Fundamentals of Cryptography

an interactive tutorial

EIDMA-Stieltjes week

22 September 2003

Leiden

Henk van Tilborg
Eindhoven University of Technology

1 Introduction

What are the reasons to use cryptographic algorithms?

- **Confidentiality (or Privacy)**

When transmitting data, one does not want an eavesdropper to understand the contents of the transmitted messages. The same is true for stored data that should be protected against unauthorized access, for instance by hackers.

- **Authentication**

This property is the equivalent of a signature. The receiver of a message wants proof that a message comes from a certain party and not from somebody else (even if the original party later wants to deny it).

- **Integrity**

This means that the receiver of certain data has evidence that no changes have been made by a third party.

2 Symmetric Systems

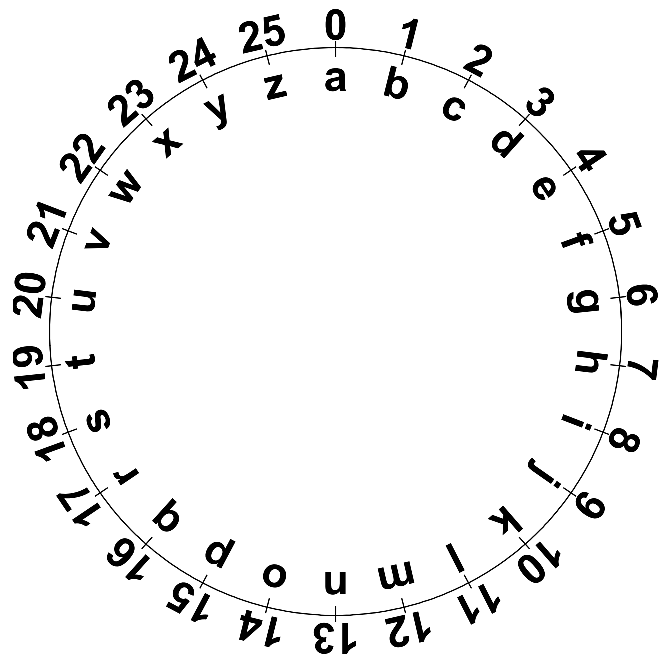
2.1 Classical Systems

2.1.1 Caesar Cipher

Shift each letter in the text cyclicly over *k* places. So, with *k* = 7 one gets the following encryption of the word cleopatra (note that the letter *z* is mapped to *a*):

cleopatra $\xrightarrow{+1}$ dmfpqbusb $\xrightarrow{+1}$ engqrcvtc $\xrightarrow{+1}$ fohrsdwud $\xrightarrow{+1}$ gpistexve $\xrightarrow{+1}$ hqjtufywf $\xrightarrow{+1}$ irkuvgzxg $\xrightarrow{+1}$ jslvwhayh

To do this in *Mathematica*, we need modular arithmetic (replace *a* by 0, *b* by 1,…,*z* by 25 and make your calculations modulo 26).



```
CaesarCipher[plaintext_, key_] :=     FromCharacterCode[
  Mod[ ToCharacterCode[plaintext] - 97 + key, 26] + 97]
```

```
plaintext = "cleopatraisanegyptianqueeen";
key = 7;
CaesarCipher[plaintext, key]
```

An easy way to break the system is to try out all possible keys. This method is called exhaustive key search.

The cryptanalysis of the ciphertext "xyuysuyifvyxi".

```
ciphertext = "xyuysuyifvyxi";
Table[{key, CaesarCipher[ciphertext, -key]}, {key, 0, 4}] //
 TableForm
```

So, the key k was $-4 \equiv 22 \pmod{26}$.

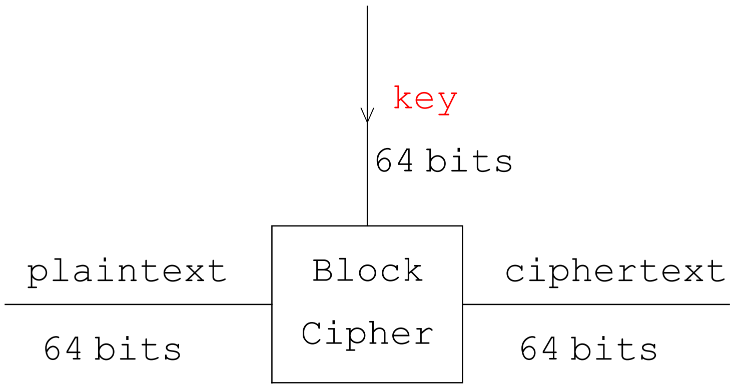
2.2 Block Ciphers

2.2.1 Some General Principles

Block ciphers handle n bits at a time (like $n = 64,\ 128$).

They have no memory (to store previous input).

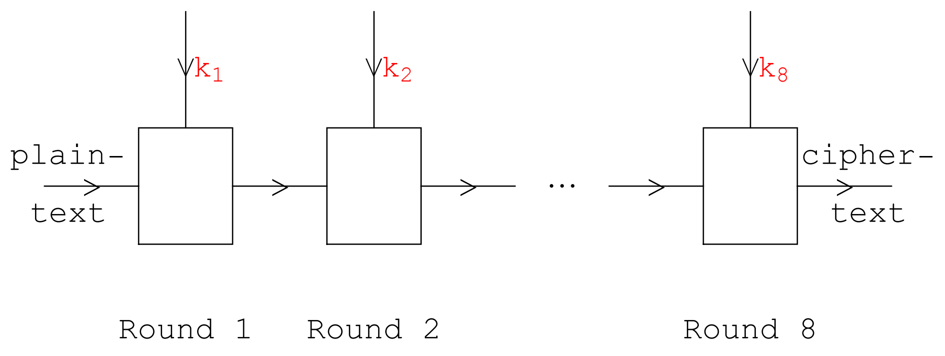
There can operate at very high speeds.



Often, the same device can be used for encryption and decryption.

Typically, the block cipher consists of a sequence of identical looking rounds each operating under a *round key* that is computed from the key k.

Each round is designed to realize "confusion" and "diffusion" in order to obscure dependencies and other statistical properties of the plaintext.



Note that the same plaintext will result in the same ciphertext as long as the key has not been changed. To avoid this situation feedback is introduced. Examples are given below.

2.2.2 Advanced Encryption Standard (AES), Rijndael

Like most modern block ciphers, Rijndael is an iterated block cipher: it specifies a transformation, also called the round function, and the number of times this function is iterated on the data block to be encrypted/decrypted, also referred to as the number of rounds. The block size is 128, 192 or 256.

The round function consists of the following operations:

- **ByteSub (affects individual bytes),**
- **ShiftRow (shifts rows),**
- **MixColumn (affects each column),**
- **RoundKey addition (overall XOR).**

These are applied to the intermediate cipher result, also called the State: a 4×4, 4×6, resp. 4×8 matrix of which the entries consist of 8 bits, i.e. one byte. Below the block length will be 192. One gets

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

where each $a_{i,j}$ consists of 8 bits, so it has the form $\{(a_{i,j})_0, (a_{i,j})_1, \ldots, (a_{i,j})_7\}$. For example, $a_{0,0} = \{1, 0, 1, 1, 0, 0, 0, 1\}$.

