6.7 Fast exponentiation

6.8 Introduction to cryptography

6.9 The RSA cryptosystem

In cryptography it is important to be able to find b^n efficiently, where b, n are large integers.

We can use an algorithm that employs the binary expansion of the exponent n. To compute b^n, the algorithm computes b, b^2, $(b^2)^2$,$(b^4)^2$, ... till some point, and then multiplies all of them.

Input: Positive integers x and y.
Output: x^y

```
procedure fastExponentiation(x,y)
p := 1     // p holds the partial result.
s := x     // s holds the current (x^2)^j
r := y     // r is used for binary expansion of y

While ( r > 0 )
      If ( r mod 2 = 1 ), p := p·s
      s := s·s
      r := r div 2
End-while
Return(p)
```

Example: Find 7^{16}

p = 1, s = 7, r = 16

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49 7^2
r = 16 div 2 = 8

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49 7^2
r = 16 div 2 = 8

p = 1, s = 49, r = 8

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49 7^2
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401 $(7^2)^2$
r = 8 div 2 = 4

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401 7^4
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401 7^4
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801 $(7^4)^2$
r = 4 div 2 = 2

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 5 764 801, r = 2

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801
r = 4 div 2 = 2

7^8

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801 7^8
r = 4 div 2 = 2

p = 1, s = 5 764 801, r = 2

4 mod 2 = 0
s = $5\ 764\ 801^2 = 33\ 232\ 930\ 569\ 601$ $(7^8)^2$
r = 2 div 2 = 1

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801
r = 4 div 2 = 2

p = 1, s = 5 764 801, r = 2

4 mod 2 = 0
s = 5 764 801^2 = 33 232 930 569 601 7^{16}
r = 2 div 2 = 1

p = 1, s = 33 232 930 569 601, r = 1

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801
r = 4 div 2 = 2

p = 1, s = 5 764 801, r = 2

4 mod 2 = 0
s = 5 764 801^2 = 33 232 930 569 601 7^{16}
r = 2 div 2 = 1

p = 1, s = 33 232 930 569 601, r = 1

1 mod 2 = 1
p = 1 * 33 232 930 569 601
s = 33 232 930 569 601^2 = ...
r = 1 div 2 = 0

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801
r = 4 div 2 = 2

p = 1, s = 5 764 801, r = 2

4 mod 2 = 0
s = 5 764 801^2 = 33 232 930 569 601 7^{16}
r = 2 div 2 = 1

p = 1, s = 33 232 930 569 601, r = 1

1 mod 2 = 1
p = 1 * 33 232 930 569 601 7^{16}
s = 33 232 930 569 601^2 = ...
r = 1 div 2 = 0

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

Modular Exponentiation

Example: Find 7^{16}

p = 1, s = 7, r = 16

16 mod 2 = 0
s = 7*7 = 49
r = 16 div 2 = 8

p = 1, s = 49, r = 8

8 mod 2 = 0
s = 49*49 = 2401
r = 8 div 2 = 4

p = 1, s = 2 401, r = 4

4 mod 2 = 0
s = 2 401*2 401 = 5 764 801
r = 4 div 2 = 2

p = 1, s = 5 764 801, r = 2

4 mod 2 = 0
s = 5 764 801^2 = 33 232 930 569 601
r = 2 div 2 = 1

p = 1, s = 33 232 930 569 601, r = 1

1 mod 2 = 1
p = 1 * 33 232 930 569 601
s = 33 232 930 569 601^2 = ...
r = 1 div 2 = 0

7^{16}

```
p := 1
s := x
r := y

While ( r > 0 )
    If ( r mod 2 = 1 )
       p := p·s
    s := s·s
    r := r div 2
End-while
Return(p)
```

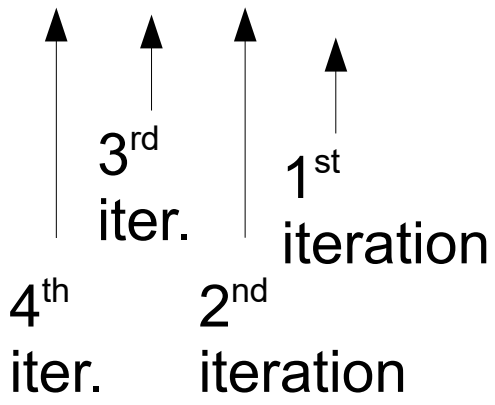
Return(33 232 930 569 601)

Example: Find 7^{16}

What does the algorithm do?

$16 = (1\ 0000)_2 = 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 0 * 2^0$

Therefore $7^{16} = 7^{1*2^4+0*2^3+0*2^2+0*2^1+0*2^0} = 7^{2^4} * 7^0 * 7^0 * 7^0 * 7^0$



Modular Exponentiation

In cryptography it is important to be able to find b^n ***mod*** m efficiently, where b, n, and m are large integers.

As we have discussed, it is impractical to first compute b^n and then find its remainder when divided by m, because b^n will be a huge number.

Modular Exponentiation

In cryptography it is important to be able to find b^n ***mod*** m efficiently, where b, n, and m are large integers.

As we have discussed, it is impractical to first compute b^n and then find its remainder when divided by m, because b^n will be a huge number.

```
Input: Positive integers x and y.
Output: x^y mod n

p := 1    //p holds the partial result.
s := x    //s holds the current (x^2)^j
r := y    //r is used to compute the binary expansion of y

While ( r > 0 )
      If ( r mod 2 = 1 )
            p := p·s mod n
      s := s·s mod n
      r := r div 2
End-while

Return(p)
```

