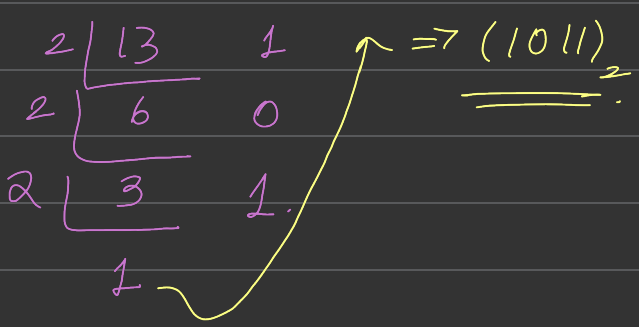
✱ Number format conversion

(i) Decimal (Base 10) → Binary (Base 2).

(i) Keep dividing decimal by 2 till 1.

Eg:- $(13)_{10} \longrightarrow (?)_2$.



(ii) Binary to decimal

(i) Start from right → left, multiply binary number with 2^x. $x = 0 \rightarrow n-1$ (right → left), add all.

$(1011)_2 \longrightarrow (13)_2 \Rightarrow$

1	0	1	1
$\times 2^3$	$\times 2^2$	$\times 2^1$	$\times 2^0$
8	0	2	1

$13 \leftarrow (8+2+1) \leftarrow$

* Code (Decimal to binary).

```
string decToBinary(int num){
    string decNum = "";
    while(num != 1){
        decNum.push_back(num % 2);
        num /= 2;
    }
    reverse(decNum.begin(), decNum.end());
    return decNum;
}
```

* TC :- $O(\log_2 N)$. {Similar logic as Binary search}.
* SC :- $O(\log_2 N)$.

☆ Code (Binary to decimal).

```
int binaryToDec (string binNum){

int len = binNum.size(), p2 = 1, num = 0;

for(i = len-1 → 0){

    if (binNum[i] == '1'){

        num += p2;
    }
    p2 = p2 * 2;   → { To simulate pow(x,2) }
}
return num;
}
```

☆ TC :- O (length (Num)).

☆ SC :- O(1) → { No extra space apart from ans storing }.

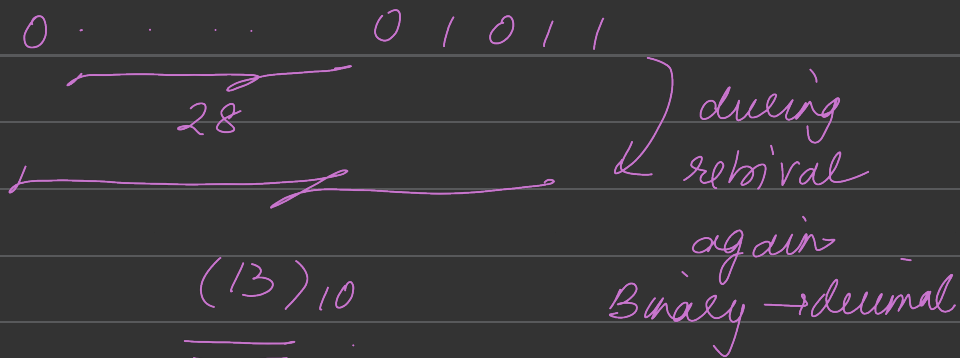
(Q) How computer stores int?

Eg int num = 13 → -13? 31st bit is used to store sign.

(i) convert decimal → binary :- (1011)

(ii) But computer has to store 32 bits

∴ first 28 bits 0, rest binary equivalent of decimal



* Long Long :- 64 bits

☆ Compliments:

(i) 1's compliment

(i) convert Decimal to Binary
(ii) flip all bits (0 → 1, 1 → 0)

Eg:- $(13)_{10} \xrightarrow{\text{Base 2}} (1011)_2$

↓ flip Bits

$(0100) \rightarrow 4$

(ii) 2's compliment

(i) find 1's compliment
(ii) Add 1 to 1's compliment (Binary addition).

$$\begin{array}{r} 0100 \\ +\ 0001 \\ \hline 0101 \end{array} \rightarrow \boxed{5}$$

✱ Operators.

