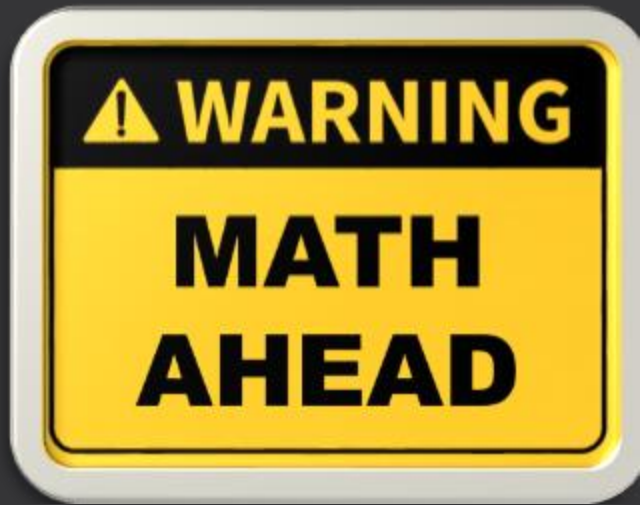
IT 20.01

Week 10

2022

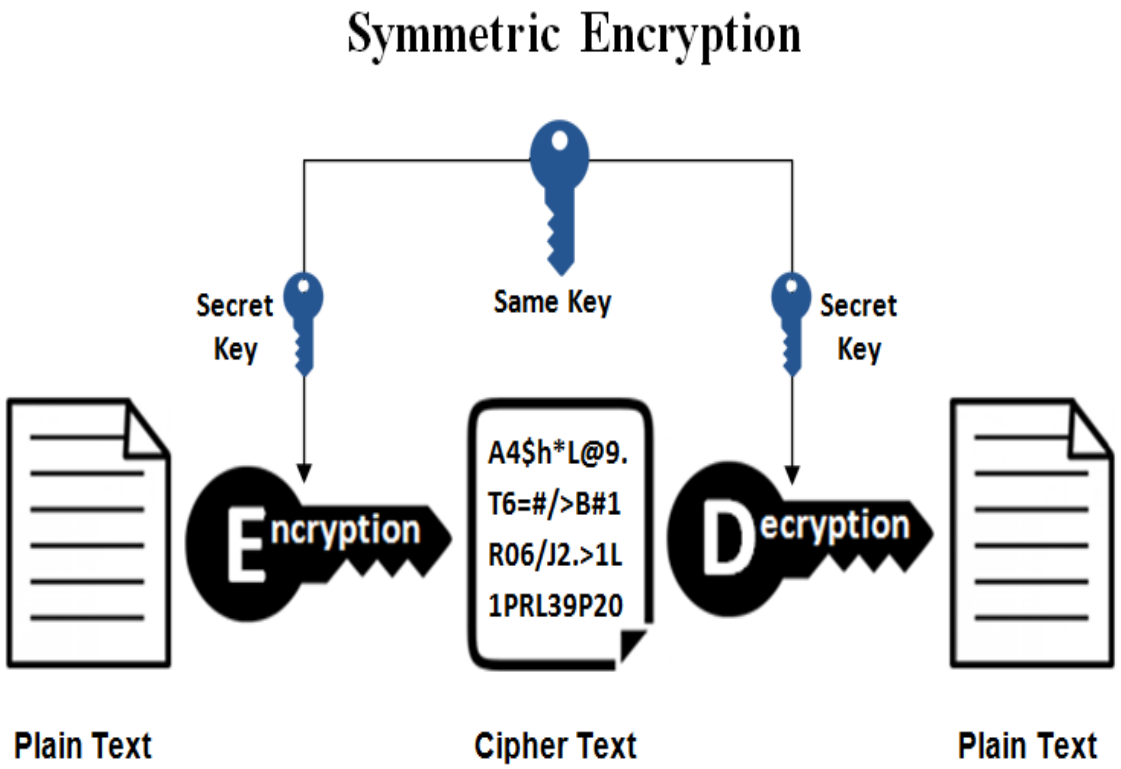
Finding & Generating Prime Numbers. Generating Keys for the Public Key Cipher

- Aziz Ahmadov
- Roman Tolstosheyev
- Mahammad Aghakishiyev
- Azer Sadykhzadeh
- Ravan Gajievi
- Huseyn Aghazada



Symmetric Encryption

- **A single, shared key**
  - Encrypt with the key
  - Decrypt with the same key
  - If it gets out, you'll need another key
- **Secret key algorithm**
  - A shared secret
- **Doesn't scale very well**
  - Difficult to distribute
- **Very fast to use**
  - Often combined with asymmetric encryption



