```java
package Level_01_Bit_Manipulation;

public class Introduction_to_Bit_Manipulation {

    // Whenever we are storing something that will store in Binary manner in the machine but when we are retrieve then we will find same as store.
// for ex. if we write int n = 57 then it will save as 111001 in memory and when you are going to print it then it will return 57 again.

    // For the integer value we have 4 types in Java.
    /*
    1 -> byte -> 8 bits, it can represent 2^8 distinct numbers
    2 -> short -> 16 bits, it can represent 2^16 distinct numbers
    3 -> int -> 32 bits, it can represent 2^32 distinct numbers
    4 -> long -> 64 bits, it can represent 2^64 distinct numbers

      but how they are stored +ve , -ve let us see that,

       let us assume an example which help us to understand all these
```

for ex. we have another representation of int like nibble which has 4 bits, it's an assumption, it's not a part of Java.

Approach - 01 ->

so we can store 2^4 distinct numbers that is 16 let us see all those numbers

0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101, 1110,1111

0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 , 13 , 14 , 15

if this is like that means 0 to 15 that means 0 to 2^4 - 1. it's not possible because we have only +ve numbers here but what about -ve numbers

Note -> Approach 1 has failed because there is no any provision for the -ve numbers.

Approach 02 ->

So let us assume that most significant bit (MSB) is denoting the sign of the number and apart from that denoting the numbers value assume if MSB(sabse left wale ko) is 0 then +ve and if 1 then -ve.

let us see that this right approach or not

0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111

0, 1 , 2 , 3 , 4 , 5 , 6 , 7 , -0 , -1 , -2 , -3 , -4 , -5 , -6 , -7

Note -> Approach 2 has failed as well due to two reasons first one is there is no any +0 and -0 concepts in maths

and second one is if 0000 to 00001 then we see numbers increase in Binary but after -0 it is decreasing instead of increasing like

if 0000 is -0 then 0001 must be -0 + 1 = 1 but not it is showing -1 that is also wrong.

Approach 03 -> It's a right approach which is used in Java let us explore it...>

0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111

approach 3 is also recognize sign with help of MSB that is left most bit.

but it does not say that for ex. 0000 then 0 for sign and 000 for value.

Note -> it means all 4 bits will use for the value then what is difference ?

so for the +ve value means if MSB is 0 then there is no problem take all bits and make number that will a +ve number for ex 0010 -> 2

Note -> But for the -ve numbers means when MSB is found as 1 then means whenever we found MSB as 1 then 100% it's a -ve number

but how we can know about that number then the Answer is find the 2's complement of that number

Note -> So for the -ve numbers we need to find 2's complement to find out that number.

Note -> 2's complement means => 1's complement + 1

for ex. 1000 -> i's - 0111 then 0111 + 0001 = 1000 and converts it in to decimal that is 8 and put - on front of this means => -8

Imp. Note-*** So Rule =>   0 MSB -> simple convert in to decimal

1 MSB -> 2's complement + 1 => convert it in to Decimal and put it -ve sign with that number.

So According to approach three we can show it like that

0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111

0, 1 , 2 , 3 , 4 , 5 , 6 , 7 , -8 , -7 , -6 , -5 , -4 , -3 , -2 , -1 so if we go for the next of -1 then that will be 0.

so it's flows in circular manner. --> this is flow after 7 there is no any +ve number in 4 bits that's why we jump on -8.

for the shake of easy we can understand it from -8.

So imp thing is that Range be circular.

So we can see easily that all numbers are distinct and also in increasing order where ever they are starting.

for decimal to binary to decimal for -ve numbers firstly find out the binary representation of that number and take 2's complement set MSB as 1.

So if are following Approach three then we can easily see that for the 4 bits the min number is -2^3 = -8 and the max is 2^3-1 = 7.

so we can say that for 4 bits we have range -2^3 to 2^3 - 1 .

*** means if we have n bits then we can say that Decimal range for that bits is = (-2^n-1 to (+2^n-1)-1)

So let us see the range for our Java Integers

1 -> byte -> 8 bits, it can represent 2^8 distinct numbers -> Range => (-2^7 to +2^7-1)

2 -> short -> 16 bits, it can represent 2^16 distinct numbers -> Range => (-2^15 to +2^15-1)

3 -> int -> 32 bits, it can represent 2^32 distinct numbers -> Range => (-2^31 to +2^31-1)

4 -> long -> 64 bits, it can represent 2^64 distinct numbers -> Range => (-2^63 to +2^63-1)

*/

/*

-> If we put number from out of bound then what happened ?

To understand it again we will take same example as previous, like 4 bits data types.

nibble as 4 bits

for ex. if we write nibble x = 12; then it indeed it stored as -4. because of range -8 to 7.

12 in binary 1100 which -4 for the nibble or for range -8 to 7.

another example -> if we want to store 16 in nibble, so 16 in binary 10000 it has 5 bits but in nibble has only 4 bits

so it will assume only 4 bits from the last that is 0000 and store 0 because in binary 0 -> 0000 for 4 bits.

another most important thing we need to understand that is 1010 = -6 in nibble but if talk for byte then

byte have 8 bits means 00001010 = 10 because it has bigger range that's why he can store it as +ve because MSB is 0.

