Cryptography: notes

Niccolò Simonato

November 21, 2022

Contents

Chapter 1

Elements of Number Theory

1.1 Definitions of Number Theory

1.1.1 The cyclic group $\mathbb{Z}_n^*$

The cyclic group $\mathbb{Z}_n^*$ is defined as

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n : (a, n) = 1\}$$

The **generator** of $\mathbb{Z}_n^*$ is a number in $\mathbb{Z}_n^*$ such that:

$$\forall a \in \mathbb{Z}_n^* : \exists i : a \equiv_m g^i$$

For this reason, $\mathbb{Z}_n^*$ is also referred as $< g >$.
An interesting property is that only for $[1, 2, 4, \phi, \phi^\alpha, 2\phi^\alpha]$, $\mathbb{Z}_n^*$ is cyclic, where ϕ is a prime number and ϕ^α is a power of a prime number.

1.1.2 Pseudoprime number

For an integer $a > 1$, if a composite integer x divides $a^{x-1} - 1$, then x is called a **Fermat pseudoprime** to base a.

$$x \text{ is spsp(a)} \iff x | a^{x-1} - 1$$

In other words, a composite integer is a Fermat pseudoprime to base a if it successfully passes the Fermat primality test for the base a.
x is $spsp(a)$ is a predicate, that means that some x is a *strong pseudoprime number in base a*.

1.1.3 Carmichael numbers

Carmichael numbers are defined as follows:

- Let n be a composite number;
- Then:
$$b^n \equiv_n b \implies b \text{ is a } \textbf{Carmichael number}$$

1.2 Reminders of Modular Arithmetic

1.2.1 Little Fermat's Theorem

Theorem 1 (Little Fermat's Theorem)**.** *Consider what follows:*

- *Let p be a prime number and $p \nmid a$.*
- *Then*
$$a^{p-1} \equiv_p 1$$

Proof.
- Let's consider:
$$A = \{na \bmod p : n \in [1, p-1]\}$$
- A has all distinct elements due to the Bijection Lemma (Lemma 1).
- Then
$$A = \mathbb{Z}_n^*$$
- Then,
$$\prod_{n \in A} n \equiv_p \prod_{n=1}^{p-1} na \implies (p-1)! \equiv_p (p-1)! a^{p-1}$$
- Therefore, $a^{p-1} \equiv_p 1$ for the Wilson's Theorem 1.2.2.

□

1.2.2 Wilson's Theorem

Theorem 2 (Wilson's Theorem)**.** *Consider what follows:*

- *Let $n \in \mathbb{N} \setminus \{0, 1\}$*
- *Then,*
$$(n-1)! \equiv_n -1$$

Proof. The proof is by induction on n:

- **Base case**: $n = 2$

$$1! \equiv_2 -1$$

- **Inductive step**: The theorem is assumed to be true up until $n-1$. Let's consider the case of n:
  - Consider the polynomial

    $$g(x) = (x-1)(x-2)\dots(x-(n-1))$$

  - g has degree $p-1$ and costant term $(p-1)!$. It's roots are in $[1, p-1]$.
  - Consider

    $$h(x) = x^{p-1} - 1$$

    h has also degree $p-1$ and leading term x^{p-1}.
  - Let $f(x) = g(x) - h(x)$.
  - Then, f has degree at most $p-2$ (since the leading terms cancel), and modulo p also has the $n-1$ roots $1, 2, ..., n-1$.
  - But Lagrange's theorem says it cannot have more than $p-2$ roots.
  - $$\therefore f \equiv_n 0$$
  - Its costant term,

    $$(n-1)! + 1 \equiv_n 0 \iff (n-1)! \equiv_n -1$$

□

1.2.3 Euler-Fermat's Theorem

Theorem 3 (Euler-Fermat's Theorem)**.** *Consider what follows:*

- *Let $n \in \mathbb{Z}$ be a number.*
- *Then,*

$$a^{\varphi(n)} \equiv_n 1$$

1.2.4 Bézout's identity

Theorem 4 (Bezout's identity)**.** *Consider what follows:*

- *Let a and b be integers with greatest common divisor d.*
- *Then there exist integers x and y such that $ax + by = d$.*
- *Moreover, the integers of the form $az + bt$ are exactly the multiples of d.*

1.2.5 Chinese Reminder's Theorem

Theorem 5 (Chinese Reminder's Theorem)**.** *Given a system of congruences:*

$$\begin{aligned} x &\equiv_{m_1} a_1 \\ x &\equiv_{m_2} a_2 \\ \ldots \\ x &\equiv_{m_r} a_r \end{aligned}$$

In which each module is prime with each others, that is:

$$\forall i \neq j : (m_i, m_j) = 1$$

then ***there exists a simultaneous solution*** *x* ***to all of the congruences****, and* ***any two solutions are congruent to one another modulo****.*

$$M = m_1 m_2 \ldots m_r$$

Proof. Consider what follows:

- Suppose that x' and $x"$ are two solutions.
- Let $x = x' - x''$.
- Then x must be congruent to 0 modulo each m_i and hence modulo M.
- Let $M_i = \frac{M}{m_i}$, to be the product of all of the moduli except for the i-th.
- Then $GCD(m_i, M_i) = 1$ and therefore $\exists N_i : M_i N_i \equiv_{m_i} 1$.
- Set $x = \sum_i a_i M_i N_i$;
- Then, for each i we see that the terms in the sum other than the i-th term are all divisible by m_i, because $m_i | M_j$ when $i \neq j$.
- Thus, for each i we have: $x \equiv_{m_i} a_i M_i N_i \equiv_{m_i} a_i$,

$\square$

1.2.6 Bijection of a modular function Lemma

Lemma 1 (Bijection of a modular function Lemma)**.** *Consider:*

- $$a \in \mathbb{Z}_n^*$$
- $$f : \mathbb{Z}_n \to \mathbb{Z}_n$$

Then, f is a bijection.

Proof. Part 1: f is injective.

- Assume $f(x_1) = f(x_2) \in \mathbb{Z}_n \iff ax_1 \equiv_n ax_2$.
- Then $\exists k \in \mathbb{Z} : ax_1 - ax_2 = kn$.
- Therefore,

$$a(x_1 - x_2) = kn \iff a^{-1}a(x_1 - x_2) = a^{-1}kn$$
$$\iff (x_1 - x_2) = a^{-1}kn \iff x_1 - x_2 \equiv_m 0.$$

- $\therefore x_1 \equiv_m x_2$, so f is an injection.

