Todays Content:

- How to store -ve numbers
- Significance of MSB
  - Most Significant Bit
- Datatype Ranges

8 bit Number

LSB → SignBit ⇒ By making it 1, value is -ve

10 : 0 0 0 0 1 0 1 0

-10 : 1 0 0 0 1 0 1 0

4 : 0 0 0 0 0 1 0 0

-4 : 1 0 0 0 0 1 0 0

+

10 : 0 0 0 0 1 0 1 0

2^3 2^2 2^1 2^0

-14 : 1 0 0 0 1 1 1 0

Note: This way of storing -ve numbers, won't work because, all operations will go wrong

#Issue: 2

0 : 0 0 0 0 0 0 0 0

0 = -0 : 1 0 0 0 0 0 0 0

for same number, we have 2 different binary representations, which is invalid.

2's complement

a, 2's a = - a

- 1's a + 1 ⇒ 1's a replace all 0 ⇄ 1 in a = ~a
- ~a + 1 ⇒ This will give - a

8 bit Numbers:

10 = 0 0 0 0 1 0 1 0

~10 = 1 1 1 1 0 1 0 1

1 = 0 0 0 0 0 0 0 1

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

-10 = 1 1 1 1 0 1 1 0

↳ Convert to Decimal:

128 + 64 + 32 + 16 + 4 + 2 = 246

Correct Conversion MSB = -ve

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

-10 = 1 1 1 1 0 1 1 0 = -10

↳ Convert to Binary:

-128 + 64 + 32 + 16 + 4 + 2 = -10

118

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

Obs1: 7th bit base value is high

Obs2: $2^7 > 2^6 + 2^5 + .. 2^0$

128 = 127

Col: In general LMB is known as Most Significant Bit → [MSB]

Note: Base value of MSB is actually -ve

// Verify:

$-2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

25 : 0 0 0 1 1 0 0 1

↓

~25 : 1 1 1 0 0 1 1 0

+1 : 0 0 0 0 0 0 0 1

$-2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

-25 : 1 1 1 0 0 1 1 1

↳ Binary to Decimal:

$-128 + 64 + 32 + 4 + 2 + 1 = -25$

$-128 + 103$

// 4 bit Numbers : $-2^3\ 2^2\ 2^1\ 2^0$

LMB → MSB

$-2^3\ 2^2\ 2^1\ 2^0$		Decimal
1 0 1 0	→	$-2^3 + 2^1 \Rightarrow -6$
1 1 1 1	→	$-2^3 + 2^2 + 2^1 + 2^0 \Rightarrow -1$
0 0 1 1	→	$2^1 + 2^0 \Rightarrow 3$
0 1 0 1	→	$2^2 + 2^0 \Rightarrow 5$

Bit Number	MSB base value
4	-2^3
8	-2^7
16	-2^{15}
N	-2^{N-1}

Note: → Signed Numbers : Base value of MSB is -ve

↳ Unsigned Numbers : Base value of MSB is +ve

Signed

Signed 4 bit

-2^3	2^2	2^1	2^0	decimal
1	0	0	1	→ -7
1	1	0	0	→ -4

Unsigned

Unsigned 4 bit

2^3	2^2	2^1	2^0	decimal
1	0	0	1	→ 9
1	1	0	0	→ 12

Note: We cannot store -ve numbers in un-assigned

// How can system differentiate?

Signed:

```
byte  a → 8 bit signed number
short b → 16 bit signed number
int   c → 32 bit signed number
long  d → 64 bit signed number
```

↳ // By default it's signed

Unsigned:

```
unsigned byte  a → 8 bit unsigned number
unsigned short b → 16 bit unsigned number
unsigned int   c → 32 bit unsigned number
unsigned long  d → 64 bit unsigned number
```

↳ // need unsigned keyword

↳ Keyword: C/C++/C#

Signed Range

// 2 bit signed number

-2^1	2^0	→ Decimal	→ Min	Max
0	0	0	-2	1
0	1	1		
1	0	-2		
1	1	-1		

// 3 bit signed number

-2^2	2^1	2^0	→ Decimal	→ Min	Max
0	0	0	0	-4	3
0	0	1	1		
0	1	0	2		
0	1	1	3		
1	0	0	-4		
1	0	1	-3		
1	1	0	-2		
1	1	1	-1		

# N bit Signed	Min	Max
2	-2	1
3	-4	3
4 : $-2^3\ 2^2\ 2^1\ 2^0$	-8 1 0 0 0	7 0 1 1 1
5 : $-2^4\ 2^3\ 2^2\ 2^1\ 2^0$	-16 1 0 0 0 0	15 0 1 1 1 1
6 : $-2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$	-32 1 0 0 0 0 0	31 0 1 1 1 1 1

N bit number :

