Number Theory

⇒ Modular Arithmetic

① $(a+b)\%\ m = ((a\%m) + (b\%m))\%\ m$

② $(a*b)\%\ m = ((a\%m) * (b\%m))\%\ m$

③ $(a-b)\%\ m = ((a\%m) - (b\%m) + m)\%\ m$

④ $(a/b)\%\ m = ((a\%m) * (b^{-1})\%m)\%\ m$

~~(b)~~ $(b^{-1}) \Rightarrow$ Multiplicative inverse

Q:- Find factorial of a given no. n.
$0 \le n \le 100$.

print the answer modulo m, where $m = 10^9 + 7$.

Here, the factorial value can be very very large such that it can't even fit in the long long data type.

∴ It is given in questions that report your answer $\%\ (10^9 + 7)$, such that every answer will be less than $10^9 + 7$ & this ~~is~~ no. can be easily fit inside the int data ~~dat~~ type.

Significance of $10^9 + 7$:-

- very close to integer max^M range.
- since, it is a prime no., I can calculate the multiplicative inverse from 1 to (10^9+7)

Hence, It is an ideal choice.

// Code

```
int n;  cin>>n;
int m = 1000000007;
long long fact = 1;
for (int i=2; i<=n; i++){
        fact = (fact * i) % m;
}
return fact;
```

By using the modulo property i.e., we can calculate it at each iteration

for eg → 5! % m

⇒ (1 * 2 * 3 * 4 * 5) % m

= ((1×2×3×4)%m * (5%m))%m

(5) ⇒ 24 × 5 = 120

((1×2×3)%m * (4%m))%m

(4) ⇒ 6 × 4 = 24

((1×2)%m * (3%m))%m

(3) ⇒ 2 × 3 = 6

((1%m) * (2%m))%m

(1) × (2) = 2

Hence, 120 % m = 120 (answer)

⇒ Binary Numbers & Bit Basics

0 → 0
1 → 01
2 → 10
3 → 11
4 → 100
5 → 101
6 → 110
7 → 111

- ~~last~~ LSB of every odd no. is '1'.
- LSB of every even no. is '0'.

→

A	B	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

AND → &
OR → |
XOR → ^
NOT → !
Left shift → <<
Right shift → >>

3 << 2 ⇒ 1100
3 >> 1 ⇒ 01

In general ⇒

$$x << i = x * 2^i$$

$$x >> i = \frac{x}{2^i}$$

We can store a max of 32 bits in an integer data type.

for 4 bits, max^m no. formed can be :-

1 1 1 1

$\Rightarrow 2^4 - 1$

$\therefore$ max^m no. formed by 32 bits is :-

1 1 1 1 1 ..... 1 (32 times) $= 2^{32} - 1$

* Trick to Calculate power of 2 :-

$1 << 0 = 2^0 = 1$

$1 << 1 = 2^1 = 2$

$1 << 2 = 2^2 = 4$

$1 << 3 = 2^3 = 8$

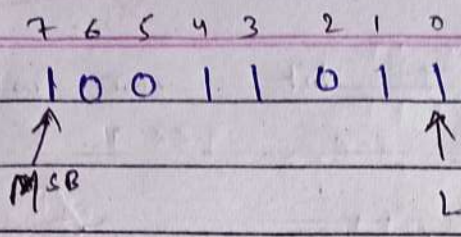
$\boxed{1 << n = 2^n}$

→ Value of INT_MAX = 2147483647

Signed

In integer, we can store $2^{31} - 1$ nos bc one bit is reserved for sign.

But in unsigned integer, we can store $2^{32} - 1$ numbers.



Set bit → If any bit is 1
Unset bit → " " " " 0.

* How to check if any bit is set or unset.

e.g → 0 0 0 1 0 1 1

Check if 3rd bit is set or not.

Take ~~such~~ a no. such that its 3rd bit is set & others are unset.
→ 0 0 0 1 0 0 0

Now, AND both these no's.

⇒ 0 0 0 1 0 1 1
& 0 0 0 1 0 0 0
0 0 0 1 0 0 0
↳ After AND, this bit is 1, i.e, the 3rd bit of original no. is set.
If it is 0, then that bit is unset

* How to find that particular with whom we are ANDing.