Part 2: f is surjective.

- Let $b \in \mathbb{Z}_n$.
- Let $\overline{x} = a^{-1}b \in \mathbb{Z}_n$.
- Then, $f(\overline{x}) \equiv_m a \cdot \overline{x}$
  $\equiv_m a \cdot a^{-1} \cdot b \equiv_m b$.
- Therefore, f is surjective.

Since f is injective $\wedge$ f is surjective, then f is bijective. □

1.2.7 Euler's φ function Lemmas

Lemma 2 (Sum of the prime divisors' φ-function)**.**

$$\forall n \in \mathbb{N} \setminus \{0\} : \sum_{\frac{n}{d}} \varphi(d) = n$$

Where $\frac{n}{d}$ is the set of prime divisors of n.

Proof. The proof is by **contradiction**:

- Let B be
$$\{\frac{h}{n} : h \in \mathbb{Z}_n \wedge n \in \mathbb{N}\}$$
- Therefore,
$$B = \cup_{\frac{n}{d}} \{a \in \{1, \dots, n\} \wedge (a, d) = 1\}$$
- Then,
$$n = |B| = \sum_{\frac{d}{n}} \varphi(d)$$
- Consider
$$(a_1, d_1) = 1 \wedge (a_2, d_2) = 1, \text{where} d_1 | n \wedge d_2 | n$$
- Then,
$$\frac{a_1}{d_1} = \frac{a_2}{d_2} \iff a_1 d_2 = a_2 d_1$$
- Then,
$$d_1 | a_1 d_2 \implies d_1 | d_2 \wedge d_2 | a_2 d_1 \implies d_2 | d_1$$
- Therefore, $d_1 = d_2$. This is clearly a *contradiction*, because in B each divisor is counted once.

□

Lemma 3 (Number of divisors of a prime number's power)**.** *Let $p \in \mathbb{N}$ be a prime number, and $\alpha \in \mathbb{N}$.*
Then,

$$\varphi(p^\alpha) = p^{\alpha - 1}(p - 1)$$

Proof. The proof is by **induction** on α.
Case base: $\alpha = 1$

$$p = \sum_{\frac{d}{p}} \varphi(d) = \varphi(1) + \varphi(p) = 1 + \varphi(p) \implies \varphi(p) = p - 1$$

Case base: $\alpha = 2$

$$\begin{aligned} p^2 &= \sum_{\frac{d}{p^2}} \varphi(d) = \varphi(1) + \varphi(p) + \varphi(p^2) \\ &= 1 + p - 1 + \varphi(p^2) \implies \varphi(p^2) = p^2 - p - 1 + 1 \\ &= p \cdot (p-1) = p^{\alpha-1}(p-1) \end{aligned}$$

Inductive Step: we can now assume that $\varphi(p^\alpha) = p^{\alpha-1}(p-1)$ up until $\alpha - 1$. We'll proceed now to demonstrate that this is also valid for each α.

$$\begin{aligned} p^\alpha &= \sum_{\frac{d}{p^\alpha}} \varphi(d) \\ &= \sum_{i=0}^{\alpha-1} \varphi(p^i) + \varphi(p^\alpha) \\ &\implies \varphi(p^\alpha) = p^\alpha - p^{\alpha-1} = p^{\alpha-1}(p-1). \end{aligned}$$

□

1.2.8 Multiplicativity of the Euler's φ-function

Theorem 6 (Multiplicativity of the Euler's φ-function)**.** *Let's consider the Euler's function $\varphi(n) = |\mathbb{Z}_n^*$.*
Let's consider that $n = \prod_{i=1}^{r} p_i^{\alpha_i}$, where p_i is a prime number, and $\alpha_i \in \mathbb{N}$.
Then, =

1.3 Theorems for Cryptography purposes

1.3.1 The Miller-Rabin Theorem

Theorem 7 (The Miller-Rabin Theorem)**.** *The **Miller-Rabin Theorem states that**:*

- *Let p be a prime number, such that $p \geq s$*
- *Let $a \in \mathbb{N} : p \nmid a$*

- *Let $p-1=2^s d$, where d is odd.*
- *Then,*
$$a^d \equiv_p 1 \vee a^{2^r d} \equiv_p -1, \textit{for some } r \in \{0,1,\ldots,s-1\}$$

Proof. Let's consider the first case:

- $x^2 \equiv_p \pm 1$, for some $x \in \mathbb{N}$.
- $a \in \mathbb{N} \wedge p \nmid a \implies a^{p-1} \equiv_p 1$, due to the Little Fermat's Theorem 1.2.1.
- Since
$$p-1=2^s \cdot d \wedge 2 \nmid d \implies a^{p-1} \equiv_p (a^{\frac{p-1}{2}})^2 \equiv_p 1$$
- Then, is proved that
$$a^{\frac{p-1}{2}} \equiv_p \pm 1$$
- Therefore:
$$a^{2^r d} \equiv_p -1 \text{for} r = s-1$$

Consider now the case when $a^{2^r d} \equiv_p 1$:

- If $s=1$, then $\frac{p-1}{2}=d \implies a^d \equiv_p 1$
- If $s=2$, then $\frac{p-1}{4}=d \implies a^d \equiv_p 1$
- In general, for $s \geq 3$ we can consider successive square roots, until $r=0$: then, $a^{\frac{p-1}{2^s}} \equiv_p 1$

□

1.3.2 Miller's Theorem

Theorem 8 (Miller's Theorem). *The Miller's Theorem states what follow:*

- *Let n be a composite and odd number.*
- *Then, n is $spsp(a)$ for at most $\frac{1}{4}$ of the $a_i \in \mathbb{Z}_n^*$*

The following lemma is a consequence of this theorem.

