Public-key Cryptography

Theory and Practice

Abhijit Das

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Chapter 3: Algebraic and Number-theoretic Computations

Integer Arithmetic

Integer Arithmetic

- In cryptography, we deal with very large integers with full precision.

Integer Arithmetic

- In cryptography, we deal with very large integers with full precision.
- Standard data types in programming languages cannot handle big integers.

Integer Arithmetic

- In cryptography, we deal with very large integers with full precision.
- Standard data types in programming languages cannot handle big integers.
- Special data types (like arrays of integers) are needed.

Integer Arithmetic

- In cryptography, we deal with very large integers with full precision.
- Standard data types in programming languages cannot handle big integers.
- Special data types (like arrays of integers) are needed.
- The arithmetic routines on these specific data types have to be implemented.

Integer Arithmetic

- In cryptography, we deal with very large integers with full precision.
- Standard data types in programming languages cannot handle big integers.
- Special data types (like arrays of integers) are needed.
- The arithmetic routines on these specific data types have to be implemented.
- One may use an available library (like GMP).

Integer Arithmetic

- In cryptography, we deal with very large integers with full precision.
- Standard data types in programming languages cannot handle big integers.
- Special data types (like arrays of integers) are needed.
- The arithmetic routines on these specific data types have to be implemented.
- One may use an available library (like GMP).
- Size of an integer n is $O(\log |n|)$.

Basic Integer Operations

Let a, b be two integer operands.

High-school algorithms

Operation	**Running time**
$a + b$	$O(\max(\log a, \log b))$
$a - b$	$O(\max(\log a, \log b))$
ab	$O((\log a)(\log b))$
a^2	$O(\log^2 a)$
$(a \text{ quot } b)$ and/or $(a \text{ rem } b)$	$O((\log a)(\log b))$

Basic Integer Operations

Let a, b be two integer operands.

High-school algorithms

Operation	Running time
$a + b$	$O(\max(\log a, \log b))$
$a - b$	$O(\max(\log a, \log b))$
ab	$O((\log a)(\log b))$
a^2	$O(\log^2 a)$
$(a \operatorname{quot} b)$ and/or $(a \operatorname{rem} b)$	$O((\log a)(\log b))$

Fast multiplication: Assume a, b are of the same size s.

Basic Integer Operations

Let a, b be two integer operands.

High-school algorithms

Operation	Running time
$a + b$	$O(\max(\log a, \log b))$
$a - b$	$O(\max(\log a, \log b))$
ab	$O((\log a)(\log b))$
a^2	$O(\log^2 a)$
$(a \text{ quot } b)$ and/or $(a \text{ rem } b)$	$O((\log a)(\log b))$

Fast multiplication: Assume a, b are of the same size s.

- **Karatsuba multiplication**: $O(s^{1.585})$

Basic Integer Operations

Let a, b be two integer operands.

High-school algorithms

Operation	Running time
$a + b$	$O(\max(\log a, \log b))$
$a - b$	$O(\max(\log a, \log b))$
ab	$O((\log a)(\log b))$
a^2	$O(\log^2 a)$
$(a \text{ quot } b)$ and/or $(a \text{ rem } b)$	$O((\log a)(\log b))$

Fast multiplication: Assume a, b are of the same size s.

- **Karatsuba multiplication**: $O(s^{1.585})$
- **FFT multiplication**: $O(s \log s)$
  [not frequently used in cryptography]

Binary GCD

To compute the GCD of two positive integers a and b.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha} a'$ and $b = 2^{\beta} b'$ with a', b' odd.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha}a'$ and $b = 2^{\beta}b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha}a'$ and $b = 2^{\beta}b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha}a'$ and $b = 2^{\beta}b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

- $\gcd(a, b) = \gcd(a - b, b)$.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha} a'$ and $b = 2^{\beta} b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha, \beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

- $\gcd(a, b) = \gcd(a - b, b)$.
- Write $a - b = 2^{\gamma} c$ with $\gamma \geqslant 1$ and c odd.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha} a'$ and $b = 2^{\beta} b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

- $\gcd(a, b) = \gcd(a - b, b)$.
- Write $a - b = 2^{\gamma} c$ with $\gamma \geqslant 1$ and c odd.
- Then, $\gcd(a, b) = \gcd(c, b)$.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha}a'$ and $b = 2^{\beta}b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

- $\gcd(a, b) = \gcd(a - b, b)$.
- Write $a - b = 2^{\gamma}c$ with $\gamma \geqslant 1$ and c odd.
- Then, $\gcd(a, b) = \gcd(c, b)$.
- Repeat until one operand reduces to 0.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha} a'$ and $b = 2^{\beta} b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

- $\gcd(a, b) = \gcd(a - b, b)$.
- Write $a - b = 2^{\gamma} c$ with $\gamma \geqslant 1$ and c odd.
- Then, $\gcd(a, b) = \gcd(c, b)$.
- Repeat until one operand reduces to 0.

Running time of Euclidean gcd: $O(\max(\log a, \log b)^3)$.

Binary GCD

To compute the GCD of two positive integers a and b.

Write $a = 2^{\alpha}a'$ and $b = 2^{\beta}b'$ with a', b' odd.

$\gcd(a, b) = 2^{\min(\alpha,\beta)} \gcd(a', b')$.

Assume that both a, b are odd and $a \geqslant b$.

- $\gcd(a, b) = \gcd(a - b, b)$.
- Write $a - b = 2^{\gamma}c$ with $\gamma \geqslant 1$ and c odd.
- Then, $\gcd(a, b) = \gcd(c, b)$.
- Repeat until one operand reduces to 0.

Running time of Euclidean gcd: $O(\max(\log a, \log b)^3)$.

Running time of binary gcd: $O(\max(\log a, \log b)^2)$.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\left[\begin{array}{lll} r_0 = a, & u_0 = 1, & v_0 = 0, \\ r_1 = b, & u_1 = 0, & v_1 = 1. \end{array}\right]$

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\left[\begin{array}{lll} r_0 = a, & u_0 = 1, & v_0 = 0, \\ r_1 = b, & u_1 = 0, & v_1 = 1. \end{array}\right]$

Iteration: For $i = 2, 3, 4, \ldots$, do the following:

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\left[\begin{array}{lllllllll} r_0 & = & a, & u_0 & = & 1, & v_0 & = & 0, \\ r_1 & = & b, & u_1 & = & 0, & v_1 & = & 1. \end{array}\right]$

Iteration: For $i = 2, 3, 4, \ldots$, do the following:

- Compute the quotient $q_i = r_{i-2} \text{ quot } r_{i-1}$.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\left[\begin{array}{lll} r_0 = a, & u_0 = 1, & v_0 = 0, \\ r_1 = b, & u_1 = 0, & v_1 = 1. \end{array}\right]$

Iteration: For $i = 2, 3, 4, \ldots$, do the following:

- Compute the quotient $q_i = r_{i-2} \text{ quot } r_{i-1}$.
- Compute $r_i = r_{i-2} - q_i r_{i-1}$.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\left[\begin{array}{lll} r_0 = a, & u_0 = 1, & v_0 = 0, \\ r_1 = b, & u_1 = 0, & v_1 = 1. \end{array}\right]$

Iteration: For $i = 2, 3, 4, \ldots$, do the following:

- Compute the quotient $q_i = r_{i-2} \operatorname{quot} r_{i-1}$.
- Compute $r_i = r_{i-2} - q_i r_{i-1}$.
- Compute $u_i = u_{i-2} - q_i u_{i-1}$.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\begin{bmatrix} r_0 = a, & u_0 = 1, & v_0 = 0, \\ r_1 = b, & u_1 = 0, & v_1 = 1. \end{bmatrix}$

Iteration: For $i = 2, 3, 4, \ldots$, do the following:

- Compute the quotient $q_i = r_{i-2} \text{ quot } r_{i-1}$.
- Compute $r_i = r_{i-2} - q_i r_{i-1}$.
- Compute $u_i = u_{i-2} - q_i u_{i-1}$.
- Compute $v_i = v_{i-2} - q_i v_{i-1}$.

Extended Euclidean GCD

To compute the GCD of two positive integers a and b.

Define three sequences r_i, u_i, v_i.

Initialize: $\begin{bmatrix} r_0 = a, & u_0 = 1, & v_0 = 0, \\ r_1 = b, & u_1 = 0, & v_1 = 1. \end{bmatrix}$

Iteration: For $i = 2, 3, 4, \ldots$, do the following:

- Compute the quotient $q_i = r_{i-2} \text{ quot } r_{i-1}$.
- Compute $r_i = r_{i-2} - q_i r_{i-1}$.
- Compute $u_i = u_{i-2} - q_i u_{i-1}$.
- Compute $v_i = v_{i-2} - q_i v_{i-1}$.
- Break if $r_i = 0$.

Extended Euclidean GCD (contd.)

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.
- Suppose the loop terminates for $i = j$ (that is, $r_j = 0$).

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.
- Suppose the loop terminates for $i = j$ (that is, $r_j = 0$).
- $\gcd(a, b) = r_{j-1} = u_{j-1} a + v_{j-1} b$.

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.
- Suppose the loop terminates for $i = j$ (that is, $r_j = 0$).
- $\gcd(a, b) = r_{j-1} = u_{j-1} a + v_{j-1} b$.

- One needs to remember the r, u, v values only from the two previous iterations.

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.
- Suppose the loop terminates for $i = j$ (that is, $r_j = 0$).
- $\gcd(a, b) = r_{j-1} = u_{j-1}a + v_{j-1}b$.

- One needs to remember the r, u, v values only from the two previous iterations.
- One can compute only the r and u sequences in the loop.

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.
- Suppose the loop terminates for $i = j$ (that is, $r_j = 0$).
- $\gcd(a, b) = r_{j-1} = u_{j-1}a + v_{j-1}b$.

- One needs to remember the r, u, v values only from the two previous iterations.
- One can compute only the r and u sequences in the loop.
- One gets $v_{j-1} = (r_{j-1} - u_{j-1}a)/b$.

Extended Euclidean GCD (contd.)

- We maintain the invariance $u_i a + v_i b = r_i$ for all i.
- Suppose the loop terminates for $i = j$ (that is, $r_j = 0$).
- $\gcd(a, b) = r_{j-1} = u_{j-1}a + v_{j-1}b$.

- One needs to remember the r, u, v values only from the two previous iterations.
- One can compute only the r and u sequences in the loop.
- One gets $v_{j-1} = (r_{j-1} - u_{j-1}a)/b$.

