ChatGPT - Bitwise Concepts for FAANG

Give complete notes related bitwise concepts for FAANG preparation. "Must Include all patterns related to bitwise operators." If you are giving code, add binary representation in comments, Give input, output of the code. Explain the question of the code with sample input & output. Last but not least, your answer must cover all patterns related to FAANG interviews

3/3

Here's a comprehensive guide to **bitwise concepts** for FAANG preparation, including all patterns, practical examples, and sample inputs/outputs with explanations. Let's break it into clear sections:

---

1. Basics of Bitwise Operators

Operator	Symbol	Description
AND	`&`	Sets each bit to 1 if both bits are 1
OR	`	`
XOR	`^`	Sets each bit to 1 if only one of two bits is 1
NOT	`~`	Inverts all the bits (1's complement)
Left Shift	`<<`	Shifts bits to the left, adding zeros on the right
Right Shift	`>>`	Shifts bits to the right, keeping the sign bit intact
Unsigned Right Shift	`>>>`	Shifts bits to the right, filling with zeros

---

2. Core Patterns for Bitwise Operators

Pattern 1: Checking if a Number is Even or Odd

- **Concept**: The least significant bit (LSB) determines odd/even. If the LSB is 1, the number is odd; if it's 0, the number is even.

- **Code**:

```java
public class EvenOddCheck {
    public static void main(String[] args) {
        int num = 5;
        System.out.println((num & 1) == 0 ? "Even" : "Odd");
        // Explanation:
        // 5 in binary:   101
        // AND with 1:    001
        // Result:        001 (Odd)
    }
}
```

- **Input**: `num = 5`
- **Output**: `Odd`

Pattern 2: Swapping Two Numbers Without Using a Temporary Variable

- **Concept**: Use XOR to toggle bits.
- **Code**:

```java
public class SwapNumbers {
    public static void main(String[] args) {
        int a = 5, b = 3;
        a = a ^ b; // a = 5 ^ 3 => 101 ^ 011 = 110 (6)
        b = a ^ b; // b = 6 ^ 3 => 110 ^ 011 = 101 (5)
        a = a ^ b; // a = 6 ^ 5 => 110 ^ 101 = 011 (3)
        System.out.println("a = " + a + ", b = " + b);
    }
}
```

- **Input**: `a = 5`, `b = 3`
- **Output**: `a = 3, b = 5`

Pattern 3: Finding the Only Non-Repeating Element

- **Concept**: XOR of two identical numbers is 0. XORing all numbers gives the unique number.
- **Code**:

```java
public class SingleNonRepeating {
    public static void main(String[] args) {
        int[] nums = {2, 3, 2, 4, 4};
        int result = 0;
        for (int num : nums) {
            result ^= num;
            // Explanation:
            // Step 1: 0 ^ 2 = 2
            // Step 2: 2 ^ 3 = 1
            // Step 3: 1 ^ 2 = 3
            // Step 4: 3 ^ 4 = 7
            // Step 5: 7 ^ 4 = 3 (final unique element)
        }
        System.out.println("Unique element: " + result);
    }
}
```

- **Input**: `nums = {2, 3, 2, 4, 4}`
- **Output**: `Unique element: 3`

Pattern 4: Checking if a Number is a Power of Two

- **Concept**: A power of 2 has only one set bit.
- **Formula**: `(n & (n - 1)) == 0`
- **Code**:

```java
public class PowerOfTwo {
    public static void main(String[] args) {
        int n = 16;
        boolean isPowerOfTwo = (n > 0) && ((n & (n - 1)) == 0);
        // Explanation:
        // n = 16 in binary: 10000
```

```
        // n - 1 = 15 in binary: 01111
        // AND result: 10000 & 01111 = 00000 (true)
        System.out.println(isPowerOfTwo ? "Yes" : "No");
    }
}
```

- **Input**: `n = 16`
- **Output**: `Yes`

Pattern 5: Counting Set Bits (Hamming Weight)

- **Concept**: Count the number of 1's in the binary representation.
- **Code**:

```java
public class CountSetBits {
    public static void main(String[] args) {
        int n = 13; // 1101 in binary
        int count = 0;
        while (n > 0) {
            count += n & 1;
            n >>= 1; // Right shift by 1
        }
        System.out.println("Set bits count: " + count);
    }
}
```

