BIT MANIPULATION (GFG)

Date
DELTA Pg No.

① Bitwise (AND) (&) → 1 → both are 1
0 → otherwise

x = 3, y = 6

Binary Representation:

x: 00 ... 011
y: 00 ... 110
(x & y): 00 ... 010

30 bit representation
→ 64 bit "
→ 16 " "
(depends on compiler)

```
int main() {
    int x = 3, y = 6;
    cout << (x & y) << endl;
    cout << (x | y) << endl;
    cout << (x ^ y) << endl;
    return 0;
}
```

Output
↳ 2
7
5

② Bitwise (XOR) (^) → 1 → either of two is 1 and other is 0
→ 0 → otherwise

x = 3, y = 6

x: 00 ... 0011
y: 00 ... 0110
(x ^ y): 00 ... 0101

Date
DELTA Pg No.

③ Left Shift Operator (<<)

$x << n$ → how many left shifts to do.

↓

no. whose binary representation is to be shifted

```
int main() {
int x = 3;

cout << (x << 1) << endl;
cout << (x << 2) << endl;
int y = 4
int z = (x << y);
cout << z;
return 0;
}
```

Output: 6 → 3 × 2¹
12 → 3 × 2²
48 → 3 × 2⁴

x : 000 --- 011
(x<<1) : 000 -- 110 → Shifts by 1 at put 1 trailing 0.
(x<<2) : 000 --- 1100
(x<<3) : 00 --- 11000
(x<<y) : 00 --- 110000 → Similar with these.

* (x << y) is equivalent to $x * 2^y$, if first few bits in binary representation of no. are 0.

④ Right Shift Operator (>>)

(x >> n)

↳ bits in binary representation are shifted by n positions and last n bit is ignored and n leading 0 are added.

Date
DELTA Pg No.

x : 00 ---- 01000001 → 33
(x>>1) : 00 ---- 00100000 → 16 → $\lfloor 33/2^1 \rfloor = 16$
(x>>2) : 00 ---- 00010000 → 8 → $\lfloor 33/2^2 \rfloor = 8$
(x>>4) : 00 ---- 00000010 → 2 → $\lfloor 33/2^4 \rfloor = 2$

* (x >> y) is equivalent to floor $(x/2^y)$ i.e

$$\left\lfloor \frac{x}{2^y} \right\rfloor$$

(5) **Bitwise Not (~)** → Invert all the bits in binary representation of a no.

x = 1
x = 000 --- 01
~x = 111 --- 10

In 32 bit representation of a Number :-

000 --- 00 → 32 0's
⋮
111 --- 1 → 32 1's

→ 0 → $2^{32} - 1$ → 4294967296 − 1
→ 4294967295
→ Max value of 32 bit integer.

∴ x = 1
~x = $(2^{32} - 1) - 1$
∴ x = 5
~x = $(2^{31} - 1) - 5$
→ 4294967290

```
int main() {
    unsigned int x = 1;
    cout << (~x) << endl;
    x = 5;
    cout << (~x) << endl;
    return 0;
}
```

-2^{31} to $2^{(31)}-1$ → 31 since one bit is used as sign bit.

Date
DELTA Pg No.

→ Bitwise Not for Signed Input → use leading bit as signed bit

$x = 1$, $(2^{32}-1-1)$

x: (0) --- 01 ↗ ↓ 0 → +ve

$\sim x$: (1) --- 10 $2^{32}-2$ 1 → -ve

↳ signed bit ↳ here $x=2$

∴ if $x \to +ve$, $\sim x \to -ve$

$2^{31}-1$ → first bit 0 then 31 1's

-2^{31} → first bit 1 then 31 0's

Signed no. are usually stored as 2's complement representation i.e ↳ if no. is +ve then put 1st bit 0 and then represent rest of the no.

↳ if no. is -ve, put a first bit 1 and then rest as 2's complement representation bit.

↳ Why? Since otherwise we will have +0 & -0 that is not possible.

↳ Allows us to have only 1 representation of 0.

✳ 2's complement of x in n bits representation ↳ for -ve no.

$$= 2^n - x$$

eg, $n = 3$,

$-2 \Rightarrow 2^3 - 2 : 6 \to 110$

```
int main() {
    int x = 1;
    cout << (~x) << endl;   → -2
    x = 5;
    cout << (~x) << endl;
}
```

↳ $2^{32} - 1 - 5 = 2^{32} - 6$

↳ OP: -6 → this is also dependent on compiler.

NOTE: Left Shift and Right shift of -ve no. is undefined according to C++ standards.

↳ Also since standard does not say anything about internal representation of -ve no., it is not recommend to use (~) operator with -ve numbers.

• Check if k^{th}-bit is set (from Right side)

IP: n = 5, k = 1 → 00..00101 (bit 1 = 1)

OP: Yes

IP: n = 8, k = 2 → 00..01000 (bit 2 = 0)

OP: No

IP: n = 0, k = 3 → 00..000 (bit 3 = 0)

OP: NO

k ≤ no. of bits in binary representation.

Date
DELTA Pg No.

CODE :-

① Using Left Shift

Making a no. that has only Kth bit set

```
int n, k;
cin >> n >> k;
if (n & (1 << (k-1))) {
        cout << "Yes" << endl;
} else {
        cout << "No" << endl;
}
```

→ making Mask using 1

② Using Right Shift

Changing the no. itself

→ Shifting kth it to 1st position then taking disjunction with 1.

```
if (((n >> (k-1)) & 1) == 1)
        cout << "Yes";
else
        cout << "No";
```

- Count Set Bits (non-Negative integers)

I/P: n = 5 → n = (1)0(1) → (2)
O/P: 2

I/P: n = 7 → n = (1)(1)(1) → (3)
O/P: 3

I/P: n = 13 → n = (1)(1)0(1) → (3)
O/P: 3