- The binary gcd algorithm can be similarly modified so as to compute the u and v sequences maintaining the invariant $u_i a + v_i b = r_i$ for all i.

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

i	q_i	r_i	u_i	v_i	$u_i a + v_i b$
0	–	78	1	0	78
1	–	21	0	1	21

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

i	q_i	r_i	u_i	v_i	$u_i a + v_i b$
0	–	78	1	0	78
1	–	21	0	1	21
2	3	15	1	-3	15

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

i	q_i	r_i	u_i	v_i	$u_i a + v_i b$
0	–	78	1	0	78
1	–	21	0	1	21
2	3	15	1	-3	15
3	1	6	-1	4	6

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

i	q_i	r_i	u_i	v_i	$u_i a + v_i b$
0	–	78	1	0	78
1	–	21	0	1	21
2	3	15	1	-3	15
3	1	6	-1	4	6
4	2	3	3	-11	3

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

i	q_i	r_i	u_i	v_i	$u_i a + v_i b$
0	–	78	1	0	78
1	–	21	0	1	21
2	3	15	1	-3	15
3	1	6	-1	4	6
4	2	3	3	-11	3
5	2	0	-7	26	0

Extended Euclidean GCD (Example)

To compute $\gcd(78, 21) = 78u + 21v$.

i	q_i	r_i	u_i	v_i	$u_i a + v_i b$
0	–	78	1	0	78
1	–	21	0	1	21
2	3	15	1	−3	15
3	1	6	−1	4	6
4	2	3	3	−11	3
5	2	0	−7	26	0

Thus, $\gcd(78, 21) = 3 = 3 \times 78 + (-11) \times 21$.

Modular Integer Arithmetic

Let $n \in \mathbb{N}$. Define $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$.

Modular Integer Arithmetic

Let $n \in \mathbb{N}$. Define $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$.

- **Addition:** $a + b \pmod{n} = \begin{cases} a+b & \text{if } a+b < n \\ a+b-n & \text{if } a+b \geqslant n \end{cases}$

Modular Integer Arithmetic

Let $n \in \mathbb{N}$. Define $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$.

- **Addition:** $a + b \pmod{n} = \begin{cases} a+b & \text{if } a+b < n \\ a+b-n & \text{if } a+b \geqslant n \end{cases}$
- **Subtraction:** $a - b \pmod{n} = \begin{cases} a-b & \text{if } a \geqslant b \\ a-b+n & \text{if } a < b \end{cases}$

Modular Integer Arithmetic

Let $n \in \mathbb{N}$. Define $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$.

- **Addition:** $a + b \pmod{n} = \begin{cases} a+b & \text{if } a+b < n \\ a+b-n & \text{if } a+b \geqslant n \end{cases}$
- **Subtraction:** $a - b \pmod{n} = \begin{cases} a-b & \text{if } a \geqslant b \\ a-b+n & \text{if } a < b \end{cases}$
- **Multiplication:** $ab \pmod{n} = (ab) \operatorname{rem} n$.

Modular Integer Arithmetic

Let $n \in \mathbb{N}$. Define $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$.

- **Addition:** $a + b \pmod{n} = \begin{cases} a+b & \text{if } a+b < n \\ a+b-n & \text{if } a+b \geqslant n \end{cases}$
- **Subtraction:** $a - b \pmod{n} = \begin{cases} a-b & \text{if } a \geqslant b \\ a-b+n & \text{if } a < b \end{cases}$
- **Multiplication:** $ab \pmod{n} = (ab) \operatorname{rem} n$.
- **Inverse:** $a \in \mathbb{Z}_n^*$ is invertible if and only if $\gcd(a, n) = 1$.
  But then $1 = ua + vn$ for some integers u, v.
  Take $a^{-1} \equiv u \pmod{n}$.

Example of Modular Arithmetic

Take $n = 257$, $a = 127$, $b = 217$.

Example of Modular Arithmetic

Take $n = 257$, $a = 127$, $b = 217$.

- **Addition:** $a + b = 344 > 257$, so $a + b \equiv 344 - 257 \equiv 87 \pmod{n}$.

Example of Modular Arithmetic

Take $n = 257$, $a = 127$, $b = 217$.

- **Addition:** $a + b = 344 > 257$, so $a + b \equiv 344 - 257 \equiv 87 \pmod{n}$.
- **Subtraction:** $a - b = -90 < 0$, so $a - b \equiv -90 + 257 \equiv 167 \pmod{n}$.

Example of Modular Arithmetic

Take $n = 257$, $a = 127$, $b = 217$.

- **Addition:** $a + b = 344 > 257$, so $a + b \equiv 344 - 257 \equiv 87 \pmod{n}$.
- **Subtraction:** $a - b = -90 < 0$, so $a - b \equiv -90 + 257 \equiv 167 \pmod{n}$.
- **Multiplication:** $ab \equiv (127 \times 217) \mathrel{\mathrm{rem}} 257 \equiv 27559 \mathrel{\mathrm{rem}} 257 \equiv 60 \pmod{n}$.

Example of Modular Arithmetic

Take $n = 257$, $a = 127$, $b = 217$.

- **Addition:** $a + b = 344 > 257$, so
  $a + b \equiv 344 - 257 \equiv 87 \pmod{n}$.
- **Subtraction:** $a - b = -90 < 0$, so
  $a - b \equiv -90 + 257 \equiv 167 \pmod{n}$.
- **Multiplication:**
  $ab \equiv (127 \times 217) \operatorname{rem} 257 \equiv 27559 \operatorname{rem} 257 \equiv 60 \pmod{n}$.
- **Inverse:** $\gcd(b, n) = 1 = (-45)b + 38n$, so
  $b^{-1} \equiv -45 + 257 \equiv 212 \pmod{n}$.

Example of Modular Arithmetic

Take $n = 257$, $a = 127$, $b = 217$.

- **Addition:** $a + b = 344 > 257$, so
  $a + b \equiv 344 - 257 \equiv 87 \pmod{n}$.
- **Subtraction:** $a - b = -90 < 0$, so
  $a - b \equiv -90 + 257 \equiv 167 \pmod{n}$.
- **Multiplication:**
  $ab \equiv (127 \times 217) \operatorname{rem} 257 \equiv 27559 \operatorname{rem} 257 \equiv 60 \pmod{n}$.
- **Inverse:** $\gcd(b, n) = 1 = (-45)b + 38n$, so
  $b^{-1} \equiv -45 + 257 \equiv 212 \pmod{n}$.
- **Division:**
  $a/b \equiv ab^{-1} \equiv (127 \times 212) \operatorname{rem} 257 \equiv 196 \pmod{n}$.

Modular Exponentiation: Slow Algorithm

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.
- **Example:** $n = 257$, $a = 127$, $e = 217$.

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.
- **Example:** $n = 257$, $a = 127$, $e = 217$.

$$a^2 \equiv a \times a \equiv 195 \pmod{n},$$

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.

- **Example:** $n = 257$, $a = 127$, $e = 217$.

$$
\begin{aligned}
a^2 &\equiv a \times a \equiv 195 \pmod{n}, \\
a^3 &\equiv a^2 \times a \equiv 195 \times 127 \equiv 93 \pmod{n},
\end{aligned}
$$

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.
- **Example:** $n = 257$, $a = 127$, $e = 217$.

$$\begin{aligned} a^2 &\equiv a \times a \equiv 195 \pmod{n}, \\ a^3 &\equiv a^2 \times a \equiv 195 \times 127 \equiv 93 \pmod{n}, \\ a^4 &\equiv a^3 \times a \equiv 93 \times 127 \equiv 246 \pmod{n}, \end{aligned}$$

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.
- **Example:** $n = 257$, $a = 127$, $e = 217$.

$$
\begin{aligned}
a^2 &\equiv a \times a \equiv 195 \pmod{n}, \\
a^3 &\equiv a^2 \times a \equiv 195 \times 127 \equiv 93 \pmod{n}, \\
a^4 &\equiv a^3 \times a \equiv 93 \times 127 \equiv 246 \pmod{n}, \\
&\ldots
\end{aligned}
$$

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.
- **Example:** $n = 257$, $a = 127$, $e = 217$.

$$\begin{aligned}
a^2 &\equiv a \times a \equiv 195 \pmod{n}, \\
a^3 &\equiv a^2 \times a \equiv 195 \times 127 \equiv 93 \pmod{n}, \\
a^4 &\equiv a^3 \times a \equiv 93 \times 127 \equiv 246 \pmod{n}, \\
&\cdots \\
a^{216} &\equiv a^{215} \times a \equiv 131 \times 127 \equiv 189 \pmod{n},
\end{aligned}$$

Modular Exponentiation: Slow Algorithm

- Let $n \in \mathbb{N}$, $a \in \mathbb{Z}_n$ and $e \in \mathbb{N}_0$. To compute $a^e \pmod{n}$.
- Compute $a, a^2, a^3, \ldots, a^e$ successively by multiplying with a modulo n.
- **Example:** $n = 257, a = 127, e = 217.$

$$\begin{aligned}
a^2 &\equiv a \times a \equiv 195 \pmod{n}, \\
a^3 &\equiv a^2 \times a \equiv 195 \times 127 \equiv 93 \pmod{n}, \\
a^4 &\equiv a^3 \times a \equiv 93 \times 127 \equiv 246 \pmod{n}, \\
&\ldots \\
a^{216} &\equiv a^{215} \times a \equiv 131 \times 127 \equiv 189 \pmod{n}, \\
a^{217} &\equiv a^{216} \times a \equiv 189 \times 127 \equiv 102 \pmod{n}.
\end{aligned}$$

Right-to-left Modular Exponentiation

To compute $a^e \pmod{n}$.

Right-to-left Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_12^1 + e_02^0$.

Right-to-left Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_12^1 + e_02^0$.
- $a^e \equiv \left(a^{2^{l-1}}\right)^{e_{l-1}} \left(a^{2^{l-2}}\right)^{e_{l-2}} \cdots \left(a^{2^1}\right)^{e_1} \left(a^{2^0}\right)^{e_0} \pmod{n}$.

Right-to-left Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- $a^e \equiv \left(a^{2^{l-1}}\right)^{e_{l-1}} \left(a^{2^{l-2}}\right)^{e_{l-2}} \cdots \left(a^{2^1}\right)^{e_1} \left(a^{2^0}\right)^{e_0} \pmod{n}$.
- Compute $a, a^2, a^{2^2}, a^{2^3}, \ldots, a^{2^{l-1}}$ and multiply those a^{2^i} modulo n for which $e_i = 1$. Also for $i \geqslant 1$, we have $a^{2^i} \equiv \left(a^{2^{i-1}}\right)^2 \pmod{n}$.

