* Maths

① Prime no. ∴ A no which is divisible by itself and by 1 is known as prime no's
Eg:- 2, 3, 5, 7, 13...

∴ Write a program to find whether 13 is prime or not.

Ans:
So all the no that come till 13 are
~~1~~, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ~~13~~

- We know that if we're trying for 13 we should ignore 1 & 13.
- If any of remaing no. dividers of multiple of 13 then it'll be a prime no.
- Obviously a no. greater than 13 will not divide 13, so ?3 x ? = 13
- ∴ 13 x 1 = 13 & ~~13~~ 1 x 13 = 13 - (we're ignoring this)
- Some other no x some other no = 13
- Here is it that any possible no that allows us to do that?
- So we're gonna check is 2 ~~divind~~ dividing 13 No, is 3 No ....

therefore

② Eg:- 36 Not prime / 2 = 36 ....
36 is divisible by No 1 x 36, 2 x 18, 3 x 12, 4 x 9, 6 x 6, 9 x 4, 12 x 3, 18 x 2, 36 x 1.
So these all no gives the outcome as 36.
As we can see that 36, 18, 12, 9 are getting repeated. So if you're checking that 3 x 12 = 36, so do you need to check whether 12 x 3 = 36. So these are all rearrangements. Hence ignore the repeated ones

- So instead of checking all the (n-2) nos.
- Only make checks for nos. less than the square root of the no itself
- So this has reduced the complexity to $(\sqrt{n})$

- Code :-

```
public class Prime {
    public static void main(String[] args){
        int n = 20;
        for(int i = 1; i<=n; i++){
            System.out.println(i + " " + isprime(i));
        }
    }
    static boolean isprime(int n){
        if(n<=1){                    // 1 is neither prime
            return false;            // nor composite.
        }

        int c = 2;                   - Start from dividing by 2
        while(c*c <= n){             - till the square
            if(n%c==0){
                return false;        - (not prime)
            }
            c++;
        }
        return true;                 - otherwise prime.
    }
}
```

It means that while $c <= \sqrt{n}$

$\therefore c = \sqrt{n}$

$\therefore c^2 = n$

$\therefore c \times c = n.$

* Sieve of Eratosthenes :- It is an ancient algo for finding all prime nos. upto any given limit.

- When you know that 2 is prime (if let's say you start from 2). that does that not mean all the multiples of 2 will be not be prime
- So if you know that 2 is prime. Do you need to check whether 2's multiples are primes? NO.
- 2 already ~~are~~ is dividing these nos. then no need to make any check. similarly, for ~~prime~~ 3.

Note:

- If you eliminate all the multiples of 2 so the multiple of 3 will also get eliminate (a little bit).
- Via 2 if you're eliminating 18. So when you're eliminating multiples of 3 over here you will see the 18 is also already eliminated. when we were eliminating the multiples of 2

* let's ~~visualize~~ Visualize.

prime. eliminate the multiple

(2)	(3)	~~4~~	(5)	~~6~~	(7)	~~8~~	~~9~~	10
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	30
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	38	~~39~~	~~40~~

So let's we've all these nos listed. In these nos. we've to find what are all the no. that are prime?

So it will be like Hey! you don't need to check this whenever you land on it. How do we maintain these? or How do you we maintain which no. is eliminated & which no is taken as a prime no?

Ans: We can creat an array & we can keep them as a boolean array which will contain true or false.

∴ ◯ → false (prime)

✕ → True (Not prime)

- So whenever you land on a index value that is ✕ skip it.
- so the array will move forward. & Now one'll check for 3, so it'll be like 3 neither ◯ nor ✕. So it will check whether 3 is prime or not. yes 3 is prime then eliminate it's multiple.
- Now we'll go to 4. But in the index value the 4 is already ✕ that means the factor of 4 is has been already appeared in the past which is true of because of 2 the skip it. so Now 5 yes it is prime then eliminate the multiples. Now 6. No, its not that means the factor of 6 had been appeard. i.e 2 & 3. so skip it.
- $\sqrt{40} = 6.32\ldots$ (around 6) so it is a same concept as previous of Eg 36. Eg:- $9 \times 4 = 36$ & it is something that has already been eliminated. so the further you will make is obviously gonna go till the square root. So check it till the square root //.