That no. is (1 << i) for ith bit.

for checking 1st bit, we will take a no., 1, i.e, 1<<1 = 10

for checking 2nd bit, the no. will be 1<<2
= 100

* If I subtract 1 from a no. which is power of 2, then, I get a no. which consists of all 1's and 1 bit less than the original no.

100	1000	10000
-1	-1	-1
11	111	1111

// Code for checking set bit & not set bit

```
int a=9;
int i=3;
if ((a & (1<< i)) != 0){
    Cout <<"set bit";
}
else {
    Cout <<"Not set bit";
}
```

⇒ 00001001
& 00001000
00001000 → i.e, 3rd bit of 'a' is set bit.

* To set a bit at i^{th} position

```
int a = 7;
int i = 3;
```

// for setting 3rd bit as set, do
a OR (1<<i)

→ 00000111
(OR) 00001000
00001111 ⇒ a | (1<<i)

→ To unset a bit at i^{th} position

∴ ~ (complement operator)
if a = 00001011
then, ~a = 11110100

For unsetting 1 to 0, then find a no. such that all other bits are 1 and that particular i^{th} bit is 0. such that ANDing 'a' and that no. will unset the i^{th} bit.

unset the 3rd bit.
So, the required no. for ANDing is:-
11110111
↑
its 3rd bit is zero

and all other bits are 1, so that other bits will be preserved & 3rd bit will be unset.

→ 0 0 0 0 1 0 1 1

& 1 1 1 1 0 1 1 1

0 0 0 0 0 0 1 1

this bit is unsetted.

How to get this no.

1 << 3 = 0 0 0 0 1 0 0 0

Now, ~(1 << 3) = 1 1 1 1 0 1 1 1

→ To unset a bit, just do:-

a & (~(1<<i))

→ toggle a bit (change i^{th} bit from 0 to 1 or 1 to 0).

- If any bit is 0 → change it to 1.
- " " " 1 → " " " 0.

For toggling any bit, XOR the number with such a no. that its i^{th} bit is 1 & others are zero.

bcz, we know, $0 \wedge 1 = 1$

and $1 \wedge 1 = 0$

So, if the i^{th} bit is 0, XORing it with 1 toggle it to 1 & vice-versa

That particular no. is (1 << i).

a = 9 = 00001001

i = 2 ⇒ 00000100

XOR ⇒ 00001101

→ 2nd bit is toggled from OFF to ON.

a ^ (1<< i) ⇒ For toggling

⇒ // To find bit count

Bit count is the no. of set bits in a binary no.

For every bit, we check if the bit is set or not, then do count++.

```
int count = 0;
for(int i=31; i>=0; i--){
    if ((a & (1 << i)) != 0){
        count++;
    }
}
cout << count;
```

→ Checking for set bit.

// Builtin function for counting set bit

__builtin_popcount(a); ⇒ Works for integers only.

__builtin_popcountll((1ll << 35)-1);

↳ works for long long

// Decimal to binary

Let's say we have to represent the no. in 8 bits binary.

```
for(int i=7; i>=0; i--){
    cout << ((num >> i) & i);
}
```

if num = 10, then binary ⇒ 00001010.

→ Using this method, we can also count the set bits.

~~instead of count~~

```
for (int i=7; i>=0; i--){
    count t = ((num>>i) & 1);
}
```

⇒ // How to check even or odd.

0 0 0 → even
0 0 1 → odd
0 1 0 → even
0 1 1 → odd
1 0 0 → even
1 0 1 → odd
1 1 0 → even
1 1 1 → odd.

Here, we can see that ~~even~~ the 0th bit or LSB of every odd no. is 1 & even no. is 0.

So, by doing (num & 1) we can get either 0 or 1.
0 indicating even no. & 1 is odd no.

```
// if (num & 1) {
        cout << odd ;
   }
   else {
        cout << even;
   }
```

(num & 1) is actually faster than (num % 2 == 0) for checking even or odd.

Basically, / and % operators are slow that bit wise operators.

⇒ Divide by 2:-

We know, $n >> i = \frac{n}{2^i}$,

now, instead of doing $(n/2)$ we can do $(n >> 1)$ i.e, equal to $n/2$.

since, bitwise operators are faster. So use this.

⇒ Upper case & lower case conversion using bit manipulation.