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$,

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$,

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$, $a^{2^3} \equiv (246)^2 \equiv 121 \pmod{n}$,

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$, $a^{2^3} \equiv (246)^2 \equiv 121 \pmod{n}$, $a^{2^4} \equiv (121)^2 \equiv 249 \pmod{n}$,

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$, $a^{2^3} \equiv (246)^2 \equiv 121 \pmod{n}$, $a^{2^4} \equiv (121)^2 \equiv 249 \pmod{n}$, $a^{2^5} \equiv (249)^2 \equiv 64 \pmod{n}$,

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$, $a^{2^3} \equiv (246)^2 \equiv 121 \pmod{n}$, $a^{2^4} \equiv (121)^2 \equiv 249 \pmod{n}$, $a^{2^5} \equiv (249)^2 \equiv 64 \pmod{n}$, $a^{2^6} \equiv (64)^2 \equiv 241 \pmod{n}$ and

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$, $a^{2^3} \equiv (246)^2 \equiv 121 \pmod{n}$, $a^{2^4} \equiv (121)^2 \equiv 249 \pmod{n}$, $a^{2^5} \equiv (249)^2 \equiv 64 \pmod{n}$, $a^{2^6} \equiv (64)^2 \equiv 241 \pmod{n}$ and $a^{2^7} \equiv (241)^2 \equiv 256 \pmod{n}$.

Right-to-left Modular Exponentiation (Example)

Take $n = 257$, $a = 127$, $e = 217$.

- $e = (11011001)_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^0$. So $a^e \equiv a^{2^7} a^{2^6} a^{2^4} a^{2^3} a^{2^0} \pmod{n}$.
- $a^2 \equiv 195 \pmod{n}$, $a^{2^2} \equiv (195)^2 \equiv 246 \pmod{n}$, $a^{2^3} \equiv (246)^2 \equiv 121 \pmod{n}$, $a^{2^4} \equiv (121)^2 \equiv 249 \pmod{n}$, $a^{2^5} \equiv (249)^2 \equiv 64 \pmod{n}$, $a^{2^6} \equiv (64)^2 \equiv 241 \pmod{n}$ and $a^{2^7} \equiv (241)^2 \equiv 256 \pmod{n}$.
- $a^e \equiv 256 \times 241 \times 249 \times 121 \times 127 \equiv 102 \pmod{n}$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2} \ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_12^1 + e_02^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2} \ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2}\ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.
- $\epsilon_l = 0$, and $\epsilon_i = 2\epsilon_{i+1} + e_i$ for $i < l$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2}\ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.
- $\epsilon_l = 0$, and $\epsilon_i = 2\epsilon_{i+1} + e_i$ for $i < l$.
- $a^{\epsilon_l} \equiv 1 \pmod{n}$ and $a^{\epsilon_i} \equiv (a^{\epsilon_{i+1}})^2 \times a^{e_i} \pmod{n}$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2}\ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.
- $\epsilon_l = 0$, and $\epsilon_i = 2\epsilon_{i+1} + e_i$ for $i < l$.
- $a^{\epsilon_l} \equiv 1 \pmod{n}$ and $a^{\epsilon_i} \equiv (a^{\epsilon_{i+1}})^2 \times a^{e_i} \pmod{n}$.
- Finally, $\epsilon_0 = e$, so output $a^{\epsilon_0} \pmod{n}$.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2}\ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.
- $\epsilon_l = 0$, and $\epsilon_i = 2\epsilon_{i+1} + e_i$ for $i < l$.
- $a^{\epsilon_l} \equiv 1 \pmod{n}$ and $a^{\epsilon_i} \equiv (a^{\epsilon_{i+1}})^2 \times a^{e_i} \pmod{n}$.
- Finally, $\epsilon_0 = e$, so output $a^{\epsilon_0} \pmod{n}$.
- Initialize *product* to 1 (corresponds to $i = l$).

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2}\ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.
- $\epsilon_l = 0$, and $\epsilon_i = 2\epsilon_{i+1} + e_i$ for $i < l$.
- $a^{\epsilon_l} \equiv 1 \pmod{n}$ and $a^{\epsilon_i} \equiv (a^{\epsilon_{i+1}})^2 \times a^{e_i} \pmod{n}$.
- Finally, $\epsilon_0 = e$, so output $a^{\epsilon_0} \pmod{n}$.
- Initialize *product* to 1 (corresponds to $i = l$).
- For $i = l-1, l-2, \ldots, 1, 0$, square *product*.
  If $e_i = 1$, then multiply product by a.

Left-to-right Modular Exponentiation

To compute $a^e \pmod{n}$.

- Binary representation: $e = (e_{l-1}e_{l-2}\ldots e_1e_0)_2 = e_{l-1}2^{l-1} + e_{l-2}2^{l-2} + \cdots + e_1 2^1 + e_0 2^0$.
- Define $\epsilon_i = (e_{l-1}e_{l-2}\ldots e_i)_2$ for $i = l, l-1, l-2, \ldots, 0$.
- $\epsilon_l = 0$, and $\epsilon_i = 2\epsilon_{i+1} + e_i$ for $i < l$.
- $a^{\epsilon_l} \equiv 1 \pmod{n}$ and $a^{\epsilon_i} \equiv (a^{\epsilon_{i+1}})^2 \times a^{e_i} \pmod{n}$.
- Finally, $\epsilon_0 = e$, so output $a^{\epsilon_0} \pmod{n}$.
- Initialize *product* to 1 (corresponds to $i = l$).
- For $i = l-1, l-2, \ldots, 1, 0$, square *product*.
  If $e_i = 1$, then multiply product by a.
- Square-and-(conditionally)-multiply algorithm

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	–	0	1

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	$-$	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod{n}$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod{n}$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod{n}$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod{n}$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod{n}$
5	0	$(110)_2 = 6$	$93^2 \equiv 168 \pmod{n}$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod{n}$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod{n}$
5	0	$(110)_2 = 6$	$93^2 \equiv 168 \pmod{n}$
4	1	$(1101)_2 = 13$	$168^2 \times 127 \equiv 69 \pmod{n}$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod n$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod n$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod n$
5	0	$(110)_2 = 6$	$93^2 \equiv 168 \pmod n$
4	1	$(1101)_2 = 13$	$168^2 \times 127 \equiv 69 \pmod n$
3	1	$(11011)_2 = 27$	$69^2 \times 127 \equiv 183 \pmod n$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod{n}$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod{n}$
5	0	$(110)_2 = 6$	$93^2 \equiv 168 \pmod{n}$
4	1	$(1101)_2 = 13$	$168^2 \times 127 \equiv 69 \pmod{n}$
3	1	$(11011)_2 = 27$	$69^2 \times 127 \equiv 183 \pmod{n}$
2	0	$(110110)_2 = 54$	$183^2 \equiv 79 \pmod{n}$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod{n}$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod{n}$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod{n}$
5	0	$(110)_2 = 6$	$93^2 \equiv 168 \pmod{n}$
4	1	$(1101)_2 = 13$	$168^2 \times 127 \equiv 69 \pmod{n}$
3	1	$(11011)_2 = 27$	$69^2 \times 127 \equiv 183 \pmod{n}$
2	0	$(110110)_2 = 54$	$183^2 \equiv 79 \pmod{n}$
1	0	$(1101100)_2 = 108$	$79^2 \equiv 73 \pmod{n}$

Left-to-right Modular Exponentiation (Example)

Take $n = 257$, $a = 127$ and $e = 217$.
We have the binary representation: $e = (11011001)_2$.

i	e_i	ϵ_i	$a^{\epsilon_i} \pmod n$
8	–	0	1
7	1	$(1)_2 = 1$	$1^2 \times 127 \equiv 127 \pmod n$
6	1	$(11)_2 = 3$	$127^2 \times 127 \equiv 93 \pmod n$
5	0	$(110)_2 = 6$	$93^2 \equiv 168 \pmod n$
4	1	$(1101)_2 = 13$	$168^2 \times 127 \equiv 69 \pmod n$
3	1	$(11011)_2 = 27$	$69^2 \times 127 \equiv 183 \pmod n$
2	0	$(110110)_2 = 54$	$183^2 \equiv 79 \pmod n$
1	0	$(1101100)_2 = 108$	$79^2 \equiv 73 \pmod n$
0	1	$(11011001)_2 = 217$	$73^2 \times 127 \equiv 102 \pmod n$

Primality Testing

Primality Testing

- A fundamental problem in computational number theory.

Primality Testing

- A fundamental problem in computational number theory.
- Probabilistic (that is, randomized) algorithms solve the problem reasonably efficiently with arbitrarily small probability of error.

Primality Testing

- A fundamental problem in computational number theory.
- Probabilistic (that is, randomized) algorithms solve the problem reasonably efficiently with arbitrarily small probability of error.
- Some of these probabilistic algorithms can be converted to deterministic polynomial-time algorithms under certain unproven assumptions (Extended Riemann Hypothesis).

Primality Testing

- A fundamental problem in computational number theory.
- Probabilistic (that is, randomized) algorithms solve the problem reasonably efficiently with arbitrarily small probability of error.
- Some of these probabilistic algorithms can be converted to deterministic polynomial-time algorithms under certain unproven assumptions (Extended Riemann Hypothesis).
- The first known deterministic polynomial-time algorithm with proofs not dependent on any conjectures is from Agarwal, Kayal and Saxena (2002).

Primality Testing

- A fundamental problem in computational number theory.
- Probabilistic (that is, randomized) algorithms solve the problem reasonably efficiently with arbitrarily small probability of error.
- Some of these probabilistic algorithms can be converted to deterministic polynomial-time algorithms under certain unproven assumptions (Extended Riemann Hypothesis).
- The first known deterministic polynomial-time algorithm with proofs not dependent on any conjectures is from Agarwal, Kayal and Saxena (2002).
- The AKS algorithm is not yet practical.

