BIT Manipulation

By- Dharmendra Kumar Yadav

INDEX

Bit Manipulation

&, |, ~, !, >>, <<, XOR

1. The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.
5. The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.
6. The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it.

(a) int a = 5, b = 9

AND

```
  00000101
& 00001001
----------
  00000001   = 1
```

(b) OR

```
  00000101
| 00001001
----------
  00001101   = 13
```

(c) XOR

```
  00000101
^ 00001001
----------
  00001100   = 12
```

Reference - GFG

(d) ~ operator :- compliment

```
int a = 5;
```

// By default it is signed integer, stored in 2's compliment manner having range from $-(2^{n-1})$ to $+(2^{n-1}-1)$.

By default int size is 32 bits so, it will have range of -2^{32-1} to $2^{32-1}-1$

So a = 0 00000000 . 00000000 00000000 00000101

Sign bit → 8 zeroes . 8 zeroes 8 zeroes

~a = 1 11111111 11111111 11111111 11111010

NOTE :- MSB in signed integers represents the sign of the number.
0 = Positive Number, 1 negative number.

→ since after negative, MSB bit is 1, so the number is negative, Now to find the exact value of it, just do 2's complement

2's complement of ~a = 00000000 00000000 00000000 00000110
= 6

→ Since, it is negative, ~a = −6.

b) Take unsigned int a = 5

→ Unsigned int ranges from 0 to 2^n-1.

So a = 00000000 00000000 00000000 00000101

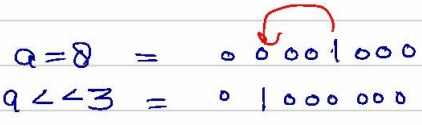
8 zeroes 8 zeroes 8 zeroes

~a = 11111111 11111111 11111111 11111010

since, we are using unsigned int, so ~a will be the decimal representation of it.

~a = 4294967290,

e:) left shift :-



∴ left shift does nothing but multiply the number by 2, K times.

$$a = a \times 2^3 = 8 \times 8 = 64.$$

f:) Right shift :-

∴ Right shift divides the number by 2, K times, where K is the number of right shift.

$$a = \frac{a}{2^k} = \frac{8}{2^3} = \frac{8}{8} = 1.$$

https://yadavdharm.wordpress.com/

```c
// C Program to demonstrate use of bitwise operators
#include <stdio.h>
int main()
{
    // a = 5(00000101), b = 9(00001001)
    int a = 5, b = 9;

    // The result is 00000001
    printf("a = %d, b = %d\n", a, b);
    printf("a&b = %d\n", a & b);

    // The result is 00001101
    printf("a|b = %d\n", a | b);

    // The result is 00001100
    printf("a^b = %d\n", a ^ b);

    // The result is 11111010
    printf("~a = %d\n", a = ~a);

    // The result is 00010010
    printf("b<<1 = %d\n", b << 1);

    // The result is 00000100
    printf("b>>1 = %d\n", b >> 1);

    return 0;
}

```

∴ The values will be output in 2's complement manner, since we are using %d to represent them.

Q 1.)Check if the ith bit is set or not

$N = 13$, $i = 3^{rd}$ bit

binary of $= (1\ 1\ 0\ 1)_2$ → 1st bit

We can take mask 0100 and perform & operation, we will get ith bit.

mask = 1<<(i-1).

$$= \begin{array}{r} 1\ 1\ 0\ 1 \\ 0\ 1\ 0\ 0 \\ \hline 0\ 1\ 0\ 0 \end{array} = 4$$

4 != 0, so ith bit is set,
if ans = 0, ith bit is not set.

```
1 void isKthBitSet(int n, int k)
2 {
3     int mask = 1 << (k - 1);
4     if (n & mask)
5         printf("SET");
6     else
7         printf("NOT SET");
8 }
```

Q 2.) Set the ith bit of a number.

$N = 13$, $i = 2$

$= (1\ 1\ 0\ 1)_2$

Take mask = 0010 and perform Bitwise |.

mask = 1<<(i-1)

$$= \begin{array}{r} 1\ 1\ 0\ 1 \\ 0\ 0\ 1\ 0 \\ \hline 1\ 1\ 1\ 1 \end{array} = 15$$

```
1 int setKthBit(int n, int k)
2 {
3     int mask = 1<<k-1;
4     return (mask | n);
5 }
```

Q 3.) clear the ith bit of a number.

$N = 13$

$= 1101$, $i = 4^{th}$

mask = 1 << i-1

= 1000

~mask = 0111

= 1101

& 0111

0101 = 5

```c
int clearBit(int n, int k)
{   int mask =  (1<<k-1);
     mask = ~mask;
     return (n & mask);
}

int main()
{
    int n = 5, k = 1;
    printf("%d\n", clearBit(n, k));
    return 0;
}
```

Q 4.) Remove the last set bit of a number. (Rightmost bit)

N = 214

= 1 1 0 1 0 1 1 0

Output = 212

N = 1 1 0 1 0 1 1 0

& N-1 = 1 1 0 1 0 1 0 1

Ans = 1 1 0 1 0 1 0 0

① 13 = 1101

12 = & 1100

1100

② 8 = 1000

7 = 0111

0000

```c
int fun(unsigned int n)
{
    return n & (n - 1);
}
```

Find the position of rightmost set bit:

(n & ~(n-1) → will return the binary number containing the rightmost set bit as 1.

→ we can do log of above binary number and we will get position of rightmost set bit.

Q 5.) Find whether a number is even or odd

There are four ways to find whether a number is even or odd:

1. Using Mod Operator(%)
2. Using Division Operator(/)
3. Using Bitwise AND operator(&)
4. Using Left shift and right shift operator(<<,>>)

1. Using Mod Operator:

It is the most used method,
ex - 7 % 2 = 1] if remainder is 1 means odd.
6 % 2 = 0] if remainder is 0 means even.

```cpp
#include <iostream>
using namespace std;

void find_even_odd_using_mod(int num){
    if(num%2==1){
        cout<<"Number is odd.";
    }
    else {
    cout<<"Number is even.";
    }
}

int main() {
   int num;
   cin>>num;
   find_even_odd_using_mod(num);
}

```

2. Using Division Operator:

→ first divide and multiplied the number by 2, if the answer is same as number, then it is even else odd.

ex - num = 9

$\left(\frac{9}{2}\right) \times 2 = 8$

$8 \neq 9$ → odd

num = 16

$\left(\frac{16}{2}\right) \times 2 = 16$

$16 = 16$ → even

```cpp
#include <iostream>
using namespace std;

void
find_even_odd_using_divisionandmul(int
num){
    int calcnum = (num/2) * 2;
    if(calcnum==num){
        cout<<"Number is even.";
    }
    else {
    cout<<"Number is odd.";
    }
}

int main() {
   int num;
   cin>>num;
  find_even_odd_using_divisionandmul(num);
}

```

3. Using Bitwise AND Operator:

ex - num = 10

```
 1010
&0001
 0000 → even
```

num = 9

```
 1001
&0001
 0001 → odd
```

Bit 1	Bit 2	&
0	0	0
0	1	0
1	0	0
1	1	1

even : if n&1 == 0
odd : if n&1 == 1

```cpp
#include <iostream>
using namespace std;

void find_even_odd_using_bitwiseand(int num){

    if(num & 1){
        cout<<"Number is odd.";
    }
    else {
    cout<<"Number is even.";
    }
}

int main() {
   int num;
   cin>>num;
   find_even_odd_using_bitwiseand(num);
}
```

4.) Using Left shift and Right Shift operator (<<,>>)

→ It is same as method 2, here we will do operations on bits than number.
→ for division, use Right shift,
→ for multiplication, use Left shift.

→ So combined operation will be :- (num >>1) <<1
→ Num will be even if ((num >>1)<<1) == num, else odd.

```cpp
#include <iostream>
using namespace std;

void find_even_odd_using_leftandright(int num){
    int calcnum = (num>>1)<<1;
     if(calcnum==num){
         cout<<"Number is even.";
     }
     else {
     cout<<"Number is odd.";
     }
}

int main() {
    int num;
    cin>>num;
    find_even_odd_using_leftandright(num);
}
```

Q 6.) Check if the number is a power of 2?

N = 8

1000

& N-1 = 0111

ans = 0000

→ if ans = 0, it is power of 2.

```cpp
#include<iostream>
using namespace std;

bool powerof2(int n)
{   int mask = n-1;
    return !(n & mask);
}

int main()
{
    int n;
    cin>>n;
    if(powerof2(n)){
        cout<<n<<" is power of 2";
    }
    else {
       cout<<n<<" is not a power of 2";
    }
    return 0;
}

```

Q 7.) Check if a number is a power of 4:

Approach 1:

Divide the Number by 4 untill we get 1, if the remainder after modulus is not 0, that means it is not a power of 4,
also if the remainder is 0, that doesn't mean, it is a power of 4, example - 8 is not a power of 4.
So if the remainder is 0,divide it by 4,untill n does not equals to 1 or modulus of n!=0.

$$TC = O(\log_4 N)$$

```cpp
#include<iostream>
using namespace std;

bool isPowerOfFour(int n){
    if(n == 0)
        return 0;
    while(n != 1)
    {
    if(n % 4 != 0)
        return 0;
        n = n / 4;
    }
    return 1;
}

int main()
{
int number;
cin>>number;
if(isPowerOfFour(number))
    cout<<number<<" is a power of 4";
else
    cout<<number<<" is not a power of 4";
}

```

Approach 2 :

If a given number is a power of 2 and its only set bit is present at even position like 0,2,4,6,8, then it will also be power of 4.

We can check power of two by doing bitwise and of N and (N-1).

To check the position of set bits, we will create a mask, now understanding the pattern of bits is really important to understand mask.

→ first, check if it is a power of 2 or not.

→ If we do bitwise AND of n with mask, and negat it, if result after negation is 1, then it is power of 4, otherwise not.

→ Now to get the result as 1, bitwise AND should be 0, means we need to create a mask such that Bitwise AND of n and mask will be 0.

→ lets take an exampk : N=16.