Let's see the pattern b/w uppercase & lowercase letters.

A ⇒ 0 0 0 0 0 | 0 | 0 0 0 0 1
a ⇒ 0 0 0 0 1 | 1 | 0 0 0 0 1

B ⇒ 0 0 0 0 0 | 0 | 0 0 0 1 0
b ⇒ 0 0 0 0 1 | 1 | 0 0 0 1 0

C → 0 0 0 0 0 | 0 | 0 0 0 1 1
c → 0 0 0 0 1 | 1 | 0 0 0 1 1

D → 0 0 0 0 0 | 0 | 0 0 1 0 0
d → 0 0 0 0 1 | 1 | 0 0 1 0 0

E → 0 0 0 0 0 | 0 | 0 0 1 0 1
e → 0 0 0 0 1 | 1 | 0 0 1 0 1

The pattern b/w any uppercase & lowercase letter is → "the 5th bit of uppercase letter is unset while that of lowercase letter is set".

* To convert Uppercase to lowercase, set the 5th bit
* To " lowercase " upper, unset " ".

```
char A = 'A';
char a = A | (1<<5);        ⇒ a
```

set the 5th bit

```
Char x = 'x';
Char X = x & (~(1<<5));     ⇒ X
```

Unset the 5th bit

→ Instead of doing 1<<5, we can find which character is at ASCII value 1<<5.
i.e.

char(1<<5) = ' ' (space).

So, if we OR then upper case character & space, then we get the corresponding lower case letter.

```
Char a = ('A' | ' ');        ⇒ a
```

→ we can't find ~(1<<5) bcz it is out of range of characters.

But we can do a modification here.

We know, C = 'c' & (~(1<<5))

i.e, 0 0 0 0 1 0 0 0 1 1

& 1 1 1 1 1 0 1 1 1 1

Instead of this, we can say that we need 0 0 0 0 1 0 1 1 1 1 . , bcz all other zeroes after the left most '1' will always be going to '0', so rather than taking 1 1 1 1 0 1 1 1 1, we can only take 0 0 0 0 1 0 1 1 1 1 .

∴ And Char value of this no. is underscore '_'

- So, simply AND the lower case letter and '_' (underscore) to get the corresponding upper case letter.

cout << char('c' & '_'); ⇒ C

* To convert upper to lower ⇒ char('A' | ' ').
* To convert lower to upper ⇒ char('a' & '_').

⇒ Clear LSB's

n = 000001111011

let's say upto 4th bit, ~~we~~ from LSB, we have to clear all bits.

To do so,

000001111011

& 111111100000 → I need to AND ~~with~~ n with this no.

To get this no.,
I can invert 000000011111

we know, $2^i - 1$ will a no. which has all bits 1 & has a total of $(i-1)$ bits as '1'.

So, to get 11111 we can do $2^5 - 1$
i.e, $(1 << 5) - 1$.

i.e, 000000011111 = $(1 << 5) - 1$
and, 111111100000 = $\sim((1<<5) - 1)$

we have to AND this no. with n, to clear all LSB's upto 4th bit.

So, to clear upto ith bit from LSB :-

$$(n \;\&\; (\sim(1 << (i+1)) - 1))$$

if $i = 4$, and $n =$ 00000111011

the o/p will be → 00000100000

all bits upto 4th bit has cleared.

⇒ Clear MSBs

Clear all the MSB's upto ith bit.

Let $i = 3$,

$n =$ 00000111011

clear all these to zero

So, I need to do AND ^ 00000001111

Such that:- 00000111011

& 00000001111

⇒ 00000001011 → Required answer

How to get 00000001111

Same as before → 00000001000 − 1

i.e, (1 << 4) − 1

i.e, (1 << (i+1)) − 1

So, the required answer will be :-

$$n \;\&\; \left(\left(1 << (i+1)\right) - 1\right)$$

⇒ ~~the~~ Check for any number if it is a power of 2 or not

Any power of 2 will be like this :-

n = 00100000

If we do AND with (n-1), then it will be always zero, bez, (n-1) will ~~be~~ be like :- 00011111

n ⇒ 00100000

& (n-1) ⇒ 00011111

00000000 ⇒ if it is zero, then 'n' is power of 2, else not.

```
// if (n & (n-1)){
    cout << " Not power of 2";
}
else {
    cout << " Power of 2";
}
```

⇒ Swap 2 no.s using XOR

* Important property of XOR

$x \wedge x = 0$

$x \wedge 0 = x$

* XOR is associative:-

$x \wedge (y \wedge z) = y \wedge (x \wedge z) = z \wedge (x \wedge y)$

* XOR is commutative:-

$x \wedge y = y \wedge x$

eg → a = 4, b = 6.

