Introduction to Bitwise Manipulation

Let’s go over the Bitwise Manipulation pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following...

About the pattern

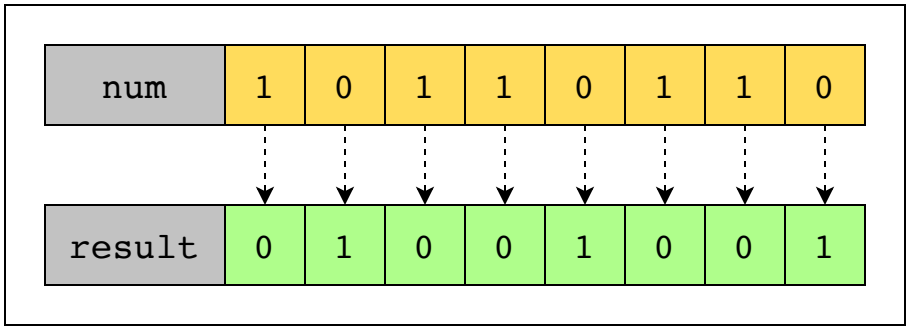
In programming, everything is stored in the computer’s memory as sequences of 0s and 1s, which are called bits. **Bitwise manipulation** is the process of modifying bits algorithmically using bitwise operations. Logical bitwise operations are the fastest computations because processors natively support them. This approach generally leads to efficient solutions in problems where we can efficiently transform the input into its binary form or manipulate it directly at the bit level to produce the required output.

We’ll be focusing on performing bitwise operations on the following two categories of binary numbers:

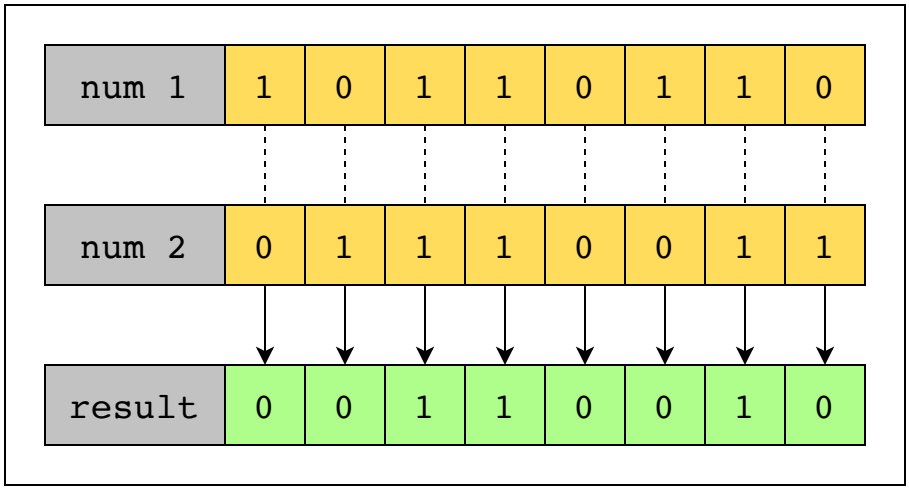
- **Unsigned binary numbers:** These numbers represent nonnegative integers.
- **Signed binary numbers:** Signed binary numbers represent both positive and negative integers. They include a sign bit (such as the leftmost bit in two’s complement representation) to indicate the sign of the number.

A bitwise operation works on a bit string, bit array, or a binary numeral. Bitwise operators take bits as their operands and calculate the corresponding bit value in the result. They include:

- **Logical NOT:** This is a unary operator that flips the value of the bit. If the bit it 1, we flip it to change a 0 and vice versa. Below is an example of how this operator works on an unsigned binary numeral:

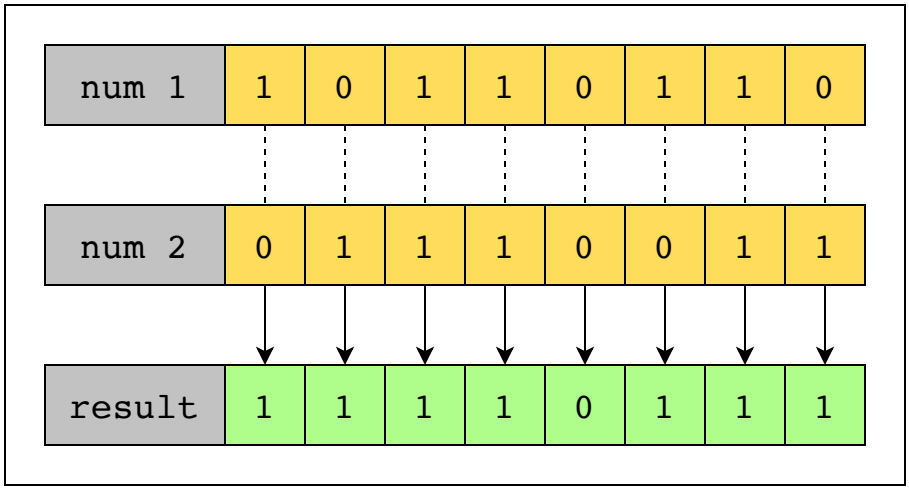


- **Logical AND:** This is a binary operator that evaluates two bits to 1 if both those bits are also 1. Otherwise, the evaluated result is always 0. Below is an example of how this operator works on two unsigned binary numerals:

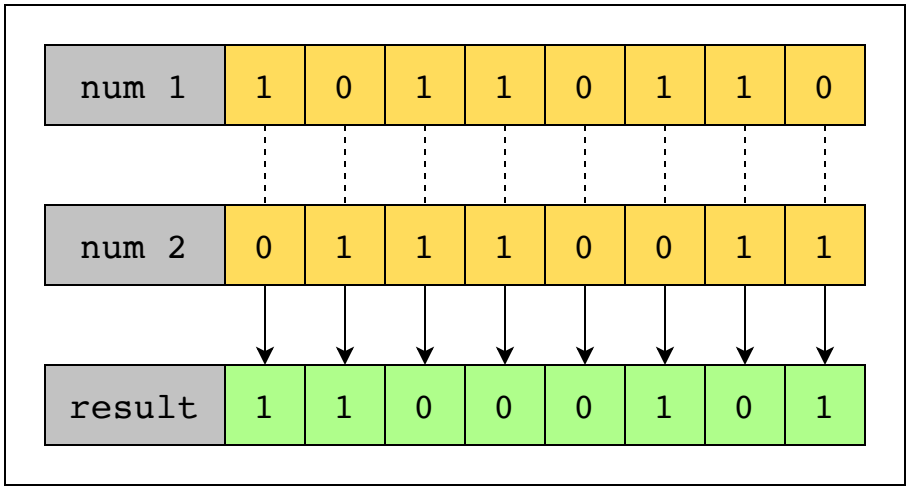


- **Logical OR:** This is a binary operator that evaluates two bits to 1 if at least one bit is also 1. Otherwise, the evaluated result is always 0.

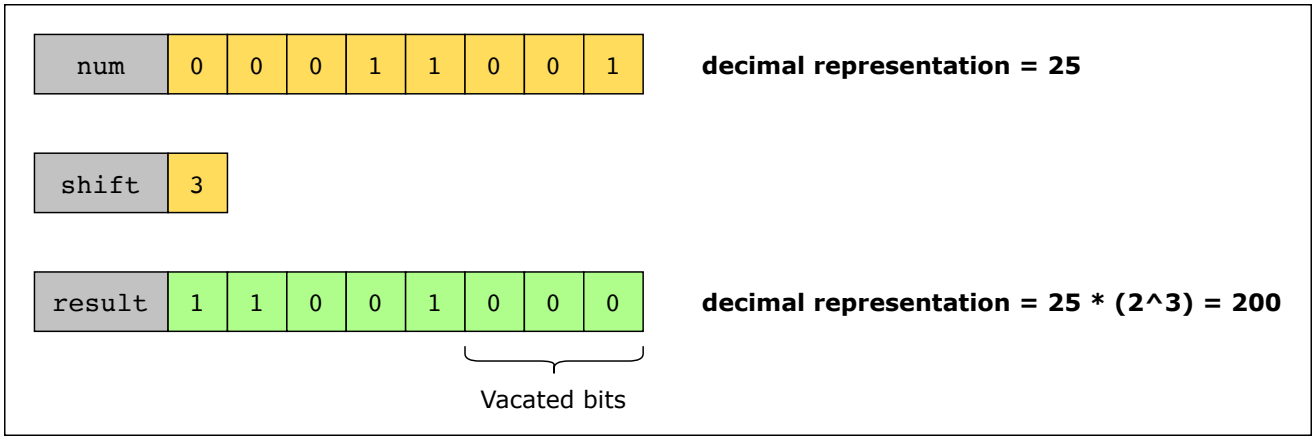
Below is an example of how this operator works on two unsigned binary numerals:



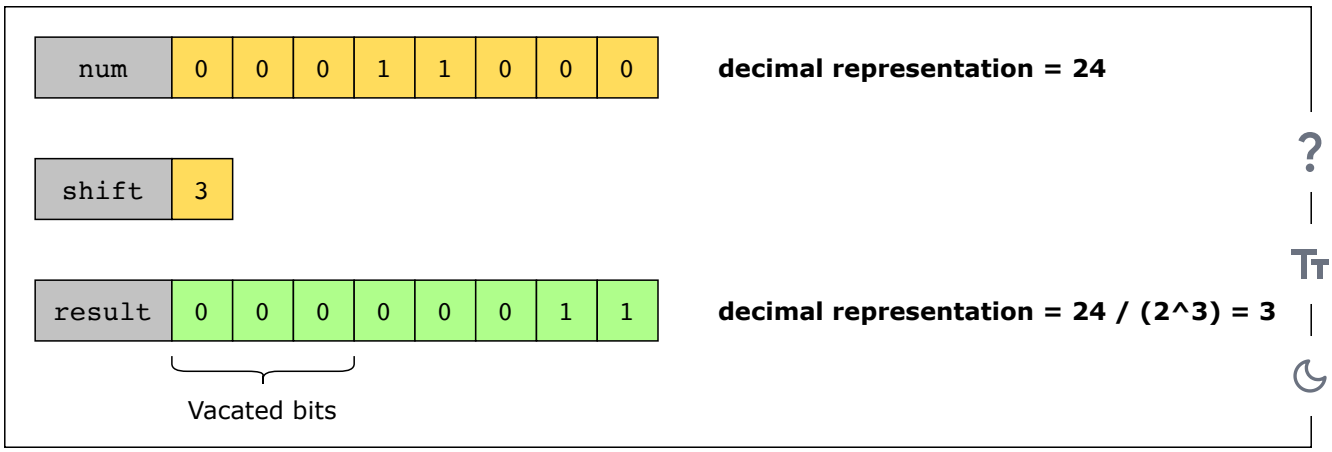
- **Logical XOR:** This is a binary operator that evaluates two bits to 1 only if both those bits are different, i.e., one is 0, and the other is 1. Otherwise, the evaluated result is always 0. Below is an example of how this operator works on two unsigned binary numerals:



- **Logical left shift:** This is a binary operator that shifts all of the bits in an unsigned binary number to the left by a specified number of places, filling the vacated bit(s) on the right with zero(s). Each shift to the left will multiply the number by 2, so performing a shift n places to the left on a binary number is equivalent to multiplying the decimal representation of that number by 2^n. Below is an example of how this operator works on an unsigned binary numeral:



- **Logical right shift:** This is a binary operator that shifts all of the bits in an unsigned binary number to the right by a specified number of places, filling the vacated bit(s) on the left with zero(s). Each shift to the right will divide the number by 2, so performing a shift n places to the right on a binary number is equivalent to dividing the decimal representation of that number by 2^n. Below is an example of how this operator works on an unsigned binary numeral:

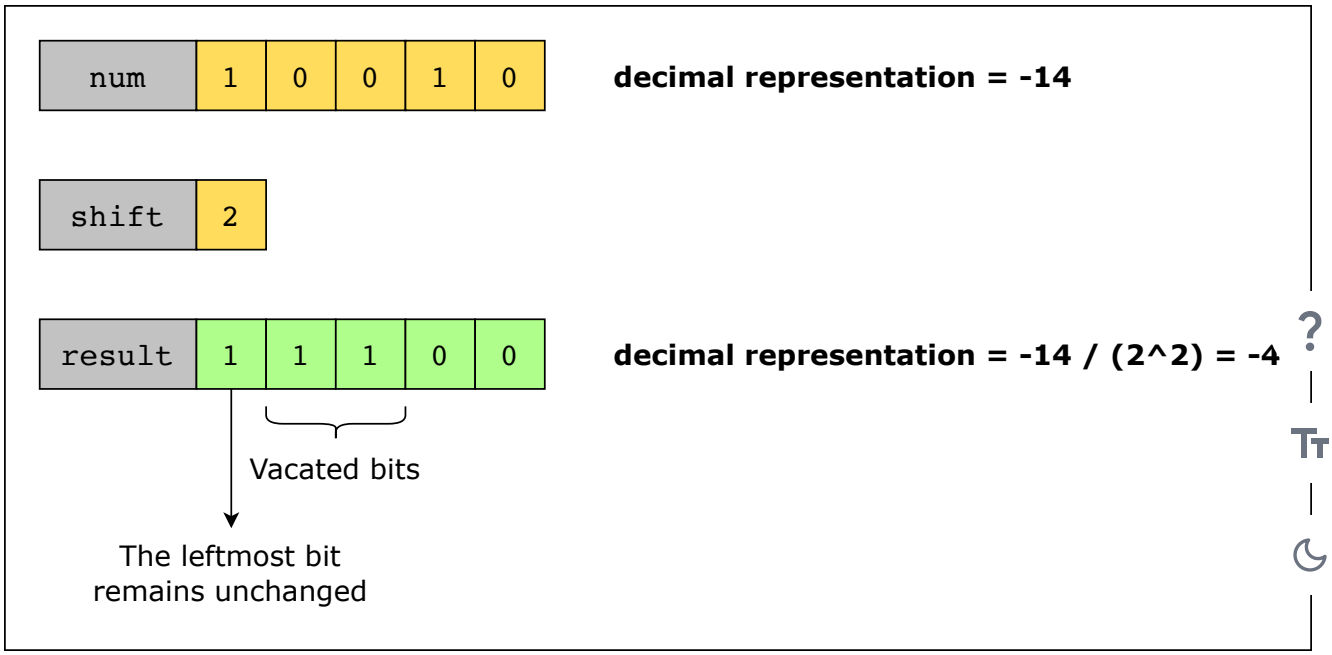


Signed numbers cannot be multiplied or divided using logical shifts because shifting the sign bit alone is inadequate; instead, arithmetic shifts are required for signed numbers.