(i) AND operator

All true => True

1 false => false.

x	y	x & y
0	0	0
0	1	0
1	0	0
1	1	1

(ii) OR operator

1 True => True

All false => false

x	y	x | y
0	0	0
0	1	1
1	0	1
1	1	1

(iii) XOR operator

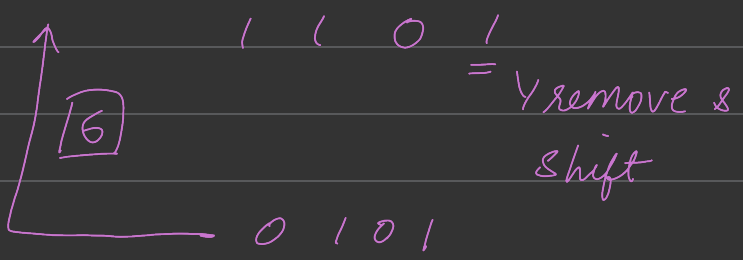
→ No. of 1's odd => 1 (True)
→ No. of 1's even => 0 (false).

x	y	x ^ y
0	0	0
0	1	1
1	0	1
1	1	0

(IV) SHIFT

(a) Right shift (>>)

Eg:- 13 >> 1.



Eg:- 13 >> 2.

1 1 0 1

↘ ↘ ✗ ✗

0 0 1 1 → 3

$\therefore x >> k = \frac{x}{2^k}$

↓ Proof.

13: (1 0 1 1) → $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

↓ right shifted ↑ one 2 reduced.

(0 1 0 1) → $1*2^2 + 0*2^1 + 1*2^0$

⇒ How negative number is stored?

→ 31st bit is used to store sign

(i) find 2's compliment

Eg:- -13 0 . . . . . 1 1 0 1.

↳ 1's comp:- 1 1 . . . . 0 0 1 0

+1 1

1 1 . . . . 0 0 1 1

↓ Negative Number

☆ Largest Number Integer can Store

0 1 1 1

↓

+ve Num

$= 2^{30} + 2^{29} + \ldots + 2^{0}$.

$= (2^{31} - 1)$ INT_MAX.

☆ Smallest Num. Int can store

-2^{31} INT_MIN.

$2^{31} \rightarrow$ 1 0 . . . . . 0.

↓ 1's comp.

Add 1 → 0 1 1 . . . . 1 + 1

1 . . . . 0 0

↓

{-ve num}

(II) Left shift (<<) $13 << 1$.

↳ Last bit goes off the cliff

26

$$a << k = a * 2^k.$$

(V) NOT operator (~)

(i) flip all bits

(ii) if -ve num store 2's compliment
else store normal

Eg:- ~5

Bin :- 0 - - - - 1 0 1

flip

1 1 . . . 0 1 0

sign bit

-ve?

Yes

2's comp

+1

0 0 . . . 1 0 1

1

1 0 0 . . . 1 1 0

-6

(Q) Swap 2 numbers (without 3rd variable)

(i) using XOR operator

(i) $a = a \wedge b$ — (i)

(ii) $b = a \wedge b \Rightarrow (a \wedge b) \wedge b = a$

$\therefore \boxed{b = a}$ — (ii)

(iii) $a = a \wedge b \rightarrow a = (a \wedge b) \wedge b$

(i)

$\Rightarrow a = (a \wedge b) \wedge a$ — from (ii)

$\therefore \boxed{a = b}$

Eg :- $a = 5, b = 3$

(i) $a = a \wedge b$ $\longrightarrow$

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

$\Rightarrow a = 6$

(ii) $b = a \wedge b$ $\Rightarrow$

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

$\therefore \boxed{b = 5}$.

