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 some times used to improve the efficiency of a program.

For example: To check if a number is even or odd. This can be easily done by using Bitwise-AND(&) operator. If the last bit of the operator is set than it is ODD otherwise it is EVEN. Therefore, if num & 1 not equals to zero than num is ODD otherwise it is EVEN.

Bitwise Operators

The operators that works at Bit level are called bitwise operators. In general there are six types of Bitwise Operators as described below:

- **& (bitwise AND)** Takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. Suppose **A = 5** and **B = 3**, therefore **A & B = 1**.
- **| (bitwise OR)** Takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1. Suppose **A = 5** and **B = 3**, therefore **A | B = 7**.
- **^ (bitwise XOR)** Takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different. Suppose **A = 5** and **B = 3**, therefore **A ^ B = 6**.
- **<< (left shift)** Takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
- **>> (right shift)** Takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
- **~ (bitwise NOT)** Takes one number and inverts all bits of it. Suppose **A = 5**, therefore **~A = 2**.

Important Facts about Bitwise Operators:

- The left shift and right shift operators cannot be used with negative numbers.
- The bitwise XOR operator is the most useful operator from technical interview perspective. We will see some very useful applications of the XOR operator later in the course.
- The bitwise operators should not be used in place of logical operators.
- The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively.
- The & operator can be used to quickly check if a number is odd or even. The value of expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.

Bit Algorithms | Important Tactics

Let's look at some of the useful tactics of the Bitwise Operators which can be helpful in solving a lot of problems really easily and quickly.

1. **How to set a bit in the number 'num' :** If we want to set a bit at nth position in number 'num' ,it can be done using 'OR' operator(|).
    - First we left shift '1' to n position via (1 << n)
    - Then, use 'OR' operator to set bit at that position.'OR' operator is used because it will set the bit even if the bit is unset previously in binary representation of number 'num'.

2. **How to unset/clear a bit at n'th position in the number 'num' :**

    Suppose we want to unset a bit at nth position in number 'num' then we have to do this with the help of 'AND' (&) operator.

    - First we left shift '1' to n position via (1 << n) than we use bitwise NOT operator '~' to unset this shifted '1'.
    - Now after clearing this left shifted '1' i.e making it to '0' we will 'AND'(&) with the number 'num' that will unset bit at nth position position.

3. **Toggling a bit at nth position :** Toggling means to turn bit 'on'(1) if it was 'off'(0) and to turn 'off'(0) if it was 'on'(1) previously.We will be using 'XOR' operator here which is this '^'. The reason behind 'XOR' operator is because of its properties.
    - Properties of 'XOR' operator.
        - 1^1 = 0
        - 0^0 = 0
        - 1^0 = 1
        - 0^1 = 1
    - If two bits are different then 'XOR' operator returns a set bit(1) else it returns an unset bit(0).

4. **Checking if bit at nth position is set or unset:**

   It is quite easily doable using 'AND' operator.

   - Left shift '1' to given position and then 'AND'('&').

   If the result of the AND operation is 1 then the bit at nth position is set otherwise it is unset.

5. **Divide by 2**:

```
x = x >> 1;
```

   Logic: When we do arithmetic right shift, every bit is shifted to right and blank position is substituted with sign bit of number, 0 in case of positive and 1 in case of negative number. Since every bit is a power of 2, with each shift we are reducing the value of each bit by factor of 2 which is equivalent to division of x by 2.
   Example:

```
x = 18(00010010)
x >> 1 = 9 (00001001)
```

6. **Multiplying by 2**:

```
x = x << 1;
```

   Logic: When we do arithmetic left shift, every bit is shifted to left and blank position is substituted with 0 . Since every bit is a power of 2, with each shift we are increasing the value of each bit by a factor of 2 which is equivalent to multiplication of x by 2.
   Example:

```
x = 18(00010010)
x << 1 = 36 (00100100)
```

7. **Find log base 2 of a 32 bit integer**:

```
int log2(int x)
{
    int res = 0;
    while (x >>= 1)
        res++;
    return res;
}
```

Logic: We right shift x repeatedly until it becomes 0, meanwhile we keep count on the shift operation. This count value is the log2(x).

8. **Flipping the bits of a number**: It can be done by a simple way, just simply subtract the number from the value obtained when all the bits are equal to 1 . **For example**:

```
Number : Given Number
Value  : A number with all bits set in given number.
Flipped number = Value - Number.

Example :
Number = 23,
Binary form: 10111;
After flipping digits number will be: 01000;
Value: 11111 = 31;
```

9. **Swapping Two Numbers**: We can easily swap two numbers say **a** and **b** by using the XOR(^) operator by applying below operations:

```
a ^= b;
b ^= a;
a ^= b;
```

Basic Problems on Bit Manipulation

Below are some problems which can be solved very easily using some of the basic concepts of Bit Manipulation. Let's look at each of these problems and the Bitwise approach of solving them:

- **Problem 1**: Given a number N, the task is to check whether the given number is a power of 2 or not.

  For Example:

```
Input : N = 4
Output : Yes
2^2 = 4

Input : N = 7
Output : No

Input : N = 32
Output : Yes
2^5 = 32
```

  Bitwise Solution: If we subtract a number which is a power of 2 1 then all of it's unset bits after the only set bit become set; and the set bit become unset.

