# ★ BIT MANIPULATION



* In this, we will study program & solve at the bit level.
* Decimal Number System → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
* Binary Number System → 0 & 1.

* Decimal to Binary : (10)

$(1010)_2 = (10)_{10}$ (8 4 2 1)

| | | |
|---|---|---|
| 2 | 10 | 0 |
| 2 | 5 | 1 |
| 2 | 2 | 0 |
| | 1 | 1 |
| | 0 | |

* Binary to Decimal : (100)

$1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

$4 + 0 + 0$

$4$

$(4)_{10} = (100)_2$ (4 2 1)

# In Binary, before 1 any or how much zero are there it's can be excluded.

Ex: 000100 00000010 00101

* Bit-wise Operators : When operations are performed on the bits of computer, then we use Bit-wise operators.

~~Bitwise AND~~ 1. Binary AND &
~~Bitwise~~ 2. Binary OR |
3. Binary XOR ^
4. Binary One's Complement ~
5. Binary left shift <<
6. Binary Right shift >>

1. Binary AND (&) : Ampersand "&".

- 0 & 1 → 0
- 0 & 1 → 0
- 1 & 0 → 0
- 1 & 1 → 1

Ex: A = 0101 B = 0110

```
  0101
  0110
  0100 = (4)10
```

2. Binary OR (|) : Pipeline "|".

- 0 | 0 → 0
- 0 | 1 → 1
- 1 | 0 → 1
- 1 | 1 → 1

Ex: A = 0101 B = 0110

```
  0101
  0110
  0111 = (7)10
```

3. Binary XOR (^) : "^".

- 0 ^ 0 → 0
- 0 ^ 1 → 1
- 1 ^ 0 → 1
- 1 ^ 1 → 0

Ex: A = 0101 B = 0110

```
  0101
  0110
  0011 = (3)10
```

4. Binary One's Complement (~) :
It works on a single operand.

- ~0 → 1
- ~1 → 0

Ex: A = 0101

~A = 1010

* We will pronounce it as "not of A".

* In Binary number, we define two types of bits,
i) Least Significant Bit (LSB)
ii) Most Significant Bit (MSB)

⇒ 0 0 0 0 0 1 0 1
MSB LSB

* If MSB is 0, number is positive.
  Ex: 00000101 ⟶ +5
* If MSB is 1, number is negative.
  Ex: 10000101 ⟶ −5
* Generally, ~ is calculated as :
  ⇒ num → 1's complement (~num) → add 1 → 2's complement

Ex: $00000101$ , $(11111010)_2 \rightarrow (-6)_{10}$

$\pm 1$

$(00000110)_2$ — negative

$(-6)_{10}$

Ex: ~0 → 1

~5 → 00000101

* ① 1's complement (not(n) / ~(n)) ② add 1 — 2's complement

11111010

↓ 1's complement

00000101

↓ +1

$(00000110)_2 \rightarrow (-6)_{10}$ — negative.

Ex: ~0 → 00000000

11111111

↓ 1's complement

00000000

↓ +1

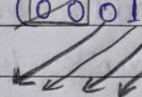
$(00000001)_2 \rightarrow (-1)_{10}$ — negative

5. Binary Left Shift (<<)

* If a << b, then value of 'a' is shifted left by the $b^{th}$ times.

Ex: 5 << 2

$(00\,0\,1\,0\,1\,1)_2$ << 2



$\Rightarrow (0\,1\,0\,1\,0\,0)_2 = (20)_{10}$

* The empty spaces are filled with zero "0".

$\Rightarrow$ $\boxed{a << b = a * 2^b}$

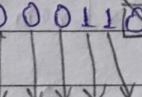
Ex: $5 << 2 = 5 * 2^2 = 5 * 4 = 20.$

6. Binary Right Shift (>>)

* If a >> b, then value of 'a' is shifted right by the $b^{th}$ times.

Ex: 6 >> 1

$(0\,0\,0\,1\,1\,0)_2$ >> 1



$\Rightarrow (0\,0\,0\,0\,1\,1)_2 = (3)_{10}$

* The empty spaces will be be filled with the zero "0".

$\Rightarrow$ $\boxed{a >> b = \frac{a}{2^b}}$

Ex: $6 >> 1 = \frac{6}{2^1} = \frac{6^3}{2} = 3.$

Ques. Check if a number is Even or Odd.

0 → 0 0 0
1 → 0 0 1
2 → 0 1 0
3 → 0 1 1
4 → 1 0 0
5 → 1 0 1

* Notice that,
• odd's have → LSB = 1
& • even's have → LSB = 0.