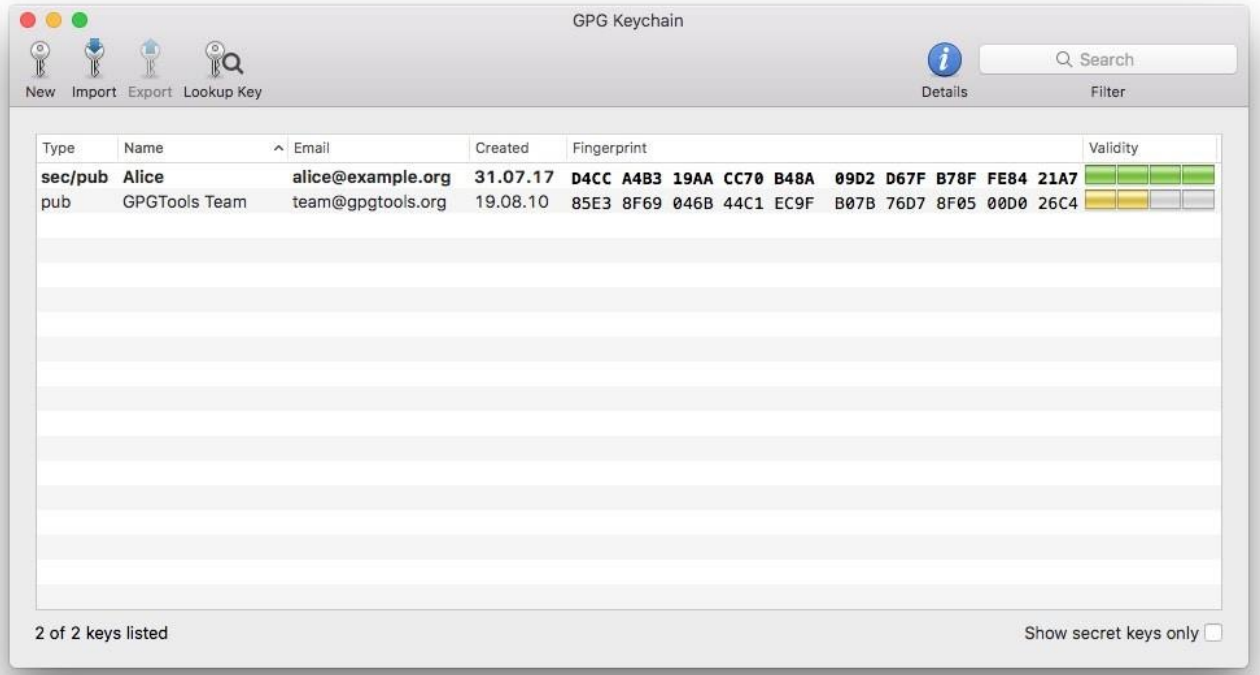
Asymmetric Encryption

- **Public key cryptography**
  - Two (or more) mathematically related keys

Private key

- Keep this private

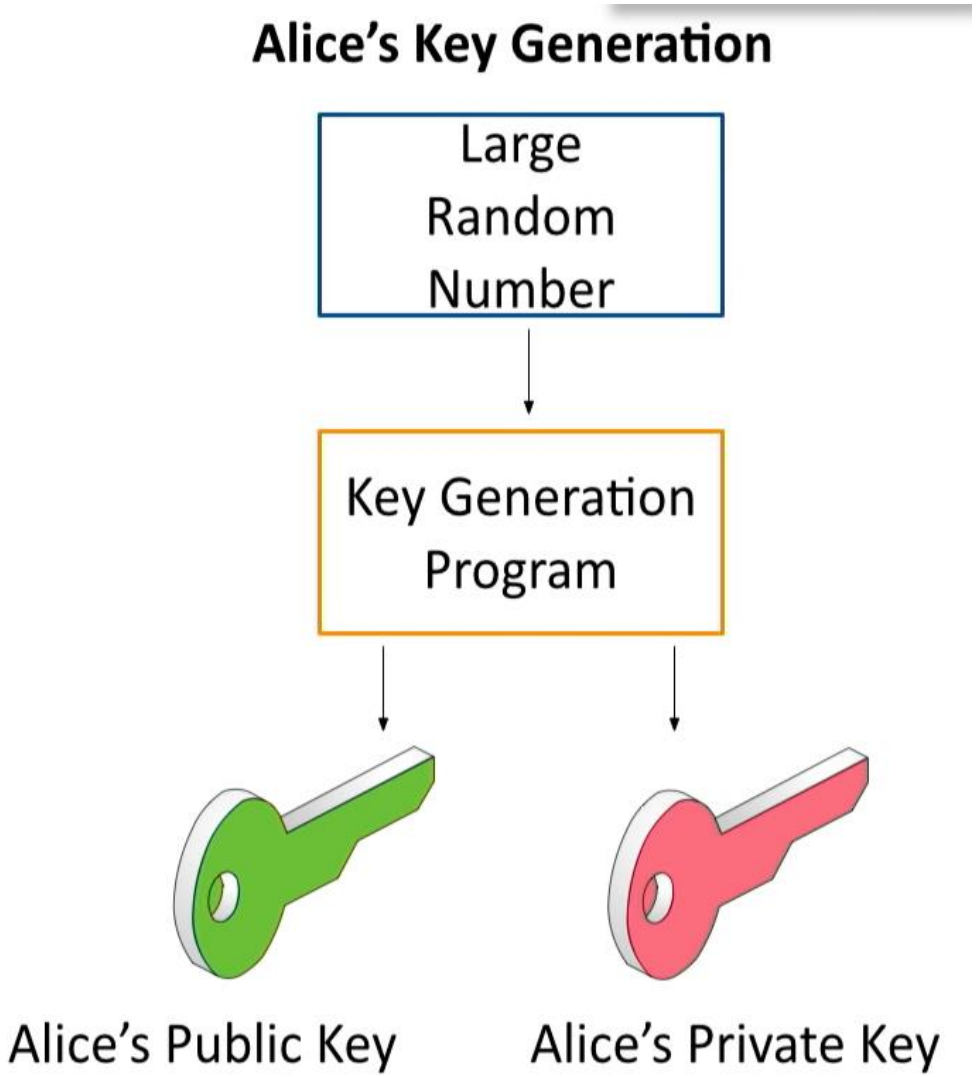
- **Public key**
  - Anyone can see this key
  - Give it away



