Advanced Algorithms

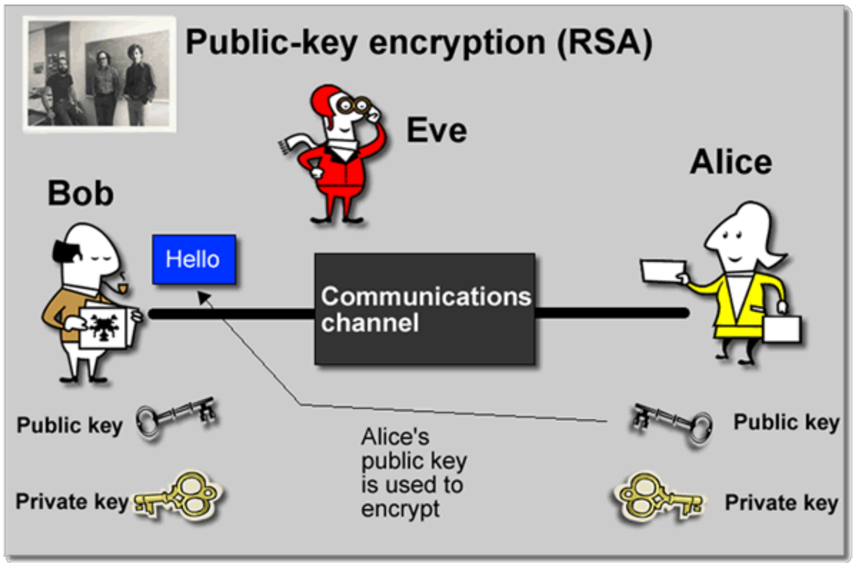
Floriano Zini

Free University of Bozen-Bolzano
Faculty of Computer Science

Academic Year 2013-2014

Lecture 2 – Algorithms with numbers

RSA Algorithm



Why does RSA work?

- RSA is based on the contrast between two problems
- **Factoring**
  - Given a number *N*, express it as a product of its prime factors
- **Primality**
  - Given a number *N*, determine whether it is a prime
- Factoring is **hard**
  - exponential time in the number of bits of *N*
- Primality is **efficient**
  - polynomial time in the number of bits of *N*

Why does RSA work? (cont.)

- Alice can **easily create** her **public** and **private keys**
- Bob can **easily encrypt** a message using Alice's **public key**
- Alice can **easily decrypt** the message using her own **private key**
- **Eve** (who doesn't know Alice's private key) can try to **decrypt** the message but this takes **exponential time**!

- For implementing RSA we need some algorithms that work on numbers ….

Addition

- The sum of any three single-digit numbers is at most two digits long
  - E.g., in base *10:* $9 + 9 + 9 = 27$
- This rule holds in any base $b \geq 2$
- In base *2:* $1_2 + 1_2 + 1_2 = 11_2 = 3_{10}$

Carry:	1			1	1	1		
		1	1	0	1	0	1	(53)
		1	0	0	0	1	1	(35)
	1	0	1	1	0	0	0	(88)

- The algorithm for sum is trivially correct

Addition (cont.)

- **Given two binary numbers *x* and *y*, how long does it take to add them?**
  - We want the answer expressed as a function of the size of the input
- Suppose *x* and *y* are each *n* bits long
- The sum of *x* and *y* is *n+1* bits long at most
  - E.g. n=2 → $11_2 + 11_2 = 110_2$
- Each individual bit of this sum gets computed in a fixed amount of time
- The total running time for the addition algorithm is of the form
  - $c_0 + c_1 * n$, where c_0 and c_1 are some constants, i.e., $O(n)$

Addition (cont.)

- **Is there a faster algorithm?**
- In order to add two *n*-bit numbers we must at least read them and write down the answer, and even that requires *n* operations.
- So the addition algorithm is optimal, up to multiplicative constants!
- Why *O(n)* operations? Isn't binary addition performed by computers in one instruction?
  - Yes, up to word length of today's computer – 32 bits perhaps
  - But it is often useful and necessary to handle very large numbers (several thousands bits long)

Multiplication

- The algorithm for multiplying two numbers *x* and *y* is to
  - create an array of intermediate sums, each representing the product of *x* by a single digit of *y*
  - These values are appropriately left-shifted and then added up
- For example: 13*11 in binary notation

$$
\begin{array}{cccccccc l}
 & & & & 1 & 1 & 0 & 1 & \\
 & & & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & 1 & 1 & 0 & 1 & \text{(1101 times 1)} \\
 & & & 1 & 1 & 0 & 1 & & \text{(1101 times 1, shifted once)} \\
 & & 0 & 0 & 0 & 0 & & & \text{(1101 times 0, shifted twice)} \\
+ & 1 & 1 & 0 & 1 & & & & \text{(1101 times 1, shifted thrice)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{(binary 143)}
\end{array}
$$

Multiplication (cont.)

$$
\begin{array}{cccccccc l}
 & & & & 1 & 1 & 0 & 1 & \\
 & & & \times & 1 & 0 & 1 & 1 & \\
\hline
 & & & & 1 & 1 & 0 & 1 & \text{(1101 times 1)} \\
 & & & 1 & 1 & 0 & 1 & & \text{(1101 times 1, shifted once)} \\
 & & 0 & 0 & 0 & 0 & & & \text{(1101 times 0, shifted twice)} \\
+ & 1 & 1 & 0 & 1 & & & & \text{(1101 times 1, shifted thrice)} \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \text{(binary 143)}
\end{array}
$$

And the complexity?

