

BIT

Manipulation

Kapil Yadav

INDEX

- Q 1.) Check if the ith bit is set or not
- Q 2.) Set the ith bit of a number.
- Q 3.) clear the ith bit of a number.
- Q 4.) Remove the last set bit of a number.
- Q 5.) Find whether a number is even or odd
- Q 6.) Check if the number is a power of 2?
- Q 7.) Check if a number is a power of 4?
- Q 8.) Check if a number is a power of 8?
- Q 9.) Check if a number is a power of 16?
- Q 10.) Toggle ith Bit of a number?
- Q 11.) Count the number of set bits in a number
- Q 12.) Find the two non-repeating elements in an array of repeating elements/ Unique Numbers 2
- Q 13.) Convert Uppercase to LowerCase:
- Q 14.). Convert Lowercase to Uppercase
- Q 15.). Invert Alphabet's Case
- Q 16.). Find Letter Position in alphabet
- Q 17.) Given a set of numbers where all elements occur an even number of times except one number, find the odd occurring number.
- Q 18.) Swap two numbers using Bit manipulation:
- Q 19.) Calculate XOR from 1 to n
- Q 20.) Find XOR of numbers from the range [L,R]
- Q 21.) Check whether the number is even or not
- Q 22.) Find the XOR of the XOR of all subsets of an array:
- Q 23.) Count Number of bits to be flipped to convert A to B:
- Q 24.) Find missing number in an array:
- Q 25.) Print the binary representation of decimal number:
- Q 26.) Reverse the bits of a number:
- Q 27.) Swap the ith and Jth bit.
- Q 28.) Swap all even and odd bits
- Q 29.) Copy set bits in a range, toggle set bits in a range:
- Q 30.) Divide two integers without using Multiplication, Division and mod operator:
- Q 31.) One unique rest thrice
- Q 32.) Reduce a Number to 1
- Q 33.) Detect if two integers have opposite sign
- Q 34.) Add 1 to an integer
- Q 35.) Find Xor of a number without using XOR operator
- Q 36.) Determine if two integers are equal without using comparison and arithmetic operators
- Q 37.) Find minimum or maximum of two integers without using branching
- Q 38.) Find missing and repeating number / Set mismatch:
- Q 39.) Maximum Product of Word Lengths
- Q 40.) Check if a String Contains all binary codes of size k
- Q 41.) Find the Duplicate Number

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.

int a = 5, b = 9

$$\begin{array}{r} 00000101 \\ \& 00001001 \\ \hline 00000001 \end{array} = 1$$

$$\begin{array}{r} 00000101 \\ | 00001001 \\ \hline 00001101 \end{array} = 13$$

~~XOR~~

$$\begin{array}{r} 00000101 \\ \wedge 00001001 \\ \hline 00001100 = 12 \end{array}$$

Reference - GFG

Q) ~ operator:-

int a = 5;

// By default it is signed integer, stored in 2's complement 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 $a =$ 
 Signbit → [] 8 zeroes [] 8 zeroes [] 8 zeroes
 $\sim a =$ [] 11111111 11111111 11111111 11111010

NOTE:- MSB in signed integers represents the sign of the number.

O = 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 $\sim a = 00000000 00000000 00000000 00000110$
 $= 6$

→ Since, it is negative, $\sim a = -6$.

b.) Take unsigned int a=5

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

So $a =$ 
 $\sim a =$ 11111111 11111111 11111111 11111010

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

$\sim a = 4294967290$,

c) left shift :-

$$a=8 = 0\ 0001000$$

$$a \ll 3 = 01000000 = 64,$$

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

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

f) Right shift :-

$$a=8 = 00001000$$

$$a \gg 3 = 00000001 = 1$$

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

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



```
1 // C Program to demonstrate use of bitwise operators
2 #include <stdio.h>
3 int main()
4 {
5     // a = 5(00000101), b = 9(00001001)
6     int a = 5, b = 9;
7
8     // The result is 00000001
9     printf("a = %d, b = %d\n", a, b);
10    printf("a&b = %d\n", a & b);
11
12    // The result is 00001101
13    printf("a|b = %d\n", a | b);
14
15    // The result is 00001100
16    printf("a^b = %d\n", a ^ b);
17
18    // The result is 11111010
19    printf("~a = %d\n", a = ~a);
20
21    // The result is 00010010
22    printf("b<<1 = %d\n", b << 1);
23
24    // The result is 00000100
25    printf("b>>1 = %d\n", b >> 1);
26
27    return 0;
28 }
29
```

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

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

$$N = 13, i = 3^{\text{rd}} \text{ bit}$$
$$= (1\ 1\ 0\ 1)_2$$

1 1 0 1
 ^{3rd bit}

We can take mask $0\ 1\ 0\ 0$ and perform & operation, we will get i th bit.

$$\text{mask} = 1 \ll (i-1).$$

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

$4 \& 1 = 0$, so i th bit is set,
if ans = 0, i th 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 i th bit of a number.

$$N = 13, i = 2$$
$$= (1\ 1\ 0\ 1)_2$$

Take mask = $0\ 0\ 1\ 0$ and perform Bitwise |.

$$\text{mask} = 1 \ll (i-1)$$

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



```
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.

$$N = 13$$

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

$$\text{mask} = 1 \ll i-1$$

$$= 1000$$

$$\sim \text{mask} = 0111$$

$$= 1101$$

$$\begin{array}{r} 2 \\ \times 0111 \\ \hline 0101 = 5 \end{array}$$

```
1 int clearBit(int n, int k)
2 {   int mask = (1<<k-1);
3     mask = ~mask;
4     return (n & mask);
5 }
6
7 int main()
8 {
9     int n = 5, k = 1;
10    printf("%d\n", clearBit(n, k));
11    return 0;
12 }
```

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

(Rightmost bit)

$$N = 214$$

$$= 11010110$$

$$\text{Output} = 212$$

$$\begin{array}{l} N = 11010110 \\ \cancel{N-1} = 11010101 \\ \text{Ans} = \underline{11010100} \end{array}$$

$$\begin{array}{l} ① 13 = 1101 \\ 12 = \cancel{1100} \\ \underline{1100} \end{array}$$

$$\begin{array}{l} ② 8 = 1000 \\ 7 = \cancel{0111} \\ \underline{0000} \end{array}$$

```
1 int fun(unsigned int n)
2 {
3     return n & (n - 1);
4 }
```

Find the position of rightmost set bit:

$(n \& \sim(n-1)) \rightarrow$ will return the binary number containing the rightmost set bit as 1.

\rightarrow 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.



```
1 #include <iostream>
2 using namespace std;
3
4 void find_even_odd_using_mod(int num){
5     if(num%2==1){
6         cout<<"Number is odd.";
7     }
8     else {
9         cout<<"Number is even.";
10    }
11 }
12
13 int main() {
14     int num;
15     cin>>num;
16     find_even_odd_using_mod(num);
17 }
18
```

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

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

$$8 \neq 9 \rightarrow \text{odd}$$

num = 16

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

$$16 = 16 \rightarrow \text{even}$$



```
1 #include <iostream>
2 using namespace std;
3
4 void
    find_even_odd_using_divisionandmul(int
        num){
5     int calcnum = (num/2) * 2;
6     if(calcnum==num){
7         cout<<"Number is even.";
8     }
9     else {
10        cout<<"Number is odd.";
11    }
12 }
13
14 int main() {
15     int num;
16     cin>>num;
17     find_even_odd_using_divisionandmul(num);
18 }
19
```

3. Using Bitwise AND Operator:

ex - num = 10

$$\begin{array}{r} 1010 \\ \& 0001 \\ \hline 0000 \end{array} \rightarrow \text{even}$$

num = 9

$$\begin{array}{r} 1001 \\ \& 0001 \\ \hline 0001 \end{array} \rightarrow \text{odd}$$

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

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



```
1 #include <iostream>
2 using namespace std;
3
4 void find_even_odd_using_bitwiseand(int
5     num){
6     if(num & 1){
7         cout<<"Number is odd.";
8     }
9     else {
10        cout<<"Number is even.";
11    }
12 }
13
14 int main() {
15     int num;
16     cin>>num;
17     find_even_odd_using_bitwiseand(num);
18 }
```

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 :- $(\text{num} >> 1) \ll 1$
- Num will be even if $((\text{num} >> 1) \ll 1) == \text{num}$, else odd.

```
1 #include <iostream>
2 using namespace std;
3
4 void find_even_odd_using_leftandright(int num){
5     int calcnum = (num>>1)<<1;
6     if(calcnum==num){
7         cout<<"Number is even.";
8     }
9     else {
10        cout<<"Number is odd.";
11    }
12 }
13
14 int main() {
15     int num;
16     cin>>num;
17     find_even_odd_using_leftandright(num);
18 }
```

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

$$N = 8$$

1 0 0 0

$$2^{N-1} = \underline{0 \ 1 \ 1 \ 1}$$

ans = 0 0 0 0

→ if ans=0, it is power of 2.