modifications from fast exponentiation are shown in pink

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 1, s = 7, r = 644

```
p := 1, s := x
r := y

While ( r > 0 )
  If ( r mod 2 = 1 )
     p := p·s mod n
  s := s·s mod n
  r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 1, s = 7, r = 644

644 mod 2 != 1, *hence p is not updated*
s = 7 * 7 mod 645 = 49 mod 645 = 49
r = 644 div 2 = 322

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 1, s = 7, r = 644

644 mod 2 != 1, *hence p is not updated*
s = 7 * 7 mod 645 = 49 mod 645 = 49
r = 644 div 2 = 322

p = 1, s = 49, r = 322

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Modular Exponentiation

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 1, s = 7, r = 644

644 mod 2 != 1, *hence p is not updated*
s = 7 * 7 mod 645 = 49 mod 645 = 49
r = 644 div 2 = 322

p = 1, s = 49, r = 322

322 mod 2 != 1, *hence p is not updated*
r = 49^2 mod 645 = 2401 mod 645 = 466
r = 322 div 2 = 161

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 1, s = 7, r = 644

644 mod 2 != 1, *hence p is not updated*
s = 7 * 7 mod 645 = 49 mod 645 = 49
r = 644 div 2 = 322

p = 1, s = 49, r = 322

322 mod 2 != 1, *hence p is not updated*
r = 49^2 mod 645 = 2401 mod 645 = 466
r = 322 div 2 = 161

p = 1, s = 466, r = 161

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Modular Exponentiation

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 1, s = 7, r = 644

644 mod 2 != 1, *hence p is not updated*
s = 7 * 7 mod 645 = 49 mod 645 = 49
r = 644 div 2 = 322

p = 1, s = 49, r = 322

322 mod 2 != 1, *hence p is not updated*
r = 49^2 mod 645 = 2401 mod 645 = 466
r = 322 div 2 = 161

p = 1, s = 466, r = 161

161 mod 2 = 1, *hence p is updated*
p = 1 * 466 mod 645 = 466
s = 466^2 mod 645 = 436
r = 161 div 2 = 80

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 1, s = 7, r = 644

644 mod 2 != 1, *hence p is not updated*
s = 7 * 7 mod 645 = 49 mod 645 = 49
r = 644 div 2 = 322

p = 1, s = 49, r = 322

322 mod 2 != 1, *hence p is not updated*
r = 49^2 mod 645 = 2401 mod 645 = 466
r = 322 div 2 = 161

p = 1, s = 466, r = 161

161 mod 2 = 1, *hence p is updated*
p = 1 * 466 mod 645 = 466
s = 466^2 mod 645 = 436
r = 161 div 2 = 80

p = 466, s = 436, r = 80

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 466, s = 436, r = 80

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 466, s = 436, r = 80

80 mod 2 != 1, *hence p is not updated*
$s = 436^2 \bmod 645 = 466$
r = 80 div 2 = 40

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 466, s = 436, r = 80

80 mod 2 != 1, *hence p is not updated*
s = 436^2 mod 645 = 466
r = 80 div 2 = 40

p = 466, s = 466, r = 40

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 466, s = 436, r = 80

80 mod 2 != 1, *hence p is not updated*
s = 436^2 mod 645 = 466
r = 80 div 2 = 40

p = 466, s = 466, r = 40

40 mod 2 != 1, *hence p is not updated*
s = 466^2 mod 645 = 436
r = 40 div 2 = 20

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 466, s = 436, r = 80

80 mod 2 != 1, *hence p is not updated*
s = 436^2 mod 645 = 466
r = 80 div 2 = 40

p = 466, s = 466, r = 40

40 mod 2 != 1, *hence p is not updated*
s = 466^2 mod 645 = 436
r = 40 div 2 = 20

p = 466, s = 436, r = 20

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 466, s = 436, r = 80

80 mod 2 != 1, *hence p is not updated*
$s = 436^2 \bmod 645 = 466$
r = 80 div 2 = 40

p = 466, s = 466, r = 40

40 mod 2 != 1, *hence p is not updated*
$s = 466^2 \bmod 645 = 436$
r = 40 div 2 = 20

p = 466, s = 436, r = 20

20 mod 2 != 1, *hence p is not updated*
$s = 436^2 \bmod 645 = 466$
r = 20 div 2 = 10

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 466, s = 436, r = 80

80 mod 2 != 1, *hence p is not updated*
$s = 436^2 \bmod 645 = 466$
r = 80 div 2 = 40

p = 466, s = 466, r = 40

40 mod 2 != 1, *hence p is not updated*
$s = 466^2 \bmod 645 = 436$
r = 40 div 2 = 20

p = 466, s = 436, r = 20

20 mod 2 != 1, *hence p is not updated*
$s = 436^2 \bmod 645 = 466$
r = 20 div 2 = 10

p = 466, s = 466, r = 10

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

644=(10 1000 0100)$_2$

p = 466, s = 466, r = 10

10 mod 2 != 1, *hence p is not updated*
s = 466^2 mod 645 = 436
r = 10 div 2 = 5

p = 466, s = 436, r = 5

5 mod 2 = 1, *hence p is updated*
p = 466 * 436 mod 645 = 1
s = 436^2 mod 645 = 466
r = 5 div 2 = 2

p = 1 , s = 466, r = 2

2 mod 2 != 1, *hence p is not updated*
s = 466^2 mod 645 = 436
r = 2 div 2 = 1

p = 1, s = 436, r=1

```
p := 1, s := x
r := y

While ( r > 0 )
   If ( r mod 2 = 1 )
      p := p·s mod n
   s := s·s mod n
   r := r div 2
End-while

