M1 Applied Mathematics: Cryptology complement

Clément Pernet

January 30, 2024

Contents

Chapter 1

Introduction

This class which complements the *Introduction to Cryptology* class, is meant to give to students of the M1 Applied Mathematics, complementary notions on the mathematical fundations of cryptology, and more generally on computer algebra.

The content lies somewhere in-between an *Applied Algebra* course and a *Computational Algorithm and complexity* course. Indeed we will focus on a constructive description of the algebra notions, by connecting them systematically to algorithms with a special care on their cost analysis.

1.1 Groups, rings, and fields

We recall informally the definition of the elementary algebraic structures which will be used throughtout the course. For the sake of simplicity, we will only deal with commutative structures.

Definition 1.1.1

A commutative group (or Abelian group) $(G, *, 1)$ is a set G containing an element 1, with a law $* : G \times G \to G$ satisfying the following conditions: $\forall x, y, z \in G$

$$x * 1 = 1 * x = x \qquad 1 \textit{ is the neutral element for } * \tag{1.1}$$

$$(x * y) * z = x * (y * z) \qquad * \textit{ is associative} \tag{1.2}$$

$$x * y = y * x \qquad * \textit{ is commutative} \tag{1.3}$$

$$\exists t \in G, x * t = 1 \qquad \textit{inversibility of } * \tag{1.4}$$

Example 1.1.1

1. $(\mathbb{Z}, +, 0)$
2. $(\mathbb{Q} \setminus \{0\}, \times, 1)$

Definition 1.1.2

A commutative ring $(R, +, \times, 0, 1)$ is an Abelian group $(R, +, 0)$ equipped with a second law

$\times : R \to R \times R$ such that $\forall x, y, z$

$$x \times 1 = 1 \times x = x \quad \text{1 is the neutral element for } \times \tag{1.5}$$
$$x \times 0 = 0 \times x = 0 \quad \text{0 is absorbant} \tag{1.6}$$
$$x \times (y \times z) = (x \times y) \times z \quad \times \text{ is associative} \tag{1.7}$$
$$x \times y = y \times x \quad \times \text{ is commutative} \tag{1.8}$$
$$(x + y) \times z = x \times z + y \times z \quad \times \text{ is distibutive over } + \tag{1.9}$$

Note every element of a ring may not have inverses with respect to the law ×.

Example 1.1.2

1. $(\mathbb{Z}, +, \times, 0, 1)$
2. $(\mathbb{Z}/n\mathbb{Z}, +, \times, 0, 1)$
3. $(\mathbb{Z}[X], +, \times, 0, 1)$
4. $(\mathbb{Q}, +, \times, 0, 1)$

Definition 1.1.3

A commutative field is a ring $(F, \times, +, 0, 1)$ where $(F \setminus \{0\}, \times, 1)$ is a group.

Example 1.1.3

1. *$(\mathbb{Z}, +, \times, 0, 1)$ is not a field, as only 1 and -1 have inverse for $\times$*
2. *$(\mathbb{Z}/4\mathbb{Z}, +, \times, 0, 1)$ is not a field as 2 has no inverse for $\times$*
3. *$(\mathbb{Z}/5\mathbb{Z}, +, \times, 0, 1)$ is a field as $1 \times 1 = 1, 2 \times 3 = 1$ and $4 \times 4 = 1$*
4. *$(\mathbb{Q}, +, \times, 0, 1)$ is a field*

Note that we intentionally avoid considering non-commutative algebraic structures since we will only focus on commutative ones in the course. In particular, we will almost always deal with finite sets (groups, rings, and fields), namely sets having a finite number of elements and in this setting, Wedderbrun's Theorem (which proof is beyond the scope of this course) states that every finite field is commutative.

Chapter 2

Three fundamental algorithms

Pretty much every important result presented in this course will be connected to one of three fundamental algorithms which we will present in this chapter.

For the sake of simplicity we will present them in their simplest setting, namely working over integers, modular integers or polynomials over an arbitrary ring.

Before defining more formally the algebraic structures in the next chapters, we will recall informally that the sets of integers, modular integers and polynomials have a ring structure, which means that they can be equipped with two laws, an addition and a multiplications with their respective neutral elements 0 and 1, with the usual properties of commutativity, associativity, distributivity, and the property that every element x has an opposite $-x$, i.e. an inverse with respect to the addition law.

We recall that $\mathbb{Z}$ denotes the ring the integers.

2.1 The extended Euclidean Algorithm

2.1.1 Euclidean division

Theorem 2.1.1

For every $a, b \in \mathbb{Z}$, there is a unique pair $q, r \in \mathbb{Z}$ with $0 \leq r < |b|$ such that $a = bq + r$.