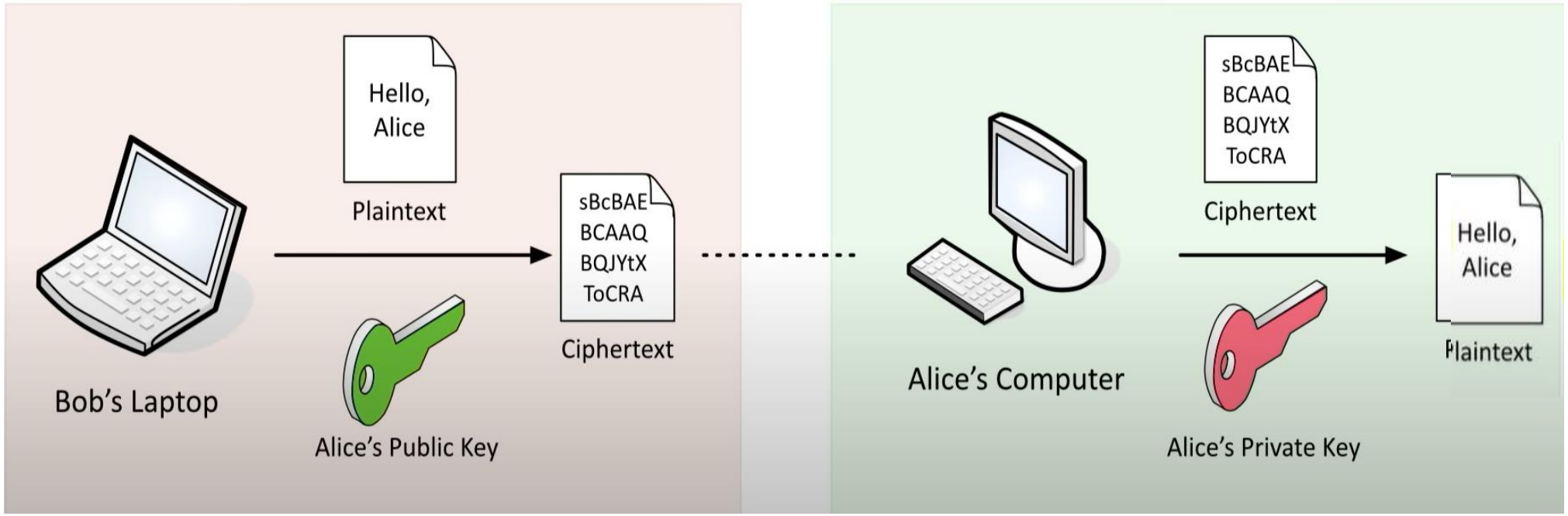
GPG keychain

The key pair

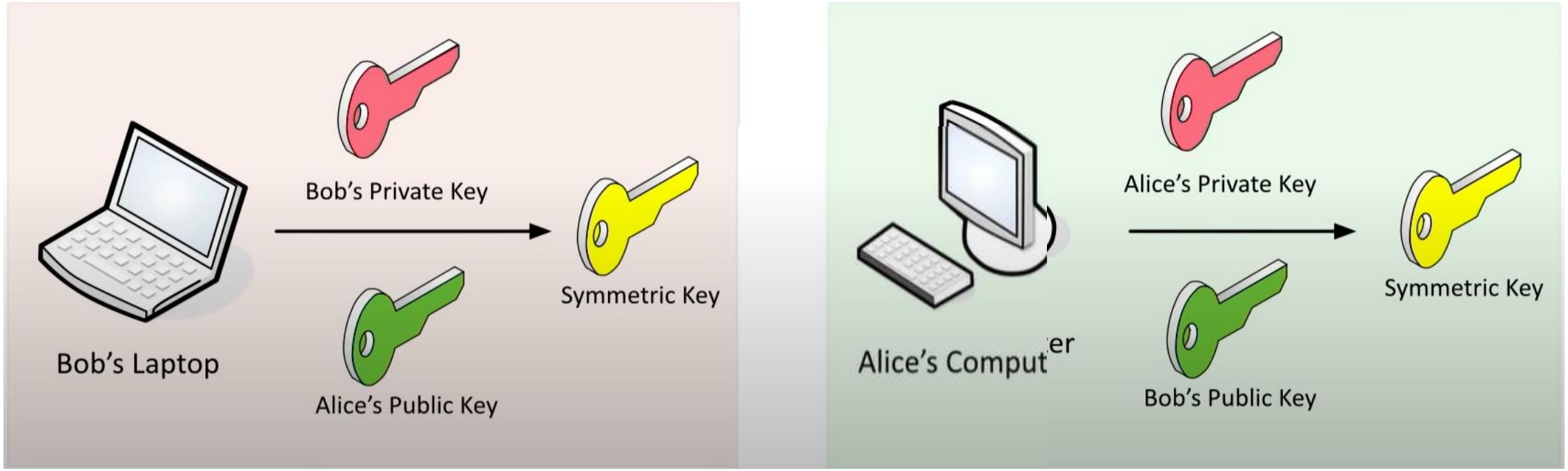
- **Asymmetric encryption**
  - Public key cryptography
- **Key generation**
  - Generate both public and private key at the same time
  - Randomization
  - Prime numbers
  - Mathematical functions