Because after that it get repeated. So do the check till square root only.

- Now all the no. that are remaining should be prime.

so we have:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37.

- Code :- intially the boolean array the contains all the elements "the value will be false". ↓ print(primes[0])

```
public class Seive {
    public static void main(String[] args){
        int n = 40;
        boolean [] primes = new boolean[n+1];   // boolean array
        sieve(n, primes);
    }
```

Here we're taking the no. as index so if 40 should be included. or if n should be included then size should be n+1. because array starts from 0. ∴ n+1

// false in array means no. is prime ↓

```
static void sieve (int n, boolean [] primes){
    // intiall we know that all the elements here in boolean will be false.
    for(int i = 2; i*i <= n; i++){
        if(!primes[i]){     // if primes is false or if (primes[i] == false)
            for(int j = i*2; j <= n; j+=i){
                primes[j] = true;
            }
        }
```

Here we can say that ↑

So whenever we find an element that is false we can say that this is a prime no. & then make all of the multiple of that no. true because its will be not prime.

Change all the multiples to true.

Ans:

```
for (int i = 2; i <= n; i++){
    if (!primes[i]){
        System.out.println(i + " ");
    }
}
}
```

Note:

- If the element contains the index elements are False (it means that it's prime).
- If the element is true. it means that it's already visited that means it's not prime. Because it's a multiple of some no. that we've seen in the past.

* j = j + i (in for loop). is for element.
  Eg: 2 ∴ 2 + 2 + 2 & it like multiples
  3 ∴ 3 + 3 + 3

- Space complexity :- O(n)
- Time complexity :- O(N * log(log N))

* Finding a square root of a number.

Let's say you've to given 36 find it sqrt.
as we know that the sqroot is going to be less than 36.
suppose we know that the given no. is perfect sqrt
Here we're starting to check from a sequence of a no. "like a sorted sequence of a no"
And whenever you're dealing with sorted sequence of a no or you're to find a no. in the range of sorted no. apply Binary Search.

So likewise we can start from 0 & take mid as 18

0(s) 18(m) 36(e)

① Now is m x m = 36 ? - NO, then where do you need to look or where'd the no. lie. is the no. will be more than 18 or less?
- As we know that 18 x 18 != 36. Hence reduce the ans. so it will lie bet^n 0 to 18 range. & the e = m-1 otherwise s = m+1. (for perfect sqrt no).

② Eg:- 40 - sqrt = 6.32 → ?
you can get this by applying above way

① How can we get the decimal value?
Ans: We can start by incrementing the decimal value it's place. so let's say ans that we've right now.
root = 6. - (6. given ↑) (Now start by check).
Eg:- 6.1 x 6.1 - -> (increment the 1st decimal value. is 6.1 x 6.1 < 40 yes. then increment again.

Now it will be equal to

$6.2 \times 6.2 < 40$ yes increment

$6.3 \times 6.3 < 40$ yes incr.

$6.4 \times 6.4 > 40$ No Hence reverse back

$\therefore$ 6.3 will be the ans.

* Suppose you want to do this for 2 digit places.

Ans:- So first you were adding 0.1.
We you can do the same thing as above for 0.01.
Now you can keep on add for 0.01 if you wanted till '2 decimal' places.
Same for 3. and so on..
Every time just increase the decimal pts.

*

Time complexity :- $O(\log n)$.