Fermat Test

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.
- However, $8^{35-1} \equiv 29 \not\equiv 1 \pmod{35}$. So, 6 fails to prove the compositeness of 35, but 8 proves it.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.
- However, $8^{35-1} \equiv 29 \not\equiv 1 \pmod{35}$. So, 6 fails to prove the compositeness of 35, but 8 proves it.
- An integer n is called a **pseudoprime** to a base a with $\gcd(a, n) = 1$, if $a^{n-1} \equiv 1 \pmod{n}$.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.
- However, $8^{35-1} \equiv 29 \not\equiv 1 \pmod{35}$. So, 6 fails to prove the compositeness of 35, but 8 proves it.
- An integer n is called a **pseudoprime** to a base a with $\gcd(a, n) = 1$, if $a^{n-1} \equiv 1 \pmod{n}$.
- A prime is a pseudoprime to every coprime base.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.
- However, $8^{35-1} \equiv 29 \not\equiv 1 \pmod{35}$. So, 6 fails to prove the compositeness of 35, but 8 proves it.
- An integer n is called a **pseudoprime** to a base a with $\gcd(a, n) = 1$, if $a^{n-1} \equiv 1 \pmod{n}$.
- A prime is a pseudoprime to every coprime base.
- A prime has **no witnesses** to its compositeness.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.
- However, $8^{35-1} \equiv 29 \not\equiv 1 \pmod{35}$. So, 6 fails to prove the compositeness of 35, but 8 proves it.
- An integer n is called a **pseudoprime** to a base a with $\gcd(a, n) = 1$, if $a^{n-1} \equiv 1 \pmod{n}$.
- A prime is a pseudoprime to every coprime base.
- A prime has **no witnesses** to its compositeness.
- If a composite integer n is not a pseudoprime to some base, then n is not a pseudoprime to at least half of the bases in $\mathbb{Z}_n^*$.

Fermat Test

- Fermat's little theorem: If n is prime, then $a^{n-1} \equiv 1 \pmod{n}$ for all a coprime to n.
- The converse is not true: $6^{35-1} \equiv (6^2)^{17} \equiv 1 \pmod{35}$.
- However, $8^{35-1} \equiv 29 \not\equiv 1 \pmod{35}$. So, 6 fails to prove the compositeness of 35, but 8 proves it.
- An integer n is called a **pseudoprime** to a base a with $\gcd(a, n) = 1$, if $a^{n-1} \equiv 1 \pmod{n}$.
- A prime is a pseudoprime to every coprime base.
- A prime has **no witnesses** to its compositeness.
- If a composite integer n is not a pseudoprime to some base, then n is not a pseudoprime to at least half of the bases in $\mathbb{Z}_n^*$.
- In that case, the density of witnesses for the compositeness of n is at least $1/2$.

Fermat Test (contd.)

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.
- If $a_i^{n-1} \equiv 1 \pmod{n}$ for all i, declare n as prime.

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.
- If $a_i^{n-1} \equiv 1 \pmod{n}$ for all i, declare n as prime.
- If $a_i^{n-1} \not\equiv 1 \pmod{n}$ for some i, declare n as composite.

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.
- If $a_i^{n-1} \equiv 1 \pmod{n}$ for all i, declare n as prime.
- If $a_i^{n-1} \not\equiv 1 \pmod{n}$ for some i, declare n as composite.

- If this test declares n as composite, there is no error.

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.
- If $a_i^{n-1} \equiv 1 \pmod{n}$ for all i, declare n as prime.
- If $a_i^{n-1} \not\equiv 1 \pmod{n}$ for some i, declare n as composite.

- If this test declares n as composite, there is no error.
- If this test declares n as prime, there may be an error.

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.
- If $a_i^{n-1} \equiv 1 \pmod{n}$ for all i, declare n as prime.
- If $a_i^{n-1} \not\equiv 1 \pmod{n}$ for some i, declare n as composite.

- If this test declares n as composite, there is no error.
- If this test declares n as prime, there may be an error.
- If n has (at least) one witness for its compositeness, then the probability of error is $\leqslant 1/2^t$.

Fermat Test (contd.)

- Choose t random bases $a_1, a_2, \ldots, a_t \in \mathbb{Z}_n^*$.
- If $a_i^{n-1} \equiv 1 \pmod{n}$ for all i, declare n as prime.
- If $a_i^{n-1} \not\equiv 1 \pmod{n}$ for some i, declare n as composite.

- If this test declares n as composite, there is no error.
- If this test declares n as prime, there may be an error.
- If n has (at least) one witness for its compositeness, then the probability of error is $\leqslant 1/2^t$.
- By choosing t suitably, this probability can be made very low.

Carmichael Numbers

There exist composite integers which have no (coprime) witnesses of compositeness.

These are called **Carmichael numbers**.

Carmichael Numbers

There exist composite integers which have no (coprime) witnesses of compositeness.

These are called **Carmichael numbers**.

- Although not common, Carmichael numbers are infinite in number.

Carmichael Numbers

There exist composite integers which have no (coprime) witnesses of compositeness.

These are called **Carmichael numbers**.

- Although not common, Carmichael numbers are infinite in number.
- The smallest Carmichael number is $561 = 3 \times 11 \times 17$.

Carmichael Numbers

There exist composite integers which have no (coprime) witnesses of compositeness.

These are called **Carmichael numbers**.

- Although not common, Carmichael numbers are infinite in number.
- The smallest Carmichael number is $561 = 3 \times 11 \times 17$.
- A Carmichael number must be odd, square-free, and the product of at least three (distinct) primes.

Carmichael Numbers

There exist composite integers which have no (coprime) witnesses of compositeness.

These are called **Carmichael numbers**.

- Although not common, Carmichael numbers are infinite in number.
- The smallest Carmichael number is $561 = 3 \times 11 \times 17$.
- A Carmichael number must be odd, square-free, and the product of at least three (distinct) primes.
- For every prime divisor p of a Carmichael number n, we must have $(p - 1) \mid (n - 1)$.

Euler (or Solovay-Strassen) Test

An integer $n \in \mathbb{N}$ is called an **Euler pseudoprime** or a **Solovay-Strassen pseudoprime** to base a (with $\gcd(a, n) = 1$) if $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

Euler (or Solovay-Strassen) Test

An integer $n \in \mathbb{N}$ is called an **Euler pseudoprime** or a **Solovay-Strassen pseudoprime** to base a (with $\gcd(a, n) = 1$) if $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

- If n is an Euler pseudoprime to base a, then n is also a (Fermat) pseudoprime to base a. The converse is not true.

Euler (or Solovay-Strassen) Test

An integer $n \in \mathbb{N}$ is called an **Euler pseudoprime** or a **Solovay-Strassen pseudoprime** to base a (with $\gcd(a, n) = 1$) if $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

- If n is an Euler pseudoprime to base a, then n is also a (Fermat) pseudoprime to base a. The converse is not true.
- By Euler's criterion, a prime is Euler pseudoprime to all coprime bases.

Euler (or Solovay-Strassen) Test

An integer $n \in \mathbb{N}$ is called an **Euler pseudoprime** or a **Solovay-Strassen pseudoprime** to base a (with $\gcd(a, n) = 1$) if $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

- If n is an Euler pseudoprime to base a, then n is also a (Fermat) pseudoprime to base a. The converse is not true.
- By Euler's criterion, a prime is Euler pseudoprime to all coprime bases.
- A composite integer n is Euler pseudoprime to at most half the bases in $\mathbb{Z}_n^*$.

Euler (or Solovay-Strassen) Test

An integer $n \in \mathbb{N}$ is called an **Euler pseudoprime** or a **Solovay-Strassen pseudoprime** to base a (with $\gcd(a, n) = 1$) if $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

- If n is an Euler pseudoprime to base a, then n is also a (Fermat) pseudoprime to base a. The converse is not true.
- By Euler's criterion, a prime is Euler pseudoprime to all coprime bases.
- A composite integer n is Euler pseudoprime to at most half the bases in $\mathbb{Z}_n^*$.
- Even Carmichael numbers possess compositeness witnesses under the revised criterion.

Euler (or Solovay-Strassen) Test

An integer $n \in \mathbb{N}$ is called an **Euler pseudoprime** or a **Solovay-Strassen pseudoprime** to base a (with $\gcd(a, n) = 1$) if $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \pmod{n}$, where $\left(\frac{a}{n}\right)$ is the Jacobi symbol.

- If n is an Euler pseudoprime to base a, then n is also a (Fermat) pseudoprime to base a. The converse is not true.
- By Euler's criterion, a prime is Euler pseudoprime to all coprime bases.
- A composite integer n is Euler pseudoprime to at most half the bases in $\mathbb{Z}_n^*$.
- Even Carmichael numbers possess compositeness witnesses under the revised criterion.

  Example: $5^{(561-1)/2} \equiv 67 \pmod{561}$, whereas $\left(\frac{5}{561}\right) = 1$.

Miller-Rabin Test

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.
- An odd composite integer which is not a prime power has at least four modular square roots of 1.

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.
- An odd composite integer which is not a prime power has at least four modular square roots of 1.
- Suppose $a^{n-1} \equiv 1 \pmod{n}$ (with $\gcd(a, n) = 1$).
  Write $n - 1 = 2^r n'$ with n' odd and $r \in \mathbb{N}$.

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.
- An odd composite integer which is not a prime power has at least four modular square roots of 1.
- Suppose $a^{n-1} \equiv 1 \pmod{n}$ (with $\gcd(a, n) = 1$). Write $n - 1 = 2^r n'$ with n' odd and $r \in \mathbb{N}$.
- Consider the sequence $b_i \equiv (a^{n'})^{2^i} \pmod{n}$ for $i = 0, 1, 2, \ldots, r$.

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.
- An odd composite integer which is not a prime power has at least four modular square roots of 1.
- Suppose $a^{n-1} \equiv 1 \pmod{n}$ (with $\gcd(a, n) = 1$).
  Write $n - 1 = 2^r n'$ with n' odd and $r \in \mathbb{N}$.
- Consider the sequence $b_i \equiv (a^{n'})^{2^i} \pmod{n}$ for $i = 0, 1, 2, \ldots, r$.
- We have $b_r \equiv 1 \pmod{n}$.
  Let j be the smallest index with $b_j \equiv 1 \pmod{n}$.
  Suppose $j > 0$. Then b_{j-1} is a modular square root of 1.

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.
- An odd composite integer which is not a prime power has at least four modular square roots of 1.
- Suppose $a^{n-1} \equiv 1 \pmod{n}$ (with $\gcd(a, n) = 1$).
  Write $n - 1 = 2^r n'$ with n' odd and $r \in \mathbb{N}$.
- Consider the sequence $b_i \equiv (a^{n'})^{2^i} \pmod{n}$ for $i = 0, 1, 2, \ldots, r$.
- We have $b_r \equiv 1 \pmod{n}$.
  Let j be the smallest index with $b_j \equiv 1 \pmod{n}$.
  Suppose $j > 0$. Then b_{j-1} is a modular square root of 1.
- If $b_{j-1} \not\equiv -1 \pmod{n}$, then n is composite.