Asymmetric Encryption

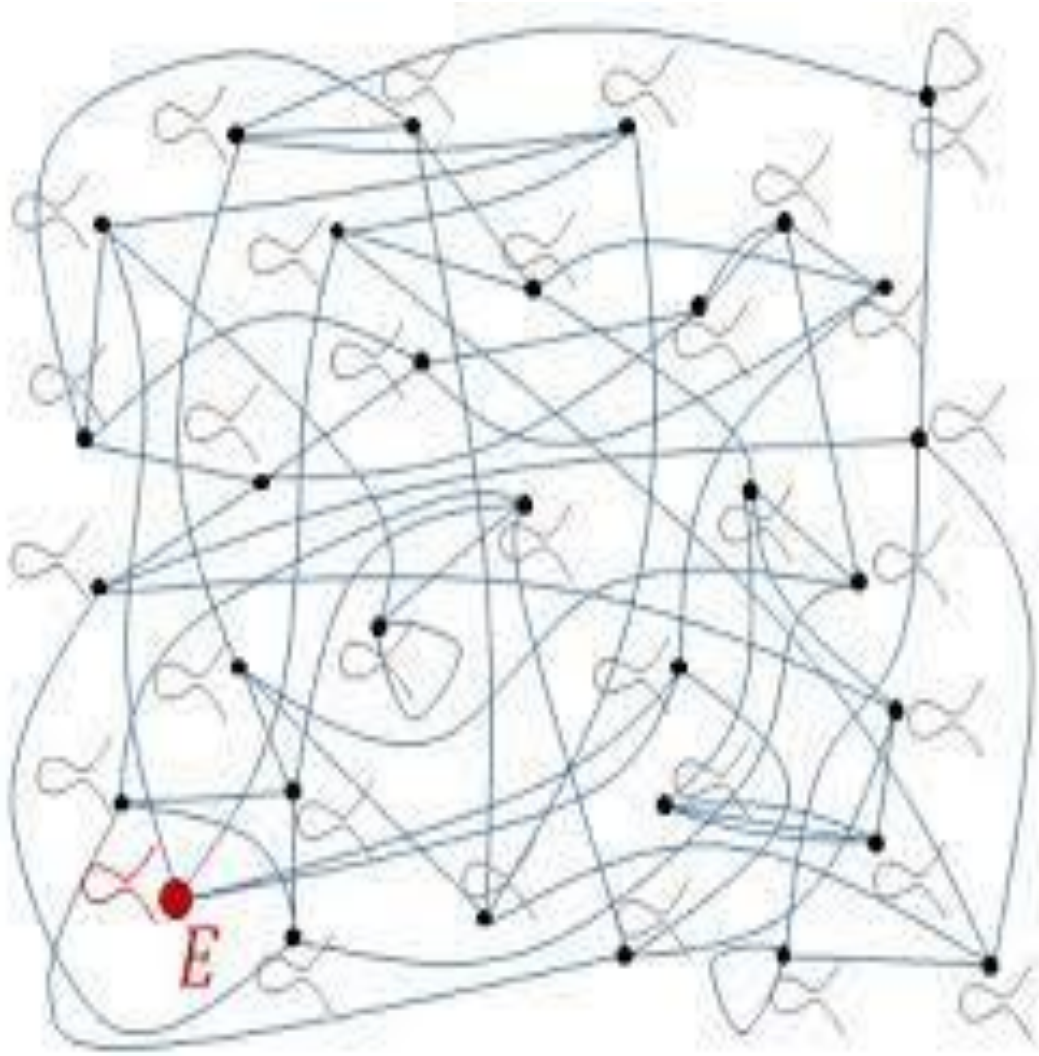


Asymmetric Encryption



Elliptic curve cryptography (ECC)

- **Asymmetric encryption**
  - Requires large integers composed of two or more large prime numbers
- **Instead of numbers, use curves !!!**
  - Smaller storage and transmission requirements
  - Perfect mobile devices



Public Key Cryptography

There is a huge difference between private key crytography and public key cryptosystems: the aspiration itself !!!

ASYMMETRIC CRYPROGRAPHY (public key cryptosystems)

→ we use trapdoor functions: so we rely heavily on the fact that there are some operations that are extremely hard to do (exponential running time complexity)

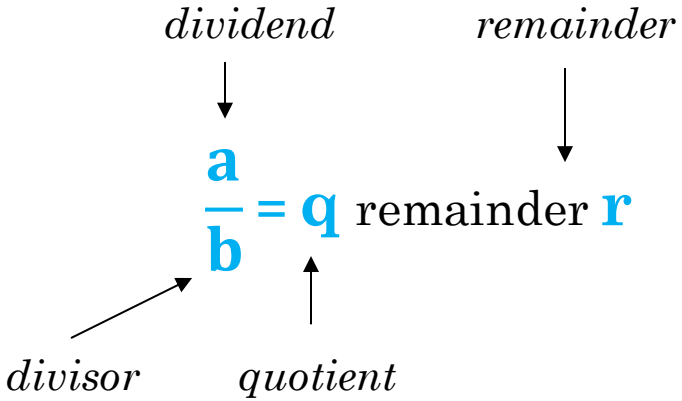
→ this is why we have to talk about modular arithmetic

For example: prime factorization or the discrete logarithm problem

PUBLIC KEY CRYPTOSYSTEMS ARE ABOUT PRIME NUMBERS !!!