Code:- Binary Search SQRT

```java
public class Sqrt{
    public static void main(String[] args){
        int n = 40;
(precision value) int p = 3; till how many no. of dec. do we want
        System.out.printf("%.3f", sqrt(n, p));
    } It is a place holder. %.3 means it'll give the value till dec. 3 only.
    static double sqrt(int n, int p){
        int s = 0;
   end. int e = n; = no. it self.
        double root = 0.0;
        while(s <= e){ for int value.
            int m = s + (e - s) / 2;
            if (m * m == n) {
                return m;
            }
            if(m * m > n){
                e = m - 1;
            }
            else{
                s = m + 1;
            }
        }
        double in = 0.1; (increment)
        for (int i = 0; i < p; i++){
            while( root * root <= n){
                root += in;              } add.
            }
            root -= in;   } -> go back.
            in /= 10;     } after going back ÷ by 10
        }
        return root;
```

* Newton Raphson Method :-

When we were finding a root previously we were taking it, eg" Binary search or Hit & trial"
Here the formula says that

$$\text{next} = \frac{\left(x + \frac{N}{x}\right)}{2}$$

next → actual sqrt $= \sqrt{N}$
x → sqrt that you assumed.
} formula.

Q Why the formula works?

Ans

let's say for a given no. n we're saying

$$\sqrt{N} = \frac{\left(x + \frac{N}{x}\right)}{2}$$

We know that x is a closest sqrt of n that you can reach. (your own guess)
Now imagine your guess was correct then $x = \sqrt{N}$ (actual sqrt of n).

$$\therefore \sqrt{N} = \frac{\sqrt{N} + \frac{N}{\sqrt{N}}}{2} \quad \therefore \sqrt{N} = \frac{2\sqrt{N}}{2}$$

$$\therefore \sqrt{N} = \sqrt{N} \; //.$$

so if your guess was the answer. the eq" statisfies.
so we're trying to minimise the error as minimal as possible.
So if x is the sqrt that i'm assuming & root is the actual sqrt then what is the error?

Ans Error = | root - x |

Eg:- root = 36.4
guess x = 36.1
error = 0.3 //.

- Now we're going to keep changing the value of x till this error becomes minimal.
- Initially what we gonna do is :-

① Assign x to N itself. so we gonna keep calculate the root using "formula" but it'll see like it isn't working. like you have just put n here.

② You'll find your answer when error < 1 :-
But what if the error is not < 1 then we're going to update our root. so we're going to root value as whatever the entire value is.

③ Update the value of x. & x = root value.

* Time complexity = O((log N) f(N))
* (fft)

f(n) = Cost of calculating $\frac{f(x)}{f'(x)}$

with n - digit precision.

Code :-

```
public class NewtonSQRT {
    public static void main (String[] args) {
        System.out.println( sqrt(40));
    }
    static double sqrt (double n) {
```

as we need x so that x is assign to N :. double.

```
        double x = n;
        double root; // scoping.
```

if infinite loop → while (true) { it's only going to break when the error is < 1.

$\sqrt{N} = \frac{x + \frac{N}{x}}{2}$ → `root = 0.5 * ( x + (n/x));` $\frac{1}{2} = 0.5$ (only error is allowed of 0.5)

```
        if (Math.abs(root - x) < 0.5) {
```

absoult value — Precision value

```
            break; // in this case we have
        }              found our Ans.

        x = root; // updating our root
    }

    return root;
}
}
```

so it will be much more closer to the actual value. so as the barrior is reduced the error will be reduced as well. but more steps will be taken then

* Factors of a number.
A no. that divides another no. it's termed as factors of that no.
Eg:- 20 = 1, 2, 4, 5, 10, 20.

Time complexity :- O(n) space : O(1).

Code :-

```
public static Factors{
    public static void main (String[] args){
        factors1 ( 20);
    }
    static void factors1(int n){
        for (int i=1 ; i<=n ; i++){
            if ( n%.i == 0){
                system.out.println(i);
            }
        }
    }
}
```

↑ 1st way

Time complexity :- O(n)

— x —

There is a problem over here the no. are getting repeated. So if we look at " if(n %. i == 0)" Here it gonna check for the n no of time & & algo the times get repeated. Eg is 20 %.1 & = 20%1=0 .... So. Only check till the sqrt of n. because After words it's getting repeated.