Step 1 → a = a ^ b

Step 2 → b = b ^ a // b = a

↳ // b = b ^ (a ^ b)

= b ^ b ^ a

= 0 ^ a

= a

Step 3 → a = a ^ b

↳ a = (a ^ b) ^ a

= a ^ b ^ a = a ^ a ^ b

= 0 ^ b = b

// a = b

Hence, two values has been swapped.

Q:- Given array A of n integers. All integers are present in even count except one. Find that one integer which has odd count in O(N) time complexity & O(1) space.

Sol→ Let's suppose the array is:-

2 4 6 7 7 4 2 2 2

Here, we can see that 2, 4, & 7 have even no. of count in the array except 6.

If we XOR all the elements, then the paired elements will cancel out each other (as $x \wedge x = 0$) & at last 6 will be the answer.

// code

```
int n; cin >> n;  int x;
int ans = 0;
for (int i=0; i<n; i++){
    cin >> x;
    ans ^= x;
}
cout << ans;
```

T ⇒ O(n)

S ⇒ O(1)

⇒ Bitmasking

Suppose there are 4 fruits.

Apple → 0
Orange → 1
Banana → 2
Lichi → 3

} Representation of fruits by no's

There are 3 children (let say).

Child - 1 :→ 2, 3
Child - 2 → 0, 1, 2
Child - 3 → 1, 3

→ Represents which fruit who have ~~is present~~

If we store the choices of fruits of different children in different arrays then it is time & space taking.
furthermore, if we are to find the union & intersection, then it take O(n) time.
To reduce this space & time, we use bit mask.

Child - 1 ⇒ 2, 3.
For the preferences 2 & 3, we can generate such a binary no., so that its 2nd & 3rd bit are set.
i.e, 1100

Child - 2 ⇒ 0, 1, 2 → 0111
Child - 3 ⇒ 1, 3 → 1010

1 represents the preference of child.

Now, if we ~~need~~ with to find intersection, i.e, common fruits b/w child 1 & 2, the simply do AND operation b/w child-1 & 2.

i.e, 1100
& 0111
0100 ⇒ It is clear that the 2nd fruit is common b/w child-1 & child-2.

If we want to find union, then do OR.

1100
OR & 0111 Time ⇒ O(1)
1111 Space → O(1)

But this technique only works for small ranges.
bcz range of integer = 32 bit
" " long long = 64 bit.

we can't represent more than 64 items using bit mask.

Q:- There are N <= 5000 workers. Each worker is available for some days of this month (30 days). For each worker, given a set of no's in the interval [1,30] representing his/her availability. Find two workers that are best for the job - maximize the no. of days when both these are available.

→ Input:- n ⇒ no. of workers

days → for each worker, how many days he/she is working

a[days] → array representing on which days he/she is working

```
int n;
cin >> n;
vector<int> masks(n, 0);   // This vector will store the bit mask value of each day on which the worker is present.

for(int i=0; i<n; i++){
    int num_workers;
    cin>> num_workers;
    int mask = 0;
    for(int j=0; j< num_workers ; j++){
        int day;
        cin>> day;
        mask = (mask | (1 << day));
    }
    mask[i] = mask;
}
```

This loop will convert every day input into bit-mask.

```
int max_days = 0;
int person1 = -1, person2 = -1;
for (int i=0; i<n; i++){
    for (int j = i+1; j<n ; j++){
        int intersection = (masks[i] & masks[j]);
        int common_days = __builtin_popcount(intersection);
        if (common_days > max_days)
        {
            max_days = common_days;
            person1 = i;
            person2 = j;
        }
    }
}
cout << person1 << " " << person2 << " " << max_days << endl;
```

This will output the two persons whose intersection is max. i.e. how many and are common days b/w them.