Modular Arithmetic

„Modular arithmetic is a system of arithmetic for integers where numbers wrap around upon reaching a certain value - the modulus”



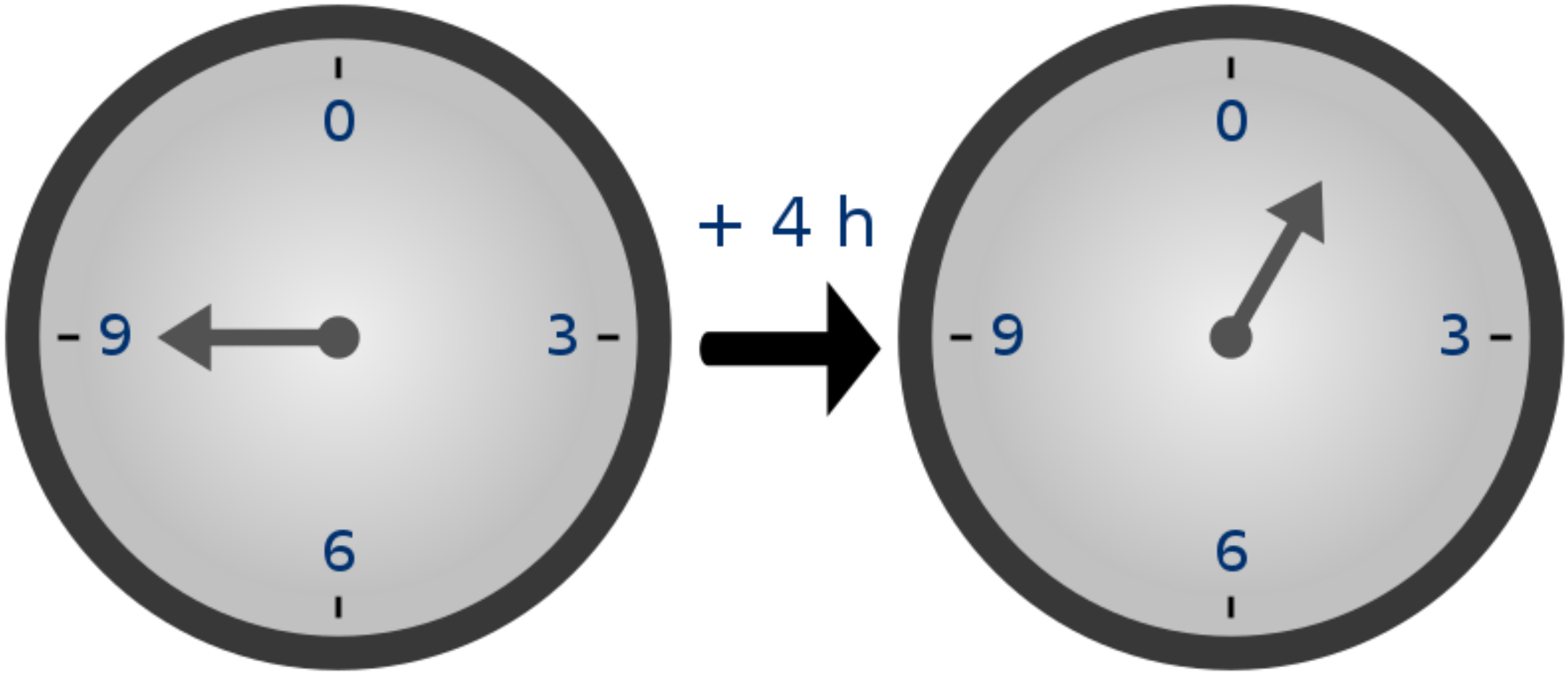
9 mod **4** = **1** because **9** = **2**x**4**+**1**

13 mod **10** = **3** because**13**=**1**x**10**+**3**

→ in modular arithmetic we are only interested in the **remainder**

→ to get the remainder we can use the modulo operator (%)

a mod **b** = remainder when we divide **a** by **b**


0
9
3
6
+ 4 h
0
9
3
6

Modular Arithmetic

CONGRUENCE

Two integers (**a** and **b**) are said to be congruent if they have the same remainder when divided by a specified integer **m**

$a \equiv b \pmod{m}$ $17 \equiv 32 \pmod{5}$

→ congruence is not an operation it just defines a relationship between **a** and **b**

→ the **mod m** operation partition all the natural numbers into **m** subgroups

For example: the **mod 2** operator decides whether a given number is even or odd (so **2** groups)

Modular Arithmetic

FERMAT'S LITTLE THEOREM

What is a prime number?

A prime number greater than **1** whose only factors are **1** and itself
~ numbers that have more than **2** factors are called composite numbers

Let **p** be a prime number then for any integer **a** (**a** is not divisible by **p**) the number

$\mathbf{a^{p-1}}$ is an integer multiple of **p**.

$$a^{p-1} \equiv 1 \pmod{p} \quad \text{or} \quad a^{p-1} - 1 \equiv 0 \pmod{p}$$