(iii) $a = a \wedge b$ $\Rightarrow$

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

$\boxed{a = 3}$.

```
void SwapNumbers (int a, int b)
{
    cout << "Before Swap: " << a << b;

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

    cout << "After swap: " << a << b;
}
```

(Q) Divide two numbers without using {/, -, *}.

→ Input :- Dividend = 10, Divisor = 3

→ output :- 3.33 ~ 3 (Quotient).

* Brute force.

→ Add up the divisor until you reach divided

↳ Numbers of times you were able to add is ⇒ Quotient

for I/p (3+3+3 → 9).

```
sum = 0, cnt = 0;
while ( sum + divisor <= dividend){
        cnt ++;
        sum += divisor;
}
```

→ (worst case divisor = 1).

* TC :- O(Dividend)
* SC :- O(1)

★ Bit operator approach.

→ Intution here is that instead of taking divisor close to dividend, we bring dividend closer to divisor, by reducing some blocks from dividend.

Eg :- 10

10 − 6 → 4

4 − 3 → 1 (1 < 3)

↓

cannot divide

→ Here (6) and (3) are derived from powers of 2.

★ TC :- $O(\log N)^2$ →

```
while () {
    while () {
    }
}
```

★ SC :- $O(1)$.

⇒ Intuition behind powers of 2.

→ Eg $N = 22$, $d = 3$ ⇒ $q = 7$

⇒ we know we need to multiply $d \to q$ times to get closer to N.

⇒ we use power's of 2 is multiply approach

Eg $3 * 7$ ⇝ Express 7 in 2 power terms.

$3 * (2^2 + 2^1 + 2^0) \Rightarrow (3*2^2) + (3*2) + (3*2^0)$

∴ if we remove $(3*2^2)$ from $N = 22$

$22 - 12 = 10$.

$10 \to$ Remove $(3*2) \Rightarrow 4$.

$4 \to$ remove $(3*2^0) \Rightarrow 1$ $(1 < 3)$

Ans $= (2^2 + 2^1 + 2^0)$

```
int quotient (int dividend, int divisor) {
    sign = false;
// If any of dividend or divisor < 0 => sign = True.

    long N = abs((long) dividend);
    long D = abs((long) divisor);
    long Q = 0;
while (N >= D) {
        powerCnt = 0;

    while (N >= (D << (powerCnt + 1))) {
        powerCnt++;
    }

    Q += (1 << powerCnt);
    N -= (D << powerCnt);
}
// Check for Q > INT_MAX/MIN & sign case
return Q;
}
```

=> what is cd << power(count).

$$x << y = x * 2^{y}$$

Eg:- 3 << 2 =>

0011.

↓ left shift by 1

0110.

↓ by 2.

1100. → 12

(3 * 2^2)

(Q) Check if N^{th} Bit is set

→ PS!- Given a decimal, check if n^{th} bit of that num is set to 1.

→ Input :- 13 (1101) K=2

→ output :- True.

* Approach #1 (Bruteforce :-

(i) convert decimal → binary
(ii) Iterate through binary & check K^{th} bit

* TC :- O(N) + O(logN).
* SC :- O(log N).

{ Left shift :- Move 1
{ Right shift :- Move num.

☆ Approach # 2 (SHIFT Operator).

(i) Left shift 1 By K : (1<<K). —(A)

(ii) perform (Num & A).

=> Intution :-

↳ Left shift 1 By K will move 1 to K^{th} position
↳ of (bit[K] & 1) {after shift}.

bit[K] = 1
else 0.

☆ Code :-

return (N & (1<<K)) != 0.

☆ By right shift :- return (1 & (N>>K)) != 0.

☆ TC = SC = O(1).

(Q) Set i^{th} Bit to 1.

→ Input N = 9, K = 2

1 0 0 1
↳1
=> 1 1 0 0

→ output :- 13

* Optimal Approach (Bring 1 under neath K^{th} bit).

(A)

(i) find 1 << K

(ii) if ((N & (1<<K)) == 0)
return N + (1<<K).