Lemma 4 (Corollary of the Miller's Theorem). *If n is composite and odd, then*

$$\exists a \in \mathbb{Z}_n^* : a \leq b, \textit{ such that } n \textit{ is not } spsp(a)$$

1.3.3 Ankey-Montgomery-Bach Theorem

Theorem 9 (Ankey-Montgomery-Bach Theorem). *The Ankey-Montgomery-Bach Theorem states that:*

- *If the GRH*[1] *holds;*
- *If n is composite and odd;*

Then,

$$\exists \in [2, 2log^2(n)] \text{ such that } n \text{ is not } spsp(a)$$

1.4 Euler's Product Theorem

Theorem 10 (Euler's Product Theorem). *If $\mathbb{R}e(s) > 1$, then*

$$\zeta(s) = \sum_{n=1}^{+\infty} \frac{1}{n^2} = \prod_p (1 - \frac{1}{p^2})^{-1}$$

[1] Generalized Riemann Hypothesis

Chapter 2

Efficient implementations of elementary operations

2.1 Notation

- Let b be a numeric base.
- Let n be a number in N.
- Length of a number: $l_b(n)$, k. It's equal to $\log(n)$.
- (a, b) is the Maximum Common Divisor of a, b.
- Let $n \in N$: $n = (d_{k-1}, d_{k-2}, \dots, d_1, d_0)$[1].
- $\varphi(n)$: the number of elements a in $[1, n]$ such that $(a, n) = 1$.
- $\equiv_p$ is the equivalence in base p. Ex.: $5 \equiv_3 = 5 \bmod 3 = 2$.
- Let $\mathbb{Z}_n[X]$ be the set of polynomials in X with coefficients in $\mathbb{Z}_n$.

2.2 Classification of the algorithms' complexity

In order to better identify the classes of complexity of the algorithms, the following 3 classes are defined:

- Polynomial time: $O(\log^{\alpha}(n))$ bit operations, where $\alpha > 0$.
- Exponential time: $O(exp(c \cdot \log(n)))$ bit operations, where $c > 0$.
- Sub-exponential time: $O(exp(c \cdot \log(n))^{\alpha})$ bit operations, where $c > 0, \alpha \in]0, 1[$.

[1] $d_{k-1} \neq 0$

2.3 Basic bit operations

2.3.1 Sum of 3 bits - 3-bit-sum

Given n_1, n_2 their sum produces $n_1 + n_2$ and their carry.
Since $n_1, n_2 \in [0,1]$, then this operation can be done in $O(1)$.

2.3.2 Summation of 2 numbers

Given n_1, n_2 their sum produces $n_1 + n_2$.
Since the sum is computed bit by bit, the 3-bit-sum is performed

$$\max\{\text{lenght}(n_1), \text{length}(n_2)\}$$

times.
Each time the carry-on of the previous sum is added to the two digits.
This operation has then complexity:

$$O(\max\{lenght(n_1), \text{length}(n_2)\}) = O(\max\{\log(n_1), \log(n_2)\})$$

2.3.3 Summation of n numbers

The summation of n numbers is simply the sum of two numbers, but performed $n-1$ times.
Let's assume that:

$$\forall i \in [1, n] : M \geq a_i$$

The complexity of this operation is then:

$$O((n-1) \cdot \log(M)) = O(n)$$

2.3.4 Product of 2 numbers

If we consider the classic implementation of the binary multiplication, that is just a sequence of summations.

- The number of summations to execute is equal to the length of the smallest number, $O(\log(n))$.
- The maximum cost of a single summation is $O(\log(m))$.

- Then, $T(m \cdot n) = O(\log(m) \cdot \log(n))$, but, if we consider the worst case[2], that becomes $O(\log^2(m))$.

2.3.5 Division of 2 numbers

Let's consider the division of two numbers m, n. This operation consists in finding two numbers q, r such that $m = q \cdot n + r$.
This is achieved by performing a succession of subtractions, until the ending condition $0 \le r < n$ is reached.

- Let's consider that the number of steps of this algorithm is $O(\log(q))$.
- Moreover:
$$q \le m \therefore \#steps = O(\log(m))$$
- It's assumed that the cost of the single subtraction is $O(\log(n))$.
- Then:
$$T(\frac{m}{n}) = O(\log(n) \cdot \log(m))$$

2.3.6 Production of n numbers

Let's assume that:
$$j \in [1, s+1] \text{ and } M = \max(m_j)$$
The cost of the operation $\prod_{j=1}^{s+1} m_j$ is then $O(s^2 \cdot \log^2(M))$. This will now be considered our inductive hypothesis.
Proof by induction, on s:

- (1) **Base case**:
$$T(m_1 \cdot m_2) = O(\log(m_1) \cdot \log(m_2)) = O(k_1 \cdot k_2) \le c \cdot k_M^2$$
- (2) **Base case**:
$$\begin{aligned} T(m_1 \cdot m_2 \cdot m_3) &= T(m_1 \cdot m_2) + T((m_1 \cdot m_2) \cdot m_3) \\ &\le c \cdot k_M^2 + c \cdot k_{m_1 \cdot m_2} + k_{m_3} \\ &\le c \cdot k_M^2 + c \cdot k_{M^2} + k_M \end{aligned}$$

[2]two numbers that are equally large

- Inductive step: we assume the inductive hypothesis to be true up to s. Then:

$$\begin{aligned}
T(\prod_{j=1}^{s+1} m_j) &= T([\prod_{j=1}^{s} m_j] \cdot m_{s+1}) \\
&\leq c \cdot \sum_{j=1}^{s} (j \cdot k_M^2) \\
&= c \cdot k_M^2 \cdot \frac{s \cdot (s-1)}{2} \\
&= O(k_M^2 \cdot s^2) \\
&= O(s^2 \cdot \log^2(M))
\end{aligned}$$

Applications

- An analogous dimonstration can be used to prove that

$$T(\prod_{j=1}^{s+1} m_j \bmod n) = O(s \cdot \log^2(M))$$

- This proof can be used to show that

$$T(m!) = O(m \cdot \log^2(m))$$

2.4 Optimizations of more complex operations

2.4.1 Powers & Modular Powers

Let's consider what follows:

$$a^n = a \cdot a \cdot a \cdot \dots \cdot a$$

Where a is repeated n times.

Trivial implementation

The most trivial implementation would consists in computing the product $\prod_{j=1}^{n} a$. This would imply a cost of $O(n^2 \cdot \log^2(a))$.
What follows is a suggestion that could improve the cost of this operation.