```
● ● ●
1 #include<iostream>
2 using namespace std;
3
4 bool powerof2(int n)
5 {   int mask = n-1;
6     return !(n & mask);
7 }
8
9 int main()
10 {
11     int n;
12     cin>>n;
13     if(powerof2(n)){
14         cout<<n<<" is power of 2";
15     }
16     else {
17         cout<<n<<" is not a power of 2";
18     }
19     return 0;
20 }
21
```

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.

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



```
1 #include<iostream>
2 using namespace std;
3
4 bool isPowerOfFour(int n){
5     if(n == 0)
6         return 0;
7     while(n != 1)
8     {
9         if(n % 4 != 0)
10            return 0;
11            n = n / 4;
12    }
13    return 1;
14 }
15
16 int main()
17 {
18 int number;
19 cin>>number;
20 if(isPowerOfFour(number))
21     cout<<number<<" is a power of 4";
22 else
23     cout<<number<<" is not a power of 4";
24 }
25
```

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 negate 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 example : $N = 16$.

$$N = 16$$

\Rightarrow

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

$$N = 5$$

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

1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
A A A A A A A A

= 0xAAAAAA
Hexadecimal



```
1 #include <iostream>
2 using namespace std;
3
4 bool checkPowerOf4(int n){
5     if(n==0) return false;
6     bool powerof2 = !(n & (n - 1));
7
8     //for mask, bits will be 0, in the place which are power of 4
9     //bits will be 1, in the place which are not power of 4
10    //A = 1010, 1 and 4 are powers of 4, so mark them as 0.
11    //every even bit will be a power of 4, so mark them as 0.
12    bool mask = !(n & 0xAAAAAAA);
13    return powerof2 && mask;
14 }
15
16 int main(){
17     int n;
18     cin>>n;
19     if (checkPowerOf4(n)) {
20         cout << n << " is a power of 4";
21     }
22     else {
23         cout << n << " is not a power of 4";
24     }
25     return 0;
26 }
```

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.

```
● ● ●

1
2 #include <iostream>
3 using namespace std;
4
5 bool checkPowerOf4(unsigned n){
6     return !(n & (n - 1))&& (n % 3 == 1);
7 }
8
9 int main()
10 {
11     unsigned n;
12     cin>>n;
13
14     if (checkPowerOf4(n)) {
15         cout << n << " is a power of 4";
16     }
17     else {
18         cout << n << " is not a power of 4";
19     }
20
21     return 0;
22 }
```

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.

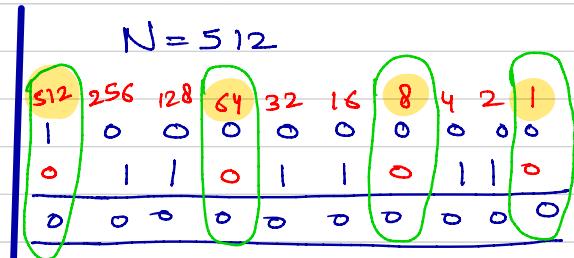
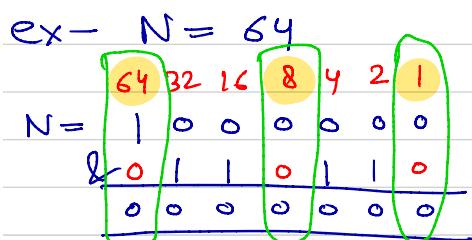
```
● ● ●

1 #include<iostream>
2 using namespace std;
3
4 bool isPowerOfEight(int n){
5     if(n == 0)
6         return 0;
7     while(n != 1)
8     {
9         if(n % 8!= 0)
10            return 0;
11         n = n / 8;
12     }
13     return 1;
14 }
15
16 int main()
17 {
18     int number;
19     cin>>number;
20     if(isPowerOfEight(number))
21         cout<<number<<" is a power of 8";
22     else
23         cout<<number<<" is not a power of 8";
24 }
```

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.



→ 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

$\underbrace{1011}_B \quad \underbrace{0110}_6 \quad \underbrace{110}_D \quad \underbrace{101}_B \quad \underbrace{0110}_6 \quad \underbrace{1101}_D \quad \underbrace{1011}_B \quad \underbrace{0110}_6$

```
1 #include <iostream>
2 using namespace std;
3
4 bool checkPowerOf8(int n){
5     if(n==0) return false;
6     bool powerof2 = !(n & (n - 1));
7
8     //for mask, bits will be 0, in the place which are power of 8
9     //bits will be 1, in the place which are not power of 8
10    //0110110, 1,8,64 are powers of 8.
11    /*starting from 0th position every third bit will be a power of 8,
12    so mark them as 0.
13    (0xB6DB6DB6)16 = (10110110110110110110110110110110)2
14 */
15
16    bool mask = !(n & 0xB6DB6DB6);
17    return powerof2 && mask;
18 }
19
20 int main(){
21     int n;
22     cin>>n;
23     if (checkPowerOf8(n)) {
24         cout << n << " is a power of 8";
25     }
26     else {
27         cout << n << " is not a power of 8";
28     }
29     return 0;
30 }
```

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.



```
1
2 #include <iostream>
3 using namespace std;
4
5 bool checkPowerOf8(unsigned n){
6     return !(n & (n - 1))&& (n % 7 == 1);
7 }
8
9 int main()
10 {
11     unsigned n;
12     cin>>n;
13
14     if (checkPowerOf8(n)) {
15         cout << n << " is a power of 8";
16     }
17     else {
18         cout << n << " is not a power of 8";
19     }
20
21     return 0;
22 }
```

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.

```
1 #include<iostream>
2 using namespace std;
3
4 bool isPowerOfSixteen(int n){
5     if(n == 0)
6         return 0;
7     while(n != 1)
8     {
9         if(n % 16!= 0)
10            return 0;
11         n = n / 16;
12     }
13     return 1;
14 }
15
16 int main()
17 {
18     int number;
19     cin>>number;
20     if(isPowerOfSixteen(number))
21         cout<<number<<" is a power of 16";
22     else
23         cout<<number<<" is not a power of 16";
24 }
25
```

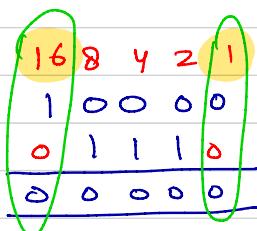
Approach 2:

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

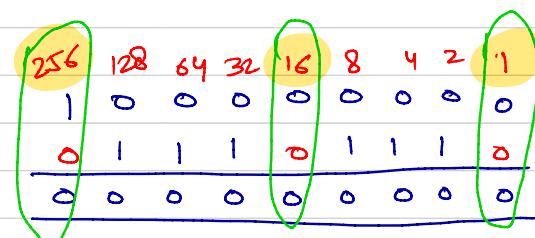
NOTE :- To create mask, those bits who are power of 16, set them as 0, rest as 1.

$$Ex - N = 16$$

$$N = 16$$



$$N = 256$$



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

$$\text{mask} = \underbrace{1110}_E \quad \underbrace{1110}_E$$

```
● ● ●
1 #include <iostream>
2 using namespace std;
3
4 bool checkPowerOf16(int n){
5     if(n==0) return false;
6     bool powerof2 = !(n & (n - 1));
7
8     //for mask, bits will be 0, in the place which are power of 16
9     //bits will be 1, in the place which are not power of 16
10    //0111101110, 1, 16, 256 are powers of 16.
11    /*starting from 0th position every fourth bit will be a power of
16,
12     so mark them as 0.
13     (0xFFFFFFFF)16 = (111011101110111011101110111011101110)
14 */
15
16    bool mask = !(n & 0xFFFFFFFF);
17    return powerof2 && mask;
18 }
19
20 int main(){
21     int n;
22     cin>>n;
23     if (checkPowerOf16(n)) {
24         cout << n << " is a power of 16";
25     }
26     else {
27         cout << n << " is not a power of 16";
28     }
29     return 0;
30 }
```

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.

```
1
2 #include <iostream>
3 using namespace std;
4
5 bool checkPowerOf16(unsigned n){
6     return !(n & (n - 1))&& (n % 15 == 1);
7 }
8
9 int main()
10 {
11     unsigned n;
12     cin>>n;
13
14     if (checkPowerOf16(n)) {
15         cout << n << " is a power of 16";
16     }
17     else {
18         cout << n << " is not a power of 16";
19     }
20
21     return 0;
22 }
```

Q 10.) Toggle ith Bit of a number:

$$N = 15, i = 3$$

1 1 1 1

$$\text{mask} = 1 \ll (i-1)$$
$$= 100$$

$$\begin{array}{r} 1 1 1 \\ \textcolor{red}{\wedge} \quad \underline{0 \ 1 \ 0 \ 0} \\ \text{ans} \Rightarrow \underline{1 \ 0 \ 1 \ 1} \end{array}$$

```
1 #include<iostream>
2 using namespace std;
3
4 int toggleIthBit(int n, int I)
5 {   int mask = 1 << (I-1);
6     return (n^ mask);
7 }
8
9 int main()
10 {
11     int n,I;
12     cin>>n>>I;
13     cout << toggleIthBit(n , I);
14     return 0;
15 }
```

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

input = 1110
output = 3.
cnt = 0;

$N = \begin{array}{r} 1110 \\ \underline{\&} \quad | \\ 0000 \end{array}$

$N = \begin{array}{r} 0111 \\ \underline{\&} \quad | \\ 0001 \end{array}$ cnt = 1

$N = \begin{array}{r} 0011 \\ \underline{\&} \quad | \\ 0001 \end{array}$, cnt = 2

$N = \begin{array}{r} 0001 \\ \underline{\&} \quad | \\ 0001 \end{array}$, cnt = 3

$N = 0$. Stop.

→ Second way, Better in Some cases :-

Brian Kernighan's Algorithm.

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

$N = 13$

$\begin{array}{r} 1101 \\ N = N \& N-1 = 1100 \\ = 1000 \\ = 0000 \end{array}$

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

```
1 unsigned int
  countSetBits(unsigned int n)
2 {
3     unsigned int count = 0;
4     while (n!=0) {
5         if(n& 1==1)
6             count++;
7         n >>= 1;
8     }
9     return count;
10 }
```

TC = $O(\log n)$



```
1 unsigned int countSetBits(int
  n)
2 {
3     unsigned int count = 0;
4     while (n!=0) {
5         n &= (n - 1);
6         count++;
7     }
8     return count;
9 }
```

TC = $O(\text{No. of set bits in number})$.

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

$\text{arr} = [1, 1, 2, 5, 3, 2, 3, 4, 7, 4]$

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) point(a[i]);
}

```

$T.C = O(N^2)$.