- **Input**: `n = 13`
- **Output**: `Set bits count: 3`

Pattern 6: Finding the K-th Bit of a Number

- **Concept**: Right shift `k` times and check the LSB.
- **Code**:

```java
public class FindKthBit {
    public static void main(String[] args) {
```

```
        int n = 13; // Binary: 1101
        int k = 2;
        int bit = (n >> k) & 1;
        System.out.println("K-th bit: " + bit);
    }
}
```

- **Input**: `n = 13`, `k = 2`
- **Output**: `K-th bit: 1`

Pattern 7: Subsets Generation Using Bit Manipulation

- **Concept**: Use numbers from 0 to $2^n - 1$, treat each bit as whether to include an element.
- **Code**:

```java
import java.util.*;

public class GenerateSubsets {
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        List<List<Integer>> subsets = new ArrayList<>();
        int n = nums.length;
        for (int i = 0; i < (1 << n); i++) {
            List<Integer> subset = new ArrayList<>();
            for (int j = 0; j < n; j++) {
                if ((i & (1 << j)) != 0) { // Check if the j-th bit is set
                    subset.add(nums[j]);
                }
            }
            subsets.add(subset);
        }
        System.out.println(subsets);
    }
}
```

- **Input**: `nums = {1, 2, 3}`
- **Output**: `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`

3. Advanced Topics

1. **Bit Masking**: Used for dynamic programming problems on subsets.
2. **Gray Code**: Generating sequences where two successive values differ in one bit.
3. **Tricks for Specific Bit Positions**:
    - Clear the k-th bit: `n & ~(1 << k)`
    - Set the k-th bit: `n | (1 << k)`
    - Toggle the k-th bit: `n ^ (1 << k)`

---

4. Practice Problems

1. Find the missing number in an array (1 to n) using XOR.
2. Solve the maximum XOR of two numbers in an array.
3. Implement bitwise addition without using `+` or `-`.

This roadmap provides a structured approach to mastering bitwise operators for FAANG interviews. Let me know if you'd like to explore any section in further detail!

Can you give detailed notes regarding patterns on Bitwise advanced topics

Here are **detailed notes** on **advanced patterns involving bitwise operations**, along with sample questions, solutions, and explanations. These advanced topics are crucial for solving problems asked in **FAANG interviews**.

---

1. Bit Masking

Concept:

Bit masking involves using a binary number (the "mask") to extract, modify, or toggle specific bits of another binary number.

Common Patterns:

1. **Extracting Specific Bits**:
   - To extract the k-th bit: `(num >> k) & 1`
   - Example: Extract the 2nd bit of `13` (`1101`):
     - `(13 >> 2) & 1 = 3 & 1 = 1`
2. **Setting Specific Bits**:
   - To set the k-th bit: `num | (1 << k)`
   - Example: Set the 1st bit of `8` (`1000`):
     - `8 | (1 << 1) = 8 | 2 = 10` (`1010`)
3. **Clearing Specific Bits**:
   - To clear the k-th bit: `num & ~(1 << k)`
   - Example: Clear the 2nd bit of `13` (`1101`):
     - `13 & ~(1 << 2) = 13 & ~4 = 13 & 11 = 9` (`1001`)

Example Problem:

- **Question**: Find all subsets of a given set using bit masking.
- **Code**:

```java
import java.util.*;

public class SubsetsUsingBitMasking {
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        int n = nums.length;
        List<List<Integer>> subsets = new ArrayList<>();
        for (int mask = 0; mask < (1 << n); mask++) { // Iterate over 2^n masks
            List<Integer> subset = new ArrayList<>();
            for (int i = 0; i < n; i++) {
                if ((mask & (1 << i)) != 0) { // Check if the i-th bit is set
                    subset.add(nums[i]);
                }
            }
            subsets.add(subset);
```

```
        }
        System.out.println(subsets);
    }
}
```

- **Input**: `{1, 2, 3}`
- **Output**: `[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]`

---

2. Gray Code

Concept:

A **Gray code** is a binary numeral system where two successive values differ in only one bit. It is used in problems involving sequences, combinatorics, and circuits.

Pattern:

To generate Gray codes of length n:

- $G(i) = i \oplus (i >> 1)$
  - Where $\oplus$ is XOR.