Square & Multiply method for scalars, modular powers

Each number in $\mathbb{Z}$ can be represented in a binary notation.
Let's consider

$$n = (b_{k-1}, b_{k-2}, \ldots, b_0) = \sum_{i=0}^{k-1} b_i \cdot 2^i$$

It is clear that we can spare a lot of computational resources by just calculating the powers of 2 and summing the ones that have $b_i = 1$. The following algorithm explains the procedure in detail. Let's compute the complexity of this algorithm:

Algorithm 1: The Square & Multiply Method

```
1  $P \leftarrow 1$;
2  $M \leftarrow m$;
3  $A \leftarrow a \bmod n$;
4  while $M > 0$ do
5      $q \leftarrow \lfloor \frac{M}{2} \rfloor$;
6      $r \leftarrow M - s \cdot q$;
7      if $r = 1$ then
8          $P \leftarrow P \cdot A \bmod n$;
9      end
10     $A \leftarrow A^2 \bmod n$;
11     $M \leftarrow q$;
12 end
13 return $P$
```

- All of the assignments $X \leftarrow Y$ are implemented in $O(\log(Y))$.
- The cost of 3 is $O(\log(a) \cdot \log(n))$, because it ensures that $A \leq n$.
- Instructions 5 and 6 can be executed in $O(\log(m))$.
- Instrucion 8 can be executed in $O(\log^2(n))$.
- The cost of 10 is $O(\log^2(n))$, because it ensures that $A \leq n$.
- The loop is executed $\log(m)$ times.
- The total cost of this algorithm is then $O(\log(n) \cdot \log(a) + \log(m) \cdot (\log^2(n) + \log(m)))$
  $= O(\log^2(m) + \log(m) \cdot \log(n))$.

This algorithm can be easily converted for the computation of non-modular powers by applying the following changes:

```
1 $A \leftarrow a \bmod n \Longrightarrow A \leftarrow a$;
2 $P \leftarrow P \cdot A \bmod n \Longrightarrow P \leftarrow P \cdot A$;
3 $A \leftarrow A^2 \bmod n \Longrightarrow A \leftarrow A^2$;
```

Square & Multiply method for polynomials

Let's consider $\Re = \frac{\mathbb{Z}_n[x]}{x^r-1}$. The modular powers of the elements in this set can be computed by using a variation of the Square & Multiply method.

- Assume that $f, g \in \Re$.
- Let
$$h(x) = f(x) \cdot g(x) = \sum_{j=0}^{2r-2} h_j \cdot x^j$$
- Where
$$h(j) = (\sum_{i=0}^{j} f_i \cdot g_{j-i} \bmod n) \bmod n$$
- Then:
$$T(h_j) = O(j \cdot \log^2(n))$$
- Then:
$$T(h(x)) = O(\sum_{j=0}^{\log^2(n)}) = O(r^2 \cdot \log^2(n))$$

This result will be useful in the following computations.
Let's now consider $\frac{h(x)}{x^r-1}$.

- When $j > r-1$, $h_j \cdot x^j$ does not take any part in the computations.
- When $j = r$, then:
$$\frac{h_r \cdot x^r}{x^r - 1} = h_r + \frac{h_r}{x^r - 1}$$
Or, in other words: $h_r \cdot x^r \equiv_{x^r-1} h_r$.
- In the other cases:
$$h_{r+i} \cdot x^{r+i} \equiv_{x^r-1} h_{r-i} \cdot x^{r-i} \text{ for } 1 \leq i \leq r-2$$

Then:

$$h(x) \equiv_{(n,x^r-1)} f(x)\cdot g(x) \equiv [\sum_{j=0}^{r-2}((h_j + h_{r+j}) \bmod n)\cdot x^j] + h_{r-1}\cdot x^{r-1}$$

Therefore:

$$T(h(x) \bmod (n, x^r-1)) = O(r^2 \cdot \log^2(n))$$

Finally, we can analyze the complexity of the computation of the modular power $h(x)$ elevated to n.
In order to optimize the use of the computational resources, we can use a variation of the Square & Multiply method (See 1); although, this time, the computation of the partial products will be conducted by using the previously explained procedure (See 3).
The cost of this method would then be

$$\begin{aligned} T(\#Loops\cdot(h(x) \bmod (n,x^r-1))) &= O(\log(n)\cdot r^2\cdot\log(n)) \\ &= O(r^2\cdot\log^3(n)) \end{aligned}$$

2.4.2 Finding the b-expansion of n (n_b)

Let's consider the cost in bit operations of the conversion of a number n to a new base b.
The algorithm used will be the classical: a succession of divisions by b.

- Let's consider
$$r_i \in \{0, 1, \ldots, b-1\}$$
- Let
$$n_b = (r_{k+1}, r_k, \ldots, r_1, r_0)$$
- Then:
$$\begin{aligned} n &= q_0\cdot b + r_0 \\ q_0 &= q_1\cdot b + r_1 \\ &\ldots \\ q_k &= 0\cdot b + q_k \end{aligned}$$
- Consider then that
$$q_k = r_{k+1}$$

- And that
$$b^{k+2} > n > b^{k+1} \iff (k+2)\cdot\log(b) \le \log(n) \le (k+1)\cdot\log(b)$$

- Therefore,
$$k = O(\frac{\log(n)}{\log(b)})$$

We can now proceed with the computation of the cost of this operation:

$$\begin{aligned} T(n_b) &= T(\#Divisions \cdot q_i \bmod b) \\ &= O(\frac{\log(n)}{\log(b)} \cdot \log(n)\cdot\log(b)) \\ &= O(\log^2(n)) \end{aligned}$$

2.4.3 How to use Bezout formula to compute modular inverses

An efficient way of computing the modular inverse of a given number a with in the group $\mathbb{Z}_m^*$ uses the corollary of the *Bezout identity* and the *Extended Euclidean Algorithm.*
That is, given $a \cdot x \equiv_m 1$, we want to compute x.
Extended Euclidean Algorithm, given a, m computes the $gcd(a, m)$ and also returns the coefficients x, y for which $ax + my = 1$.
At this point, the modular inverse of a in $\mathbb{Z}_m^*$ is x:

- Let's consider that $ax + my \equiv_m 1$;
- Since $my \equiv_m 0$, then $ax \equiv_m 1$;
- $\therefore$ $x \bmod n$ is the modular inverse of a in $\mathbb{Z}_m^*$ for its definition.