// 2nd way. Optimised one.
Time complexity : O(√n). / O(Sqrt(n)).

```
public class Facters {
    public static void main(String[] args){

        facters(20);
    }

    static void facters( int n) {
                            Math.sqrt
        for(int i = 1; i <= (n); i++) {
 2 facter wb & n/i= if (n % i == 0) {
      Eg: 36.       if ( n / i == i) { Here it won't
      repeat 6x6 twice. ↑
                    System.out.print(i);

                }else {

                    System.out.print(i+" "+n/i+" ");
                }
            }
        }
}
```

- This is just for ignoring the duplicate sorts
- So, if we try to run will be the ans which aren't sorted. Here the "i" is already in sorted order & "n" are in descending order. Here we are getting the combination of i & n. So, with the smallest one, you'll getting the largest one. and so on. So the second factor that we're getting. we can store it in some other form & print it seperately

3rd way. Optimised & sorted one.

Both time & space complexity: O(sqrt(n))
Because we're actually storing it in the list now.

```
public class Factors{
    public static void main(String[] args){
        factors(20);
    }
    static void factors(int n){
        ArrayList<Integer> list = new ArrayList<>();
        for(int i = 1; i<=Math.sqrt(n); i++){
            if(n%i==0){
                if(n/i == i){
                    System.out.println(i + " ");
                }else{
                    System.out.print(i + " ");
                    list.add(n/i); all the last no & the second half is actually going to lie in the list.
                }                                   (in descending order)
            } so in the list we're storing the half of no.
        }
        for(int i = list.size()-1; i>=0; i--){   (last to 1st)
            System.out.print(list.get(i) + " ");
        }
    }
}
```

* Modulo Properties :- % (remainder symbol)

① These are like distributive properties

$(a + b) \% m = ((a \% m) + (b \% m)) \% m$

② $(a - b) \% m = ((a \% m) - (b \% m) + m) \% m$

③ $(a * b) \% m = ((a \% m) * (b \% m)) \% m$

④ $\left(\frac{a}{b}\right) \% m = ((a \% m) * (b^{-1} \% m)) \% m$

§ This is basically known as multiplicative modulo inverse §.

⑤ $(a \% m) \% m = a \% m$ - § Important §

⑥ $m \% m = 0$ $\forall x \in$ +ve integers.
§ for all x belonging to positive integers

⑦ Extra :
If p is a prime no. - § application §
which is not a divisor of b,
then $(a\, b^{p-1} \% p) = a \% p$
due to Fermet's little theorem.

* Multiplicative Modulo Inverse (MMI). in brief.

Eg: $b^{-1} \% m$

So, this basically means multiplicative modulo inverse.

Eg:

$(6 * y) \% 7 = 1$

So here y is basically (mmi) for 6.

$\therefore$ y = mmi for 6.

$\therefore$ y = 6

$\therefore$ $(6 * 6) \% 7 = 1$

$\therefore$ $36 \% 7 = 1$

- Hence proved.

