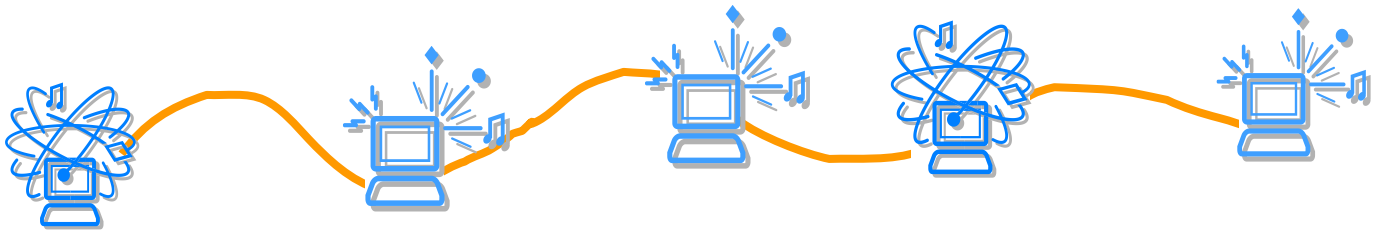
Segurança em Redes

Conceitos de criptografia – Cifra assimétrica



Redes de Comunicação de Dados

Departamento de Engenharia da Electrónica e das Telecomunicações e de Computadores

Instituto Superior de Engenharia de Lisboa

Bibliografia extra

No que se refere a aritmética modular, para além da bibliografia aconselhada para a unidade curricular:

"Cryptography and Network Security, William Stallings"

pode ser consultado:

https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

“Matemática” utilizada nas cifras simétricas

- São utilizadas funções muito simples:
  - Ou exclusivo (xor)
  - Permutação/transposição de dados
  - Substituição de dados
  - *Shifts*

Chaves de cifra simétrica

- **Idealmente as chaves deveriam ser números aleatórios puros**

- Embora isto seja possível:
  - As chaves são pré-colocadas em cada extremo
  - Números aleatórios puros podem ser gerados capturando ruído estrelar, ruído dos díodos, etc,.
  - As partes comunicantes necessitam apenas acordar em que parte da chave gerada vão começar
  - A chave não tem de obedecer a qualquer função matemática além da aleatoriedade

- A maioria das implementações usam chaves pseudo-aleatórias.

Problema da distribuição de chaves

- As chaves secretas devem ser preposicionadas em todos os locais antes de poderem ocorrer comunicações seguras.

- Como fazer isso?
  - Transporte físico seguro
  - Transporte electrónico seguro

A procura de uma forma de conseguir a distribuição de chaves secretas levou ao desenvolvimento da:

Criptografia de chave pública ou assimétrica

(public key cryptography)

Symmetric (DES) vs. Public Key (RSA)

- Exponentiation of RSA is expensive !

- **AES and DES are much faster**
  - **100 times faster in software**
  - **1,000 to 10,000 times faster in hardware**

- RSA often used in combination in AES and DES
  - Pass the *session key* with RSA

Criptografia assimétrica

Condições para os algoritmos de chave pública:

- Par de chaves (pública e privada) fáceis de gerar: K_U e K_R
- Fácil de calcular: $C=E_{KU}(M)$ e $M=D_{KR}(C)$
- Muito difícil achar a chave privada (K_R) a partir da chave pública (K_U)
- Muito difícil achar M, tendo apenas K_U e C
- $M = E_{KU}(D_{KR}(M))$; utilização possível para assinatura digital
  - M: Texto em claro
  - C: Texto cifrado
  - K_u: Chave pública
  - K_r: Chave privada
  - $E_{Ku}()$: Função de cifrar
  - $D_{Kr}()$: Função de decifrar

As cifras assimétricas baseiam-se na matemática da teoria dos números.

Mathematics and cryptography

A large proportion of modern, **asymmetric cryptography is based on mathematical knowledge on the properties (“laws") of whole numbers**, which are investigated in **elementary number theory**. Here, the word “elementary" means that questions raised in number theory are essentially rooted in the set of natural and whole numbers.

Further **mathematical disciplines currently used in cryptography** include:

- Group theory
- Combination theory
- Complexity theory
- Ergodic theory
- Information theory.

[The CrypTool Script Cryptography, Mathematics, and More, Prof. Bernhard Esslinger and the CrypTool Development Team]

Mathematics and cryptography

Number theory or arithmetic (the emphasis here is more on the aspect of performing calculations with numbers) **was established by Carl Friedrich Gauss** as a special mathematical discipline. Its elementary features include the **greatest common divisor (gcd)**, **congruence (remainder classes)**, **factorization**, **the Euler-Fermat theorem** and **primitive roots**. However, the most important aspect is **prime numbers** and their **multiplicative operation**.

Introduction to number theory

Number theory arose from interest in **positive whole numbers 1; 2; 3; 4; …** , also referred to as the **set of natural numbers N**.

Modern number theory is made up of areas such as:

- Elementary number theory
- Algebraic number theory
- Analytic number theory
- Geometric number theory
- Combinatorial number theory
- Numeric number theory
- Probability theory.

Here we need only the **elementary number theory**.

Convention

Unless stated otherwise:

- The letters **a, b**, **c**, **d**, **e**, **k**, **n**, **m**, **p**, **q** are used to present whole numbers.
- The letters **i** and **j** represent natural numbers.
- The letters **p** always represents a prime number.
- The sets **N** = {1, 2, 3,…} and **Z** = {…,-3,-2,-1,0,1,2,3,…} are the **natural** and **whole** numbers respectively.

Divisibility, modulus and remainder classes

If whole numbers are added, subtracted or multiplied, the result is always another whole number.

The division of two whole numbers does not always result in a whole number. For example, if we divide 158 by 10 the result is the decimal number 15,8, which is not a whole number!

If, however, we divide 158 by 2 the result 79 is a whole number. In number theory we express this by saying that 158 is divisible by 2 but not by 10. In general, we say:

Definition: A whole number **n** is divisible by a whole number **d** if the quotient **n/d** is a whole number **c** such that **n = c*d**

n is called a multiple of **d**, whereas **d** is called a divisor or factor of **n**.

The mathematical notation for this is **d|n (read “d divides n")**.

Quotient Remainder theorem

The **Quotient Remainder theorem** says:

Given **any** integer ***a***, and a **positive** integer ***b***, there exist **unique integers *q* and *r*** such that

$$\mathbf{a = b * q + r} \quad \text{where } \mathbf{0 \le r < b;} \qquad \mathbf{r = [a/b]}$$

$$a = [a/b] * b + (a \bmod b)$$

We can see that this comes directly from long division. When we **divide *a* by *b*** in long division, *q* is the quotient and ***r* is the remainder**.

If we can write a number in this form, then **a mod b = r**

Note: **mod b** represents arithmetic module **b**

Aritmética modular

Exemplo:

- Módulo 3, por exemplo, divide qualquer inteiro positivo por 3 e fica com o resto (o qual terá um valor de 0, 1, ou 2).

  7 mod 3 = 1, dado que 3 cabe em 7 duas vezes, e sobra 1.

  O número pelo qual se divide (3, neste exemplo) é designado por "módulo".

- De forma semelhante, **62 mod 25 = 12**. Isto é, quando **25** é o módulo, **62 = 25 x 2 + 12** => **resto = 12**. Mas se **3** for o módulo, **62 = 3 x 20 + 2** => **resto = 2**.

Divisibility and Primality

The symbol "|" means "**divides**". $b|a$: b divides a or a is divided by b

Theorem: For all a, b, c ∈ Z, we have

(i) a|a, 1 | a, and a|0

(ii) 0|a if and only if a = 0

(iii) a|b and a | c implies a|(b + c)

(iv) a|b implies a|−b

(v) a|b and b|c implies a|c

(vi) If $b|g$ and $b|h$ then $b|(mg + nh)$ for any m e n integer

Theorem: For all a, b ∈ Z, we have: **a|b and b|a if and only if a = ±b**

Aritmética modular

Em aritmética modular, o resultado da operação sobre um número é igual ao resto da divisão desse número por outro. Isto quer dizer:

"dividir por **p** e manter o resto **r**, **0<= r < p**“

mod p representa “aritmética segundo o módulo p”

Por exemplo (p=5):

11 mod 5 = 1 (11 = 5x2 + 1)

Existem muitos números que têm a mesma representação modular:

21 mod 5 = 1 ; 41 mod 5 = 1 ; 56 mod 5 = 1 ; etc.

21, 41 e 56 são congruentes módulo 5 (quando divididos pelo mesmo valor, neste caso 5, dão o mesmo resto, neste caso 1)

The modulo operation - working with congruence

When we investigate divisibility, it is only the remainder of the division that is important.
When dividing a number **n** by **m**, we often use the following notation:

$$n = c*m+r \;\rightarrow\; n/m = c + r/m$$

where **c** is a whole number and **r** is a number with the values **0, 1,…,m-1**. This notation is called division with remainder, whereby **c** is called the whole-number "quotient" and **r** is the "remainder" of the division.

Example: **19/7= 2 + 5/7; (m = 7; c = 2; r = 5)**

What do the numbers 5, 12, 19, 26,… have in common for division by 7? The remainder is always r = 5. For division by 7, only the following remainders are possible:

r = 0, 1, 2, …, 6.

The numbers that result in the same remainder r when divided by 7 are combined to form the "**remainder class r modulo 7**". Two numbers **a** and **b** belonging to the same remainder class modulo 7 are said to be "**congruent modulo 7".** Or in general:

Definition: The **remainder class r modulo m** is the set of all whole numbers **a** that have the same remainder **r** when divided by **m.**

Aritmética modular - Congruência

Em geral escreve-se **a ≡ b mod n**, com o significado de:

a mod n = b mod n

Exemplo:

67 ≡ 11 mod 7 dado que 67 mod 7 **= 4** e 11 mod 7 **= 4**

Então **a ≡ b mod n** significa que tendo os inteiros **k1** e **k2**,

a = **k1** x n + r **(**0<= r < n), b = **k2** x n + r

o que, subtraindo, dá:

a - b = (**k1** x n + r) – (**k2** x n + r) = **k1** x n – **k2** x n + r – r = (**k1-k2**) x n,

de maneira a **n** dividir **a-b** um número inteiro de vezes (nomeadamente, **k1-k2** vezes). Isto é, "**n** divide **a-b**", o que também pode ser escrito "**n**|**(a-b**)".