„Fermat's little theorem"

```
while True:
    a='no'
    try:
        a,b=map(int,input("Enter two integers: ").split())
        c=int(input("Enter the divisor: "))
        f=int(input("Enter integer to check whether prime or not: "))
        break
    except ValueError:
        print("Input numbers correctly!!!")
    except KeyboardInterrupt:
        print("If you want to exit just type 'yes'!!!\n")
        a=input("Here: ").lower()
    if a=='yes':
        print("Exiting...")
        break
try:
    def isCongruence(a,b):
        if a%c==b%c: return True
        return False

    def isPrime(a):
        for i in range(2,int(a**0.5)+1):
            if a%2==0: return False
        return True

    if (isCongruence(a,b)):k=' '
    else:k=" not "
    print("{0} and {1} are{3}congruence when divided by {2}".format(a,b,c,k))

    if(isPrime(f)):k=' '
    else:k=" not "
    print("{0} is{1}prime number".format(f,k))
except:
    pass
```

```
================ RESTART: C:/Users/Nici/Desktop/ikincigunders.py ==============
Enter two integers: 13 23
Enter the divisor: 5
Enter integer to check whether prime or not: efwe
Input numbers correctly!!!
Enter two integers: 13 23
Enter the divisor: 5
Enter integer to check whether prime or not: 7
Enter two integers:
================ RESTART: C:/Users/Nici/Desktop/ikincigunders.py ==============
Enter two integers: 13
Input numbers correctly!!!
Enter two integers: 13 23
Enter the divisor: 5
Enter integer to check whether prime or not: 7
13 and 23 are congruence when divided by 5
7 is prime number
>>>
```

Modular Arithmetic

FINDING PRIME NUMBERS

Running time complexity:

→ in other algorithms and data structures running time analysis the size of the input is straightforward

Sorting: the input is the **N** numbers we want to sort and the running time complexity is **O(NlogN)**

Shortest path: input is the **N** vertexes in the graph

→ now the input is a large number BUT we represent the numbers in binary in computer science

Modular Arithmetic

FINDING PRIME NUMBERS

- Native approach $O(n^2)$

1. Iterate over as much integers as required or possible
2. Check if current number N is divisible by at least one number in range from 2 to $N - 1$;

- Smart approach

1. Iterate over as much integers as required or possible
2. Check if current number N is divisible by at least one number in range from 2 to $\sqrt{N}$;

Modular Arithmetic

FINDING PRIME NUMBERS

Running time complexity:

When dealing with the naive primality test we end up with $\mathbf{O(\sqrt{N})}$ running time
BUT now the input is a large number …

→ we have to use a different approach

→ the input length **n** is the number of bits in the input

So in our examples the input is a decimal number. So first of all we have to define the number of bits of a decimal number

The input length in binary is $\mathbf{n = \log_2 N}$

It means that $\mathbf{O(\sqrt{N})}$ is in fact $\mathbf{O(2^{\frac{n}{2}})}$ which is exponential running time

~ of course it makes sense because the naive primality testing algorithm is quite a slow approach

*deciding whether a number is prime is crucial in **RSA** so exponential running time is too slow*

Modular Arithmetic

FINDING PRIME NUMBERS

- Eratosthenes sieve

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting in increments of p from 2p to n, and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself should not be marked).
4. Find the smallest number in the list greater than p that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.

	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
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

Modular Arithmetic

FINDING PRIME NUMBERS

Fermat's algorithm: we can use Fermat's little theorem to check whether a given **N** number if prime or not

*„number **a** is the witness for compositeness of **N**"*

$$a^{N-1} \equiv 1 \pmod{N}$$

If this relation is true than we know that **N** is a prime

In other words: if **N** is a prime number then for every **1 <= a < N** number $a^{N-1} \equiv 1 \pmod{N}$ which means in programming that a^{N-1} **% N = 1**

For example: since **5** is prime thats why 2^4 **% 5 = 1** so **5** is prime

Running time complexity: $O(k \log^3 n)$

Modular Arithmetic

FINDING PRIME NUMBERS

Question 1: Is 5 prime?

Solution:

$a^p - a \longrightarrow$ 'p' is prime if this is a multiple of 'p' for all $1 \leq a < p$.

$1^5 - 1 \equiv 1 - 1 = 0$

$2^5 - 2 \equiv 32 - 2 = 30$

$3^5 - 3 = 243 - 3 = 240$

$4^5 - 4 = 1024 - 4 = 1020$

Modular Arithmetic

FINDING PRIME NUMBERS

Fermat's algorithm: we can use Fermat's little theorem to check whether a given **N** number if prime or not

$$a^{N-1} \equiv 1 \pmod{N}$$

repeat **k** times:

generate **a** random number in the range **[2,N-2]**

if $\mathbf{a^{N-1}}$ **% N = 1** then **N** is probably prime

FERMAT'S ALGORITHM IS PROBABILISTIC !!!

~ the probability of producing incorrect results for composite numbers is low and can be reduced by doing more **k** iterations

this test fails with Carmichael numbers
PROBLEM

if **gcd(a,n) = 1** then Fermat test is not valid

Modular Arithmetic

INTEGER FACTORIZATION

„Integer factorization is the decomposition of a composite number into a product of smaller integers: usually we are interested in prime numbers"

THIS IS CALLED PRIME FACTORIZATION

Fundamental Theorem of Arithmetic

This theorem states that every positive integer can be written uniquely as a product of prime numbers

For example: **210 = 2 x 3 x 5 x 7**

→ prime factorization is a „trapdoor-function"

→ extremely easy to compute the result by multiplying the factors but extremely hard to find the factors for large numbers

Modular Arithmetic

INTEGER FACTORIZATION

Trapdoor-functions are crucial in cryptography: the difficulty of factoring large integers is the basis of some modern cryptographic algorithms (**RSA**)

→ **SSL** encryption used for **TCP/IP** connections relies on the security of the **RSA** algorithm

→ if a fast approach is invented to factor large integers then internet sites would no longer be secure

Modular Arithmetic

DISCRETE LOGARITHM PROBLEM

Calculating the discrete logarithm is another trapdoor-function

$$a \equiv b^{c} \pmod{m}$$

If we know **b**, **c** and **m** then this is called **modular exponentiation** which is not that hard to solve

→ what is the inverse of this operation?