Return(p)
```

Example: Find 7^{644} ***mod*** *645*

$644=(10\ 1000\ 0100)_2$

p = 1, s = 436, r=1

1 mod 2 = 1, hence p is updated
p = 1 * 436 mod 645 = 436
s = 436^2 mod 645 = 466
r = 1 div 2 = 0

p = 436, s = 466, r = 0

STOP

Return(436)

7^{644} ***mod*** *645 = 436*

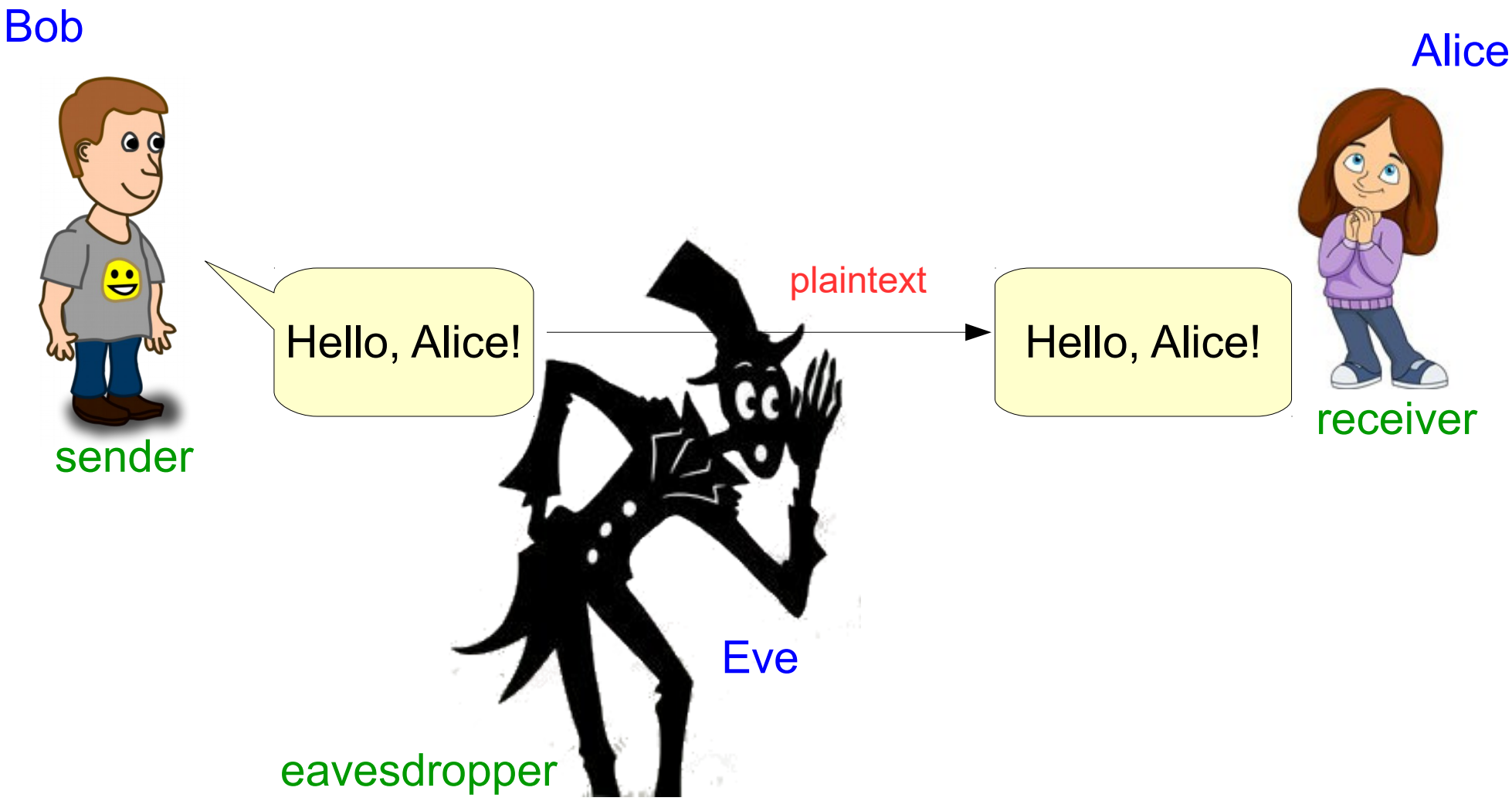
```
p := 1, s := x
r := y

While ( r > 0 )
  If ( r mod 2 = 1 )
     p := p·s mod n
  s := s·s mod n
  r := r div 2
End-while