Sometimes, we use the one-dimensional ordering (columnwise) i.e.
$a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, \ldots, a_{3,5}$.

□ One Round

ByteSub

This is the only non-linear part in each round.

Apply to each byte $a_{i,j}$ two operations:

1) Interpret $a_{i,j}$ as element in GF(2^8) and replace it by its multiplicative inverse, if it is not 0, otherwise leave it the same.

2) Replace the resulting 8-tuple, say $(x_0, x_1, \ldots, x_7)$ by

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

The finite field GF(2^8) is made by means of the irreducible polynomial
$m(\alpha) = 1 + \alpha + \alpha^3 + \alpha^4 + \alpha^8$. This polynomial is not primitive!

Note that both operations are invertible.

Instead of performing these calculations, one can also replace them by one substitution table: the ByteSub S-box.

ShiftRow

The rows of the State are shifted cyclically to the left using different offsets: do not shift row 0, shift row 1 over c_1 bytes, row 2 over c_2 bytes, and row 3 over c_3 bytes, where

	c_1	c_2	c_3
128	1	2	3
192	1	2	3
256	1	3	4

.

So

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

becomes

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,0}$
$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,0}$	$a_{2,1}$
$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$

MixColumn

Interpret each column as a polynomial of degree 3 over GF(2^8) and multiply it with

$$(1+\alpha)\,x^3 + x^2 + x + \alpha$$

modulo $x^4 + 1$.

Note that the above polynomial is invertible modulo $x^4 + 1$.

```
<<Algebra`FiniteFields`
```

```
f128 = GF[2, {1, 1, 0, 1, 1, 0, 0, 0, 1}];
one = f128[{1, 0, 0, 0, 0, 0, 0, 0}]
α = f128[{0, 1, 0, 0, 0, 0, 0, 0}]
```

```
g[x_] = (1 + α) x^3 + one x^2 + one x + α
```

Suppose that the first column looks like

```
col = {α^100, α^255, α^200, α};
col // TableForm
```

```
colpol[x_] = col[[1]] + col[[2]] x + col[[3]] x^2 + col[[4]] x^3
```

```
pr[x_] = ownexpand[colpol[x] * g[x]]
prod[x_] = PolynomialMod[pr[x], x^4 - 1]
```

The inverse operation is a multiplication by

```
h[x_] = (1 + α + α^3) x^3 + (1 + α^2 + α^3) x^2 + (1 + α^3) x + (α + α^2 + α^3) ;
ownexpand[PolynomialMod[g[x] * h[x], x^4 - 1]]
```

```
ownexpand[PolynomialMod[prod[x] * h[x], x^4 - 1]]
```

Round Key Addition

XOR the whole matrix with a similar sized matrix (i.e. the Round Key) obtained from the cipher key in a way that depends on the round index.

Note that the XOR applied to a byte, really is an XOR applied to the 8 bits in the byte.

For example, if

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$\oplus$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$k_{0,4}$	$k_{0,5}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$	$k_{1,4}$	$k_{1,5}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$	$k_{2,4}$	$k_{2,5}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$k_{3,4}$	$k_{3,5}$

=

$u_{0,0}$	$u_{0,1}$	$u_{0,2}$	$u_{0,3}$	$u_{0,4}$	$u_{0,5}$
$u_{1,0}$	$u_{1,1}$	$u_{1,2}$	$u_{1,3}$	$u_{1,4}$	$u_{1,5}$
$u_{2,0}$	$u_{2,1}$	$u_{2,2}$	$u_{2,3}$	$u_{2,4}$	$u_{2,5}$
$u_{3,0}$	$u_{3,1}$	$u_{3,2}$	$u_{3,3}$	$u_{3,4}$	$u_{3,5}$

.

with $u_{0,0} = a_{0,0} \oplus k_{0,0}$, the coordinate-wise exclusive or.

```
a0,0 = {1, 1, 1, 1, 0, 0, 0, 0}; k0,0 = {1, 1, 0, 0, 1, 0, 1, 0};
Mod[a0,0 + k0,0, 2]
```

There is also an initial Round Key addition and one final round that differs slightly from the others (the MixColumn is omitted) .

3 The Principle of Public Key Cryptography

Alice and Bob want a method for encryption that does not involve a common secret key $k_{A,B}$ that has been agreed upon beforehand.

They also want techniques for authentication and integrity.

3.1 Setting It Up

Every participant P makes two matching algorithms:

- public algorithm PubAlgP,
- secret algorithm SecAlgP.

Because PubAlgP is public, it should not be possible to compute SecAlgP out of PubAlgP.

3.2 Encryption

Encryption of message m by Alice to Bob.

Alice sends:

$$c = \text{PubAlgBob}(m).$$

Bob decrypts c as follows:

$$\text{SecAlgBob}(c) = \text{SecAlgBob}(\text{PubAlgBob}(m)) = m.$$

Question:
Is it possible that the plaintext remains secret while you know the ciphertext and how is was encrypted?

It should be against your intuition that this works. If you know c and PubAlgBob then you should be able to find m. In the worst case, you can find m by encrypting out all possible plaintexts until you will find c.

Computationally, the above system is possible! The number of possibilities to check should be too much even when many computers work together for many hours.

4 Mathematical Principles

4.1 Different Complexities

Exponentiation goes very easy:

To compute 6^{4371} (mod 99991), we make the following table

6^1	6
6^2	36
6^4	1296
6^8	79760
6^{16}	30198
6^{32}	1284
6^{64}	48800
6^{128}	54344
6^{256}	36151
6^{512}	12431
6^{1024}	43666
6^{2048}	91168
6^{4096}	52331

So, $6^{4371} \equiv 6^{4096}\, 6^{256}\, 6^{16}\, 6^2\, 6^1 \equiv 52331 \times 36151 \times 30198 \times 36 \times 6 \equiv 34455 \pmod{99991}$.

The answer 34455 can be checked with the Mod function.

```
Mod[52331 * 36151 * 30198 * 36 * 6, 99991]
Mod[6^4371, 99991]
```

Faster is the PowerMod function which uses the method above.

```
PowerMod[6, 4371, 99991]
```

So, for a fast exponentiation one uses the binary expansion of the exponent to compute 6^{4371} .

The binary expansion of the exponent can be found with the IntegerDigits function.

```
IntegerDigits[4371, 2]
```

From this we can also compute 6^{4371} on the fly as follows (all calculations are modulo 99991):

$$((((((((((((1*6)^2)^2)^2)^2\,6)^2)^2)^2)^2\,6)^2)^2)^2\,6)^2\,6$$

A more formal verification:

$$((((((((((((1*x)^2)^2)^2)^2\,x)^2)^2)^2)^2\,x)^2)^2)^2\,x)^2\,x$$

The Paul Kocher's *timing attack* is based on the difference in time needed for taking the square and for multiplying.

The "opposite" operations of exponentiation are intractable for large moduli (unless they have a special form).

a) Determine e such that $6^e \equiv 34455 \pmod{99991}$.

Since e can be written as $\log_6 34455$, the problem of computing e is called the logarithm problem.

This observation is at the base of the Diffie-Hellman system.