* LOGIC : We will use AND (&), → num & (001)$_2$ ← bitmask

Ex: (3)$_{10}$ → (011)$_2$

& 001

001 → (1)$_{10}$ → odd

(4)$_{10}$ → (100)$_2$

& 001

000 → (0)$_{10}$ → even

⇒
```
public static void oddOrEven (int n) {
    int bitmask = 1;
    if ((n & bitmask) == 0) {
        //EVEN
        Syso ("It is even number.");
    } else {
        //ODD
        Syso ("It's odd number.");
    }
}
psvm {
    oddOrEven (11);
    oddOrEven (14);
}
```

# * Operations on Binary Operands:

## 1. Get ith bit

Ex: 0 0 0 1 1 1 1 0 , i = 2 ; In code, $i^{th}$ bit

$i^{th}$ bit = 1.

0 1

* num & (1<<i)

Bitmask

* CODING LOGIC, $(15)_{10}$ → 00001111 , → i = 6 → $(00000010)_2$

& 00000100

00000100 ≠ 0 → $i^{th}$ bit = 1.

*** num & (1<<i) → = 0 (0), ≠ 0 (1)

⇒

```
public static int getIthBit(int n, int i) {
    int bitmask = 1 << i;
    if ((n & bitmask) == 0) {
        return 0;
    } else {
        return 1;
    }
}
```

## 2. Set ith bit

Ex: 00001010 , i = $(2)_{10}$

| 00000100 ← (00000100)$_2$ (Set)

00001110

* CODING LOGIC, ⇒ num | (1<<i)

bitmask

⇒ public static int set_i_bit (int n, int i) {
int bitmask = 1 << i;
return n | bitmask;
}

3. Clear ith bit

Ex: 1010 ⟶ i = 1 ⟶ 1000

* LOGIC: (10)$_{10}$ → (1010)$_2$ → 1 << i
i = 1 → ~(1 << i)
⇒ num & (~(1<<i))
Bitmask

→ 1 << i = 0010
→ ~(1<<i) = 1101 ⟹ 1010
& 1101
(1000)$_2$

⇒ public static int ~~set~~ clear_i_bit (int n, int i) {
int bitmask = ~(1<<i);
return n & bitmask;
}

4. Update ith bit

* Value = 0 or 1 will be given.
* And ith position will also be input.
* Firstly, clear ith position & set ith position.
* Input num, ith pos, int newBit (can be 0 or 1.)

LOGIC: clear → nB = 0 , set → nB = 1
nB << i cleared value (nB << i) | n cleared value
0 << i | n
n (no change) ith bit set (change)

⇒ public static int update_i_bit (int n, int i, int newBit) {

```
/* if (newBit == 0) {
       return clear_i_bit (n, i);      // Pg no. → 101
   } else {
       return set_i_bit (n, i);        // Pg no. → 101
   } */
```

OR

```
n = clear_i_bit (n, i);
int Bitmask = newBit << i;
return n | Bitmask;
}
```

* Clear Last i bits

$(15)_{10} = n = (1\,1\,1\,1)_2$, i = 2.

clear last 2 bits

$1\,1\,0\,0 \rightarrow$ i = 2

LOGIC: $\sim 0 = -1 = (1\,1\,1\,1\,0\,0\,0\,0)_2$

⇒ num & (-1 << i)

or

(~0 << i)

⇒ public static int clear_ith_bits (int n, int i) {

```
    int bitmask = (~0) << i;
    return n & bitmask;
}
```

* Clear Range of i & j bits.

n = 1001 1 1 0 1 0 0 1 1, i = 2, j = 7

(7 ... 2)

⇒ 1001000000 11

LOGIC : 1 0 0 1 1 1 0 1 0 0 1 1
7 & 2

1 1 1 1 0 0 0 0 0 0 1 1 → a | b

⇓

1 0 0 1 0 0 0 0 0 0 1 1

a = 1 1 1 1 0 0 0 0 0 0 0 0
b = 0 0 0 0 0 0 0 0 0 0 1 1

∵ a = (~0) << (j+1)

b = (1<<i) − 1
(2^i)

b ⇒ 01 = (1)10 = 2^1 − 1
011 = (3)10 = 2^2 − 1
0111 = (7)10 = 2^3 − 1
01111 = (15)10 = 2^4 − 1
011111 = (31)10 = 2^5 − 1
∴ 2^i − 1

⇒
```
public static int clear_ij_range_bits(int n, int i, int j){
    int a = ((~0) << (j+1));
    int b = (1 << i) -1 ;
    int bitmask = a | b;
    return n & bitmask;
}
```

* Check if a number is a Power of 2 or not

4 → 2^2 ✓
8 → 2^3 ✓
7 → 2^n ×
10 → 2^n ×

4 → 100 & → 0
3 → 011

8 → 1000 & → 0
7 → 0111

16 → 10000 & → 0
15 → 01111

∴ num & (num−1) = 0 ∵ {num = 2^x.