Miller-Rabin Test

- An odd prime has exactly two modular square roots of 1.
- An odd composite integer which is not a prime power has at least four modular square roots of 1.
- Suppose $a^{n-1} \equiv 1 \pmod{n}$ (with $\gcd(a, n) = 1$).
  Write $n - 1 = 2^r n'$ with n' odd and $r \in \mathbb{N}$.
- Consider the sequence $b_i \equiv (a^{n'})^{2^i} \pmod{n}$ for $i = 0, 1, 2, \ldots, r$.
- We have $b_r \equiv 1 \pmod{n}$.
  Let j be the smallest index with $b_j \equiv 1 \pmod{n}$.
  Suppose $j > 0$. Then b_{j-1} is a modular square root of 1.
- If $b_{j-1} \not\equiv -1 \pmod{n}$, then n is composite.
- Compute b_0 by modular exponentiation, and then compute $b_i \equiv b_{i-1}^2 \pmod{n}$ for $i = 1, 2, \ldots$.

Miller-Rabin Test (contd.)

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod n$ or $b_{j-1} \equiv -1 \pmod n$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.
- This is true even for Carmichael numbers.

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.
- This is true even for Carmichael numbers.

Example: $n = 561 = 2^4 \times 35 + 1$, so $r = 4$ and $n' = 35$. For the base $a = 2$, we have:

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.
- This is true even for Carmichael numbers.

Example: $n = 561 = 2^4 \times 35 + 1$, so $r = 4$ and $n' = 35$.
For the base $a = 2$, we have:
$b_0 \equiv a^{n'} \equiv 263 \pmod{n}$,

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.
- This is true even for Carmichael numbers.

Example: $n = 561 = 2^4 \times 35 + 1$, so $r = 4$ and $n' = 35$.
For the base $a = 2$, we have:
$b_0 \equiv a^{n'} \equiv 263 \pmod{n}$, $b_1 \equiv a^{2n'} \equiv 166 \pmod{n}$,

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.
- This is true even for Carmichael numbers.

  Example: $n = 561 = 2^4 \times 35 + 1$, so $r = 4$ and $n' = 35$.
  For the base $a = 2$, we have:
  $b_0 \equiv a^{n'} \equiv 263 \pmod{n}$, $b_1 \equiv a^{2n'} \equiv 166 \pmod{n}$,
  $b_2 \equiv a^{2^2 n'} \equiv 67 \pmod{n}$,

Miller-Rabin Test (contd.)

- n is called a **Miller-Rabin pseudoprime** or a **strong pseudoprime** to the base a, if $b_0 \equiv 1 \pmod{n}$ or $b_{j-1} \equiv -1 \pmod{n}$ for some $j \in \{1, 2, \ldots, r\}$.
- A strong pseudoprime is also an Euler pseudoprime (but not conversely) and so a Fermat pseudoprime.
- If n is an odd composite integer (but not a prime power), then n is a strong pseudoprime to at most $1/4$-th of the bases in $\mathbb{Z}_n^*$.
- This is true even for Carmichael numbers.

Example: $n = 561 = 2^4 \times 35 + 1$, so $r = 4$ and $n' = 35$.
For the base $a = 2$, we have:
$b_0 \equiv a^{n'} \equiv 263 \pmod{n}$, $b_1 \equiv a^{2n'} \equiv 166 \pmod{n}$,
$b_2 \equiv a^{2^2 n'} \equiv 67 \pmod{n}$, $b_3 \equiv a^{2^3 n'} \equiv 1 \pmod{n}$.
Thus, 67 is a non-trivial square root of 1 modulo 561.

The Agarwal-Kayal-Saxena (AKS) Test

The Agarwal-Kayal-Saxena (AKS) Test

- Deterministic test, unconditionally polynomial-time.

The Agarwal-Kayal-Saxena (AKS) Test

- Deterministic test, unconditionally polynomial-time.
- $(x+a)^n \equiv x^n + a \pmod{n}$ (for every a) if and only if n is prime.

The Agarwal-Kayal-Saxena (AKS) Test

- Deterministic test, unconditionally polynomial-time.
- $(x+a)^n \equiv x^n + a \pmod{n}$ (for every a) if and only if n is prime.
- Compute $(x+a)^n$ and $x^n + a$ modulo n and some *suitably* chosen polynomials $x^r - 1$ with small r.

The Agarwal-Kayal-Saxena (AKS) Test

- Deterministic test, unconditionally polynomial-time.
- $(x+a)^n \equiv x^n + a \pmod{n}$ (for every a) if and only if n is prime.
- Compute $(x+a)^n$ and $x^n + a$ modulo n and some *suitably* chosen polynomials $x^r - 1$ with small r.
- A suitable $r = \mathrm{O}(\ln^6 n)$ can be found. For this r, at most $2\sqrt{r}\ln n$ values of a need to be tried.

The Agarwal-Kayal-Saxena (AKS) Test

- Deterministic test, unconditionally polynomial-time.
- $(x+a)^n \equiv x^n + a \pmod{n}$ (for every a) if and only if n is prime.
- Compute $(x+a)^n$ and $x^n + a$ modulo n and some *suitably* chosen polynomials $x^r - 1$ with small r.
- A suitable $r = \mathrm{O}(\ln^6 n)$ can be found. For this r, at most $2\sqrt{r}\ln n$ values of a need to be tried.
- The original AKS algorithm runs in $\tilde{\mathrm{O}}(\ln^{12} n)$ time.

The Agarwal-Kayal-Saxena (AKS) Test

- Deterministic test, unconditionally polynomial-time.
- $(x+a)^n \equiv x^n + a \pmod{n}$ (for every a) if and only if n is prime.
- Compute $(x+a)^n$ and $x^n + a$ modulo n and some *suitably* chosen polynomials $x^r - 1$ with small r.
- A suitable $r = \mathrm{O}(\ln^6 n)$ can be found. For this r, at most $2\sqrt{r}\ln n$ values of a need to be tried.
- The original AKS algorithm runs in $\mathrm{O}\tilde{}(\ln^{12} n)$ time.
- Lenstra and Pomerance's improvement reduces the running time to $\mathrm{O}\tilde{}(\ln^6 n)$.

How to Choose *Cryptographic* Primes?

How to Choose *Cryptographic* Primes?

- Primes are abundant in nature ($\mathbb{N}$).

How to Choose *Cryptographic* Primes?

- Primes are abundant in nature ($\mathbb{N}$).
- A random search quickly gives t-bit primes. $O(t)$ random values need to be tried. Performance increases several times by using sieving techniques.

How to Choose *Cryptographic* Primes?

- Primes are abundant in nature ($\mathbb{N}$).
- A random search quickly gives t-bit primes. $O(t)$ random values need to be tried. Performance increases several times by using sieving techniques.
- Random primes are not necessarily secure for cryptographic use.

How to Choose *Cryptographic* Primes?

- Primes are abundant in nature ($\mathbb{N}$).
- A random search quickly gives t-bit primes. $O(t)$ random values need to be tried. Performance increases several times by using sieving techniques.
- Random primes are not necessarily secure for cryptographic use.
- A **safe prime** p is an odd prime with $(p-1)/2$ prime.

How to Choose *Cryptographic* Primes?

- Primes are abundant in nature ($\mathbb{N}$).
- A random search quickly gives t-bit primes. $O(t)$ random values need to be tried. Performance increases several times by using sieving techniques.
- Random primes are not necessarily secure for cryptographic use.
- A **safe prime** p is an odd prime with $(p-1)/2$ prime.
- A **strong prime** p is an odd prime, such that

  $p-1$ has a large prime divisor (call it q),
  $p+1$ has a large prime divisor, and
  $q-1$ has a large prime divisor.

  Here, “large” means “of bit length $\geqslant 160$”.

How to Choose *Cryptographic* Primes?

- Primes are abundant in nature ($\mathbb{N}$).
- A random search quickly gives t-bit primes. $O(t)$ random values need to be tried. Performance increases several times by using sieving techniques.
- Random primes are not necessarily secure for cryptographic use.
- A **safe prime** p is an odd prime with $(p-1)/2$ prime.
- A **strong prime** p is an odd prime, such that

  $p-1$ has a large prime divisor (call it q),
  $p+1$ has a large prime divisor, and
  $q-1$ has a large prime divisor.

  Here, "large" means "of bit length $\geqslant 160$".
- The search for random primes can be modified to generate safe and strong primes.

Arithmetic in Finite Fields

Arithmetic in Finite Fields

- The most practical finite fields are the prime fields $\mathbb{F}_p$ and the fields $\mathbb{F}_{2^n}$ of characteristic 2.

Arithmetic in Finite Fields

- The most practical finite fields are the prime fields $\mathbb{F}_p$ and the fields $\mathbb{F}_{2^n}$ of characteristic 2.
- The arithmetic of $\mathbb{F}_p$ is integer arithmetic modulo p.

Arithmetic in Finite Fields

- The most practical finite fields are the prime fields $\mathbb{F}_p$ and the fields $\mathbb{F}_{2^n}$ of characteristic 2.
- The arithmetic of $\mathbb{F}_p$ is integer arithmetic modulo p.
- The arithmetic of $\mathbb{F}_{2^n} = \mathbb{F}_2(\theta)$ (with $f(\theta) = 0$) is polynomial arithmetic modulo 2 and the defining polynomial $f(x)$.

Arithmetic in Finite Fields

- The most practical finite fields are the prime fields $\mathbb{F}_p$ and the fields $\mathbb{F}_{2^n}$ of characteristic 2.
- The arithmetic of $\mathbb{F}_p$ is integer arithmetic modulo p.
- The arithmetic of $\mathbb{F}_{2^n} = \mathbb{F}_2(\theta)$ (with $f(\theta) = 0$) is polynomial arithmetic modulo 2 and the defining polynomial $f(x)$.
- In cryptographic protocols, the extension degrees n may be several thousands.

Arithmetic in Finite Fields

- The most practical finite fields are the prime fields $\mathbb{F}_p$ and the fields $\mathbb{F}_{2^n}$ of characteristic 2.
- The arithmetic of $\mathbb{F}_p$ is integer arithmetic modulo p.
- The arithmetic of $\mathbb{F}_{2^n} = \mathbb{F}_2(\theta)$ (with $f(\theta) = 0$) is polynomial arithmetic modulo 2 and the defining polynomial $f(x)$.
- In cryptographic protocols, the extension degrees n may be several thousands.
- It is necessary to study the arithmetic of such *big* polynomials.