(iii) else N

(B) → Return (N | (1<<K))

(Q) clear the k^{th} bit

→ Input N = 13, K = 2 ⟶ (1 1 0 1)

↓ ⇒ 0

→ Output :- 11 ← 1 0 1 1

↓

11

☆ Optimal Approach

(i) Left shift : 1 << K — (A)

(ii) find ~A

(iii) (N & (~A))

```
 1101
 0100
   ↓
 1101
 1011
------
 1001
------
```

(Q) Toggle the k^{th} bit

→ Input :- N = 13 , K = 2

→ output :- 9

(1 1 0 1)

↓ flip

9 ← (1 0 1 1)

☆ Optimal Approach

(i) find (1 << K) — (A)

(ii) return (N ^ A)

(Q) Flip the Last Set Bit

→ Input :- N = 16 (10000).

→ Output :- 15 (01111). ← flip to 0

✱ Optimal Approach

(i) If we take N-1

↳ The rightMost set bit is ⇒ 0

↳ All bits after it become 1.

Eg:- 40 → (101000).

39 → (100111)

∴ Return (N & N-1). → (N | N+1).

1 for setting last unset bit

(Q) Count Set Bits

→ Input :- 13 (1101)

→ Output :- 3

* Brute / optimal

(i) while converting decimal to binary
if Remainder 1
↳ Add to counter
else skip

(ii) At the end if N == 1 => cnt ++
(iii) return cnt.

⇒ Other Approach.

```
while (n > 1){
    cnt += (n & 1);
    n = n >> 1;
}
if (n == 1) cnt++;
return cnt;
}.
```

// (Q) Flips required to reach from start → goal.

→ Input :- start = 10, goal = 7

↓

→ output :- 3 (1010 → 0111).

Optimal Approach

(i) we can use XOR operator to know if flip is needed or not, as it will output 1, only when both are different.

```
while ( start >0 || goal >0 ){

    bits += ( ( start &1) ^ (goal &1) );
    start /= 2;
}   goal /=2;
```

TC :- O(log(max(S,G)));

SC :- O(1).

(9) Power Set

→ PS:- Generate all the subsets of a given set.

→ Input :- [1,2,3]

→ output :- [[], [1], [2], [3], [1,2], [2,3], [1,2,3], [1,3]]

✱ Bit Manipulation Approach

↳ If the size of given set is N

Total subsets = 2^N

→ use ADE logic, represent $0 \rightarrow 2^N$ in binary format

→ for each of $0 \rightarrow 2^N$, whoseever bits are set, they are part of power set.

Eg:- [1,2,3] → N=3 ⇒ Total Power sets = 2^3 = 8

(4 → 2 → 1).

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

If N > 8 we use 4 bit representation (8 4 2 1).

→ At 0 ⇒ 0 set-bits ∴ powerset = [[]].

→ At 1 ⇒ 1 set-bit at index = 2.

∴ set[2] = 3 ∴ powerset = [[], [3]].

⋮

→ At 7 ⇒ set-bits = {0, 1, 2} ⇒ [1, 2, 3] in powerset.

```
vector<vector<int>> totalSubsets (vector<int> arr){
    N = arr.size();
    int totalSubsets = (1<<N)-1;   (2^N - 1)
    vector<vector<int>> subsets;
    for(int mask = 0 → totalSubsets){
        vector<int> curSubset;

        for(i = 0 → N){
// find if      if(mask & (1<<i)){
ith bit is          curSubset.push_back(arr[i]);
set or not:     }
        }
        subsets.push_back(curSubset);

    }
    return subsets;
}
```

* TC :- $O(2^N * N)$.

* SC :- $O(2^N * N)$.

(Q) XOR from 0 → N.

→ Input :- N = 4

→ output :- 4.

* XOR prefix Trick.

→ After every 4 XOR cycle a pattern repeats.