The complexity of this operation is then $O(\log(x)\log(n))$, because we have to compute the remainder of the division between x and n (this does not take into account the execution of the *Extended Euclidean Algorithm*).

2.4.4 Computing the order of an element in a cyclic group

The order of an element a in $\mathbb{Z}_p^*$ ($m = order(a)$) is the minimum m such that $a^m \equiv_p 1$.
This problem is computationally hard, because the most efficient way to compute order(a) is to brute force its value.
The only optimization available is that we don't have to compute the modulars powers of a from scratch each time, but we can save the results at each iteration. Therefore, at each step we can only compute the modular product $(a^{p-1} \cdot a) \bmod p$, that has a cost $O(\log^2(p))$. The cost of this algorithm is then $O(order(a) \cdot \log^2(p))$, because we have to compute $a^i \bmod p$ for each attempt to find $order(a)$.

2.4.5 Extended Euclidean Algorithm

The Extended Euclidean Algorithm is a variation of the classic Euclidean Algorithm, that computes the GCD between two numbers a, b.
It also provides the coefficients λ, μ such that $\lambda \cdot a + \mu \cdot b = \text{GCD}(a, b)$.
This algorithm has a cost $O(\log^3(max\{a, b\}))$.

Algorithm 2: The Extended Euclidean Algorithm

```
Data: $a, b$
Result: $(\lambda, \mu, GCD(a, b))$
$old\_r \leftarrow a$;
$r \leftarrow b$;
$old\_s \leftarrow 1$;
$s \leftarrow 0$;
$old\_t \leftarrow 0$;
$t \leftarrow 1$;
while $r \neq 0$ do
    $quotient \leftarrow floor(old\_r / r)$;
    $old\_r \leftarrow r$;
    $old\_s \leftarrow s$;
    $old\_t \leftarrow t$;
    $r \leftarrow old\_r - quotient \cdot r$;
    $s \leftarrow old\_s - quotient \cdot s$;
    $t \leftarrow old\_t - quotient \cdot t$;
end
return (s,t,old_r)
```

2.4.6 Computation of square and m-th root of n

The following algorithm can be used to compute efficiently $\lfloor \sqrt[m]{n} \rfloor$.
It is assumed that the length of the result is known and is l.
Let's consider the cost of this algorithm:

- Computing x_i^m has cost $O(\log^2(n))$
- Comparing x_i^m and n has cost $O(\log(n))$.
- The length of the loop is $O(\log(n))$ iterations.
- The total cost is therefore $O(\log^3(n))$.

Algorithm 3: The Efficient m-th root of n

```
   Data: $n, m$
   Result: $\lfloor \sqrt[m]{n} \rfloor$
1  $x_0 \leftarrow 2^{l-1}$;
2  for $i \leftarrow 1$ to $l-1$ do
3      $x_i \leftarrow x_{i-1} + 2^{l-i-1}$;
4      if $x_i^m > n$ then
5          $x_i \leftarrow x_{i-1}$;
6      end
7  end
8  return $x_{l-1}$
```

2.4.7 Compute n, m given n^m

We can extract the base and the exponent of an integer by making different attempts. Let's consider the cost of this operation:

- We have to make at most m attempts by brute force;
- At each attempt we have to compute $\lfloor sqrt[m_i] n^m \rfloor$;
- This operation has total cost of

$$\sum_{m=3}^{\log(n)} O(\frac{\log(n)}{m} \cdot \log^2(m) \cdot \log(m)) + O(\log^3(n))$$

  [3];

- That is equal to

$$O(\log^3(n)) \sum_{m=3}^{\log(n)} O(\frac{\log^3(m)}{m}) + O(\log^3(n))$$

- $$\sum_{m=3}^{\log(n)} O(\frac{\log^3(m)}{m})$$

  can be approximated by calculating the correspondant integral, to $O(\log\log(n))$.

- The final cost is therefore

$$O(\log^3(n) \cdot (\log\log(n))^2) = O(\log^{3+\epsilon}), \text{ with } \epsilon \in (0,1)$$

[3] $\frac{\log(n)}{m}$ is the length of the loop

Chapter 3

Algorithms for primality test

3.1 Miller-Rabin probabilistic primality algorithm

Algorithm 4: Miller-Rabin primality test

```
Data: $n \in \mathbb{N}$, an odd number
Result: $r$
1  Compute $s, d$ such that: $n - 1 = 2^s \cdot d$;
2  Randomly choose $a \in \mathbb{Z}_n^*$;
3  if $(a, n) > 1$ then
4      return $n$ is composite;
5  end
6  $b \leftarrow a^d \bmod n$;
7  if $b \equiv_n \pm 1$ then
8      return $n$ is prime or spsp;
9  end
10 $e \leftarrow 0$;
11 while $b \not\equiv_n \pm 1 \wedge e \leq s - 2$ do
12     $b \leftarrow b^2 \bmod n$
13 end
14 if $b \not\equiv_n 1$ then
15     return $n$ is composite;
16 end
17 return $n$ is prime or spsp;
```

3.1.1 Computational complexity of the Miller-Rabin test

Testing the primality for a single value of a has cost of $O(log^3(n))$ b.o.. Although, we could test for each value in $\mathbb{Z}_n^*$, and that would cost $O(\varphi(n) \cdot log^3(n))$ b.o..

3.2 Primality Test in $\mathbb{P}$, AKS algorithm

3.2.1 Useful Lemmas

Lemma 5 (Newton's formula lemma)**.** *This lemma states as follows:*
n is prime $\iff (x+b)^n \equiv_n x^n + b$.

