Basic Maths

In interview the coding questions are purely based on maths & by these concepts, we can find the optimal solution.

(i) Prime number

There can be multiple approaches to find whether a given number is prime or not.

Naive approach ⇒ Check for divisibility of the number from 1 to number itself & if it has exactly 2 factors, then we say the number is a prime.

* 1 is not a prime as it does not have 2 factors.
* 2 is the smallest prime number as it has 1 & 2 as factor.

There are other approaches such as the square root approach, Sieve of Eratosthenes, segmented

Sieve.

Q→ Count the number of primes strictly less than n.

i/p → 5
o/p → 2 → (2,3)

Naive approach   TC = O(n^2)

Run a loop from i = 2 to n & check whether i is prime or not. If we find that i is a prime number, then increment the count value & return count after the end of for loop.

Code

```
bool isPrime (int n) {
    if (n==0 || n==1)
        return false;
    for (int i = 2 ; i<n ; i++) {
        if (n% i ==0)
            return false;
    }
    return true;
}

int countPrime (int n) {
    int count =0;
    for (int i=2 ; i<n ; i++) {
        if (isPrime (i))
            count++;
    }
```

```
return count;
}
```

When we try to submit this on leetcode, it will give TLE as the above solution have time complexity as $O(n^2)$ as in the function named isPrime there is one for loop & this function is called in the another for loop & due to this, we obtain the nested for loops.

Better approach

We have to improve the isPrime function but how can we do it? The loop inside isPrime function is running from $i=2$ to $i<n$, so can we reduce it?

Let's assume n is non-prime

$$\underbrace{1, 2, 3 \ldots, n-1}, n$$

↳ atleast have 1 factor as we have assumed n is not prime.

This means we can say they (the number) has 2 factors a & b

$$n = a \times b$$

$$\left.\begin{matrix} a > \sqrt{n} \\ b > \sqrt{n} \end{matrix}\right\} \Rightarrow ab > n$$

But this is not possible & hence we can conclude that atleast one of the factor must be less than $\sqrt{n}$. This is not possible as $ab = n$.

If we can't find any factor less than $\sqrt{n}$, & this means n is prime. So we can reduce the for loop inside isPrime to $\sqrt{n}$.

But if we modify the for loop inside isPrime, still we won't be able to submit as still the time complexity is too high.

Modification TC = $O(n\sqrt{n})$

```
bool isPrime (int n) {

    if (n <= 1)
        return false;
    for (int i = 0; i <= sqrt(n); i++){
        if (n % i == 0)
            return false;
    }
    return true;
}
```

→ Atleast one factor should be less than $\sqrt{n}$ to be non-prime.

Better Approach - 2

Here we will be using the concept of Sieve of Eratosthenes theorm. This is used when we want to find the total count of primes less than n.

Suppose that n = 21. So make an array consisting of numbers 2 to 20 & initially we assume that all of them are prime & hence we mark all of them true.

2 3 4 5 6 7 8 9 10 11 12 13
14 15 16 17 18 19 20

* Now 2 is a prime number & when we start reading the table of 2, all of them will be non-prime. So mark 4, 6, 8, 10, 12, 14, 16, 18, 20 as false i.e non-prime.

* Now 3 is a prime number & hence mark 6, 9, 12, 15, 18 as false i.e non-prime.

* Now we go to 4 & we get to know it is already non-prime so move ahead.

* Now we go to 5 & then mark 10, 15, 20 as false i.e non-prime

So by the above dry run we need to mark the multiples of all prime as non-prime.

2 → T
3 → T
4 → F
5 → T
6 → F
7 → T
8 → F
9 → F
10 → F
11 → T
12 → F
13 → T
14 → F
15 → F
16 → F
17 → T
18 → F
19 → T
20 → F

Now just count all the T & return that number.

Code

```
int countPrimes (int n) {

    if (n == 0)
        return 0;
    vector <bool> prime (n, true);
    prime [0] = prime [1] = false;
    int count = 0;

    for (int i = 2; i < n; i++) {

        if (prime [i]) {
            count ++;
            int j = 2 * i;
            while (j < n) {
                prime [j] = false;
                j = j + i;
            }
        }
    }
    return count;
}
```

Mark the multiples of prime as non-prime

Segmented Sieve

It is simply the variation of Sieve of Eratosthenes in which we have been given a range l to h & in this range we have to count the no. of primes.

Time complexity of Sieve of Eratosthenes
$= O(n \log(\log n))$

Note → The array which we have made is known as Sieve.

$$TC \rightarrow n \times \underbrace{\left[\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \frac{n}{11} \cdots\right]}$$

Harmonic Progression of prime numbers (Tailor series

{ Take n out }

GCD / HCF
GCD is the Greatest Common Divisor. HCF is highest common factor.

a, b
a % hcf = 0
b % hcf = 0
↳ maximum

24 , 72

$24 = 1 \times 2 \times 2 \times 2 \times 3$
$72 = 1 \times 2 \times 2 \times 2 \times 3 \times 3$
$hcf = 2 \times 2 \times 2 \times 3 = 24$

Euclid's Algorithm
gcd (a, b) = gcd (a−b, b)   a > b
gcd (a, b) = gcd (b−a, a)   a < b

Also gcd (a, b) = gcd (a % b, b)   a > b
But we will be using the − operation as % is the heavy operation.