Proof. We start with the existence: first suppose $a, b \geq 0$ and consider Algorithm 1. This algorithm always terminates in a finite number of iterations, as the sequence of the r_i's a is a strictly decreasing sequence of non negative integers. The following invariant remains true

Algorithm 1 Euclidean division

```
Input: $a, b \in \mathbb{Z}_+$
Output: $q, r \in \mathbb{Z}$ such that $0 \leq r < b$ and $a = bq + r$
  $q_1 \leftarrow 0$
  $r_1 \leftarrow a$
  $i \leftarrow 1$
  while $r_i \geq b$ do
      $r_{i+1} \leftarrow r_i - b$
      $q_{i+1} \leftarrow q_i + 1$
      $i \leftarrow i + 1$
  $(q, r) \leftarrow (q_i, r_i)$
```

at each iteration of the algorithm: $a = bq_i + r_i$, hence at the last instruction, $a = bq + r$ and $0 \leq r < b$. In the case where $b < 0, a \geq 0$, setting $b' = -b$ and $q' = -q$ yields to $a = b'q' + r$ with $a, b' \geq 0$ which reduces to the first case. For $a < 0, b > 0$, setting $a' = -a, q' = -q - 1$ and $r' = b - r$ yields to $a' = q'b + r'$ which reduces to the first case. For $a, b < 0$, setting $a' = -a$, $q' = q - 1, b' = -b$ and $r' = -b - r$ yields to $a' = b'q' + r'$ which reduces to the first case.

For the uniqueness, suppose $a = bq + r$ and $a = bq' + r'$ both satisfy the conditions. Then $b(q - q') = r' - r$, hence b divides $r' - r$, but since $0 \leq |r' - r| < b$, necessarily $r = r'$ and consequently $q' - q = (r' - r)/b = 0$. □

The allows us to define the modulo operator " mod " and modular reduction operation.

Definition 2.1.1

The second output r of the euclidean division of an integer a by an integer b, is called the residue of a modulo b, denoted by $r = a \mod b$.

We can now define the ring $\mathbb{Z}/n\mathbb{Z}$ as the subset of integers $[\![0..n-1]\!]$. The additions and multiplication over $\mathbb{Z}/n\mathbb{Z}$ are defined as the addition and multiplication over $\mathbb{Z}$ followed by a reduction modulo n.

Example 2.1.1

1. *In the ring $\mathbb{Z}/7\mathbb{Z}, 5 + 4 = 2$; $3 - 5 = 5$ and $6 \times 5 = 2$*
2. *The hours in a day can be represented by $\mathbb{Z}/24\mathbb{Z}$.*

2.1.2 The Euclidean algorithm

Definition 2.1.2

The greatest common divisor of two integers a and b, denoted by $\gcd(a, b)$, is the largest positive integer g dividing both a and b.

Proposition 2.1.1

$$\begin{aligned}
\gcd(a, b) &= \gcd(-a, b) && (2.1)\\
\gcd(a, b) &= \gcd(a, -b) && (2.2)\\
\gcd(a, b) &= \gcd(b, a) && (2.3)\\
\gcd(a, b) &= \gcd(a, a - b) && (2.4)\\
\gcd(a, b) &= \gcd((a \mod b), b) && (2.5)
\end{aligned}$$

Proof. Left as an exercise. □

The properties satisfied by the greatest common denominator, presented in Proposition 2.1.1, allow one to iteratively reduce the arguments until ultimately reaching 0, at which point the gcd is trivial, since $\gcd(x, 0) = x$ for all $x \in \mathbb{Z}$.

Example 2.1.2

$$\gcd(16, -12) \overset{(2.2)}{=} \gcd(16, 12) \overset{(2.5)}{=} \gcd(4, 12) \overset{(2.4)}{=} \gcd(12, 4) \overset{(2.5)}{=} \gcd(0, 4) = 4.$$

This process is formalized in the form of the Euclidean Algorithm in Algorithm 2.

Algorithm 2 Euclidean Algorithm

```
Input: $a, b \in \mathbb{Z}$
Output: $g \in \mathbb{Z}_{\geq 0}$ such that $g = \gcd(a, b)$
  if $a < 0$ then
      $a \leftarrow -a$
  if $b < 0$ then
      $b \leftarrow -b$
  $r_0 \leftarrow a$
  $r_1 \leftarrow b$
  $i \leftarrow 1$
  while $r_i > 0$ do
      $r_{i+1} \leftarrow r_{i-1} \mod r_i$
      $i \leftarrow i + 1$
  $g \leftarrow r_{i-1}$
```

2.1.3 The extended Euclidean algorithm

Theorem 2.1.2 (*Bézout*)

> *For every $a, b \in \mathbb{Z}$, there is a pair of integers u, v such that $ua + vb = g$, where g is the greatest common divisor (GCD) of a and b.*

Proof. The existence of u, v is proven constructively by a slight generalization of the Euclidean Algorithm in the so-called Extended Euclidean Algorithm (Algorithm 3). Indeed, by induction, one proves that the invariant $u_i a + v_i b = r_i$ is maintained in each iteration of the algorithm:

$$r_{i+1} = r_{i-1} - r_i q_{i+1} = au_{i-1} + bv_{i-1} - (au_i + bv_i)q_{i+1} = au_{i+1} + bu_{i+1}$$