Aritmética modular

Então, no exemplo anterior,

$$67 \equiv 11 \bmod 7$$

Significa que, para os inteiros k1 = 9 e k2 = 1,

$$67 = 9 \times 7 + 4$$

$$11 = 1 \times 7 + 4$$

Então,

$$67-11 = (k1-k2) \times 7 = (9-1) \times 7 = 8 \times 7,$$

dado que 7 divide (67 – 11) um número inteiro de vezes (8 vezes) também se pode escrever $7 | (67-11)$ ou $7 | 56$.

Properties of Modular Arithmetic for Integers in Z_n

Property	Expression
Commutative Laws	$(w + x) \bmod n = (x + w) \bmod n$ $(w \times x) \bmod n = (x \times w) \bmod n$
Associative Laws	$[(w + x) + y] \bmod n = [w + (x + y)] \bmod n$ $[(w \times x) \times y] \bmod n = [w \times (x \times y)] \bmod n$
Distributive Law	$[w \times (x + y)] \bmod n = [(w \times x) + (w \times y)] \bmod n$
Identities	$(0 + w) \bmod n = w \bmod n$ $(1 \times w) \bmod n = w \bmod n$
Additive Inverse ($-w$)	For each $w \in Z_n$, there exists a z such that $w + z \equiv 0 \bmod n$

Aritmética modular

Módulo (mod) n: O resultado é um inteiro não-negativo menor que o inteiro n

Adição modular

Exemplos com **mod 10:**

5+5=0, 3+9=2, 2+2=4, 7+8=5, …

(a + b) mod c = (a mod c + b mod c) mod c

Subtração modular

(a - b) mod c = (a mod c - b mod c) mod c

Inverso aditivo: x+?=0 => **x + ? mod 10 = 0**

Exemplo com **x = 4** e **mod 10**: 4+6=0, 4+16=0, 4+106=0, …

Multiplication property of modular arithmetic

(a * b) mod c = (a mod c * b mod c) mod c

Example for Multiplication:

Let **a=4, b=7, c=6**

Let's verify: **(a * b)** mod c = (**a mod c** * **b mod c**) mod c

LHS = Left Hand Side of the Equation; **RHS** = Right Hand Side of the Equation

LHS = **(a * b)** mod c

LHS = **(4 * 7)** mod 6 = **28** mod 6 = **4**

RHS = (**a mod c** * **b mod c**) mod c

RHS = (**4 mod 6** * **7 mod 6**) mod 6

RHS = (**4 * 1**) mod 6 = **4 mod 6** = **4**

LHS = RHS = 4

Exponentiation property of modular arithmetic

$$a^b \bmod c = ((a \bmod c)^b) \bmod c$$

Often we want to calculate $a^b \bmod c$ for large values of b.
Unfortunately, a^b becomes very large for even modest sized values for b.

Example:

2^90 = 1237940039285380274899124224

7^256 =
2213595400046048155450188615474945937162517050260073069916366390524704974007989996848003433837940380782794455262312607598867363425940560014856027866381946458951205837379116473663246733509680721264246243189632348313601

These huge values cause our calculators and computers to return **overflow errors**.

Exponentiation property of modular arithmetic

Even if they didn't, it would take a *long time* to find the mod of these huge numbers directly.

What can we do to reduce the size of terms involved and make our calculation faster?

Suppose we want to calculate **2^{90} mod 13**, but we have a calculator that can't hold any numbers **larger than 2^{50}**.

Here is a simple **divide and conquer strategy**:

Step 1: Split a^b into **smaller parts** using **exponent rules** $2^{90} = 2^{(50+40)} = 2^{50} * 2^{40}$

Step 2: Calculate the **mod c** for **each part 2^{50}** mod 13 = **1125899906842624** mod 13 = **4**

2^{40} mod 13 = **1099511627776** mod 13 = **3**

Step 3: Use modular **multiplication properties** to **combine the parts 2^{90}** mod 13 = (**2^{50} * 2^{40}**) mod 13 = (**2^{50}** mod 13 * **2^{40}** mod 13) mod 13 = (**4** * **3**) mod 13 = **12** mod 13 = **12 or:**

2^{90} mod 13 = 12

The modulo operation - working with congruence

Definition: *Two numbers a, b ∈ N are said to be **congruent modulo n** ∈ N if and only if they have the same remainder when divided by m [a≡b (mod n)].*

Theorem: *a ≡ b (mod n) if and only if, the difference (a - b) is divisible by n, i.e. if k ∈ Z exists with (a - b) = k * n.*

These two statements are therefore equivalent.

Therefore: If m divides the difference, there exists a whole number **k** such that: **a = b+k*n**.

As an alternative to the congruence notation, we can also use the divisibility notation: **n|(a-b)**.

Example of equivalent statements:

35 ≡ 11 (mod 3) <=> 35 - 11 ≡ 0 (mod 3), where 35 - 11 = 24 is divisible by 3 without remainder while 35 / 3 and 11 / 3 leave the remainder 2.

The modulo operation - working with congruence

Example:

Remainder class 0 modulo 4 = {x|x = 4*n; n ∈ Z} = {…,-16, -12, -8, -4, 0, 4, 8, 12, 16,…}

Remainder class 3 modulo 4 = {x|x = 4* n + 3; n ∈ Z} = {…,-13, -9, -5, -1, 3, 7, 11, 15,…}

As only the remainders **0, 1, 2,…,m-1** are possible for division modulo **m**, **modular arithmetic works with n finite sets**. For each modulo **m** there are precisely **m** remainder classes.

Definition: *Two numbers a, b ∈ N are said to be **congruent modulo m** ∈ N if and only if they have the same remainder when divided by m.*

a≡b (mod m) (read **a** is congruent **b** modulo **m**)

The modulo operation - working with congruence

We write: **a≡b (mod m)** (read **a** is congruent **b** modulo **m**), which means that **a** and **b** belong to the same remainder class. The modulo is therefore the divisor. This notation was introduced by Gauss. Although the divisor is usually positive, **a** and **b** can also be any whole numbers.

Example:

19 ≡ 12 (mod 7), because the remainders are equal: 19/7 = 2 remainder **5** and 12/7 = 1 remainder **5**.

23103 ≡ 0 (mod 453), because 23103/453 = 51 remainder 0 and 0/453 = 0 remainder 0.

Theorem: a ≡ b (mod m) if and only if, the difference (a - b) is divisible by m, i.e. if q ∈ Z exists with (a - b) = q * m.

Modular Arithmetic – Properties of congruence

- **Proposition 1** (**Reflexivity** of modular congruence). **If a and n are integers, then a ≡ a (mod n).**
- **Proposition 2** (**Symmetry** of modular congruence). **For integers a, b, and n, if a ≡ b (mod n), then b ≡ a (mod n).**
- **Proposition 3** (**Transitivity** of modular congruence). For integers **a, b, c, and n, if a ≡ b (mod n) and b ≡ c (mod n), then a ≡ c (mod n).**
- **Proposition 4** (**Additivity** of modular congruence). **For integers a, b, c, d, and n, if a ≡ c (mod n) and b ≡ d (mod n), then a + b ≡ c + d (mod n).**
- **Proposition 5** (**Multiplicitivity** of modular congruence). **For integers a, b, c, d, and n, if a ≡ c (mod n) and b ≡ d (mod n), then ab ≡ cd (mod n).**

Modular Arithmetic – Properties of congruence

- **Proof of the first point:**
  1. **a ≡ b (mod n) if n (a – b)**
  2. a ≡ b (mod n) implies b ≡ a (mod n)
  3. a ≡ b (mod n) and b ≡ c (mod n) imply a ≡ c (mod n)

- To demonstrate the first point, if **n (a – b)**, then (a − b) = kn for some k
  - So we can write a = b + kn
  - Therefore, (a mod n) = (remainder when b + kn is divided by n) = (remainder when b is divided by n) = (b mod n)

    23 ≡ 8 (mod 5) because 23 −8 = 15 = 5 * 3
    −11 ≡ 5 (mod 8) because −11 −5 = −16 = 8 * (− 2)
    81 ≡ 0 (mod 27) because 81 − 0 = 81 = 27 * 3

Aritmética modular - Congruência

Outras propriedades

Se a, b, c, d, m e n forem quaisquer inteiros com

$a \equiv b \pmod m$ e $c \equiv d \pmod m$, então:

$a+c \equiv b+d \pmod m$

$a-c \equiv b-d \pmod m$

$a*c \equiv b*d \pmod m$

Não há divisão, há inversos modulares.

