUtils::maxSubarrayXOR(nums);
    // Maximum is 8 ^ 1 ^ 2 ^ 12 = 15
    assert(result == 15);
    cout << "Max subarray XOR test passed\n";
}

void testNimGame() {
    cout << "\nTesting Nim Game:\n";

    vector<int> piles1 = {3, 4, 5};
    assert(XORUtils::nimGame(piles1) == true);  // First player wins

    vector<int> piles2 = {2, 2};
    assert(XORUtils::nimGame(piles2) == false); // Second player wins
    cout << "Nim game test passed\n";
}

int main() {
    cout << "Running XOR Algorithm Tests\n";
    cout << "===========================\n";

    testBasicOperations();
    testSingleNumber();
    testTwoSingleNumbers();
    testXORBasis();
    testMaxSubarrayXOR();
    testNimGame();

    cout << "\nAll tests passed successfully!\n";
    return 0;
}
```

7.3 Advanced Problem Solutions

Problem: XOR Queries on Tree

```cpp
class TreeXOR {
private:
    vector<vector<int>> adj;
    vector<int> values;
    vector<int> xorToRoot;
    vector<int> parent;
```

```
vector<int> depth;
vector<vector<int>> up; // for LCA
int LOG;

void dfs(int u, int p, int currentXOR) {
    parent[u] = p;
    depth[u] = (p == -1) ? 0 : depth[p] + 1;
    xorToRoot[u] = currentXOR ^ values[u];

    for(int v : adj[u]) {
        if(v != p) {
            dfs(v, u, xorToRoot[u]);
        }
    }
}

void preprocess(int n) {
    LOG = ceil(log2(n));
    up.assign(n, vector<int>(LOG + 1, -1));

    for(int i = 0; i < n; i++) {
        up[i][0] = parent[i];
    }

    for(int j = 1; j <= LOG; j++) {
        for(int i = 0; i < n; i++) {
            if(up[i][j-1] != -1) {
                up[i][j] = up[up[i][j-1]][j-1];
            }
        }
    }
}

int lca(int u, int v) {
    if(depth[u] < depth[v]) swap(u, v);

    int diff = depth[u] - depth[v];
    for(int i = LOG; i >= 0; i--) {
        if(diff & (1 << i)) {
            u = up[u][i];
        }
    }

    if(u == v) return u;

    for(int i = LOG; i >= 0; i--) {
        if(up[u][i] != up[v][i]) {
                u = up[u][i];
                v = up[v][i];
            }
        }

        return parent[u];
    }

public:
    TreeXOR(int n, vector<int>& vals, vector<vector<int>>& edges)
        : values(vals) {
        adj.resize(n);
        xorToRoot.resize(n);
        parent.resize(n);
        depth.resize(n);

        for(auto& edge : edges) {
            int u = edge[0], v = edge[1];
            adj[u].push_back(v);
            adj[v].push_back(u);
        }

        dfs(0, -1, 0);
        preprocess(n);
    }

    int queryPathXOR(int u, int v) {
        int l = lca(u, v);
        // XOR from u to v = xorToRoot[u] ^ xorToRoot[v] ^ values[l]
        return xorToRoot[u] ^ xorToRoot[v] ^ values[l];
    }

    void updateValue(int u, int newValue) {
        // Update all nodes in subtree of u
        // For efficiency, use Euler Tour + Segment Tree
        values[u] = newValue;
        // Recalculate XORs (simplified - in practice use segment tree)
        dfs(u, parent[u], (parent[u] == -1) ? 0 : xorToRoot[parent[u]]);
    }
};
```

Problem: XOR of All Subsequence Sums

cpp

```cpp
long long xorOfAllSubsequenceSums(vector<int>& nums) {
    int n = nums.size();
    long long result = 0;

    // Each element appears in 2^(n-1) subsequences
    // So contribution = arr[i] * 2^(n-1)
    // We only need XOR, so consider parity

    if(n == 1) return nums[0];

    // If n > 1, all elements appear even number of times
    // So XOR of all subsequence sums = 0
    return 0;
}
```

7.4 Performance Optimizations

```cpp
// Fast XOR operations for large arrays
void xorLargeArrays(int* arr1, int* arr2, int* result, int n) {
    // Use SIMD instructions for optimization
    #pragma omp parallel for simd
    for(int i = 0; i < n; i++) {
        result[i] = arr1[i] ^ arr2[i];
    }
}

// XOR with constant memory
int xorArrayInPlace(int* arr, int n) {
    int result = 0;
    for(int i = 0; i < n; i++) {
        result ^= arr[i];
    }
    return result;
}

// Parallel XOR using divide and conquer
int parallelXOR(int* arr, int l, int r) {
    if(l == r) return arr[l];

    int mid = (l + r) / 2;
    int left_xor, right_xor;

    #pragma omp task shared(left_xor)
    left_xor = parallelXOR(arr, l, mid);

    #pragma omp task shared(right_xor)
    right_xor = parallelXOR(arr, mid + 1, r);

    #pragma omp taskwait
    return left_xor ^ right_xor;
}
```

Key Insights and Tricks

1. XOR as Addition modulo 2

- XOR is equivalent to addition without carry
- Useful in linear algebra over GF(2)

2. Prefix XOR Technique

- Similar to prefix sums but with XOR
- `xor[l..r] = prefix[r] ^ prefix[l-1]`

3. Parity Preservation

- XOR preserves parity of ones
- Useful in error detection and correction

4. Gray Code Generation

```cpp
vector<int> grayCode(int n) {
    vector<int> result;
    for(int i = 0; i < (1 << n); i++) {
        result.push_back(i ^ (i >> 1));
    }
    return result;
}
```

5. XOR Swap Trick Limitations

- Doesn't work if both variables point to same memory location

- May confuse compiler optimizations

6. XOR in Cryptography

- Basis for one-time pad encryption
- `ciphertext = plaintext ^ key`
- `plaintext = ciphertext ^ key`

Complexity Analysis

Algorithm	Time Complexity	Space Complexity
Single Number	O(n)	O(1)
Two Single Numbers	O(n)	O(1)
XOR Basis Insert	O(log max_value)	O(log max_value)
Max XOR using Trie	O(n log max_value)	O(n log max_value)
Subarray XOR Count	O(n)	O(n)
Nim Game	O(n)	O(1)
Walsh-Hadamard Transform	O(n log n)	O(n)

Practice Problems

Easy:

1. Single Number (LeetCode 136)
2. Missing Number (LeetCode 268)
3. Swap Numbers without temp
4. Check if bits alternate

Medium:

1. Single Number II (LeetCode 137)
2. Maximum XOR of Two Numbers (LeetCode 421)
3. Count Triplets That Can Form Two Arrays (LeetCode 1442)
4. XOR Queries of Subarray (LeetCode 1310)

Hard:

1. Maximum XOR With an Element From Array (LeetCode 1707)
2. Number of Wonderful Substrings (LeetCode 1915)
3. Find the Longest Valid Obstacle Course (LeetCode 1964)
4. Count Pairs With XOR in Range (LeetCode 1803)

Conclusion

XOR is a fundamental operation with rich mathematical properties and wide applications in competitive programming. Its unique properties like self-inverse, associativity, and commutativity make it invaluable for solving various problems efficiently.