19.Finding the Missing Number and Unique Element Code Examples and Approaches

1. Finding the Missing Number in an Array of 1 to n

Brute Force Approach

Concept: Check each number from 1 to n to see if it exists in the array.
Time Complexity: O(n^2)
Space Complexity: O(1)

C++

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

int findMissingBrute(vector<int>& arr, int n) {
    for (int num = 1; num <= n; num++) {
        bool found = false;
        for (int elem : arr) {
            if (elem == num) {
                found = true;
                break;
            }
        }
        if (!found) {
            return num;
        }
    }
    return -1; // No missing number (invalid input)
}
```

Hashing Approach

Concept: Use a hash array to track numbers present in the array.
Time Complexity: O(n)

Space Complexity: O(n)

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

int findMissingHash(vector<int>& arr, int n) {
    vector<bool> present(n + 1, false); // Index 0 unused
    for (int num : arr) {
        present[num] = true;
    }
    for (int i = 1; i <= n; i++) {
        if (!present[i]) {
            return i;
        }
    }
    return -1;
}
```

Summation Approach (Optimal)

Concept: Compute the sum of 1 to n and subtract the array sum.
Time Complexity: O(n)
Space Complexity: O(1)

C++

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

int findMissingSum(vector<int>& arr, int n) {
    int total = n * (n + 1) / 2;
    int actual = 0;
    for (int num : arr) {
        actual += num;
    }
    return total - actual;
}
```

XOR Approach (Optimal)

Concept: XOR all numbers from 1 to n and XOR with array elements.
Time Complexity: O(n)
Space Complexity: O(1)

C++

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

int findMissingXOR(vector<int>& arr, int n) {
    int xorTotal = 0;
    for (int i = 1; i <= n; i++) {
        xorTotal ^= i;
    }
    int xorArr = 0;
    for (int num : arr) {
        xorArr ^= num;
    }
    return xorTotal ^ xorArr;
}
```

2. Finding the Unique Element in an Array of Pairs

Problem: All numbers appear twice except one. Find the unique number.
Example: `[4, 1, 2, 1, 2]` → Unique is `4`.

Brute Force Approach

Concept: Check each element's count using nested loops.
Time Complexity: O(n^2)
Space Complexity: O(1)

C++

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

int findUniqueBrute(vector<int>& arr) {
    for (int i = 0; i < arr.size(); i++) {
        int count = 0;
        for (int j = 0; j < arr.size(); j++) {
            if (arr[j] == arr[i]) {
                count++;
            }
        }
        if (count == 1) {
            return arr[i];
        }
    }
    return -1; // No unique element
}
```

Hashing Approach (Using Map)

Concept: Track counts using a hash map.
Time Complexity: O(n)

Space Complexity: O(n)

C++

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

int findUniqueHash(vector<int>& arr) {
    unordered_map<int, int> counts;
    for (int num : arr) {
        counts[num]++;
    }
    for (auto& pair : counts) {
        if (pair.second == 1) {
            return pair.first;
        }
    }
    return -1;
}
```

XOR Approach (Optimal)

Concept: XOR all elements; duplicates cancel out.
Time Complexity: O(n)
Space Complexity: O(1)

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

int findUniqueXOR(vector<int>& arr) {
    int unique = 0;
    for (int num : arr) {
        unique ^= num;
    }
    return unique;
}
```

Key Notes:

- **Summation vs. XOR:**
  - Summation may overflow for large `n`, while XOR avoids this by using bitwise operations.
  - XOR is more efficient for the unique element problem as it requires no extra space.
- **Interview Strategy:**
  - Always start with brute force, then optimize using hashing, and finally propose XOR for optimality.
  - Discuss trade-offs (e.g., time vs. space, handling large inputs).
- **Edge Cases:**
  - Missing number in unsorted arrays.
  - Arrays with negative numbers (requires adjustment for hashing).
  - Empty input or invalid `n` (handle with error checking).

These C++ code examples and explanations provide a structured approach to solving these common interview problems efficiently.