# **Bitwise Algorithms**

**Bitwise algorithms** in Data Structures and Algorithms (DSA) involve manipulating individual bits of binary representations of numbers to perform operations efficiently. These algorithms utilize bitwise operators like **AND, OR, XOR, shift operators**, etc., to solve problems related to tasks such as setting, clearing, or toggling specific bits, checking if a number is even or odd, swapping values without using a temporary variable, and more. Bitwise algorithms are crucial in optimizing code for speed and memory usage in various programming scenarios.

## **Common Bitwise Algorithms and Operations:**

Here are some common bitwise algorithms and operations:

* **Bitwise AND (&):** Takes two numbers as input and performs a bitwise AND operation on their corresponding bits. It returns 1 only if both bits are 1; otherwise, it returns 0.
* **Bitwise OR (|):** Performs a bitwise OR operation on the corresponding bits of two numbers. It returns 1 if at least one of the bits is 1.
* **Bitwise XOR (^):** Performs a bitwise exclusive OR operation on the corresponding bits of two numbers. It returns 1 if the bits are different and 0 if they are the same.
* **Bitwise NOT (~):** Performs a bitwise NOT operation, which flips each bit of the input (1 becomes 0 and 0 becomes 1).
* **Left Shift (<<) and Right Shift (>>):** These operators shift the bits of a number to the left or right by a specified number of positions. Left shifting is equivalent to multiplying the number by 2, while right shifting is equivalent to dividing by 2.

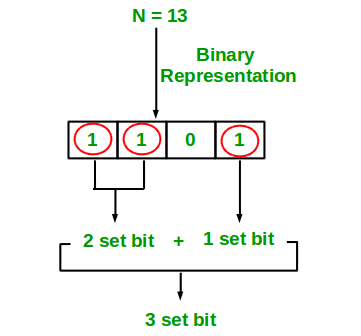
## **Applications of Bitwise Algorithms:**

* **Bit manipulation (setting, clearing, toggling bits):** Bitwise operators are often used to manipulate individual bits of numbers. This includes tasks such as setting bits (using OR), clearing bits (using AND with the complement), toggling bits (using XOR with 1), and checking the value of a specific bit.
* **Efficient storage of data:** Bitwise algorithms play a crucial role in data compression techniques like Huffman coding. They can efficiently represent and process compressed data by manipulating bits directly.
* **Cryptography:** Many cryptographic algorithms, such as AES (Advanced Encryption Standard), DES (Data Encryption Standard), and SHA (Secure Hash Algorithm), utilize bitwise operations for encryption, decryption, and hashing. Bitwise XOR, in particular, is commonly used in encryption algorithms for its simplicity and effectiveness.
* **Networking and Protocol Handling:** Bitwise algorithms are used in networking protocols for tasks like IP address manipulation, subnet masking, and packet parsing. For example, bitwise AND is used in subnet masking to determine the network address from an IP address and subnet mask.
* **Low-Level System Programming:** Bitwise operations are essential in low-level system programming for tasks such as device control, memory management, and bit-level I/O operations. They are used to manipulate hardware registers, set/clear flags, and optimize code for performance.
* **Error Detection and Correction:** Bitwise algorithms are employed in error detection and correction techniques, such as CRC (Cyclic Redundancy Check) and Hamming codes. These algorithms use bitwise XOR and other operations to detect and correct errors in transmitted data.

Write an efficient program to count the number of 1s in the binary representation of an integer.  
**Examples :**

***Input :*** *n = 6****Output :*** *2  
Binary representation of 6 is 110 and has 2 set bits*

***Input :*** *n = 13****Output :*** *3  
Binary representation of 13 is 1101 and has 3 set bits*



**Time Complexity:** O(log n)

**Auxiliary Space:** O(1)

# **Find the two non-repeating elements in an array of repeating elements/ Unique Numbers 2**

Last Updated : 28 Apr, 2024

Given an array **arr[]** containing **2\*N+2** positive numbers, out of which **2\*N** numbers exist in pairs whereas the other two number occur exactly once and are distinct. Find the other two numbers. Return in increasing order.