Example Problem:

- **Question**: Generate the Gray code sequence for n = 3.
- **Code**:

```java
import java.util.*;

public class GrayCodeGenerator {
    public static List<Integer> grayCode(int n) {
        List<Integer> result = new ArrayList<>();
        int size = 1 << n; // 2^n Gray codes
        for (int i = 0; i < size; i++) {
            result.add(i ^ (i >> 1)); // Generate Gray code
        }
        return result;
```

```
    }

    public static void main(String[] args) {
        System.out.println(grayCode(3)); // Gray code for 3 bits
    }
}
```

- **Input**: `n = 3`
- **Output**: `[0, 1, 3, 2, 6, 7, 5, 4]`

---

3. Maximum XOR of Two Numbers in an Array

Concept:

- The problem involves finding two numbers in an array such that their XOR is maximized.
- Use a **trie** structure to efficiently check bitwise prefixes.

Example Problem:

- **Question**: Find the maximum XOR of two numbers in an array.
- **Code**:

```java
class TrieNode {
    TrieNode[] children = new TrieNode[2];
}

public class MaximumXOR {
    private static TrieNode root = new TrieNode();

    public static void insert(int num) {
        TrieNode node = root;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            if (node.children[bit] == null) {
                node.children[bit] = new TrieNode();
            }
```

```java
                node = node.children[bit];
            }
        }

    public static int findMaxXOR(int num) {
        TrieNode node = root;
        int maxXOR = 0;
        for (int i = 31; i >= 0; i--) {
            int bit = (num >> i) & 1;
            int oppositeBit = 1 - bit;
            if (node.children[oppositeBit] != null) {
                maxXOR = (maxXOR << 1) | 1;
                node = node.children[oppositeBit];
            } else {
                maxXOR = maxXOR << 1;
                node = node.children[bit];
            }
        }
        return maxXOR;
    }

    public static void main(String[] args) {
        int[] nums = {3, 10, 5, 25, 2, 8};
        for (int num : nums) {
            insert(num);
        }
        int maxXOR = 0;
        for (int num : nums) {
            maxXOR = Math.max(maxXOR, findMaxXOR(num));
        }
        System.out.println("Maximum XOR: " + maxXOR);
    }
}
```

- **Input**: `nums = {3, 10, 5, 25, 2, 8}`
- **Output**: `Maximum XOR: 28`

---

4. Count Total Set Bits from 1 to N

Concept:

Count all set bits in numbers from `1` to `N` using patterns in binary representation.

- **Formula**: Total set bits from `1` to `n` = `(n / 2) * x + (n % 2) + bits from sub-problems`.

Example Problem:

- **Question**: Count all set bits from `1` to `7`.
- **Code**:

```java
public class CountTotalSetBits {
    public static int countSetBits(int n) {
        int count = 0;
        int x = 0;
        while ((1 << x) <= n) {
            count += ((n + 1) / (1 << (x + 1))) * (1 << x);
            if (((n + 1) % (1 << (x + 1))) > (1 << x)) {
                count += ((n + 1) % (1 << (x + 1))) - (1 << x);
            }
            x++;
        }
        return count;
    }

    public static void main(String[] args) {
        System.out.println(countSetBits(7)); // Total set bits from 1 to 7
    }
}
```

- **Input**: `n = 7`
- **Output**: `Total set bits: 12`

---

5. Generate All Possible XOR Subarrays

Concept:

For any subarray, calculate XOR of all elements using prefix XOR.

Example Problem:

- **Question**: Find all XOR subarrays of `[1, 2, 3]`.
- **Code**:

```java
public class XORSubarrays {
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        for (int i = 0; i < nums.length; i++) {
            int xor = 0;
            for (int j = i; j < nums.length; j++) {
                xor ^= nums[j];
                System.out.println("XOR of subarray [" + i + ", " + j + "] = " +
xor);
            }
        }
    }
}
```

- **Input**: `nums = [1, 2, 3]`
- **Output**:

```css
XOR of subarray [0, 0] = 1
XOR of subarray [0, 1] = 3
XOR of subarray [0, 2] = 0
XOR of subarray [1, 1] = 2
XOR of subarray [1, 2] = 1
XOR of subarray [2, 2] = 3
```

Summary of Advanced Patterns:

1. **Bit Masking**: Extract, set, and clear bits; solve subset problems.

2. **Gray Code**: Use XOR for generating binary sequences.
3. **Maximum XOR**: Efficiently compute XOR for maximum value using Trie.
4. **Set Bits Count**: Compute total set bits efficiently for range 1 to n.
5. **Subarray XOR**: Use prefix XOR for solving range-based XOR problems.

These patterns cover a variety of real-world FAANG-style problems. Let me know if you want any additional clarifications or specific examples!