⇒ Subset Generation b/w bitmasking

Let suppose an array ⇒ [2, 3, 7, 8, 9]

To represent this array as a bitmask, we need 5 bits, bc these are 5 elements & each index can be represented by a bit.

If the choose the subset :-
[3, 8]

then it can be represented by 01010. In this bit mask, 3rd & 1st bit are set, which means the 3rd & 1st index elements of the array are present in the subset.

e.g→ arr = [2, 4, 5]

Total no. of subset = 2^n (n ⇒ size of array)

$2^3 = 8$ subsets

[ ], [2], [4], [5], [2, 4], [2, 5], [4, 5], [2, 4, 5]

These 8 subsets can be represented by 000 - 111, where the set bit represent that index of array of which element is present in subset.

0 → 0 0 0	[ ]	
1 → 0 0 1	[2]	// ek aisa subset jisme 0th index of array present ho, i.e, arr[0].
2 → 0 1 0	[4]	
3 → 0 1 1	[2, 4]	
4 → 1 0 0	[5]	
5 → 1 0 1	[2, 5]	// similar all bit masks
6 → 1 1 0	[4, 5]	
7 → 1 1 1	[2, 4, 5]	

// code (input = [2, 4, 5] ⇒ (arr).)

```
int n = arr.size();
int subset_count = (1 << n);        // 2^n subsets.
vector<vector<int>> subsets;

for (int i=0; i< subset_count ; i++) {
    vector <int> v;
    for (int j=0; j<n; j++){
        if( (i & (1 <<j)) != 0) {
            v.push_back(arr[i]);
        }
    }
    subsets.push_back(v);
} return subsets;
```

We have 8 possibilities 000 to 111.
for every possibility, we are checking that which bit is set, & if ith bit is set, then push the arr[i] in the vector. (i.e, subset is created)
Now, push this subset into the 2-D array
ie, array of subsets.

$$T. = O(n \times 2^n)$$

⇒ GCD and LCM

12 → $2^2 \times 3^1$
18 → $2^1 \times 3^2$

GCD = Minimum of powers of prime factors.
= $\min(2^2, 2^1) * \min(3^1, 3^2)$
= $2 \times 3 = 6$

HCF = maximum of powers of prime factors
= $\max(2^2, 2^1) * \max(3^1, 3^2)$
= $2^2 \times 3^2 = 36$

GCD * HCF = product of numbers

→ GCD by long division method / Euclid's Algo

e.g → 18 & 12

```
12 ) 18 ( 1
     12
     ----
     (6) ) 12 ( 2
           12
           ----
            0
```

↙ GCD

e.g → 13 & 7

```
7 ) 13 ( 1
    7
    ----
    6 ) 7 ( 1
        6
        ----
        (1) ) 6 ( 6
              6
              ----
              0
```

↙ GCD

* Continuously divide until remainder becomes zero. If rem. is not zero then make the rem. as divisor & previous divisor as dividend & continue further. The final divisor will be the GCD

// Code

```
int gcd (int a, int b) {      // a → dividend, b → divisor

    // If the remainder is zero, then return
       divisor, else we can say, if divisor
       is zero, then return dividend.
    // otherwise, in the next division, the dividend
       will be the prev. divisor, i.e, 'b' and
       divisor will be the remainder of current
       dividend & divisor, ie, a%b
```

```
if (b == 0){
    return a;
}
else
return gcd (b, a%b);
}
```

LCM ⇒ (a*b)/ gcd (a, b);

Inbuilt function for GCD ⇒ __gcd(a,b);
T⇒(log n)
⇒

gcd of three nos ⇒ __gcd(__gcd(a,b), c);

* find min^m fraction of a & b.

eg → $\frac{12}{18}$ = $\frac{2}{3}$ is the min^m fraction

So, to achieve this, divide a & b by their gcd.

i.e, $\frac{a}{b} = \frac{a/gcd(a,b)}{b/gcd(a,b)}$

Bit manipulation

Q:- Given n, print the XOR of all no's b/w 1 to n.

let n = 8

	XOR
n = 1	1
n = 2	3
n = 3	0
n = 4	4
n = 5	1
n = 6	7
n = 7	0
n = 8	8

(0^1)
(1^2)
(1^2^3)
(1^2^3^4)
(1^2^... ^8)