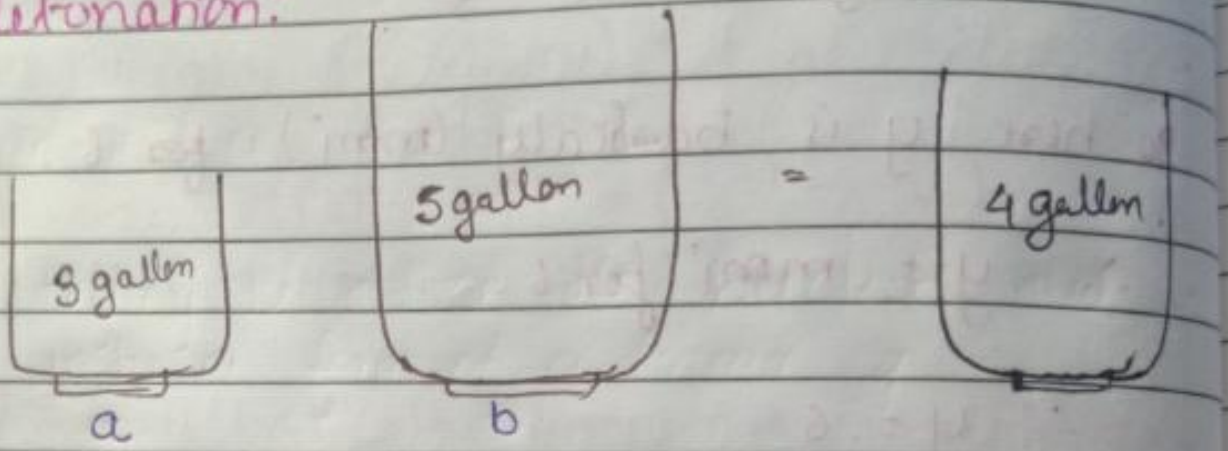
So (mmi) basically means $b^{-1} \% m$ this means that b & m are co-primes.

Note:-
Co-primes no. are the no. whose common factor is only 1 or nothing else is common except for 1 them.

Eg: 7 & 8 (What is the common factor in it?)
= 1

* Die-hard Example.

Q To disarmed a bomb "There're 2 jugs. 1 is of 5 gallon & other 1 is of 3 gall fill one of the jugs with exactly four gallons of water and place it on the sca And timer will stop. It must be precise One ounce one or more or less will resul in detonation."



Ans

① ~~This~~ First time :-

$(\underset{a}{0}, \underset{b}{0}) \longrightarrow (\underset{a}{3}, \underset{b}{0}) \longrightarrow (\underset{a}{0}, \underset{b}{3})$

② $(\underset{a}{0}, \underset{b}{3}) \longrightarrow (3, 3) \longrightarrow (1, 5)$

5 is the capacity & 3 is inside means 2 is remaining

$(1, 5) \longrightarrow (1, 0) \longrightarrow (0, 1)$

③ $(0, 1) \longrightarrow (3, 1) \longrightarrow (0, 4)$ //

suppose the jug was filled

jug a = s^1 times } (filled)
jug b = s^2 times } (emptied)

so what will be the remainder in the b jug at the end.

$$r = as^1 - bs^2 \quad -①$$

a: is the volume of jug
as^1 = it 9 is the amt of water taken
bs^2 = it is the amt of water removed.

If we take $-bs^2$ as it as in it integer.

$[r = as^1 + cs^2]$ in another way.

or

$$r = as^1 + (-bs^2).$$

But if we want to totally solve in positive & negative way.

$\therefore L = s^1a + t^1b$ - (suppose) - Ⓐ
$\therefore s^1a = L - t^1b$ - (given)

as we know that

$$r = s^1a + t^1b - t^1b - bs^2.$$

If we replace this with eqⁿ Ⓐ

$r = L - (t' + u)\, b$

if $t' + u \neq 0$

It basically says that

$[r < 0$ or $r > b]$ which is not true (the remainder can't in negative)

$t' + v = 0 \quad = \quad v = -t'$

$\therefore \; r - s'a + t'b = L.$

* We're trying to reduce the buckets that we've given into a form of a linear eq^n.

Q Why all this conversion?

Ans: Because we know by that if you want to make particular amt of water from 2 jugs it is going to be a combination of linear eq^n.
(i) it is a form of a linear eq^n.

2) & we know that we can write it infer in the form of $ax + by$ as well.

$\therefore \; r = s'a + t'b$ can also be written in $r = ax + by$ //

3) So basically to solve this water jug problem we need to have this particular eq^n.

Eg: (1) $3x + 5y = 4$ ..?