Return(p)
```

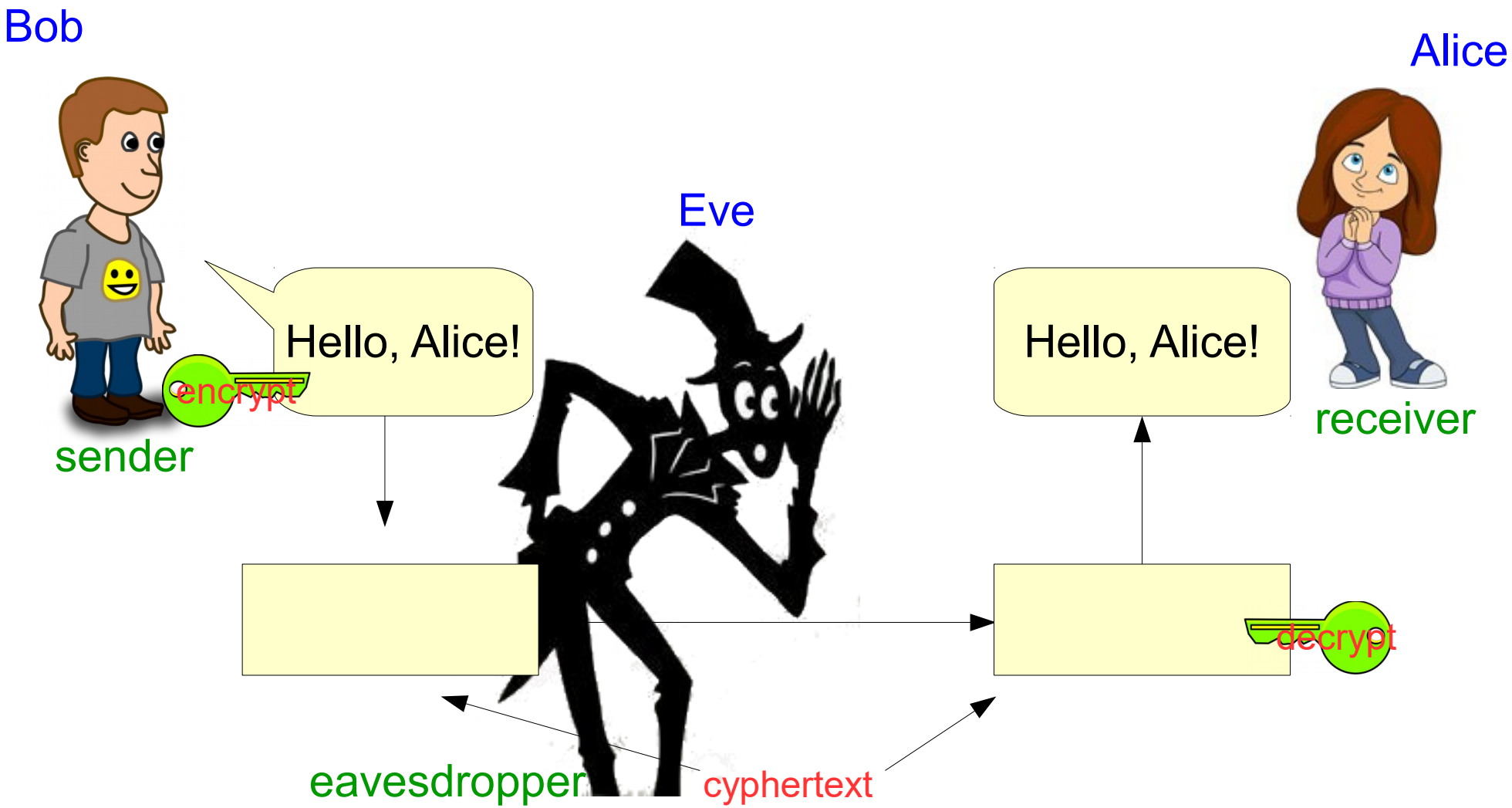
6.8 Introduction to cryptography

Cryptography is science of protecting and authenticating data and communication.
One important aspect: sending messages securely in the presence of eavesdroppers who can learn the transmitted information.



6.8 Introduction to cryptography

Cryptography is science of protecting and authenticating data and communication.
One important aspect: sending messages securely in the presence of eavesdroppers who can learn the transmitted information.



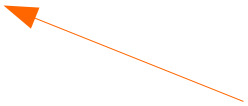
6.8 Introduction to cryptography

Often we transmit text messages. Therefore, we convert text message into an integer, then the “message” is encrypted and sent.

There are many possible ways to do the conversion.

Example: assume that the message contains only uppercase letters, space characters, and periods

A	01	N	14	_	27
B	02	O	15	.	28
C	03	P	16		
D	04	Q	17		
E	05	R	18		
F	06	S	19		
G	07	T	20		
H	08	U	21		
I	09	V	22		
J	10	W	23		
K	11	X	24		
L	12	Y	25		
M	13	Z	26		



not onto, but
has to be one-to-one

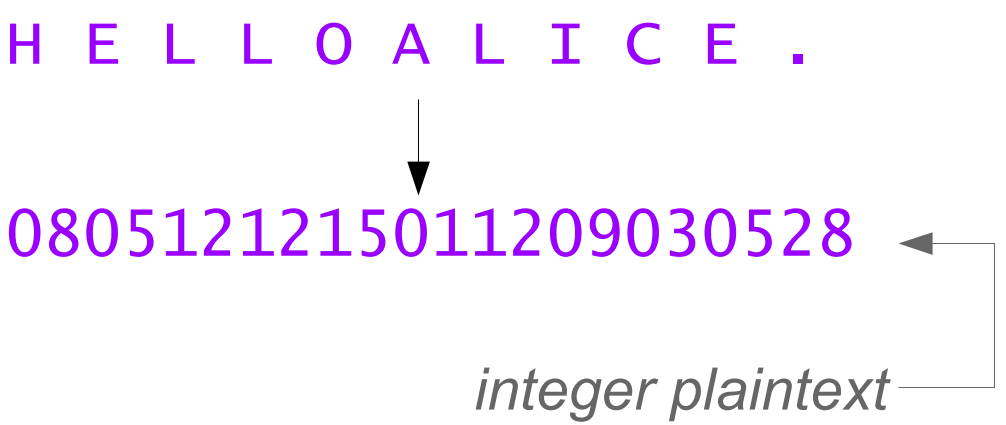
6.8 Introduction to cryptography

Often we transmit text messages. Therefore, we convert text message into an integer, then the “message” is encrypted and sent.

There are many possible ways to do the conversion.

Example: assume that the message contains only uppercase letters, space characters, and periods

A	01	N	14	_	27
B	02	O	15	.	28
C	03	P	16		
D	04	Q	17		
E	05	R	18		
F	06	S	19		
G	07	T	20		
H	08	U	21		
I	09	V	22		
J	10	W	23		
K	11	X	24		
L	12	Y	25		
M	13	Z	26		



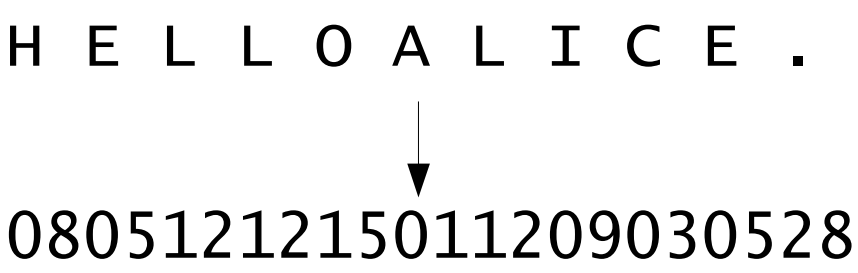
6.8 Introduction to cryptography

Often we transmit text messages. Therefore, we convert text message into an integer, then the “message” is encrypted and sent.

There are many possible ways to do the conversion.

Example: assume that the message contains only uppercase letters, space characters, and periods

A	01	N	14	_	27
B	02	O	15	.	28
C	03	P	16		
D	04	Q	17		
E	05	R	18		
F	06	S	19		
G	07	T	20		
H	08	U	21		
I	09	V	22		
J	10	W	23		
K	11	X	24		
L	12	Y	25		
M	13	Z	26		



The mapping of text messages to numbers described above gives a function from strings of length n to integers with *2n* digits.

The translation of text messages to numbers need not be secure.

6.8 Introduction to cryptography

Modern cryptosystems rely on **number theory** in which the encryption and decryption procedures are mathematical functions whose input and output are integers.

To encrypt a plaintext message: compute a mathematical function with the *integer plaintext m* as the input and the *ciphertext c* as the output.

To decrypt: is to compute the inverse of the encryption process.
Given a *ciphertext c*, the decryption process must produce the unique *plaintext m* whose encryption is *c*.

Let
M: set of all possible plaintexts, $\mathbf{M} \subset \mathbf{Z}$
C: set of all ciphertexts
then
Encryption is a function: $\mathbf{M} \rightarrow \mathbf{Z}$, with range **C**

6.8 Introduction to cryptography

A simple cryptosystem

Assume that the set of all possible plaintexts come from $\mathbf{Z}_N$ for some integer N.

Alice and Bob share a secret number $k \in \mathbf{Z}_N$.

The security of their encryption scheme rests on the assumption that no one besides them knows the number *k*.