Compare the difference in the following two plots:

```
p = 541;
Plot[Log[x], {x, 1, p}]
ListPlot[Table[{j, Mod[2^j, p]}, {j, 1, p}]]
```

b) Determine m such that $m^{4371} \equiv 34455 \pmod{99991}$.

Take the 4371th modular root of 34455.

This observation is at the base of the RSA system.

Compare the difference in the following two plots:

```
n = 541;
Plot[x^(1/3), {x, 1, n}]
ListPlot[Table[{j, Mod[j^3, n]}, {j, 1, n}]]
```

What is the complexity of a modular exponentiation? The table above has length $\log_2 m$. Since $m < p$, this length is at most $\log_2 p$. So, it takes at most $\log_2 p$ multiplications to make the table.

In the worst case, all these elements have to be multiplied. This takes another $\log_2 p$ multiplications. The overal complexity is $2 \log_2 p$.

> Theorem 4.1
> The complexity of an exponentiation in $\mathbb{Z}_p^*$ is at most $2 \log_2 p$.

5 Discrete Logarithm Based Systems

5.1 The Diffie-Hellman Key-Exchange System

5.1.1 Setting It Up

Consider the prime number $p = 99991$ and take $g = 6$.

The multiplicative order of 6 is 99990. This means the following:

The number 6 has the property that all powers 1, 6, 6^2, 6^3, …, 6^{99989} are different modulo 99991 and that $6^{99990} \equiv 1 \pmod{99991}$.

```
PowerMod[6, 99990, 99991]
```

```
FactorInteger[99990]
```

```
PowerMod[6, 99990 / 2, 99991]
PowerMod[6, 99990 / 3, 99991]
PowerMod[6, 99990 / 5, 99991]
PowerMod[6, 99990 / 11, 99991]
PowerMod[6, 99990 / 101, 99991]
```

Such an element is called a generator of $\mathbb{Z}_p^*$ or primitive element in $\mathbb{Z}_p^*$.

Alice chooses as random secret exponent SAlice = 12345 and Bob as random secret exponent SBob = 11111.

Next, Alice and Bob compute their public key:

$$6^{12345} \pmod{99991}, \qquad \text{resp.} \qquad 6^{11111} \pmod{99991}.$$

```
PAlice = PowerMod[6, 12345, 99991]
PBob = PowerMod[6, 11111, 99991]
```

So, PAlice=33190 and PBob=61056.

These public keys are made public by them.

5.1.2 The Key Determination

Alice can compute the common key $k_{A,B}$ (with Bob) by raising the publicly known PBob=61056 of Bob to the power SAlice=12345, which she only knows. She gets:

```
PowerMod[61056, 12345, 99991]
```

Bob gets the same common key $k_{A,B}$ by raising PAlice=33190 of Alice to the power SBob. Indeed, he gets:

```
PowerMod[33190, 11111, 99991]
```

Note that

$$\text{PBob}^{\text{SAlice}} = (6^{\text{SBob}})^{\text{SAlice}} = 6^{\text{SBob}\times\text{SAlice}}$$

just as

$$\text{PAlice}^{\text{SBob}} = (6^{\text{SAlice}})^{\text{SBob}} = 6^{\text{SAlice}\times\text{SBob}}$$

Remember from high school that

$$(g^m)^n = \overbrace{g^m\, g^m \ldots g^m}^{n} = g^{\overbrace{m+m+\ldots+m}^{n}} = g^{m\times n} = (g^n)^m .$$

If Eve can find SAlice from PAlice, she can also determine $k_{A,B}$, just like Bob did. To this end, she has to solve the logarithm problem

$$6^{??} \equiv 33190 \pmod{99991}.$$

In Sept. 2001, a logarithm in GF(2^{521}) was determined.

```
N[2^521, 10]
```

Time involved: sieving 21 days, linear algebra 10 days, final step 12 hours.

5.2 How to Take Discrete Logarithms

5.2.1 Exhaustive Search

Try SBob = 1, 2, 3, … until you find that $g^{\text{SBob}} \equiv$ PBob (mod p).

Complexity is p.

5.2.2 Baby-step Giant-step

Complexity of the baby-step giant-step method is p^{α} in computer time and $p^{1-\alpha}$ in memory space, where α can be chosen freely in between 0 and 1.

EXAMPLE : α=4/5

Consider the equation $6^m \equiv 55555 \pmod{99991}$ and assume that we can only store a table with 10 field elements.

We make a table of $6^i \pmod{99991}$ for $i = 0, 1, \ldots, 9$.

```
powers = Table[PowerMod[6, i, 99991], {i, 0, 9}]
```

This gives the table:

i	0	1	2	3	4	5	6	7	8	9
6^i	1	6	36	216	1296	7776	46656	79954	79760	78596

Now note that either

$0 \leq m \leq 9$ or

$0 \leq m - 10 \leq 9$or

$0 \leq m - 2.10 \leq 9$ or

$0 \leq m - 3.10 \leq 9$ or

etc.

The idea is to make (giant) steps of size 10 by checking if 55555, $55555 / 6^{10}$, $55555 / 6^{2.10}$,… is in the table.

Instead of having to divide by 6^{10}, we rather multiply by something else. We use the PowerMod function to find 6^{-10} mod 99991.

```
PowerMod[6, -10, 99991]
```

This means that $1 / 6^{10} \equiv 12339 \pmod{99991}$, i.e. dividing by 6^{10} amounts to the same as multiplying by 12339 modulo 99991

So, we check if 55555, 55555×12339, 55555×12339^2, 55555×12339^3,… is in the table above. We use the function MemberQ.

```
try = Mod[55555 * 12339^1, 99991]
MemberQ[powers, try]
```

We better make a program.

```
j = 0; try = Mod[55555, 99991];
While[Not[MemberQ[powers, try]],
  j = j + 1; try = Mod[try * 12339, 99991]];
j
try
```

We conclude that $j = 7972$. Indeed, 1296 is the element in the table above corresponding to $i = 4$.

This means that $m - 7972 \times 10 = 4$, i.e. $m = 79724$.

Indeed, $6^{79724} \equiv 55555$ mod 99991, as can be easily checked with:

```
PowerMod[6, 79724, 99991]
```

Note that the maximum number of tries is $\lceil 99991/10 \rceil = 1000 \approx p^{4/5}$ and that the length of the table is $10 \approx p^{1/5}$.

5.2.3 Pollard-ϱ

The time complexity of the Pollard-ρ method is the same as that of the Baby-Step-Giant-Step, so $\sqrt{p}$.

The advantage lies in the minimal memory requirements.

□ The Method

We shall explain the Pollard-ρ method for the special case of a multiplicative subgroup G of GF(p) of prime order. So, we want to solve m, $0 \leq m < q$, from the equation $c = g^m$, where $g \in$ GF(p) has order q, q prime, and where $c \in$ GF(q) is some given q-th root of unity.

EXAMPLE (Part I): $121^m \equiv 3435 \pmod{4679}$

We consider $p = 4679$. Note that $p - 1 = 2 \times 2339$. The number 11 is a primitive element of GF(4679) and thus $g = 11^{(q-1)/2339} = 11^2 = 121$ is the generator of a multiplicative subgroup of order 2339.

```
p = 4679; PrimeQ[p]
MultiplicativeOrder[11, p]
MultiplicativeOrder[121, p]
```

We want to solve the equation

$$121^m \equiv 3435 \pmod{4679}.$$

Note that this equation must have a solution, since 3435 is indeed a 2339-th root of unity in GF(4679).