Q. How do we know whether this (1) is true of false?

Ans:-

Check what is the minimum value that this eqⁿ can make.

$\therefore 3x + 5y = 4.$

put x & y as integer. so what is the minimum positive value you can have of eqⁿ.

Eg:- x Fⁿ -3 & y = 2

$3x + 5y = 1.$ → minimum value (positive)

Note:-

This is know as GCD or HCF of 2 no.

HCF of a & b :- minimum positive value of eqⁿ $ax + by$, where x & y are integer.

So this is what actually HCF & GCD means.

Eg:- HCF of (4, 18)

4 = 1, (2), 4

18, 1, (2), 3, 6, 9, 18 } 2 is the highest factor common in both

∴ Ans - 2

HCF (3, 9)

3 = 1, (3)
9, = 1, (3), 9
Ans = 3

min (3x + 9y) = 3

3x + 9y = 3

∴ 3(x + 3y)

put x = -2 & y = 1

= 3(-2 + 3)
= 3

Def^n:-
If you're given 2 no "a & b" HCF will be the minimum value of the eq^n of ax + by where x & y can be any ind integer you want

Q. How does it relates to a problem?
Ans① So, If you're given 2 jugs of water a & b you know that ax + by you need to form a particular value of L liter.
∴ ax + by = L.
② let's say 1 is given of 2 lit & another one is of 4 lit & you have to form 5 lit of water.
∴ 2x + 4y = 5

so the minimum value of $2x + 4y$ will be

Note: N
Whatever HCF you will get that will come out as Common.

$\therefore$ HCF (2, 4)
Ans = 2

$\therefore 2(x + 2y) = 5$

$\therefore x + 2y = 2.5$ ✗

you can't use decimal point. Here if you were given a bucket of 2 Lit & 4 Lit you can't form a water of 5 Lit.

② $3x + 6y = 9$

HCF (3, 6)
Ans 3.

$3(x + 2y) = 9$
$\therefore 3x + 2y = 3$ ✓

So, yes you can for 9 Lit of water by 3 & 6 Lit jug.

(5) $3x + 5y = 17$

HCF = (3, 5) = 1

1 (3x + 5y) = 17

Note:
If the HCF is 1 you can form any amount of water.

$\therefore 3x + 5y = 4$

$1(3x + 5y) = 4.$

∴ Via 3 & 5 gallon jug you can form any amount of water

* Euclidean Algorithm

- It is used to find the GCD or HCF.

∴ gcd (a, b) = gcd(rem(b, a), a)

Eg:- gcd (105, 224) = gcd (rem (224, 105), 105)

= gcd (14, 105) — ①

Q why is it working?
Ans We are able to substract these things like the rem out of it & reduce the value

because in the end we're substracting it

∴ gcd(105, 224) → This basically means minimum value

$105x + 224y$

↓

minimum value } converted from eq ①

↓

$14x + 105y$

Q. Why do subtract? Why is it not changing the find bro?

Ans Because the linear combination, the GCD of (105, 224) also divides a linear combination of 105 & 224

let say $224 - 2 * 105$ - the linear combination can be anything the x & y value can be anything

Eg: $224 - 2 * 105 = 14$ (rem)

so 14 is the rem.

You're reducing the y with something it's gonna be linear equation.

* Use case :-

There is one more thing known as extended euclidean Algorithm. which actually gives us the value of x & y → (adv topic)

Q So when will this GCD stop?

Ans We know that anyone of the no from (a x sby) suppose a becomes 0 basically if you keep dividing by a or b at one point one of those will become Zero. 0
- So if a becomes 0 that means the ans will not be changed and otherwise put this as recursion call.

* Code :-

```
public class GCD_LCM {
    public static void main(String[] args){

        System.out.println(gcd(2, 4));
    }

    static int gcd gcd(int a, int b) {
        if( a == 0) {
            return b;
        }

        return gcd( b % a, a);
    }
}
```

Note:-

GCD stands for greatest common divisor

* LCM:-

It stands for Least common multiple.