This shows that upon exit, $ua + vb = g$. □

Algorithm 3 Extended Euclidean Algorithm

```
Input: $a, b \in \mathbb{Z}$
Output: $(g, u, v) \in \mathbb{Z}_{\geq 0} \times \mathbb{Z} \times \mathbb{Z}$ such that $g = \gcd(a, b)$ and $g = ua + vb$.
  $r_0 \leftarrow a$; $u_0 \leftarrow 1$; $v_0 \leftarrow 0$
  $r_1 \leftarrow b$; $u_1 \leftarrow 0$; $v_0 \leftarrow 1$
  if $a < 0$ then
      $a \leftarrow -a$; $u_0 \leftarrow -u_0$
  if $b < 0$ then
      $b \leftarrow -b$; $v_0 \leftarrow -v_0$
  $i \leftarrow 1$
  while $r_i > 0$ do
      $r_{i+1}, q_{i+1} \leftarrow$ EuclideanDivision($r_{i-1}, r_i$)    ▷ such that $r_{i-1} = r_i q_{i+1} + r_{i+1}$
      $u_{i+1} \leftarrow u_{i-1} - u_i q_{i+1}$
      $v_{i+1} \leftarrow v_{i-1} - v_i q_{i+1}$
      $i \leftarrow i + 1$
  $g \leftarrow r_{i-1}$
  $u \leftarrow u_{i-1}$
  $v \leftarrow v_{i-1}$
```

Definition 2.1.3

Two integers are coprime if their gcd equals 1.

Corollary 2.1.1

Two integer a *and* b *are coprime if and only if there exist* $u, v \in \mathbb{Z}$ *such that* $ua + vb = 1$.

Proof. Theorem 2.1.2 gives one way of the equivalence. For reciprocal, suppose d divides a and b. Since $ua + vb = 1$ it has to divide 1 also, and $d = 1$. □

Corollary 2.1.2

The Bézout coefficients computed in the Extended Euclidean Algorithm are coprime.

Proof. The Consider the matrix $B_i = \begin{pmatrix} u_{i-1} & u_i \\ v_{i-1} & v_i \end{pmatrix}$. It satisfies the relation $B_{i+1} = B_i \begin{pmatrix} 0 & 1 \\ 1 & -q_{i+1} \end{pmatrix}$ and $B_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ therefore $\det(B_i) = u_{i-1}v_i - u_i v_{i-1} = (-1)^{i+1}$. Corollary 2.1.1 implies that all pairs (u_i, v_i) are coprime, and so are u and v.

□

Theorem 2.1.3 (*Gauss*)

If a *divides* bc *and* a *and* b *are coprime, then* a *divides* c.

Proof. Multiply the Bézout relation $au + vb = 1$ by c to get $auc + bvc = c$. Since a divides bc, then a divides vbc and therefore it divides the whole sum which is equal to c. □

Remark 2.1.1

When $a \in \mathbb{Z}/n\mathbb{Z}$ *is coprime with* n*, the GCD of* a *and* n *is* 1 *and the Bézout relation* $ua + vn = 1$ *can be written as* $ua = 1 \mod n$ *which shows that* a *is invertible in* $\mathbb{Z}/n\mathbb{Z}$ *and its inverse is* u*. Moreover, this also implies that this inverse can be computed by the Extended Euclidean algorithm. This will be further discussed in Chapter 3.*

2.2 The Chinese remainder algorithm

Theorem 2.2.1

Let m, n *be two coprime positive integers. For any* $x, y \in \mathbb{Z}$*, there is a unique* $z \in [\![0..mn-1]\!]$ *such that* $z = x \mod m$ *and* $z = y \mod n$.

This can be reformulated in terms of modular rings where the uniqueness result becomes the existence of a bijection, with an additional result that the ring structures are preserved.

Theorem 2.2.2

Let m, n *be two coprime positive integers. Then the rings formed by the sets* $Z/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$ *and* $Z/mn\mathbb{Z}$ *equipped with the modular additions and multiplications are isomorphic.*

Proof. Since m and n are coprime, let $p = m^{-1} \mod n$ and $q = n^{-1} \mod m$.

Consider the applications

$$\begin{array}{lccl} f: & \mathbb{Z}/mn\mathbb{Z} & \longrightarrow & \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z} \\ & z & \longmapsto & (z \mod m, z \mod n) \\ \\ g: & \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z} & \longrightarrow & \mathbb{Z}/mn\mathbb{Z} \\ & (x,y) & \longmapsto & xnp + ymq \mod mn \end{array} \tag{2.6}$$

Remark that $g(x,y) \mod n = ymq \mod n = y \mod n$ since $mq = 1 \mod n$. Similarly $g(x,y) = x \mod m$. Hence $f(g(x,y)) = (x,y)$ for all $(x,y) \in \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$, showing that f and g are bijections and $f = g^{-1}$. Since f is also linear, it is an isomorphism between the two rings. □

Corollary 2.2.1

> *Let $p_1, \ldots, p_k$ be k co-prime positive intgers. Then the rings $Z/p_1\mathbb{Z} \times \cdots \times \mathbb{Z}/p_k\mathbb{Z}$ and $Z/(p_1 \ldots p_k)\mathbb{Z}$ are isomorphic.*

Proof. By induction on k, applying Theorem 2.2.2. □

Concretely speaking, this isomorphism proposes an alternative representation for the ring $Z/(p_1 \ldots p_k)\mathbb{Z}$. This becomes even more usefull when one remarks that every ring computation over $\mathbb{Z}$ which result is in $[\![0..(p_1 \ldots p_k)]\!]$ can be embedded in $Z/(p_1 \ldots p_k)\mathbb{Z}$ and therefore benefit from this change of representation. The benefits of using this alternative representation include

1. use smaller rings, possibly of a fixed size, suited for computer arithmetic
2. spread the load of the computation into k independent tasks, therefore ready for a parallelization.
3. benefit from higher algebraic structure in each of the $\mathbb{Z}/p_i\mathbb{Z}$. Choosing p_i as prime numbers, makes each of the $Z/p_i\mathbb{Z}$ a field, and open ways for division based algorithms.