There is pattern of 4 groups

if (n % 4 == 0)
then answer will always be n.
i.e, for 4, 8, 12, 16. ---

if (n % 4 == 1)
answer = ~~(n+1)~~ 1
clearly we can see, for n=1, 5, 9, ---.
ans = 1.

if (n % 4 == 2)
ans = n+1

if (n % 4 == 3)
ans = 0

T = O(1)

Q:- Given a range [L, R], print the XOR of all elements., i.e, XOR(L ^ L+1 ^ L+2 ^ ... R-1 ^ R)

e.g → [2, 4]
i.e, 2 ^ 3 ^ 4 = 5

[3,6] = 3 ^ 4 ^ 5 ^ 6

now XOR[3,6] = XOR[1,6] ^ XOR[1,2]

= (1^2^3^4^5^6) ^ (1^2)

Cancel out

=

i.e, XOR[L,R] = XOR[1,R] ^ XOR[1,L-1]

(find XOR of all no. upto R)

(XOR of all no.s upto L-1)

⇒ O(1) sol^n

To cancel the no.s upto (L-1), we need to XOR them with (1-R) bcz, XOR of same no.s cancel out each other.

Q:- Remove the last set bit.

e.g → 1 1 0 1 1 0

last set bit

After 1st operation, it will be:- 1 1 0 1 0 0

" 2nd " " " " :- 1 1 0 0 0 0

for doing so, just perform:-

| n & n-1 |

e.g → n = 13 = 1 1 0 (1) → last set bit

n → 1 1 0 1

& n-1 → 1 1 0 0

1 1 0 0

⇒ Count the no. of set bits :-

(1st method)

```
while (n != 0){
    if (n & 1 == 1){
        Count++;
    }
    n >> 1;
}
Cout << count;
```

e.g. → n = 13
= 1101

```
1101
 & 1
----
0001   → count = 1

n >> 1 = 0110

0110
 & 1
----
0000

n >> 1 = 0011

0011
 & 1
----
0001   → count = 2

n >> 1 = 0001

0001
 & 1
----
0001   → count = 3

n >> 1 = 0000
```

no. of set bits = 3

T = O(MSB)
i.e, O(position of MSB bit)
in this case O(4)

(2nd method)

Turn off the set bits one by one & each time do Count++.

```
1101 → Turn off   count = 1
→ 1100            count = 2
   Turn off
→ 1000            count = 3
   Turn off
⇒ 0000 (stop)
```

// code

```
while (n != 0){
    n = n & (n-1);
    Count++;
}
Cout << count;
```

T = O(no. of set bits)

There is not much difference in time complexity for both methods. but the 2nd method is slightly optimal for certain cases.

Q:- Given an array. All integers appears twice except two which appears once. Print that two no's.

arr = [2, 1, 2, 5, 1, 4, 4, 7, 3, 3]

The xor of all elements will be :-

XOR = 2 ^ 1 ^ 2 ^ 5 ^ 1 ^ 4 ^ 4 ^ 7 ^ 3 ^ 3

= 5 ^ 7 ⇒ 1 0 1
1 1 1
0 1 0

= (2)

~~This 2 indicates that~~

(The binary of 2 is 010.
It's 1st bit is set, which indicates that the 1st bit of any one is 0 & that of other is 1, so that their ~~zero~~ xor is 1.)

i.e, the ~~two~~ XOR '010' is due to 5 & 7

5 (1st bit is 0)	7 (1st bit is 1)
1	2
5	2
1	7
4	3
^ 4	^ 3
5	7

Now, divide the array into 2 parts such that 1st half will have those no's whose 1st bit is zero and 2nd half will have those no.s whose 1st bit is 1. And XOR of those two halves individually will give the two no's that appears once.

It's bcz if any no. comes under any half then its duplicate will also comes in that half, whose xor will be ultimately 0. & finally that no. will be the ans that appears once.

// Code

```
xor = 0;
for (i=0 → n){
    xor = xor ^ a[i];
}

count = 0;          // It will store the index of 1st set bit of xor
while(xor != 0){
    if (xor & 1) {
        break;
    }
    count++;
    xor >>= 1;
}

// Now divide the arr into two parts
xor1 = 0 , xor2 = 0;
for(i=0 → n){
    if (a[i] & (1<< count)) {
        xor1 = xor1 ^ a[i];
    }
    else {
        xor2 = xor2 ^ a[i];
    }
}
print (xor1, xor2);
```

If the (count)th bit is set then put a[i] in 1st half & take xor of all them else put a[i] in 2nd half & xor them all.

