SCALER

Bit Manipulation - 1

TABLE OF CONTENTS

Bit-wise Operators : &, |, ^, ~, <<, >>

Truth Table

a	b	a&b	a|b	a^b	~a/!a
0	0	0	0	0	~0 = 1
0	1	0	1	1	~1 = 0
1	0	0	1	1	
1	1	1	1	0	

a & b : Returns TRUE when both A & B are set.

a | b : Returns TRUE when either A or B is set.

a ^ b : Returns TRUE when both A & B are different

Basic AND Properties

Even / Odd Number → Even no : LSB is 0
Odd no : LSB is 1

1. **A & 1** → 1 (odd)
   0 (even)

```
11 → 1 0 1 1
&
1  → 0 0 0 1
----------------
     0 0 0 1  = 1
----------------
```

```
10 → 1 0 1 0
&
1  → 0 0 0 1
----------------
     0 0 0 0  = 0
----------------
```

2. **A & 0** → 0

```
9  → 1 0 0 1
&
0  → 0 0 0 0
----------------
     0 0 0 0  = 0
----------------
```

3. **A & A** → A

```
9  → 1 0 0 1
&
9  → 1 0 0 1
----------------
     1 0 0 1  = 9
----------------
```

Basic OR Properties

1. **A | 0 → A**

```
11 →  1  0  1  1
 |
0  →  0  0  0  0
-------------------
      1  0  1  1  = 11
-------------------
```

```
10 →  1  0  1  0
 |
0  →  0  0  0  0
-------------------
      1  0  1  0  = 10
-------------------
```

2. **A | A → A**

```
9  →  1  0  0  1
 |
9  →  1  0  0  1
-------------------
      1  0  0  1  = 9
-------------------
```

3. **A | 1 → A (odd)**

A+1 (even)

```
11 →  1  0  1  1
 |
1  →  0  0  0  1
-------------------
      1  0  1  1  = 11
-------------------
```

```
10 →  1  0  1  0
 |
1  →  0  0  0  1
-------------------
      1  0  1  1  = 11
-------------------
```

Bit Manipulation-1

Basic XOR Properties

1. **A ^ 0 → A**

```
11 →  1  0  1  1
^
0  →  0  0  0  0
---------------------
      1  0  1  1  = 11
---------------------
```

```
10 →  1  0  1  0
^
0  →  0  0  0  0
---------------------
      1  0  1  0  = 10
---------------------
```

2. **A ^ A → 0** V. V. V. IMP.

```
9  →  1  0  0  1
^
9  →  1  0  0  1
---------------------
      0  0  0  0  = 0
---------------------
```

Cumulative Property → The order of the operands doesn't affect the result of the operation.

$$A \,\&\, B = B \,\&\, A$$

$$A \mid B = B \mid A$$

$$A \,\hat{}\, B = B \,\hat{}\, A$$

Associative Property → grouping of operands doesn't affect the result of the operation.

$$a \,\&\, b \,\&\, c = (a \,\&\, b) \,\&\, c = a \,\&\, (b \,\&\, c) = (a \,\&\, c) \,\&\, b$$

$$a \mid b \mid c = (a \mid b) \mid c = a \mid (b \mid c) = (a \mid c) \mid b$$

$$a \,\hat{}\, b \,\hat{}\, c = (a \,\hat{}\, b) \,\hat{}\, c = a \,\hat{}\, (b \,\hat{}\, c) = (a \,\hat{}\, c) \,\hat{}\, b$$

Quiz : 1

< *Question- 1* > : Evaluate the expression: **a ^ b ^ a ^ d ^ b**

(a^a) ^ (b^b) ^ d

0 ^ 0 ^ d

0 ^ d = d

ans : d

Quiz : 2

< *Question- 2* > : Evaluate the expression: **1 ^ 3 ^ 5 ^ 3 ^ 2 ^ 1 ^ 5**

(1 ^ 1) ^ (3 ^ 3) ^ (5 ^ 5) ^ 2

0 ^ 0 ^ 0 ^ 2

0 ^ 2 = 2

ans : 2

Left Shift Operator (<<)

- Shifts bits of a no to left by specified no of position

A = 10	0	0	0	0	1	0	1	0	10 = A
A<<1	0	0	0	1	0	1	0	0	20 = A x 2^1
A<<2	0	0	1	0	1	0	0	0	40 = A x 2^2
A<<3	0	1	0	1	0	0	0	0	80 = A x 2^3
A<<4	1	0	1	0	0	0	0	0	160 = A x 2^4
A<<5	0	1	0	0	0	0	0	0	~~320~~ = A x 2^5 64

Range of Unsigned 8 bit : [0 , 255]

Ideally it should be 320 but we are getting 64.

overflow

<< of a no beyond the bit capacity of the datatype can cause OVERFLOW.