N		XOR
0	0 → N	0
1	0 → 1	1
2	0 → 2	3
3	0 → 3	0

N		XOR
4	$0 \to 4$	4
5	$0 \to 5$	1
6	$0 \to 6$	7
7	$0 \to 7$	0

8	$0 \to 8$	8
9	$0 \to 9$	1
10	$0 \to 10$	11
11	$0 \to 11$	0.

∴ from above

(a) $N \% 4 == 0 \Rightarrow XOR = N$ (0,4,8)

(b) $N \% 4 == 1 \Rightarrow XOR = 1$ (1,5,9)

(c) $N \% 4 == 2 \Rightarrow XOR = N+1$ (2,6,10)

(d) $N \% 4 == 3 \Rightarrow XOR = 0$ (3,7,11)

∴ int XORTOZero (int N) {

if (N % 4 == 0) return N;

else if (N % 4 == 1) return 1;

else if (N % 4 == 2) return N+1;

return 0;

}

(Q) find XOR from L → R

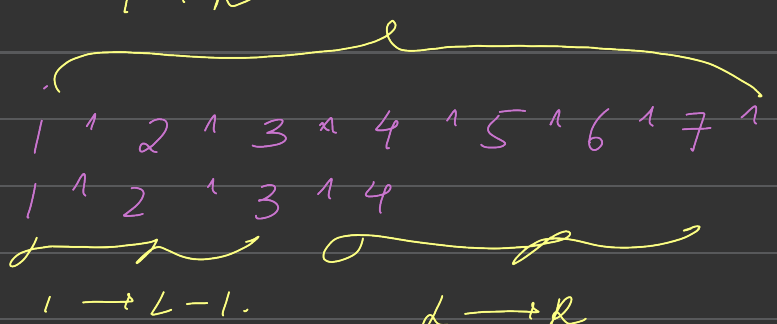
* TC :- O(1)
* SC :- O(1)

→ Input :- L = 4, R = 8

→ output :- 8 (4 ^ 5 ^ 6 ^ 7 ^ 8).

* Optimal Approach

(i) from above find XOR (1 → R)
(ii) find XOR (1 → L-1).
(iii) Remove (ii) from (i) to get L → R



∴ ANS = XOR (0 → R) ^ XOR (0 → L-1).

(Q) find Two odd occuring.

→ Ps :- Given an array where every element occurs twice, except two elements return that.

→ Input :- [4,2,4,5,2,3,3,1].

→ Output :- [5,1].

* Optimal Approach (Bucket concept).

→ Intution behind this Bucketing solution is that, in the two odd one's out, There will be at least 1 bit that differ between 2 odd elements
↳ (only then 2 exists else there would be only 1 element left).

→ Hence seperate them into 2 different Buckets, XOR each bucket element to get the answer.

→ Logic used to segregate between B1 & B2.

(i) XOR all Elements — (A).
(ii) find the right most set Bit of (A) — [B].
(iii) perform ((A)^(B)).

[(Num & (Num-1))] ^ [Num] . — [X].

Gives a binary number with right most set bit, rest all zeros.

(iv) if arr[i] has right most set bit at same index such as -[X] => B1
else ↳ B2

(V) XOR all Elements in B1 ⇒ O1
(VI) XOR all Elements in B2 ⇒ O2. } this

```
vector<int> findTwoOdds(vector<int>& arr) {
    int xor = 0;
    int N = arr.size();
    for(i = 0 → N) { XOR = XOR ^ arr[i]; }.

    int rightMostSetBit = ((XOR & (XOR-1))^XOR;

    for (i = 0 → N) {
        if ( rightMostSetBit & arr[i]) {
            B1 = B1 ^ arr[i];
        }
        else {
            B2 = B2 ^ arr[i];
    } }
    return {B1, B2};
};
```

* TC :- O(N) + O(N).
* SC :- O(1).

MATHS

(Q) print all divisors of a number

→ Input :- N = 36

→ Output :- [1, 2, 3, 4, 6, 9, 12, 18, 36].

```
for(i=1→N){
  if(N%i==0)
}
```

O(N)

* Optimal Approach

→ if we find one divisor that divides completely N, it will also help to find the other divisor

Eg :- if N = 36, one divisor is 18.