N = 16 ⇒

	16	8	4	2	1
	1	0	0	0	0
&	0	1	0	1	0
	0	0	0	0	0

N = 5

	8	4	2	1
	0	1	0	1
&	1	0	1	0
	0	0	0	0

1 and 4 are powers of 4.

→ If you clearly observe the pattern, if in mask, those who are power of 4, if we set those bits as 0, rest as 1, we can create our mask.

→ Same pattern will by followed to check power of 8 and 16.

→ Int generally is of size 4bytes = 32 bits, so if you create a mask of 32 bits, it will be

1010 1010 1010 1010 1010 1010 1010 1010

A A A A A A A A

= 0xAAAAAAAA

↳ Hexadecimal

```cpp
#include <iostream>
using namespace std;

bool checkPowerOf4(int n){
    if(n==0) return false;
    bool powerof2 = !(n & (n - 1));

    //for mask, bits will be 0, in the place which are power of 4
    //bits will be 1, in the place which are not power of 4
    //A = 1010, 1 and 4 are powers of 4, so mark yhem as 0.
    //every even bit will be a power of 4, so mark them as 0.
    bool mask = !(n & 0xAAAAAAAA);
    return  powerof2 && mask;
}

int main(){
     int n;
    cin>>n;
    if (checkPowerOf4(n)) {
        cout << n << " is a power of 4";
    }
    else {
        cout << n << " is not a power of 4";
    }
    return 0;
}
```

Approach 3 :

N is a power of 4, if it is a power of 2 and if modulus of n by 3 will give remainder 1.

```cpp

#include <iostream>
using namespace std;

bool checkPowerOf4(unsigned n){
    return !(n & (n - 1))&& (n % 3 == 1);
}

int main()
{
    unsigned n;
    cin>>n;

    if (checkPowerOf4(n)) {
        cout << n << " is a power of 4";
    }
    else {
        cout << n << " is not a power of 4";
    }

    return 0;
}
```

Q 8.) Check if a number is power of 8:

Approach 1:

Divide the Number by 8 untill we get 1, if the remainder after modulus is not 0, that means it is not a power of 8,
also if the remainder is 0, that doesn't mean, it is a power of 8, example - 16 is not a power of 8.
So if the remainder is 0,divide it by 8,untill n does not equals to 1 or modulus of n!=0.

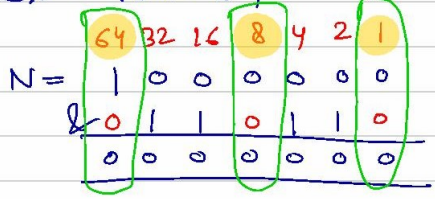
```cpp
#include<iostream>
using namespace std;

bool isPowerOfEight(int n){
    if(n == 0)
        return 0;
    while(n != 1)
    {
    if(n % 8!= 0)
        return 0;
        n = n / 8;
    }
    return 1;
}

int main()
{
  int number;
  cin>>number;
  if(isPowerOfEight(number))
      cout<<number<<" is a power of 8";
  else
      cout<<number<<" is not a power of 8";
}

```

Approach 2:

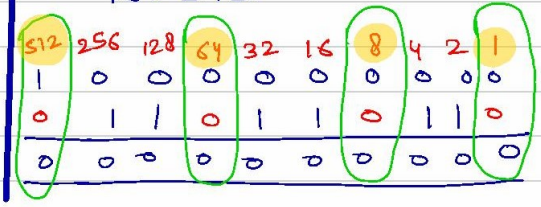
→ Same as Approach 2 of previous question, just the mask will change.

NOTE :- To create mask, those bits who are power of 8, set them as 0, rest as 1.

ex- N = 64



N = 512



→ If we clearly observe, we will find that every 3rd bit in the mask, starting from 0th position, is the power of 8, so mask will

1011 0110 1101 1011 0110 1101 1011 0110
B 6 D B 6 D B 6

```cpp
#include <iostream>
using namespace std;

bool checkPowerOf8(int n){
    if(n==0) return false;
    bool powerof2 = !(n & (n - 1));

    //for mask, bits will be 0, in the place which are power of 8
    //bits will be 1, in the place which are not power of 8
    //0110110, 1,8,64 are powers of 8.
    /*starting from 0th position every third bit will be a power of 8,
    so mark them as 0.
    (0xB6DB6DB6)16 = (10110110110110110110110110110110)2
    */

    bool mask = !(n & 0xB6DB6DB6);
    return  powerof2 && mask;
}

int main(){
     int n;
    cin>>n;
    if (checkPowerOf8(n)) {
        cout << n << " is a power of 8";
    }
    else {
        cout << n << " is not a power of 8";
    }
    return 0;
}
```

Approach 3:

N is a power of 8, if it is a power of 2 and if modulus of n by 7 will give remainder 1.

```cpp

#include <iostream>
using namespace std;

bool checkPowerOf8(unsigned n){
    return !(n & (n - 1))&& (n % 7 == 1);
}

int main()
{
    unsigned n;
    cin>>n;

    if (checkPowerOf8(n)) {
        cout << n << " is a power of 8";
    }
    else {
        cout << n << " is not a power of 8";
    }

    return 0;
}
```

Q 9.) Check if a number is a power of 16:

Approach 1:

Divide the Number by 16 untill we get 1, if the remainder after modulus is not 0, that means it is not a power of 16,
also if the remainder is 0, that doesn't mean, it is a power of 16, example - 32 is not a power of 16.
So if the remainder is 0,divide it by 16,untill n does not equals to 1 or modulus of n!=0.

```cpp
#include<iostream>
using namespace std;

bool isPowerOfSixteen(int n){
    if(n == 0)
        return 0;
    while(n != 1)
    {
    if(n % 16!= 0)
        return 0;
        n = n / 16;
    }
    return 1;
}

int main()
  {
  int number;
  cin>>number;
  if(isPowerOfSixteen(number))
      cout<<number<<" is a power of 16";
  else
      cout<<number<<" is not a power of 16";
  }

```

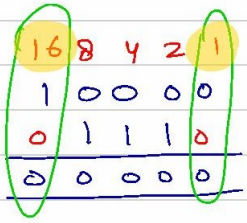
Approach 2:

Same as power of 4, power of 8, we only need to create a different mask.

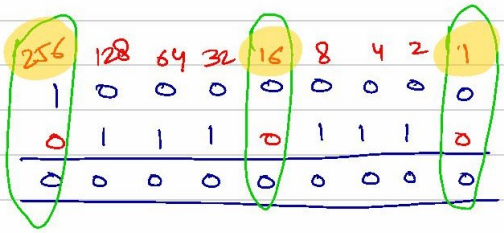
NOTE:- To create mask, those bits who are power of 16, set them as 0, rest as 1.

Ex- N = 16

N = 16



N = 256



→ If we observe, every 4th bit starting from 0th bit is power of 16, set them as 0, rest as 1.

mask = 1110 1110 1110 1110 1110 1110 1110 1110

E E E E E E E E

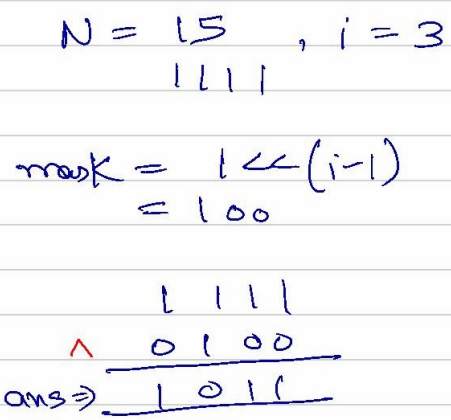
```cpp
#include <iostream>
using namespace std;

bool checkPowerOf16(int n){
    if(n==0) return false;
    bool powerof2 = !(n & (n - 1));

    //for mask, bits will be 0, in the place which are power of 16
    //bits will be 1, in the place which are not power of 16
    //0111101110, 1,16,256 are powers of 16.
    /*starting from 0th position every fourth bit will be a power of 16,
    so mark them as 0.
    (0xEEEEEEEE)16 = (11101110111011101110111011101110)2
    */

    bool mask = !(n & 0xEEEEEEEE);
    return  powerof2 && mask;
}

int main(){
     int n;
    cin>>n;
    if (checkPowerOf16(n)) {
        cout << n << " is a power of 16";
    }
    else {
        cout << n << " is not a power of 16";
    }
    return 0;
}
```

Approach 3:

N is a power of 16, if it is a power of 2 and if modulus of n by 15 will give remainder 1.

```cpp

#include <iostream>
using namespace std;

bool checkPowerOf16(unsigned n){
    return !(n & (n - 1))&& (n % 15 == 1);
}

int main()
{
    unsigned n;
    cin>>n;

    if (checkPowerOf16(n)) {
        cout << n << " is a power of 16";
    }
    else {
        cout << n << " is not a power of 16";
    }

    return 0;
}
```

Q 10.) Toggle ith Bit of a number:




```cpp
#include<iostream>
using namespace std;

int toggleIthBit(int n, int I)
{   int mask = 1 << (I-1);
    return (n^ mask);
}

int main()
{
    int n,I;
    cin>>n>>I;
    cout << toggleIthBit(n , I);
    return 0;
}
```

Q 11.) Count the number of set bits in a number

input = 1110
output = 3.

cnt = 0;

N = 1110
& 1
0000

N = 0111
& 1
0001 cnt = 1

N = 0011
& 1
0001 , cnt = 2

N = 0001
& 1
0001 , cnt = 3

N = 0. Stop.

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

TC = O(logn)

→ Second way, Better in Some cases :-

Brian Kernighan's Algorithm.

→ clear the rightmost set bit everytime, until N becomes 0.

N = 13
1101

N = N & N-1 = 1100 (1)
= 1000 (2)
= 0000 (3)

→ So we needed only 3 steps, But if N = 7, 15, it is same as previous.

```
unsigned int countSetBits(int
n)
{
    unsigned int count = 0;
    while (n!=0) {
        n &= (n - 1);
        count++;
    }
    return count;
}
```

TC = O(No. of set bits in number).

Q 12. Find the two non-repeating elements in an array of repeating elements/ Single Number III

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

Brute force :-

```
for(int i=0; i<n; i++)
{  int cnt = 0;
   for(int j=0; j<n; j++)
   {  if(a[j]==a[i]
         cnt++;
   }
   if(cnt==1)  print(a[i]);
}
```

TC = O(N²).