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)
        point(it.first);
}

```

$T.C = O(N)$.

$S.C = O(N)$.

③ XOR :-

$$\text{XOR} = 5 \wedge 7.$$

$$\begin{array}{r}
 1 \boxed{0} \ 1 \\
 1 \boxed{1} \ 1 \\
 \hline
 0 \ 1 \ 0
 \end{array}$$

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)

1
5
1
4
4

2nd bit is 1 (x)

2
2
7
3
3

$x = 7$, $result = 2$.

so, $result \wedge x$ will give y .

$$y = (x \wedge result) = 7 \wedge 2 \\ = 5,$$

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=2^2^7^13^3
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 00100000. 32

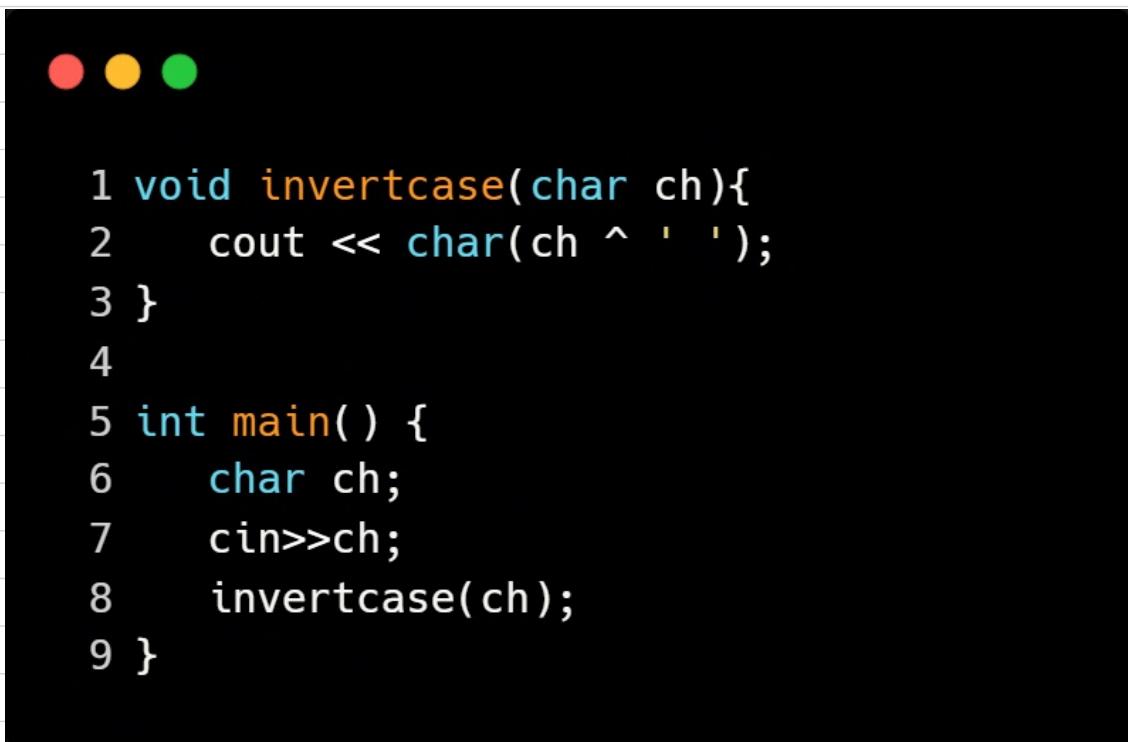
→ The ASCII code of underscore is 01011111

Explanation 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.



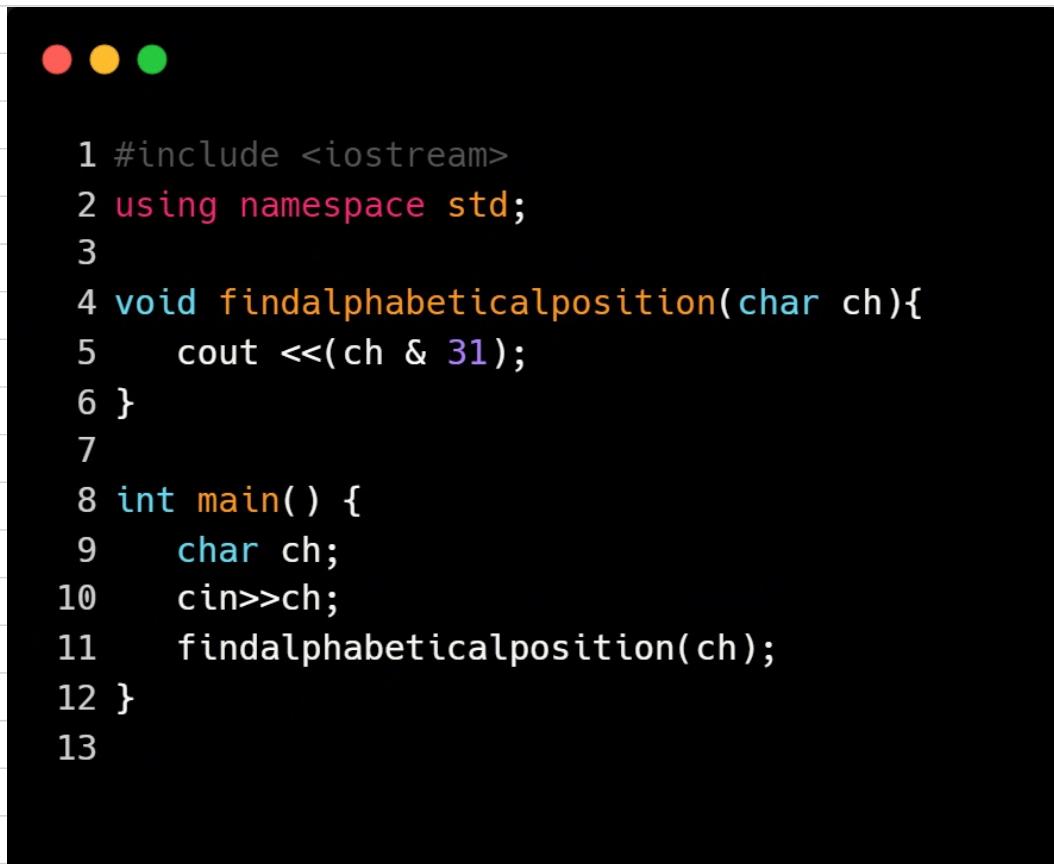
The screenshot shows a terminal window with three colored dots (red, yellow, green) at the top. The terminal displays the following C++ code:

```
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 ASCII code 31 (00011111 in binary).

$('A' \& 31) \Rightarrow 1$ } case will not matter here
 $('c' \& 31) \Rightarrow 3$



```
● ● ●

1 #include <iostream>
2 using namespace std;
3
4 void findalphabeticalposition(char ch){
5     cout <<(ch & 31);
6 }
7
8 int main() {
9     char ch;
10    cin>>ch;
11    findalphabeticalposition(ch);
12 }
13
```

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}.

$$\begin{aligned}&= 4 \wedge 2 \\&= 4 \wedge 2 \wedge 1 \wedge 2 = 4 \\&= 4 \wedge 1 \\&= 4 \wedge 1 \wedge 5 \\&= 4 \wedge 1 \wedge 5 \wedge 6 \\&= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7 \\&= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7 \wedge 5 \\&= 4 \wedge 1 \wedge 5 \wedge 6 \wedge 7 \wedge 6 \\&= 4 \wedge 1 \wedge 7 \wedge 6 \\&= 4 \wedge 1 \wedge 7 \wedge 4 \\&= 1.\end{aligned}$$

```
● ● ●  
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) $a = a \wedge b$

$$\begin{aligned}a &= 5 \wedge 7 \\b &= 5 \wedge 7 \wedge 7 = 5 \\a &= 5 \wedge 7 \wedge 5 \\&= 7\end{aligned}$$

Q 3.) Calculate XOR from 1 to n

$$\begin{aligned}n &= 5 \\output &= 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 = 1\end{aligned}$$

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

n XOR of n

1	= 1	\rightarrow if ($n \% 4 == 0$)	return n;
2	= 1 \wedge 2 = 3	\rightarrow if ($n \% 4 == 1$)	return 1;
3	= 3 \wedge 3 = 0	\rightarrow if ($n \% 4 == 2$)	return n+1;
4	= 0 \wedge 4 = 4	\rightarrow if ($n \% 4 == 3$)	return 0;
5	= 4 \wedge 5 = 1		
6	= 1 \wedge 6 = 7		
7	= 7 \wedge 7 = 0		
8	= 0 \wedge 8 = 8		

```

1 int computeXOR(int n)
2 {
3     // If n is a multiple of 4
4     if (n % 4 == 0)
5         return n;
6
7     // If n%4 gives remainder 1
8     if (n % 4 == 1)
9         return 1;
10
11    // If n%4 gives remainder 2
12    if (n % 4 == 2)
13        return n + 1;
14
15    // If n%4 gives remainder 3
16    return 0;
17 }

```

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

Input $L = 4, R = 8$
 Output $\Rightarrow 4 \oplus 5 \oplus 6 \oplus 7 \oplus 8 = \underline{\underline{8}}$

\therefore Run loop from L to R. $TC = O(N)$ ✗

\rightarrow findXOR(L-1) \rightarrow use above logic = $O(1)$.
 \rightarrow findXOR(R) \rightarrow use above logic = $O(1)$.

\Rightarrow findXOR(L-1) \oplus findXOR(R)
 $\therefore (\cancel{1 \oplus 2 \oplus 3}) \oplus (\cancel{1 \oplus 2 \oplus 3} \oplus 4 \oplus 5 \oplus 6 \oplus 7 \oplus 8)$
 $= 4 \oplus 5 \oplus 6 \oplus 7 \oplus 8 = \underline{\underline{8}}$.