18 —How?→ (18 * 2 = 36).

∴ if we find 2, $\frac{36}{2}$ is other divisor

⇒ Edge Case

↳ There are some cases when the 2 divisors are same

Eg, N = 36 and divisor = 6.

6 * 6 => 36

4 duplicate ans.

```
for(i = 1 → √n){

  if(n%i == 0){
    res.add(i);
    if(n/i != i){
      res.add(n/i);
    }
  }
}
```

☆ TC :- O(√N)

☆ SC :- O(1).

⇒ Why $\sqrt{N}$, How does it work

→ If $36 \% 2 == 0 \Rightarrow 2$ is getting multiplied by some y to give 36

$\therefore 36 \% 2 == 0 \Rightarrow 36/2 = 18$

$\therefore 18 * 2 = 36$

→ If 2 is divisor of 36, 36/2 is also.

⇒ We need not iterate till $i = 36$, consider $\sqrt{N} = 6$

1	2	3	4	5	6
↓	↓	↓	↓	X	
$36 \% 1 = 0$	$36 \% 2 = 0$	$36 \% 3 = 0$	$36 \% 4 = 0$		$36 \% 6 = 0$
$36/1 = 36$	$36/2 = 18$	$36/3 = 12$	$36/4 = 9$		$36/6 = 6$

→ If we use above we need not go to N.

$3 * 12 = 36$

$4 * 9 = 36$.

⋮

$9 * 4 = 36$

$12 * 3 = 36$

Already covered for 4 we get 9 Now we dont need 4 for 9.

(Q) Print all Prime factors of a number

→ Input : N = 100

→ output : [2, 5]

Brute

$N * \sqrt{N}$

```
for(i=2 → N){
  if(N%i==0 &&
     isPrime(i)){
  }
}
```

Optimal approach

→ use the optimal approach used to print all divisors, As we need to find all divisors that are prime.

→ If its a divisor, check for prime, find other divisor, check for prime.

After i * i the factors repeat in reverse.

```
{ for (i = 2 ⟶ √N) {

    if (N % i == 0) {

        if (isPrime(i)) ans.add(i);

    }

    if ((N/i != i) && isPrime(N/i)) {

        ans.add(N/i);

    }

}

// if N is a Prime Number
if (isPrime(N)) { ans.add(N); }

return ans;

}
```

$\Rightarrow$ TC :- $O(N * \alpha * \sqrt{N})$

↓ Rough estimation because we dont know how many times isPrime() will be called for

⋆ Prime factorisation Method

→ Convert the prime factorisation method to code

Eg :-

```
2 | 780
2 | 390
3 | 195
5 |  65
13|  13
  |   1
```

2, 3, 5, 13 : prime factors

Dry run for NN

Eg :- 36 →

```
2 | 36
2 | 18
3 |  9
3 |  3
  |  1
```

i = 2 → $\sqrt{36} = 6$

36 % 2 == 0

36 / 2 = 18

18 % 2 == 0

18 / 2 = 9

9 % 3 == 0

9 / 3 = 6

6 % 3 == 0.

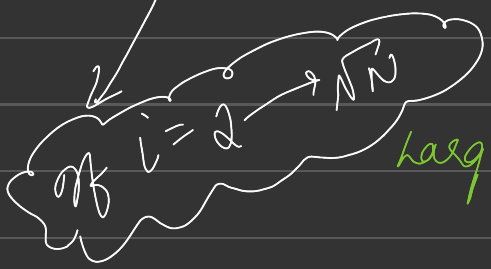
6 / 3 = 3.

3 % 3 == 0.

3 / 3 = 1

```
{ for (i = 2 → N) {
    if (N % i == 0) {
      ans.add(i);
      while (N % i == 0) {
        N /= i;
      }
    }
  }
  if (N > 1) ans.add(N); // prime number case.
}
```

* TC :- O(N). (worst case)
* SC :- O(1)



for a large N that is prime

* Optimisation for above

for (i = 2 → √N).