  For example, consider **4** (Binary representation: 100) and **16**(Binary representation: 10000), we get following after subtracting 1 from them:

```
3 -> 011
15 -> 01111
```

You can clearly see that **bitwise-AND(&)** of 4 and 3 gives zero, similarly 16 and 15 also gives zero.

So, if a number N is a power of 2 then **bitwise-AND(&)** of **N and N-1** will be zero. We can say that N is a power of 2 or not based on the value of N&(N-1).

- **Problem 2**: Given a number N, find the most significant set bit in the given number.

  Examples:

```
Input : N = 10
Output : 8
Binary representation of 10 is 1010
The most significant bit corresponds
to decimal number 8.

Input : N = 18
Output : 16
```

  Bitwise Solution: The most-significant bit in binary representation of a number is the highest ordered bit, that is it is the bit-position with highest value.

  One of the solution is first find the bit-position corresponding to the MSB in the given number, this can be done by calculating logarithm base 2 of the given number, i.e., $\mathbf{log_2(N)}$ gives the position of the MSB in N.

  Once, we know the position of the MSB, calculate the value corresponding to it by raising 2 by the power of calculated position. That is, $\mathbf{value = 2^{log_2(N)}}$.

- **Problem 3**: Given a number N, the task is to find the XOR of all numbers from 1 to N.

  Examples :

```
Input : N = 6
Output : 7
// 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6  = 7

Input : N = 7
```

```
Output : 0
// 1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 = 0
```

Solution:

1. Find the remainder of N by moduling it with 4.
2. If rem = 0, then xor will be same as N.
3. If rem = 1, then xor will be 1.
4. If rem = 2, then xor will be N+1.
5. If rem = 3 ,then xor will be 0.

How does this work?

When we do XOR of numbers, we get 0 as XOR value just before a multiple of 4. This keeps repeating before every multiple of 4.

```
Number Binary-Repr  XOR-from-1-to-n
1          1           [0001]
2         10           [0011]
3         11           [0000]  <----- We get a 0
4        100           [0100]  <----- Equals to n
5        101           [0001]
6        110           [0111]
7        111           [0000]  <----- We get 0
8       1000           [1000]  <----- Equals to n
9       1001           [0001]
10      1010           [1011]
11      1011           [0000] <------ We get 0
12      1100           [1100] <------ Equals to n
```

Maximum AND Value | Explanation

Given an array arr[] of N positive elements. The task is to find the Maximum AND Value generated by any pair of the element from the array.

Note: AND is bitwise '&' operator.

Examples:

Input: a[] = {4, 8, 12, 16}
Output: 8
The pairs 8 and 12 gives us the '&' value as 12.

Input: a[] = {4, 8, 16, 2}
Output: 0

A **naive** approach is to iterate for all the pairs using two for loops and check for the maximum '&' value of any pair.

Note: This approach will not fit in the given time limit since the complexity of the above method jis O(N^2).

```
int findMaxium(int a[], int n)
{
    int maxi = 0;
    for(int i = 0;i<n;i++)
    {
        for(int j = i+1;j<n;j++)
            maxi = max(maxi, a[i] & a[j]);
    }

    return maxi;
}
```

Efficient Approach: An efficient approach will be to look at this problem bitwise. Since we need to find the maximum '&' value. The first thing that strikes our mind is that the answer should have its LSB as far as possible. So, if two elements are considered as a pair, then their LSB should be set to as much left as possible. Let's take an example to understand this. Consider three elements {10, 8, 2}, so to get a maximum '&' value we need to take those elements whose LSB is as far as possible. In the given example, we can clearly see that 10(1010) and 8(1000) have their 4th-bit from the left set and hence will maximize the answer. Taking 2 and 10 will give our 2nd bit to be set which won't maximize our answer.

So since the constraints permit till 10^4, hence the '&' value will also be less than that. 10^4 will range in 2^0 to 2^14, which means we need to start our checking from the 15th bit. Initially we loop from 15 to 0 and check for the count of numbers whose that particular bit is set. Once we get the count more than 2, the answer will have that bit set, and for the next bit from the left to be set we need to check for both the previous all bits and the current i-th bit. The previous bits can be added to the current bit using a '|' operator. In this way, we get all the positions of the bit which are set, which can be easily represented as a number.

Note: We have started checking from bits 31 so that if the constraints were high, it can easily fit in.

```
// Utility function to check number of elements
// having set msb as of pattern
int checkBit(int pattern, int arr[], int n)
{
```

```
    int count = 0;
    for (int i = 0; i < n; i++)
        if ((pattern & arr[i]) == pattern)
            count++;
    return count;
}

// Function for finding maximum and value pair
int maxAND (int arr[], int n)
{
    int res = 0, count;

    // iterate over total of 30bits from msb to lsb
    for (int bit = 31; bit >= 0; bit--)
    {
        // find the count of element having set  msb
        count = checkBit(res | (1 << bit),arr,n);

        // if count >= 2 set particular bit in result
        if ( count >= 2 )
            res |= (1 << bit);
    }

    return res;
}
```