Proof. The proof is by identity:

- $(x+b)^n \equiv_n x^n + b \implies n$ is prime.
  - By *contradiction*:
    - Assume that n is composite.
    - Then $\exists p|n$, where $p < n$ is a prime number.
    - Consider
$$\binom{n}{p} = \frac{n \cdot n-1 \cdot \dots \cdot n-p+1}{p \cdot p-1 \cdot \dots \cdot 1} > 1$$
    - Assume that:
$$p^\alpha || n$$
    - Then
$$p \nmid n$$
    - Let
$$N = \prod_{i=n-p+1}^{n-1} i$$
    - Let
$$M = \prod_{i=1}^{p-1} i$$
    - Then,
$$\binom{n}{p} = \frac{p^\alpha \cdot N}{p \cdot M} = p^{\alpha-1} \cdot \frac{N}{M}$$
    - $$(N,p) = (M,p) = 1 \implies p^{\alpha-1} | \binom{n}{p} \wedge p^\alpha \nmid \binom{n}{p}$$

        * Therefore:

$$p^{\alpha-1} || \binom{n}{p} \text{ and } \binom{n}{p} \equiv_p 0$$

        * But this is a **contradiction**, since $\binom{n}{p} \not\equiv_p 0$, therefore n is prime.

- n is prime $\implies (x+b)^n \equiv_n x^n + b$
    - Consider that $p | \binom{p}{k}$, for $k < n$.
    - Since $\binom{p}{k} \equiv_p 0$, then

$$(x+b)^p = \sum_{k=0}^{p} \binom{p}{k} x^{p-k} \cdot b^k \equiv_p x^p + b^p \equiv_p b$$

□

Lemma 6 (Nair's Lemma)**.** *This lemma states as follows:*

- *Let $m \geq 7 \in \mathbb{Z}$*
- *Let* LCM(x, y) *be the Least Common Multiplier of x and y.*
- *Let $n \leq m \in \mathbb{Z}$.*
- *Then:*

$$LCM(m,n) \geq 2^m$$

Lemma 7 (AKS Lemma)**.** *This lemma states as follows:*

- *Let $n \geq 4$*
- *Then:*

$$\exists r \leq \lceil log_2^5(n) \rceil \text{ such that } d = ord(n)_{\mathbb{Z}_n^*} > log_2^2(n)$$

Proof. The proof is by *contradiction*:

- Let $n \geq 4 \Rightarrow \lceil log_2^5(n) \rceil \geq 32$.
- Let V be $\lceil log_2^5(n) \rceil$.
- Let

$$\Pi = n^{\lfloor log_2(V) \rfloor} \cdot \prod_{i=1}^{\lceil log_2^2(n) \rceil} (n^i - 1)$$

  .

- Let v be $\{s \in \{\ldots, v\} : s \nmid \Pi\}$

- Assume by contradiction that$v = 0$:
  - Then, by definition: $\forall s \in v : s \nmid \Pi$.
  - Consider that $lcm\{1,\ldots,V\}|\Pi$
  - Consider that:

$$\Pi \leq n^{\lfloor log_2(V)\rfloor} \cdot \prod_{i=1}^{\lceil log_2^2(n)\rceil} n^i = \tag{3.1}$$

$$n^{\lfloor log_2(V)\rfloor + \sum_{i=1}^{\lfloor log_2^2(n)\rfloor} i} = \tag{3.2}$$

$$n^{\lfloor log_2(V)\rfloor + \frac{1}{2}\lfloor log_2^2(n)\rfloor\cdot(\lfloor log_2^2(n)\rfloor+1)} < \lfloor (log_2(n))^4 \rfloor \tag{3.3}$$

$$= 2^{log_2(n)\cdot\lfloor (log_2(n))^4\rfloor} \tag{3.4}$$

$$< 2^{log_2^5(n)} \tag{3.5}$$

$$< 2^V \tag{3.6}$$

  - So, $lcm\{1,\ldots,V\}|\Pi \implies lcm\{1,\ldots,V\} \leq \Pi < 2^V$
  - Since $V \geq 32$ and $lcm\{1,\ldots,V\} \geq 2^V$ due to Lemma6, we have a **contradiction**.
  - Therefore, $v \neq 0$
- Let then r be $\min(v) \geq 2$
- Assume that q is a prime and $q|r$
  - Consider also that $r|V$, since $r \leq V \implies q^\alpha|V \implies \alpha \leq \lfloor log_2(V)\rfloor$
- Assume also that every $q|r$, also $q|n$.
- Then, $r = \prod_{q|r} q^\alpha | \prod_{q|r} q^{\lfloor log_2(V)\rfloor} | \prod_{q|n} q^{\lfloor log_2(V)\rfloor}$, where p is a prime number.
- Let n be $\prod_{q|n} q^\beta$, where $\beta \geq 1$
- Then, we have $n^{\lfloor log_2(V)\rfloor} = \prod_{q|n} q^{\beta\cdot\lfloor log_2(V)\rfloor}$
- Before, we proved:
$$r|\prod_{q|n} q^{\lfloor log_2(V)\rfloor}|n^{\lfloor log_2(V)\rfloor}|\Pi$$
- Therefore, $r|\Pi$, but $r \in v$, so $r \nmid \Pi$, that is a **Contradiction**.
- Then not every prime divisor of n is a prime divisor of r.

- Consider that $\frac{r}{(r,n)} \in v \implies \frac{r}{(r,n)} \leq r = \min(v) \implies \frac{r}{(r,n)} = r \implies \frac{r}{(r,n)} = 1$
  - By *contradiction*:
  - Assume that $\frac{r}{(r,n)} \not\in v$
  - Then, $\frac{r}{(r,n)} | \Pi$
  - Let $r = \prod_{p|r} p^{\alpha}$
  - If $p|r \wedge p \nmid n \implies p|\frac{r}{(r,n)} \implies p^{\alpha}|\frac{r}{(r,n)}$
  - But, if $\frac{r}{(r,n)}|\Pi \implies p^{\alpha}|\prod_{i=1}^{\cdots}(n^i - 1)|\Pi \implies r|\Pi$, that is a **contradiction**. Therefore, $\frac{r}{(r,n)} \in v$
- So, $\text{ord}(n)_{\mathbb{Z}_r^*} > \lfloor log_2^2(n) \rfloor$
  - By *contradiction*:
  - $\exists i \leq \lfloor log_2^2(n) \rfloor$ such that $n^i \equiv_r = 1$
  - $\implies r|\prod_{i=1}^{\lfloor log_2^2(n) \rfloor}(n^i - 1)|\Pi$, but this is a **contradiction**.