If we know **a**, **b** and **m** then this is called the **discrete logarithm problem** which is a very difficult problem to solve

Successive Squaring

- Useful if the exponent is a power of 2.
- Example: Find $3^{128} \ mod \ 7$
  - $3^2 \ mod \ 7 = 9 \ mod \ 7 = 2$
  - $3^4 \ mod \ 7 = 9^2 \ mod \ 7 = 81 \ mod \ 7 = 4$
  - $3^8 \ mod \ 7 = 4^2 \ mod \ 7 = 16 \ mod \ 7 = 2$
  - $3^{16} \ mod \ 7 = 2^2 \ mod \ 7 = 4$
  - $3^{32} \ mod \ 7 = 4^2 mod \ 7 = 2$
  - $3^{64} \ mod \ 7 = 2^2 \ mod \ 7 = 4$
  - $3^{128} \ mod \ 7 = 4^2 \ mod \ 7 = 2$
  - Thus, $3^{128} \ mod \ 7 = 2.$

Modular Exponentiation

- It is a type of exponentiation performed over a modulus

```cpp
#include <iostream>
#include <map>
using namespace std;
map<int,int> arr;
int main(){
    int a,b,n;
    cin >> a >> b >> n;
    int s;
    arr[1]=a%n;
    for (int i=2;i<=b;i=i*2){
        arr[i]=(arr[i/2]*arr[i/2])%n;
        s=i;
    }
    int k=b;
    int mod=1;
    for (int i=s;i>=1;i=i/2){
        if (k/i==1){
            mod*=arr[i];
            k=k%i;
        }
    }
    cout << mod%n << endl;
}
```

```
# Iterative Function to calculate (x^y) in O(log y)
def power(x, y):

    # Initialize result
    res = 1

    while (y > 0):

        # If y is odd, multiply x with result
        if ((y & 1) != 0):
            res = res * x

        # y must be even now
        y = y >> 1 # y = y/2
        x = x * x  # Change x to x^2

    return res

print(power(5,7))
```

```
# Iterative Function to calculate
# (x^y)%p in O(log y)
def powerAndMod(x, y, p) :
    res = 1      # Initialize result

    # Update x if it is more
    # than or equal to p
    x = x % p

    if (x == 0) :
        return 0

    while (y > 0) :

        # If y is odd, multiply
        # x with result
        if ((y & 1) == 1) :
            res = (res * x) % p

        # y must be even now
        y = y >> 1       # y = y/2
        x = (x * x) % p

    return res
print(powerAndMod(2,5,13))
```

Modular Arithmetic

DISCRETE LOGARITHM PROBLEM

Calculating the discrete logarithm is another trapdoor-function

What is the running time complexity of modular exponentiation?

Modular exponentiation is relatively straightforward operation
~ have to use exponentiation with modulo operator

O(e) running time complexity

In this case **e** is the number of digits in the exponent !!!

What is the running time complexity of discrete logarithm problem?

Finding the right exponent for the discrete logarithm problem is extremely hard: it has exponential running time complexity

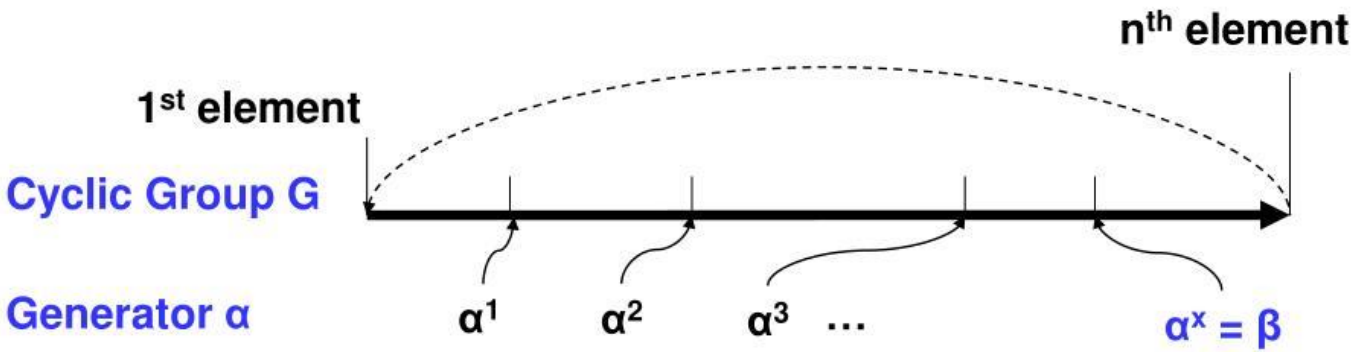
Discrete Logarithm Problem

- The discrete logarithm problem is used in cryptography. Given values for a, b, and n (where n is a prime number), the function x = (a^b) mod n is easy to compute. For example, if a = 3, b = 4, and n = 17, then x = (3^4) mod 17 = 81 mod 17 = 81 mod 17 = 13. But if you have values for x, a, and n, the value of b is very difficult to compute when the values of x, a, and n are very large.
- If you set a value for a and n, and then compute x iterating b from 1 to n-1, you will get each value from 1 to n in scrambled order — a permutation. For example, if a = 3 and n = 17, then:
- `b = 1, x = 3^1 % 17 = 3`
- `b = 2, x = 3^2 % 17 = 9`
- `b = 3, x = 3^3 % 17 = 27 % 17 = 10`
- `b = 4, x = 3^4 % 17 = 81 % 17 = 13`
- `. . .`
- `b = 16, x = 3^16 % 17 = 43046721 % 17 = 1`