(2) We can use map :-

```
unordered_map<int, int> mp;
for(int i=0; i<n; i++)
{
    mp[a[i]]++;
}
for(auto it : mp)
{ if(it.second==1)
      print(it.first);
}
```

TC = O(N).
SC = O(N).

(3.) XOR :-

XOR = 5 ^ 7.

```
  1 0 1
  1 1 1
  ------
  0 1 0
```

second bit is 1, means it is differ in both the numbers, so if I take a rightmost set bit and create two sets

2nd bit is 0 (y)	2nd bit is 1 (x)
~~1~~	~~2~~
~~5~~	~~2~~
~~1~~	7
~~4~~	~~3~~
~~4~~	~~3~~

x = 7, result = 2.
so, result ^ x will give y.

y = (x ^ result) = 7 ^ 2
= 5,
so, x = 7, y = 5.

```
1 vector<int> singleNumber(vector<int>& nums) {
2         long long result=0;
3         for(int i=0;i<nums.size();i++) {
4             result=result^nums[i];   = 2
5         }
6
7         long long mask = (result & ~(result - 1));
        //simple way to find rightmost set bit.
8
9
10        int x=0;
11        int y=0;
12
13        for(int i=0;i<nums.size();i++)
14        {
15            if((nums[i]&mask)>0)
16            {
17                x^=nums[i]; x = 2^2^7^3^3
18            }             x = 7.
19        }
20        y=result^x;  => 2^7 = 5
21
22    return vector<int>{x, y};
23    }
```

TC = O(N) + O(N).

Given an integer array nums, in which exactly two elements appear only once and all the other elements appear exactly twice. Find the two elements that appear only once. You can return the answer in any order.

You must write an algorithm that runs in linear runtime complexity and uses only constant extra space.

Alphabetical Letters Operations

1. Convert Uppercase to LowerCase:

→ We can convert uppercase to lowercase by doing Bitwise OR and space.
(ch | ' ')

```
1 void uppertolower(char ch){
2    cout <<char(ch | ' ');
3 }
4
5 int main() {
6    char ch;
7    cin>>ch;
8    uppertolower(ch);
9 }
```

2. Convert Lowercase to Uppercase:

→ We can convert a lowercase to uppercase by taking bitwise AND with underscore.
(ch & '_')

```
1 void lowertoupper(char ch){
2    cout <<char(ch & '_');
3 }
4
5 int main() {
6    char ch;
7    cin>>ch;
8    lowertoupper(ch);
9 }
```

3. Invert Alphabet's Case

→ We can invert the alphabet case from uppercase to lower or vice versa using Bitwise XOR(^) with space.

(ch ^ ' ')

NOTE:- Don't forget to typecast it in char.

→ The ASCII code of space(' ') is 0 0 1 0 0 0 0 0. 32
→ The ASCII code of underscore is 0 1 0 1 1 1 1 1

Explaination of above:-

→ Bitwise OR of an uppercase character with ' ', will set the third significant bit and we will get lowercase character.

→ Bitwise AND of a lowercase character with '_', will unset/clear the third bit and we will get uppercase character.

→ Bitwise XOR of an uppercase or lowercase characters with ' ' (space), will toggle the third significant bit.
Uppercase becomes lowercase and vice versa.

```
1 void invertcase(char ch){
2     cout << char(ch ^ ' ');
3 }
4
5 int main() {
6     char ch;
7     cin>>ch;
8     invertcase(ch);
9 }
```

4. Find Letter Position in alphabet:

→ We can find letter position by taking bitwise AND with ASCI code 31 (00011111 in binary).

('A' & 31) ⇒ 1
('c' & 31) ⇒ 3 } case will not matter here

```cpp
#include <iostream>
using namespace std;

void findalphabeticalposition(char ch){
   cout <<(ch & 31);
}

int main() {
   char ch;
   cin>>ch;
   findalphabeticalposition(ch);
}

```

The left shift and right shift operators should not be used for negative numbers

The bitwise XOR operator is the most useful operator from a technical interview perspective.

Let's solve some problems related to XOR:

Q .1) Given a set of numbers where all elements occur an even number of times except one number, find the odd occurring number.

arr[] = {4,2,2,1,5,6,7,5,6,7,4}.

$= 4 \wedge 2$

$= 4 \wedge 2 \wedge 2 = 4$

$= 4 \wedge 1$

$= 4 \wedge 1 \wedge 5$

$= 4 \wedge 1 \wedge 5 \wedge 6$

$= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7$

$= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7 \wedge 5$

$= 4 \wedge 1 \wedge 6 \wedge 7 \wedge 6$

$= 4 \wedge 1 \wedge 7 \wedge 7$

$= 4 \wedge 1 \wedge 4$

$= 1.$

```
1 int getOddOccurrence(int arr[], int arsize)
2 {
3     int res = 0;
4     for (int i = 0; i < arsize; i++)
5         res = res ^ arr[i];
6
7     return res;
8 }
```

Q 2) Swap two numbers using Bit manipulation:

$a = 5,\ b = 7$

1) $a = a \wedge b$
2) $b = a \wedge b$
3) $a = a \wedge b$

$a = 5 \wedge 7$

$b = 5 \wedge 7 \wedge 7 = 5$

$a = 5 \wedge 7 \wedge 5$

$= 7$

Q 3.) Calculate XOR from 1 to n

$n = 5$

output $= 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 = 1$

∴ We can run for loop n times. TC = O(N). ✗

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

→ if (n % 4 == 0) return n;
→ if (n % 4 == 1) return 1;
→ if (n % 4 == 2) return n+1;
→ if (n % 4 == 3) return 0;

```cpp
int computeXOR(int n)
{
  // If n is a multiple of 4
  if (n % 4 == 0)
    return n;

  // If n%4 gives remainder 1
  if (n % 4 == 1)
    return 1;

  // If n%4 gives remainder 2
  if (n % 4 == 2)
    return n + 1;

  // If n%4 gives remainder 3
  return 0;
}
```

Q 4.) Find XOR of numbers frokm the range [L,R]

Input L = 4, R = 8
Output ⇒ 4 ^ 5 ^ 6 ^ 7 ^ 8 = 8

∴ Run loop from L to R. TC = O(N) ✗

→ findXOR(L-1) → use above logic = O(1).
→ findXOR(R) → use above logic = O(1).

⇒ findXOR(L-1) ^ findXOR(R)

∴ (1 ^ 2 ^ 3) ^ (1 ^ 2 ^ 3 ^ 4 ^ 5 ^ 6 ^ 7 ^ 8)
= 4 ^ 5 ^ 6 ^ 7 ^ 8 = 8.

```cpp
int findXOR(int n)
{
    int mod = n % 4;
    if (mod == 0)
        return n;

    if (mod == 1)
        return 1;
    if (mod == 2)
        return n + 1;
    if (mod == 3)
        return 0;
}

int findXOR(int l, int r) {
    return (findXOR(l - 1) ^ findXOR(r));
}
```

Q 5.) Check whether the number is even or not

→ one way to find is :-

```
if(number % 2 == 0)
    return 0; //even
else
    return 1; //odd
```

→ using & operator

```
if(number & 1 == 0)
    return 0; //even
else
    return 1; //odd
```

∴ Using & takes less time.

Q 6.) Find the XOR of the XOR of all subsets of an array: $= 0$.

arr = [1,3,2]

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

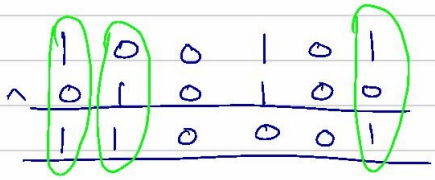
0

→ XOR will be non zero, only if there is only one element in the array. return arr[0] in that case.

```c
int findXOR(int Set[], int n)
{
  if (n == 1)
    return Set[0];
  else
    return 0;
}

int main()
{
  int Set[] = { 1, 2, 3 };
  int n = sizeof(Set) / sizeof(Set[0]);
  printf("XOR of XORs of all subsets is %d\n",
  findXOR(Set, n));
  return 0;
}
```

Q 7.) Count Number of bits to be flipped to convert A to B:

A = 37, B = 20



→ First Do XOR, result will contain no. of sets bits to be flipped.

→ Count No. of set bits in XOR of A & B using Brian Kernighan's algorithm n & (n-1).

```cpp
#include<iostream>
using namespace std;

int countflipbits(int A,int B){
    int n = A^B;
    int count = 0;
    while(n){
        n = n & (n-1);
        count++;
    }
    return count;
}

int main()
{
    int A,B;
    cin>>A>>B;
    cout<<"No of bits to be flip to Convert A to B is : "<<countflipbits(A,B);
    return 0;
}

```

Q 8.) Find missing number in an array:

Given an array nums containing n distinct numbers in the range [0,n], return the only number in the range that is missing from the array.

→ We can use XOR to find missing number.

→ first do XOR of all the elements in the array.

→ Array elements are in range of 0 to N, so XOR of 0-N with XOR of all array elements will give us the missing element.

