

Cryptography: notes

Niccolò Simonato

October 20, 2022

Contents

1	Efficient implementations of elementary operations	2
1.1	Notation	2
1.2	Classification of the algorithms' complexity	2
1.3	Basic bit operations	3
1.3.1	Sum of 3 bits - 3-bit-sum	3
1.3.2	Summation of 2 numbers	3
1.3.3	Summation of n numbers	3
1.3.4	Product of 2 numbers	3
1.3.5	Division of 2 numbers	4
1.3.6	Production of n numbers	4
1.4	Optimizations of more complex operations	5
1.4.1	Powers & Modular Powers	5
1.4.2	Finding the b -expansion of n (n_b)	7
1.4.3	How to use Bezout formula to compute modular inverses	8
1.4.4	Computing the order of an element in a cyclic group	8
1.4.5	Extended Euclidean Algorithm	8
1.4.6	Computation of square and m -th root of n	8
1.4.7	Compute n, m given n^m	10
2	Elements of Number Theory	11
2.1	Elements of Number Theory	11
2.1.1	The cyclic group \mathbb{Z}_n^*	11
2.2	Reminders of Modular Arithmetic	11
2.2.1	Little Fermat's Theorem	11
2.2.2	Euler-Fermat's Theorem	11
2.2.3	Bézout's identity	11
2.2.4	Chinese Reminder's Theorem	12
2.2.5	Bijection of a modular function Lemma	13
2.2.6	Euler's φ function Lemmas	13
2.2.7	Multiplicativity of the Euler's φ -function	14

Chapter 1

Efficient implementations of elementary operations

1.1 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 .

1.2 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 $c > 0$.

¹ $d_{k-1} \neq 0$

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

1.3 Basic bit operations

1.3.1 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 $O(1)$.

1.3.2 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 $\max\{\text{length}(n_1), \text{length}(n_2)\}$ times.
 Each time the carry on of the previous sum is added to the two digits.
 This operation has then complexity $O(\max\{\text{length}(n_1), \text{length}(n_2)\}) = O(\max\{\log(n_1), \log(n_2)\})$

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

1.3.4 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 case², that becomes $O(\log^2(m))$.

²two numbers that are equally large

1.3.5 Division of 2 numbers

Let's consider the division of two numbers m, n . This operations 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 $0 \leq r < n$ is reached.

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

1.3.6 Production of n numbers

Let's assume that $j \in [1, s+1]$ 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: $T(m_1 \cdot m_2) = O(\log(m_1) \cdot \log(m_2)) = O(k_1 \cdot k_2) \leq c \cdot k_M^2$.
- (2) Base case: $T(m_1 \cdot m_2 \cdot m_3) = T(m_1 \cdot m_2) + T((m_1 \cdot m_2) \cdot m_3)$
 $\leq c \cdot k_M^2 + c \cdot k_{m_1 \cdot m_2} + k_{m_3}$
 $\leq c \cdot k_M^2 + c \cdot k_{M^2} + k_M$
- Inductive step: we assume the inductive hypothesis to be true up to s . Then,
 $T(\prod_{j=1}^{s+1} m_j) = T([\prod_{j=1}^s m_j] \cdot m_{s+1})$
 $\leq c \cdot \sum_{j=1}^s (j \cdot k_M^2)$
 $= c \cdot k_M^2 \cdot \frac{s \cdot (s-1)}{2}$
 $= O(k_M^2 \cdot s^2)$
 $= O(s^2 \cdot \log^2(M))$

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

1.4 Optimizations of more complex operations

1.4.1 Powers & Modular Powers

Let's consider what follows: $a^n = a \cdot a \cdot a \cdots a$, where a is repeated n times.

Trivial implementation

The most trivial implementation would consist 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 $n = (b_{k-1}, b_{k-2}, \dots, b_0) = \sum_{i=0}^{k-1} b_i \cdot 2^i$.

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  $A \leftarrow a \bmod n \implies A \leftarrow a;$ 
2  $P \leftarrow P \cdot A \bmod n \implies P \leftarrow P \cdot A;$ 
3  $A \leftarrow A^2 \bmod n \implies A \leftarrow A^2;$ 

```

Square & Multiply method for polynomials

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

- Assume that $f, g \in \mathfrak{R}$.
- 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, $T(h(x)) = O(\sum_{j=0}^{\log^2(n)} T(h_j)) = O(r^2 \cdot \log^2(n))$.

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: $h_{r+i} \cdot x^{r+i} \equiv_{x^r-1} h_{r-i} \cdot x^{r-i}$ 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 $T(\#Loops \cdot (h(x) \bmod (n, x^r - 1))) = O(\log(n) \cdot r^2 \cdot \log(n)) = O(r^2 \cdot \log^3(n))$.

1.4.2 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 $r_i \in \{0, 1, \dots, b-1\}$.
- Let $n_b = (r_{k+1}, r_k, \dots, r_1, r_0)$.
- Then:
 - $n = q_0 \cdot b + r_0$
 - $q_0 = q_1 \cdot b + r_1$
 - ...
 - $q_k = 0 \cdot b + q_k$
- Consider then that $q_k = r_{k+1}$
- And that $b^{k+2} > n > b^{k+1} \rightarrow (k+2) \cdot \log(b) \leq \log(n) \leq (k+1) \cdot \log(b)$.
- $\therefore k = O(\frac{\log(n)}{\log(b)})$.

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

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

1.4.3 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 $ax \equiv_m 1$;
- $\therefore x \bmod m$ is the modular inverse of a in $\mathbb{Z}m^*$ for its definition.

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

1.4.4 Computing the order of an element in a cyclic group

The order of an element a in $\mathbb{Z}p^*$ ($m = \text{order}(a)$) is the minimum m such that $a^m \equiv_p 1$.

This problem is computationally hard, because the most efficient way to compute $\text{order}(a)$ is to brute force its value.

The only optimization available is that we don't have to compute the modular 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 $O(\text{order}(a) \cdot \log^2(p))$, because we have to compute $a^i \bmod p$ for each attempt to find $\text{order}(a)$.

1.4.5 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 $O(\log^3(\max\{a, b\}))$.

1.4.6 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:

Algorithm 2: The Extended Euclidean Algorithm

Data: a, b
Result: $(\lambda, \mu, GCD(a, b))$