Discrete Logarithm Problem

- known: large prime number p, generator g
- $g^k \bmod p = x$
- Discrete logarithm problem: given x, g, p, find k
- Table g=2, p=11

k	1	2	3	4	5	6	7	8	9	10
g^k	2	4	8	5	10	9	7	3	6	1



```
# Iterative Function to calculate (x^y) in O(log y)
def power(x, y):

    # Initialize result
    res = 1

    while (y > 0):

        # If y is odd, multiply x with result
        if ((y & 1) != 0):
            res = res * x

        # y must be even now
        y = y >> 1 # y = y/2
        x = x * x  # Change x to x^2

    return res


def findPower(a,x,n):
    for i in range(0,n):
        if (power(a,i)%n==x):
            return i
```

Public Key Cryptography

„A cryptosystem should be secure even if everything about the system except the key is public knowledge"

→ this is called the **Kerckhoff's principle**

→ it is the fundamental principle of crytography

This is why **we like prime numbers**:

If we have two prime numbers **p** and **q** then multiplying them is quite easy **M = p x q** but calculating the factors if we have **M** is extremely hard
~ this is called integer factorization

Public key cryptosystem: integer factorization is a good „trapdoor-function" which means it is easy to verify (we just have to multiply the numbers) but calculating the factors is almost impossible (without quantum computers)

Public Key Cryptography

Why is it important to use prime numbers at all?

→ factoring large numbers is usually hard: but not always

→ if a given number has smaller factors then it may happen that the factors can be found within hundreds or thousands of iterations

So somehow we have to make sure the prime factors will be large …

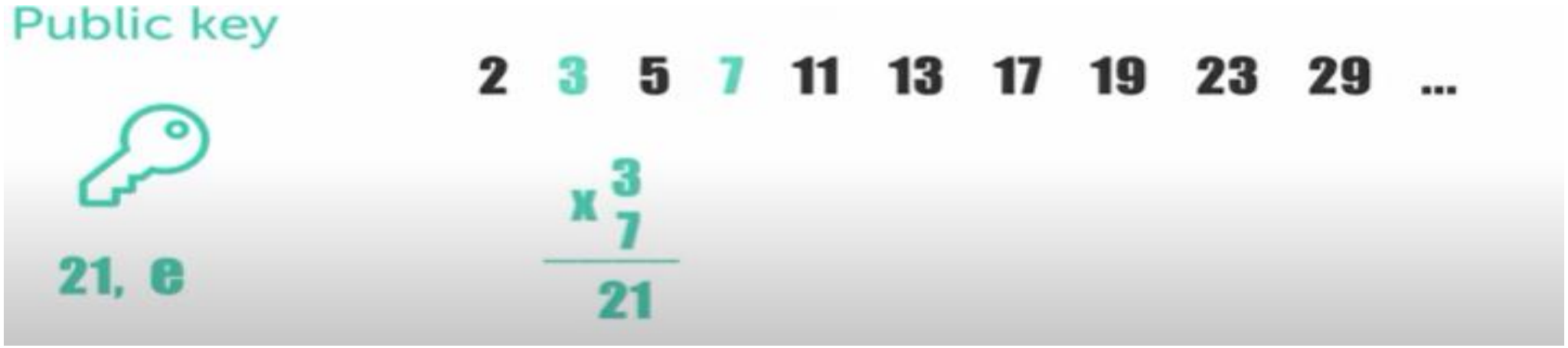
This is where prime numbers have been proved to be important: if we have **p** and **q** large prime numbers

then we can calculate **N = p*q** quite fast

What are the factors of **N**? Of course the factors are **p** and **q** and we know that these are large primes (this is exactly why we chose them)

THE REASON WHY WE USE PRIME NUMBERS IS TO MAKE SURE FACTORIZATION IS PRACTICALLY IMPOSSIBLE

Generating public key



Generating public key

Public key

21, e

3 - p
x
7 - q
---
21 - mod

Euler (phi-) function

$\varphi = (p - 1) * (q - 1) = (3 - 1) * (7 - 1) = 12$

e:

1. Prime number;
2. < φ (12); ~~2, 3,~~ 5, 7, 11
3. Coprime with φ.

Public key: { e, mod } = { 5, 21 }

Generating private key

d - reverse mod of e by φ

Private key

21, d

$$[d * e] \% \varphi = 1$$

$$[d * 5] \% 12 = 1$$

$$[17 * 5] \% 12 = 1$$

$$17*5 - 12*7 = 1$$

$$85 - 84 = 1$$

Private key: { d, mod } = { 17, 21 }

Testing of key pair

Public key:

$\{e, mod\} = \{5, 21\}$

$X < mod\ (21)$

11

11^{5}

$161\,051$

$161\,051 \% 21 = 2$

$161\,051 - 7669 * 21 = 2$

$3 * 7$

mod

21

Send message

Private key:

$\{d, mod\} = \{17, 21\}$

2

2^{17}

$131\,072$

$131\,072 \% 21 = 11$

$131\,072 - 6241 * 21 = 11$

Sources

- https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
- https://www.geeksforgeeks.org/sieve-of-eratosthenes/
- https://www.geeksforgeeks.org/primality-test-set-2-fermet-method/
- https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/
- https://www.youtube.com/watch?v=sDrXeCs3ghQ
- https://www.youtube.com/watch?v=xqPgE-hPIfE
- https://github.com/IZOBRETATEL777/Cryptography-lesson

Thank you for your attention!