T = O(n)
Space = O(1).

Q:- Given n integers, print the XOR of all subsets.

arr = [1, 3, 2]

{ } → 0
{1} → 1
{3} → 3
{2} → 2
{1, 3} → 2
{1, 2} → 3
{3, 2} → 1
{1, 2, 3} → 0

XOR = 0

1 ^ 2 = 3
2 ^ 3 = 1
3 ^ 1 = 2
Cyclic

1 ^ 2 ^ 3 = 0

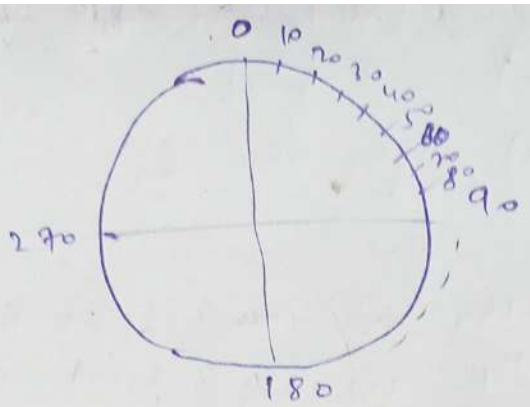
The answer will be always zero for every array bcz there are 2^n subsets of n no's & in this 2^n subsets, every no. will appear even no. of times whose XOR ultimately be 0, except for the array that has only one element...

Q:- There is a lock which has a scale of 360°. & pointer at 0. You have to rotate the lock's wheel exactly n times. & ith rotation should be a_i degrees. either in clockwise or anticlockwise, and after all rotations the pointer should point at zero. If it is possible, print Yes, else No.

Constraints:-

$1 \le n \le 15$

$0 \le a_i \le 180$



e.g → n = 3
[10, 20, 30]
if we do 10° clockwise
20° clockwise
& then 30° AC
then we reach at
0° again. Hence print Yes

* For any given pattern of angles, one thing we can say that if the sum is divisible by 360, then it will always to 0° after n rotations.

* We have to find the sum of array by either rotating clockwise (+) or AC (-).

Here, n = 15, so we can use subset generation using bit mask technique.

for eg → arr = [10, 20, 30]

no. of subsets = $2^3 = 8$.

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

1 ⇒ (+)
0 ⇒ (-)

0 0 0 represents (-10 -20 -30)
= -60
which is not equal to zero.
By doing similarly, for every subset, we will reach 1 1 0, where (10 + 20 - 30) = 0
& hence the answer is Yes

The subsets will generate the sum for every possibility.

// Pseudo code

```
for (i = 0 to (2^n - 1)) {        // For 2^n subsets.   (loop runs)
    sum = 0;
    for (bits = 0 to (n-1)) {     // Checks for every bit i'e is set or not.
        if (i & (1 << bits)) {    // if set (ie, 1) the add, otherwise subtract
            sum += a[i];
        }
        else {
            sum -= a[i];
        }
        if (sum % 360 == 0)
            cout << "yes";
            break;
    }
}
```

// This power set method works for $n \leq 16, 17, 18$. bcz after that the time limit exceeds.

⇒ Bitmask:

If we have add or remove some elements such that no duplicates will be stored, so we need a set DS., i.e, will do in log(n) Time & space complexity.

To do it in O(1) complexity, we will use Bit mask.

But bit mask will store values upto 63 only bcz in long long (maxm data type), there will be 64 bits only (0 - 63).

add (5) → for adding 5, we will set the 5th bit
add (1) → set 1st bit
add (5) → " 5th "
add (3) → " 3rd "
remove (5) → unset 5th bit
print → print the indexes of set bits

long long x = 0;
add (5) → x | (1 << 5) → 00100000
add (1) → (x | (1<<1)) → 00100010
add (5) → x | (1<<5) → 00100010
add (3) → x | (1<<3) → 00101010
remove (5) → x & (~(1<<5)) → 00001010

print →
```
for (i → 0 to 63){
    if (x & (1<<i)){
        print (i)
    }
}.
```

// output = 1, 3.