Polynomial Arithmetic

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.
- Addition is the word-by-word XOR operation.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.
- Addition is the word-by-word XOR operation.
- For multiplication, shift and XOR.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.
- Addition is the word-by-word XOR operation.
- For multiplication, shift and XOR.
- Euclidean division is again a shift-and-subtract algorithm.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.
- Addition is the word-by-word XOR operation.
- For multiplication, shift and XOR.
- Euclidean division is again a shift-and-subtract algorithm.
- GCD can be computed by repeated Euclidean division.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.
- Addition is the word-by-word XOR operation.
- For multiplication, shift and XOR.
- Euclidean division is again a shift-and-subtract algorithm.
- GCD can be computed by repeated Euclidean division.
- Modular inverse is available from extended gcd computation.

Polynomial Arithmetic

- The coefficients of polynomials over $\mathbb{F}_2$ are bits.
- Multiple coefficients are packed in a single machine word.
- Addition is the word-by-word XOR operation.
- For multiplication, shift and XOR.
- Euclidean division is again a shift-and-subtract algorithm.
- GCD can be computed by repeated Euclidean division.
- Modular inverse is available from extended gcd computation.
- **Running times:** Let the operands be $f(x), g(x) \in \mathbb{F}_2[x]$.

$f(x) + g(x)$	$O(\max(\deg f(x), \deg g(x))$
$f(x)g(x)$	$O(\deg f(x) \times \deg g(x))$
$f(x) \text{ quot } g(x)$ and/or $f(x) \text{ rem } g(x)$	$O(\deg f(x) \times \deg g(x))$
$\gcd(f(x), g(x))$	$O(\max(\deg f(x), \deg g(x))^3)$
$g(x)^{-1} \pmod{f(x)}$	$O(\max(\deg f(x), \deg g(x))^3)$

Irreducible Polynomials

Representation of $\mathbb{F}_{2^n}$ requires an irreducible polynomial.

Irreducible Polynomials

Representation of $\mathbb{F}_{2^n}$ requires an irreducible polynomial.

Testing irreducibility of $f(x) \in \mathbb{F}_2[x]$ with $\deg f(x) = n$:

Irreducible Polynomials

Representation of $\mathbb{F}_{2^n}$ requires an irreducible polynomial.

Testing irreducibility of $f(x) \in \mathbb{F}_2[x]$ with $\deg f(x) = n$:

For $i = 1, 2, 3, \ldots, \lfloor n/2 \rfloor$, compute $d_i(x) = \gcd(x^{2^i} - x, f(x))$.
If all $d_i(x) = 1$, declare $f(x)$ as irreducible.
If some $d_i(x) \neq 1$, declare $f(x)$ as reducible.

Irreducible Polynomials

Representation of $\mathbb{F}_{2^n}$ requires an irreducible polynomial.

Testing irreducibility of $f(x) \in \mathbb{F}_2[x]$ with $\deg f(x) = n$:

For $i = 1, 2, 3, \ldots, \lfloor n/2 \rfloor$, compute $d_i(x) = \gcd(x^{2^i} - x, f(x))$.
If all $d_i(x) = 1$, declare $f(x)$ as irreducible.
If some $d_i(x) \neq 1$, declare $f(x)$ as reducible.

x^{2^i} are computed iteratively modulo $f(x)$ in order to keep their degree low (that is, less than $\deg f(x)$).

Irreducible Polynomials

Representation of $\mathbb{F}_{2^n}$ requires an irreducible polynomial.

Testing irreducibility of $f(x) \in \mathbb{F}_2[x]$ with $\deg f(x) = n$:

For $i = 1, 2, 3, \ldots, \lfloor n/2 \rfloor$, compute $d_i(x) = \gcd(x^{2^i} - x, f(x))$.
If all $d_i(x) = 1$, declare $f(x)$ as irreducible.
If some $d_i(x) \neq 1$, declare $f(x)$ as reducible.

x^{2^i} are computed iteratively modulo $f(x)$ in order to keep their degree low (that is, less than $\deg f(x)$).

Locating random irreducible polynomial of degree n:

Irreducible Polynomials

Representation of $\mathbb{F}_{2^n}$ requires an irreducible polynomial.

Testing irreducibility of $f(x) \in \mathbb{F}_2[x]$ with $\deg f(x) = n$:

For $i = 1, 2, 3, \ldots, \lfloor n/2 \rfloor$, compute $d_i(x) = \gcd(x^{2^i} - x, f(x))$.
If all $d_i(x) = 1$, declare $f(x)$ as irreducible.
If some $d_i(x) \neq 1$, declare $f(x)$ as reducible.

x^{2^i} are computed iteratively modulo $f(x)$ in order to keep their degree low (that is, less than $\deg f(x)$).

Locating random irreducible polynomial of degree n:

Generate random polynomials of degree n,
until an irreducible polynomial is generated.

The density of irreducible polynomials is about $1/n$ in the set of all monic polynomials in $\mathbb{F}_2[x]$ of degree n.

Primitive elements

Primitive elements

- $\mathbb{F}_q^*$ is cyclic.

Primitive elements

- $\mathbb{F}_q^*$ is cyclic.
- The density of primitive elements in $\mathbb{F}_q^*$ is $\phi(q-1)/(q-1) \geqslant 1/(6 \ln \ln(q-1))$ for $q \geqslant 7$.

Primitive elements

- $\mathbb{F}_q^*$ is cyclic.
- The density of primitive elements in $\mathbb{F}_q^*$ is $\phi(q-1)/(q-1) \geqslant 1/(6 \ln \ln(q-1))$ for $q \geqslant 7$.
- Checking for primitive elements requires the factorization of $q-1$. Let $q-1 = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t}$.

Primitive elements

- $\mathbb{F}_q^*$ is cyclic.
- The density of primitive elements in $\mathbb{F}_q^*$ is $\phi(q-1)/(q-1) \geqslant 1/(6 \ln \ln(q-1))$ for $q \geqslant 7$.
- Checking for primitive elements requires the factorization of $q-1$. Let $q-1 = p_1^{e_1} p_2^{e_2} \cdots p_t^{e_t}$.
- An element $a \in \mathbb{F}_q^*$ is primitive if and only if $a^{(q-1)/p_i} \neq 1$ for all $i = 1, 2, \ldots, t$.

Good Finite Fields for Cryptography

Good Finite Fields for Cryptography

- Cryptosystems based on the finite field discrete logarithm problem use $\mathbb{F}_q$ with $|q| \geqslant 1024$.

Good Finite Fields for Cryptography

- Cryptosystems based on the finite field discrete logarithm problem use $\mathbb{F}_q$ with $|q| \geqslant 1024$.
- For fast implementation, one takes $q = p \in \mathbb{P}$ or $q = 2^n$.

Good Finite Fields for Cryptography

- Cryptosystems based on the finite field discrete logarithm problem use $\mathbb{F}_q$ with $|q| \geqslant 1024$.
- For fast implementation, one takes $q = p \in \mathbb{P}$ or $q = 2^n$.
- One needs generators of $\mathbb{F}_q^*$. This requires the factorization of $q - 1$. This is an impractical requirement.

Good Finite Fields for Cryptography

- Cryptosystems based on the finite field discrete logarithm problem use $\mathbb{F}_q$ with $|q| \geqslant 1024$.
- For fast implementation, one takes $q = p \in \mathbb{P}$ or $q = 2^n$.
- One needs generators of $\mathbb{F}_q^*$. This requires the factorization of $q - 1$. This is an impractical requirement.
- Elements of $\mathbb{F}_q^*$ with prime orders $r \geqslant 2^{160}$ often suffice.

Good Finite Fields for Cryptography

- Cryptosystems based on the finite field discrete logarithm problem use $\mathbb{F}_q$ with $|q| \geqslant 1024$.
- For fast implementation, one takes $q = p \in \mathbb{P}$ or $q = 2^n$.
- One needs generators of $\mathbb{F}_q^*$. This requires the factorization of $q - 1$. This is an impractical requirement.
- Elements of $\mathbb{F}_q^*$ with prime orders $r \geqslant 2^{160}$ often suffice.
- For the field $\mathbb{F}_p$, the prime p can be so chosen that $p - 1$ has a large prime divisor r. Safe and strong primes may be used.

Good Finite Fields for Cryptography

- Cryptosystems based on the finite field discrete logarithm problem use $\mathbb{F}_q$ with $|q| \geqslant 1024$.
- For fast implementation, one takes $q = p \in \mathbb{P}$ or $q = 2^n$.
- One needs generators of $\mathbb{F}_q^*$. This requires the factorization of $q - 1$. This is an impractical requirement.
- Elements of $\mathbb{F}_q^*$ with prime orders $r \geqslant 2^{160}$ often suffice.
- For the field $\mathbb{F}_p$, the prime p can be so chosen that $p - 1$ has a large prime divisor r. Safe and strong primes may be used.
- For $\mathbb{F}_{2^n}$, we have no choice but to factor $2^n - 1$. For some values of n, a complete or partial knowledge of the factorization of $2^n - 1$ may aid the choice of a suitable r.

Suitably Large Prime Factors of $2^n - 1$

Examples

Suitably Large Prime Factors of $2^n - 1$

Examples

- $2^{1279} - 1 = r$ is a 1279-bit prime.

Suitably Large Prime Factors of $2^n - 1$

Examples

- $2^{1279} - 1 = r$ is a 1279-bit prime.
- $2^{1223} - 1 = 2447 \times 31799 \times 439191833149903 \times r$, where r is an 1149-bit prime.

Suitably Large Prime Factors of $2^n - 1$

Examples

- $2^{1279} - 1 = r$ is a 1279-bit prime.
- $2^{1223} - 1 = 2447 \times 31799 \times 439191833149903 \times r$, where r is an 1149-bit prime.
- $2^{1489} - 1 = 71473 \times 27201739919 \times 5102891746468167 \times 138228440535703689 83 \times r \times m$, where $r = 1221632661129000811383093238350060632772677648 95871$ is a 167-bit prime, and m is an 1153-bit composite integer with unknown factorization.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.
- An element α of $\mathbb{F}_q^*$ is in H if and only if $\alpha^r = 1$.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.
- An element α of $\mathbb{F}_q^*$ is in H if and only if $\alpha^r = 1$.
- Since r is prime, every non-identity element of H is a generator of H.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.
- An element α of $\mathbb{F}_q^*$ is in H if and only if $\alpha^r = 1$.
- Since r is prime, every non-identity element of H is a generator of H.

Search for α

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.
- An element α of $\mathbb{F}_q^*$ is in H if and only if $\alpha^r = 1$.
- Since r is prime, every non-identity element of H is a generator of H.

Search for α

- Choose β randomly from $\mathbb{F}_q^*$.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q - 1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.
- An element α of $\mathbb{F}_q^*$ is in H if and only if $\alpha^r = 1$.
- Since r is prime, every non-identity element of H is a generator of H.

