Bit Manipulation

Introduction

Until now, we were using only integers, floating-point numbers, etc for our codes, but now we are going to look at the bits present in these data types for our calculations and problem-solving.

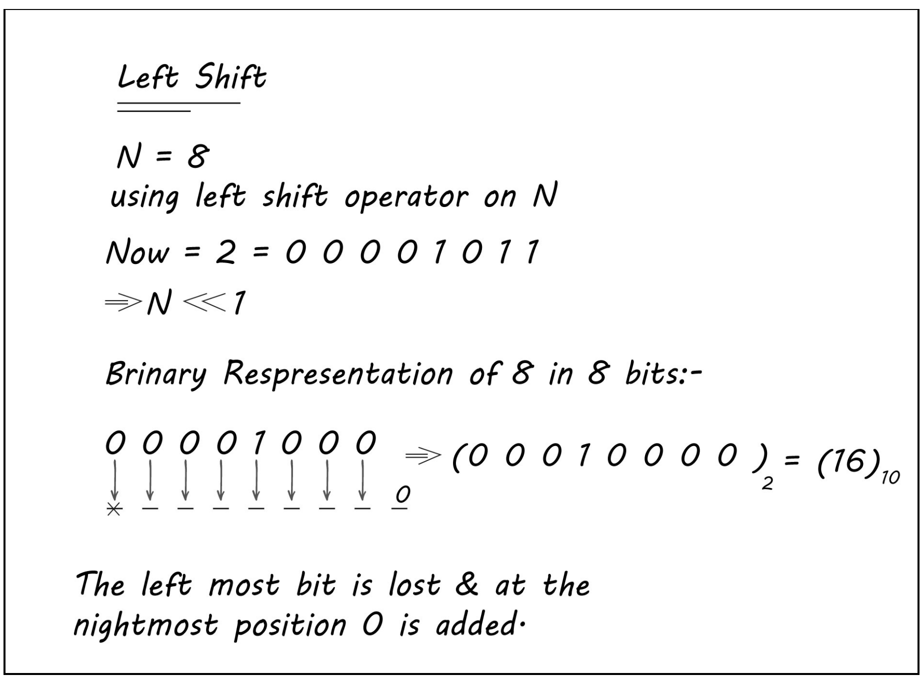
This is going to highly optimize our code, for example, an integer has 32 bits, now working on any of those 32 bits takes only O(1) time.

Shift Operators

Shift operators are of two types:

1. **Left Shift:** This operator is represented by **<<**. Example : **8 << 1 = 16**

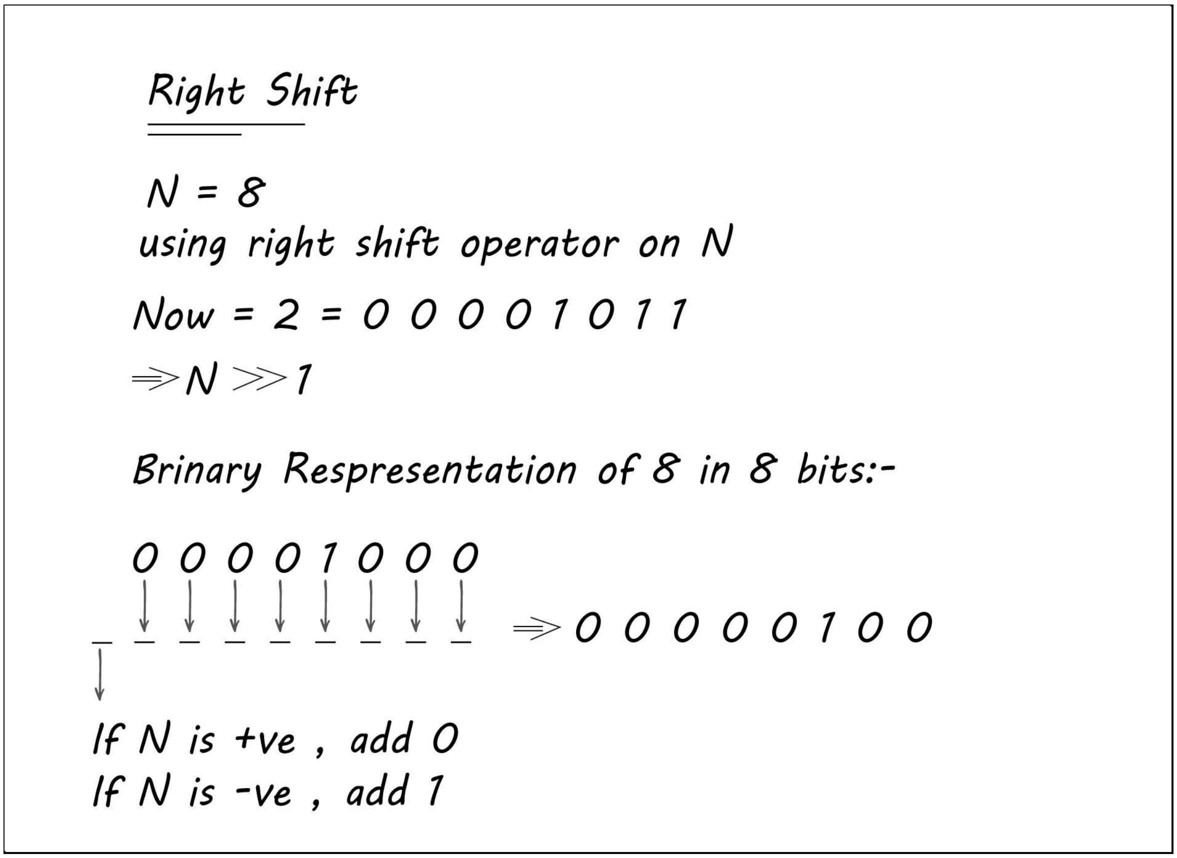
   This operator shifts every bit to its left position. For a clear understanding look at the image given below:



General Formula: N << i -> N*2^{i}

2. **Right Shift:** This operator is represented by **>>**. Example : **8 >> 1 = 4**

   This operator shifts every bit to its right position. For a clear understanding look at the image given below:



General Formula: N >> i -> floor(N/2^{i})

Some other bitwise operators

1. **& (bitwise AND)** takes 2 numbers as operands and performs logical AND on every bit of the two numbers. The result of AND is 1 only if both bits are 1.
2. **| (bitwise OR)** takes 2 numbers as operands and performs logical OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. **~ (bitwise NOT)** takes 1 number and flips all the bits of it.
4. **^ (bitwise XOR)** takes 2 numbers as operands and performs XOR on every bit of 2 numbers. The result of XOR is 1 if the two bits are different.

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	0	0

Check ith bit

ith bit is set if its value is 1 and it is not set if its value is 0. We will be given a number **N** and a position **i** and we will have to check if the bit at **ith** position is set or not. Now, let's take a look at an example to understand how we are going to proceed:

Given : **N = 5, i = 2**

The binary representation of 5 in 8 bits is **0 0 0 0 0 1 0 1**. We make a new number in 8 bit format that has only one bit set and that bit is at the **ith** position. Then we take the **&** of these two numbers. If the answer is zero, then the **ith** bit is set, else it is not set.

In the diagram, we have also shown an example where ith is not set. The example has **N = 1, i = 2**.

$N = (5)_{10} = (0\ 0\ 0\ 0\ 0\ 1\ 0\ 1)_2$

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

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

$N = (1)_{10} = (0\ 0\ 0\ 0\ 0\ 0\ 0\ 1)_2$

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

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

Flip i^{th} bit

We know that :