6.8 Introduction to cryptography

A simple cryptosystem

Assume that the set of all possible plaintexts come from $\mathbf{Z}_N$ for some integer N.
Alice and Bob share a secret number $k \in \mathbf{Z}_N$.

The security of their encryption scheme rests on the assumption that no one besides them knows the number *k*.

To encrypt a *plaintext* $m \in \mathbf{Z}_N$: compute $c = (m+k) \bmod N$

To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k) \bmod N$

6.8 Introduction to cryptography

A simple cryptosystem

Assume that the set of all possible plaintexts come from $\mathbf{Z}_N$ for some integer N.
Alice and Bob share a secret number $k \in \mathbf{Z}_N$.

The security of their encryption scheme rests on the assumption that no one besides them knows the number *k*.

To encrypt a *plaintext* $m \in \mathbf{Z}_N$: compute $c = (m+k)$ **mod** N

To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k)$ **mod** N

Simple encryption scheme requirements:

If $m_1 \neq m_2$ and $m_1, m_2 \in \mathbf{Z}_N$ then $(m_1 + k)$ **mod** N $\neq (m_2 + k)$ **mod** N
(i.e. no two distinct *plaintexts* map to same *ciphertext*)

If $m \in \mathbf{Z}_N$ then (((m + k) **mod** N) - k) mod N = m
(i.e. decryption scheme is inverse of encryption scheme)

6.8 Introduction to cryptography

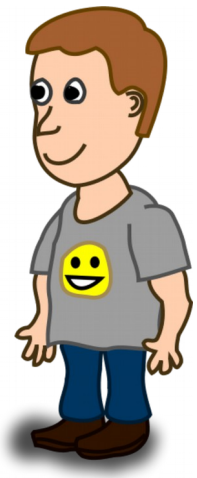
To encrypt a *plaintext* $m \in \mathbf{Z_N}$: compute $c = (m+k) \bmod N$

To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k) \bmod N$

Example: let N = 1028

Alice wants to send a “message” 978 to Bob.

Bob

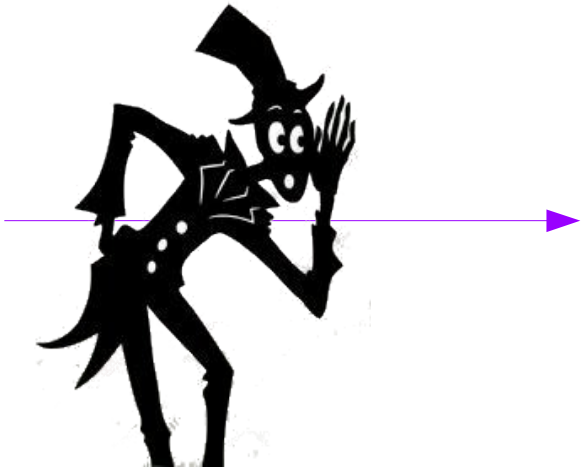


k = 678

Alice

message = “978”

k = 678



6.8 Introduction to cryptography

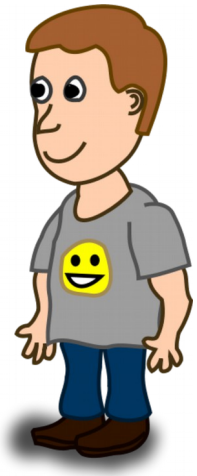
To encrypt a *plaintext* $m \in \mathbf{Z}_N$: compute $c = (m+k) \bmod N$

To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k) \bmod N$

Example: let N = 1028

Alice wants to send a "message" 978 to Bob.

Bob



k = 678

Alice

k = 678

message = "978"

c = (978+678) mod 1028 = 628

?????
628???

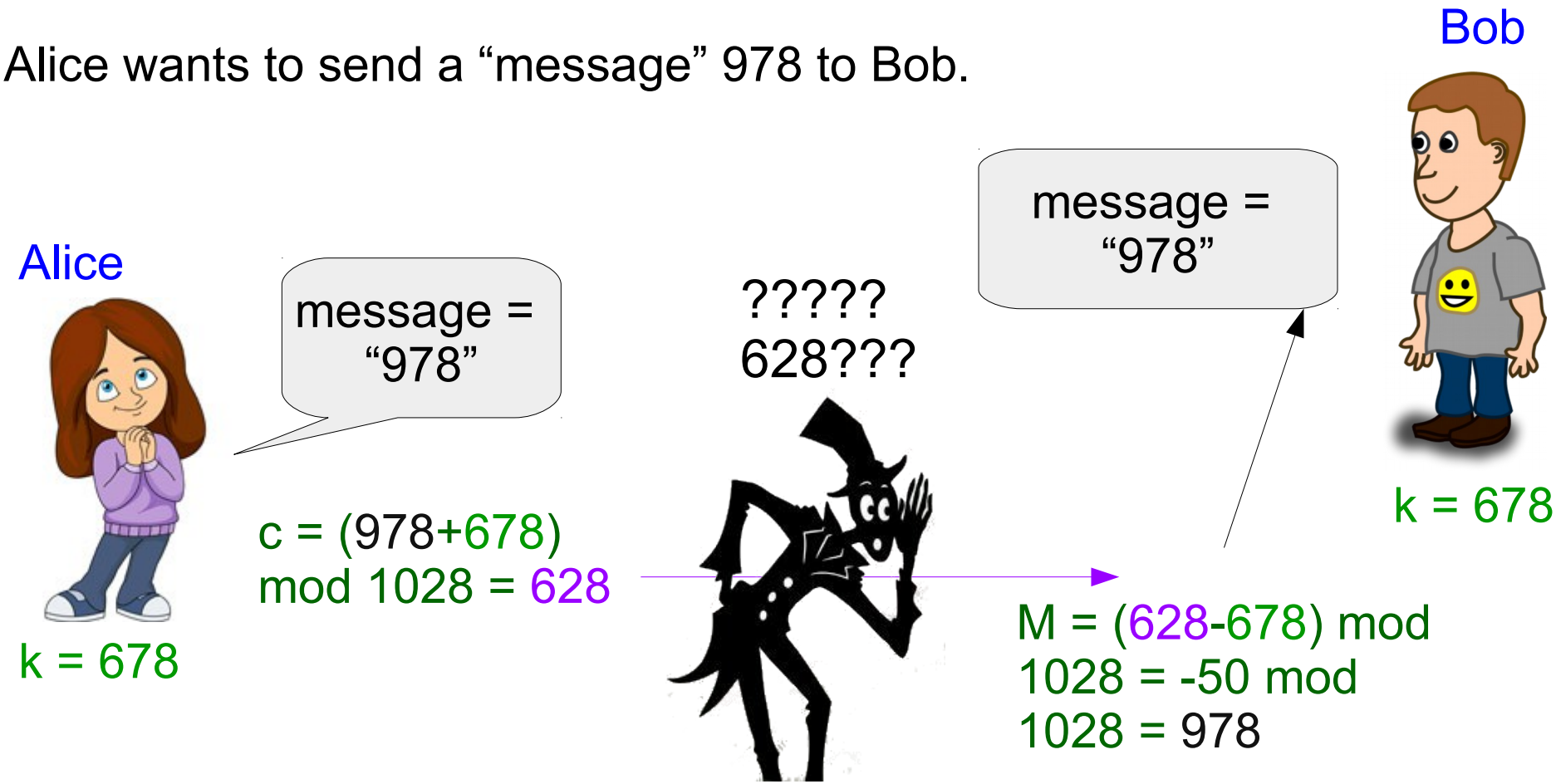
6.8 Introduction to cryptography

To encrypt a *plaintext* $m \in \mathbf{Z_N}$: compute $c = (m+k) \bmod N$

To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k) \bmod N$

Example: let N = 1028

Alice wants to send a "message" 978 to Bob.



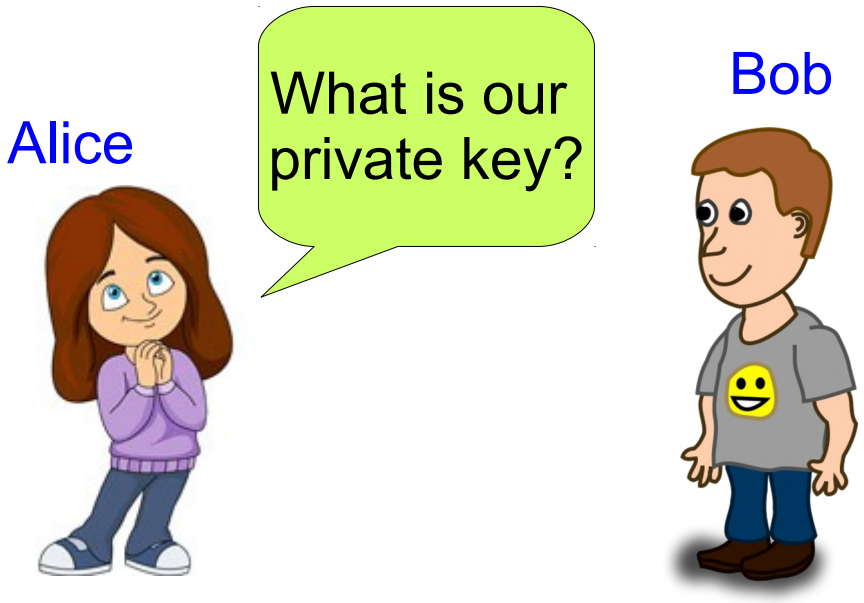
6.8 Introduction to cryptography

To encrypt a *plaintext* $m \in \mathbf{Z_N}$: compute $c = (m+k)$ **mod** N

To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k)$ **mod** N

Example: let N = 1028

Alice wants to send a "message" 978 to Bob.



The simple encryption scheme presented here is an example of ***private key*** cryptography. In a private key cryptosystem,

Alice and Bob must meet in advance (or communicate over a reliably secure channel) to decide on the value of a secret key.

6.8 Introduction to cryptography

A simple cryptosystem

Assume that the set of all possible plaintexts come from $\mathbf{Z}_N$ for some integer N.
Alice and Bob share a secret number $k \in \mathbf{Z}_N$.

The security of their encryption scheme rests on the assumption that no one besides them knows the number *k*.

To encrypt a *plaintext* $m \in \mathbf{Z}_N$: compute $c = (m+k) \bmod N$
To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k) \bmod N$

The simple cryptosystem described here is not very secure.