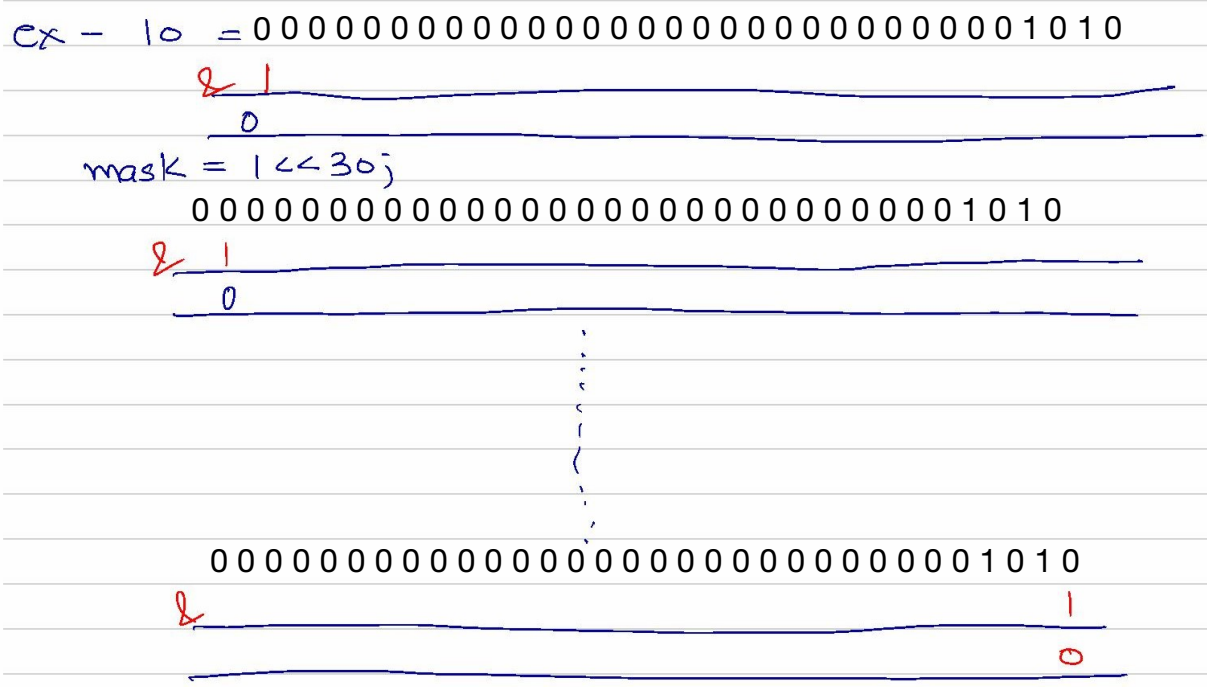
nums = [3, 0, 1]

= (3 ^ 0 ^ 1) ^ (0 ^ 1 ^ 2 ^ 3)

= 2.

```
int missingNumber(vector<int>& nums) {
        int Xor=0;
        int n = nums.size();

    for (int i = 0; i < n; i++) {
        Xor = Xor ^ nums[i];
    }
  //numbers are starting from 0
    for (int i = 0; i <= n; i++) {
        Xor = Xor ^ i;
    }
     return Xor;
    }
```

Q 9) Print the binary representation of decimal number:




```cpp
#include<iostream>
using namespace std;

void decitobinary(int num){
for(int i=31;i>=0;i--)
{
    int mask = 1<<i;
    if(num & mask){
        cout<<"1";
    } else cout<<"0";
}
}

int main()
{
    int num;
    cin>>num;
    cout<<"Binary representation of decimal number is ";
    decitobinary(num);
    return 0;
}

```

Q 10.) Reverse the bits of a number:

n = 00000000000000000000000010101010

Take ans as 0

ans = 00000000000000000000000000000000

→ We will try to create our ans from LSB to MSB, everytime we will compute our ans and left shift it by 1.

→ Now to compute our ans, we will first check whether the bit is set or not, to check it do (n&1).

→ Now OR operation of ans with n&1 will give us ans.

→ Right shift n by 1 everytime till we reach MSB(32 for unsigned int).

(1) n = 00000000000000000000000010101010

```
  0 . . . . . 0
&
  0 . . . . . 1
```

ans = 0 | 0
= 0

n = n >> 1.

(2) n = 00000000000000000000000001010101

```
  0 . . . . . 1
&
  0 . . . . . 1
```

ans << = 1.
ans = ans | (n&1)
= 0 | 1
ans = 1

(3) n = 00000000000000000000000001010100

```
  0 . . . . . 1
&
  0 . . . . . 0
```

ans << = 1 = 00000 . . . . . |0 = 2.
ans = ans | (n&1)
= 2 | 0
= 2

Everytime answer will be left shift by 1 and or operation will place the correct bit at the position.

```cpp
uint32_t reverseBits(uint32_t n) {
        int ans = 0;
         for(int i=0;i<32;i++)
         {
             ans <<= 1;
             ans = ans|(n&1);
             n >>= 1;
         }
         return ans;
    }
```

Q 11.) Swap the ith and Jth bit.

Given a number num, i and j ranges from 1 to 32.
→ we need to swap i^{th} and J^{th} bit of a number.

ex → num = 43, i = 2, j = 5.
0 0 1 0 1 0 1 1 , output = 57 (0 0 1 1 1 0 0 1)
j i

→ first move the bit value at i^{th} & j^{th} bit
Let A is the bit value at i^{th} bit:-

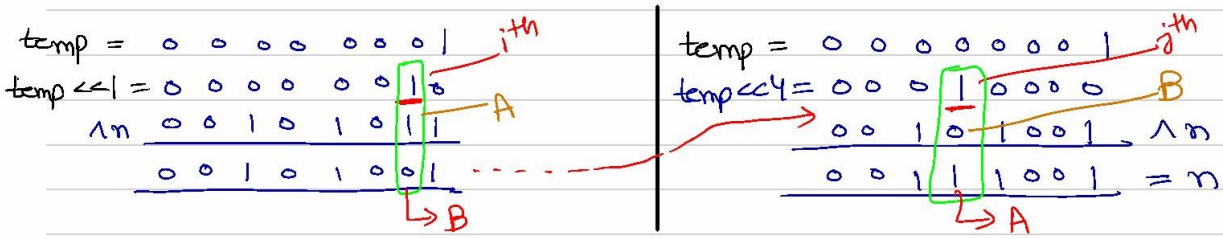
```
n       = 0 0 1 0 1 0 1 1
n>>1    = 0 0 0 1 0 1 0 1
(n>>1)&1 =                1
A       = 0 0 0 0 0 0 0 1
```

move j^{th} bit to rightmost

```
n       = 0 0 1 0 1 0 1 1
n>>4    = 0 0 0 0 0 0 1 0
(n>>4)&1 =              1
B       = 0 0 0 0 0 0 0 0
```

temp = A ∧ B = 0 0 0 0 0 0 0 1

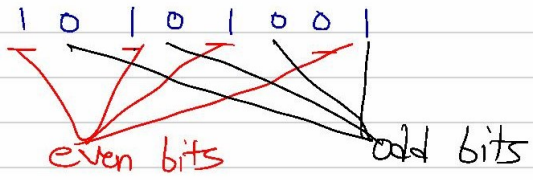
Now, A ∧ B = temp, temp ∧ B = A, temp ∧ A = B
→ Move temp first at ith position & then at jth position.




```cpp
#include<iostream>
using namespace std;

int swapBits(unsigned int n, unsigned int i, unsigned int j)
{   unsigned int A = ((n>>(i-1)) & 1);
    unsigned int B = ((n>>(j-1)) & 1);

    unsigned int temp = A ^ B;
    n = n ^ (temp<< (i-1));
    n = n ^ (temp<<(j-1));
    return n;
}

int main()
{   int n,i,j;
    cin>>n>>i>>j;
    int res = swapBits(n,i,j);
    printf("Result = %d ", res);
    return 0;
}
```

Q 12.) Swap all even and odd bits

Ex -




```
        1 0 1 0 1 0 0 1
even mask 1 0 1 0 1 0 1 0
even bits= 1 0 1 0 1 0 0 0
even bits >>= 1;
even bits = 0 1 0 1 0 1 0 0
result = 0 1 0 1 0 1 1 0
```

```
1 0 1 0 1 0 0 1
0 1 0 1 0 1 0 1   - odd mask
0 0 0 0 0 0 0 1   - odd bits
odd bits <<= 1
odd bits = 0 0 0 0 0 0 1 0
```

```cpp
#include<iostream>
using namespace std;

unsigned int swapevenoddbits(unsigned int N)
{
    unsigned int even_bits = N & 0xAAAAAAAA;
    unsigned int odd_bits = N & 0x55555555;

    // Right shift even bits
    even_bits >>= 1;

    // Left shift odd bits
    odd_bits <<= 1;

    // Combine even and odd bits
    unsigned int result = even_bits | odd_bits;
    return result;
}

int main()
{
    unsigned int N;
    cin>>N;

    cout<<swapevenoddbits(N);
    return 0;
}

```

Q 13.) Copy set bits in a range, toggle set bits in a range:

1.) Copy set bit :- we have two numbers A and B, and we want to copy the bits of B to A for a given range L to R from LSB to MSB.

→ L starts from 1 and R goes till 32.

Ex →

	8	7	6	5	4	3	2	1
A =	0	1	0	1	0	1	1	0
B =	1	0	0	1	1	0	0	1

L = 3

R = 7

if we create a mask like

mask = 0 1 1 1 1 1 0 0

and do Bitwise AND with A, we can extract set bits of A in given range.

A =	0	1	0	1	0	1	1	0
mask =	0	1	1	1	1	1	0	0
	0	1	0	1	0	1	0	0

→ Now If do Bitwise OR of (A & mask) with B, we will get our answer.

→ Creating mask is a crutial part, there can be multiple ways to create the mask.

① left shift 1 till (r-l+1)

mask =	0	0	1	0	0	0	0	0
(mask-1) =	0	0	0	1	1	1	1	1

Now left shift it by (l-1) times.

(mask-1) << (l-1) = 0 1 1 1 1 0 0

(2) Second way to create mask is :-

right mask = $(1 << r) - 1$

= 0 1 1 1 1 1 1 1

left mask = $1 << (l-1) - 1$

= 0 0 0 0 0 0 1 1

→

```
  0 1 1 1 1 1 1
  0 0 0 0 0 0 1 1
---------------
  0 1 1 1 1 0 0
---------------
```

```
#include<iostream>
using namespace std;

int CopySetBit(int X, int Y, int l, int r){

 //only 1 can overflow in case of r = 32 and l=1
 //we need to take 1ll
   int maskLength = (1ll<<(r-l+1)) - 1;

    int mask = ((maskLength)<<(l-1)) & X;

    Y = Y | mask;
    return Y;
    }

int main()
{
    unsigned int A,B,l,r;
    cin>>A>>B>>l>>r>>l;

    cout<<CopySetBit(A,B,l,r);
    return 0;
}
```

2.) Toggle bits in range:-

⇒ Mask computation is same as copy bits in range

There are two ways above.