Indeed, all 2339-th roots of unity are a zero of $x^{2339} - 1$ and this polynomial can not have more zeros.

```
PowerMod[3435, 2339, 4679]
```

In order to solve $c = g^m$, we partition the multiplicative subgroup G of GF(p) of order q, in three subsets G_i, $i = 0, 1, 2$, as follows:

$$x \in G_i \qquad \Longleftrightarrow \qquad x \equiv i \pmod 3.$$

So, $G_0 = \{0, 3, \ldots\}$, $G_1 = \{1, 4, \ldots\}$, and $G_2 = \{2, 5, \ldots\}$.

We define a sequence $\{x_i\}_{i \geq 0}$ in GF(p) recursively by $x_0 = 1$ and

$$x_{i+1} = f(x_i) = \begin{cases} x_i^2 \pmod p, & \text{if } x_i \in G_0, \\ c.x_i \pmod p, & \text{if } x_i \in G_1, \\ g.x_i \pmod p, & \text{if } x_i \in G_2. \end{cases} \tag{5.1}$$

Clearly, the sequence $\{x_i\}_{i \geq 0}$ will eventually cycle. This behaviour explains the name ρ-method.

With the sequence $\{x_i\}_{i \geq 0}$ we associate two other sequences $\{a_i\}_{i \geq 0}$ and $\{b_i\}_{i \geq 0}$ in such a way that for all $i \geq 0$

$$x_i = g^{a_i} c^{b_i}.$$

To this end, take $a_0 = b_0 = 0$ and use the recursions

$$a_{i+1} = \begin{cases} 2\,a_i \pmod q, & \text{if } x_i \in G_0, \\ a_i, & \text{if } x_i \in G_1, \\ a_i + 1 \pmod q, & \text{if } x_i \in G_2. \end{cases}$$

$$b_{i+1} = \begin{cases} 2\,b_i \pmod q, & \text{if } x_i \in G_0, \\ b_i + 1 \pmod q, & \text{if } x_i \in G_1, \\ b_i, & \text{if } x_i \in G_2. \end{cases}$$

Note that by induction

$$\begin{aligned} x_{i+1} &= x_i^2 = (g^{a_i} c^{b_i})^2 = g^{2\,a_i} c^{2\,b_i} = g^{a_{i+1}} c^{b_{i+1}}, \text{ if } x_i \in G_0, \\ x_{i+1} &= c.x_i = c.g^{a_i} c^{b_i} = g^{a_i} c^{b_i+1} = g^{a_{i+1}} c^{b_{i+1}}, \text{ if } x_i \in G_1, \\ x_{i+1} &= g.x_i = g.g^{a_i} c^{b_i} = g^{a_i+1} c^{b_i} = g^{a_{i+1}} c^{b_{i+1}}, \text{ if } x_i \in G_2. \end{aligned}$$

As soon as we have two distinct indices i and j with $x_i = x_j$ we are done.

Indeed, if $g^{a_i} c^{b_i} = g^{a_j} c^{b_j}$, $i < j$, then $g^{a_i - a_j} = c^{b_j - b_i}$.

Provided that $b_i \neq b_j$, we have found the solution

$$m \equiv (a_j - a_i) / (b_i - b_j) \pmod q.$$

If $b_i = b_j$ (with negligible probability), put $c' = c.g$ and solve $c' = g^{m'}$, where $m' = m + 1$.

To find indices i and j with $x_i = x_j$, we follow Floyd's cycle-finding algorithm: find an index i such that $x_i = x_{2i}$ (so, take $j = 2i$).

To this end, we start with the pair (x_1, x_2), calculate (x_2, x_4), then (x_3, x_6), and so on, each time calculating (x_{i+1}, x_{2i+2}) from the previously calculated (x_i, x_{2i}) by the defining rules

$$x_{i+1} = f(x_i),$$

$$x_{2i+2} = f(f(x_{2i})).$$

In this way, huge storage requirements can be avoided.

EXAMPLE (Part II): $121^m \equiv 3435 \pmod{4679}$

We want to solve the equation:

$$121^m \equiv 3435 \pmod{4679}.$$

The recurrence relation for the $\{x_i\}_{i\geq 0}$ sequence can be evaluated by means of the `Which` *and* `Mod` *functions.*

```
RecX[x_, g_, c_, p_] := Which[ Mod[x, 3] == 0, Mod[x^2, p],
  Mod[x, 3] == 1, Mod[c * x, p], Mod[x, 3] == 2, Mod[g * x, p] ]
```

The smallest index i, $i \geq 1$, satisfying $x_i = x_{2i}$ can be found with the help of the `While` *function.*

```
g = 121; c = 3435; p = 4679;
x1 = RecX[1, g, c, p];
x2 = RecX[x1, g, c, p];
i = 1;
While[x1 != x2, x1 = RecX[x1, g, c, p];
  x2 = RecX[RecX[x2, g, c, p], g, c, p]; i = i + 1];
i
```

So, $x_{76} = x_{152}$ and $m \equiv (a_{152} - a_{76}) / (b_{76} - b_{152}) \pmod{2339}$. However, above we did not update the values of the sequences a_i and b_i. We will do that now.

```
RecurrDef[{x_, a_, b_}] := Which[Mod[x, 3] == 0,
  {Mod[x^2, p], Mod[2 a, q], Mod[2 b, q]},
  Mod[x, 3] == 1, {Mod[c * x, p] , a, Mod[b + 1, q]},
  Mod[x, 3] == 2, {Mod[g * x, p], Mod[a + 1, q], b}]
```

```
g = 121; c = 3435; p = 4679; q = 2339;
x1 = 1; a1 = 0; b1 = 0;
x2 = 1; a2 = 0; b2 = 0;
{x1, a1, b1} = RecurrDef[{x1, a1, b1}]; i = 1;
{x2, a2, b2} = RecurrDef[RecurrDef[{x2, a2, b2}]];
While[x1 != x2, {x1, a1, b1} = RecurrDef[{x1, a1, b1}];
  {x2, a2, b2} = RecurrDef[RecurrDef[{x2, a2, b2}]]; i = i + 1];
Print["i=", i]
Print["xi=", x1, ", ai=", a1, ", bi=", b1];
Print["x2i=", x2, ", a2i=", a2, ", b2i=", b2];
```

Indeed, the relation $\alpha^{a_i} c^{b_i}$ gives the same value for $i = 76$ and $i = 2 \times 76$:

```
Mod[PowerMod[g, a1, p] * PowerMod[c, b1, p], p]
Mod[PowerMod[g, a2, p] * PowerMod[c, b2, p], p]
```

The solution m of $121^m \equiv 3435 \ (mod\ 4679)$ can now be determined from
$m \equiv (286 - 84) / (2191 - 915) \ (mod\ 2339)$.

```
m = Mod[(a2 - a1) * PowerMod[b1 - b2, -1, q], q]
```

That $m = 1111$ is indeed the solution can be checked with

```
PowerMod[g, 1111, p] == c
```

The ρ in the name of this algorithm reflects the shape of the $\{x_i\}_{i \geq 0}$-sequence: after a while it starts cycling around. The memory requirements of Floyd's cycle finding algorithm are indeed minimal. The expected running time is $\sqrt{q}$.