* TC :- O(√N)

(9) Count Primes

→ PS :- Given an integer N, return count of all prime numbers strictly lesser than N.

→ Input : N = 10 → output = 4 {2, 3, 5, 7}.

✱ Bruteforce

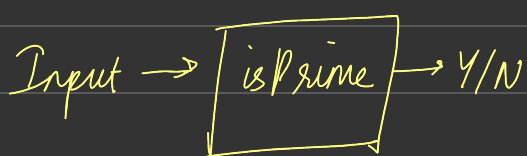
```
for (i = 2 → N) {
    if (isPrime(i)) { count++; }
}
```

✱ TC :- $O(N * \sqrt{N})$. → cannot be computed, depends on Num(primes)

✱ SC :- $O(1)$.

☆ Sieve of Eratosthenes

→ Here, the Goal is to prepare a black box, which can help to find if a number is prime or not in O(1).



O(1) BlackBox

⇒ Idea / Intuition

→ If a Number is prime, all of its multiples will never be prime

Eg :- 2 is prime

$2 * 2 = 4$
$2 * 3 = 6$
$\vdots$
$2 * 11 = 22$

} can never be prime

⇒ Sieve says, maintain an array prime[N+1].

→ Initialised with 1.

→ Iterate from 2 → N

↳ if we encounter any prime, mark all of its multiples as 0.

Blackbox

```
for (i = 2 → N) {
    if (prime[i] == 1) {
        for (J = 2*i; J <= N; J += i) {
            prime[J] = 0;
        }
    }
}
```

5*2 = 10
5*3 = 15
↓
5

i = 5

J ↳ 5*2 = 10

J = 10+5 ⇒ 5*3 = 15

☆ Optimisation

↳ Do we really need to check all of its multiples or that can be optimised?

Eg:- N = 30

prime = 2	prime = 3	prime = 5
Multiples:	Multiples:-	Multiples:-
2 * 2 = 4	3 * 2 = 6	5 * 2 = 10
2 * 3 = 6	3 * 3 = 9	5 * 3 = 15
2 * 4 = 8		5 * 4 = 20
2 * 5 = 10		5 * 5 = 25

∴ we can start check from Number itself

→ As we can see above, as we go further on the number line, the multiples have already been pre-computed

Eg:- 5 * 4

↳ would have already been pre-computed by 2

∴ Optimised Black Box

```
for (i = 2 → N) {
    if (prime[i] == 1) {
        for (J = i * i; J <= N; J += i) {
            prime[J] = 0;
        }
    }
}
```

(Q) Do we really need to go to N?

→ No, since J starts from i * i, it makes sense to iterate i from $2 \rightarrow \sqrt{N}$

★ TC :- $O(N \log(\log N)) + O(N)$

↳ to print 'N' primes

↓

prime Harmonic Series

(Q) Prime factorization of a numbers

→ PS:- perform the prime factorisation of a number and find all the factors involved (even if we get same).

→ Input :- 60

→ Output :- 2 2 3 5

2	60
2	30
3	15
5	5
	1

★ Just approach (Modified version of print prime factors).

ans.add(i) is before while in finding prime factors.

```
for (i = 2 → √N) {
    if (N % i == 0) {
        while (N % i == 0) {
            ans.add(i); N /= i;
        }
    }
}
```

★ TC :- O(√N) ★ SC :- O(1)

→ Above code is inefficient when the input is Queries ⇒ Get prime factorisation for multiple numbers.

↳ ✱ TC :- $O(Q * \sqrt{N})$.

✱ Optimised Approach via Sieve.

→ prepare a spf[N+1] array, spf = Smallest prime factor.
→ initialised as spf[i] = i

Q Why spf ?

→ In standard approach, we waste by checking if each Element is prime factor or not. Eg :- N = 25

↳ 1 → 25 } extra
2 → 25 }
5 → 25

→ spf will save us with these,

→ If prime[i] == i. [either i is smallest SPF or i has not been visited yet].

true →

Go to all multiples of i and mark SPf[J] = i iff SPf[J] == i.

why?