```
1  $old\_r \leftarrow a$ ;  
2  $r \leftarrow b$ ;  
3  $old\_s \leftarrow 1$ ;  
4  $s \leftarrow 0$ ;  
5  $old\_t \leftarrow 0$ ;  
6  $t \leftarrow 1$ ;  
7 while  $r \neq 0$  do  
8    $quotient \leftarrow floor(old\_r/r)$ ;  
9    $old\_r \leftarrow r$ ;  
10   $old\_s \leftarrow s$ ;  
11   $old\_t \leftarrow t$ ;  
12   $r \leftarrow old\_r - quotient \cdot r$ ;  
13   $s \leftarrow old\_s - quotient \cdot s$ ;  
14   $t \leftarrow old\_t - quotient \cdot t$ ;  
15 end  
16 return  $(s, t, old\_r)$ 
```

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  $x_{l-1}$ 
```

- 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 $O(\log^3(n))$.

1.4.7 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 $\lfloor \sqrt[m]{m_i} n^m \rfloor$;
- 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 correspondent integral, to $O(\log \log(n))$.
- The final cost is therefore $O(\log^3(n) \cdot (\log \log(n))^2)$
 $= O(\log^{3+\epsilon}(n))$, with $\epsilon \in (0, 1)$

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

Chapter 2

Elements of Number Theory

2.1 Elements of Number Theory

2.1.1 The cyclic group \mathbb{Z}_n^*

The cyclic group \mathbb{Z}_n^* is defined as $\{a \in \mathbb{Z}_n : (a, n) = 1\}$.

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 $\langle g \rangle$.

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.

2.2 Reminders of Modular Arithmetic

2.2.1 Little Fermat's Theorem

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

2.2.2 Euler-Fermat's Theorem

- Let $n \in \mathbb{Z}$ be a number.
- Then, $a^{\varphi(n)} \equiv_n 1$.

2.2.3 Bézout's identity

- 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 .

2.2.4 Chinese Remainder's Theorem

Theorem 1 (Chinese Remainder's Theorem). *Given a system of congruences:*

- $x \equiv_{m_1} a_1$
- $x \equiv_{m_2} a_2$
- ...
- $x \equiv_{m_r} a_r$

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 $M = m_1 m_2 \dots m_r$.

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 $M_i = M/m_i$, to be the product of all of the moduli except for the i -th.
- Then $\text{GCD}(m_i, M_i) = 1$ 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: $x \equiv_{m_i} a_i M_i N_i \equiv_{m_i} a_i$,

□

2.2.5 Bijection of a modular function Lemma

Lemma 1 (Bijection of a modular function Lemma). *Consider $a \in \mathbb{Z}_n^*$ and $f : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$. 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 $\bar{x} = a^{-1}b \in \mathbb{Z}_n$.
- Then, $f(\bar{x}) \equiv_m a \cdot \bar{x}$
 $\equiv_m a \cdot a^{-1} \cdot b \equiv_m b$.
- Therefore, f is surjective.

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

2.2.6 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. • 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, $n = |B| = \sum_{\frac{n}{d}} \varphi(d)$.
- Consider $(a_1, d_1) = 1 \wedge (a_2, d_2) = 1$
, 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, $d_1 | a_1 d_2 \Rightarrow d_1 | d_2 \wedge d_2 | a_2 d_1 \Rightarrow d_2 | d_1$.
- $\therefore d_1 = d_2$. 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: $\alpha = 1$

Then, $p = \sum_{\substack{d \\ p}} \varphi(d) = \varphi(1) + \varphi(p) = 1 + \varphi(p)$
 $\Rightarrow \varphi(p) = p - 1$.

Case base: $\alpha = 2$

Then, $p^2 = \sum_{\substack{d \\ p^2}} \varphi(d) = \varphi(1) + \varphi(p) + \varphi(p^2)$
 $= 1 + p - 1 + \varphi(p^2) \Rightarrow \varphi(p^2) = p^2 - p - 1 + 1$
 $= p \cdot (p - 1) = p^{\alpha-1}(p - 1)$

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 .

$p^\alpha = \sum_{\substack{d \\ p^\alpha}} \varphi(d)$
 $= \sum_{i=0}^{\alpha-1} \varphi(p^i) + \varphi(p^\alpha)$
 $\Rightarrow \varphi(p^\alpha) = p^\alpha - p^{\alpha-1} = p^{\alpha-1}(p - 1)$.

□

2.2.7 Multiplicativity of the Euler's φ -function

Theorem 2 (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, =

Proof.

□

Useful Facts

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

List of Algorithms

1	The Square & Multiply Method	5
2	The Extended Euclidean Algorithm	9
3	The Efficient m-th root of n	9

List of Theorems

1	Theorem (Chinese Remainder's Theorem)	12
2	Theorem (Multiplicativity of the Euler's φ -function)	14

List of Lemmas

1	Lemma (Bijection of a modular function Lemma)	13
2	Lemma (Sum of the prime divisors' φ -function)	13
3	Lemma (Number of divisors of a prime number's power)	14