```

1 int findXOR(int n)
2 {
3     int mod = n % 4;
4     if (mod == 0)
5         return n;
6
7     if (mod == 1)
8         return 1;
9     if (mod == 2)
10        return n + 1;
11    if (mod == 3)
12        return 0;
13 }
14
15 int findXOR(int l, int r) {
16     return (findXOR(l - 1) ^ findXOR(r));
17 }

```

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	
<hr/>	
0	

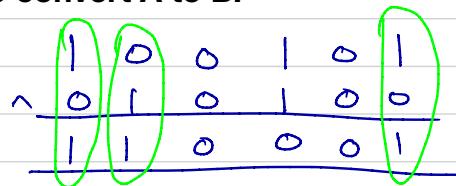
→ XOR will be non zero, only if there is only one element in the array.
return arr[0] in that case.



```
1 int findXOR(int Set[], int n)
2 {
3     if (n == 1)
4         return Set[0];
5     else
6         return 0;
7 }
8
9 int main()
10 {
11     int Set[] = { 1, 2, 3 };
12     int n = sizeof(Set) / sizeof(Set[0]);
13     printf("XOR of XORs of all subsets is %d\n",
14         findXOR(Set, n));
15     return 0;
16 }
```

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 set bits to be flipped.
- Count No. of set bits in XOR of A & B using Brian Kernighan's algorithm $n \& (n-1)$.

```
1 #include<iostream>
2 using namespace std;
3
4 int countflipbits(int A,int B){
5     int n = A^B;
6     int count = 0;
7     while(n){
8         n = n & (n-1);
9         count++;
10    }
11    return count;
12 }
13
14 int main()
15 {
16     int A,B;
17     cin>>A>>B;
18     cout<<"No of bits to be flip to Convert A
19         to B is : "<<countflipbits(A,B);
20     return 0;
21 }
```

Q 8.) Find missing number in an array:

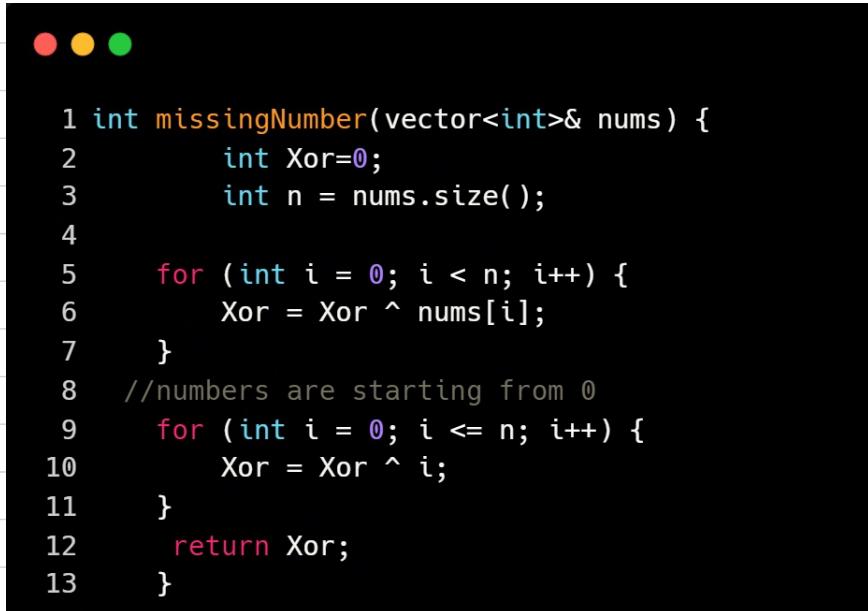
Given an array `nums` containing n distinct numbers in the range $[0, n]$, return the only number in the range that is missing from the array.

→ 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.

$$\begin{aligned} \text{nums} &= [3, 0, 1] \\ &= (\cancel{3} \wedge \cancel{0} \wedge \cancel{1}) \wedge (\cancel{0} \wedge \cancel{1} \wedge \cancel{2} \wedge \cancel{3}) \\ &= 2. \end{aligned}$$



```
1 int missingNumber(vector<int>& nums) {
2     int Xor=0;
3     int n = nums.size();
4
5     for (int i = 0; i < n; i++) {
6         Xor = Xor ^ nums[i];
7     }
8     //numbers are starting from 0
9     for (int i = 0; i <= n; i++) {
10        Xor = Xor ^ i;
11    }
12    return Xor;
13 }
```

Q 9) Print the binary representation of decimal number:

81

0

mask = 1 << 30;

8

0

10

2

1

2

```
1 #include<iostream>
2 using namespace std;
3
4 void decitobinary(int num){
5 for(int i=31;i>=0;i--)
6 {
7     int mask = 1<<i;
8     if(num & mask){
9         cout<<"1";
10    } else cout<<"0";
11 }
12 }
13
14 int main()
15 {
16     int num;
17     cin>>num;
18     cout<<"Binary representation of decimal
number is ";
19     decitobinary(num);
20     return 0;
21 }
22
```

Q 10.) Reverse the bits of a number:

Take ans as 0

→ 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).

A diagram showing a horizontal beam. At the left end, there is a fixed support labeled 'O'. At the right end, there is a roller support labeled 'O''. The beam is straight between the supports.

$$\begin{aligned} \text{Ans} &= 0 \mid 0 \\ &= 0 \end{aligned}$$

$\beta = \beta >> 1$.

$$2 \quad n = \frac{00000000000000000000000000001010101}{l} \quad l$$

ans << =1;

$$\text{ans} = \begin{cases} \text{ans} \\ 0 \end{cases} \quad |(n \& 1)$$

$$\text{ans} = 1$$

$$\begin{array}{r}
 \text{ans} \ll= 1 \\
 \text{ans} = \text{ans} | (n \& 1) \\
 = 2 | 0 \\
 = 2
 \end{array}
 \quad
 \begin{array}{r}
 00000 \quad \cdots \cdots | 0 \quad = 2
 \end{array}$$

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



```
1 uint32_t reverseBits(uint32_t n) {  
2     int ans = 0;  
3     for(int i=0;i<32;i++)  
4     {  
5         ans <=> 1;  
6         ans = ans|(n&1);  
7         n >>= 1;  
8     }  
9     return ans;  
10 }
```

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 \Rightarrow num = 43, i = 2, j = 5.

$0\ 0\ 1\ 0\ 1\ 0\ 1\ 1$, output = 57 ($0\ 0\ 1\ 1\ 1\ 0\ 0\ 1$)

→ first move the bit value at i^{th} & j^{th} bit

Let A is the bit value at ith bit:-

$$n = 00101011$$

$$n \gg l = 0 \ 0 \ 0 \mid 0 \mid 0 \mid 0$$

$$(n \gg 1) \& 1 = \underline{\hspace{10em}}$$

A = 00000001

move j^{th} bit to rightmost

$$n = 00101^0011$$

$$n \gg 4 = 0 \ 0 \ 0 \ 0 \ 0 \ 0 \mid 0$$

$$(n>4) \Delta l = \frac{1}{B = \overbrace{\quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0}^{\Delta l}}$$

$$\text{temp} = A \wedge B = 00000001$$

Now, $A \wedge B = \text{temp}$, $\text{temp} \wedge B = A$, $\text{temp} \wedge A = B$

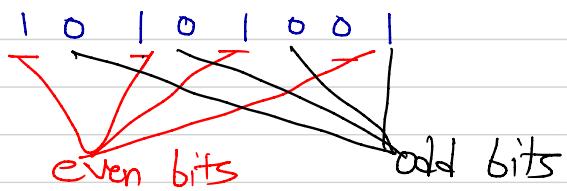
→ Move temp first at i^{th} position & then at j^{th} position.

$\text{temp} = 0\ 0\ 0\ 0\ 0\ 0\ 0\ 1$
 $\text{temp} \ll 1 = 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0$ ← ith
 In $\underline{\quad 0\ 0\ 1\ 0\ 1\ 0\ 1\ 1}$ A
 $\underline{\quad 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1}$ B → B