- If *x* and *y* are both *n* bits, then there are n intermediate rows, with lengths of up to *2n* bits
- The total time taken to add up these rows, doing two numbers at a time, is $O(n^2)$ that is:

$$\underbrace{O(n) + O(n) + \cdots + O(n)}_{n-1 \text{ times}},$$

Multiplication (cont.)

- Another algorithm

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is odd.} \end{cases}$$

```
function multiply(x, y)
Input: Two n-bit integers x and y, where y ≥ 0
Output: Their product

if y = 0: return 0
z = multiply(x, ⌊y/2⌋)
if y is even:
   return 2z
else:
   return x + 2z
```

- Is it correct?
  - Yes! It mimics the rule and handle the base case (*y=0*) properly


```
function multiply(x, y)
Input: Two n-bit integers x and y, where y ≥ 0
Output: Their product

if y = 0: return 0
z = multiply(x, ⌊y/2⌋)
if y is even:
   return 2z
else:
   return x + 2z
```

How long does it take?

- It terminates after *n* recursive calls, because at each call *y* is halved
- At each call it performs *O(n)* bit operations
- **The total time is thus $O(n^2)$**

Division

- To divide an integer x by another integer $y<>0$ means to find a quotient q and a remainder r, where

$$x = y*q + r \text{ and } r < y$$

```
function divide(x, y)
Input: Two n-bit integers x and y, where y ≥ 1
Output: The quotient and remainder of x divided by y

if x = 0: return (q, r) = (0, 0)
(q, r) = divide(⌊x/2⌋, y)
q = 2·q, r = 2·r
if x is odd: r = r + 1
if r ≥ y: r = r − y, q = q + 1
return (q, r)
```

- As multiplication, it takes $O(n^2)$ time

Modular arithmetic

- With repeated addition or multiplication, numbers can get cumbersomely large
- For cryptography it is necessary to deal with numbers that are significantly **large** but whose range is **limited**
- Modular arithmetic deals with restricted ranges of integers
- We define *x modulo N* to be the remainder when x is divided by N
  - If $x = q*N + r$ with $0 \leq r < N$ then *x modulo N* is equal to r

Modular arithmetic (cont.)

- *x and* y are *congruent modulo N* if they differ by a multiple of *N*

  $x \equiv y \pmod N$ *iff N divides* $(x-y)$
- E.g.
  - 253 ≡ 13 (mod 60) because 60 divides 253-13
  - 59 ≡ -1 (mod 60) because 60 divides 59-(-1)=60

Two interpretations

- Modular arithmetic limits numbers to a predefined range *{0, 1, . . . , N-1}* and wraps around whenever you leave this range, like the hand of a clock
  - Example of addition modulo 8

- Modular arithmetic It deals with all the integers, but divides them into *N* equivalence classes of the form $\{i + k*N \mid k \in Z\}$ for some *i* between *0* and $N - 1$

Which are the classes of equivalence modulo 3?

…	−9	−6	−3	0	3	6	9	…
…	−8	−5	−2	1	4	7	10	…
…	−7	−4	−1	2	5	8	11	…

$c_0 = \{0 + k*3 \mid k \in Z\}$

$c_1 = \{1 + k*3 \mid k \in Z\}$

$c_2 = \{2 + k*3 \mid k \in Z\}$

Useful rules

- **Substitution rule**
  - If $x \equiv x'$ *(mod N)* and $y \equiv y'$ *(mod N)*, then $x+y \equiv x'+y'$ *(mod N)* and $x*y \equiv x'*y'$ *(mod N)*
  - Example:
    - $5 \equiv 8$ *(mod 3)* and $-2 \equiv 1$ *(mod 3)*
    - $5+(-2) \equiv 8+1$ *(mod 3)*

…	−9	−6	−3	0	3	6	9	…
…	−8	−5	−2	1	4	7	10	…
…	−7	−4	−1	2	5	8	11	…

  - Any member of an equivalence class is substitutable for any other
- **Algebraic rules**
  - $x+(y+z) \equiv (x+y)+z$ *(mod N)* Associativity
  - $x*y \equiv y*x$ *(mod N)* Commutativity
  - $x*(y + z) \equiv x*y + x*z$ *(mod N)* Distributivity
- Example of rule application
  - $2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1$ *(mod 31)*

Modular addition

- We want to calculate $x + y$ *modulo* N
  - $0 \leq x, y \leq N-1 \rightarrow 0 \leq x + y \leq 2*(N-1)$
  - If $x + y > N\text{-}1$ then
    $x + y \pmod N = x + y - N \leq N-1$
- The computation consists of an addition, and possibly a subtraction, of numbers that never exceed $2*N$
- The running time is linear in the sizes of x, y
  - **Modular addition is $O(n)$**, where $n \approx \log N$

Modular multiplication

- We want to calculate $x * y$ *modulo* N
  - $0 \leq x, y \leq N-1 \rightarrow 0 \leq x * y \leq (N-1)^2$
  - If $x * y > N\text{-}1$ then $x * y \pmod n$ is obtained by calculating the reminder upon dividing it by N
- The computation consists of a multiplication, and possibly a reminder of numbers $\leq (N-1)^2$
  - $(N-1)^2$ is at most $2*\log(N\text{-}1) \leq 2*n$ bits long
- Multiplication is $O(n^2)$, so it is division (used for reminder)
  - **Modular multiplication is a $O(n^2)$ operation**

Modular division

- Not quite so easy
- In ordinary arithmetic there is just one tricky case
  - division by zero
- In modular arithmetic there are potentially other tricky cases
- Whenever **modular division** is legal, it can be managed in cubic time, $O(n^3)$