1. if Eve can ever get ahold of one *plaintext* and its corresponding *ciphertext*, she can determine *k* and decrypt all further *plaintexts* from Alice to Bob.

6.8 Introduction to cryptography

A simple cryptosystem

Assume that the set of all possible plaintexts come from $\mathbf{Z}_N$ for some integer N.
Alice and Bob share a secret number $k \in \mathbf{Z}_N$.

The security of their encryption scheme rests on the assumption that no one besides them knows the number *k*.

To encrypt a *plaintext* $m \in \mathbf{Z}_N$: compute $c = (m+k) \bmod N$
To decrypt a *cyphertext* $c \in \mathbf{C}$: compute $m = (c-k) \bmod N$

The simple cryptosystem described here is not very secure.

2. English text has many well understood patterns.
For example the letters "ing" often occur in sequence but the letters "qx" almost never do. Eve will know something about the pattern of likely messages based on the patterns of English text. If she is allowed to see a large enough number of encrypted messages, she can match the pattern of ciphertexts with the pattern of likely plaintexts and infer k.

6.9 The RSA cryptosystem

In the *simple cryptosystem* Alice and Bob need to meet either in person or communicate by a perfectly secure channel in order to agree on the value of encryption and decryption keys.

It is not always convenient to arrange such a hand-off of the keys.

Another issue: private keys need to be renewed periodically, since they eventually become compromised after multiple uses.

6.9 The RSA cryptosystem

Public key cryptography

Public key cryptosystems were first realized in the 1970s

Bob has an *encryption key* (*public key*) that he provides publicly so that anyone can use it to send him an encrypted message.
Bob holds a matching *decryption key* (*private key*) that he keeps privately to decrypt messages.

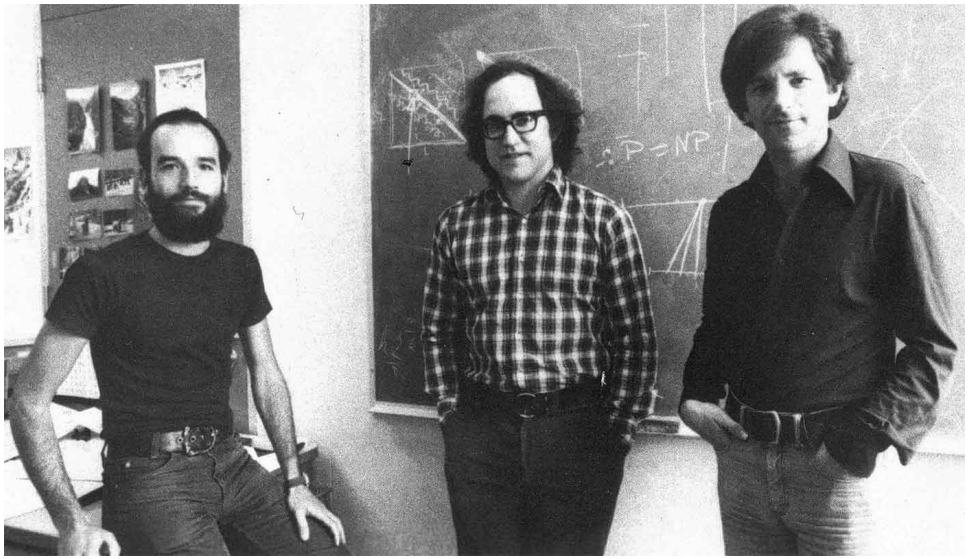
While anyone can use the public key to encrypt a message, the security of the scheme depends on the fact that it is difficult to decrypt the message without having the matching private decryption key.

6.9 The RSA cryptosystem

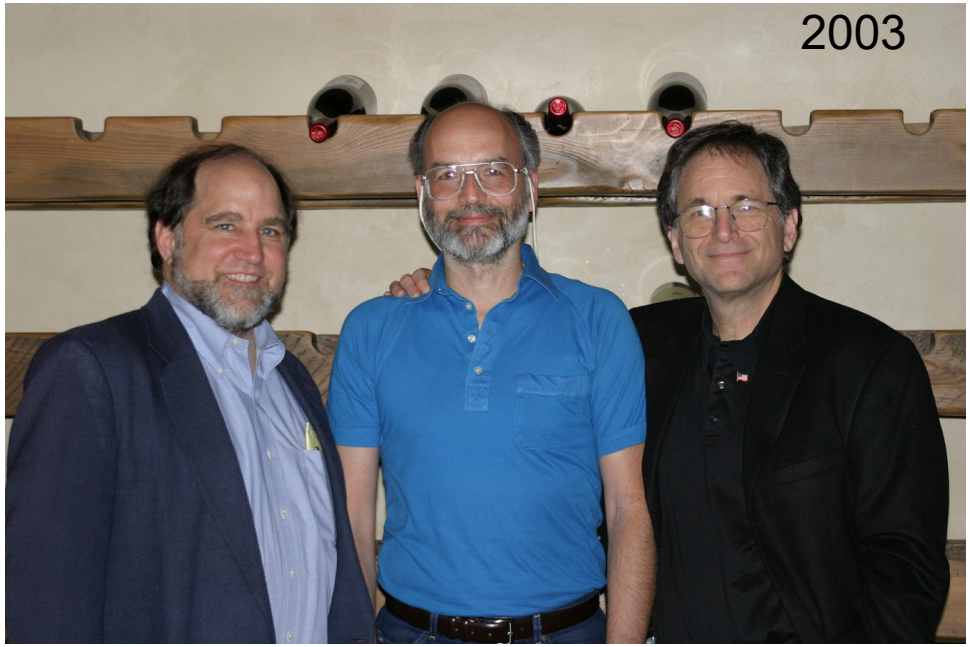
The most widely used public key cryptosystem, developed by Rivest, Adelman, and Shamir in 1978, called the *RSA cryptosystem*.

The security of the RSA cryptosystem rests on the assumption that it is difficult to factor large numbers.

If there were an efficient algorithm to factor numbers, then messages encrypted through RSA could be decrypted by someone who does not hold the matching decryption key.



at the time of creation



Ron Rivest Adi Sharir Leonard Adelman

6.9 The RSA cryptosystem

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q (*several hundred digits*)
2. Bob computes N = pq and $\varphi = (p-1)(q-1)$
(*! assumption: given pq, it would be infeasible to discover p or q*)
3. Bob finds an integer e such that gcd(e, φ) = 1 (i.e. *relatively prime*)
(*e is often chosen to be a prime number*)
4. Bob computes the multiplicative inverse of e **mod** φ:
an integer d such that (ed mod φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

6.9 The RSA cryptosystem

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q (*several hundred digits*)
2. Bob computes N = pq and φ = (p-1)(q-1)
(*! assumption: given pq, it would be infeasible to discover p or q*)