```
1 #include<iostream>
2 using namespace std;
3
4 int swapBits(unsigned int n, unsigned int i, unsigned int j)
5 {   unsigned int A = ((n>>(i-1)) & 1);
6     unsigned int B = ((n>>(j-1)) & 1);
7
8     unsigned int temp = A ^ B;
9     n = n ^ (temp<< (i-1));
10    n = n ^ (temp<<(j-1));
11    return n;
12 }
13
14 int main()
15 {   int n,i,j;
16     cin>>n>>i>>j;
17     int res = swapBits(n,i,j);
18     printf( "Result = %d ", res);
19     return 0;
20 }
```

Q 12.) Swap all even and odd bits

Ex -



$$\begin{array}{r} 10101001 \\ \text{even mask} \quad \underline{10101010} \\ \text{even bits} = \underline{10101010} \end{array}$$

$$\begin{array}{r} 10101001 \\ \text{odd mask} \quad \underline{01010010} \\ \text{odd bits} = \underline{00000010} \end{array}$$

$$\begin{aligned} \text{even bits} &>= 1; \\ \text{even bits} &= 01010100 \\ \text{result} &= 01010110 \end{aligned}$$

$$\begin{aligned} \text{odd bits} &<= 1 \\ \text{odd bits} &= 00000010 \end{aligned}$$



```
1 #include<iostream>
2 using namespace std;
3
4 unsigned int swapevenoddbits(unsigned int N)
5 {
6     unsigned int even_bits = N & 0xAAAAAAA;
7     unsigned int odd_bits = N & 0x5555555;
8
9     // Right shift even bits
10    even_bits >>= 1;
11
12    // Left shift odd bits
13    odd_bits <<= 1;
14
15    // Combine even and odd bits
16    unsigned int result = even_bits | odd_bits;
17    return result;
18 }
19
20 int main()
21 {
22     unsigned int N;
23     cin>>N;
24
25     cout<<swapevenoddbits(N);
26     return 0;
27 }
28
```

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 →

$$\begin{array}{r} \text{A} = 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \\ \text{B} = 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \\ \text{L} = 3 \\ \text{R} = 7 \end{array}$$

8 7 6 5 4 3 2 1

If we create a mask like

$$\text{mask} = 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0$$

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

$$\begin{array}{r} \text{A} = 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 0 \\ \text{mask} = 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \\ \hline \text{Result} = 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \end{array}$$

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

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

① left shift 1 till $(2-l+1)$

$$\begin{array}{r} \text{mask} = 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \\ (\text{mask}-1) = 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \end{array}$$

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

$$(\text{mask}-1) \ll (l-1) = 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0$$

② Second way to create mask is :-

$$\begin{aligned}\text{right mask} &= (1 \ll \sigma) - 1 \\ &= \underline{\quad 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \quad} \\ \text{left mask} &= 1 \ll (l-1) - 1 \\ &= \underline{\quad 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \quad}\end{aligned}$$

] → $\begin{array}{r} 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \\ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \\ \hline 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0 \end{array}$



```
1 #include<iostream>
2 using namespace std;
3
4 int CopySetBit(int X, int Y, int l, int r){
5
6     //only 1 can overflow in case of r = 32 and l=1
7     //we need to take lll
8     int maskLength = (1ll<<(r-l+1)) - 1;
9
10    int mask = ((maskLength)<<(l-1)) & X;
11
12    Y = Y | mask;
13    return Y;
14 }
15
16 int main()
17 {
18     unsigned int A,B,l,r;
19     cin>>A>>B>>l>>r>>l;
20
21     cout<<CopySetBit(A,B,l,r);
22     return 0;
23 }
```

2.) Toggle bits in range :-

→ Mask computation is same as copy bits in range

There are two ways above.

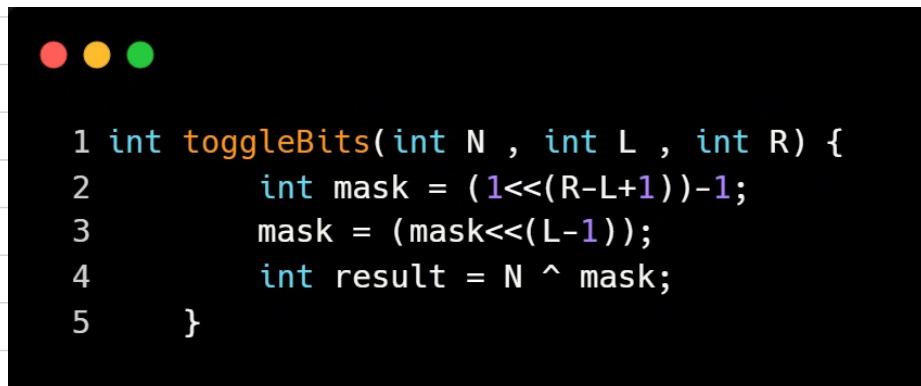
① $\text{mask} = (1 \ll (r-l+1)) - 1;$
 $\text{mask} = (\text{mask} \ll (l-1));$

② $\text{mask} = ((1 \ll r) - 1) \wedge ((1 \ll (l-1)) - 1).$

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

$\text{num} = \text{num} \wedge \text{mask}.$

Cx-
mask 1 1 0 1 1 0 1 1, $i = 2, j = 6$
 0 0 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

$$\text{Quotient} = \left\lfloor \frac{10}{3} \right\rfloor = 3.$$

Approach 1: Repeated subtraction:-

Keep subtracting the divisor from dividend until dividend becomes less than divisor.

So, dividend will get reduced to become remainder & no. of times will subtract divisor from dividend will become the quotient.

Ex.

$$\begin{aligned} 10 - 3 &= 7 \\ 7 - 3 &= 4 \\ 4 - 3 &= 1 < 3 \end{aligned} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} 3 \quad q = 3$$

Time Complexity = $O(\text{dividend})$

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 $\text{divisor} \ll i$ 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 $(\text{temp} + (\text{divisor} \ll i))$ 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 except for one, which appears exactly once. Find the single element and return it.

ex- $\text{nums} = [1, 1, 5, 1]$
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 ($\text{num}[0]$) = $\text{num}[1]$)

2. Right Boundary condition :-

ex - 4 4 4 5 5 5 6
→ check if ($\text{nums}[n-2]$) = $\text{nums}[n-1]$)

3. Somewhere except Boundary

ex - 1 1 1 2 3 3 3

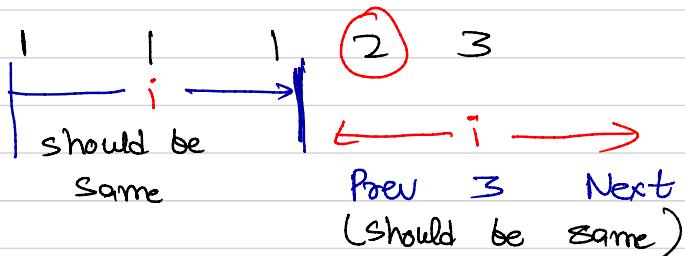
run a while loop, start comparing from index 1,

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



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

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



```
1 //better approach : Sorting + linear traversal
2     int singleNumber(vector<int>& nums) {
3         int n = nums.size();
4         sort(nums.begin(),nums.end());
5         if(n<3) return nums[0];
6
7         //check boundary conditions
8         //2 4 4 4 5 5 5
9         if(nums[0]!=nums[1]) return nums[0];
10        //another boundary condition
11        // 4 4 4 5 5 5 6
12        if(nums[n-2]!=nums[n-1]) return nums[n-1];
13        int i=1;
14        while(i<n){
15            if(nums[i]!=nums[i-1]){
16                return nums[i-1];
17            }
18            i+=3;
19        }
20        return -1;
21    }
```

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

\downarrow \downarrow
Sorting Traversing

Method 3: Unordered_map

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



```
1 //3rd approach : unordered_map : TC : O(N) , SC : O(N)
2     int singleNumber(vector<int>& nums) {
3         int n = nums.size();
4         unordered_map<int,int> mp;
5         for(int i=0;i<n;i++){
6             mp[nums[i]]++;
7         }
8
9         for(auto it : mp){
10             if(it.second==1){
11                 return it.first;
12             }
13         }
14
15     return -1;
16 }
```

$$TC = O(N)$$

$$SC = O(N)$$

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	}	2
0	1	0		
0	1	0	}	3
0	1	1		
1	0	1	}	5
1	0	1		
1	0	1		

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