$a^n \equiv b^n \pmod m$

Se gcd(c, m) = 1 e $a*c \equiv b*c \pmod m$ então $a \equiv b \pmod m$

[gcd – máximo divisor comum]

Greatest Common Divisor (gcd)

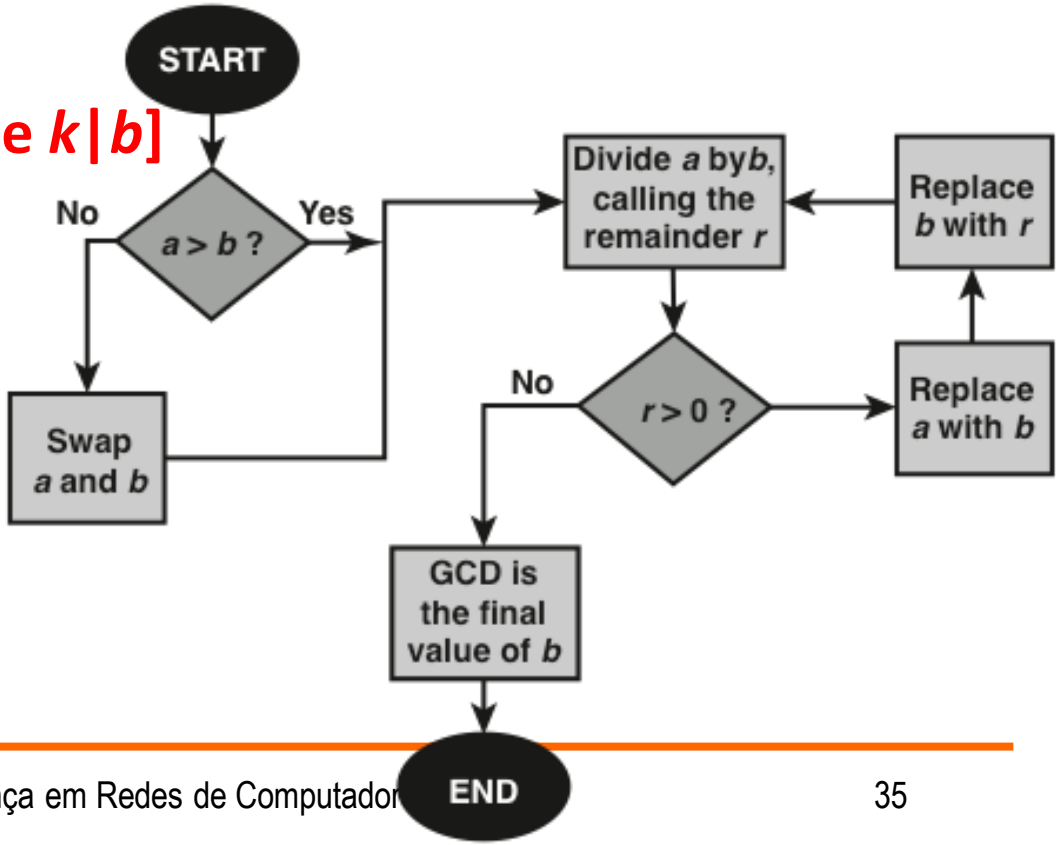
Um inteiro positivo *k* diz-se **máximo divisor comum** (**gcd – *greatest common divisor***) de *a* e *b* se:

- *k* for divisor de *a* e de *b* (*k*|*a* e *k*|*b*);
- qualquer divisor de *a* e *b* for um divisor de *k*

Definição equivalente:

gcd(*a*,*b*) = max [*k*, tal que *k*|*a* e *k*|*b*]

gcd (26, 7) = 1
26 = 7 * 3 + 5
7 = 5 * 1 + 2
5 = 2*2 + 1
2 = 1*2 + 0



Euclidean Algorithm

The Euclidean Algorithm for finding gcd(a, b) is as follows:

- If a = 0 then gcd(a, b) = b, since the gcd(0,b) = b, and we can stop.
- If b = 0 then gcd(a, b) = a, since the gcd(a,0) = a, and we can stop.
- Write *a* in quotient remainder form (a = b·q + r)
- Find gcd(b, r) using the Euclidean Algorithm since **gcd(a, b) = gcd(b, r)**

Algoritmo de Euclides

É um algoritmo simples para determinar o máximo divisor comum (**gcd**) de dois inteiros positivos *a* e *b* (*a*>=*b*):

```
function gcd(a, b) {
   if b = 0
        return a
   else
        return gcd(b, a mod b)
}
```

```
function gcd(a,b) {
      while b<>0 {
            r=a mod b;
            a=b;
            b=r;
      }
      return a
}
```

[http://en.wikipedia.org/wiki/Euclidean_algorithm]

Euclidean Algorithm

Example:

Find the **gcd(270, 192)**: A=270, B=192

A ≠0, B ≠0

Use long division to find that 270/192 = 1 with a remainder of 78. We can write this as:
270 = 192 * 1 +78

Find gcd (192,78), since gcd (270,192)= gcd (192,78): A=192, B=78

A ≠0, B ≠0

Use long division to find that 192/78 = 2 with a remainder of 36. We can write this as:

192 = 78 * 2 + 36

Find gcd (78,36), since gcd (192,78)= gcd (78,36): A=78, B=36

A ≠0, B ≠0

Use long division to find that 78/36 = 2 with a remainder of 6. We can write this as:

78 = 36 * 2 + 6

(next slide) …

Euclidean Algorithm (cont.)

Find gcd(36,6), since gcd(78,36)=gcd(36,6): A=36, B=6

A ≠0, B ≠0

Use long division to find that 36/6 = 6 with a remainder of 0. We can write this as:

36 = 6 * 6 + 0

Find gcd(6,0), since gcd(36,6)=gcd(6,0): A=6, B=0

A ≠0

B =0, gcd(6,0) = 6

So we have shown:

gcd(270,192) = gcd(192,78) = gcd(78,36) = gcd(36,6) = gcd(6,0) = **6**

gcd(270,192) = 6

[https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/the-euclidean-algorithm]

Euclidean Algorithm

Uma maneira simples de calcular o gcd manualmente!

Exemplo: gcd(270, 192) = ?

270 = 192 (1) + 78

192 = 78(2)+36

78 = 36(2) + **6**

36 = **6**(6) + 0

donde **gcd(270, 192) = 6**

Euclidean Algorithm (cont.)

Example: **gcd(2016, 1959) = ?**

2016 = 1959 (x) + y;	x = 2016 mod 1959 = 57;	y = 2016/1959 = 1
2016 = 1959 (1) + 57		
1959 = 57 (x) + y;	x = 1959 mod 57 = 21;	y = 1959/57 = 34
1959 = 57 (34) + 21		
57 = 21 (x) + y;	x = 57 mod 21 = 15;	y = 57/21 = 2
57 = 21 (2) + 15		
21 = 15 (x) + y;	x = 21 mod 15 = 6;	y = 21/15 = 1
21 = 15 (1) + 6		
15 = 6 (x) + y;	x = 15 mod 6 = 3;	y = 15/6 = 2
15 = 6 (2) + **3**		
6 = 3 (x) + y;	x = 6 mod 3 = 0;	y = 6/3 = 2
6 = **3** (2) + 0		

gcd(2016, 1959) = 3

Modular inverse

In modular arithmetic we do not use division operation. However, we do have modular inverses.

The **modular inverse of a mod c is a^{-1}**

$(a * a^{-1}) \equiv 1 \bmod c$ or equivalently **$(a * a^{-1}) \bmod c = 1$**

Only the numbers coprime to c (numbers that share no prime factors with c) have a modular inverse (mod c)

Inverso multiplicative (*modular inverse*)

Inverso multiplicativo

Se x * y ≡ 1 mod n ou x*y mod n = 1, então x e y são inversos multiplicativos mod n um do outro

– Exemplo: **3 × 7 ≡ 1 mod 10** ou **21 = k x 10 + 1 (k=2)**

x, primos relativos de 10

Multiplicação módulo 10

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	0	2	4	6	8
3	0	3	6	9	2	5	8	1	4	7
4	0	4	8	2	6	0	4	8	2	6
5	0	5	0	5	0	5	0	5	0	5
6	0	6	2	8	4	0	6	2	8	4
7	0	7	4	1	8	5	2	9	6	3
8	0	8	6	4	2	0	8	6	4	2
9	0	9	8	7	6	5	4	3	2	1

x tem um inverso multiplicativo módulo n sse x for primo relativo de n

Olhando para a tabela acima indique quais são os primos relativos de n = 10?

How to find a modular inverse

A **naive method** of finding a modular inverse for **a mod c** is:

step 1: Calculate **a * b mod c** for **b** values from **0** through **c-1**

step 2: The modular inverse of ***a* mod *c*** is the ***b*** value that makes

a * b mod c = 1

Note: The term ***b* mod *c*** can only be an integer value from **0** through ***c*-1**, so testing larger values for ***b*** is redundant.

How to find a modular inverse

Example: a=3, c=7

Step 1. Calculate a * b mod c for b values 0 through c-1

3 * 0 ≡ 0 (mod 7)

3 * 1 ≡ 3 (mod 7)

3 * 2 ≡ 6 (mod 7)

3 * 3 ≡ 9 ≡ 2 (mod 7)

3 * 4 ≡ 12 ≡ 5 (mod 7)

3 * 5 ≡ 15 (mod 7) ≡ 1 (mod 7) **<------ FOUND INVERSE!**

3 * 6 ≡ 18 (mod 7) ≡ 4 (mod 7)

Step 2. The modular inverse of *a* mod *c* is the *b* value that makes

***a*b* mod *c* = 1**

5 is the modular inverse of 3 mod 7 since 3*5 mod 7 = 1

How to find a modular inverse

Example: a=2, c=6

Step 1. Calculate a * b mod c for b values 0 through c-1

2 * 0 ≡ 0 (mod 6)

2 * 1 ≡ 2 (mod 6)

2 * 2 ≡ 4 (mod 6)

2 * 3 ≡ 6 ≡ 0 (mod 6)

2 * 4 ≡ 8 ≡ 2 (mod 6)

2 * 5 ≡ 10 ≡ 4 (mod 6)

Step 2. The modular inverse of *a* mod *c* is the b value that makes *a* * *b* mod *c* = 1

In the example above there isn't any. No value of *b* makes *a* * *b* mod *c* = 1. Therefore, ***a* has no modular inverse (mod 6).** This is because **2 is not coprime to 6** (they share the prime factor 2).

Computing multiplicative inverses in Z_n

Algorithm

INPUT: **a mod n** $\in Z_n$.

OUTPUT: $\mathbf{a^{-1}}$ **mod n**, provided that it exists.

1. Use the **extended Euclidean algorithm** to find integers **x** and **y** such that **ax + ny = d**, where **d = gcd(a, n)**. This is known as the Bezout's Identity
2. If **d > 1**, then $\mathbf{a^{-1}}$ **mod n** does not exist. Otherwise, return(**x**).