These points will be further developped in Chapter 3.

Remark 2.2.1

> *This representation Theorem is an integer analogous to the Lagrange interpolation Theorem (Theorem 2.2.3) for polynomials. Indeed, remark that the residue of a polynomial f modulo $X - x_i$ is the evalution of f in x_i: $f \mod (X - x_i) = f(x_i)$.*

Theorem 2.2.3 (*Lagrange*)

> *Over a field K, given n distinct elements $x_1, \ldots, x_n$ and n elements $y_1, \ldots, y_n$, there is a unique polynomial $f \in K[X]$ of degree $< n$ satisfying*
>
> $$f(x_i) = y_i \quad \forall i \in [\![1..n]\!]. \tag{2.7}$$

Proof. Consider the family of polynomials $L_i(X) = \prod_{j \neq i}(X - x_j)/\prod_{j \neq i}(x_i - x_j)$ and the polynomial $f(X) = \sum_{i=1}^{n} y_i L_i(X)$ of degree $\leq n-1$. Since $L_i(x_j) = \delta_{i,j}$, we have $f(x_i) = y_i$ for all $i \in [\![1..n]\!]$. Suppose $g \in K[X]$ of degree $< n$ also verifies Equation (2.7), then $f - g$ vanishes in n distinct points. Since its degree is $< n$ it must be the zero polynomial, which proves the uniqueness of f. □

The Chinese remainder Theorem, applied to the ring of univariate polynomials $K[X]$ is actually a slight generalization of Lagrange interpolation theorem, as it allows a representation not only by evaluations in distinct points (corresponding to residues modulo degree one polynomials), but also by residues modulo coprime polynomials of arbitrary degrees.

Corollary 2.2.2

> *Let K be a field, and $p_1, \dots p_k$ be k coprime polynomials over $K[X]$. Given k polynomials $a_1, \dots, a_k$ over $K[X]$, there exists a unique polynomial f of degree $< n = \sum_{i=1}^{k} \deg(p_i)$, such that $f = a_i \mod p_i$ for all $i \in [\![1..k]\!]$.*

2.3 The Square and Multiply Algorithm

Over a group, a fundamental operation is to compose the same element n times with the group law. We will present the algorithm in the setting of a multiplicative group (with a law denoted by × and the composition $\underbrace{a \times \cdots \times a}_{n \text{ times}}$ denoted by a^n), but it applies simularly for additive groups (which law is denoted by + and the composition $\underbrace{a + \cdots + a}_{n \text{ times}}$) is denoted by $n.a$).

Instead of iteratively multiplying by a an accumulator n times, the `SquareAndMultiply` algorithm relies on a divide and conquer approach based on the relations:

$$\begin{cases} a^n = (a^{\lfloor n/2 \rfloor})^2 & \text{for } n \text{ even} \\ a^n = (a^{\lfloor n/2 \rfloor})^2 \times a & \text{for } n \text{ odd,} \end{cases} \tag{2.8}$$

which reduces the number of group operation from n to $2 \log_2 n$.

Algorithm 4 Square And Multiply (recursive)

```
Input: $a \in G$ an element of a group
Input: $n \in \mathbb{N}$
Output: $r = a^n$
  $x \leftarrow$ SquareAndMultiply($a$, $\lfloor n/2 \rfloor$)
  $r \leftarrow x \times x$
  if $n$ is odd then
      $r \leftarrow r \times a$
  return $r$
```

The sequence of tests during the execution of this algorithm on an instance correspond to tests on the value of each bit of the binary representation of n. Consequently an iterative version of this algorithm follows from the following relation

$$a^n = a^{\sum_{i=0}^{\lfloor \log_2 n \rfloor} n_i 2^i} = \prod_{i=0}^{\lfloor \log_2 n \rfloor} \left(a^{2^i}\right)^{n_i}. \tag{2.9}$$

Algorithm 5 Square And Multiply (iterative)

```
Input: $a \in G$ an element of a group
Input: $n \in \mathbb{N}$
Output: $r = a^n$
  Let $(n_0, \dots, n_k) \in \{0,1\}^k$ be the binary representation of $n = \sum_{i=0}^{k} n_i 2^i$ with $k = \lfloor \log_2 n \rfloor$.
  $b \leftarrow a$
  $r \leftarrow 1$
  for $i \leftarrow 0 \dots k$ do
      if $n_i = 1$ then
          $r \leftarrow r \times b$
      $b \leftarrow b \times b$
  return $r$
```

Chapter 3

Effective arithmetic

3.1 The Ring of integers $\mathbb{Z}$

A challenge with integer arithmetic is the bitsize of the integers considered. For small enough integers, fixed size arithmetic can be used and rely on efficient hardware implementations. Otherwise, arthemtic of larger integer must be emulated by software, in the so-called *multiprecision* arithmetic.