```

1 //4th approach : counting Set bits in every number
array
2 //TC: O(32N) every case not better than 3rd approach
3     int singleNumber(vector<int>& nums) {
4         int n = nums.size();
5         int result = 0;
6
7         int leftshift, numberofsetbits;
8
9         // Iterate through every bit
10        for(int i = 0; i < 32; i++) {
11
12            numberofsetbits = 0;
13            leftshift = (1 << i);
14            for (int j = 0; j < n; j++) {
15                if (nums[j] & leftshift)
16                    numberofsetbits++;
17            }
18            if ((numberofsetbits % 3) != 0)
19                result+= leftshift;
20        }
21
22        return result;
23    }

```

Method : 5 Bit Manipulation:

→ We will use two variables ones and twos.

ones will store the element, which is occurring only once.

twos will store the element, which is occurring twice.

→ When an element is occurring thrice, ones and twos will store 0.

ex- [2 2 2 3]

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

$$\boxed{\begin{aligned} \text{ones} &= (\text{ones} \wedge \text{ele}) \vee (\neg \text{twos}); \\ \text{twos} &= (\text{twos} \wedge \text{ele}) \vee (\neg \text{ones}); \end{aligned}}$$

$$\begin{array}{r} \text{ones} = 00 \\ \wedge 10 \\ \hline 10 \\ \times 11 \\ \hline 10 \end{array}$$

ones = 2

$$\begin{array}{r} \text{twos} = 00 \\ \wedge 10 \\ \hline 10 \\ \times 01 \\ \hline 00 \end{array}$$

twos = 0

$$\begin{array}{r} i=1, \text{ ones} = 10 \\ \wedge 10 \\ \hline 00 \\ \times 11 \\ \hline 00 \end{array}$$

ones = 0

$$\begin{array}{r} \text{twos} = 00 \\ \wedge 10 \\ \hline 10 \\ \times 11 \\ \hline 10 \end{array}$$

twos = 2

$$\begin{array}{r} i=2, \text{ ones} = 00 \\ \wedge 10 \\ \hline 10 \\ \times 01 \\ \hline 00 \end{array}$$

$$\begin{array}{r} \text{twos} = 10 \\ \wedge 10 \\ \hline 00 \\ \times 11 \\ \hline 00 \end{array}$$

$$\begin{array}{r} i=3, \text{ ones} = 00 \\ \wedge 11 \\ \hline 11 \\ \times 11 \\ \hline 11 \end{array}$$

ones = 3

$$\begin{array}{r} \text{twos} = 00 \\ \wedge 11 \\ \hline 11 \\ \times 00 \\ \hline 00 \end{array}$$

twos = 0



```
1 //5th approach : Bit manipulation (Not intuitive)
2 //TC: O(N)
3 //our answer will store in ones
4     int singleNumber(vector<int>& nums) {
5         int ones = 0;
6         int twos = 0;
7
8         for(auto ele: nums){
9             ones = (ones^ele) & (~twos);
10            twos = (twos^ele) & (~ones);
11        }
12        return ones;
13    }
```

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:

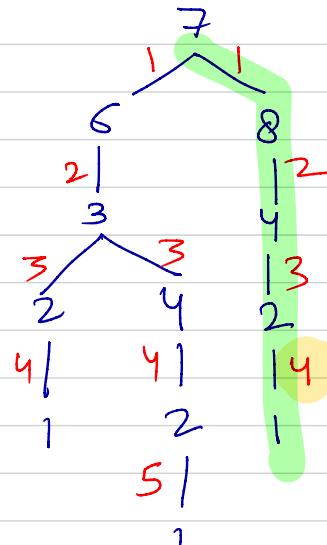
Operation 1: If the number is even then divide the number by 2

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

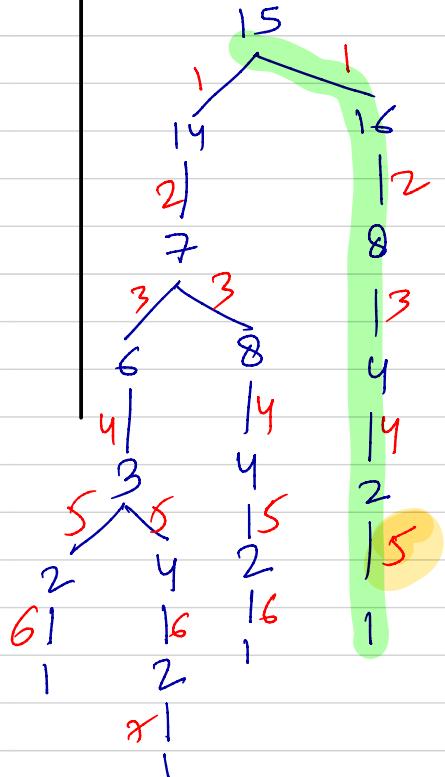
ex-1.) $N = 4$



$N = 7$



$N = 15$



Method 1:

We can use recursion until $n=1$, for even numbers, we can return $1 + \text{funct}(n/2)$,

for odd numbers, we can return : $1 + \min(\text{funct}(n-1), \text{funct}(n+1))$.

```
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. $4x$

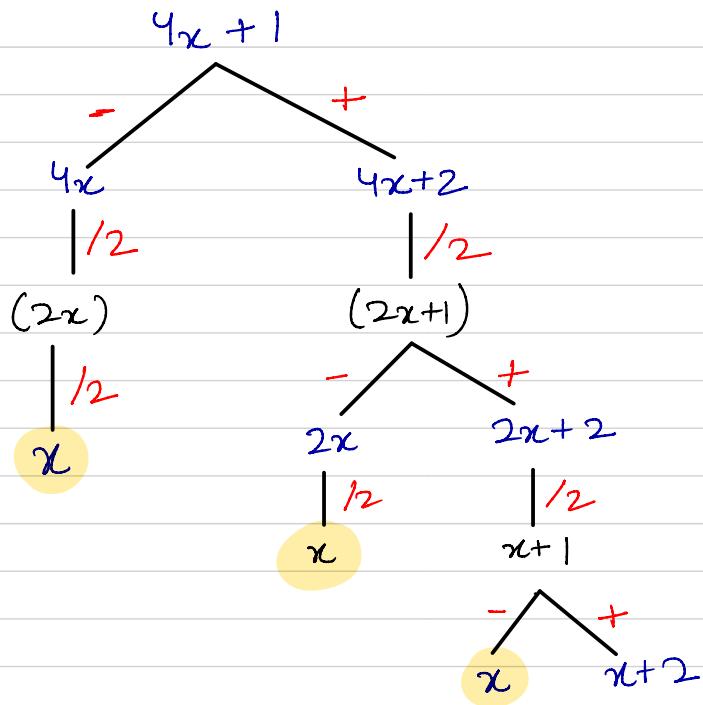
Odd number can be represent as $4x+1$ and $4x+3$.

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.

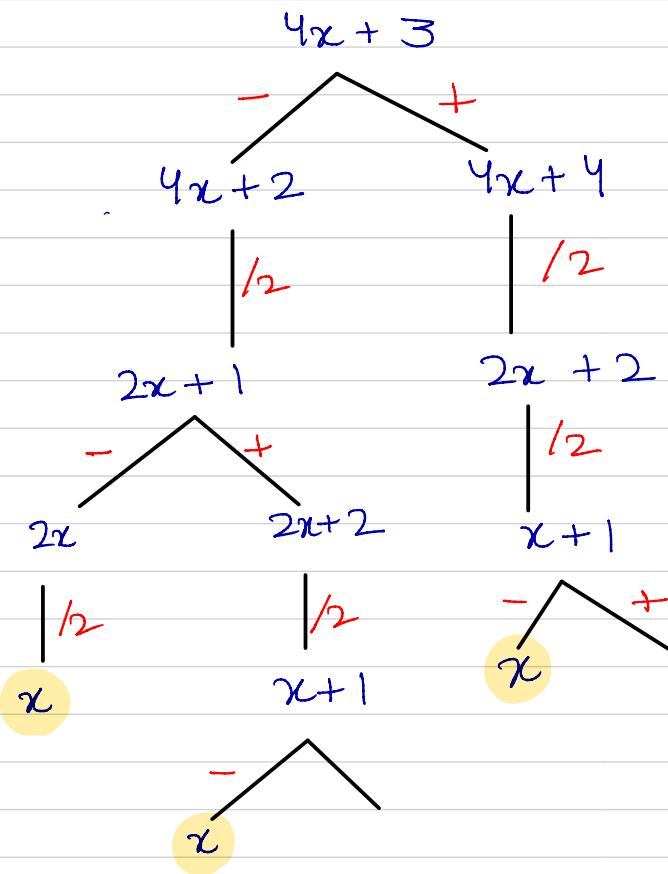
→ Lets take some examples:-

1.) Take $4x+1$ case first ,



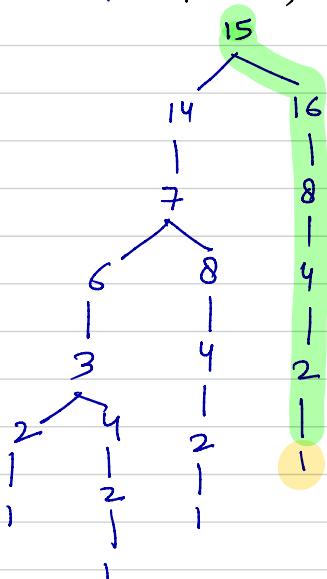
→ for case $4x+1$, we will reach to the 1, in minimum steps using $n-1$ at every step having odd number.

2) Take $4x + 3$ case

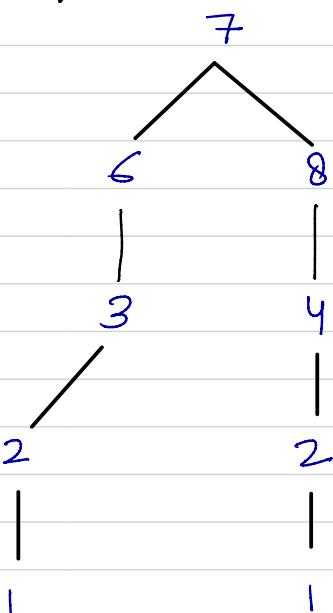


\Rightarrow In above case if $x-1$ will give min answer then both - & + takes same step, but if $x+1$ will give min ans then + path takes min step, so + path is better than -.