See slide about **extended Euclidean algorithm**

Prime numbers

Definition: Prime numbers are natural numbers greater than 1 that can only be divided by 1 and themselves.

All other natural numbers that are not prime are **composite numbers**.

By definition, **1 is not a prime number**.

Theorem: There are infinitely many primes.

Números primos

Um dos métodos mais antigos de determinação dos números primos é o "**Sieve of Eratosthenes**":

> Começando em 2 eliminam-se todos os múltiplos até ao valor onde se quer calcular os números primos, depois faz-se o mesmo com o próximo número que ainda não eliminado (3), depois com o próximo (5), e assim sucessivamente até se chegar ao último número restante do conjunto onde se pretende encontrar os primos.

Largest known prime number

As of **January 2016**, the **largest known prime number** is $2^{74,207,281} - 1$, a number with **22,338,618 digits**. It was found in 2016 by the Great Internet Mersenne Prime Search (GIMPS).

The fast Fourier transform implementation of the Lucas–Lehmer primality test for Mersenne numbers is fast compared to other known primality tests for other kinds of numbers.

In mathematics, a **Mersenne prime** is a prime number that is one less than a power of two. That is, it is a prime number that can be written in the form $M_n = 2^n - 1$ for some integer n.

[http://www.mersenne.org/]
[https://en.wikipedia.org/wiki/Largest_known_prime_number]

Composite numbers - Factorization

Definition: Natural numbers greater than 1 that are not prime are called **composite numbers**. These have at least two prime factors other than 1.

Any integer a > 1 can be factored in a unique way as

$$a = {}_{p1}\, a1 *{}_{p2}\, a2 * .... *\ {}_{Pp1} a1$$

Where $p_1 < p_2 < .... < p_t$ are prime numbers and where each a_i is a positive integer

This is known as the **fundamental theorem of arithmetic (Gauss 1801)**

Examples of the decomposition of such composite numbers into prime factors:

4 = 2*2

6 = 2*3

91 = 7*13

767 = 13*59

5324 = 2*11*113

Factorization

"All whole numbers greater than 1 can be expressed as a product of prime numbers in a unique way."

This is the claim of the **1st fundamental theorem of number theory** (= fundamental theorem of arithmetic = fundamental building block of all positive integers). This was formulated precisely for the first time by Carl Friedrich Gauss in his Disquisitiones Arithmeticae (1801).

Theorem (Gauss 1801): Every even natural number greater than 1 can be written as the product of prime numbers. Given two such decompositions $a = p_1*p_2*\ldots*p_n = q_1*q_2*\ldots*q_m$, these can be resorted such that $n = m$ and, for all i, $p_i = q_i$.

Factorização

Teorema fundamental da aritmética (fatorização com números primos)

Todos os números naturais, excetuando 1, podem ser expressos como fatores primos de forma única, exceto a ordem dos fatores:

ou

Qualquer inteiro positivo *n* > 1 pode ser fatorizado de forma única como:

$$n = p_1^{\alpha 1} \times p_2^{\alpha 2} \times \ldots \times p_t^{\alpha t}$$

onde $p_1 > p_2 > \ldots > p_t$ são números primos distintos entre si e onde $\alpha_i >= 1$ são inteiros positivos - **Teorema fundamental da aritmética**.

Por exemplo:

$91 = 7 \times 13$ e $11011 = 7 \times 11^2 \times 13$

Primos relativos (*coprime*)

Números primos relativos (*coprime*)

Se gcd(*a*,*b*) = 1 então *a* e *b* são primos relativos (*coprime*)

Exemplo:

- 15 e 7 são primos relativos
- 17 e 10 são primos relativos
- 15 e 5 não são primos relativos (têm o fator 5 em comum)
- 24 e 5 são primos relativos
- 24 e 10 não são primos relativos (têm o fator 2 em comum)

Algoritmo de Euclides estendido

0 **algoritmo estendido de Euclides** é uma extensão do algoritmo de Euclides para calcular o máximo divisor comum (greatest common divisor (gcd)) dos inteiros **a** e **b** e os inteiros ***x*** e ***y*** da identidade de Bézout (Bézout's identity): **ax+by=gcd(a,b)**

```
function extended_gcd(a, b)
x := 0; lastx := 1;
y := 1; lasty := 0
while b ≠ 0
   temp := b
   quotient := a div b
   b := a mod b
   a := temp
   temp := x; x := lastx-quotient*x; lastx := temp;
   temp := y; y := lasty-quotient*y; lasty := temp
return {lastx, lasty, a} <= x, y e gcd(a, b) da identidade de Bézout
```

Extended Euclid's Algorithm

Example

2016 * x + 1959 * y = gcd (2016, 1959)

First calculate the gcd:

gcd(2016, 1959)

Using Euclidian algorithm:

2016 = 1959 (1) + 57

1959 = 57 (34) + 21

57 = 21 (2) + 15

21 = 15 (1) + 6

15 = 6 (2) + **3** **[gcd]**

6 = 3 (2) + 0

gcd (2016, 1959) = 3

After the gcd, calculate: **2016 * x + 1959 * y = 3**

3 = 15 + 6 (-2)

= 15 + (21 + 15 (-1)) (-2)

= 15 + 21 (-2) + 15 (2)

= 15 (3) + 21 (-2)

= (57 + 21 (-2)) (3) + 21 (-2)

= 57 (3) + 21 (-6) + 21 (-2)

= 57 (3) + 21 (-8)

= 57 (3) + (1959 + 57 (-34)) (-8)

= 57 (3) + 1959 (-8) + 57 (272)

= 57 (275) + 1959 (-8)

= (2016 + 1959 (-1)) (275) + 1959 (-8)

= 2016 (275) + 1959 (-283)

x = 275, y = -283

Neste caso como o gcd não é 1 não existe inverso multiplicativo

3 = 2016 (275) + 1959 (-283)

Totiente de Euler, Φ(n) [phi]

Definição: *Número de inteiros positivos <n que são primos relativos de n.*

- Por definição: **$\phi(1) = 1$**
- Se **n** for primo então **$\phi(n) = n - 1$**
- Se **p** e **q** forem primos então, se ***n = p x q:***

$$\phi(n) = \phi(p \times q) = \phi(p) \times \phi(q) = (p-1)(q-1)$$

- Se $n = p_1^{e1} \times p_2^{e2} \times \ldots \times p_k^{ek}$ for a fatorização de n, com números primos então

$$\phi(n) = n \times (1-1/p_1) \times (1-1/p_2) \times \ldots \times (1-1/p_k)$$

Definition: The Euler function $\phi(n)$ specifies the number of elements in Z^*_n . $\phi(n)$ also species **how many whole numbers have multiplicative inverses in *mod n*.**

Aritmética modular

Example

$c = 7^{27} \bmod 30$ $= 7^{(27 \bmod \phi(30))} \bmod 30$

[Lembrar que: $x^y \bmod n = x^{(y \bmod \phi(n))} \bmod n$]

The $\phi(30) = 8$, so this reduces to

$c = 7^{27 \bmod 8} \bmod 30 = 7^3 \bmod 30$ [27 mod 8 = 3]

$= (7*7*7) \bmod 30 = ((49 \bmod 30)(7 \bmod 30)) \bmod 30$

$= ((19)(7)) \bmod 30$

$= (133) \bmod 30$

$= 13$

Fast calculation of high powers

RSA encryption and decryption entails calculating high powers modulo m. For example, the calculation (100^5) (mod 3) exceeds the 32-bit long integer number range provided we calculate a^n by actually multiplying a with itself n times in line with the definition. In the case of extremely large numbers, even a fast computer chip would take longer than the age of the universe to calculate a single exponential. Luckily, there is an extremely efective shortcut for calculating exponentials (but not for calculating logarithms).

If the expression is divided differently using the rules of modular arithmetic, then the calculation does not even exceed the 16-bit short integer number range:

$$(a^5) \equiv (((a^2 \bmod m)^2 \bmod m)\ a) \bmod m$$

We can generalize this by representing the exponent as a binary number.

For example, the naive method would require 36 multiplications in order to calculate an for n = 37. However, if we write n in the binary representation as $100101 = 1 * 2^5 + 1 * 2^2 + 1 * 2^0$, then we can rewrite the expression as:

$$a^{37} = a^{2^5+2^2+2^0} = a^{2^5} * a^{2^2} * a^{2^0} = a^{32} * a^4 * a^1$$

[**Note:** In binary to do **x** to the power of the n^{th} power of 2 (x^{2^n}) is only shift **x n** times to the left]

Aritmética modular

Example

Find the multiplicative inverse of **1579 modulo 7351** or the ***a*** that ***a* * 1579 mod 7351 = 1**.

According to **Euler-Fermat: $a^{\phi(n)} = 1 \bmod n$ for all a in Z^*_n**. If we divide both sides by ***a***, we get: $a^{\phi(n)-1} \equiv a^{-1} \bmod n$. For the special case that the modulo is prime, we have $\phi(n) = p - 1$.

Therefore, if **n = p** and **p** is prime, the modular inverse is $a^{-1} = a^{\phi(n)-1} = a^{(p-1)-1} = a^{p-2} \pmod p$

For our example, this means:

Since the modulus 7351 is prime and p-2 = 7349:

$$1579^{-1} = 1579^{7349} \bmod 7351$$

By cleverly breaking down the exponent, we can calculate this power relatively easily:

$$7349 = 4096 + 2048 + 1024 + 128 + 32 + 16 + 4 + 1$$

$$1579^{7349} \bmod 7351 = 1579^{(4096 + 2048 + 1024 + 128 + 32 + 16 + 4 + 1)} \bmod 7351 = \mathbf{4716}$$

$$1579^{-1} = \mathbf{4716} \bmod 7351$$

Chinese Remainder Theorem

Theorem: Let $m_1,\ldots,m_n$ be pairwise coprime (that is $\gcd(m_i, m_j) = 1$ whenever $i \neq j$). Then the system of n equations

$x \equiv a_1 \pmod{m_1}$

...