① `mask = (1 << (r-l+1)) - 1;`
`mask = (mask << (l-1));`

② `mask = ((1<<r)-1) ^ ((1 << (l-1))-1).`

⇒ Now after computing the mask, if we do XOR of mask with num, we will get toggle bits in range L to R.

`num = num ^ mask.`

Ex- `1 1 0 1 1 0 1 1` , i = 2, j = 6

mask `0 0 1 1 1 1 1 0`

`1 1 1 0 0 1 0 1` → result.

```
1 int toggleBits(int N , int L , int R) {
2         int mask = (1<<(R-L+1))-1;
3         mask = (mask<<(L-1));
4         int result = N ^ mask;
5     }
```

Q 14.) Divide two integers without using Multiplication, Division and mod operator:

ex- Dividend = 10
Divisor = 3
Quotient = $\left\lfloor \frac{10}{3} \right\rfloor = 3$.

Approach 1: Repeated subtraction :-

Keep subtracting the divisor from dividend untill dividend becomes less than divisor.
So, dividend will get reduced to became remainder & no. of times will subtract divisor from dividend will became the quotient.

Eg. $10 - 3 = 7$
$7 - 3 = 4$
$4 - 3 = 1 < 3$
} (3) $q = 3$

Time Complexity = O(dividend)
Space Complexity = O(1)

Approach 2 :-

As every number can be represented in base 2 (0 or 1), represent the quotient in binary form by using the shift operator as given below:

1. Determine the most significant bit in the divisor. This can easily be calculated by iterating on the bit position i from 31 to 1.
2. Find the first bit for which ***divisor << i*** is less than dividend and keep updating the i^{th} bit position for which it is true.
3. Add the result in the *temp* variable for checking the next position such that **(temp + (divisor << i))** is less than the **dividend**.
4. Return the final answer of the quotient after updating with a corresponding sign.

Q 15.) SINGLE NUMBER II

Given an integer array nums where every element appears thrice times except for one, which appears exactly once. Find the single element and return it.

ex – nums = [1, 1, 5, 1]
output = 5

Method 1: Brute Force

→ We can run two for loops and count the elements, if the count of any element is 1, after iterating the complete array, it will be our answer.

```
1 //brute force
2     int singleNumber(vector<int>& nums) {
3         int nsize = nums.size();
4 
5         for(int i=0;i<nsize;i++){
6              int count = 0;
7             for(int j=0;j<nsize;j++){
8                 {
9                     if(nums[i]==nums[j]){
10                        count++;
11                    }
12                }
13            }
14            if(count==1){
15                return nums[i];
16            }
17        }
18        return -1;
19    }
```

TC = $O(N^2)$

Method - 2 : Sorting + linear traversal

Ex - 5 4 2 4 4 5 5

After Sorting :- 2 4 4 4 5 5 5

→ We will have three conditions :-
1. Left Boundary condition
2. Right Boundary condition
3. other than Boundary element

1. Left Boundary condition :-
ex - 2 4 4 4 5 5
→ check if (num[0] != num[1])

2. Right Boundary condition :-
ex - 4 4 4 5 5 5 6
→ check if (nums[n-2] != nums[n-1])

3. Somewhere except Boundary
ex - 1 1 1 2 3 3 3

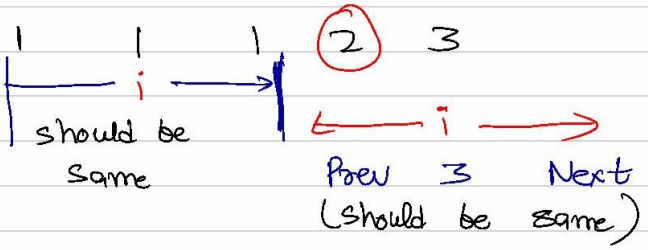
run a while loop, start comparing from index 1,

```
{ if (nums[i] != nums[i-1])
        return nums[i-1];
  }
  i+=3;
```

1 1 1 2 3 3 3
i
i

→ if current element is same as previous, then surely its next element will also be the same.

→ If we increament by 3, and its prev elem doesn't equal to current element then prev element is ans.



```cpp
//better approach : Sorting + linear traversal
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        sort(nums.begin(),nums.end());
        if(n<3) return nums[0];

        //check boundary conditions
        //2 4 4 4 5 5 5
        if(nums[0]!=nums[1]) return nums[0];
        //another boundary condition
        // 4 4 4 5 5 5 6
        if(nums[n-2]!=nums[n-1]) return nums[n-1];
        int i=1;
        while(i<n){
            if(nums[i]!=nums[i-1]){
                return nums[i-1];
            }
            i+=3;
        }
        return -1;
    }
```

$TC = O(N \log N) + O(N)$

$O(N \log N)$ ↓ sorting

$O(N)$ ↓ Traversing

Method 3: Unordered_map

→ We can store the count of elements in unordered_map and traverse it.

```cpp
//3rd approach : unordered_map : TC : O(N) , SC : O(N)
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        unordered_map<int,int> mp;
        for(int i=0;i<n;i++){
            mp[nums[i]]++;
        }

        for(auto it : mp){
            if(it.second==1){
                return it.first;
            }
        }

    return -1;
    }
```

TC = O(N)
SC = O(N)

Method 4: Counting Set Bits in Every Number array

ex - [2 2 2 3 5 5 5]

→ Check for all 32 bits.
→ Use shift operation to find if current bit is set or not.
→ if the number is only appearing once, the set bit at that particular position will not be multiple of 3.
so add that in ans.

0 1 0
0 1 0 } 2
0 1 0

0 1 1 ← 3

1 0 1
1 0 1 } 5
1 0 1

result ↓ leftshift ↓

i = 0, leftshift = 1, numberofsetbits = 4, result = 0 + 1 = 1
i = 1, leftshift = 2, numberofsetbits = 4, result = 1 + 2 = 3
i = 2, leftshift = 4, numberofsetbits = 3, result = 3 + 4 = 7

i = 31

```cpp
//4th approach : counting Set bits in every number array
//TC: O(32N) every case not better than 3rd approach
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        int result = 0;

    int leftshift, numberofsetbits;

    // Iterate through every bit
    for(int i = 0; i <32; i++) {

        numberofsetbits = 0;
        leftshift = (1 << i);
        for (int j = 0; j < n; j++) {
            if (nums[j] & leftshift)
                numberofsetbits++;
        }
        if ((numberofsetbits % 3) != 0)
            result+= leftshift;
    }

    return result;
    }
```

Method : 5 Bit Manipulation:

→ We will use two variables ones and twos.
Ones will store the element, which is occuring only once.
twos will store the element, which is occuring twice.
→ when an element is occuring thrice, ones and twos will store 0.

ex- [2 2 2 3]

i=0, ones = 0, element = 2 (10), twos = 0 (00)

```
ones = (ones ^ ele) & (~twos);
twos = (twos ^ ele) & (~ones);
```

ones = 0 0
^ 1 0
1 0
& 1 1
1 0

ones = 2

twos = 0 0
^ 1 0
1 0
& 0 1
0 0

twos = 0

i=1, ones = 1 0
^ 1 0
0 0
& 1 1
0 0

ones = 0

twos = 0 0
^ 1 0
1 0
& 1 1
1 0

twos = 2

i=2, ones = 0 0
^ 1 0
1 0
& 0 1
0 0

twos = 1 0
^ 1 0
0 0
& 1 1
0 0

i=3, ones = 0 0
^ 1 1
1 1
& 1 1
1 1

ones = 3

twos = 0 0
^ 1 1
1 1
& 0 0
0 0

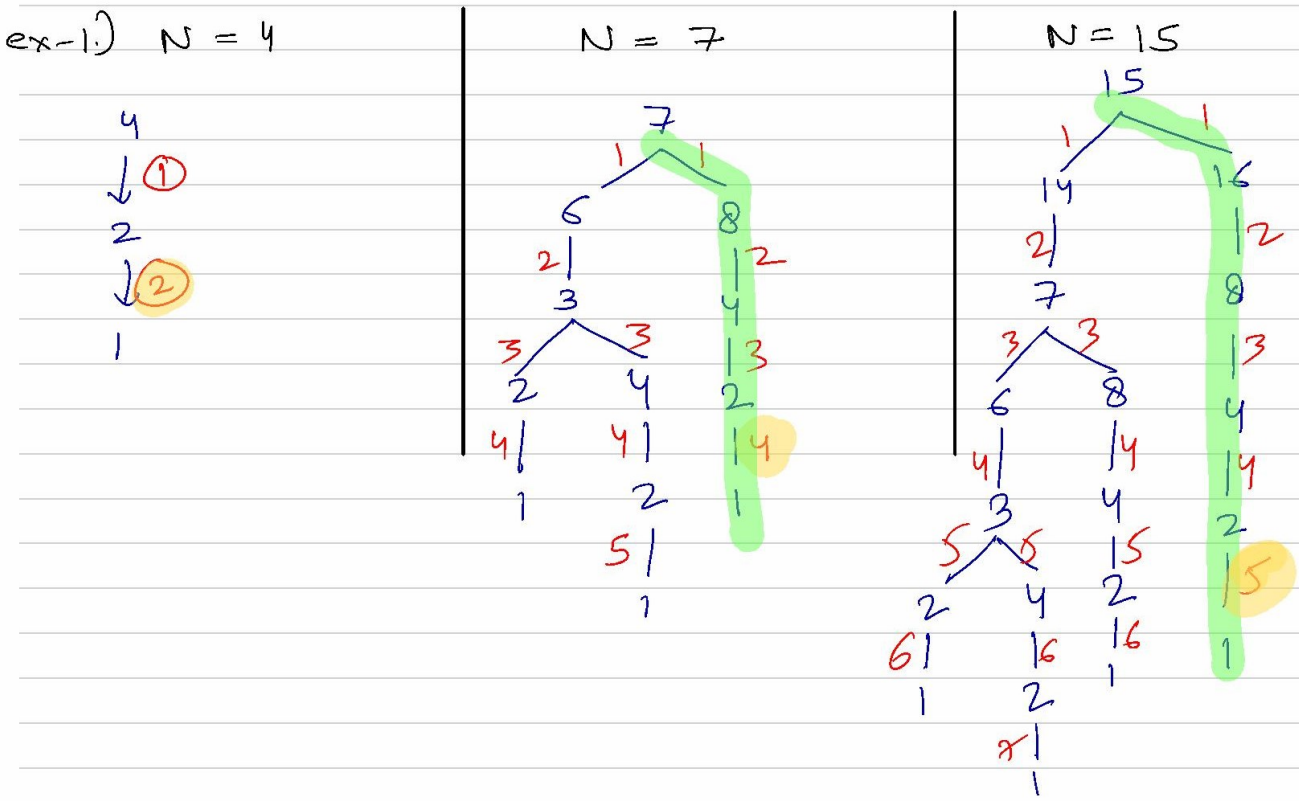
twos = 0

```cpp
//5th approach : Bit manipulation (Not intuitive)
//TC: O(N)
//our answer will store in ones
    int singleNumber(vector<int>& nums) {
        int ones =0;
        int twos =0;

        for(auto ele: nums){
            ones = (ones^ele) & (~twos);
            twos = (twos^ele) & (~ones);
        }
        return ones;
    }
```

Q 16.) Reduce a Number to 1:

Given a number, our task is to reduce the given number N to 1 in the minimum number of steps.
We can perform two types of operations in each step:

Operation 1: If the number is even then divide the number by 2
Operation 2: If the number is odd, then we are allowed to either (N+1) or (N-1).