1 ^ 0 = 1

0 ^ 0 = 0

Therefore, **X ^ 0 = X** and

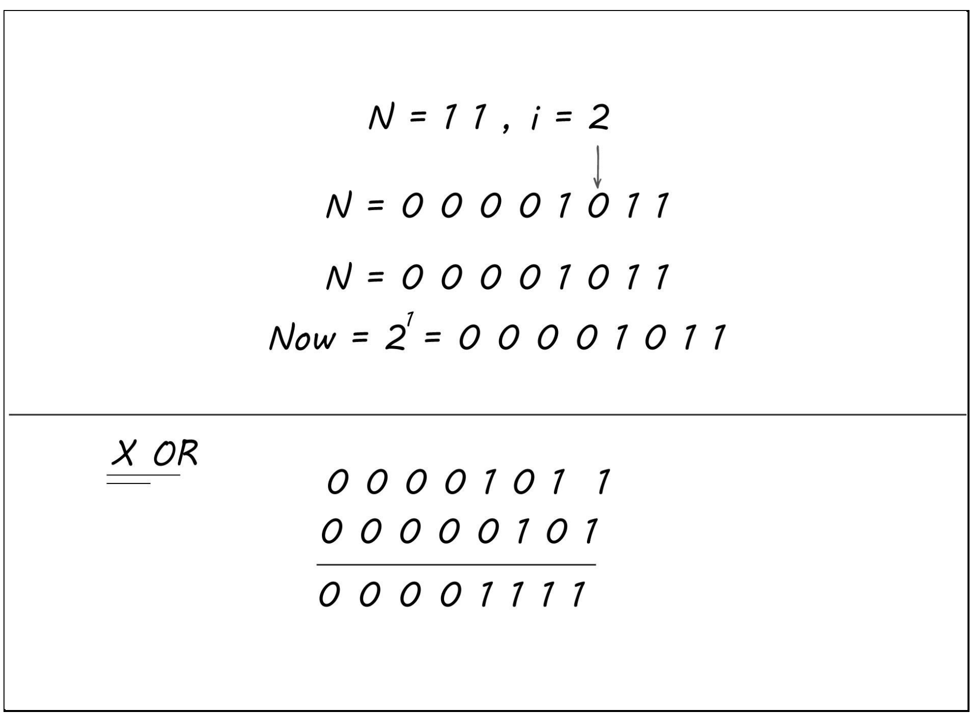
0 ^ 1 = 1

1 ^ 1 = 0

Therefore, **X ^ 1 = ~X**

This means that to flip a bit we can take its XOR with **2^i**.

Example:



2^i can be calculated using the left shift operator, i.e., 2^i = 1 << i.

Check N: odd or even

The basic way to do this is taking modulo by 2:

If (N % 2 == 0) {

Even

} else {

Odd

}

Optimized Approach: Sometimes, in competitive programming our solution might not work by 0.1 seconds or a very small time, so to optimize this we can use bitwise operators.

Let's take a look at the binary representation of a few numbers:

11 : 0 0 0 0 1 0 1 1

8 : 0 0 0 0 1 0 0 0

9 : 0 0 0 0 1 0 0 1

10 : 0 0 0 0 1 0 1 0

The last bit of every odd number is 1 and for every even number it is 0 because the last bit is represented by $\mathbf{2^0}$. So we can just check the last bit of a number to check whether a number is odd or even. Try to implement this by using the concept of checking i^{th} bit taught before.

Check N whether it is power of 2 or not

The basic way to do this is :

```
while ( ( N % 2 == 0) ) {
        N = N / 2;
}
If ( N == 1 ) {
        POWER OF 2
} else {
        NOT POWER OF 2
}
```

The above approach runs in **O($\log_2$(N))** time.

Optimized Approach :

Let's take a look at the binary representation of some numbers and see if we can infer anything from them :

2 : **0 0 0 0 0 0 0 1 0**

4 : **0 0 0 0 0 0 1 0 0**

8 : **0 0 0 0 0 1 0 0 0**

11 : **0 0 0 0 0 1 0 1 1**

16 : **0 0 0 0 1 0 0 0 0**

15 : **0 0 0 0 0 1 1 1 1**

There is one common pattern, that the numbers which are powers of 2 have only one bit set. Now take a look at the numbers 16 and 15 :

16 : **0 0 0 0 1 0 0 0 0**

15 : **0 0 0 0 0 1 1 1 1**

If we take **16 & 15**, then we get 0. This happens for all numbers **N and N-1** if N is a power of 2.

Generalized Formula: If N & (N-1) == 0, then N is a power of 2.

Remove all set bits from LSB to i :

LSB or Least Significant Bit is the rightmost bit.

Let's take a binary number N and **i = 3**,

N = 0 1 1 1 0 1 0 1 1 0

Now, take another number **A = 1 << (i+1)**,

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

B = A-1 = 0 0 0 0 0 0 1 1 1 1

C = ~B = 1 1 1 1 1 1 0 0 0 0

Final answer = N & C = 0 1 1 1 0 1 0 0 0

Generalized Formula: N & M where M = ~((1 << (i + 1)) - 1)