3.1.1 Small integers

In practice, most programming languages propose integer base types, in order to represent a commonly used subset of $\mathbb{Z}$: in `C`, these are the types `short, int, long, long long`, and their unsigned variants `unsigned short`, `unsigned int`, `unsigned long`, `unsigned long long`.

To ensure portability, the bit-size of these types is not specified (or only partially), which eventually led to more troubles than advantages. It is now recommend to prefer their fixed size variants, like the one proposed in the `stdint.h` library (in `C`) or `cstdint.h` (in `C++`): `int16_t, int32_t, int64_t, uint16_t, uint32_t, uint64_t`.

In addition, the numerical base types include the floating points types: single precision `float` (32 bits) and doule precision `double` (64 bits) defined by the IEEE-754 norm. These types can represent exactly all signed integers of 24 bits (for `float`) and 53 bits (for `double`). Eventhough there capacity of representation is suboptimal (32 bits to represent a 24 bit range), the efficiency of the hardware arithmetic implemented in most of nowadays processors is in favour of the floating point arithmetic. The computational throughput when performing a large number of multiplications and additions (which is a frequent pattern) is in most cases twice as large for floating point types compared to integer types of the same size. Table 3.1 summarizes the representation range and computational throughput for the base integer types.

3.1.2 Multiprecision integers

As computer hardware can not natively support integer arithmetic of arbitrary size, software must implement it based on fixed size integer arithmetic: large integers are stored in arrays of 64 bits words, named *limbs*, usually of type `uint64_t`. For instance an integer n has bit-size $s = \lceil \log_2 n \rceil$ and is stored as an array of $\ell = \lceil \log_{2^{64}} n \rceil = \lceil s/64 \rceil$ limbs.

The multiprecision arithmetic is no longer constant time, and its cost depend on the bit-size s of the input.

Type	min	max	# add / cycle	# mul / cycle	# muladd / cycle
`int32_t`	$-2^{31}+1$	$2^{31}-1$	48	32	16
`uint32_t`	0	$2^{32}-1$	48	32	16
`float`	-2^{24}	2^{24}	32	32	32
`int64_t`	$-2^{63}+1$	$2^{63}-1$	24	16	8
`uint64_t`	0	$2^{64}-1$	24	16	8
`double`	-2^{53}	2^{53}	16	16	16

Table 3.1: Base type representing integers. The computational throughput is that on one core of an Intel Xeon Gold 6126, with AVX-512 vectorized instructions.

Addition

Multiprecision addition is done in the straighforward way: adding each corresponding limb from the lowest to the highest and propagating a carry. This algorithm has linear time, its uses:

$$T_{\texttt{Add}}(s) = 2\ell = O(s)$$

fixed size integer arithmetic operations.

In order to avoid mentionning the size of the limbs, costs are given in terms of number of bit operations, which is a constant multiple of the number of limb operations. This is called the bit complexity.

Multiplication

Multiplication of multiprecision integer is a richer question. The schoolbook multiplication algorithm, applied to the base 2^{64} representation gives a quadratic cost. More precisely, multiplying an s-bit integer by a t-bit integer involves $\lceil\frac{s}{64}\rceil\lceil\frac{t}{64}\rceil$ multiplications and as many additions for an overall bit complexity of

$$T_{\texttt{Mul}}(s,t) = O(st)$$

Yet, Karatsuba first showed that this complexity was not optimal and proposed an algorithm multiplying two integers of bit-size s in bit complexity $O(s^{\log_2 3}) = O(s^{1.585})$.

The algorithm relies on the following formula for multiplying two degree 1 polynomials,

$$(p_0 + p_1X) \times (q_0 + q_1X) = p_0q_0 + ((p_0+p_1)(q_0+q_1) - p_0q_0 - p_1q_1)X + p_1q_1X^2,$$

which involves 3 multiplications and 6 additions. Using this formula in a divide and conquer scheme leads to an algorithm multiplying two polynomials of degree $n-1$ in time

$$T_{\texttt{Mul}}(n) = 3T_{\texttt{Mul}}(n/2) + 8n$$

which solves in

$$T_{\texttt{Mul}}(n) = O(3^{\log_2 n}) = O(n^{\log_2 3})$$

This algorithm then directly translate to the multiplication of integers and yields the same estimate for the bit complexity.

More generally the cost analysis of Divide and Conquer algorithms can be automated by the so-called *Master Theorem*, given in Theorem 3.1.1.

Theorem 3.1.1

Let $T(n) \geq 0$ satisfying the recurring relation $T(n) = aT(n/b) + f(n)$ and $T(1) = K)$ a constant. Let $\alpha = \log_b a$, then

1. *If $f(n) = O(n^{\alpha-\varepsilon})$ for some $\varepsilon > 0$ then $T(n) = \Theta(n^{\alpha})$;*
2. *If $f(n) = \Theta(n^{\alpha})$ then $T(n) = \Theta(n^{\alpha}) \log n$;*
3. *If $f(n) = \Omega(n^{\alpha+\varepsilon})$ for some $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for $0 < c < 1$ and n sufficently large, then $T(n) = \Theta(f(n)$;*

Example 3.1.1

For Karatsuba's algorithm, the division in halves produces three recursive calls, hence the exponent $\alpha = \log_2 3 \approx 1.585$. Toom's algorithm computes the product of two degree 2 (size 3) polynomials in five multiplications, leading to a cost of $\Theta(n^{\log_3 5}) = O(n^{1.465})$.