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 stored 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 numbers will give negative number, so

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

```
● ● ●  
1 int main()  
2 {  
3     int num1, num2;  
4     cin >> num1 >> num2;  
5  
6     if ((num1 ^ num2) < 0){  
7         cout << "Opposite Sign" << endl;  
8     }  
9     else cout << "Not Opposite sign" << endl;  
10  
11    return 0;  
12 }
```

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 $(x+1) \Rightarrow -(\sim x)$.

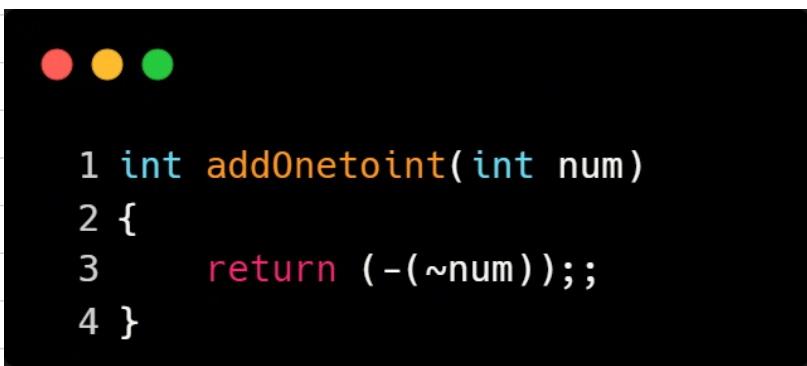
ex- 4

$\begin{array}{r} 00000000 \\ \sim | \underline{\quad} \quad | \quad | \quad) \rightarrow 1111011 = \end{array}$

2's complement of $\sim x =$

$\begin{array}{r} 00000000 \\ \quad \quad \quad 00000001 = (-5) \end{array}$

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



```
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
17 }
```

Method 2: we can use the property of xor.

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



```
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 $b \wedge ((a \wedge b) \& - (a < b))$.
 - If 'b' is minimum ' $a < b$ ' comes out to be all zeroes. ($a \wedge b$) & '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. ($a \wedge b$) & '1' comes out to ($a \wedge b$). Therefore expression value comes out to be 'a' finally which is the minimum.
- We can write maximum as $a \wedge ((a \wedge b) \& - (a < b))$.
 - If 'a' is maximum ' $a < b$ ' comes out to be all zeroes. ($a \wedge b$) & '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. ($a \wedge b$) & '1' comes out to ($a \wedge b$). Therefore expression value comes out to be 'b' finally which is the maximum.

Method-2

- We can write the minimum as $b + ((a - b) \& ((a - b) >> (\text{noOfBitsInInt} - 1)))$. 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 $a - ((a - b) \& ((a - b) >> (\text{noOfBitsInInt} - 1)))$. 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.

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.

```
1 vector<int> findErrorNums(vector<int>& arr) {
2     int n = arr.size();
3     vector<int> temp(n+1,0);
4     vector<int> ans(2,0);
5     for(int i=0;i<n;i++){
6         temp[arr[i]]++;
7     }
8     for(int i=1;i<=n;i++){
9         if(temp[i]==0){
10             ans[1] = i;
11         }
12         if(temp[i]==2){
13             ans[0] = i;
14         }
15     }
16     return ans;
```

Method -2 :

Ex - 1 1 2 3 4 5

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

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

$\Rightarrow S$ - array elements

$$(1+2+3+4+5+6) - (1+2+3+4+5+1) \\ 5 - 1 \\ = x - y$$

$\Rightarrow S^2 - (\text{array elements})^2$

$$= 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 - (1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 1^2) \\ = 25 - 1 (x^2 - y^2) \\ = 24$$

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

$$\Rightarrow (x+y) \underbrace{(x-y)}_{S} = P$$

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

$$x-y = S \\ y = x-S$$

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

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

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

$$\Rightarrow y = x - S$$



```
1 vector<int>missing_repeated_number(const vector<int> &nums) {
2     long long int len = nums.size();
3
4     long long int S = (len * (len+1)) /2;
5     long long int P = (len * (len +1) *(2*len +1)) /6;
6     long long int missingNumber=0, repeating=0;
7
8     for(int i=0;i<nums.size(); i++){
9         S -= (long long int)nums[i];
10        P -= (long long int)nums[i]*(long long int)nums[i];
11    }
12
13    missingNumber = (S + P/S)/2;
14
15    repeating = missingNumber - S;
16
17    vector <int> ans;
18
19    ans.push_back(repeating);
20    ans.push_back(missingNumber);
21
22 }
```

Method - 3: Using XOR

ex- 1 1 2 3 4 6

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

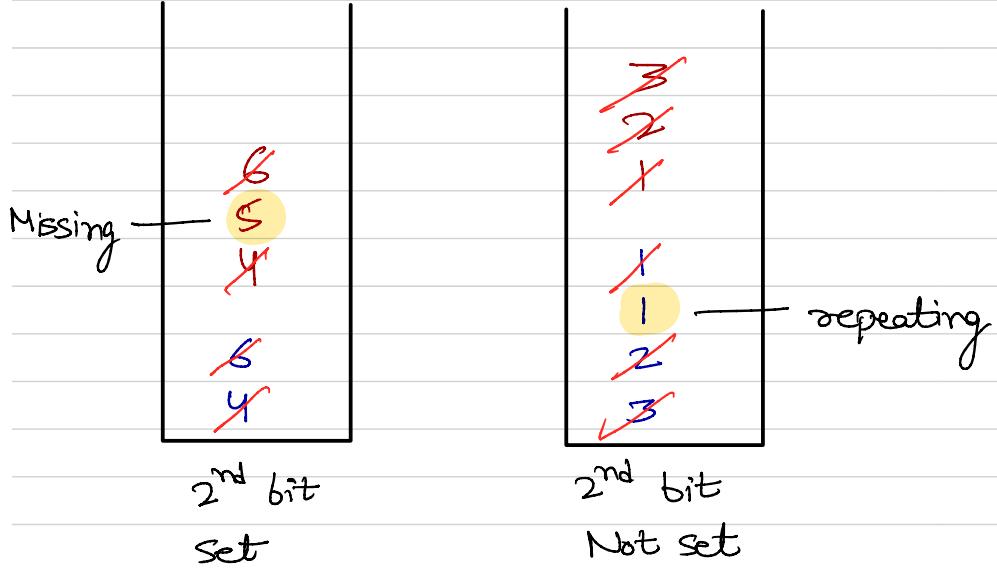
$$\Rightarrow (1 \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 6) \wedge (1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6)$$

$$\begin{array}{ccc} \Rightarrow & 1 & \wedge \quad 5 \\ & \downarrow & \downarrow \\ & \text{Repeating} & \text{missing} \\ & \downarrow & \downarrow \\ \Rightarrow & x & \wedge \quad y = 4 = 100 \end{array}$$

→ 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.

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





```
1 //third method : Optimal Approach using XOR
2     vector<int> findErrorNums(vector<int>& nums) {
3         int xor1;
4
5         /* Will have only single set bit of xor1 */
6         int setbitposition;
7
8         int x = 0; // missing
9         int y = 0; // repeated
10        int n = nums.size();
11
12        xor1 = nums[0];
13
14        /* Get the xor of all numsay elements */
15        for (int i = 1; i < n; i++)
16            xor1 = xor1 ^ nums[i];
17
18        /* XOR the previous result with numbers from 1 to n */
19        for (int i = 1; i <= n; i++)
20            xor1 = xor1 ^ i;
21
22        /* Get the rightmost set bit in setbitposition */
23        setbitposition = xor1 & ~(xor1 - 1);
24
25        for (int i = 0; i < n; i++) {
26            if (nums[i] & setbitposition)
27                /* nums[i] belongs to first set */
28                x = x ^ nums[i];
29            else
30                /* nums[i] belongs to second set */
31                y = y ^ nums[i];
32        }
33
34        for (int i = 1; i <= n; i++) {
35            if (i & setbitposition)
36                /* i belongs to first set */
37                x = x ^ i;
38            else
39                /* i belongs to second set */
40                y = y ^ i;
41        }
42
43        // NB! numbers can be swapped, maybe do a check ?
44        int x_count = 0;
45        for (int i=0; i<n; i++) {
46            if (nums[i]==x)
47                x_count++;
48        }
49
50        if (x_count==0)
51            return {y, x};
52
53        return {x, y};
54    }
```

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`

→	<code>a</code>	=	Set 0 th bit
	<code>b</code>	=	Set 1 st bit
	<code>c</code>	=	Set 2 nd bit
	<code>w</code>	=	Set 22 nd bit

(119 - 27 = 22)

`0000000000000000` | 1 111] save state for
 ↓ 18 zeroes ↓↓↓ c b a all strings like
 w

`foo`

<code>f</code>	-	Set 5 th bit
<code>o</code>	-	Set 14 th bit
<code>o</code>	-	Set 14 th bit

0 ————— 0 | 1 ————— 1 ————— 0
 ↓ ↓ ↓
 0 f

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