Ex → gcd (72, 24)

gcd (48, 24)
gcd (24, 24)
gcd (0, 24)
4 Ans

We have to apply the above formulae until any one of the parameter becomes 0.

Code

```
int gcd (int a, int b) {
    // Base Case
    if (a == 0)
            return b;
    if (b == 0)
            return a;
    while (a > 0 && b > 0) {
        // Run loop until any of parameter becomes 0
            if (a > b)
                a = a - b;
            else
                b = b - a;
    }
    if (a == 0)
        return b;
    else
        return a;
}
```

LCM

If we have found the hcf, then we can simply find lcm also.

$lcm(a, b) * hcf(a, b) = a * b$

However we can prove the Euclid's algorithm via mathematical induction but we don't have to deep dive.

Modulo Arithemetic

$a \% n$, then the answer will lie b/w 0 to n-1 where 0 & n-1 are inclusive

$10 \% 3 \Rightarrow ans = [0, 1, 2]$

$\hookrightarrow 1$

Properties of modulo

These properties will be used to handle the cases of overflow as it might happen sometimes that it can't get stored in a particular datatype, so we basically apply modulo operation.

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

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

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

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

We need to remember the above 4 properties to solve the coding problems in a better way.

Fast Exponentiation

Suppose that we want to find 2^{10}, then multiply 2, 10 times with each other.

$$2^{10} \Rightarrow \underbrace{2 \times 2 \times \cdots \times 2}_{10 \text{ times}}$$

Normal solⁿ to find a ^ b will be O(b)

The method is that

2^10
```
ans = 1
for (int i = 1 ; i < = 10 ; i++) {
        ans = ans * 2 ;
}
cout << ans ;
```

This is also known as slow exponentiation.

Code of slow exponentiation

a^b
```
int power (int a, int b) {

    int ans = 1 ;
    for (int i = 1 ; i< = b ; i++) {
        ans = ans * a ;
    }
    return ans ;
}
```

Time complexity = O(b)

We use the fast exponentiation method to avoid any TLE errors. This method has complexity as O(log b)

a ^ b

(i) b = even

$$a \wedge b = \left(a^{\frac{b}{2}}\right)^2$$

$2^{10} \Rightarrow (2^5)^2$

(ii) $b \Rightarrow$ odd $\qquad a^b = (a^{b/2})^2 \cdot a$

$2^{11} \Rightarrow (2^5)^2 \cdot 2$

Ex $\rightarrow$ 2^5

$(2^4) \cdot 2$

$(2^2 \cdot 2^2) \cdot 2$

$(2^1 \cdot 2^1) \cdot (2^1 \cdot 2^1) \cdot 2$

We see how we are dividing. This is basically known as divide & conquer approach.

Code of Fast exponentiation

```
int power (int a, int b) {
        int ans = 1;
        while (b>0) {
            if (b & 1) {
                ans = ans * a;
            }
            a = a * a;
            b = b>>1;
        }
        return ans;
}
```

Dry run

1) ans = 1, a = 5, b = 4

Is b odd, the answer is no. So we skip the if condition.

$a = 5 \times 5 = 25$

$b = 4 >> 1 = 2$

2) $a = 25$, $b = 2$, $ans = 1$

b is even so we skip the if block.

$a = 25 \times 25 = 625$
$b = 2 >> 1 = 1$

3) $a = 625$, $b = 1$, $ans = 1$

b is odd & now we go in the if block

$ans = 1 \times 625 = 625$
$b = 1 >> 1 = 0$
Hence exit the while loop & return 625.

Ex → $a = 2$, $b = 5$, $ans = 1$

1) $b \Rightarrow$ odd
$ans = 1 \times 2 = 2$
$b = 5 >> 1 = 2$
$a = a \times a = 2 \times 2 = 4$

2) $b = 2$, $a = 4$, $ans = 2$

$b \Rightarrow$ even

Now don't go in the if block.
$a = 4 \times 4 = 16$
$b = 2 >> 1 = 1$

3) $b = 1$, $a = 16$, $ans = 2$

$b \Rightarrow$ odd

$ans = 2 \times 16 = 32$

$a = 16 \times 16 = 256$

$b = 1 >> 1 = 0$

Hence exit the while loop & return ans which is 32.

Q→ Modular exponentiation for large numbers.

$(x^n) \% m$

i/p → $x = 3, n = 2, m = 4$

o/p → 1 as $(3^2) \% 4 = 9 \% 4 = 1$

Code

It is important to note that it is always safe to do mod if we might get a very large number.

```
long long int powMod(long long int x, long
long int n, long long int m){

    long long int ans = 1;

    While (n > 0) {
        if (n & 1) {
            ans = (ans * x) % m; // May
                                    be a bigger no. out
        }                           of range
        x = (x * x) % m;
        n = n >> 1;
    }
    return(ans % m);
}
```

Advanced topics for competitive programming
scope will be

1) Pigeonhole
2) Catalan number (BST)
3) Inclusion Exclusion Principle
4) Chinese Reminder Theorm
5) Lucas Theorm
6) Fermat's Theorm
7) Probability concepts.