We only need to do these for prime numbers.

If (P[i] != i)

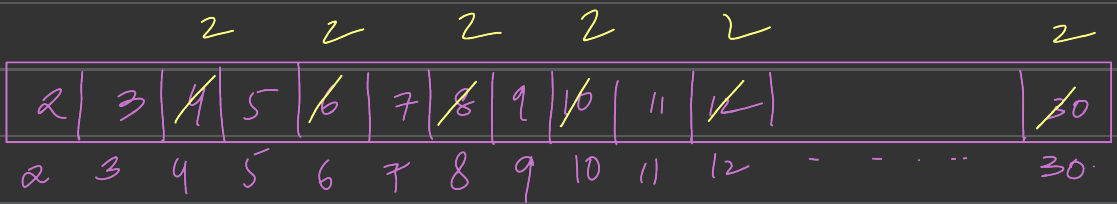
A multiple of i had marked i

It's not a prime

i is the smallest prime factor for these numbers.

→ If false ⇒ someone smaller than i has already marked themselves for SPf[J] and we are anyways looking for smallest numbers.

Eg:- N = 30



(i) 2 → 2 == 2 ∴ Mark multiples of 2 for SPF

(ii) 3 → 3 == 3 ⇒ Mark 3X for SPF

↳ SPF[6] != 6 it is Equals to = 2

∴ 2 had marked 2 as SPF[6].

⇒ This way we are preparing SPF for each Element, which we can use during prime factorisation.

Eg:-

2 | 30 → comes from SPF[30]
3 | 15 → SPF[15] = 3
5 | 5 → SPF[5].
1

```
int MAX = 100005;
int spf[MAX]; // initialised as spf[i]=i;

void sieve() {

    for(i=2 → √MAX){
        if(prime[i]==i){

            for(j=i*i → MAX){

                if(prime[j]==j){
                    spf[j]=i;
                }
            }
        }
    }

vector<int> primefactorisation(int N){

    while(N>1){
        ans.add(spf[N]);
        N = N/spf[N];
    }
}
```

* TC :- $O(N \log(\log N)) + \log N$
* SC :- $O(N)$

N /= spf[N];

→ To prepare SPf, $\log(\log N)$ because of prime Harmonic sequence

* Efficient when we have Queries

* TC :- $O(N * \log_2 N) + O(Q * \log_2 N)$;

* SC :- $O(N)$.

```
for (i = 0 → queries.size()) {
    N = queries[i];
    while (N > 1) {
        ans[i].add(spf[N]);
    } N = N / spf[N];
}
```

(Q) find Power of a Number. pow(x,N)

→ Input x = 2.10000, N = 3.

→ Output : 9.26100

* Power Approach.

→ Based on odd/even power

Eg:- 2^5

odd => split

$2 \cdot 2^4$ => $[x * x^{N-1}]$.

x(*). Add to ans

even

$(2^2)^{4/2}$ => $[x \cdot x]^{N/2}$

Repeat until N = 1.

```
double findPower(a,N){
ans = 1;
while (N>0){

if (N%2==1){
    N--;
    ans *= x;
} else {

    N/=2;
    a*=2;
}
}
return ans;
}

double pow(x,N){
    long exp = N;
    p = findPower(x, abs(n));
    if (N<0) return 1/p; else return P;
}
```

* TC :- O(log2 N).
* SC :- O(1)

✱ Binary Exponentiation

Eg:- $x^{13} \Rightarrow (x^8) * (x^4) * (x^1)$.

$(13)_{10} \longrightarrow (1101)_2$.

⇒ pattern :-

↳ find powers of x with LSB = 1

↳ then to get next LSB

N /= 2.

```
long P = N;
while (P > 0) {
    if ((P & 1) == 1) {
        res *= x;
    }
    x *= x;
    P >>= 1;  → (P = P/2)
}
```