$x \equiv a_n \pmod{m_n}$

has a unique solution modulo M where $\mathbf{M = m_1 x \ldots x m_n}$ for $1 \leq i \leq n$, which is given by: $\mathbf{x \equiv a_1 M_1 y_1 + a_2 M_2 y_2 + \ldots + a_n M_n y_n \pmod{M}}$ [or $x \equiv \sum_{i=1}^{n} a_i M_i y_i \pmod{M}$], where $\mathbf{M_i = M/m_i}$ and $\mathbf{y_i \equiv (M_i)^{-1} \pmod{m_i}}$ for $1 \leq i \leq n$ is the unique solution.

Chinese Remainder Theorem

Example 1

Find **x** so that:

x = 2 mod **5**

x = 3 mod **7**

p = 5, q = 7, p * q = **35**

x ≡ 17 mod 35

Example 2

Find **x** so that:

x = 2 mod **3**

x = 2 mod **4**

x = 1 mod **5**

p = 3, q = 4, r = 5, p*q*r = **60**

x ≡ 26 mod 60

Complexity

Certain problems are computationally hard . . .

Integer Factorisation

- If p and q are unknown primes, given n = p*q, find p and q
- Largest RSA number factored into two primes is 768 bits (232 decimal digits)

Euler's Totient

- Given composite n, find $\phi(n)$
- Harder than integer factorisation

Discrete Logarithms

- Given b, a and p, find i such that $i = d*\log_{a;p}(b)$
- Comparable to integer factorisation

State of the art (factorization, April/2020)

"Among the *b*-bit numbers, the most difficult to factor in practice using existing algorithms are those that are products of two primes of similar size. For this reason, these are the integers used in cryptographic applications. The largest such semiprime yet factored was RSA-250, a **829-bit number with 250 decimal digits, in February 2020.** The total computation time was roughly **2700 core-years of computing using Intel Xeon Gold 6130 at 2.1 GHz. Like all recent factorization records, this factorization was completed with a highly optimized implementation of the general number field sieve run on hundreds of machines."** [Wikipedia]

Diffie-Hellman - Troca de chaves

O esquema foi publicado inicialmente por Whitfield Diffie e Martin Hellman em 1976. Descrevia um método prático que permitia a partilha de segredos através de um canal de comunicações inseguro.

Clifford Cocks, um matemático britânico, a trabalhar para os serviços secretos britânicos descreveu um sistema semelhante num documento interno em 1973, mas a sua proposta foi considerada apenas um curiosidade. Só foi conhecida em 1997 dada a sua classificação como secreta.

Foi a partir de artigos como o de Diffie-Hellman que foi desenvolvido um novo tipo de criptografia designada por criptografia de chave assimétrica.

O algoritmo RSA foi descrito publicamente em 1978 por Ron Rivest, Adi Shamir, e Leonard Adleman no MIT;

Troca de chaves - Diffie-Hellman (1)

- Alice e Bob acordam num número primo elevado, ***n***, e num inteiro ***g***, (onde ***g é uma raiz primitiva de n)***. A razão de **g** ter de ser uma raiz primitiva de **n**, tem a ver com o facto de assim os valores de **X** não ficarem limitados pois $\mathbf{g^x \bmod n}$ pode gerar qualquer valor entre **1** e **n-1** (definição de raiz primitiva) e não um subconjunto dos valores possíveis.
- Os valores ***g*** e ***n*** não necessitam ser mantidos secretos.

- Alice escolhe um número aleatório grande ***x*** e envia **X** a Bob:

$$X = g^x \bmod n$$

- Bob escolhe um número aleatório grande ***y*** e envia **Y** a Alice:

$$Y = g^y \bmod n$$

- ***Nota: x** e **y** nunca são transmitidos*

Troca de chaves - Diffie-Hellman

- Alice calcula:

$$k = Y^x \bmod n = (g^y \bmod n)^x \bmod n = g^{xy} \bmod n$$

- Bob calcula:

$$k' = X^y \bmod n = (g^x \bmod n)^y \bmod n = g^{xy} \bmod n$$

mas:

$$k = k' = g^{xy} \bmod n$$

- *Assim, Bob e Alice têm agora a chave secreta, **k**, que podem partilhar para cifrar as comunicações*

- *Os bisbilhoteiros sabem apenas n, g, X e Y que são necessários para calcular k mas não suficientes*

Segurança Diffie-Hellman

- A segurança D-H depende da dificuldade de fatorizar grandes números (dimensão do n)
- É computacionalmente muito díficil (não realizável) recuperar *x* e *y* a partir dos dados conhecidos de um bisbilhoteiro por qualquer forma que não seja a pesquiza exaustiva da chave
- Um Diffie-Hellman um pouco mais seguro: X.1035

  http://www.itu.int/rec/T-REC-X.1035-200702-I/en

Obrigações:

- ***n*** **deve ser um número primo muito grande**
- ***((n-1)/2)*** **deve também ser primo**
- ***g*** **pode ser pequeno – mesmo um algarismo**

Desvantagens do Diffie-Hellman

- **Lento!**
  - Computacionalmente intensivo
  - Requer várias trocas de comunicações

- Exemplo:
  - Usar o D-H para estabelecer uma chave de sessão num telefone celular pode chegar quase a um minuto!

- Resultado: Foram estabelecidos outros protocolos de troca de chaves que são mais eficientes

Objectivo inicial

- Diffie e Hellman não pretenderam criar outro tipo de criptografia

  O objectivo era achar uma forma de estabelecer chaves de sessão simétricas sem a necessidade de colocação das chaves por outro meio

  – Isto é, para resolver o problema da **distribuição das chaves**

  Esta é **ainda** hoje a utilização principal da troca D-H

Mas ...

- A troca de chaves de Diffie-Hellman mostrou-se muito útil

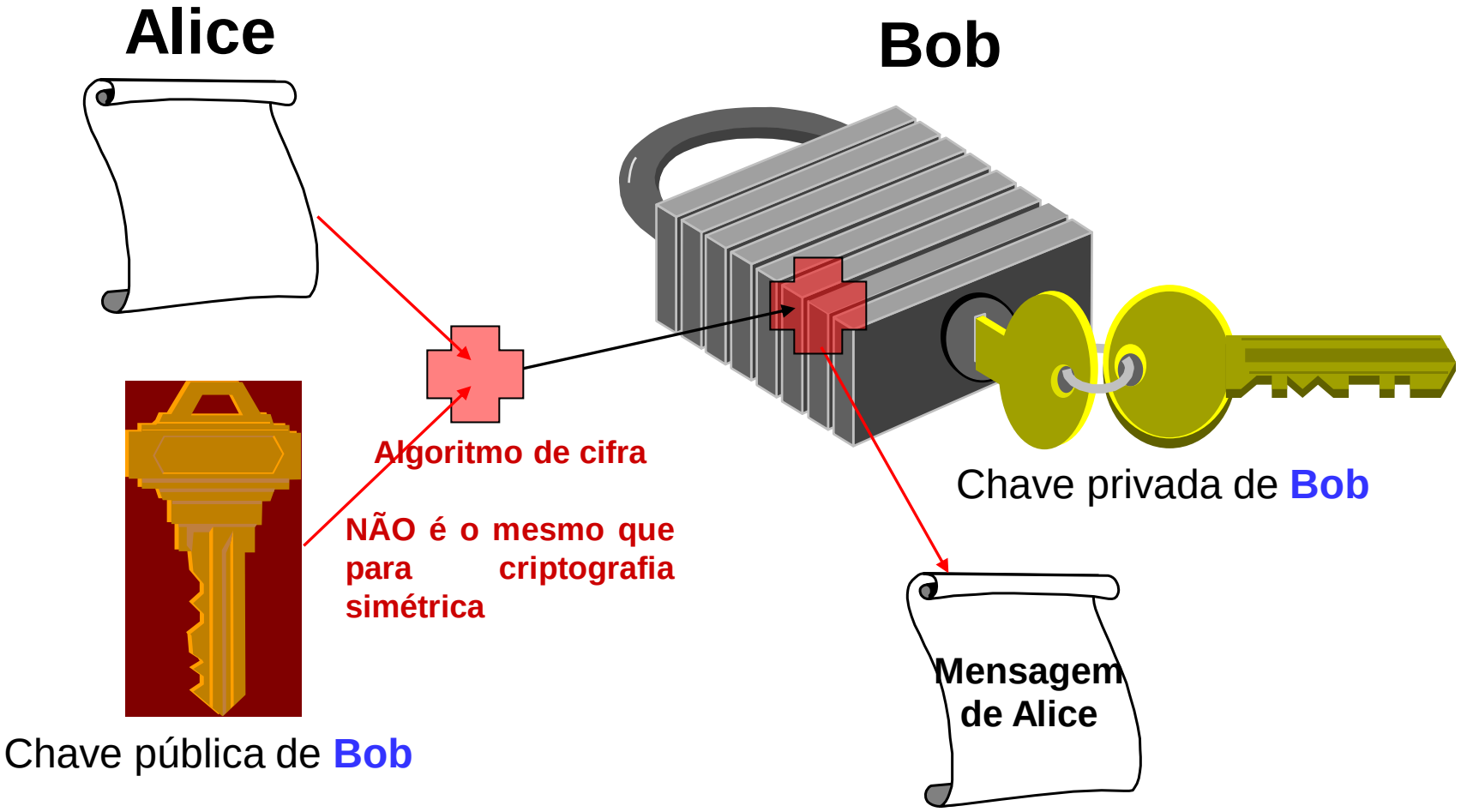
- Outros descobriram que havia outras utilizações para este princípio geral da criptografia e, assim, foram desenvolvidos vários algoritmos para cifrar dados:

    - RSA
    - El Gamal
    - etc.