More generally Toom-Cook's algorithm apply an evaluation-interpolation scheme to generate a family of algorithms multiplying polynomials of degree k in $2k-1$ products. Applied in a Divide and conquer scheme this gives an algorithm with cost $O(n^{\log_k 2k-1})$ proving that for any $\varepsilon > 0$ there is an algorithms multiplying polynomials of degree n in time $O(n^{1+\varepsilon})$, provided that the field has enough points of evaluation. See exercise in TD2 for further details.

The best known algorithms to multiply integer are now based on the Fast Fourier transform and have cost $O(n \log n)$. The same cost holds for polynomial over a finite field. To avoid the numerous logarithmic factors, it is common practice to introduce the soft-O notation: $f(n) = \tilde{O}(g(n))$ if $f(n) = O(g(n) \log^c g(n))$ for some $c > 0$.

Theorem 3.1.2

Let $\mathrm{I}(n)$ the cost of multiplying two n bit integers and by $\mathrm{M}(d)$ the cost and the cost of multiplying two degree d polynomials. The best estimates are $\mathrm{I}(n) = \tilde{O}(n)$ bit operations and is $\mathrm{M}(d) = \tilde{O}(d)$ field operations over a finite field.

Euclidean divisions and GCD

Theorem 3.1.3

The Euclidean division of $a \in \mathbb{Z}$ by $b \in Z$ can be computed in $O(\log(a/b) \log b)$ bit operations.

Proof. Algorithm 1 runs in $O(\frac{a}{b} \log b)$ bit operations and is therefore of no practical use. Instead, using the school-book division algorithm, one iteratively compute each bit of q (hence in $\log a/b$ iterations), each iteration being a substraction of a multiple of b which leads to the cost $\log(a/b) \log b$. The computation of the Bezout coefficients are bounded by this cost estimate. □

Theorem 3.1.4

The extended Euclidean algorithm run on two positive integers a and b costs $O(\log(a) \log(b))$ bit operations.

Proof. In the computation of the next residue r_{i+1}, the euclidean division of r_{i-1} by r_i costs $O((\log r_{i-1} - \log r_i) \log r_i)$. Summing over all iterations, since the sequence of r_i is decreasing,

$$\sum_{i=1}^{k} (\log r_{i-1} - \log r_i) \log r_i \leq \log r_0 \log r_1 = \log a \log b.$$

□

These cost estimates can be improved using divide and conquer approaches to reduce most of the computations to multiplications and therefore express the cost as a function of $\mathrm{I}(s)$ where s is the bit size of the instance. Such algorithms are beyond the scope of this lecture and we will not prove the following Theorem.

Theorem 3.1.5

1. *The Euclidean division of integers of bitsize bounded by s can be computed in $O(\mathrm{I}(s))$ bit operations.*
2. *The GCD of two integers of bitsize bounded by s can be computed in $O(\mathrm{I}(s)\log s)$ bit operations.*

3.2 The ring $\mathbb{Z}/n\mathbb{Z}$

Additions and multiplications

The straightforward implementation of the ring $\mathbb{Z}/n\mathbb{Z}$ is by embedding each input into $\mathbb{Z}$, use the suited arithmetic (fixed size or multiprecision, depending on the size of n), and perform a reduction modulo n of the result, by Algorithm 1. Over base type integer representations, this is achieved with either the `%` operator (for integral types) or by a call to the `fmod` function (for floating point types). In both cases, they do not correspond to hardware implemented operations, but to library calls, which typically incur a number of about 10 clock cycles for each division.

When n is too large to allow the use of hardware arithmetic, the reduction modulo n is performed using a Euclidean division in $O((\log x - \log n)\log n)$ where x is the output of the integer operation ($+$ or $\times$) to be reduced. This is respectively $O(\log n)$ for an addition, and $O(\log^2 n)$ for a multiplication.

Corollary 3.2.1

Over $\mathbb{Z}/n\mathbb{Z}$ additions and subtractions cost $O(\log n)$ bit operations and multiplications cost $O(\log^2 n)$ or $O(\mathrm{I}(\log n))$ using fast arithmetic.

Invertible elements and the multiplicative group

Theorem 3.2.1

An element $a \in \mathbb{Z}/n\mathbb{Z}$ has a multiplicative inverse if and only if $\gcd(a, n) = 1$.

Proof. $\gcd(a, n) = 1 \Leftrightarrow ua + vn = 1 \Leftrightarrow u = a^{-1} \mod n$. □

Corollary 3.2.2

$Z/n\mathbb{Z}$ is a field if and only if n is prime.

Proof. If n is prime, then it is coprime with any $x \in [\![1..n-1]\!]$ and by Corollary 2.1.1, $\exists u \in \mathbb{Z}, ux = 1 \mod n$, hence all non-zero elements of $\mathbb{Z}/n\mathbb{Z}$ are invertible. Reciprocally, if all non-zero elements of $\mathbb{Z}/n\mathbb{Z}$ are invertible, then they are coprime with n, hence n is prime. □

Now computationnaly speaking, this means that computing the inverse of an element a in $\mathbb{Z}/n\mathbb{Z}$ reduces to running the extended GCD algorithm on a and n.