Method 1:

We can use recursion untill n=1, for even numbers, we can return 1 + funct(n/2),
for odd numbers, we can return : 1 + min(funct(n-1),funct(n+1)).

```
1 int minways(int n)
2 {
3     if (n == 1)
4         return 0;
5     else if (n % 2 == 0)
6         return 1 + minways(n / 2);
7     else
8         return 1 + min(minways(n - 1),minways(n + 1));
9 }
```

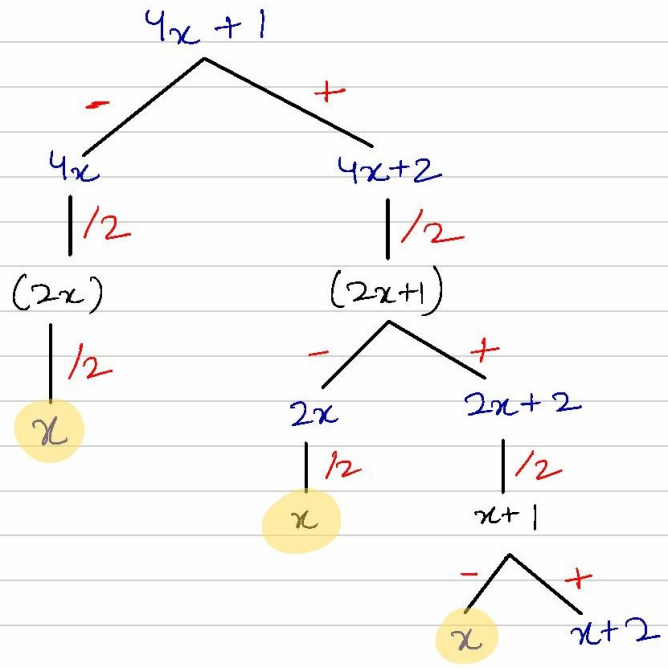
Method 2: Using Bit Manipulation

Even number can be represent as 4x + 0, i.e. 4x
Odd number can be represent as 4x+1 and 4x+3.

for Even number, we know that we need to divide it by 2,
but for odd number we have two cases(4x+1, 4x+3)and two operations (n-1,n+1).So We need to identify, which case is suitable for which operation.

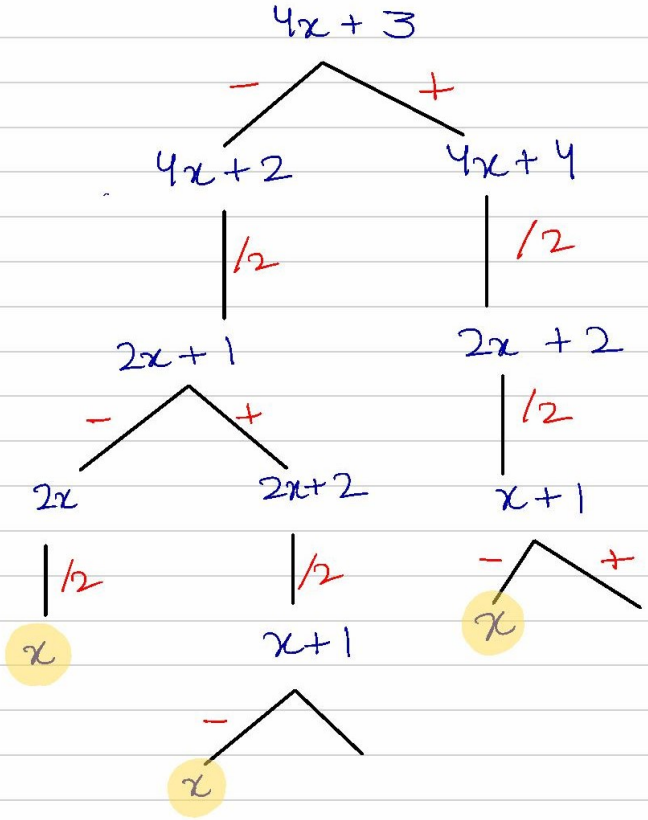
→ Lets take some examples :-

1) Take 4x+1 case first,



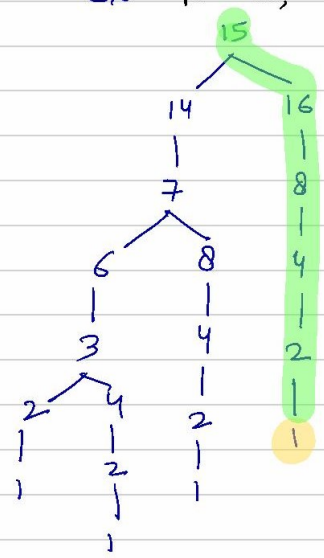
→ for case 4x+1, we will reach to the 1, in minimum steps using n-1 at every step having odd number.

2.) Take $4x+3$ case

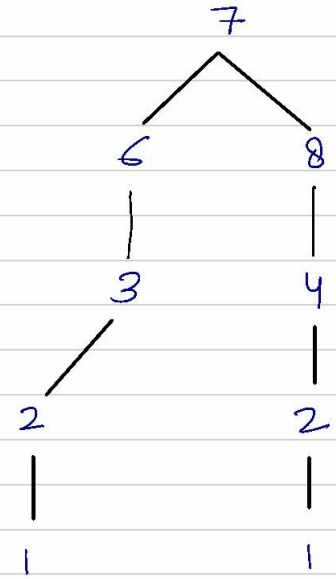


→ In above case if $x-1$ will give min answer then both − & + takes same step, but if $x+1$ will give min ans then + path takes min step, so + path is better than −.

ex- N=15,



ex - N=7



3.) If N=3 :-

If N=3, n-1 will give minimum steps.

Now, N%2, can be represented as ((n&1)==0)

(N%4==3) can be represented as ((N&3)==3)

(N%4==1) $\Rightarrow$ ((N&3)==1)

NOTE:- N+1 can be overflow, so take long.

```
1 int countSteps(int n)
2 {
3   int count = 0;
4   while (n > 1) {
5       count++;
6
7       // num even, divide by 2
8       if ((n&1)==0)
9           n>>=1;
10
11      // num odd, n%4 == 1 or n==3(special edge case),
12      else if ((n & 3) == 1 || (n==3) )
13          n -= 1;
14
15      // num odd, n%4 == 3, increment by 1
16      else if((n&3) == 3)
17          n += 1;
18  }
19 return count;
20 }
21
```

Q 17.) Detect if two integers have opposite sign:

→ Signed integers in computer are store in 2's complement form. where MSB bit represent the sign of the number
1 → Negative integer
0 → Positive integer

So, if two integers have opposite sign, the XOR of two number will give negative number, so

```
if((A ^B) <0)
    "Opposite sign";
else
    "Not opposite sign";
```

```cpp
int main()
{
    int num1,num2;
    cin>>num1>>num2;

    if((num1^num2)<0){
        cout<<"Opposite Sign"<<endl;
    }
    else cout<<"Not Opposite sign"<<endl;

    return 0;
}
```

Q 18.) Add 1 to an integer:

→ We need to add one to a given number without using +, −, ×, /, ++, −− .... operator.

Method 1 :- flip all the bits after rightmost 0 bit.
→ flip the rightmost 0 bit.
⇒ we will get our answer.

ex − 5
101
↓
110
= 6

ex − 11
1011
↓
1100
= 12

```
1 int addOnetoint(int num)
2 {   int leftshift = 1;
3
4     // Flip all the set bits until we find a 0
5     while(num & leftshift)
6     {   num = num ^ leftshift;
7         leftshift <<= 1;
8     }
9
10    // flip the rightmost 0 bit
11    num = num ^ leftshift;
12    return num;
13 }
```

METHOD-2 :- Let's say x is the numerical value of a number. then $\sim x = -(x+1)$ (in 2's complement)
so to get $(x+1) \Rightarrow -(\sim x)$.

ex- 4

2's complement of $\sim x$ =

$-(\sim x) = -(-5) = 5$

```
1 int addOnetoint(int num)
2 {
3     return (-(~num));;
4 }
```

Q 19.) Find Xor of a number without using XOR operator:

Method 1: Traverse all bits one by one.
for every pair of bits, check if both are same or not.

```
1 int XOR(int num1, int num2)
2 {
3     int res = 0;
4
5     // Assuming 32-bit Integer
6     for (int i = 31; i >= 0; i--)
7     {
8     bool bit1 = num1 & (1 << i);
9     bool bit2 = num2 & (1 << i);
10
11    bool XoredBit = (bit1 & bit2) ? 0 : (bit1 | bit2);
12
13    res <<= 1;
14    res |= XoredBit;
15    }
16    return res;
17 }
```

Method 2: we can use the property of XOR.

$$a \wedge b = a\bar{b} + b\bar{a}$$

```
1 int XOR(int num1, int num2)
2 {
3     int res = (num1 & (~num2)) | ((~num1 )& num2);
4     return res;
5 }
```

Q 20.) Determine if two integers are equal without using comparison and arithmetic operators

Method 1: Using XOR :-

```
if (a ^ b)
    "Not same";
else
    "Same";
```

```
1 void areSame(int num1, int num2)
2 {
3     if (num1 ^ num2)
4         cout << "Not Same";
5     else
6         cout << "Same";
7 }
```

Method :- 2

```
if ((a & ~b) == 0)
    "same";
else    "Not same";
```

```
1 void areSame(int num1, int num2)
2 {
3     if ((num1 & (~num2))==0)
4         cout << "Same";
5     else
6         cout << "Not  Same";
7 }
```

Q 21.) Find minimum or maximum of two integers without using branching

Method 1 :-

Let us assume 'b' is minimum and 'a' is maximum among 'a' and 'b'.('a' < 'b') is the comparison we will be using. We will calculate minimum and maximum as follows:

- We can write the minimum as 'b' ^ (('a' ^ 'b') & - ('a' < 'b')).
  - If 'b' is minimum 'a' < 'b' comes out to be all zeroes. ('a' ^ 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'b' finally which is the minimum.
  - If 'a' is minimum 'a' < 'b' comes out to be all ones. ('a' ^ 'b') & '1' comes out to ('a' ^ 'b'). Therefore expression value comes out to be 'a' finally which is the minimum.

