Cryptography: notes

Niccolò Simonato

October 12, 2022

Contents

Chapter 1

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

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

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

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 case[2], that becomes $O(log^2(m))$.

[2] two 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 $T(\prod_{j=1}^{s+1} m_j \text{ mod } n) = O(s \cdot log^2(M))$
- 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: $a^n = a \cdot a \cdot a \cdot \dots \cdot a$, 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 $n = (b_{k-1}, b_{k-2}, \dots, b_0) = \sum_{i=0}^{k-1} b_i \cdot 2^i$.
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:

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

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: $h_{r+i} \cdot x^{r+i} \equiv_{x^r - 1} h_{r-i} \cdot x^{r-i}$ 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 representation 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:
  - $n = q_0 \cdot b + r_0$
  - $q_0 = q_1 \cdot b + r_1$
  - $\ldots$
  - $q_k = 0 \cdot b + q_k$
- 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 Modular inverses

1.5 Reminders of Modular Arithmetic

1.5.1 Little Fermat's Theorem

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

1.5.2 Euler-Fermat's Theorem

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

1.6 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