Corollary 3.2.3

> *The inverse of an element in $\mathbb{Z}/n\mathbb{Z}$ can be computed in $O(\log^2 n)$ or $O(\mathrm{I}(\log n)\log\log n)$ bit operations.*

Theorem 3.2.2

> *Over a ring R, the set of invertible elements forms a group called the multilpicative group of R denoted by R^*.*

Proof. The neutral element for the multiplication 1 is always invertible. If x and y are invertible, then $x \times y$ also is, since $(x \times y) \times (y^{-1} \times x^{-1}) = 1$. Lastly, $(x^{-1})^{-1} = x$ shows that R^* is stable by inversion. □

For instance:

1. the multiplicative group of a field is formed by all the field elements except $0: K^* = K\backslash\{0\}$
2. Over $\mathbb{Z}/n\mathbb{Z}$, the inverses of any element of R^* are still in R^*.

Definition 3.2.1

> *The Euler Totient function, denoted by $\varphi(n)$ gives for any $n \in \mathbb{Z}_{\geq 2}$ the number of invertibles in $\mathbb{Z}/n\mathbb{Z}$: $\varphi(n) = |(\mathbb{Z}/n\mathbb{Z})^*|$.*

Proposition 3.2.1

> 1. *$\varphi(p) = p - 1$ for p prime,*
> 2. *$\varphi(p^k) = (p-1)p^{k-1}$ for p prime,*
> 3. *$\varphi(mn) = \varphi(m)\varphi(n)$ for m and n coprime.*

Proof.

1. comes directly from Corollary 3.2.2.
2. The non invertible elements of $\mathbb{Z}/p^k\mathbb{Z}$ are all multiples of p. There are p^{k-1} of them.
3. By Theorem 2.2.2.

□

As a consequence, the Euler totient function can be given for every integer from its decomposition in prime factors: for $n = \prod_{i=1}^{k} p_i^{e_i}$

$$\varphi(n) = \prod_{i=1}^{k} p_i^{e_i-1}(p_i - 1) = n\prod_{i=1}^{k}(1 - \frac{1}{p_i}) \tag{3.1}$$

Theorem 3.2.3 (*Lagrange*)

> *Over a finite group G, for all $a \in G$, $a^{|G|} = 1$.*

Proof. Consider $f_a : x \in G \mapsto ax$. Then $\prod_{x\in G} f_a(x) = a^{|G|}\prod_{x\in G} x$, but since f_a is a bijection, this is also equal to $\prod_{x\in G}$. Hence $a^{|G|} = 1$. □

Corollary 3.2.4

> *In a finite group G, the order of any element divides the order of the group.*

Proof. Let o be the order of an element $g \in G$, i.e. o is the smallest positive integer such that $g^o = 1$. Let n be the order of G and consider the Euclidean division $n = oq + r$ with $0 \leq r < o$. Then $1 = g^n = g^r(g^o)^q = g^r$. Thus $r = 0$ since o is minimal, hence o divides n. □

Theorem 3.2.4 (***Euler***)

If $GCD(a,n) = 1$ *then* $a^{\varphi(n)} = 1 \mod n$.

Proof. Applying Lagrange Theorem in $(\mathbb{Z}/n\mathbb{Z})^*$. □

Exercise 3.2.1

Assuming that the factorization of n is known,

1. *deduce two alternative algorithms computing the inverse of an invertible element in $\mathbb{Z}/n\mathbb{Z}$.*

2. *What is their bit complexity ?*

Theorem 3.2.5 (***Fermat***)

If p is prime, then for all $a \in \mathbb{Z}/p\mathbb{Z}$, $a^p = a \mod p$.

Corollary 3.2.5 (***RSA theorem***)

Let $n = pq$ with p, q prime numbers and $e \in \mathbb{Z}$ coprime with $\varphi(n) = (p-1)(q-1)$. Then the application

$$\begin{array}{rccc} E: & \mathbb{Z}/n\mathbb{Z} & \rightarrow & \mathbb{Z}/n\mathbb{Z} \\ & x & \mapsto & x^e \end{array}$$

is a bijection and its reciprocal is

$$\begin{array}{rccc} E: & \mathbb{Z}/n\mathbb{Z} & \rightarrow & \mathbb{Z}/n\mathbb{Z} \\ & x & \mapsto & x^d \end{array}$$

where $d = e^{-1} \mod \varphi(n)$

Proof. Since $ed = 1 \mod \varphi(n)$ there is a $k \in \mathbb{Z}$ such that $ed = 1 + k\varphi(n)$. Hence for all $x \in \mathbb{Z}/n\mathbb{Z}$, we have $x^{ed} = x^{1+k\varphi(n)} \mod n$.

Over $\mathbb{Z}/p\mathbb{Z}$, if $x = 0 \mod p$ then $x^{1+k\varphi(n)} = 0 = x$. Otherwise, $x^{\varphi(n)} = (x^{p-1})^{q-1} = 1$ by Theorem 3.2.4 and again $x^{1+k\varphi(n)} = x \mod p$. Similarly, modulo q we have $x^{1+k\varphi(n)} = x \mod q$, and by the Chinese remainder Theorem we have $x^{ed} = x^{1+k\varphi(n)} = x \mod n$. □

3.3 Finite Fields

3.3.1 Prime fields

We saw in Corollary 3.2.2 that any $\mathbb{Z}/p\mathbb{Z}$ for a prime p is a field. Computationnaly speaking they inherit from the arithmetic developped for the rings $\mathbb{Z}/n\mathbb{Z}$. The division is simply obtained by the computation of an inverse (using the extended Euclidean Algorithm) followed by a multiplication: $x/y = x \times y^{-1}$.