5.2.4 Pohlig-Hellman

Complexity of the **Pohlig-Hellman** method depends on factorization of $p-1$. It can be much faster than $\sqrt{p}$ operations, if $p-1$ has only small prime divisors. but is, in general, still exponential in behaviour.

□ Special Case: $p-1=2^n$

Examples of prime numbers that are a power of 2 plus one are given by $p=17$, $p=257$, and $p=2^{16}+1$.

```
n = 16; PrimeQ[2^n + 1]
```

So, let g be a generator (primitive element) in $\mathbb{Z}_p^*$. The problem is to find m, $0 \le m \le q-2$, satisfying

$$g^m \equiv c \,(\text{mod } p)$$

for given value of c.

Let $m_0, m_1, \ldots, m_{n-1}$ be the binary representation of the unknown m, i.e.

$$m = m_0 + m_1 .2 + \ldots + m_{n-1} .2^{n-1}, \qquad m_i \in \{0, 1\}, \quad 0 \le i \le n-1.$$

Of course, it suffices to compute the unknown m_i's. Since g is a generator of $\mathbb{Z}_p^*$ we know that

$$g^{p-1} \equiv 1 \,(\text{mod } p) \qquad \text{and} \qquad g^i \not\equiv 1 \,(\text{mod } p) \text{ for } 0 < i < p-1.$$

It also follows that $g^{(p-1)/2} \equiv -1 \,(\text{mod } p)$, because

- the square of $g^{(p-1)/2}$ is 1,
- $g^{(p-1)/2} \not\equiv 1 \,(\text{mod } p)$.

We also use here that the quadratic equation $x^2 \equiv 1 \,(\text{mod } p)$ has ± 1 as only roots. Hence

$$c^{(p-1)/2} \equiv (g^m)^{(p-1)/2} \equiv g^{m(p-1)/2} \equiv g^{(m_0 + m_1 .2 + \ldots + m_{n-1} .2^{n-1})(p-1)/2}$$

$$\overset{g\text{ prim.}}{\equiv} \quad g^{m_0 (p-1)/2} \equiv \begin{cases} +1, & \text{if } m_0 = 0, \\ -1, & \text{if } m_0 = 1. \end{cases}$$

Therefore, the evaluation of $c^{(p-1)/2}$ modulo p, which takes at most 2. $\lceil \log_2 p \rceil$ multiplications, as we have seen in Section 4.4), yields m_0.

Compute $c_1 = c.g^{-m_0} = g^{m_1 .2 + m_2 .2^2 + \ldots + m_{n-1} .2^{n-1}}$.

Now m_1 can be determined in the same way as above from

$$\begin{aligned} c_1^{(p-1)/4} &\equiv g^{(m_1 2 + m_2 .2^2 + \ldots + m_{n-1} .2^{n-1})(p-1)/4} \\ &\equiv g^{m_1 (p-1)/2} \equiv \begin{cases} 1, & \text{if } m_1 = 0, \\ -1, & \text{if } m_1 = 1. \end{cases} \end{aligned}$$

Compute $c_2 = c_1.g^{-2 m_1} = c.g^{-(m_0 + m_1 .2)}$ and determine m_2 from $(c_2)^{(p-1)/8}$. Repeat this process until also m_{n-1} (and thus m) has been determined.

EXAMPLE

Consider the equation $3^m \equiv 7 \bmod 17$. So, $p = 17$, $g = 3$, and $c = 7$. Note that $g^{-1} = 6$.

Writing $m = m_0 + 2\, m_1 + 4\, m_2 + 8\, m_3$, we find m_0 by evaluating $c^{(p-1)/2}$ (modulo p).

```
PowerMod[7, 8, 17]
```

Since this is -1 we know that $m_0 = 1$. Compute $c_1 \equiv c / 3 \equiv 6. c \equiv 8 \bmod 17$. Then m_1 can be found from $c_1{}^{(p-1)/4}$ (modulo p):

```
PowerMod[8, 4, 17]
```

Again this is -1, so $m_1 = 1$. Compute $c_2 \equiv c_1 / 3^2 \equiv 6^2 . c_1 \equiv 16 \bmod 17$. Then m_2 can be found from $c_2{}^{(p-1)/8}$ (modulo p):

```
PowerMod[16, 2, 17]
```

Since the outcome is 1, we have $m_2 = 0$. So, $c_3 = c_2$ and m_3 can be found from $c_3{}^{(p-1)/16}$ (modulo p):

```
PowerMod[16, 1, 17]
```

We now also have $m_3 = 1$ and thus $m = 1.2^0 + 1.2^1 + 0.2^2 + 1.2^3 = 11$. We can check this with:

```
PowerMod[3, 11, 17]
```

The above algorithm finds m from c in at most

$$n.(2.\lceil \log_2 p \rceil + 2) \approx 2.(\log_2 p)^2 \approx 2\,n^2,$$

operations, where the term +2 comes from the evaluation of the c_i's (one squaring and possibly one multiplication to compute c_{i-1}) and the n from the number of exponentiations.

Comparing with the complexity of an exponentiation (see Theorem 4.3), we observe that for $p = 2^n + 1$,

- using the Diffie-Hellman scheme takes $2\,n$ multiplications

- breaking it takes $\approx 2\,n^2$ multiplications

A quadratic relation, which is not significant enough to make the system secure.

The method above can be generalized to prime numbers p with the property that $p-1$ only contains small primefactors.

5.2.5 The Index-Calculus Method

Complexity of the index-calculus method is subexponential!

Depending on the implementation it may look like $e^{\sqrt{\ln n \ln\ln n}}$, where n is the size of the multiplicative group.

□ GF(p)

EXAMPLE: p=99991, g=6

Consider $\mathbb{Z}^*_{99991}$ with generator $g = 6$ and say that we want to solve

$$6^m \equiv 55555 \pmod{99991}$$

As factor base S we take the set of prime numbers {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}.

Next, we try to write all the elements in the factor base as powers of 6 mod 99991, i.e. we need to solve the logarithm problem for all the elements in the factor base.

We achieve this by finding powers of 6 that reduced modulo 99991 can be expressed as product of elements in {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}.

A number with this property is called smooth with respect to this factorbase.

(Here, we shall use the function `FactorInteger`, but that does function does much more than is needed.)

```
p = 99991;
try = PowerMod[6, 812, p]
FactorInteger[try]
```

After some trial and error we find the following five succesful attempts.

```
FactorInteger[PowerMod[6, 219, p]]
FactorInteger[PowerMod[6, 813, p]]
FactorInteger[PowerMod[6, 2150, p]]
FactorInteger[PowerMod[6, 2151, p]]
FactorInteger[PowerMod[6, 7003, p]]
FactorInteger[PowerMod[6, 10028, p]]
FactorInteger[PowerMod[6, 12067, p]]
FactorInteger[PowerMod[6, 20019, p]]
FactorInteger[PowerMod[6, 30042, p]]
FactorInteger[PowerMod[6, 30057, p]]
```

Write

$$\begin{array}{ll}
2 \equiv 6^{m_1} \pmod{99991}, & 3 \equiv 6^{m_2} \pmod{99991}, \\
5 \equiv 6^{m_3} \pmod{99991}, & 7 \equiv 6^{m_4} \pmod{99991}, \\
11 \equiv 6^{m_5} \pmod{99991} & 13 \equiv 6^{m_6} \pmod{99991}, \\
17 \equiv 6^{m_7} \pmod{99991}, & 19 \equiv 6^{m_8} \pmod{99991}, \\
23 \equiv 6^{m_9} \pmod{99991} & 29 \equiv 6^{m_{10}} \pmod{99991}.
\end{array}$$

We get ten congruence relations modulo 99990.