Decimal to Binary , Binary to Decimal

-> +ve -> -ve -> MSB = 0 -> MSB = 1

-> convert to -> leave the -> convert -> take 2's complement

Binary -ve sign to decimal -> convert to decimal

-> Fit in bits -> convert to Binary -> give +ve sign -> give it -ve sign.

-> Fit in bits

-> take 2's complement

-> and store it.

Fit in bits -> if we want 4 bits number, but we have 5 bits number then take 4 bits from the last and leave the remaining things.

if we have 5 bits number but our range is 8 bits then append zero as prefix for ex. 10000 in byte -> 00010000 like that.

ex. to convert -ve number in binary -> -7 so leave the -ve sign and convert it into binary that is 111

-> and fit in bits if our range is for 4 bits then it will be 0111

-> take 2's complement => 1000+1 = 1001 = -7. like that.

*/

/*

Bitwise Operators -->

| -> OR, & -> AND, ^ -> XOR, << -> Left Shift, >> -> Right Shift, >>> -> Triple right Shift, ~ -> 1's complement.

Imp -> if we have a number x, and we want to take 2's complement of that number then, right with -x which is in 2's complement.

=> OR(|) -> It can be used to switch on the bits generally.

if a belongs to (0, 1) means 'a' is a bit.

a | 1 = 1, a | 0 = a or we can say that OR gives 1 if any bit is one if both are 0 then it will give 0.

or if we take OR with 0 for any number it will return that number. so in OR 1 is powerful and 0 is powerless

=> AND(&) -> It can be used to switch off the bits generally.

a & 0 = 0, a & 1 = a. in AND 0 is powerful and 1 is powerless. or we can say in the case of AND result will be one when both bits are 1 otherwise, if anyone of them is 0 then result will be 0.

=> XOR(^) -> It can be used for Toggle to a bits.

a ^ 1 = ~a, a ^ 0 = a, a ^ a = 0 means here 1 is powerful as well.

or in other words if both bits are same it will give 0 as result and 1 as result if both are different.

=> Left Shift (<<) ->

if x = 00101011 -> x << 3 => so 3 0's will add from the last and whatever in front 3 bits will be removed.

x << 3 = 01011000

x << b = x * x^b -> it works like that also.

=> Right Shift (>>) ->

we can say it's reverse of left shift

y = 10100110 -> y >> 3 => so it will append 3 bits on front same as MSB. if MSB = 0 then it will add 0 and if 1 then 1.

and drop 3 bits from the last. y >> 3 = 11110100 y >> b = y / 2^b -> it works like that also.

=> Triple Right Shift (>>>) => It will do work same as Right shift but, it's not depend on MSB it will always append 0 on front

y >>> 3 = 00010100.

=> NOT (~) -> it will reverse all bits.

*/

// Question -> We need to do these operations like try to do,

// on a bit, off a bit, toggle a bit, check a bit that it's on or off.

// OR is best, AND is best XOR is best we can use anyone, but we use AND.

/*

=> ON to do it we use OR ->

if x = 10110101

- means for 0 check it if it is on then it's alright, and if not then ON it.

So to do it we need to make mask which have every element is 0 instead of the target element, we put 1 for that.

x = 10110101 | 00001000 = 10111101. because we know that in OR 0 is powerless but one is powerful because 0 | 0 = 1 , 0 | 1 = 1, 1 | 1 = 1.

to make Bit mask we will use Left Shift Operator.(0000001 and try to implement left shift.

=> OFF a bit to do it we use &(AND) operator -> Here we make mask as well but for the AND we know that 1 is powerless then make 11111011

mask will consist 1 except for the target bit.and for target bit we put 0.

so here we will make bit mask same as ON but instead of using that we will take 1's complement

for ex. -> 0000001 -> left shift, 0001000 and take one's complement -> 1110111. like that toggle

=> Toggle -> use XOR operator -> make bit mask same as OFF and take XOR with x and will find our answer.

because in XOR 0 is powerless and 1 is powerful.

```
    => For check a bit we can use OR, AND both ->


                              11010110 & 00010000 = 00010000 means target bit is
ON, here only make mask and take & we can find easily

              and in the case of OR we need to 1's complement of Mask for ex,
11010110 | 11101111 = if target bit is one then it will give 1 otherwise 0.

               ex. 11000110 & 00010000 = 00000000

              means x & mask == 0 means bit is 0  otherwise 1.

     */



    public static void main(String[] args) {

       // System.out.println((0 ^ 1) + " " + (1 ^ 0)); // both will give 1 and 1 is
powerful because in XOR a ^ 1 = ~a , a can be 0 or 1.

        int i = 4;// do Ith bit ON

        int j = 3; // do Jth bit OFF

        int k = 5; // toggle Kth bit

        int l = 2; // Check Lth bit is ON or OFF


        int n = 12;
        int onMask = (1 << i); // at the time of using Binary Operator we need
to take brackets because all those have less precedence than '='.

        int offMask = ~(1 << j);

        int toggleMask = (1 << k);// there is also required regular mask that is
nothing but which is used in ON Mask.

        int checkMask = (1 << l); // 1 means
00000000000000000000000000000001 this is for int.


        System.out.println("For Ith bit, to do ON : " + (n | onMask));

        System.out.println("For Jth bit, to do OFF : " + (n & offMask));

        System.out.println("For toggle Kth bit : " + (n ^ toggleMask));

        System.out.println("To check Lth bit is ON or OFF : " + (((n &
checkMask) == 0) ? "OFF" : "ON"));

    }

}
```