3. Bob finds an integer e such that gcd(e, φ) = 1 (i.e. *relatively prime*)
(*e is often chosen to be a prime number*)
4. Bob computes the multiplicative inverse of e **mod** φ:
an integer d such that (ed **mod** φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

Example:
1. Let p = 17 and q = 23

6.9 The RSA cryptosystem

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q (*several hundred digits*)
2. Bob computes N = pq and φ = (p-1)(q-1)
(*! assumption: given pq, it would be infeasible to discover p or q*)
3. Bob finds an integer e such that gcd(e, φ) = 1 (i.e. *relatively prime*)
(*e is often chosen to be a prime number*)
4. Bob computes the multiplicative inverse of e **mod** φ:
an integer d such that (ed **mod** φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

Example:
1. Let p = 17 and q = 23
2. N = pq = 17 * 23 = 391 φ = (p-1)(q-1) = 16 * 22 = 352

6.9 The RSA cryptosystem

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q (*several hundred digits*)
2. Bob computes N = pq and φ = (p-1)(q-1)
(*! assumption: given pq, it would be infeasible to discover p or q*)
3. Bob finds an integer e such that gcd(e, φ) = 1 (i.e. *relatively prime*)
(*e is often chosen to be a prime number*)
4. Bob computes the multiplicative inverse of e **mod** φ:
an integer d such that (ed **mod** φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

Example:
1. Let p = 17 and q = 23
2. N = pq = 17 * 23 = 391 φ = (p-1)(q-1) = 16 * 22 = 352
3. 352 = 2^5*11, so let's pick e = 197 *(prime number)*

6.9 The RSA cryptosystem

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q (*several hundred digits*)
2. Bob computes N = pq and φ = (p-1)(q-1)
(*! assumption: given pq, it would be infeasible to discover p or q*)
3. Bob finds an integer e such that gcd(e, φ) = 1 (i.e. *relatively prime*)
(*e is often chosen to be a prime number*)
4. Bob computes the multiplicative inverse of e **mod** φ:
an integer d such that (ed **mod** φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

Example:
1. Let p = 17 and q = 23
2. N = pq = 17 * 23 = 391 φ = (p-1)(q-1) = 16 * 22 = 352
3. 352 = 2^5*11, so let's pick e = 197 *(prime number)*
4. using *Extended Euclid's Algorithm* for finding gcd(197,352),
we will find s and t, such that
1 = s*197 + t*352 … *see the process in a separate file*... s = 109, t = -61
d = s **mod** φ = 109 **mod** 352 = 109

6.9 The RSA cryptosystem

Preparation of public and private keys in RSA

1. Bob selects two large prime numbers, p and q (*several hundred digits*)
2. Bob computes N = pq and φ = (p-1)(q-1)
(*! assumption: given pq, it would be infeasible to discover p or q*)
3. Bob finds an integer e such that gcd(e, φ) = 1 (i.e. *relatively prime*)
(*e is often chosen to be a prime number*)
4. Bob computes the multiplicative inverse of e **mod** φ:
an integer d such that (ed **mod** φ) = 1
5. Public (encryption) key: N and e
6. Private (decryption) key: d

5. public key: N = 391, e = 197
6. private key: d = 109

Example:
1. Let p = 17 and q = 23
2. N = pq = 17 * 23 = 391 φ = (p-1)(q-1) = 16 * 22 = 352
3. 352 = 2^5*11, so let's pick e = 197 *(prime number)*
4. using *Extended Euclid's Algorithm* for finding gcd(197,352),
we will find s and t, such that
1 = s*197 + t*352 … *see the process in a separate file*... s = 109, t = -61
d = s **mod** φ = 109 **mod** 352 = 109

