Prime Number-

basic counting numbers ⟨0, 1, 2, ...⟩

Whole number greater than 1 whose only factors are 1 and the number itself...

A number that divides another no. evenly i.e., with no remainder

2
3
5
7
11
13
17
19
⋮

Applications-

- Cryptography ⟨H.W.⟩
- Random number generator
- Developing machine tools

Primality testing-

Question-

Given a positive integer, check if the number is prime or not

n = 11
→ true

n = 15
→ false

n = 1
→ false

Solution-

- number → n
- 2...(n-1) and check if it divides n (%)

Time complexity = O(n)
Space Complexity = O(1)

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

Optimizations-

1. We are checking till n. But do we need to? $\sqrt{n}$ → because a large factor of n must be a multiple of smaller factor that is already checked.

n = 48

2 ... 47

→ 2 ... $\sqrt{48}$

→ 2 ... 6

→ (8)? but 8 divides 48.

BUT we already checked for 6 right.

So, we only have to go till $\sqrt{n}$

(2) → All primes are of the form

$6k \pm 1$

except 2 and 3

All integers can be expressed in the form

$6k + i$ < But why $6k+i$ >

Why not? $7k + i$

i = -1, 0, 1, 2, 3, 4

$2x$ $3x$

$(6k-1)$ $(6k+0)$ $(6k+1)$ $(6k+2)$ $(6k+3)$

$(6k+4)$

$\text{mod } 2 \equiv 0$ $\text{mod } 3 \equiv 0$

$6k-1$, $6k+1$ could be prime numbers?

```
boolean isPrime(int n) {
    if(n<=3) {
        return n > 1;
    }
}
```

```
if(n % 2 == 0 || n % 3 == 0) {
    return false;
}
i = 5;
while(i*i <= n) {
    if (n % i == 0 or n % (i+2)==0){
        return false;
    }
    i += 6;
}
return true;
}
```

→ 3 times more optimized than just checking till $\sqrt{n}$...

Count Primes (Leetcode) #204 →

Question-

Given an integer n, return the number of primes that strictly less than n.

n = 10
→ 4 | 2, 3, 5, 7

n = 0
→ 0

n = 1
→ 0

Intuition-

→ 0 to n ...
isPrime function $\sqrt{n}$

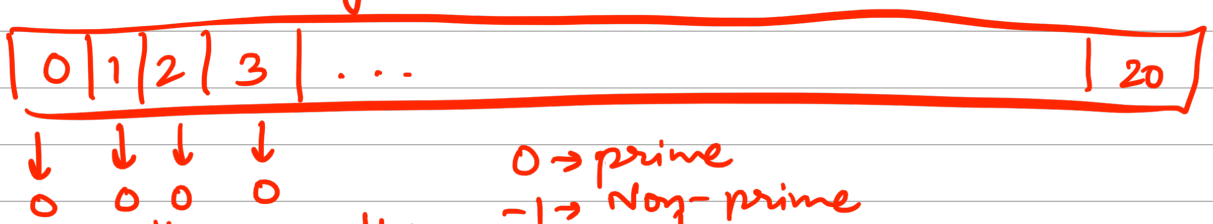
T = $O(n.\sqrt{n})$
S = O(1)

— Instead of checking if each number is prime or not, what if we mark the multiples of a prime number as non-prime?

Approach: Sieve Of Eratosthenes —

n = 21

→ Create array of 21 integers



→ We will use this array to mark primes and non-primes

→ Let's start with (2) <Smallest prime number>

→ We can mark the multiples of this number as non-primes in the array

→ set -1 for them

Almost,
Half of the elements are gone. Since they are marked non-prime

→ (3)

→ 4? marked as non prime... So, one of its factor already did the word. No need.

→ (5)? $5 \times 5 > 20$

$2 \times 2 \ldots +2$

$3 \times 3 \ldots +3$

fundamental theorem of Arithmetic-

→ Unique factorization theorem
→ Unique prime factorization theorem.

Every number greater than 1 is either a prime itself or can be represented as a product of prime numbers...

n = 50
7x7 n > 7x7

7x2 = 14 = (2) x 7
7x3 = 21 = (3) x 7
7x4 = 28 = (4) x 7
7x5 = 35 = (5) x 7
7x6 = 42 = (6) x 7

```
int countPrimes(int n) {
        if (n<=2) {
            return 0;
        }

        boolean[] numbers = new boolean[n];

        for(int p=2; p<=(int) Math.sqrt(n);
                p+=1) {

true → not prime        if(numbers[p]==false) {
false → prime
```

```
for ( int j = p * p; j < n;
      j += p ) {
    numbers[j] = true;
  }
}

int numberOfPrimes = 0;

for (int i = 2; i < n; i += 1) {
  if ( numbers[i] == false) {
    numberOfPrimes += 1;
  }
}

return numberOfPrimes;
}
```

$T = O(\sqrt{n} \cdot \log\log n)$

for 2, we cross out $n/2$ numbers

for 3, — n — $n/3$ numbers

for 5, — n — $n/5$ numbers...

$$O\Big(\underbrace{n/2 + n/3 + n/5 + n/7 + n/11 \cdots + \frac{n}{\text{last Prime} < n}}_{\sqrt{n}}\Big)$$

$S = O(n)$

$$(\sqrt{n})\, O\Big(\underbrace{1/2 + 1/3 + 1/5 + 1/7 + 1/11 + \cdots}_{\sim \log(\log(n))}\Big)$$

What would cause an algorithm to have a $O(\log(\log n))$ complexity?

(H.W.)

Logarithms –

- An important concept in complexity analysis
- A base is always associated with an algorithm
- In computer science, the base is almost always 2

$\log_2(x) = y$

$2^y = x$

Binary logarithm

- We dont even write base 2.

$\log(x) = y$

$2^y = x$

$\log_2(1) = ?$

$2^? = 1$

$? = y = 0$

$\log(4) = ?$

$2^{?} = 4$

$? = y = 2$

$\log 16 = ?$

$2^{?} = 16$

$y = ? = 4$

logarithm of base 2 of a number; is 2 power what?

$2^{x+1} \Rightarrow 2^{x} \cdot 2$

$2^{?} = N$

$2^{?+1} = N \times 2$

if N doubles, the ? increases by 1

$2^{20} = 1M$

$2^{30} = 1B$

What we are trying to say when N increases by a lot, $\log(N)$ or (?) increases by a tiny quantity

$O(\log N)$ is so much better than $O(n)$

- Binary logarithm
- logarithm is so important in time complexity.

arr = [0 1 2 3 | 4 5 6 7 8]
len = 8

[0 1 | 2 3]

[0 | 1]

→ We are left with 0 and we do something with it

→ At every step, we are halving our input

Now, let's say we have doubled the input size (16)

It only adds one more step...

So, log(N) time complexity is so cool...

→ Binary Trees, Arrays, ...

→ halving the input ... reduces our steps by 1 ?
→ doubling the input ... increases our steps by 1 ?

T ≈ O(log N).