For example, $6^{813} \equiv 17986 \equiv 2^1 .17^1 .23^2 \pmod{99991}$ can be rewritten as

$$6^{813} \equiv (6^{m_1})^1.(6^{m_7})^1.(6^{m_9})^2 \equiv 6^{m_1+m_7+2m_9} \pmod{99991}.$$

Comparing the exponents on both sides gives the relation

$$813 \equiv m_1 + m_7 + 2\, m_9 \pmod{99990}.$$

The ten congruence relations modulo 99990 that we get are

$$\begin{aligned}
219 &\equiv m_4 + m_6 + m_8 + m_9 \pmod{99990},\\
813 &\equiv m_1 + m_7 + 2\,m_9 \pmod{99990},\\
2150 &\equiv m_3 + m_8 + m_{10} \pmod{99990},\\
2151 &\equiv m_1 + m_2 + m_3 + m_8 + m_{10} \pmod{99990},\\
7003 &\equiv 3\,m_2 + m_4 + 2\,m_7 \pmod{99990},\\
10028 &\equiv 3\,m_1 + m_4 + m_8 + m_{10} \pmod{99990},\\
12067 &\equiv 2\,m_4 + m_6 + m_8 \pmod{99990},\\
20019 &\equiv m_1 + 2\,m_3 + m_6 + m_{10} \pmod{99990},\\
30042 &\equiv 6\,m_1 + 2\,m_9 \pmod{99990},\\
30057 &\equiv 3\,m_1 + 2\,m_5 + m_6 \pmod{99990}.
\end{aligned}$$

The above system of relations can now easily be solved:

```
Solve[{m4 + m6 + m8 + m9 == 219 ,
  m1 + m7 + 2 * m9 == 813, m3 + m8 + m10 == 2150,
  m1 + m2 + m3 + m8 + m10 == 2151, 3 m2 + m4 + 2 * m7 == 7003,
  3 * m1 + m4 + m8 + m10 == 10028, 2 * m4 + m6 + m8 == 12067,
  m1 + 2 * m3 + m6 + m10 == 20019, 6 * m1 + 2 * m9 == 30042,
  3 * m1 + 2 * m5 + m6 == 30057, Modulus == p - 1},
    {m1, m2, m3, m4, m5, m6, m7, m8, m9, m10}]
```

So, we know that

$$\begin{array}{ll}
2 \equiv 6^{22146} \pmod{99991}, & 3 \equiv 6^{77845} \pmod{99991},\\
5 \equiv 6^{68986} \pmod{99991}, & 7 \equiv 6^{10426} \pmod{99991},\\
11 \equiv 6^{77314} \pmod{99991} & 13 \equiv 6^{8971} \pmod{99991}\\
17 \equiv 6^{81501} \pmod{99991} & 19 \equiv 6^{82234} \pmod{99991}\\
23 \equiv 6^{98568} \pmod{99991} & 29 \equiv 6^{50910} \pmod{99991}.
\end{array}$$

Let us now find a solution of $6^m \equiv 55555 \pmod{99991}$.

From

```
FactorInteger[55555]
FactorInteger[Mod[6^13 55555, 99991]]
```

we see that 55555 can not be expressed as product of elements of S, but

$$6^{13} \times 55555 \equiv 3^5\, 13^1\, 17^1 \equiv (6^{m_2})^5\, (6^{m_6})^1\, (6^{m_7})^1 \equiv 6^{5\,m_2}\, 6^{m_6}\, 6^{m_7} \pmod{99991}.$$

Since we are trying to solve $6^m \equiv 55555 \pmod{99991}$ we conclude that

$$6^{13+m} \equiv 6^{5\, m_2 + m_6 + m_7} \pmod{99991}.$$

$$13 + m \equiv 5.\, m_2 + m_6 + m_7 \equiv 5 \times 77845 + 8971 + 81501 \pmod{99990}.$$

We conclude that m is given by

```
Mod[5 * 77845 + 8971 + 81501 - 13, p - 1]
```

So

$$m \equiv 79724 \pmod{99990}.$$

This can easily be checked with

```
PowerMod[6, 79724, 99991]
```

6 Elliptic Curve Based Systems

6.1 The Definition of an Elliptic Curve

Elliptic curves are defined by the so-called Weierstrass equation:

$$y^2 + u.x.y + v.y = x^3 + a.x^2 + b.x + c. \tag{6.1}$$

The coefficients will be in $\mathbb{Z}_p$ or in $GF(2^m)$. Here we only consider the $\mathbb{Z}_p$ case.

For $p \geqslant 5$ one can simplify this equation by means of elementary transformations:

$$y^2 = x^3 + a.x + b. \tag{6.2}$$

Definition 6.1
An elliptic curve $\mathcal{E}$ over $GF(q)$ is defined as the set of points (x, y) satisfying (7.1) together we single element O, called the point at infinity.

```
<< Graphics`ImplicitPlot`
```

```
elliptic = ImplicitPlot[y^2 == x^3 - 5 x + 3, {x, -3, 3}]
```

Substitute a random value for x in

$$y^2 = x^3 + a.x + b.$$

On the average, half of time the right hand side will be a quadratic residue (=perfect square) modulo p, leading to two values for y.

```
p = 31;
Solve[ {y^2 == x^3 - 5 x + 3, x == 3, Modulus == p}, {y}]
```

6.2 Lines Intersecting Elliptic Curves

Consider two points on an elliptic curve, say $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, with different x-coordinates. Let $y = u.x + v$ be the line through them.

Substitute $y = u.x + v$ in

$$y^2 = x^3 + a.x + b.$$

One gets a third degree equation in x:

$$(u.x + v)^2 = x^3 + a.x + b$$

This third degree equation has x_1 and x_2 as roots, i.e. it contains the factors $x - x_1$ and $x - x_2$.

So, there must be a third root x_3. It can easily be computed by comparing the coefficient of x^2 in:

$$x^3 + a.x + b - (u.x + v)^2 = (x - x_1)(x - x_2)(x - x_3).$$

Compute $y_3 = u.x_3 + v$. Then (x_3, y_3) is also on the curve.