Criptografia assimétrica

- Também conhecida por criptografia de chave pública
- Até Diffie-Hellman em 1976, este tema era tabú. Ainda hoje não é muito intuitivo.
  - A NSA e os Serviços Secretos Britânicos dizem ter descoberto o princípio anos antes mas mantiveram o segredo
- A chave tem duas partes
  - Pública: Todos a sabem ou podem saber
  - Privada: Apenas conhecida duma entidade ← ***critico!!!***
- Baseado em grandes números primos

Criptografia assimétrica



Alguma coisa diferente

- De forma clara, a criptografia assimétrica difere de uma forma básica da criptogradia simétrica
  - As chaves são relacionadas matematicamente e não podem ser apenas números aleatórios puros
  - Os algoritmos são bastante diferentes dos utilizados no universo das cifras simétricas de Feistel e das *S-boxes*

- É uma substituição da cifra simétrica ou um complemento?

Propriedades da criptografia assimétrica

- A função de encriptação é de sentido único (*one-way*)

- O processo de encriptação permite inverter o uso das chaves
  - Pode cifrar com a chave pública e decifrar com a privada ou o contrário

- E então?
  - Pode esta aproximação ser utilizado para assinar documentos?
  - Um documento assinado pode ser utilizado para autenticação?

E ...

- **A criptografia assimétrica é baseada em aritmética modular**
- A aritmética modular torna computacionalmente irrealizável a recuperação do número cujo módulo é conhecido, desde que certas condições sejam garantidas:

  – 0<M<n [n=pq; p e q são números primos grandes]

  – Sendo M a mensagem, M e n devem ser primos relativos [a probabilidade disto não acontecer é aproximadamente por 1/p+1/q. Dado os elevados valores p e q isto acontecer é muito pouco provável. Caso necessário pode ser realizado algum *padding* sobre M.]

Fraqueza criptográfica

- Todos os sistemas de criptografia dependem da segurança da chave para a sua segurança
  - Se for um sistema simétrico, quem tiver a chave lê o conteúdo
  - Os sistemas assimétricos baseiam-se no facto da chave privada se manter secreta.

- Ataques ao pressuposto funcionam melhor que ataques à porta da frente

Utilização da criptografia assimétrica

- Útil apenas para a troca de chaves?

- Até agora provou ser mais útil do que isso
  - Pode ser usada para fornecer serviços semelhantes aos da criptografia simétrica, por exemplo:
    - Confidencialidade
    - Integridade
    - Autenticidade

- Tal como todos os sistemas de chaves assimétricas, o RSA depende, para garantir a segurança, da dificuldade em factorizar grandes números.

Algoritmo de encriptação RSA

- Ron **R**ivest, Adi **S**hamir, Len **A**delman
  - Publicado pela primeira vez em 1978, no MIT
  - Cifra de bloco, chave assimétrica
  - Textos em claro (M) e cifrados (C) são tratados como valores inteiros entre 0 e n-1, para qualquer n que for parte das chaves (n = p x q)
  - Dimensão da chave: variável (superior a 512 bit, normalmente 1024 ou 2048 bit)
  - Entrada: Variável (< dimensão da chave)
  - Saída: Dimensão da chave
  - Vantagem: Facilidade de gestão da chave
  - Desvantagem: Muito mais lento que a criptografia simétrica

Algoritmo de encriptação RSA

Ver [http://certauth.epfl.ch/rsa/rsa.pdf]

- Escolher dois números primos muito grandes, **p** e **q**, fazer **n = p x q** donde

$$\phi(n) = \phi(p.q) = \phi(p).\phi(q)=(p-1).(q-1)$$

- Escolher ***e*** de maneira a $1 < e < \phi(n)$ e a ***e*** ser primo relativo de $\phi(n)$ [**gcd(ϕ(n),e)=1**; se se escolher um número primo para ***e*** apenas se terá de verificar se ele não é um factor de ***n***]
- Encontrar o valor **d** inverso multiplicativo de **e** módulo **ϕ(n)**, i.e.,

$$e.d \equiv 1 \pmod{\phi(n)} = 1+k.(p-1).(q-1)$$

[resolve-se usando o **algoritmo estendido de Euclides** ou o **teorema de Euler**]

- Se <e, n> forem a chave pública, <d, n> serão a chave privada que deve ser protegida.

Mecânica do RSA

- **M** = texto em claro (que se pretende enviar cifrado)
- **C** = texto cifrado

Cifra: $C = M^e \bmod n$

Decifra: $M = C^d \bmod n$

(Nota: Considere que todas as equações têm "(mod n)" implicito**)**

$$C^d = (M^e \bmod n)^d = (M^e)^d = M^{ed} = M^{1.\bmod \phi(n)}$$
$$= M^{1+k.\,\phi(n)} = M^{1+k.\,(p-1)(q-1)} = M.M^{k.\,(p-1)(q-1)} = M.(M^{(p-1)(q-1)})^k$$
$$= M.(1 \bmod (p.q))^k = M.1^k = M$$

Nota: Se **y ≡ 1 (mod ϕ (n))**, então, para qualquer número **x:** $\mathbf{x^y \bmod n = x \bmod n}$

e

$\mathbf{x^{(p-1)(q-1)} \equiv 1 (mod\ p.q)}$ para quaisquer números primos p e q e qualquer inteiro x que não tenha factores comuns com p*q [Teorema fundamental do RSA].

E...

- **Chave pública: $K_U = \{e,n\}$**
- **Chave privada: $K_R = \{d,n\}$**
- Requisitos para isto funcionar:
  - ***e***, ***d***, ***n*** têm valores tais que $M^{ed} = M \bmod n$ para todo o $M<n$
  - Fácil de calcular M^e e C^d *para* $M<n$
  - Impossível calcular ***d*** dado ***e*** e ***n***

Computacionalmente seguro se ***e, n*** forem suficientemente grandes

Key generation for RSA public-key encryption

Algorithm

SUMMARY: Each entity creates an RSA public key and a corresponding private key.

Each entity **A** should do the following:

1. Generate two large random (and distinct) **primes p and q**, each roughly the same size.
2. Compute **n = pq** and **Φ(n) = (p − 1)(q − 1)**.
3. Select a **random integer *e*, 1 < *e* < Φ(n)**, such that **gcd(*e*, Φ(n)) = 1**.
4. Use the **extended Euclidean algorithm** to compute the unique integer ***d***, 1 < *d* < Φ(n), such that ***ed* ≡ 1 (mod Φ(n))**.
5. **A**'s **public key** is **(*n*; *e*)**; **A**'s **private** key is ***d***.

Importância do RSA

- O RSA é o algoritmo de cifra assimétrica mais utilizado
- O RSA foi patenteado pelos seus inventores, mas a patente expirou em 2000
- O RSA é agora de utilização livre e foi incorporado em inúmeros produtos comuns, tal como *browsers Web*, VPN, etc.

Exemplo do RSA

- Selecionar dois primos: ***p = 7, q = 17***
- Calcular ***n = p x q = 7 x 17 = 119***
- Calcular ***Φ(n) = (7-1)(17-1) = 6 x 16 = 96***
- Selecionar ***e*** primo relativo de ***Φ(n)***, isto é [gcd(e, 96)=1]
  - Neste exemplo, ***e = 5***
- Calcular ***d*** *onde de* = 1 (mod $\Phi(n)$) -> 5d ≡ 1(mod 96) -> $d = e^{-1}$ (mod $\Phi(n)$) = 5^{-1} mod 96 -> ***d = 77*** **Ver próximos *slides***

Chave pública **Chave privada**

- **$K_U = \{5, 119\}$** **$K_R = \{77, 119\}$**

Outra perspectiva

$d \equiv e^{-1} \pmod{\Phi(n)}$ parece complicado dado $e^{-1} < 1$

Multiplicar ambos os lados por **e**, o que dá $de = 1 \pmod{\Phi(n)}$, onde neste caso $\Phi(n) = 96$, logo

$$de \equiv 1 \pmod{\Phi(n)} = 1 \pmod{96}$$

e foi seleccionado como sendo *5*, então devemos encontrar o valor para **d** que satisfaça a equação acima

$$d \times 5 \equiv 1 \pmod{96} \Rightarrow (d \times 5) \bmod 96 = 1 \Rightarrow d = 77$$

77 é o valor de **d**: $5 \times 77 \equiv \underline{1} \pmod{96}$ ou **77 x 5 mod 96 = 385 mod 96 => 4 x 96 + <u>1</u>, o resto é <u>1</u>**

Outra perspectiva

Usando o algoritmo de Euclides estendido o qual permite calcular **x**, **y** e **gcd(a, b)** em:

$$a.x + b.y = gcd\ (a, b)$$

onde **a** e **b** são dados e, no caso de **a** e **b** serem inverso multiplicativo, **gcd(a, b) = 1** passa a ser: **a.x + b.y = 1**

Se se pensar em $e.d \equiv 1 \pmod{\Phi(n)}$ pela definição de congruência $\Phi(n) | (e.d-1)$, o que significa que $e.d-1 = k.\Phi(n)$ com **e** e $\Phi(n)$ dados, **d** o inverso multiplicativo e **k** um inteiro múltiplo que será descartado.

Esta é a forma da função que o algoritmo de Euclides estendido resolve, com a diferença que **gcd(e, Φ(n)) = 1** é predeterminado em vez de ser calculado. Daqui a necessidade de ***e*** ser primo relativo (*coprime*) do módulo **Φ(n)** ou o inverso multiplicativo não existirá. Usando os nomes habituais das variáveis do RSA e o algoritmo de Euclides estendido:

$$e.d + \Phi(n).y = gcd\ (e, \Phi(n)) \quad \text{ou} \quad e.d + \Phi(n).y = 1$$

substituindo pelos valores anteriores e aplicando o **algoritmo estendido de Euclides** vem:

5.d+96.y=1 vem **d=77**

Outra perspectiva

e.d+ Φ(n).y=gcd (e, Φ(n)) **=>** substituindo e aplicando o **algoritmo estendido de Euclides** a **5.d+96.y=1 => d=77**

gcd (5, 96) = ?