**Example:**

***Input:*** *N = 2, arr[] = {1, 2, 3, 2, 1, 4}****Output:****3 4****Explanation:*** *3 and 4 occur exactly once.*

***Input:*** *N = 1, arr[] = {2, 1, 3, 2}****Output:*** *1 3****Explanation:*** *1 3 occur exactly once.*

## **Non Repeating Numbers using** [**Sorting**](https://www.geeksforgeeks.org/sorting-algorithms/)**:**

*First, sort all the elements. In the sorted array, by comparing adjacent elements we can easily get the non-repeating elements.*

**Time complexity:** O(n log n)  
**Auxiliary Space:** O(1)

## **Non Repeating Numbers using XOR:**

*First, calculate the XOR of all the array elements.* ***xor = arr[0]^arr[1]^arr[2]…..arr[n-1]***

*All the bits that are set in xor will be set in one non-repeating element (****x*** *or* ***y****) and not in others. So if we take any set bit of xor and divide the elements of the array in two sets – one set of elements with same bit set and another set with same bit not set. By doing so, we will get* ***x*** *in one set and* ***y*** *in another set. Now if we do XOR of all the elements in the first set, we will get the first non-repeating element, and by doing same in other sets we will get the second non-repeating element.*

**Illustration:**

*We have the array: [2, 4, 7, 9, 2, 4]*

* *XOR = 2 ^ 4 ^ 7 ^ 9 ^ 2 ^ 4 = 2 ^ 2 ^ 4 ^ 4 ^ 7 ^ 9 = 0 ^ 0 ^ 7 ^ 9 = 7 ^ 9 = 14*
* *The rightmost set bit in binary representation of 14 is at position 1 (from the right).*
* *Divide the elements into two groups based on the rightmost set bit.*
  + *Group 1 (rightmost bit set at position 1): [2, 7, 2]*
  + *Group 2 (rightmost bit not set at position 1): [4, 9, 4]*
* *XOR all elements in Group 1 to find one non-repeating element.*
  + *Non-repeating element 1 = 2 ^ 7 ^ 2 = 7*
* *XOR all elements in Group 2 to find the other non-repeating element.*
  + *Non-repeating element 2 = 4 ^ 9 ^ 4 = 9*
* *The two non-repeating elements are 7 and 9,*

# **Swap bits in a given number**

Given a number x and two positions (from the right side) in the binary representation of x, write a function that swaps n bits at the given two positions and returns the result. It is also given that the two sets of bits do not overlap.

**Method 1**   
Let p1 and p2 be the two given positions.

**Example 1**

Input:

x = 47 (00101111)

p1 = 1 (Start from the second bit from the right side)

p2 = 5 (Start from the 6th bit from the right side)

n = 3 (No of bits to be swapped)

Output:

227 (11100011)

The 3 bits starting from the second bit (from the right side) are

swapped with 3 bits starting from 6th position (from the right side)

**Example 2**

Input:

x = 28 (11100)

p1 = 0 (Start from first bit from right side)

p2 = 3 (Start from 4th bit from right side)

n = 2 (No of bits to be swapped)

Output:

7 (00111)

The 2 bits starting from 0th position (from right side) are

swapped with 2 bits starting from 4th position (from right side)

**Solution**   
We need to swap two sets of bits. XOR can be used in a similar way as it is used to [swap 2 numbers](http://en.wikipedia.org/wiki/XOR_swap_algorithm). Following is the algorithm.

1) Move all bits of the first set to the rightmost side

set1 = (x >> p1) & ((1U << n) - 1)

Here the expression *(1U << n) - 1* gives a number that

contains last n bits set and other bits as 0. We do &

with this expression so that bits other than the last

n bits become 0.