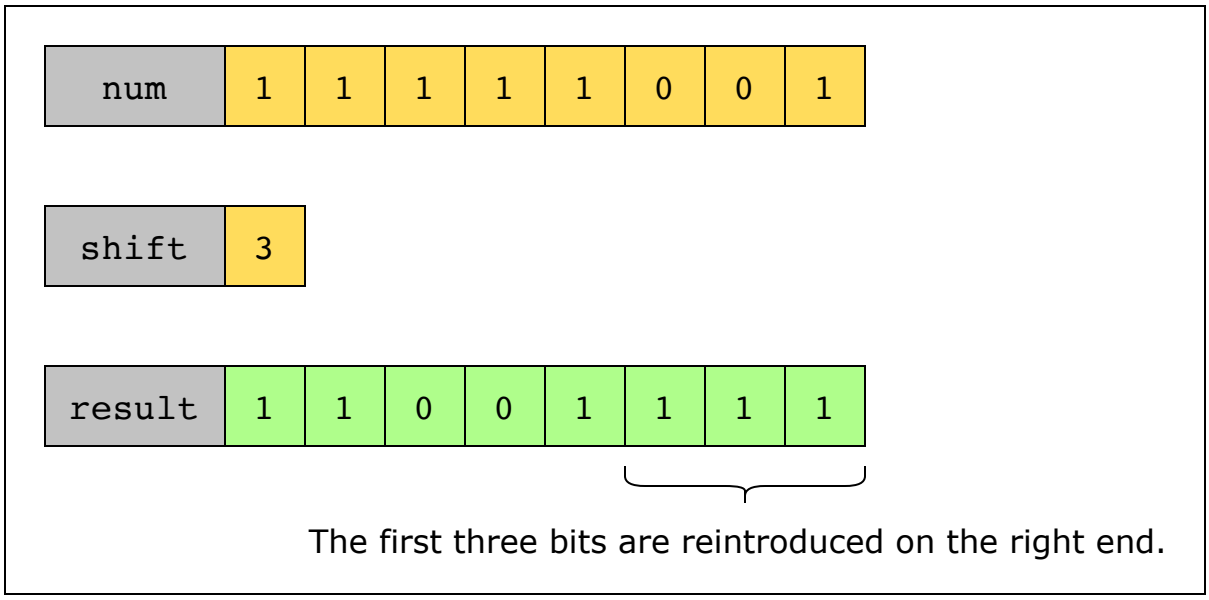
- **Arithmetic shifts:** This is a binary operator that, unlike logical shifts, maintains the sign bit (the leftmost bit) by keeping its position unchanged. The remaining bits are shifted and the vacated bits are filled as follows:

Operation	Sign	Method
Arithmetic left shift	Positive	Fill the vacated bits
Arithmetic left shift	Negative	Fill the vacated bits
Arithmetic right shift	Positive	Fill the vacated bits
Arithmetic right shift	Negative	Fill the vacated bits

Below is an example of how the arithmetic right shift operator works on a signed, negative binary numeral represented in two's complement:



- **Cyclic shifts:** A binary operator where the bits of a binary number are shifted to the left or right, and the vacated bits are reintroduced at the opposite end. Below is an example of how the cyclic left shift operator works on an unsigned binary numeral:

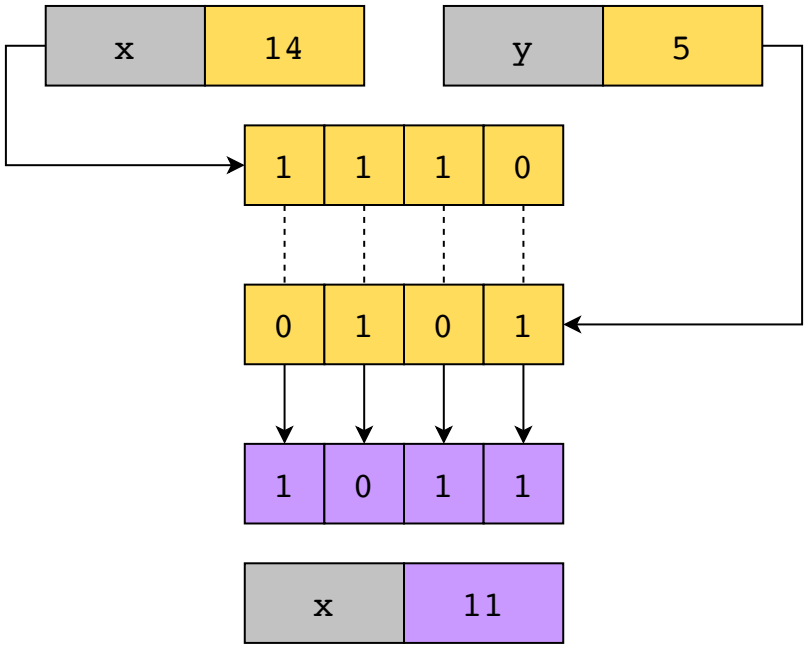


Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Swap without extra space:** Swap two numbers without using a temporary variable.