Conclusion: the line through (x_1, y_1) and (x_2, y_2) will intersect the curve in a third point!

```
Block[{$DisplayFunction = Identity},
elliptic = ImplicitPlot[ y^2 == x^3 - 5 x - 3, {x, -3, 4}];
linea = Plot[ x + 2, {x, -3, 4}]];
Show[linea, elliptic]
```

```
NSolve[ {y^2 == x^3 - 5 x - 3, y == x + 2}, {x, y}]
```

The same is true for tangent lines. They will also intersect the curve in another point (except when it is a double tangent).

```
Block[{$DisplayFunction = Identity},
elliptic = ImplicitPlot[ y^2 == x^3 - 5 x - 3, {x, -3, 4}];
linea = Plot[- x, {x, -3, 4}];];
Show[linea, elliptic]
```

Also modulo a prime number, lines through two points of an elliptic curve will intersect it in a third point.

```
Solve[ {y^2 == x^3 - 5 x + 3, y == x - 5, Modulus == 11}, {x, y}]
```

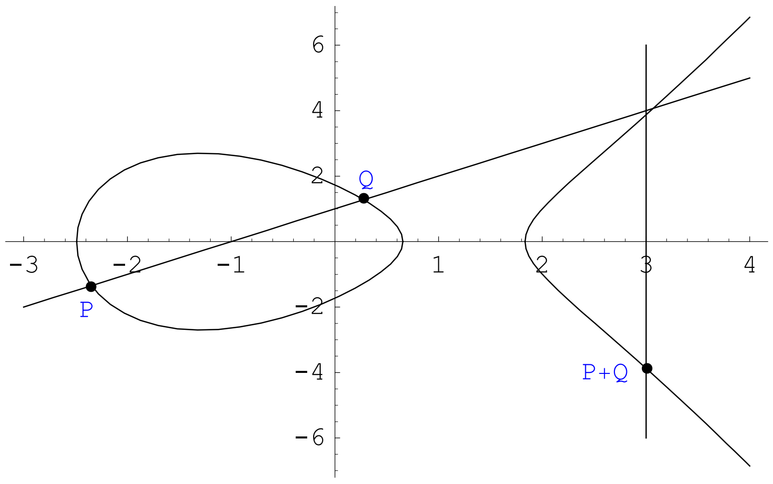
6.3 Adding Two Points on the Curve

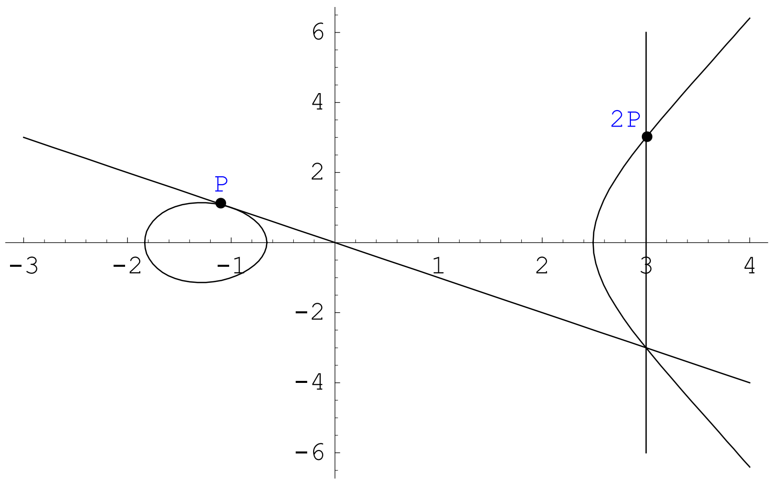
We are now ready to define an addition on $\mathcal{E}$.

To add points P_1 and P_2, both not at infinity, execute the following two steps:

1) Compute the line $\mathcal{L}$ through P_1 and P_2 (or tangent line though P_1, if $P_1 = P_2$) and find the third point of intersection with $\mathcal{E}$. Let this be Q.

2) The sum $P_1 + P_2$ is defined as $P_3 := -Q$.



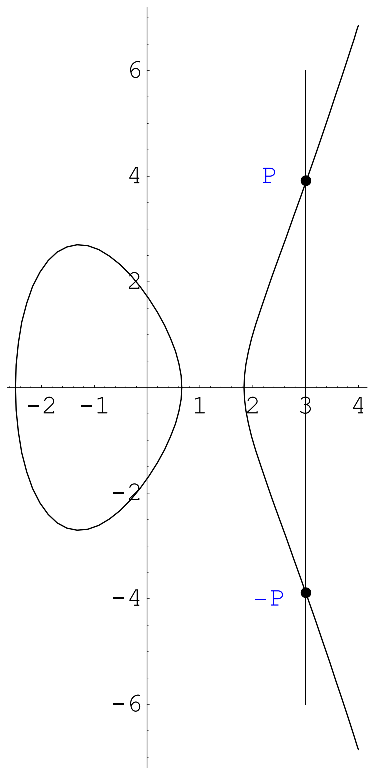


Interpret the point O at infinity as the intersection point of all vertical lines.

Consistent with the above definition of addition we get

$$O + P = P + O = P,$$

if $P = (x, y)$, then $-P = (x, -y)$.



With this addition we have a group structure on $\mathcal{E}$. It is in general non-trivial to determine the order of this group. We quote:

Theorem 6.1 *Hasse*
Let N be the number of points on an elliptic curve $\mathcal{E}$ over GF(q). Then

$$|N-(q+1)| \leq 2\sqrt{q}$$

Theorem 6.2
The additive group of an elliptic curve $\mathcal{E}$ over GF(q) is isomorphic to

$$\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2},$$

where n_2 divides both n_1 and (q-1) and where n_2 can be 1.

6.4 The Logarithm System on Elliptic Curves

6.4.1 The Discrete Logaritm Problem over Elliptic Curves

We have just seen how to to add points on an elliptic curve $\mathcal{E}$. This is an operation with relatively low complexity. To compute scalar multiples of a point P

$$\overbrace{P+P+\cdots+P}^{n}$$

for some integer n, we can copy the ideas of Section 4.4.

EXAMPLE: n=171

The binary expansion of 171 is given by

```
IntegerDigits[171, 2]
```

So, to compute 171 P, it suffices to compute

$$\begin{aligned}
&2P = P + P,\\
&4P = 2P + 2P,\\
&8P = 4P + 4P,\\
&\vdots\\
&64P = 32P + 32P,\\
&128P = 64P + 64P
\end{aligned}$$

and add the suitable terms. This can be done on the fly as follows:

```
P =.;
2 (2 (2 (2 (2 (2 (2 P) + P)) + P)) + P) + P
```

Note that we only needed:

- addition of a point to itself
- addition of P to a point.

The opposite problem is much harder.

Definition 6.2
Let $\mathcal{E}$ be an elliptic curve over GF(q). Let P be a point on $\mathcal{E}$ and let Q be a scalar multiple of P.
The discrete logarithm problem over an elliptic curve is to determine the solution n of

$$n.P = Q.$$

Remember that the (additive) *order* of a point P is defined as the smallest positive integer m such that $m\,P = O$.

It turns out that all the methods to solve the discrete logarithm problem over elliptic curves have a complexity of the form m^{α}, for some $\alpha > 0$.

So, they are exponentially slower than the (logarithmic) complexity of computing scalar multiples of P.

6.4.2 The Diffie-Hellman System over Elliptic Curves

As system parameters one needs

- an elliptic curve $\mathcal{E}$ over a finite field GF(q),
- a point P on $\mathcal{E}$ of high order.

To

Each user U of the system, selects a secret scalar m_U, computes the point $Q_U = m_U P$ and makes Q_U public.