□

3.2.2 Agrawal - Kayal - Saxema Theorem

Theorem 11 (Agrawal - Kayal - Saxema Theorem)**.** *Let* $n \geq 4 \in \mathbb{N}$, *and let* $0 < r < n$ *such that* $(n,r) = 1$ *and* $order(n) > (log_2(n))^2$. *Then:*

$$n \textit{ is prime} \iff \begin{cases} n \textit{ is not a perfect power} \\ \nexists p \leq r \\ (x+b)^n \equiv_{(n,x^r-1)} x^n + b \textit{ for every} b \in \mathbb{N} \textit{ s.t. } 1 \leq b \leq \sqrt{n} \cdot log_2(n) \end{cases}$$

3.2.3 AKS algorithm

Correctness

Theorem 12 (Correctness of the AKS algorithm)**.** *The AKS algorithm for the primality test of a given number n is correct.*
n is prime $\iff$ *the algorithm returns* $TRUE$.

Proof. The proof examines the execution case by case.

- Let's assume that n is prime:
  - Then, the algorithm cannot stop at step 3 or at step 8.

Algorithm 5: AKS primality test pseudocode

```
Data: $n \in \mathbb{N}$
Result: $TRUE$ if $n$ is prime
if $n = \alpha^{\beta}$, where $\alpha, \beta > 1 \in \mathbb{N}$ then
    return $FALSE$
end
$r \leftarrow \arg\min_x (x, n) = 1$;
$d \leftarrow ord(n)_{\mathbb{Z}_r^*} > \lceil log_2^2(n) \rceil$;
if $\exists b \leq r : 1 < (b, n) < n$ then
    return $FALSE$
end
if $n \leq r$ then
    return $TRUE$
end
if $\exists b \in \mathbb{N} : 1 \leq b \leq \sqrt{r} \cdot log_2(n) \wedge (x+b)^n \not\equiv_{(x^r - 1, n)} x^n + b$ then
    return $FALSE$
end
return $TRUE$;
```

  - By Lemma 5, $(x+b)^n \equiv_n x^n + b \forall b \in \mathbb{Z} \therefore$ the algorithm cannot terminate at step 14.
  - Therefore, the algorithm can only terminate at step 11 or 15, so: n is prime $\implies$ the algorithm returns $TRU£$.

- Let's assume that the algorithm returns $TRUE$:
  - Then, the algorithm has terminated at step 11 or 15.
  - If the algorithm has terminated at step 11, then $n \leq r$. Since we checked at 8 that (b, n) is trivial $\forall b \leq r$, then n has no trivial divisors, hence it's prime.
  - If the algorithm has terminated at step 15, let's consider that at 3 and at 8 we verified that condition 1 and 3 of the Theorem 11 hold, respectively.
  - Then, it's verified that: the algorithm returns $TRUE \Rightarrow n$ is prime.
- Therefore, is verified that n is prime $\iff$ the algorithm returns $TRUE$.

□

Complexity

Theorem 13 (Complexity of the AKS algorithm)**.** *The AKS algorithm sustains the following costs for each step:*

- *Step 3 (checking if $n = a^b$) costs $O((log(n))^{3+\epsilon})$ b.o..*
- *Step 5 (picking r) has cost of $O((log(n))^{7+\epsilon})$ b.o., in the worst case.*
- *Step 8 (computing (b, n) multiple times) has cost of $O((log(n))^{7+\epsilon})$ b.o., in the worst case.*
- *Step 14 (verifying that $(x+b)^n \not\equiv_n x^n + b$) has cost of $O(log(n) \cdot log(r))$ b.o.*
- *The total cost of the AKS algorithm is then $O(r^{5/2} log^4(n))$ b.o.*

Chapter 4

Factoring Problem

4.1 Factoring Problem

This section explores the state of the art on the factoring problem, that is at the base of the attacks of numerous crypto algorithms.

4.1.1 Eratostene's sieve

In ancient times, a Greek mathematician named Eratostene came up with an intuitive factoring method, based on the extraction of the prime numbers up to a given N. Let's consider the cost of this algorithm:

- For each $p \le N^{1/2}$, one squaring operation is executed, and

  $$l_p = \lfloor \frac{N}{p} \rfloor - p \le \frac{N}{p}$$

  sums are executed.

- Since $p^2 + kp \le N$ for $k \le l_p$, then:

  $$\begin{aligned} &\sum_{p \le N^{1/2}} (\log^2(p) + \sum_{k=1}^{l_p} \log(p^2 + kp)) \le \\ &\sum_{p \le N^{1/2}} (\log^2(p) + \log(N) \sum_{p \le N^{1/2}} \frac{N}{p}) \\ &\qquad = N \log(N) \sum_{p \le N^{1/2}} \frac{1}{p} + O(N^{1/2} \log(N)) \\ &\qquad = O(N \log(N) \log(\log(N))) \end{aligned}$$

Algorithm 6: Eratostene's sieve

```
Data: $N \in \mathbb{N}$
Result: $c$, the list of booleans that represents the prime integers up to $N$
1  $c[N] \leftarrow \{True * N\}$;
2  $p \leftarrow 2$; while $p^2 \leq N$ do
3      $n \leftarrow p^2$;
4      while $n \leq N$ do
5          $c[N] \leftarrow False$;
6          $n \leftarrow p + n$;
7      end
8      repeat
9          $p \leftarrow p + 1$;
10     until $c[p] = True$;
11 end
12 return $c$;
```

4.1.2 Trial division method

The simplest algorithm to factorize a given number is to proceed by attempts.
This algorithm tries to do it in the most efficient way possible, but it is still less efficient than the methods that will be proposed later. If the list of prime numbers up to

Algorithm 7: Trial-division method

```
Data: $N \in \mathbb{N}$, $L$ the list of prime numbers up to $\sqrt{N}$
Result: $c$, the list of booleans that represents the prime integers up to $N$
1  $c \leftarrow$ List($empty$);
2  for $m \in L$ do
3      if $m|N$ then
4          $c \leftarrow$append($c, m$);
           /*Appends the element $m$ to the list $c$                */
5      end
6  end
7  return $c$;
```

$\sqrt{N}$ is provided in input, it is necessary to compute, in the worst case, $\sqrt{N}$ reminders, each one at the cost of $\log^2(N)$.
Therefore, the cost of this method is $O(\sqrt{N}\log^2(N))$.