⇒
```
public static boolean isPowerOfTwo (int num) {
    return (num & (num-1)) == 0;
}
```

* **Count Set Bits in a Number.**

• $(10)_{10} = (1010)_2$ ⟶ no. of set bits = 2.

**LOGIC:**

1st iteration: n = 1010 — count = 0

LSB — n >> 1

2nd iteration: n = 0101 — increase → count = 1

LSB — n >> 1

3rd iteration: n = 0010 — unchange → count = 1

LSB — n >> 1

4th iteration: n = 0001 — count = 2

LSB — n >> 1

5th iteration: n = 0000 — count = 2.

if n = 0 then stop.

**Final ans : 2.**

* If n is a number then to represent it in binary bits it takes, $n = \log n$ or $n = \log n + 1$

Ex: 16 → 10000 (5)

$= \log_2 16 + 1 = 4 + 1$

$= 5$

⇒
```
public static int countSetBits (int n) {
    int count = 0;
    while (n > 0) {
        if ((n & 1) != 0) {
            count++;
        }
    n = n >> 1;
    } return count; }
```

# * Fast Exponentiation

Ex: $3^{5} \rightarrow 3^{101}$

$\rightarrow a = 3$

$101 \rightarrow 1\ 0\ 1$ (powers: $4\ 2\ 1$)

$(a^4)(1)(a^1)$

**Normal Exponentiation**

$a^n \rightarrow a.a.a.a \cdots a_n$

$O(n)$

1st → $a = 3$ ; $ans = 1$

$ans = 1 \times 3^1 = 3$

$a = a^2 = 9$

2nd → $a = 9$ ; $ans = 3$

$ans = 3 \times 1 = 3$

$a = a^2 = 81$.

3rd → $a = 81$ ; $ans = 3$

$ans = 3 \times 3^4 = 243$

∴ Ans : $3^5 = 243$

// Time Complexity, ⇒ $O(\log n)$.

⇒

```
public static int fastExpo (int base, int pow) {
    int ans = 1;
    while (pow > 0) {
        if ((pow & 1) != 0) {
            ans = ans * base;
        }
        base = base * base;
        pow = pow >> 1;
    }
    return ans;
}
```

# QUESTIONS

Q1. What is the value of $x \wedge x$ for any value of $x$?
We know that, in XOR ($\wedge$) the same number always gives 0. So, $\Rightarrow \boxed{x \wedge x = 0}$

Q2. Swap two numbers without using any third variable.

```
public static void main (String [] args) {
    int x = 4, y = 10; //(0100)2  //(1010)2
    System.out.println ("Before Swaping: x= "+x+ " y= "+y);

    x = x ^ y ;x=0100 ^ 1010 = 1110
    y = x ^ y ; y = 1110 ^ 1010 = 0100
    x = x ^ y ; x = 1110 ^0100 = 1010

    System.out.println ("After swapping: x= "+x+ " y= "+y);
}
```

Q3. Add 1 to an integer using Bit Manipulation.

→ $-x = \sim x + 1$

→ $-\sim x = x + 1$

```
public static void main (String[] args) {
    int x = 6;
    System.out.println (x + " + " + 1 + " is: " + -~x);
    x = -4;
    System.out.println (x + " + " + 1 + " is: " + -~x);
    x = 0;
    System.out.println (x + " + " + 1 + " is: " + -~x);
}
```

Q4. Convert uppercase characters to lowercase using bits.

```
public static void main (String[] args) {
    for (char ch = 'A'; ch <= 'Z'; ch++) {
        System.out.println ((char) (ch | ' '));
    }
}
```

* We can visit for good read of tricks using bits:
  https://graphics.stanford.edu/~seander/bithacks.html.

* Modular Exponentiation

→ It can be implemented using the "fast exponentiation" algorithm, also known as Exponentiation by squaring.

→ The algorithm calculates the modular exponentiation efficiently, avoiding large intermediate results that could lead to overflow.

→ It uses property :-

⇒ $(a * b) \% p = ((a \% p) * (b \% p)) \% p$

→ It is same as fast exponentiation only difference is that it use "%".

NEXT PAGE

```
public static long mod_expo(long base, long expo, long modulus){
    long result = 1;
    base %= modulus;
    while (expo > 0) {
        if (expo % 2 == 1) {
            result = (result * base) % modulus;
        }
        base = (base * base) % modulus;
        expo /= 2;
    }
    return result;
}
```

* base = 2312
* expo = 3434
* modulus = 6789

Output: 6343