$-2^{N-1}\ 2^{N-2}\ \ldots\ 2^2\ 2^1\ 2^0$

Min: $-2^{N-1}\ 2^{N-2}\ \ldots\ 2^2\ 2^1\ 2^0$

1 0 ... 0 0 0

-2^{N-1} :

Max: $-2^{N-1}\ 2^{N-2}\ \ldots\ 2^2\ 2^1\ 2^0$

0 1 ... 1 1 1

// Obs :

N bit signed Number

Min	Max
-2^{N-1}	$2^{N-1}-1$

$2^0 + 2^1 + 2^2 + 2^3 + \ldots 2^{N-2}$

GP ⇒ Sum of Terms GP :

$a = 2^0$

$r = 2$

$t = N-1$

$$\frac{a*(r^t-1)}{r-1}$$

$$= 1*\frac{(2^{N-1}-1)}{2-1}$$

$$= 2^{N-1}-1$$

Datatype & Ranges :

	Size	#bits	Min -2^{N-1}	Max $2^{N-1}-1$
Byte/Char	1 Byte	8	$N: [-2^7, 2^7-1] \Rightarrow [-128, 127]$	
Short	2 Byte	16	$N: [-2^{15}, 2^{15}-1] \Rightarrow [-32768, 32767]$	
Int	4 Byte	32	$N: [-2^{31}, 2^{31}-1]$	INT_MIN, INT_MAX
			Int : $-2*10^9$, $2*10^9-1$	
long	8 Byte	64	$N: [-2^{63}, 2^{63}-1]$	LONG_MIN, LONG_MAX
			long : $-8*10^{18}$, $8*10^{18}$	

// Approximations :

$2^{10} = 1024 \approx 1000 = 10^3 \Rightarrow$ $\boxed{2^{10} \approx 10^3}$

$2^{10} = 10^3$ (Take cube on both sides)

$2^{10} * 2^{10} * 2^{10} = 10^3 * 10^3 * 10^3$

$2^{30} = 10^9$ ⟶ Multiply 2 both sides

$\rightarrow 2 * 2^{30} = 2 * 10^9$

$\boxed{2^{31} \approx 2 * 10^9}$

↓ Square on both sides

$2^{30} * 2^{30} = 10^9 * 10^9$

$2^{60} = 10^{18}$ ⟶ Multiply 8 both sides :

$8 * 2^{60} = 8 * 10^{18} \rightarrow \boxed{2^{63} \approx 8 * 10^{18}}$

1Q) Given N ar[] elements, calculate sum of array elements

```
long sumarr(int ar[]){
    int n = ar.length
    int sum = 0
    long
    for(int i=0; i<n; i++){
        sum = sum + ar[i]
    }
    return sum;
}
```

Constraints :

Based on constrains we can also estimate datatype of Input/Output variables

Constraints

$1 \le N \le 10^5$ → size of array

$1 \le ar[i] \le 10^6$ → Each ar[] element

Min: 1

TestCase:

N = 1

{1}

Max: 10^{11}

TestCase:

$N = 10^5$

$\{10^6, 10^6, 10^6, \ldots 10^6\}$

10^5 elements

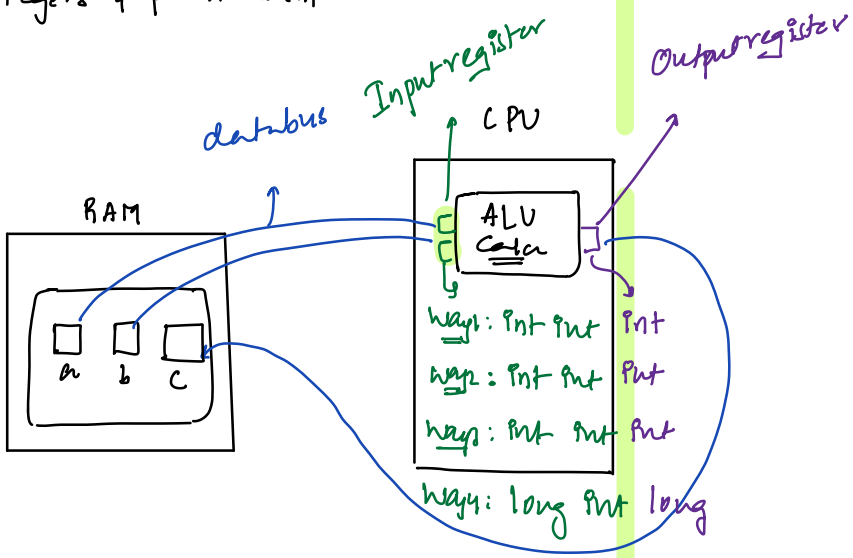
Obs: $[1\ 10^{11}]$ in a Int,

We need long datatype

// program to multiply 2 integers & print them

Constraints :