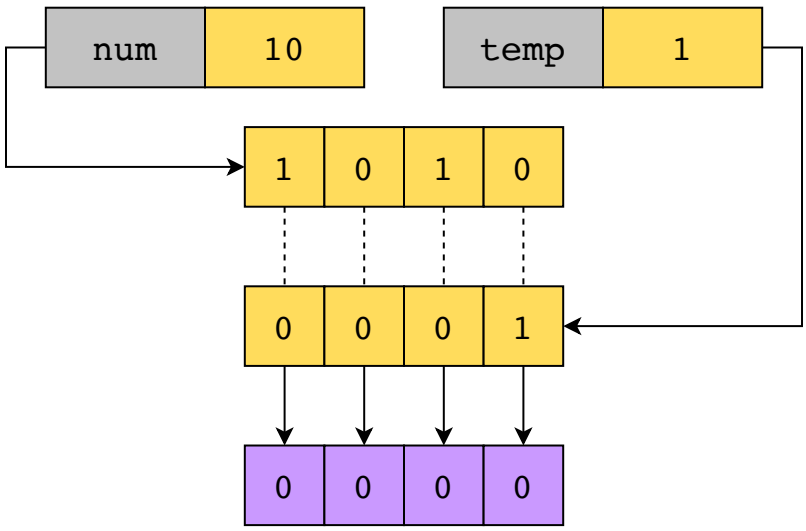
Step 1: We will first XOR **x** and **y** and store the result in **x**, which now contains a binary numeral containing 1s where the bits of **x** and **y** differ, and 0 where they dont.



1 of 4

2. **Check for even/odd without division/modulus:** Given a nonzero integer, determine whether it is even or odd without using division or modulus operators.

Step 1: We will first AND the number with **1**.



1 of 2

Does your problem match this pattern?

Yes, if both of these conditions are fulfilled:

- **Binary representation:** The input data can be usefully manipulated at the level of the primitive bitwise logical operations in order to compute some portion or all of the solution.
- **Efficient sorting:** The input data is unsorted, and the answer seems to require sorting, but we want to do better than $O(nlogn)$.

Real-world problems

Many problems in the real world share the bitwise manipulation pattern. Let's look at some examples.

- **Compression algorithms:** Bitwise algorithms are fundamental in compression techniques like Huffman coding, aiding in the efficient encoding and decoding of data at the bit level. They facilitate compact representation of variable-length codes by concatenating bits, and optimizing storage and transmission. By employing bitwise AND, OR, and shifting operations, compression algorithms achieve data compression without loss of information, crucial for resource-constrained environments.
- **Status register:** In the status register of a computer processor, each bit conveys a distinct significance. For instance, the initial bit of the status register denotes whether the outcome of an arithmetic operation is zero, known as the zero flag. This bit's value can be inspected, altered, or cleared by using a mask of identical length, wherein the relevant bit is set to 1. In the provided scenario, the mask selected would be 10000000, aligning with the zero flag situated at the first position.
- **Cryptography:** Cyclic shifts are commonly employed in cryptographic algorithms to introduce confusion and diffusion, enhancing security. By applying cyclic shifts, the relationship between the input and output data becomes complex, making it harder for attackers to decipher the original information. Additionally, cyclic shifts contribute to the avalanche effect, ensuring that even small changes in the input lead to significant changes in the output, strengthening the cryptographic algorithm's resilience against variou attacks.
- **Hash functions:** Bitwise operations are used to compute checksur in hash functions like Cyclic Redundancy Check (CRC) and Adler-32. These checksums are used for error detection and data integrity verification.

Strategy time!

Match the problems that can be solved using the bitwise manipulation pattern.

> **Note**: Select a problem in the left-hand column by clicking it, and then click one of the two options in the right-hand column.

Match The Answer

Select an option from the left-hand side

Check if an integer is a power of 2.	Bitwise Manipulation
Given a set of positive integers and a target sum, determine whether there is a subset of the given set that adds up exactly to the target sum.	Some other pattern