Alice and Bob can now agree on the common key

$$K_{A,B} = m_A m_B P.$$

Alice can find this common key by computing $m_A Q_B$ with her secret scalar m_A and Bob's public Q_B.

Bob can do likewise.

This system is summarized in the following table.

system parameters	elliptic curve $\mathcal{E}$ P of high order
secret key of U	m_U
public point of U	$Q_U = m_U P$
common key of A and B	$K_{A,B} = m_A m_B P$
Alice computes	$m_A Q_B$
Bob computes	$m_B Q_A$

The Diffie-Hellman Key Exchange System
over Elliptic Curves

Table 6.1

At the time of this writing, it is advised to take the order of P about 150-180 digits long.

EXAMPLE

Consider the elliptic curve $\mathcal{E}$ over $\mathbb{Z}_{863}$ defined by $y^2 = x^3 + 100\,x^2 + 10\,x + 1$. The point $P = \{121, 517\}$ lies on it as can be checked with the *Mathematica* function Mod.

```
p = 863;
a = 100; b = 10; c = 1;
x = 121; y = 517;
Mod[y^2 - (x^3 + a * x^2 + b * x + c), p] == 0
```

The order of P is 432. To check this we make use of the factorization of 432 and use the ECScalarMultiplication function defined in Section 7.8.

```
FactorInteger[432]
```

```
P = {121, 517};
R = ECScalarMultiply[p, a, b, c, 432, P]
```

Suppose that Alice has chosen $m_A = 130$ and Bob $m_B = 258$. Then $Q_A = (162, 663)$ and $Q_B = (307, 674)$, as can be checked with the ECScalarMultiply function.

```
QAlice = ECScalarMultiply[p, a, b, c, 130, P]
QBob = ECScalarMultiply[p, a, b, c, 258, P]
```

Alice can compute the common key $K_{A,B}$ with the calculation $K_{A,B} = m_A\,Q_B$, where $m_A = 130$ is her secret key. She finds

```
ECScalarMultiply[p, a, b, c, 130, QBob]
```

Likewise, Bob can compute the common key $K_{A,B}$ with the calculation $K_{A,B} = m_B\,Q_A$, where $m_B = 258$ is his secret key. He also finds

```
ECScalarMultiply[p, a, b, c, 258, QAlice]
```

Now that the Diffie-Hellman key exchange system over elliptic curves has been described, it really is a straightforward exercise to show the ElGamal protocol and the other systems, described in Section 5.2, can be rewritten in the language of elliptic curves.

6.5 Why is it Attractive?

In Section 5.2, various methods are described to take the discrete logarithm over a finite field.

Exhaustive search, the Pohlig-Hellman algorithm, the baby-step giant-step method, and the Pollard-ϱ method can all be directly translated into elliptic curve terminology. Use

modular arithmetic		on an elliptic curve
multiplication	$\longleftrightarrow$	addition
exponentiation	$\longleftrightarrow$	scalar multiplication

□ Exhaustive Search

Obviously, one can can try $n = 1, 2, \ldots$ until $n\,P = Q$. The workfactor is upperbounded by the order m of P.

□ Baby-Step Giant-Step Method

Also here the generalization from Section 5.3.2 is obvious.

For instance, if one wants to store only 10 elements, one makes a table of $\{O, P, 2\,P, \ldots, 9\,P\}$, but sorted in a suitable way for easy access.

To solve $n\,P = Q$, one looks for the first time that in the list

$$Q,\ Q - 10\,P,\ Q - 20\,P,\ Q - 30\,P,\ \ldots$$

an element of the table is found. Suppose that $Q - 10\,s\,P$ occurs as $r\,P$, $0 \le r \le 9$, in the table, then $Q = (10\,s + r)\,P$, i.e. $n = 10\,s + r$.

Note that is better to compute $\hat{P} = -10\,P$ once and look in the list
$Q,\ Q + \hat{P},\ Q + 2\,\hat{P},\ Q + 3\,\hat{P},\ \ldots$ for a match with the table.

□ Index-calculus method

The index-calculus method has defeated any attempt to transfer it efficiently to the elliptic curve setting.

That is of great cryptographic significance, because the index-calculus method was the only one with a subexponential complexity.

This means that in regular discrete-logarithm-like systems the index-calculus method is the governing factor in determining the size of its parameters (to keep the system computationally secure).

□ Consequences

Since the index-calculus method is no longer around in the elliptic curve setting, one can afford much smaller parameters to achieve the same level of security.

Compare the complexity of Pollard-ρ with that of the index calculus for k-bits long numbers:

```
Table[{k, N[√(Pi * 2^k / 2), 3],
   N[Exp[1.923 * ∛(Log[2^k]) * ∛((Log[Log[2^k]])^2)], 3]},
                                          {k, 100, 300, 50}] //
 TableForm
```

For instance, in Sept. 1999 a group of 200 international researchers led by INRIA broke the 97 bits EEC challenge from Certicom. The computing power used by them was twice as much as the 512 bits RSA challenge broken a few weeks earlier.

There are special attacks on discrete logarithm based elliptic curve cryptosystems. For instance, sometimes one can translate the discrete logarithm problem for elliptic curves to the standard discrete logarithm problem!

These attacks make it necessary to avoid special classes of elliptic curves. In particular, one should not use.

singular curves,
supersingular curves,
anomalous curves.

6.6 Elliptic Addition and Scalar Multiplication Functions

```
ECAdd[p_, a_, b_, c_, P_List, Q_List] :=
 Module[{lam, x3, y3, P3},
  Which[
   P == {O}, Q,
   Q == {O}, P,
   P[[1]] != Q[[1]],
          lam = Mod[
     (Q[[2]] - P[[2]]) PowerMod[Q[[1]] - P[[1]], p - 2, p], p];
          x3 = Mod[lam^2 - a - P[[1]] - Q[[1]], p];
          y3 = Mod[- (lam (x3 - P[[1]]) + P[[2]]), p];
          {x3, y3},
   (P == Q) ∧ (P[[2]] == 0), {O},
   (P == Q) ∧ (P != {O}),
         lam = Mod[ (3 * P[[1]]^2 + 2 a * P[[1]] + b)
      PowerMod[2 P[[2]], p - 2, p], p];
          x3 = Mod[lam^2 - a - P[[1]] - Q[[1]], p];
          y3 = Mod[- (lam (x3 - P[[1]]) + P[[2]]), p];
          {x3, y3},
   (P[[1]] == Q[[1]]) ∧ (P[[2]] != Q[[2]]), {O}]]
```

```
ECScalarMultiply[p_, a_, b_, c_, 0, P_List] := {O};
ECScalarMultiply[p_, a_, b_, c_, 1, P_List] := Mod[P, p]
ECScalarMultiply[p_, a_, b_, c_, n_?EvenQ, P_List] :=
  Module[{pn2 = ECScalarMultiply[p, a, b, c, n / 2, P]},
   ECAdd[p, a, b, c, pn2, pn2]];
ECScalarMultiply[p_, a_, b_, c_, n_?OddQ, P_List] :=
  Module[{pn1 = ECScalarMultiply[p, a, b, c, n - 1, P]},
   ECAdd[p, a, b, c, pn1, P]];
```