```
1 int maxProduct(vector<string>& words) {
2     int n = words.size();
3     int ans = 0;
4     vector<int> state(n);
5     for(int i=0;i<n;i++){
6
7         for(char ch:words[i]){
8             //set the bits corresponding to the particular character
9             state[i] |= 1<<(ch-'a');
10        }
11
12        for(int j=0;j<i;j++){
13            //if no common letter between two strings, then find max
14            if(!(state[i] & state[j])){
15                int currans = words[i].size()* words[j].size();
16                ans = max(ans,currans);
17            }
18        }
19    }
20    return ans;
21 }
```

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 $10^9 + 7$.

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	$1 \times 2^2 + 2$	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 = $P \times 2^D + x$

$(P \ll D) + x$

```
● ● ●

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 - 1000 101110, 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 $\neq 2^k$, so return FALSE



```
1 bool hasAllCodes(string s, int k) {
2     int n = s.size();
3     if(k>n) return false;
4
5     unordered_set<string> set;
6     for(int i = 0; i <= n - k; i++)
7         set.insert(s.substr(i, k));
8     return set.size() == (1 << k);
9 }
```

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 ($N \log N$)

→ 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

$$\begin{aligned} TC &= O(N \log N) \\ SC &= O(1) \end{aligned}$$



```
1 int findDuplicate(vector<int>& nums) {
2     int n =  nums.size();
3     int ind=0;
4     sort(nums.begin(),nums.end());
5
6     for(int i =1;i<n;i++)
7     {
8         if(nums[i]==nums[i-1])
9         {
10             return ind =  nums[i];
11             break;
12         }
13     }
14     return ind;
15 }
```

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

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

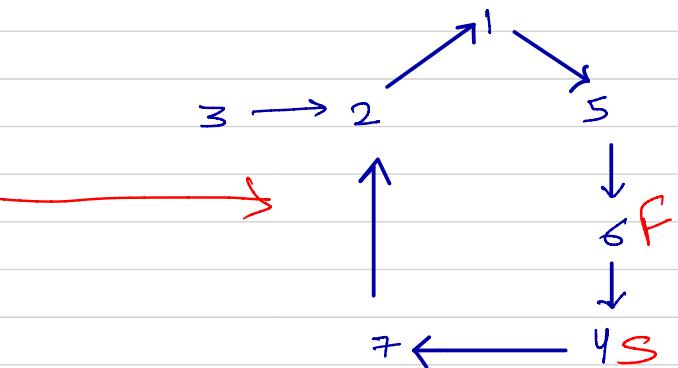
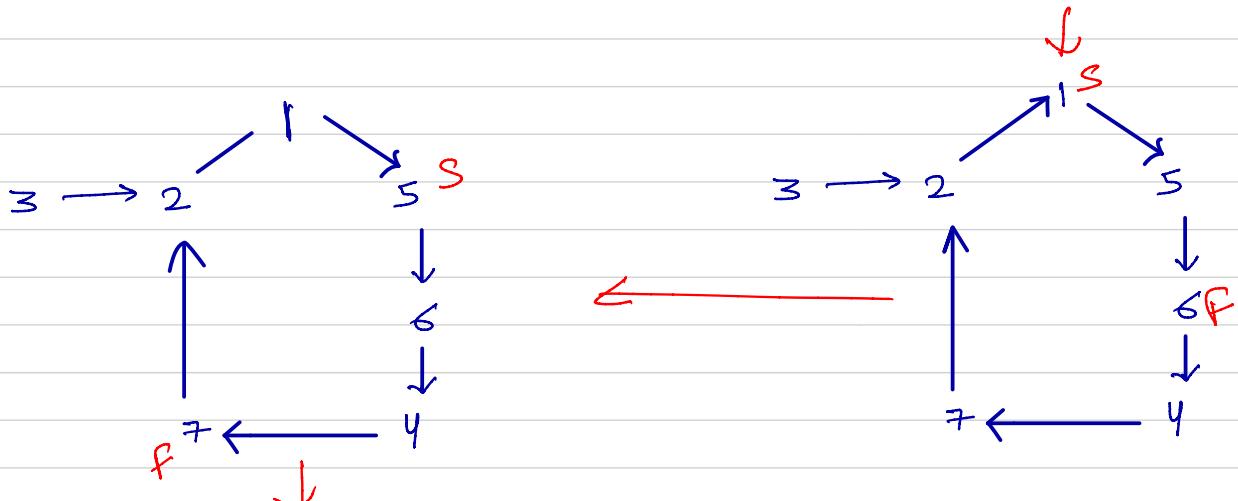
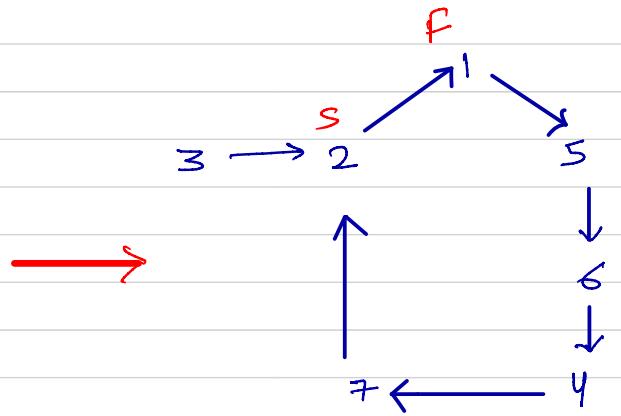
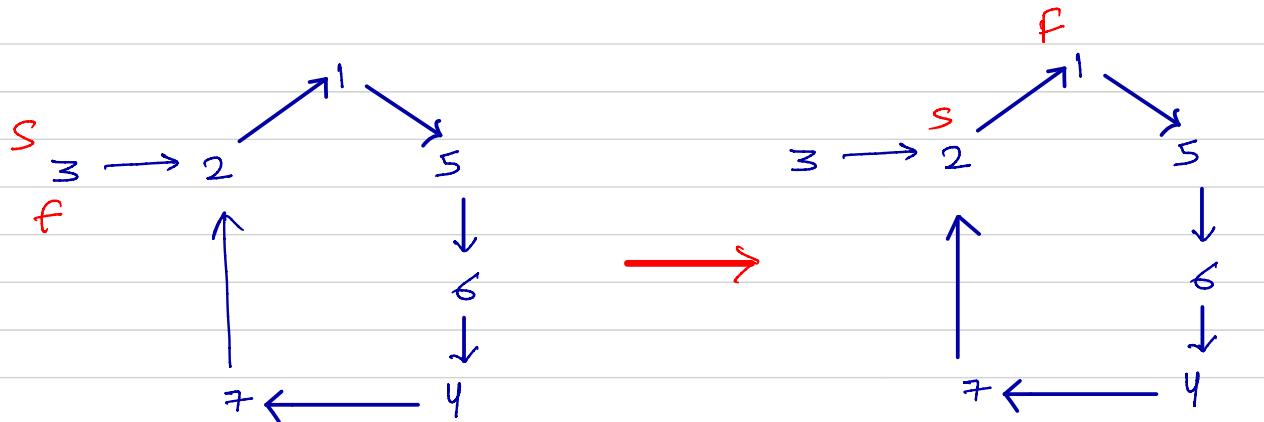
$$TC = O(N)$$

$$SC = O(N)$$

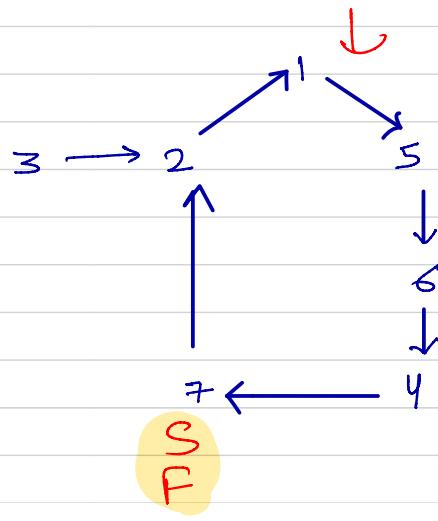
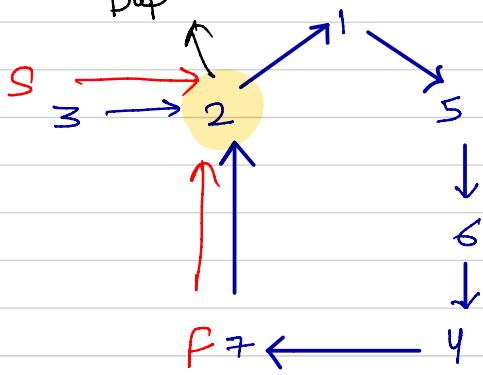
```
● ● ●
1 int findDuplicate(vector<int>& nums) {
2     int n =  nums.size();
3     int ind=0;
4     vector<int> cnt(n+1,0);
5     for(int i=0;i<n;i++){
6         cnt[nums[i]]= cnt[nums[i]]+1;
7     }
8     for(int i=1;i<=n;i++){
9         if(cnt[i]>1) return i;
10    }
11    return 0;
12 }
```

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



Duplicate Number



```
1 int findDuplicate(vector<int>& nums) {  
2     int slow = nums[0];  
3     int fast = nums[0];  
4     do{  
5         slow = nums[slow];  
6         fast = nums[nums[fast]];  
7     } while(slow!=fast);  
8  
9     slow = nums[0];  
10    while(slow!=fast){  
11        slow = nums[slow];  
12        fast = nums[fast];  
13    }  
14    return slow;  
15}
```

$$TC = O(N)$$

$$SC = O(1)$$

believe in the process

Kapil Yadav