Class 5

Detailed Notes

Problem 1: Single Number

Link: Single Number

Solution:

```
int Solution::singleNumber(const vector<int> &A) {
    int num = A[0];
    for (int i = 1; i < A.size(); i++) {
        num = num ^ A[i];
    }
    return num;
}
```

Explanation:

This function finds the element that appears only once in an array where every other element appears twice. It uses the XOR operation, which has the property that x ⊕ x = 0 and x ⊕ 0 = x. Thus, XORing all elements results in the single number since pairs cancel out.

Why it works:

The XOR operation effectively removes pairs of the same number, leaving only the number that appears once.

Time Complexity: O(n), where n is the number of elements in the array.

Space Complexity: O(1), as no extra space is needed aside from a few variables.

Problem 2: Single Number II

Link: Single Number II

Solution:

```
int Solution::singleNumber(const vector<int> &A) {
    int arr[32] = {0};
    for (int i = 0; i < A.size(); i++) {
        for (int j = 31; j >= 0; j--) {
            if ((A[i] & (1 << j)) > 0) {
                arr[j]++;
            }
        }
    }
    long long num = 0;
    for (int i = 0; i < 32; i++) {
        if ((arr[i] % 3) != 0) {
            num += pow(2, i);
        }
    }
    return num;
}
```

Explanation:

This solution finds the element that appears exactly once in an array where every other element appears three times. It counts the occurrence of each bit position across all numbers. Any bit count not divisible by three indicates the bit belongs to the single number.

Why it works:

Since the number appears exactly once, and all others three times, only bits from the single number will contribute a remainder when the total bit counts are divided by three.

Time Complexity: O(n), where n is the length of the array, and each number is processed in constant time.

Space Complexity: O(1), using a fixed-size array for the 32-bit integers.

Problem 3: Min XOR Value

Link: Min XOR Value

Solution:

```
int Solution::findMinXor(vector<int> &A) {
    sort(A.begin(), A.end());
    int ans = INT_MAX;
    for (int i = 1; i < A.size(); i++) {
        ans = min(ans, (A[i] ^ A[i - 1]));
    }
    return ans;
}
```

Explanation:

The function finds the minimum XOR of two numbers in an array. By sorting the array first, the minimum XOR value is likely to occur between consecutive numbers due to smaller differences in their binary representations.

Why it works:

Sorting the array ensures that numbers with fewer differences in their higher-order bits are adjacent, minimizing the XOR result.

Time Complexity: O(n log n), due to the sorting step, followed by an O(n) scan.

Space Complexity: O(1), using constant space beyond the input storage.

Problem 4: Sum of Pairwise Hamming Distance

Link: Sum of Pairwise Hamming Distance

Solution:

```
int Solution::hammingDistance(const vector<int> &A) {
    int arr[32] = {0};
    for (int i = 0; i < A.size(); i++) {
        for (int j = 0; j < 32; j++) {
            if ((A[i] & (1 << j)) > 0) {
                arr[j]++;
            }
        }
    }
    long long ans = 0;
    for (int i = 0; i < 32; i++) {
        ans += ((long long)arr[i] * (A.size() - arr[i])) % 1000000007;
        ans = ans % 1000000007;
    }
    return (ans * 2) % 1000000007;
}
```

Explanation:

The function calculates the sum of pairwise Hamming distances between all integers in an array. The Hamming distance between two integers is the number of positions at which the corresponding bits are different. By counting the number of set bits at each position across all numbers, the sum can be computed efficiently.

Why it works:

Each bit contributes to the Hamming distance between all pairs where it is set in one integer and not set in the other. This count is efficiently calculated using the bit counts.

Time Complexity: O(n), where n is the number of integers, with each integer processed in constant time due to the 32-bit fixed length.

Space Complexity: O(1), using a fixed-size array.

Problem 5: Pairs With Given XOR

Link: Pairs With Given XOR

Solution:

```
int Solution::solve(vector<int> &A, int B) {
    map<int, int> m;
    int count = 0;
    for (int i = 0; i < A.size(); i++) {
        count += m[A[i] ^ B];
        m[A[i]]++;
    }
    return count;
}
```

Explanation:

The function counts pairs of integers in the array that have a specific XOR value B. It uses a map to keep track of how many times each integer appears. For each element, it checks if there exists a previously encountered integer whose XOR with it equals B.

Why it works:

This approach efficiently matches each number with potential pairs identified through the XOR operation. It utilizes the property that if $a \oplus b = B$, then $a = b \oplus B$.

Time Complexity: O(n), where n is the size of the array.

Space Complexity: O(u), where u is the number of unique elements in the array.