$a << n = a * 2^n$

$1 << n = 2^n$

Right Shift Operator (>>)

- Shifts bits of a no to right by specified no of position

A = 10 0 0 0 0 1 0 1 0 → x A=10

A >> 1 0 0 0 0 0 1 0 1 → x 5 = A/2¹

A >> 2 0 0 0 0 0 0 1 0 → x 2 = A/2²

A >> 3 0 0 0 0 0 0 0 1 → x 1 = A/2³

A >> 4 0 0 0 0 0 0 0 0 0 = A/2⁴

$a >> n = a/2^n$

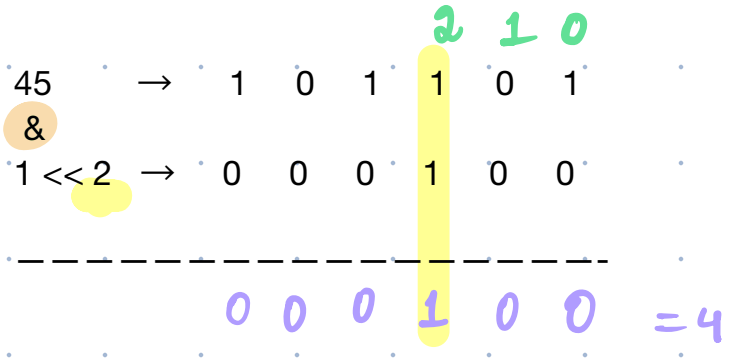
$1 >> n = 1/2^n$

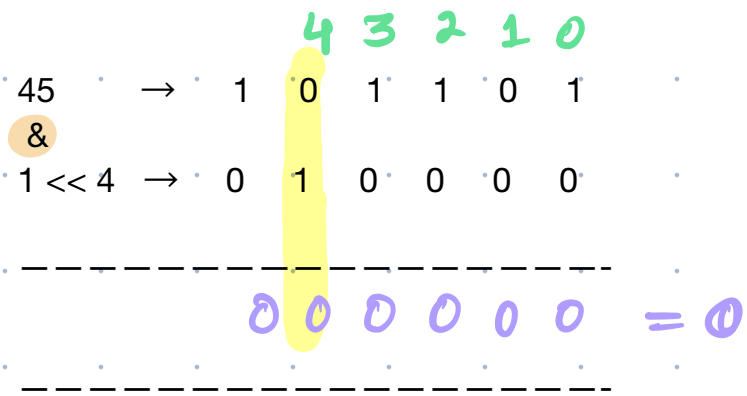
1 << 3

$1 \times 2^3 = 8$

Power of Left Shift Operator

1. AND Operator →





N & (1<<i) : check whether ith bit is SET or not.

2. OR Operator →

```
45      →   1   0   1   1   0   1
|
1 << 2  →   0   0   0   1   0   0
-----------------------------------
            1   0   1   1   0   1
-----------------------------------
```

```
45      →   1   0   1   1   0   1
|
1 << 4  →   0   1   0   0   0   0
-----------------------------------
            1   1   1   1   0   1
-----------------------------------
```

N | (1<<i) : Sets ith bit

3. XOR Operator →

```
45      →   1   0   1   1   0   1
^
1 << 2  →   0   0   0   1   0   0
-----------------------------------
            1   0   1   0   0   1
-----------------------------------
```

```
45      →   1   0   1   1   0   1
^
1 << 4  →   0   1   0   0   0   0
-----------------------------------
            1   1   1   1   0   1
-----------------------------------
```

N ^ (1<<i) : Toggles ith bit

Break till 8:20

< *Question* > : Check whether ith bit is set or not.

```
bool checkbit(int N, int i) {
    if((N & (1<<i))>0) {
        return true;
    }
    else
        return false;
}
```

T.C : O(1)

S.C : O(1)

< *Question* > : Given an integer N. Count the set-bits in N.

Example :

Iterate over all bits & check if it is SET or n't and update count.

```
int countbits (int N) {
    count = 0;
    for (i = 0 ; i < 32 ; i++) {
        if (checkbit(N, i))
            count++;
    }

    return count;
}
```

T.C : O(1)

S.C : O(1)

Issue : Hardcoding so this would work only for 32 bits. Not for other datatypes like LONG etc.

N = 000.....1

For these 31 unset bits loop will run extra. If i can somehow stop my loop for these UNSET bits it will be OPTIMISED.