- We can write maximum as 'a' ^ (('a' ^ 'b') & - ('a' < 'b')).
  - If 'a' is maximum 'a' < 'b' comes out to be all zeroes. ('a' ^ 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'a' finally which is the maximum.
  - If 'b' is maximum 'a' < 'b' comes out to be all ones. ('a' ^ 'b') & '1' comes out to ('a' ^ 'b'). Therefore expression value comes out to be 'b' finally which is the maximum.

Method-2

- We can write the minimum as 'b' + (('a' - 'b') & (('a' - 'b') >> (noOfBitsInInt - 1))). On right shifting 'a' - 'b' by 1 less than no of bits in int we get the most significant bit.
  - If 'b' is minimum 'a' - 'b' comes out to be non-negative and on right shifting, we get '0'. ('a' - 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'b' finally which is the minimum.
  - If 'a' is minimum 'a' - 'b' comes out to be negative and on right shifting, we get '1'. ('a' - 'b') & '1' comes out to ('a' - 'b'). Therefore expression value comes out to be 'a' finally which is the minimum.

- We can write maximum as 'a' - (('a' - 'b') & (('a' - 'b') >> (noOfBitsInInt - 1))). On right shifting 'a' - 'b' by 1 less than no of bits in int we get most significant bit.
  - If 'a' is maximum 'a' - 'b' comes out to be non-negative and on right shifting, we get '0'. ('a' - 'b') & '0' comes out to '0'. Therefore expression value comes out to be 'a' finally which is the maximum.
  - If 'b' is maximum 'a' - 'b' comes out to be negative and on right shifting, we get '1'. ('a' - 'b') & '1' comes out to ('a' - 'b'). Therefore expression value comes out to be 'b' finally which is the maximum.

Reference - codestudio

LEETCODE PROBLEMS:

Q 1.) Find missing and repeating number / Set mismatch:

You have a set of integers s, which originally contains all the numbers from 1 to n. Unfortunately, due to some error, one of the numbers in s got duplicated to another number in the set, which results in repetition of one number and loss of another number.

You are given an integer array nums representing the data status of this set after the error.

Find the number that occurs twice and the number that is missing and return them in the form of an array.

```cpp
vector<int> findErrorNums(vector<int>& arr) {
        int n  = arr.size();
        vector<int> temp(n+1,0);
       vector<int> ans(2,0);
       for(int i=0;i<n;i++){
           temp[arr[i]]++;
       }
       for(int i=1;i<=n;i++){
           if(temp[i]==0){
               ans[1] =i;
           }
           if(temp[i]==2){
               ans[0] = i;
           }
       }
       return ans;
```

Method -2 :

Ex - 1 1 2 3 4 6

Sum of N numbers $= \frac{N \times (N+1)}{2} = S$

Summation of square of a number $= \frac{n \times (n+1) \times (2n+1)}{6} = P$

$\Rightarrow$ S - array elements

$(1+2+3+4+5+6) - (1+2+3+4+6+1)$

$5 - 1$

$= x - y$

$\Rightarrow S^2$ - (array elements)^2

$= 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 - (1^2 + 2^2 + 3^2 + 4^2 + 6^2 + 1^2)$

$= 25 - 1 \quad (x^2 - y^2)$

$= 24$

$$\left[\begin{array}{l} x^2 - y^2 = 24 \\ x - y = 4 \end{array} \right] \rightarrow (x+y)(x-y) = 24$$

$\Rightarrow (x+y)(x-y) = P$

$\Rightarrow (x+y) \cdot S = P$

$\Rightarrow (x + x - S) \cdot S = P$

$\Rightarrow 2x = \frac{P}{S} + S$

$\Rightarrow x = \left(\frac{P}{S} + S\right)/2;$

$\Rightarrow y = x - S$

$x - y = S$

$y = x - S$

```cpp
vector<int>missing_repeated_number(const vector<int> &nums) {
    long long int len = nums.size();

    long long int S = (len * (len+1) ) /2;
    long long int P = (len * (len +1) *(2*len +1) )/6;
    long long int missingNumber=0, repeating=0;

    for(int i=0;i<nums.size(); i++){
       S -= (long long int)nums[i];
       P -= (long long int)nums[i]*(long long int)nums[i];
    }

    missingNumber = (S + P/S)/2;

    repeating = missingNumber - S;

    vector <int> ans;

    ans.push_back(repeating);
    ans.push_back(missingNumber);
    return ans;
}
```

Method - 3: Using XOR

ex- 1 1 2 3 4 6

→ Do XOR of all the elements and elements from (1 to N).

⇒ $(1 \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 6) \wedge (1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6)$

⇒ $1 \wedge 5$

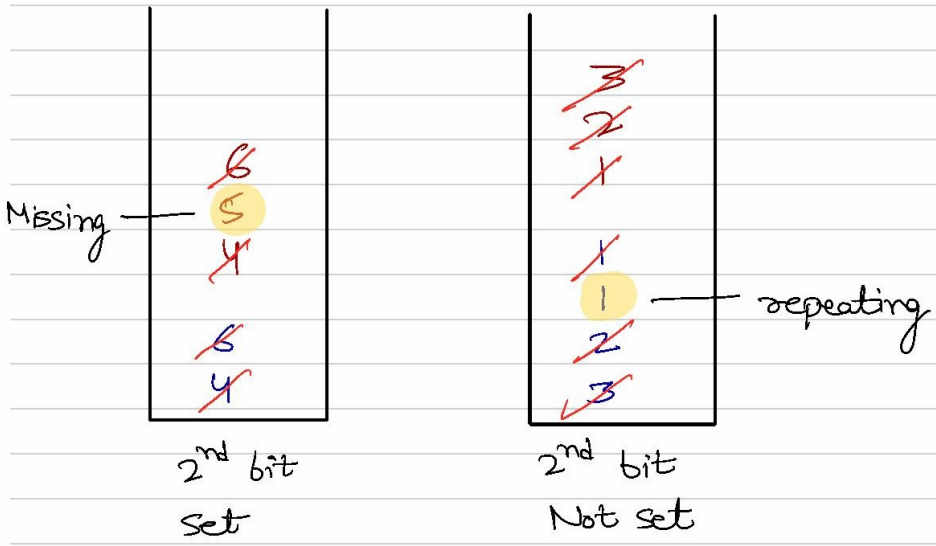
↓ Repeating ↓ missing

⇒ $x \wedge y = 4 = 100$

→ XOR of 1 and 0 is only 1, if XOR of any bit is set means the index bit is either set on x or on y, but not on both.

→ We can take any set bit, but here we will take rightmost set bit and divide the elements based on the set bit or unset bit at the Rightmost set bit in XOR.

→ $x \wedge y = 4$ (100) 2^{nd} bit is set.



```cpp
//third method : Optimal Approach using XO
    vector<int> findErrorNums(vector<int>& nums) {
      int xor1;

    /* Will have only single set bit of xor1 */
    int setbitposition;

    int x = 0; // missing
    int y = 0; // repeated
    int n = nums.size();

    xor1 = nums[0];

    /* Get the xor of all numsay elements */
    for (int i = 1; i < n; i++)
        xor1 = xor1 ^ nums[i];

    /* XOR the previous result with numbers from 1 to n */
    for (int i = 1; i <= n; i++)
        xor1 = xor1 ^ i;

    /* Get the rightmost set bit in setbitposition */
    setbitposition = xor1 & ~(xor1 - 1);

    for (int i = 0; i < n; i++) {
        if (nums[i] & setbitposition)
            /* nums[i] belongs to first set */
            x = x ^ nums[i];
        else
            /* nums[i] belongs to second set */
            y = y ^ nums[i];
    }

    for (int i = 1; i <= n; i++) {
        if (i & setbitposition)
            /* i belongs to first set */
            x = x ^ i;
        else
            /* i belongs to second set */
            y = y ^ i;
    }

    // NB! numbers can be swapped, maybe do a check ?
    int x_count = 0;
    for (int i=0; i<n; i++) {
        if (nums[i]==x)
            x_count++;
    }

    if (x_count==0)
        return {y, x};

    return {x, y};
    }
```

Q 2.) Maximum Product of Word Lengths (Amazon, google)

Given a string array `words`, return *the maximum value of* `length(word[i]) * length(word[j])` *where the two words do not share common letters*. If no such two words exist, return `0`.