4.1.3 Fermat factoring method

This method aims to factorize a number N in two numbers p, q, such that $N = p \cdot q$. Also, there must be some x, y such that:

$$N = x^2 - y^2 = (x+y)(x-y)$$

If N is odd it can be shown that $y = \frac{N-1}{2}, x = \frac{N+1}{2}$. Let's now consider the cost of this

Algorithm 8: Fermat's factoring method

```
  Data: $N \in \mathbb{N}$
  Result: $p, q$
1 $y \leftarrow 1$;
2 repeat
3     $x \leftarrow N + y^2$;
4     if $(\lfloor\sqrt{N+y^2}\rfloor)^2 = N + y^2$ then
5         return $x, y$
6     end
      /*Checks if $N + y^2$ is a perfect square    */
7     $y \leftarrow y + 1$;
8 until $y = \frac{N-1}{2}$;
```

algorithm:

- Each iteration has a cost of

$$O(\log^2(y) + \log^2(N+y^2) + \log^3(N+y^2)) = \\ O(\log^3(N+y^2))$$

- The loop is repeated $O(\log^A(N))$ times.
- Therefore, the complexity of this algorithm is $O(\log^{A+3}(N))$ b.o..

4.1.4 Pollard's rho-method

The Pollard's ρ-method tries to factorize a number N, by attempting to find a collision when applying multiple times the same function. That can be summarized as follows:

- Let $F : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$.
- Let $x_0 \in \mathbb{Z}_N^*$ be a randomly chosen seed.

- Let $F^{(i)}(x_0) = F \circ F \circ \cdots \circ F(x_0)$.
- We want to find i, j such that $F^{(i)}(x_0) \equiv_N F^{(j)}(x_0)$, and compute $(N, F^{(i)}(x_0) - F^{(i)}(x_0))$

Let's now investigate why this algorithm is correct:

- Assume that p is prime and that $p|N$.
- Build a sequence of T numbers: $\{F_{x_0}\}_{k \le T \in \mathbb{N}}$
- Assume that exists a collision, so:

$$\begin{aligned} &\exists F^{(i)}(x_0) \equiv_p F^{(j)}(x_0) \\ \iff &\ p|F^{(i)}(x_0) - F^{(j)}(x_0) \\ \iff &\ (p, F^{(i)}(x_0) - F^{(j)}(x_0)) > 1 \end{aligned}$$

- Due to the Birthday Paradox, we know that:

$$\mathbb{P}[\exists F^{(i)}(x_0) \equiv_p F^{(j)}(x_0)] = \frac{1}{2} \text{ when } T \le \sqrt{p}$$
$$\text{Since } p|N \implies p \le \sqrt{N} \implies T = O(\sqrt[4]{N})$$

- Therefore, probably we will find the collision.

The cost of the algorithm is easy to compute:

- We have approximately $O(T^2) = O(\sqrt{N})$ steps, that is the quantity necessary to make the Birthday Paraodx hypothesis hold;
- Each one has a cost of $O(log^2(N))$ b.o.
- The expected cost is then O($\sqrt{(}N) log^2(N)$) b.o.

4.1.5 Pomerance's quadratic sieve

Algorithm 9: Pollard's ρ-method

```
   Data: $N, T \in \mathbb{N}, F : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$
   Result: List of $d$, non-trivial factors of $N$
1  $x_0$ is randomly chosen in $\mathbb{Z}_N^*$;
2  $m \leftarrow 1$;
3  $y_1 \leftarrow F(x_0)$;
4  $y_2 \leftarrow F(y_1)$;
5  while $m \leq T$ do
6      $d \leftarrow (N, y_1 - y_2)$;
7      if $d > 1 \wedge d < N$ then
8          return $d$
9      end
10     $m \leftarrow \mathrm{m} + 1$;
11     $y_1 \leftarrow F(y_1)$;
12     $y_2 \leftarrow F(F(y_2))$;
13 end
```

Chapter 5

Attacks

5.1 Birthday Paradox

This Paradox is not an actual one, but is the result of a cognitive bias.
The idea is that in a room full of people, the probability of having no one with matching birthdays is actually lower than expected.
Consider what follows:

- Let n be the number of possible birthdays
- Let N be the number of people.
- Let S be a space $\{1, \dots, n\}^N \ni (c_1, \dots, c_N)$.
- The probability of a given combination of birthdays to happen is then:

$$\mathbb{P}[(c_1, \dots, c_N)] = n^{-N} = \frac{1}{|S|}$$

- Let's now compute the probability of having **no collisions**:

$$\begin{aligned} p(N) &= \mathbb{P}[\forall i \neq j : c_i \neq c_j] \\ &= \frac{\prod_{i=0}^{N-1} n - i}{n^N} \\ &= \frac{n \cdot (n-1) \cdot \dots \cdot (n-N+1)}{n \cdot n \cdot \dots \cdot n} \\ &= \prod_{i=0}^{N-1} 1 - \frac{i}{n} \end{aligned}$$

- Let'now compute the probability of having **at least one collision**:

$$\begin{aligned} q(N) &= 1 - p(N) \geq 1 - \epsilon \\ &\iff p(N) \leq \epsilon \end{aligned}$$

- Let's now consider that:

$$\begin{aligned} p(N) = \prod_{i=0}^{N-1} 1 - \frac{i}{n} &\leq \prod_{i=0}^{N-1} e^{-\frac{i}{n}} \\ &= e^{\sum_{i=0}^{N-1} -\frac{i}{n}} \\ &= e^{-\frac{1}{n}\sum_{i=0}^{N-1} i} \\ &= e^{-\frac{(N)(N-1)}{2n}} \leq \epsilon \end{aligned}$$

  Therefore,

$$N \geq \frac{1 + \sqrt{1 - 8n\log(\epsilon)}}{2} \simeq \sqrt{n|\log(\epsilon)|}$$

- So, when $N \simeq n$, then $\epsilon \simeq \frac{1}{2}$

Useful Facts

- The GMP library is a free library for arbitrary precision arithmetic. It implements all the basic arithmetic operations with the maximum efficency possible.

List of Algorithms

List of Theorems

List of Lemmas