Eg:- multiples of 2 = 2, 4, 6, 8, 12...

It means a number that is divisible by the num (both).

Eg:- LCM (a, b) = no. divisible by both a & b

① lcm (2, 4) = which is the smallest no that is divisibly by both 2 & 4.?
- 4

② lcm (3, 7) = 21

③ Note:-

let say we have a & b

suppose d = gcd (a, b) - this basically means that a gets divided by d & b also gets divided by d. because d is a factor

$\therefore f = \frac{a}{d}$, $g = \frac{b}{d}$.

$\therefore a = fd \quad , \quad b = gd$

** let say lcm = C

lcm (a, b) - { Has to be divisible of a & b }

$= lcm(fd, gd)$

** We know that f & g have no other common factor otherwise d would be bigger than that $\therefore$ d is the highest common factor

$\therefore$ f & g will have no other common factor.

Eg:- a = 9 & b = 18

$\therefore f = \frac{a}{d}$ { d = HCF }
$\therefore$ g = HCF

$\therefore f = \frac{9}{9}$

$\therefore f = 1 \quad - (1)$

$\therefore g = \frac{b}{d}$

$g = \frac{18}{9}$

$g = 2 \quad - (2)$

Here f & g have no other factor in common. Why?
Because the highest thing the was already common from it has already been taken down (out) in form of HCF
Hence it will have no other thing in common factor.

Hypothetically; it was possible then d would be bigger.
Eg:-
let say hcf = 3

$$f = \frac{9}{3} \qquad g = \frac{18}{3}$$

$$f = 3 \quad , \quad g = 6$$

Now to it does have something in common now
So 3 is common

Hence it is wrong. ∴ HCF should be bigger now. because one ~~not~~ more thing you're getting out of it common. add that into HCF aswell.

∴ HCF of 9 & 3 is not 3 it's actually 9, so another common stuff if 3 is common so take 3 & multiply with that hcf
∴ hcf = 3 × 3 = 9
It's now correct.

So now we know that HCF of f & g have no common factor.

Q What does that mean?

Ans So it basically means that our LCM (c), that is going to be multiple

Q. What is that thing that divides both your a & b? think.

Ans So we know a = fd & g = bd b = gd.

Q. What is the smallest no that can divide a & b?

As We know a = fd & b = gd & we also know that f & g have nothing common factors.

- So if we say f * g * d = LCM. Because this is the only way our above conditions are satisfied.

So basically we're trying to find, that if a no is dividing both a & b

* If you need to find a number that both a & b divides basically

LCM means a no. that divides is divided by both a & b.

So, here we need a no. that is divided by both a & b

∴ that no should contain a * b

Basically we need a no. that is divisible by a & b also

should not that no. contain or the factor of that no. contain a & b itself

Eg :- 9 & 18

or

17 & 19.

Q Find a no. that ~~divides~~ divisible by both 17 & 19?

Ans Here the no should contain 17 & 19 ~~as well~~ in it.

∴ $a \times b$

$= fd \times gd$ — {d is repeating twice Hence remove.}

∴ $LCM = f \times g \times d$

& we know that

$a \times b = fd \times gd$

if we take g common

∴ $= d * dfg$ {HCF = d} given.

$= hcf * LCM$ {dfg = LCM}

∴ $HCF \times LCM = a \times b$

formula :-

$$LCM = \frac{a \times b}{HCF(a,b)}$$

$$LCM(a,b) = \frac{a \times b}{HCF(a,b)}$$

Code :-

```
public class GCD_LCM {
    public static void main(String[] args) {

        System.out.println(lcm(17, 18));
    }
    static int gcd(int a, int b) {
        if (a == 0) {
            return b;
        }

        return gcd(b % a, a);
    }

    static int lcm(int a, int b) {

        return a * b / gcd(a, b);

    }
}
```