Approach - 2: **Using Right Shift**

Check 0^{th} bit and right shift no by 1 position.

Keep on doing this till it becomes 0.

N & 1 → 0 (LSB is 0)

N & 1 → 1 (LSB is 1)

		ans
N = 10	1 0 1 0 & 1 = 0	0
N >> 1	0 1 0 1 & 1 = 1	1
N >> 2	0 0 1 0 & 1 = 0	1
N >> 3	0 0 0 1 & 1 = 1	2
N >> 4	0 0 0 0 STOP	

ans : 2

```
ans = 0;
while(n > 0){
    if((n & 1) == 1)
        ans++;
    n = n >> 1;   // n = n/2
}
return ans;
```

T.C : O(logN)

S.C : O(1)

< *Question* > : Unset the ith bit of N if it is **set.**

check if ith bit is SET then unset/toggle it.

```
if (checkbit(N,i) == true) {
    N = N ^ (1<<i);   // Toggle the ith bit
}
```

T.C : O(1)
S.C : O(1)

Scenerio

IRCTC (India's train ticketing system) wants to improve how it shows train options to its users. They've decided that trains which run more **frequently** should appear higher up in the search results. To figure this out, they look at a **28-day period** to see how often each train runs.

Problem

For **each** train, they've come up with a **special number**. This isn't just any number, though. If you were to write it down in binary form (which is like a special code of 0s and 1s), each of the 28 **digits** corresponds to a day in that **period**. A **'1'** means the train runs on that day, and a **'0'** means it doesn't.

Task

Your task is to help **IRCTC** by writing a program. Given a list **A** of these **special numbers** for different **trains**, your program should find the train that runs the most.

Examples

Input 1 : A = $[4369, 8738, 349525]$

Output 1 : $[2]$

Explanation :

Train No. (Index)	Binary Representation	Count Set bits
0	00000000000000010001000000001	3
1	00000000000000010001000000010	3
2	00000001010101010101010101	11

- Clearly the train which is most frequent is **Train 2**. Therefore the output is $[2]$

Input 2: A = $[255, 127, 63]$

Output 2: $[0]$

Explanation :

Train No. (Index)	Binary Representation	Count Set bits
0	00000000000000000000011111111	8
1	00000000000000000000001111111	7
2	00000000000000000000000111111	6

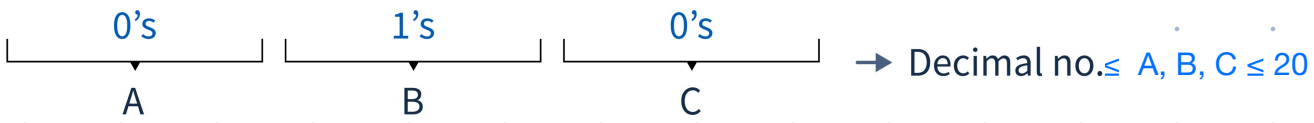
Max count of 1's for each train & return train having Max no of 1's.

```
index ;
max_count = 0 ;

for (i = 0 ; i < N ; i++) {
    current_count = count set bit (arr[i]);
    if (current_count > max_count) {
        max_count = current_count ;
        index = i ;
    }
}
return index ;
```

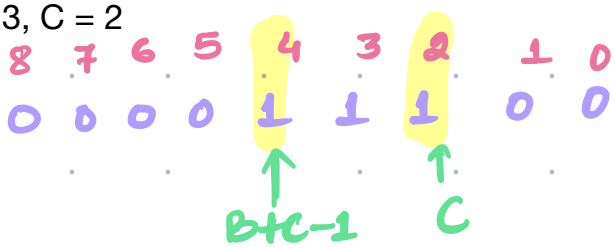
Set Bits in a Range

< *Question* > : Given three integer - A, B, C . It denotes A 0's followed by B 1's followed by C 0's. Return the resulting number.



Example :

A = 4, B = 3, C = 2



ans : 28

A = 1, B = 6, C = 3

ans : 504

Initially my no is 0.

Starting from C ⟶ C+B-1 , set the bits.

```
long ans = 0;
for(i=c; i<= c+B-1; i++){
    ans = ans | (1<<i); // Set bit
}
return ans;
```

$1 \le B, C \le 20$

$B + C = 40$

$i = 40$

$1 << 40$ can't be stored in INT (Overflow)

∴ use LONG.

// Set i^{th} bit = Adding 2^i in the no.

$i = 2$
$i = 3$
$i = 4$
$i = 5$
$3+2-1$
4

A = 4 B = 3 C = 2

ans = 0000 ---- 0000100

64 bits