Search for α

- Choose β randomly from $\mathbb{F}_q^*$.
- Set $\alpha = \beta^{(q-1)/r}$.

Elements of Large Orders in $\mathbb{F}_q^*$

Let r be a prime divisor of $q-1$ with $|r| \geqslant 160$.
Goal: To obtain an element $\alpha \in \mathbb{F}_q^*$ with $\operatorname{ord} \alpha = r$.

Mathematical facts

- $\mathbb{F}_q^*$ is cyclic and contains a unique subgroup H of order r.
- An element α of $\mathbb{F}_q^*$ is in H if and only if $\alpha^r = 1$.
- Since r is prime, every non-identity element of H is a generator of H.

Search for α

- Choose β randomly from $\mathbb{F}_q^*$.
- Set $\alpha = \beta^{(q-1)/r}$.
- If $\alpha \neq 1$, return α, else choose another β and repeat.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.
- Polynomial-time randomized algorithms are known.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.
- Polynomial-time randomized algorithms are known.
- A common approach is to use the following three steps.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.
- Polynomial-time randomized algorithms are known.
- A common approach is to use the following three steps.
  - **Square-free factorization (SFF):** Express $f(x)$ as a product of square-free polynomials.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.
- Polynomial-time randomized algorithms are known.
- A common approach is to use the following three steps.
  - **Square-free factorization (SFF):** Express $f(x)$ as a product of square-free polynomials.
  - **Distinct-degree factorization (DDF):** Let $f(x)$ be square-free. Express $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.
- Polynomial-time randomized algorithms are known.
- A common approach is to use the following three steps.
  - **Square-free factorization (SFF):** Express $f(x)$ as a product of square-free polynomials.
  - **Distinct-degree factorization (DDF):** Let $f(x)$ be square-free. Express $f(x) = f_1(x)f_2(x) \cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.
  - **Equal-degree factorization (EDF):** Let $f(x)$ be a square-free product of irreducible polynomials of the same known degree. Determine all these irreducible factors.

Factoring Polynomials Over Finite Fields

To factor $f(x) \in \mathbb{F}_q[x]$ with $\deg f(x) = d$. Let $q = p^n$.

- No deterministic polynomial-time algorithm is known.
- Polynomial-time randomized algorithms are known.
- A common approach is to use the following three steps.
  - **Square-free factorization (SFF):** Express $f(x)$ as a product of square-free polynomials.
  - **Distinct-degree factorization (DDF):** Let $f(x)$ be square-free. Express $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.
  - **Equal-degree factorization (EDF):** Let $f(x)$ be a square-free product of irreducible polynomials of the same known degree. Determine all these irreducible factors.
- The only probabilistic part is EDF.

Square-free Factorization (SFF)

Square-free Factorization (SFF)

- Compute the formal derivative $f'(x)$.

Square-free Factorization (SFF)

- Compute the formal derivative $f'(x)$.
- If $f'(x) = 0$, then $f(x)$ must be of the form

$$a_1 x^{pe_1} + a_2 x^{pe_2} + \cdots + a_k x^{pe_k}.$$

Write $f(x) = g(x)^p$, where

$$g(x) = a_1^{p^{n-1}} x^{e_1} + a_2^{p^{n-1}} x^{e_2} + \cdots + a_k^{p^{n-1}} x^{e_k}.$$

Recursively compute the SFF of $g(x)$.

Square-free Factorization (SFF)

- Compute the formal derivative $f'(x)$.
- If $f'(x) = 0$, then $f(x)$ must be of the form

$$a_1 x^{pe_1} + a_2 x^{pe_2} + \cdots + a_k x^{pe_k}.$$

  Write $f(x) = g(x)^p$, where

$$g(x) = a_1^{p^{n-1}} x^{e_1} + a_2^{p^{n-1}} x^{e_2} + \cdots + a_k^{p^{n-1}} x^{e_k}.$$

  Recursively compute the SFF of $g(x)$.
- If $f'(x) \neq 0$, then $f(x)/\gcd(f(x), f'(x))$ is square-free. Recursively compute the SFF of $\gcd(f(x), f'(x))$.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.
- $\gcd(f(x)/(f_1(x)f_2(x)\cdots f_{i-1}(x)), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ of degree equal to i.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.
- $\gcd(f(x)/(f_1(x)f_2(x)\cdots f_{i-1}(x)), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ of degree equal to i.
- For $i = 1, 2, 3, \ldots$, do the following:

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.
- $\gcd(f(x)/(f_1(x)f_2(x)\cdots f_{i-1}(x)), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ of degree equal to i.
- For $i = 1, 2, 3, \ldots$, do the following:
    - Compute $g_i(x) \equiv x^{q^i} - x \pmod{f(x)}$.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x) \cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.
- $\gcd(f(x)/(f_1(x)f_2(x) \cdots f_{i-1}(x)), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ of degree equal to i.
- For $i = 1, 2, 3, \ldots$, do the following:
  - Compute $g_i(x) \equiv x^{q^i} - x \pmod{f(x)}$.
  - Compute $f_i(x) = \gcd(f(x), g_i(x))$.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.
- $\gcd(f(x)/(f_1(x)f_2(x)\cdots f_{i-1}(x)), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ of degree equal to i.
- For $i = 1, 2, 3, \ldots$, do the following:
  - Compute $g_i(x) \equiv x^{q^i} - x \pmod{f(x)}$.
  - Compute $f_i(x) = \gcd(f(x), g_i(x))$.
  - Replace $f(x)$ by $f(x)/f_i(x)$.

Distinct-degree Factorization (DDF)

Let $f(x) \in \mathbb{F}_q = \mathbb{F}_{p^n}$ be a square-free polynomial of degree d.
Goal: To write $f(x) = f_1(x)f_2(x)\cdots f_d(x)$, where $f_i(x)$ is the product of irreducible factors of $f(x)$ of degree i.

- $x^{q^i} - x$ is the product of all monic irreducible polynomials of $\mathbb{F}_q[x]$ with degrees dividing i.
- $\gcd(f(x), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ with degrees dividing i.
- $\gcd(f(x)/(f_1(x)f_2(x)\cdots f_{i-1}(x)), x^{q^i} - x)$ is the product of all irreducible factors of $f(x)$ of degree equal to i.
- For $i = 1, 2, 3, \ldots$, do the following:
  - Compute $g_i(x) \equiv x^{q^i} - x \pmod{f(x)}$.
  - Compute $f_i(x) = \gcd(f(x), g_i(x))$.
  - Replace $f(x)$ by $f(x)/f_i(x)$.
  - If $f(x) = 1$, break.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} - x \mid g(x)^{q^\delta} - g(x)$, so $f(x) \mid g(x)^{q^\delta} - g(x)$.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} - x \mid g(x)^{q^\delta} - g(x)$, so $f(x) \mid g(x)^{q^\delta} - g(x)$.
- $g(x)^{q^\delta} - g(x) = g(x)(g(x)^{(q^\delta - 1)/2} - 1)(g(x)^{(q^\delta - 1)/2} + 1)$.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} - x \mid g(x)^{q^\delta} - g(x)$, so $f(x) \mid g(x)^{q^\delta} - g(x)$.
- $g(x)^{q^\delta} - g(x) = g(x)(g(x)^{(q^\delta-1)/2} - 1)(g(x)^{(q^\delta-1)/2} + 1)$.
- Compute $h(x) = \gcd(f(x), g(x)^{(q^\delta-1)/2} - 1)$.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} - x \mid g(x)^{q^\delta} - g(x)$, so $f(x) \mid g(x)^{q^\delta} - g(x)$.
- $g(x)^{q^\delta} - g(x) = g(x)(g(x)^{(q^\delta - 1)/2} - 1)(g(x)^{(q^\delta - 1)/2} + 1)$.
- Compute $h(x) = \gcd(f(x), g(x)^{(q^\delta - 1)/2} - 1)$.
- $h(x)$ is a non-trivial factor of $f(x)$ with probability $1/2$.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} - x \mid g(x)^{q^\delta} - g(x)$, so $f(x) \mid g(x)^{q^\delta} - g(x)$.
- $g(x)^{q^\delta} - g(x) = g(x)(g(x)^{(q^\delta-1)/2} - 1)(g(x)^{(q^\delta-1)/2} + 1)$.
- Compute $h(x) = \gcd(f(x), g(x)^{(q^\delta-1)/2} - 1)$.
- $h(x)$ is a non-trivial factor of $f(x)$ with probability $1/2$.
- If a non-trivial split is obtained, recursively compute the EDF of $h(x)$ and $f(x)/h(x)$.

Equal-degree Factorization (EDF)

Let $f(x) \in \mathbb{F}_q[x]$ be a square-free polynomial of degree d with each irreducible factor of degree δ.

Case 1: q is odd.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} - x \mid g(x)^{q^\delta} - g(x)$, so $f(x) \mid g(x)^{q^\delta} - g(x)$.
- $g(x)^{q^\delta} - g(x) = g(x)(g(x)^{(q^\delta-1)/2} - 1)(g(x)^{(q^\delta-1)/2} + 1)$.
- Compute $h(x) = \gcd(f(x), g(x)^{(q^\delta-1)/2} - 1)$.
- $h(x)$ is a non-trivial factor of $f(x)$ with probability $1/2$.
- If a non-trivial split is obtained, recursively compute the EDF of $h(x)$ and $f(x)/h(x)$.
- Otherwise, choose a different $g(x)$ and repeat the above steps.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} + x \mid g(x)^{q^\delta} + g(x)$, so $f(x) \mid g(x)^{q^\delta} + g(x)$.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} + x \mid g(x)^{q^\delta} + g(x)$, so $f(x) \mid g(x)^{q^\delta} + g(x)$.
- $g(x)^{q^\delta} + g(x) = g_1(x)(g_1(x) + 1)$, where
  $g_1(x) = g(x)^{2^{n\delta-1}} + g(x)^{2^{n\delta-2}} + \cdots + g(x)^2 + g(x)$.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} + x \mid g(x)^{q^\delta} + g(x)$, so $f(x) \mid g(x)^{q^\delta} + g(x)$.
- $g(x)^{q^\delta} + g(x) = g_1(x)(g_1(x) + 1)$, where
  $g_1(x) = g(x)^{2^{n\delta-1}} + g(x)^{2^{n\delta-2}} + \cdots + g(x)^2 + g(x)$.
- Compute $h(x) = \gcd(f(x), g_1(x))$.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} + x \mid g(x)^{q^\delta} + g(x)$, so $f(x) \mid g(x)^{q^\delta} + g(x)$.
- $g(x)^{q^\delta} + g(x) = g_1(x)(g_1(x) + 1)$, where $g_1(x) = g(x)^{2^{n\delta-1}} + g(x)^{2^{n\delta-2}} + \cdots + g(x)^2 + g(x)$.
- Compute $h(x) = \gcd(f(x), g_1(x))$.
- $h(x)$ is a non-trivial factor of $f(x)$ with probability $1/2$.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} + x \mid g(x)^{q^\delta} + g(x)$, so $f(x) \mid g(x)^{q^\delta} + g(x)$.
- $g(x)^{q^\delta} + g(x) = g_1(x)(g_1(x) + 1)$, where
  $g_1(x) = g(x)^{2^{n\delta-1}} + g(x)^{2^{n\delta-2}} + \cdots + g(x)^2 + g(x)$.
- Compute $h(x) = \gcd(f(x), g_1(x))$.
- $h(x)$ is a non-trivial factor of $f(x)$ with probability $1/2$.
- If a non-trivial split is obtained, recursively compute the EDF of $h(x)$ and $f(x)/h(x)$.

