**BIT MANIPULATION**

1. **INTRODUCTION**

**Bitwise Operators** are special operators that operate directly on the binary representation of numbers. There are 6 primary Bitwise operators in C++. Most of the Bitwise operators are binary in nature, i.e., they operate on two operands. The only exception is Bitwise NOT operator which is unary in nature, i.e., it operates on a single operand. These operators are as follows –

* **Bitwise AND (&)**
* **Bitwise OR (|)**
* **Bitwise XOR (^)**
* **Bitwise NOT (~)**
* **Left-shift (<<)**
* **Right-shift (>>)**

The Bitwise Algorithms are used to perform operations at bit-level or to manipulate bits in different ways. The bitwise operations are found to be much faster and are sometimes used to improve the efficiency of a program. The time & space complexity of Bitwise operators is O(1).

A bitwise summary of the first three operators is shown in the table below –

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **A & B** | **A | B** | **A ^ B** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

**Bitwise AND:** It performs a logical AND operation on each pair of bits (as per the above table) and sets/unsets the corresponding bit in the resultant number.

**Bitwise OR:** It performs a logical OR operation on each pair of bits (as per the above table) and sets/unsets the corresponding bit in the resultant number.

**Bitwise XOR:** It performs a logical XOR operation on each pair of bits (as per the above table) and sets/unsets the corresponding bit in the resultant number.

The other three operators depend highly on how the numbers are represented in their binary form in the system. Their behavior is undefined in many cases like negative numbers, floating-point numbers, etc. The Left-shift and Right-shift operators cannot be used on negative numbers.

**Bitwise NOT Operator:** It toggles all the bits of the binary representation of a number, i.e., all 1-bits are changed to 0-bits and vice-versa. It works perfectly for unsigned integers as there is no sign issue.

**Left-shift Operator:** Let the number **A** be represented as a **n**-bit number, then **A << x** (where **x** represents the number of left shifts to be performed) shifts the **(n - x)** trailing bits to the left and adds **x** 0 bits at the end.

**Right-shift Operator:** Let the number **A** be represented as a **n**-bit number, then **A >> x** (where **x** represents the number of right shifts to be performed) shifts the **(n - x)** leading bits to the right and appends **x** 0 bits at the start ignoring the last **x** bits.

Let the number A = 45 (101101) and B = 89 (1011001), then –  
A & B = 9 (1001)  
A | B = 125 (1111101)  
A ^ B = 116 (1110100)  
A << 3 = 360 (101101000)  
B >> 2 = 22 (10110)  
~A = 4294967251 (11111111111111111111111111010010)

The Left-shift and Right-shift operators follow a very simple rule where,   
A << x = A \* 2xA >> x = ⌊ A / 2x ⌋

1. **IMPORTANT TACTICS**

* **How to set nth-bit in a number?**

Left-shift **1** to nth position using **1 << (n-1)**, then perform Bitwise OR operation (|) with the number. This will set the bit to 1. The OR operator is used as it requires only 1 set-bit among the two bits it operates on which we have provided in the form of 1 << (n-1).

* **How to unset nth-bit in a number?**

Left-shift **1** to nth position using **1 << (n-1),** then perform Bitwise NOT operation (~) with the number. This will unset the shifted **1**. Perform Bitwise AND Operation (&) with the number which will unset the bit at nth position.

* **Check if a Bit is set at the nth position**

Left-shift **1** to nth position using **1 << (n-1)**. This way only nth bit is set. Perform Bitwise AND Operation (&) with the number. If the result is zero, the bit is unset, else it is set.

* **Swapping two numbers using Bit Manipulation**

We can easily swap two numbers **a** and **b** using the Bitwise XOR Operator (^) by applying the below operations.

a = a^b;

b = b^a;

a = a^b;

* **Flipping the Bits of a number**

We can flip the bits of a number by simply subtracting the number from the number obtained if all the bits were set. Let the number be **n** where **n = 23**, then its binary form is 10111. If all the bits were set, it would be 11111 or **b = 31**. We can simply subtract **n** from **b** to get the resultant number, i.e. the result will be **b - n = 8 (01000)**.

* **XOR of all numbers from 1 to N**

The generic formula to calculate the XOR of numbers from 1 to N is – 1 ^ 2 ^ 3 ^ … ^ (N-2) ^ (N-1) ^ N. Let the result be X.

But, we have an even shorter method to calculate it. It is given as stated below –

1. Calculate N % 4, i.e., the remainder after dividing N by 4. Let the result be R.
2. If R = 0, 1, 2 or 3, then X = N, 1, N+1 and 0 respectively.

* **Check if a number is a power of 2**

If a number is a power of 2, its MSB is set and all other bits are unset. Now, if we subtract 1 from this number all the bits except MSB become set. Hence, if we perform Bitwise AND Operation on these two numbers, the result will always be 0. Therefore, if N & N-1 yields 0, then N is a power of 2, otherwise not.

* **Find the MSB of a number**

For any given number N, if X = ⌊log2 N⌋, then 2X represents the MSB in decimal number.

These tactics can be used in Data Structures & Algorithms to optimize the performance related to bit operations.

1. **Binary Representation of Negative Numbers**

C++ does not define the way in which negative numbers are represented in Binary form, but most of the times the negative numbers are represented as a 2’s complement in the memory. The 2’s complement of the negative of number X is calculated by inverting all the bits of the number X and then adding 1 to it. One direct formula can be used to represent 2N - X in binary form to represent negative X in 2’s complement. The MSB in 2’s Complement is always 1.

There are other forms of representing negative numbers – Signed MSB and 1’s Complement, but both face an issue where the number 0 has two representations. This is unacceptable as it makes Arithmetic Operations chaotic and ambiguous. The idea of 2’s Complement derives from the idea of subtracting the number from 0.

1. **Placeholder**