6.9 The RSA cryptosystem

When Alice wants send a *plaintext* message *m* to Bob, the RSA scheme requires that:

- $m \in \mathbf{Z}_N$, and
- *m* is not a multiple of p or q.

p and q are primes with hundreds of digits, hence it is extremely unlikely that m is a multiple of primes p or q.

Alice encrypts her plaintext using e and N to produce ciphertext *c* as follows:

$c = m^e \bmod N$ (encryption)

Alice transmits c to Bob. Bob decrypts the cyphertext using d to recover m from c:

$m = c^d \bmod N$ (decryption)

6.9 The RSA cryptosystem

When Alice wants send a *plaintext* message *m* to Bob, the RSA scheme requires that:

- $m \in \mathbf{Z}_N$, and
- *m* is not a multiple of p or q.

p and q are primes with hundreds of digits, hence it is extremely unlikely that m is a multiple of primes p or q.

Alice encrypts her plaintext using e and N to produce ciphertext *c* as follows:

$c = m^e \bmod N$ (encryption)

Alice transmits c to Bob. Bob decrypts the cyphertext using d to recover m from c:

$m = c^d \bmod N$ (decryption)

Example (continues): public key: N = 391, e = 197 private key: d = 109

not a multiple of 17 or 23

Alice will encode message 223.

6.9 The RSA cryptosystem

When Alice wants send a *plaintext* message *m* to Bob, the RSA scheme requires that:

- $m \in \mathbf{Z}_N$, and
- *m* is not a multiple of p or q.

p and q are primes with hundreds of digits, hence it is extremely unlikely that m is a multiple of primes p or q.

Alice encrypts her plaintext using e and N to produce ciphertext *c* as follows:

$c = m^e \bmod N$ (encryption)

Alice transmits c to Bob. Bob decrypts the cyphertext using d to recover m from c:

$m = c^d \bmod N$ (decryption)

Example (continues): public key: N = 391, e = 197 private key: d = 109

not a multiple of 17 or 23

Alice will encode message 223:

c = 223^{197} **mod** 391 *- here we will recall our modular exponentiation (Section 6.7)* ***c = 151***

6.9 The RSA cryptosystem

When Alice wants send a *plaintext* message *m* to Bob, the RSA scheme requires that:

- $m \in \mathbf{Z}_N$, and
- *m* is not a multiple of p or q.

p and q are primes with hundreds of digits, hence it is extremely unlikely that m is a multiple of primes p or q.

Alice encrypts her plaintext using e and N to produce ciphertext *c* as follows:

$c = m^e \bmod N$ (encryption)

Alice transmits c to Bob. Bob decrypts the cyphertext using d to recover m from c:

$m = c^d \bmod N$ (decryption)

Example (continues): public key: N = 391, e = 197 private key: d = 109

When Bob get message from Alice (*151*), he will decrypt it:

6.9 The RSA cryptosystem

When Alice wants send a *plaintext* message *m* to Bob, the RSA scheme requires that:

- $m \in \mathbf{Z}_N$, and
- *m* is not a multiple of p or q.

p and q are primes with hundreds of digits, hence it is extremely unlikely that m is a multiple of primes p or q.

Alice encrypts her plaintext using e and N to produce ciphertext *c* as follows:

$c = m^e \bmod N$ (encryption)

Alice transmits c to Bob. Bob decrypts the cyphertext using d to recover m from c:

$m = c^d \bmod N$ (decryption)

Example (continues): public key: N = 391, e = 197 private key: d = 109

When Bob get message from Alice (151), he will decrypt it:

m = 151^{109} **mod** *391* - *here we will again recall our modular exponentiation*

m = 223 *(Section 6.7)*