96 = 5 (x) + y

96 = 5 (19) + **1**

5 = 1 (5) + 0

Euclides estendido:

1 = e.d + Φ(n).y

1 = 5 (-19) + 96

1 = 5 **(-19)** + 96 (**1**) => d = -19 e y = 1 mas dado que 1 < d < 96 [Φ(n)] => d = – 19 + 96 = **77**

Como *d* deve obedecer a 1 < *d* < Φ(n) e *d* = -19 e Φ(n) = 96 a condição de ser maior que 1 não se verifica. Podemos somar 96 a -19 obtendo o valor 77 o qual já verifica a condição anterior, ficando *d* = 77.

Encriptação/desencriptação RSA

Exemplo:

Usando K_U, K_R que calculado antes, seja ***M = 19*** *(plaintext)*

Public key: e, n — Private key: d, n

$K_U = \{\mathbf{5}, 119\}$ $K_R = \{\mathbf{77}, 119\}$

– Encriptação de M=19:

$M^e \bmod n = 19^5 \bmod 119 = 66 = C$ *(ciphertext)*

– Desencriptação

$C^d \bmod n = 66^{77} \bmod 119 = 19 = M$ *(plaintext)*

Calcular a chave privada (exemplo)

Assumindo p = 37 e q = 89, n = p x q = 3293 e Φ(n) = (p - 1)(q - 1) = (37 – 1) (89 – 1) = 36 x 88 = 3168

Escolhendo e = 25 é necessário calcular d.

e.d+ Φ(n).y=gcd (e, Φ(n)) => e.d+ Φ(n).y=gcd (e, Φ(n)) => substituindo e aplicando o algoritmo estendido de Euclides a **25.d+3168.y=1 => d=2281**

gcd (25, 3168) = 1

3168 = 25 (x) + y

3168 = 25 (126) + 18 → 3168 + 25(-126) = 18

25 = 18 (1) + 7

18 = 7 (2) + 4 → 18 + 7(-2) = 4

7 = 4 (1) + 3 → 7 + 4(-1) = 3

4 = 3 (1) + **1**

3 = 1 (3) + 0

Euclides estendido:

1 = 4 + 3(-1) = 4 + (7 + 4(-1))(-1) = 4 + 7(-1) + 4 (1) = 4 (2) + 7 (-1)

= (18 + 7 (-2))(2) + 7 (-1) = 18 (2) + 7 (-5) = 18 (2) + (25 + 18 (-1)) (-5) = 18 (7) + 25 (-5)

= (3168 + 25 (-126))(7) + 25 (-5) = **3168 (7) + 25 (-887) => d = 3168 + (-887) = 2281**

Como d deve obedecer a 0 ≤ d < Φ(n) e d = -887 e Φ(n) = 3168 a condição de d ser maior que 0 não se verifica. Podemos somar 3168 a -887 obtendo o valor 2281 o qual já verifica a condição anterior, ficando d = 2281.

RSA Explained (again)

Let's say that your WEB Browser has a piece of data, say number **14** (we'll call it a Plain message and label it as **M=14**) and it wants to encrypt this Plain message first and then send it to the Server. Upon receipt of this encrypted message, the Server wants to decrypt it to its original value.

Before any communication happens, the Server had calculated, in advance, its public (**n =33** and $\mathbf{k_u}$**=7**) and private ($\mathbf{K_r}$**=3**) keys. Now, to initiate the transaction, the Browser sends this message to the server: Hey Server, please send me your public key. The Server obliges: Here it comes, it's **n=33,** $\mathbf{k_u}$**=7**. After receiving the Server's public key, the Browser converts the Plain message **M=14** into the Encrypted message **C=20** and sends it to the Server. The Server receives this encrypted message **C=20** and using its secret key $\mathbf{K_r}$**=3** (and publicly known key **n=33**) decrypts the **C=20** message into its original Plain message **M=14**.

RSA Explained (again)

Generating Public and Private Keys

First, as we mentioned above, before any transmission happens, the Server had calculated its public and secret keys. Here is how:

1. Pick two prime numbers, we'll pick **p = 3** and **q** = **11**
2. Calculate **n** = p * q = 3 * 11 = **33**
3. Calculate ***Φ(n)*** = (p - 1) * (q - 1) = (3 - 1) * (11 - 1) = **20**
4. Choose a prime number k_r, such that k_r is co-prime to ***Φ(n)***, i.e, ***Φ(n)*** is not divisible by k_r. We have several choices for k_u: 7, 11, 13, 17, 19 (we cannot use 5, because 20 is divisible by 5). Let's pick $\mathbf{k_u=7}$ (smaller k_u, "less math").
5. So, the numbers **n = 33** and $\mathbf{k_u = 7}$ become the Server's public key.
6. Now, still done in advance of any transmission, the Server has to calculate it's secret key. Here is how: $\mathbf{k_u * k_r = 1 \bmod \Phi(n)}$ **-> 7 *** $\mathbf{k_r}$ **= 1 mod 20 -> (7 *** $\mathbf{k_r}$ **) / 20 = ?**

note the remainder of 1 (the "?" here means: "something, but don't worry about it"; we are only interested in the remainder). Since we selected (on purpose) to work with small numbers, we can easily conclude that 21 / 20 gives "something" with the remainder of 1. So, **7 *** $\mathbf{k_r}$ **= 21**, and $\mathbf{k_r = 3}$. [**Note:** 21 mod 20 = 1] $\mathbf{k_r}$ is is our secret key. We MUST NOT give this key away.

RSA Explained (again)

Now, after the Server has done the above preparatory calculations in advance, we can begin our message transmission from our Browser to the Server. First, the Browser requests from the Server, the Server's public key, which the Server obliges, i.e., it sends **n=33** and **k_u=7** back to the Browser. Now, we said that the Browser has a Plain message **M=14**, and it wants to encrypt it, before sending it to the Server. Here is how the encryption happens on the Browser.

Encrypting the message

Here is the encryption math that Browser executes.

$$C = M^{k_u} \bmod n$$

"^" means "to the power of"; **M** is the Plain message we want to encrypt; **n** and **k_u** are Server's public key

C is our Encrypted message we want to generate

After plugging in the values, this equation is solved as follows:

$$C = 14^7 \bmod 33$$

This equation in English says: raise 14 to the power of 7, divide this by 33, giving the remainder of E.

105413504 / 33 = 3194348.606 (well, I lied when I said that this is "Pencil and Paper" method only. You might want to use a calculator here). **3194348 * 33 = 10541348**

C = 105413504 - 10541348 = 20

So, our Encrypted message is **C=20**. This is now the value that the Browser is going to send to the Server. When the Server receives this message, it then proceeds to Decrypt it, as follows.

RSA Explained (again)

Decrypting the Message

Here is the decryption math the Server executes to recover the original Plain text message which the Browser started with.

$$C = M^{k_u} \bmod n$$

C is the Encrypted message just received, k_r is the Server's secret key

M is the Plain message we are trying to recover

n is Server's public key (well part of; remember that Server's public key was calculated in Section 1 as consisting of two numbers: **n=33** and k_u**=7**).

After plugging in the values:

$$M = C^{K_r} \bmod n = 20^3 \bmod 33 = 8000 \bmod 33$$

8000 / 33 = ? with the remainder of **M**. So to calculate this remainder, we do:

8000 / 33 = 242.424242... , 242 * 33 = 7986 , M = 8000 - 7986 = 14,

which is exactly the **Plain text message** that the Browser started with!

Well that's about it. While we did not discuss the theory behind the formulae involved I hope that you got at least a basic idea of how the public key cryptography using the RSA algorithm works.

Exemplo com RSA

Assumindo o algoritmo de chave pública RSA com **p=3** e **q=11** (números primos), vem **n = 33 [3 x 11]** e **$\phi(n) = (p-1)x(q-1) = 20$**. "Escolhendo" **$k_u = e = 3$** e **$k_r = d = 7$** vem [face ao exemplo anterior este tem o valor das chaves escolhido ao contrário] :

Cálculo no emissor				**Cálculo no receptor**		
Plaintext (M)			**Ciphertext (C)**		**M - Após decifrar**	
Simbólico	**Numérico**	**M^3**	**M^3 (mod 33)**	**C^7**	**C^7 (mod 33)**	**Simbólico**
S	19	6859	28	13492928512	19	**S**
U	21	9261	21	1801088541	21	**U**
Z	26	17576	20	1280000000	26	**Z**
A	1	1	1	1	1	**A**
N	14	2744	5	78125	14	**N**
N	14	2744	5	78125	14	**N**
E	5	125	26	8031810176	5	**E**

Quebrar o RSA

- Descobrir a chave privada *d*
  - Fácil de fazer se *p* e *q*, factores de *n*, forem conhecidos
  - A parte difícil é factorizar *n*
  - **Factorizar *n* com 129 dígitos já foi feito**
- Encontrar as raízes de e^{th} mod *n*
  - Não se conhece nenhum método
- Procura da chave através de métodos da força bruta

Segurança do RSA na prática

Escolher *n* suficientemente grande

- **Com 129 digitos ≈ 428 bits já foi factorizado**
- Mais próximo dos 512 bits para descanso da consciência
- Melhor: Escolher *n* > *1000* bits (1024, 2048, 4096)
  - Nem todas as implementações permitem chaves > 2048
- Verificar qual a segurança necessária, chaves maiores implicam tempos maiores de computação e, por isso, são mais lentas a encriptar e desencriptar

Guardar a chave privada cuidadosamente!

Segurança do RSA na prática

Má utilização do RSA

Alice usa a chave pública de Bob para cifrar uma mensagem para enviar a Bob.

Se Frank conhecer a chave pública de Bob e souber que a mensagem cifrada é **uma de várias mensagens possíveis**, ele pode usar a chave pública para cifrar várias mensagens e comparar as mensagens cifradas por ele com a mensagem cifrada por Alice.

Eficiência do cálculo

- Encontrar números primos **p** e **q** grandes
- Encontrar **d** e **e**
- Exponenciar grandes números de várias centenas de bits

How can we calculate "a^b mod c" quickly, if b is a power of 2 ?