$1 \le (a, b) \le 10^9$




```
int main() {
    int a, b;
    Read a b

    way-1, * overflow
        int c = a * b
        print(c)
    way-2, * overflow
        long c = a * b   // 2nd fails: Because a*b, overflow already
        print(c)         happens, we are just storing wrong data
    way-3, * overflows   in c
        long c = long(a*b) ⇒ // BODMAS
        print(c)             ↳ // already overflow happened
    way-4
        long c = (long) a * b
        print(c)          ↳ // first we typecast
                            // long * int ⇒ long hence no overflow
    way-5
        long a1 = a, b1 = b
        long c = a1 * b1
        print(c)
}
```

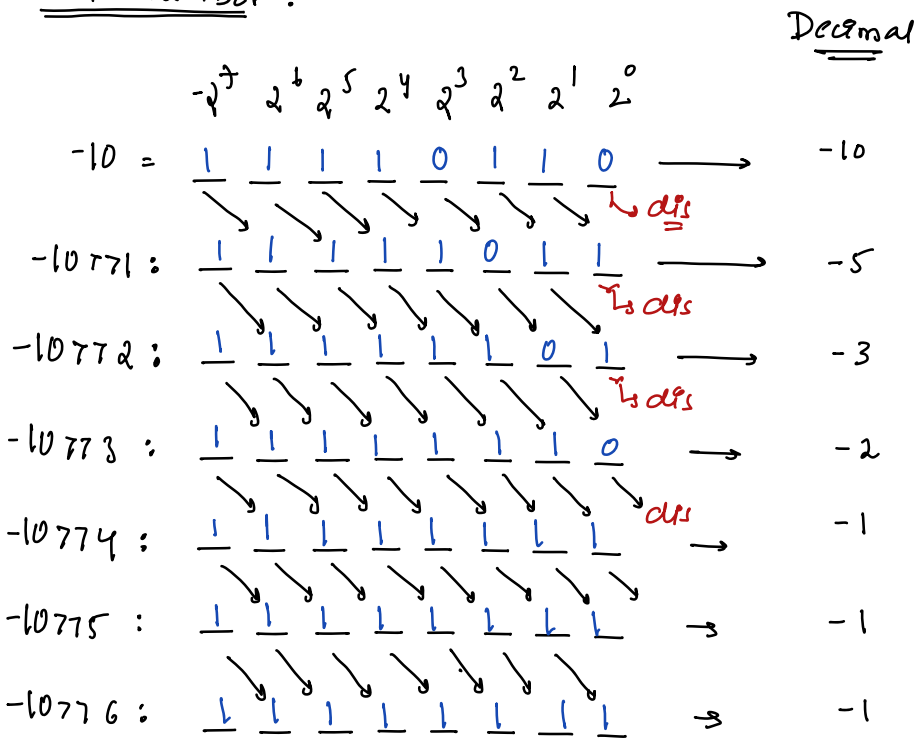
Note: When ever we multiply 2 integers, take care about overflow

// Doubts :

↳ >> on negative numbers

↳ When we apply >>, MSB bit is retained ∀ only for >>

8 bit Number :



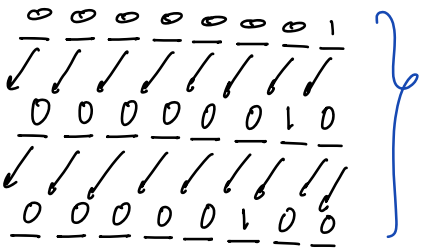
// for -ve numbers, value will saturate at -1, no matter how many times we perform >>, you will only get -1

int a = 10

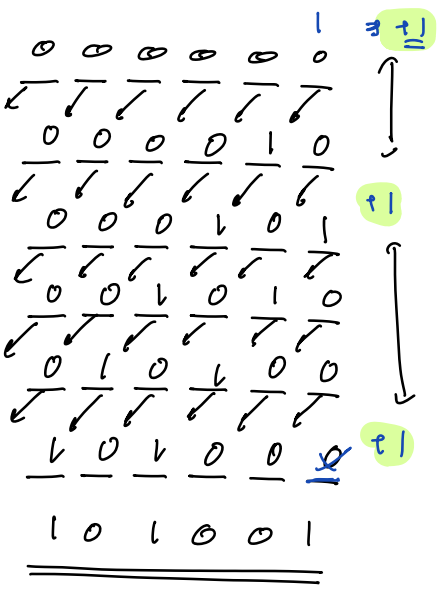
↓

MSB

//



*2 ⇒ left shift by 1 time



// vru → 4 bit signed datatype

vru n = 6 → 0 1 1 0 ⇒

vru n = -7 → 1 0 0 1 ⇒

$-2^3\ 2^2\ 2^1\ 2^0$

n = 7 : 0 1 1 1

$-2^3\ 2^2\ 2^1\ 2^0$