Equal-degree Factorization (contd.)

Case 2: $q = 2^n$.

- Take a random polynomial $g(x) \in \mathbb{F}_q[x]$ of small degree.
- $x^{q^\delta} + x \mid g(x)^{q^\delta} + g(x)$, so $f(x) \mid g(x)^{q^\delta} + g(x)$.
- $g(x)^{q^\delta} + g(x) = g_1(x)(g_1(x) + 1)$, where
  $g_1(x) = g(x)^{2^{n\delta-1}} + g(x)^{2^{n\delta-2}} + \cdots + g(x)^2 + g(x)$.
- Compute $h(x) = \gcd(f(x), g_1(x))$.
- $h(x)$ is a non-trivial factor of $f(x)$ with probability $1/2$.
- If a non-trivial split is obtained, recursively compute the EDF of $h(x)$ and $f(x)/h(x)$.
- Otherwise, choose a different $g(x)$ and repeat the above steps.

Finding Roots of Polynomials Over Finite Fields

Let $f(x) \in \mathbb{F}_q[x]$ be a non-constant polynomial.
Goal: To compute all the roots of $f(x)$ in $\mathbb{F}_q$.

Finding Roots of Polynomials Over Finite Fields

Let $f(x) \in \mathbb{F}_q[x]$ be a non-constant polynomial.
Goal: To compute all the roots of $f(x)$ in $\mathbb{F}_q$.

- Use a special case of the polynomial factoring algorithm.

Finding Roots of Polynomials Over Finite Fields

Let $f(x) \in \mathbb{F}_q[x]$ be a non-constant polynomial.
Goal: To compute all the roots of $f(x)$ in $\mathbb{F}_q$.

- Use a special case of the polynomial factoring algorithm.
- Compute $f_1(x) = \gcd(f(x), x^q - x)$, where $x^q - x$ is computed modulo $f(x)$.

Finding Roots of Polynomials Over Finite Fields

Let $f(x) \in \mathbb{F}_q[x]$ be a non-constant polynomial.
Goal: To compute all the roots of $f(x)$ in $\mathbb{F}_q$.

- Use a special case of the polynomial factoring algorithm.
- Compute $f_1(x) = \gcd(f(x), x^q - x)$, where $x^q - x$ is computed modulo $f(x)$.
- $f_1(x)$ is the product of all (pairwise distinct) linear factors of $f(x)$, that is, $f_1(x)$ has exactly the same roots as $f(x)$.

Finding Roots of Polynomials Over Finite Fields

Let $f(x) \in \mathbb{F}_q[x]$ be a non-constant polynomial.
Goal: To compute all the roots of $f(x)$ in $\mathbb{F}_q$.

- Use a special case of the polynomial factoring algorithm.
- Compute $f_1(x) = \gcd(f(x), x^q - x)$, where $x^q - x$ is computed modulo $f(x)$.
- $f_1(x)$ is the product of all (pairwise distinct) linear factors of $f(x)$, that is, $f_1(x)$ has exactly the same roots as $f(x)$.
- Call EDF on $f_1(x)$ with $\delta = 1$.

Finding Roots of Polynomials Over Finite Fields

Let $f(x) \in \mathbb{F}_q[x]$ be a non-constant polynomial.
Goal: To compute all the roots of $f(x)$ in $\mathbb{F}_q$.

- Use a special case of the polynomial factoring algorithm.
- Compute $f_1(x) = \gcd(f(x), x^q - x)$, where $x^q - x$ is computed modulo $f(x)$.
- $f_1(x)$ is the product of all (pairwise distinct) linear factors of $f(x)$, that is, $f_1(x)$ has exactly the same roots as $f(x)$.
- Call EDF on $f_1(x)$ with $\delta = 1$.
- In the EDF, one typically chooses $g(x) = x + b$ for random $b \in \mathbb{F}_q$.

Arithmetic of Elliptic Curves

Let E be an elliptic curve defined over $\mathbb{F}_q$.

Arithmetic of Elliptic Curves

Let E be an elliptic curve defined over $\mathbb{F}_q$.

- Each finite point in $E(\mathbb{F}_q)$ is represented by a pair of field elements and takes $O(\log q)$ space.

Arithmetic of Elliptic Curves

Let E be an elliptic curve defined over $\mathbb{F}_q$.

- Each finite point in $E(\mathbb{F}_q)$ is represented by a pair of field elements and takes $\mathrm{O}(\log q)$ space.
- Point addition and doubling require a few operations in the field $\mathbb{F}_q$.

Arithmetic of Elliptic Curves

Let E be an elliptic curve defined over $\mathbb{F}_q$.

- Each finite point in $E(\mathbb{F}_q)$ is represented by a pair of field elements and takes $O(\log q)$ space.
- Point addition and doubling require a few operations in the field $\mathbb{F}_q$.
- Computation of mP for $m \in \mathbb{N}$ and $P \in E(\mathbb{F}_q)$ is the additive analog of modular exponentiation and can be performed by a repeated double-and-add algorithm.

Arithmetic of Elliptic Curves

Let E be an elliptic curve defined over $\mathbb{F}_q$.

- Each finite point in $E(\mathbb{F}_q)$ is represented by a pair of field elements and takes $O(\log q)$ space.
- Point addition and doubling require a few operations in the field $\mathbb{F}_q$.
- Computation of mP for $m \in \mathbb{N}$ and $P \in E(\mathbb{F}_q)$ is the additive analog of modular exponentiation and can be performed by a repeated double-and-add algorithm.
- A random finite point $(h, k) \in E(\mathbb{F}_q)$ can be computed by first choosing h and then solving a quadratic equation in k.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.
- The algorithm is reasonably efficient for prime fields.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.
- The algorithm is reasonably efficient for prime fields.
- $|E(\mathbb{F}_q)| = q + 1 - t$ with $-2\sqrt{q} \leqslant t \leqslant 2\sqrt{q}$.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.
- The algorithm is reasonably efficient for prime fields.
- $|E(\mathbb{F}_q)| = q + 1 - t$ with $-2\sqrt{q} \leqslant t \leqslant 2\sqrt{q}$.
- Choose small primes $p_1, p_2, \ldots, p_r$ with $p_1 p_2 \cdots p_r > 4\sqrt{q}$.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.
- The algorithm is reasonably efficient for prime fields.
- $|E(\mathbb{F}_q)| = q + 1 - t$ with $-2\sqrt{q} \leqslant t \leqslant 2\sqrt{q}$.
- Choose small primes $p_1, p_2, \ldots, p_r$ with $p_1 p_2 \cdots p_r > 4\sqrt{q}$.
- Determine t modulo each p_i.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.
- The algorithm is reasonably efficient for prime fields.
- $|E(\mathbb{F}_q)| = q + 1 - t$ with $-2\sqrt{q} \leqslant t \leqslant 2\sqrt{q}$.
- Choose small primes $p_1, p_2, \ldots, p_r$ with $p_1 p_2 \cdots p_r > 4\sqrt{q}$.
- Determine t modulo each p_i.
- Combine these values by CRT.

Point Counting

For selecting cryptographically good elliptic curves E over $\mathbb{F}_q$, we need to count the size of $E(\mathbb{F}_q)$.

- The **SEA (Schoof-Elkies-Atkins) algorithm** is used.
- The algorithm is reasonably efficient for prime fields.
- $|E(\mathbb{F}_q)| = q + 1 - t$ with $-2\sqrt{q} \leqslant t \leqslant 2\sqrt{q}$.
- Choose small primes $p_1, p_2, \ldots, p_r$ with $p_1 p_2 \cdots p_r > 4\sqrt{q}$.
- Determine t modulo each p_i.
- Combine these values by CRT.
- This gives a unique value of t in the range $-2\sqrt{q} \leqslant t \leqslant 2\sqrt{q}$.

Good Elliptic Curves for Cryptography

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.
- Randomly select an elliptic curve E over $\mathbb{F}_q$.

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.
- Randomly select an elliptic curve E over $\mathbb{F}_q$.
- Determine $|E(\mathbb{F}_q)|$.

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.
- Randomly select an elliptic curve E over $\mathbb{F}_q$.
- Determine $|E(\mathbb{F}_q)|$.
- If E is anomalous or supersingular, choose another E and repeat.

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.
- Randomly select an elliptic curve E over $\mathbb{F}_q$.
- Determine $|E(\mathbb{F}_q)|$.
- If E is anomalous or supersingular, choose another E and repeat.
- Factor $|E(\mathbb{F}_q)|$, and check whether E has a point of prime order $r \geqslant 2^{160}$.

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.
- Randomly select an elliptic curve E over $\mathbb{F}_q$.
- Determine $|E(\mathbb{F}_q)|$.
- If E is anomalous or supersingular, choose another E and repeat.
- Factor $|E(\mathbb{F}_q)|$, and check whether E has a point of prime order $r \geqslant 2^{160}$.
- If so, return E.

Good Elliptic Curves for Cryptography

- First, choose a ground field $\mathbb{F}_q$. Security requirements demand $|q|$ in the range 160–300 bits.
- Randomly select an elliptic curve E over $\mathbb{F}_q$.
- Determine $|E(\mathbb{F}_q)|$.
- If E is anomalous or supersingular, choose another E and repeat.
- Factor $|E(\mathbb{F}_q)|$, and check whether E has a point of prime order $r \geqslant 2^{160}$.
- If so, return E.
- Otherwise, choose another E and repeat.