Example 1:

```
Input: words = ["abcw","baz","foo","bar","xtfn","abcdef"]
Output: 16
Explanation: The two words can be "abcw", "xtfn".
```

Example 2:

```
Input: words = ["a","ab","abc","d","cd","bcd","abcd"]
Output: 4
Explanation: The two words can be "ab", "cd".
```

Example 3:

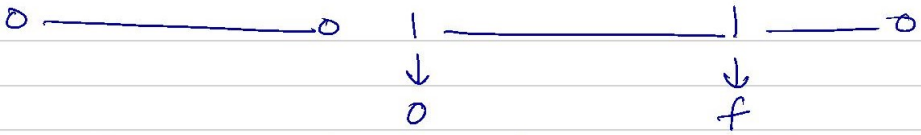
```
Input: words = ["a","aa","aaa","aaaa"]
Output: 0
Explanation: No such pair of words.
```

ex- words = ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]

→ Set the bit corresponding to particular character in the string.

abcw →
- a = set 0th bit
- b = set 1st bit
- c = set 2nd bit
- w = set 22nd bit (119 - 27 = 22)

foo
- f - set 5th bit
- o - set 14th bit
- o - set 14th bit



⇒ Do & of states of both strings, if no character is common, then find the max of ans & word size of both string.

```cpp
int maxProduct(vector<string>& words) {
  int n = words.size();
  int ans = 0;
  vector<int> state(n);
  for(int i=0;i<n;i++){

      for(char ch:words[i]){
        //set the bits corresponding to the particular character
          state[i] |= 1<<(ch-'a');
      }

      for(int j=0;j<i;j++){
          //if no common letter between two strings, then find max
          if(!(state[i] & state[j])){
              int currans = words[i].size()* words[j].size();
              ans = max(ans,currans);
          }
      }
  }
        return ans;
    }
```

Q 3.) Concatenation of Consecutive Binary Numbers

Given an integer n, return the decimal value of the binary string formed by concatenating the binary representations of 1 to n in order, modulo 109 + 7.

ex- N = 1, "1"
output = 1

N = 3,
1 = 1
2 = 10
3 = 11
after concatenation : 11011
output = 27.

ex- N = 4, Range will be from 1 to 4
String = 11011100
⇓ decimal value % (10^9+7)
220

Method 1:-
1. Create String
2. Parse string and find the decimal value.

NOTE :- String creation as well as parsing will consume a lot of time.
→ String concatenation is a costly process.

→ We can optimize the above approach.

NOTE - If we observe, appending binary string at the end of an existing string will cause a left-shift effect by the size of the appended string.

Ex= N=4

OBSERVATION

N	String	Calculation	Value
1	1	—	1
2	1 10	$1 \times 2^2 + 2$ (2 — value of appended string; 2 — size of appended string)	6
3	1 10 11	$6 \times 2^2 + 3$	27
4	1 10 11 100	$27 \times 2^3 + 4$	220

→ if previously calculated value = P

current Number = x

∴ No. of digits in $x = 1 + \log_2(x) = D$

∴ New decimal value = $P \times 2^D + x$

$(P << D) + x$

```
1 int concatenatedBinary(int n) {
2         long long int val = 0;
3         int i=1;
4         int mod = 1e9 + 7;
5         while(i<=n){
6             int digits = (1+log2(i));
7             val = ((val<<digits)%mod + i)%mod;
8             i+=1;
9         }
10        return val;
11    }
```

TC = O(N)

Q 4.) Check if a String Contains all binary codes of size k

Given a binary string `s` and an integer `k`, return `true` *if every binary code of length* `k` *is a substring of* `s`. Otherwise, return `false`.

Example 1:

```
Input: s = "00110110", k = 2
Output: true
Explanation: The binary codes of length 2 are "00", "01", "10" and "11". They can be all found as substrings at indices 0, 1, 3 and 2 respectively.
```

Example 2:

```
Input: s = "0110", k = 1
Output: true
Explanation: The binary codes of length 1 are "0" and "1", it is clear that both exist as a substring.
```

Example 3:

```
Input: s = "0110", k = 2
Output: false
Explanation: The binary code "00" is of length 2 and does not exist in the array.
```

Method 1 :- We can generate all the substrings of size K and compare one by one.

→ It will be very costly.

→ There will be 2^K substrings, so TC will be $O(2^K \times n)$.

Method 2 :- We can check all the substrings in the given string itself.

→ If we generate all the substrings of size K and store them in set, and check the size of set.

→ If size of set == 2^K, then we can say, we can generate all the substrings of size K, hence return true, else false.

ex - 10001110, K = 3

set = {100, 000, 001, 010, 101, 011, 111, 110}

↳ set size = 2^K = 8, so, TRUE.

ex - 01001011, K = 3

set = {010, 001, 011, 100, 101}

set size = 5 ≠ 2^K, so return FALSE

```cpp
bool hasAllCodes(string s, int k) {
    int n = s.size();
    if(k>n) return false;

    unordered_set<string> set;
    for(int i = 0; i <= n - k; i++)
        set.insert(s.substr(i, k));
    return set.size() == (1 << k);
    }
```

Q 5.) Find the Duplicate Number

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and uses only constant extra space.

Method 1: Sorting (NlogN)

→ We can sort the array, start comparing the array elements.

ex - 3 1 2 4 6 5 6 7

Sorting :- 1 2 3 4 5 6 6 7

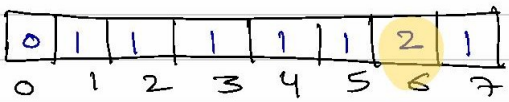
TC = O(NlogN)
SC = O(1)

```cpp
int findDuplicate(vector<int>& nums) {
        int n =  nums.size();
        int ind=0;
        sort(nums.begin(),nums.end());

        for(int i =1;i<n;i++)
        {
            if(nums[i]==nums[i-1])
            {
                return ind =  nums[i];
                break;
            }
        }
        return ind;
        }
```

Method 2 :- Use extra space / Hashing

→ Store the count of every element in a vector.
→ Iterate through the count vector

ex- 3 1 2 4 6 5 6 7

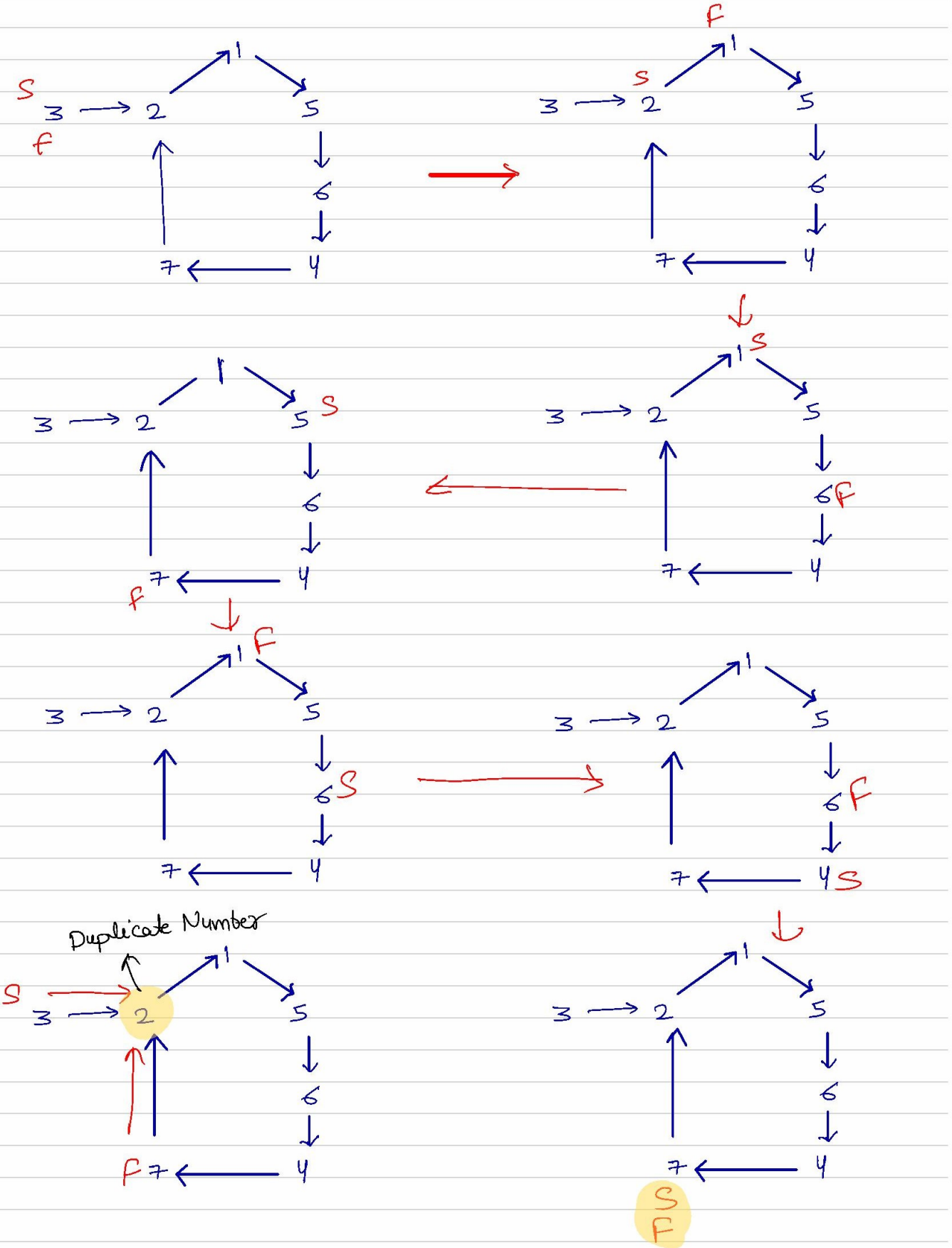


TC = O(N)
SC = O(N)

```cpp
int findDuplicate(vector<int>& nums) {
        int n =  nums.size();
        int ind=0;
        vector<int> cnt(n+1,0);
        for(int i=0;i<n;i++){
            cnt[nums[i]]= cnt[nums[i]]+1;
        }
        for(int i=1;i<=n;i++){
            if(cnt[i]>1) return i;
        }
    return 0;
    }
```

Method 3 : Linked List (slow/fast) cycle method :-

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



```cpp
int findDuplicate(vector<int>& nums) {
        int slow = nums[0];
        int fast = nums[0];
       do{
           slow = nums[slow];
           fast = nums[nums[fast]];
       } while(slow!=fast);

        slow = nums[0];
        while(slow!=fast){
            slow = nums[slow];
            fast = nums[fast];
        }
        return slow;
    }
```

TC = O(N)
SC = O(1)