2) Move all bits of second set to rightmost side

set2 = (x >> p2) & ((1U << n) - 1)

3) XOR the two sets of bits

xor = (set1 ^ set2)

4) Put the xor bits back to their original positions.

xor = (xor << p1) | (xor << p2)

5) Finally, XOR the xor with original number so

that the two sets are swapped.

result = x ^ xor

**Method 2 –**This solution focuses on calculating the values of bits to be swapped using AND gate. Then we can set/unset those bits based on whether the bits are to be swapped. For the number of bits to be swapped (n) –

* Calculate shift1 = The value after setting bit at p1 position to 1
* Calculate shift2 = The value after setting bit at p2 position to 1
* value1 = Number to check if num at position p1 is set or not.
* value2 = Number to check if num at position p2 is set or not.
* If value1 and value2 are different is when we have to swap the bits

**Example:**

[28 0 3 2] num=28 (11100) p1=0 p2=3 n=2

Given = 11100

Required output = 00111 i.e. (00)1(11) msb 2 bits replaced with lsb 2 bits

n=2

p1=0, p2=3

shift1= 1, shift2= 1000

value1= 0, value2= 1000

After swap

num= 10101

n=3

p1=1, p2=4

shift1= 10, shift2= 10000

value1= 0, value2= 10000

After swap

num= 00111

# **Count total set bits in first N Natural Numbers (all numbers from 1 to N)**

Last Updated : 13 May, 2024

Given a positive integer **N**, the task is to count the total number of set bits in [**binary representation**](https://www.geeksforgeeks.org/binary-representation-of-a-given-number/) of all numbers from **1 to N**.

**Examples:**

***Input:*** *N = 3****Output:*** *4****Explanation:*** *Numbers from 1 to 3: {1, 2, 3}  
Binary Representation of 1: 01 -> Set bits = 1  
Binary Representation of 2: 10 -> Set bits = 1  
Binary Representation of 3: 11 -> Set bits = 2  
Total set bits from 1 till 3 = 1 + 1 + 2 = 4*

***Input:*** *N = 6****Output:*** *9*

## **Count total set bits by converting each number into its Binary Representation:**

*The idea is to convert each number from 1 till N into* ***binary****, and**count the set bits in each number separately.*

Follow the steps below to understand how:

* Traverse a loop from **1** to **N**
* For each integer in **1** to **N**:
  + Convert the number to its binary representation
  + Add the count of **1s** in the binary representation to the answer.
* Return the total set bits count.

**Time Complexity:** O(N\*log(N)), where N is the given integer and log(N) time is used for the binary conversion of the number.  
**Auxiliary Space:** O(1).

**Method 2 (Simple and efficient than Method 1)**   
If we observe bits from rightmost side at distance i than bits get inverted after 2^i position in vertical sequence.   
for example n = 5;   
0 = 0000   
1 = 0001   
2 = 0010   
3 = 0011   
4 = 0100   
5 = 0101  
Observe the right most bit (i = 0) the bits get flipped after (2^0 = 1)   
Observe the 3rd rightmost bit (i = 2) the bits get flipped after (2^2 = 4)   
So, We can count bits in vertical fashion such that at i’th right most position bits will be get flipped after 2^i iteration;

# **Sum of bitwise AND of all possible subsets of given set**

Last Updated : 17 Aug, 2022

Given an array, we need to calculate the Sum of Bit-wise AND of all possible subsets of the given array.

**Examples:**

**Input :** 1 2 3

**Output :** 9

For [1, 2, 3], all possible subsets are {1},

{2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}

Bitwise AND of these subsets are, 1 + 2 +

3 + 0 + 1 + 2 + 0 = 9.

So, the answer would be 9.

**Input :** 1 2 3 4

**Output :** 13

In a **Better** approach, we are trying to calculate which array element is responsible for producing the sum into a subset.

Let’s start with the least significant bit. To remove the contribution from other bits, we calculate number AND bit for all numbers in the set. Any subset of this that contains a 0 will not give any contribution. All nonempty subsets that only consist of 1’s will give 1 in contribution. In total there will be 2^n – 1 such subset each giving 1 in contribution. The same goes for the other bit. We get [0, 2, 2], 3 subset each giving 2. Total 3\*1 + 3\*2 = 9

Array = {1, 2, 3}

Binary representation

positions 2 1 0

1 0 0 1

2 0 1 0

3 0 1 1

[ 0 2 2 ]

Count set bit for each position

[ 0 3 3 ] subset produced by each

position 2^n -1 i.e. n is total sum

for each position [ 0, 3\*2^1, 3\*2^0 ]

Now calculate the sum by multiplying

the position value i.e 2^0, 2^1 ... .

0 + 6 + 3 = 9