Using modular multiplication rules:

$a^2 \bmod c = (a * a) \bmod c = ((a \bmod c) * (a \bmod c)) \bmod c$

We can use this to calculate $7^{256} \bmod 13$ quickly:

$7^1 \bmod 13 = 7$

$7^2 \bmod 13 = (7^1 * 7^1) \bmod 13 = (7^1 \bmod 13 * 7^1 \bmod 13) \bmod 13 = (7 * 7) \bmod 13 = 49 \bmod 13 = 10$

$7^4 \bmod 13 = (7^2 * 7^2) \bmod 13 = (7^2 \bmod 13 * 7^2 \bmod 13) \bmod 13 = (10 * 10) \bmod 13 = 100 \bmod 13 = 9$

$7^8 \bmod 13 = (7^4 * 7^4) \bmod 13 = (7^4 \bmod 13 * 7^4 \bmod 13) \bmod 13 = (9 * 9) \bmod 13 = 81 \bmod 13 = 3$

We continue in this manner, substituting previous results into our equations.

...after 5 iterations we hit:

$7^{256} \bmod 13 = (7^{128} * 7^{128}) \bmod 13 = (7^{128} \bmod 13 * 7^{128} \bmod 13) \bmod 13 = (3 * 3) \bmod 13 = 9 \bmod 13 = 9$

This has given us a method to calculate $a^b \bmod c$ quickly provided that **b is a power of 2**.

How can we calculate $a^b \bmod C$ quickly for any b?

How can we calculate "a^b mod c" quickly for any B ?

a^b mod c

Step 1: Divide *b* into powers of 2 by writing it in **binary**

Step 2: Calculate "mod c" of the powers of two ≤ *b*

Step 3: Use modular multiplication properties to combine the calculated "mod c" values

How can we calculate "a^b mod c" quickly for any b ?

5^{117} mod 19 = ?

Step 1: Divide *b* into powers of 2 by writing it in binary

$b = 117 = 1110101_2 = 1*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 64+32+16+0+4+0+1$

5^{117} mod 19 = $5^{(64+32+16+4+1)}$ mod 19 = ($5^{64} * 5^{32} * 5^{16} * 5^4 * 5^1$) mod 19

Step 2: Calculate mod C of the powers of two ≤ B

5^1 mod 19 = **5**

5^2 mod 19 = ($5^1 * 5^1$) mod 19 = (**5^1 mod 19** * **5^1 mod 19**) mod 19 = (**5 * 5**) mod 19 = **25** mod 19 **= 6**

5^4 mod 19 = ($5^2 * 5^2$) mod 19 = (**5^2 mod 19 * 5^2 mod 19**) mod 19 = (**6 * 6**) mod 19 = **36** mod 19 **= 17**

5^8 mod 19 = ($5^4 * 5^4$) mod 19 = (**5^4 mod 19 * 5^4 mod 19**) mod 19 = (**17 * 17**) mod 19 = **289** mod 19 **= 4**

5^{16} mod 19 = ($5^8 * 5^8$) mod 19 = (**5^8 mod 19 * 5^8 mod 19**) mod 19 = (**4 * 4**) mod 19 = **16** mod 19 **= 16**

5^{32} mod 19 = ($5^{16} * 5^{16}$) mod 19 = (**5^{16} mod 19 * 5^{16} mod 19**) mod 19 = (**16 * 16**) mod 19 = **256** mod 19 = **9**

5^{64} mod 19 = ($5^{32} * 5^{32}$) mod 19 = (**5^{32} mod 19 * 5^{32} mod 19**) mod 19 = (**9 * 9**) mod 19 = **81** mod 19 **= 5**

Step 3: Use modular multiplication properties to combine the calculated mod C values

5^{117} mod 19 = ($5^1 * 5^4 * 5^{16} * 5^{32} * 5^{64}$) mod 19 =

(**5^1 mod 19 * 5^4 mod 19 * 5^{16} mod 19 * 5^{32} mod 19 * 5^{64} mod 19**) mod 19 = (**5 * 17 * 16 * 9 * 5**) mod 19 =

61200 mod 19 = **1**

5^{117} mod 19 = 1

Exponenciação de grandes números

- **Exemplo**
  - $123^{54} \bmod 678$
  - $54 = 110110_2$
  - $123^{54} = ((((123)^2 123)^2)^2 123)^2 123)^2$
- Outro exemplo:

  $3^{54} = 3^{(32+16+4+2)} = (3^{32} x 3^{16} x 3^4 x 3^2) = (((((3)^2 x 3)^2)^2 x 3)^2 x 3)^2$
- **Complexidade $O(\log_2 x)$**

  O número de multiplicações e de divisões sobe linearmente com o comprimento do expoente em bits (não com o valor do expoente)

Descobrir números primos grandes

- A probabilidade de um número **n** escolhido aleatoriamente ser um número primo é **1/ln n**
  - Para um número de 100 dígitos a probabilidade é de **1 em 230**

- Teste se um determinado número **n** é primo
  - Teorema de Fermat: Se **p** for primo e **0<a<p**, então $\mathbf{a^{p-1} = 1 \bmod p}$
  - Para um número **n** não primo de 100 bits, a probabilidade de $\mathbf{a^{n-1} = 1 \bmod n}$ é de cerca de **1 em 10^{13}**

Descobrir números primos grandes

Mais informação:

- http://members.cox.net/mathmistakes/primes.htm
- http://www.ams.org/notices/200305/fea-bornemann.pdf

Encontrar e e d

- **e**: Chave pública, pode ser escolhida qualquer de entre os primos relativos de **ϕ(n)**

- **d**: Chave privada, é calculada com a ajuda do algoritmo de Euclides,

$$e \times d = 1 \bmod \phi(n)$$

- Um **e** pequeno torna as operaçõs com a chave pública (por exemplo, encriptação, verificação da assinatura) mais rápidas, enquanto as operações com a chave privada (por exemplo, desencriptação, realização da assinatura) não se alteram

- **d** não deve ser pequeno

Valores comuns para e

- 3 e 65537 [$2^{16} + 1$]

- Vantagem: Cálculo eficiente
  - 3: duas multiplicações
  - **65537**: dezassete multiplicações [Nota: Chave bastante usada]

Problemas de e = 3

#1:

- Se $\mathbf{M < n^{1/3}}$, então $\mathbf{M = C^{1/3}}$
- **Solução**: Tornar **M** (*pad*) maior que $\mathbf{n^{1/3}}$

#2:

- Se uma mensagem M for encriptada com três chaves públicas (**<e, n>**) $\mathbf{<3,n_1>}$, $\mathbf{<3, n_2>}$, $\mathbf{<3, n_3>}$ pelo teorema chinês do resto pode-se calcular $\mathbf{C = M^3 \bmod n_1\ n_2\ n_3}$ a partir de $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$. Dado que $\mathbf{M < C_1}$, $\mathbf{M < C_2}$, $\mathbf{M < C_3}$ então $\mathbf{M = C^{1/3}}$
- **Solução**: Tornar **M** [Nota: usar *padding*] diferente para gerar $\mathbf{C_1}$, $\mathbf{C_2}$, $\mathbf{C_3}$

#3:

- Arranjar **p** e **q** tal que 3 seja primo relativo de $\mathbf{\phi(n) = (p-1)(q-1)}$

RSA vs integridade

- **O RSA só por si não garante integridade**
  - Dadas as cifras de m_1 e m_2, o atacante pode criar a cifra de $m_1 \cdot m_2$
    - $(m_1^e) \cdot (m_2^e) \bmod n = (m_1 \cdot m_2)^e \bmod n$
  - O atacante pode converter m em m^k sem decifrar
    - $(m_1^e)^k \bmod n = (m^k)^e \bmod n$

- Na prática é cifrado EM = OAEP (M) em vez do texto em claro M:

  EM = 0x00 || ***maskedSeed*** || ***maskedDB*** [ver *slide* seguinte]

- A cifra resultante é *plaintext-aware*: Impossível calcular uma cifra válida sem se conhecer o texto em claro.

RSA vs integridade

EM = 0x00 || maskedSeed || maskedDB

em que:

DB = *lHash* || *PS* || *0x01* || *M,* em que ***lHash*** = *Hash(L) e*
PS = *k – mLen – 2hLen – 2* octetos a zero, o comprimento de *PS* pode ser zero*; L,* se não for indicado outro valor, pode ser uma *string* vazia (*k representa o comprimento em octetos do* RSA modulus *n*)

seed **=** *string* aleatória com comprimento *hLen* [*hlen* – comprimento do *hash*]

dbMask = *MGF (seed, k – hLen – 1)* [*MGF(s,l)* – gera uma valor pseudo-aleatório de comprimento *l* a partir da semente *s*]

maskedDB = *DB* ⊕ *dbMask.*

seedMask = MGF (maskedDB, hLen)

maskedSeed = *seed* ⊕ *seedMask*

[https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf]

Comparação de criptosistemas

- **Chave simétrica**
  - Mesma chave em ambos os extremos
  - Gestão de chaves é um problema; requer canal seguro para a passagem das chaves
  - Rápido
- **Chaves assimétricas**
  - Duas chaves
    - Chave pública, conhecida de todos
    - Chave privada, só o dono a deve conhecer
  - A gestão de chaves é um problema menor
  - Computacionalmente intensivo, pelo que é lento

Criptografia híbrida

- Muito utilizada nas comunicações modernas

- Utiliza criptografia assimétrica para:
  - Passar as chaves de cifra simétrica da sessão
  - Autenticar a sessão (por vezes)

- Utiliza criptografia simétrica para:
  - Fornecer confidencialidade aos dados
  - Autenticar a sessão (por vezes)

Necessitamos de tudo isto?

- A criptografia simétrica é rápida

- **A criptografia assimétrica é lenta**
  - **Até 1000x mais lenta que a simétrica**

- Então, queremos utilizar a criptografia assimétrica -- a qual não requer chaves pré-entregues – para criar e/ou trocar chaves de sessão simétricas de maneira aos dados serem entregues rapidamente