3.3.2 Extension fields

A natural question is to ask whether there are other finite fields than the prime fields, or at least fields that are non isomorphic to prime fields. The answer is positive, as we will present an construction producing fields with non-prime cardinality.

The quotient ring

Consider a ring R and an ideal $I \subset R$, namely, a subset stable by multiplication by any element of R: $\forall x \in R, a \in I, ax \in I$. One can define an equivalence relation $\equiv_I$ by $x \equiv_I y \Leftrightarrow x - y \in I$. The equivalence classes for this relation are the subsets $\bar{x} = x + I = \{x + y, y \in I\}$.

Definition 3.3.1

> *The set of all equivalence classes for the relation $\equiv_I$ in R is a ring called the quotient ring and denoted by R/I.*

We will now focus on the ring of univariate polynomials $K[X]$ over a field K. A polynomial $Q \in K[X]$ defines an ideal $(Q) = \{PQ, P \in K[X]\}$ and the quotient ring $K[X]/(Q)$ consists of all polynomials of degree $< \deg Q$, i.e. all possible residues modulo Q.

This set can be equipped with an addition $+_Q$, the natural polynomial addition, and a multiplication $\times_Q$, the multiplication modulo Q, to form a ring.

Definition 3.3.2

> $(K(X)/(Q), +_Q, \times_Q, 0, 1)$ *is a commutative ring called the quotient ring of* $K[X]$ *by* Q.

Theorem 3.3.1

> $K[X]/(Q)$ *is a field if and only if* Q *is irreducible.*

Proof. Similarly as for Corollary 3.2.2. □

Example 3.3.1

> *Over* $\mathbb{Z}/2\mathbb{Z}[X]$, *let* $P = (X + 1)(X^2 + X + 1)$.
>
> 1. *Then* $\mathbb{Z}/2\mathbb{Z}[X]/(P)$ *is not a field:* $X + 1$ *has no inverse since* $(X+1)(X^2+X+1) = 0$.
> 2. *But* $\mathbb{Z}/2\mathbb{Z}[X]/(X^2 + X + 1)$ *is a field. its elements are* $\{0, 1, X, X + 1\}$.

Corollary 3.3.1

> *The finite field* $\bar{K} = K[X]/(Q)$ *where* Q *is irreducible is a vector space over* K *of dimension* $k = \deg Q$. *Consequently* $\#\bar{K} = (\#K)^k$.

Reciprocally, we will show that any finite field is equivalent to either a prime field or a field $K[X]/(Q)$ where K is a prime field. This implies that any finite field has a cardinality of the form p^k for p prime and $k \in \mathbb{Z}_{>0}$. We therefore denote them by $\mathbb{F}_{p^k}$. When $k > 0$, they are called extension fields or Galois fields.

Structure of the finite fields

Let K be a finite field. In the additive group $(K, +, 0_K)$, the order of 1_K is necessarily finite, it is called the characteristic if K.

Lemma 3.3.1

> *The characteristic* p *of a finite field is necessarily a prime number. Moreover, the set* $k =$

$\{n.1_K | n \in \mathbb{Z}\}$ is a subfield of K of p elements isomorphic to the prime field $\mathbb{Z}/p\mathbb{Z}$.

Proof. Suppose that the characteristic is $p = mn$ and let $x = m.1_K \neq 0_K$ and $n = q.1_K \neq 0_K$. Yet, $xy = 0_K$ and y invertible implies $x = 0_K$, a contradiction.

The function $n \in \mathbb{Z}/p\mathbb{Z} \to n.1_K$ is a bijection between $\mathbb{Z}/p\mathbb{Z}$ and k. Since $n.1_K \times m.1_K = (mn).1_K$ and $n.1_K + m.1_K = (m+n).1_K$, it is an isomorphism and k is therefore a field. □

Hence a prime subfield k always exists in any finite field.

Theorem 3.3.2

A finite field K is a vector space over its prime subfield k.

Proof. K contains k, is stable by addition and external multiplication by k. □

Corollary 3.3.2

The cardinality of a finite field is of the form p^k where p is prime and $k \in \mathbb{Z}_{>0}$.

We will show that for all field $\mathbb{F}_p$ and all positive integer k, there exist an irreducible polynomial of degree k in $\mathbb{F}_p$. It is thus possible to be a finite field with p^k elements for any choice of p prime and $k \geq 0$: the prime fields $\mathbb{Z}/p\mathbb{Z}$ when $k = 0$, and the extension of the prime fields with an irreducible polynomial of degree k. Up to an isomorphism, these are the only finite field.

Theorem 3.3.3

The multiplicative group of a finite field is cyclic.

For this proof, we will need the following lemma:

Lemma 3.3.2

Let K be a finite field and n_d the number of elements of K^* of order d. If $n_d > 0$ then $n_d = \varphi(d)$.

Proof. Let α an element of order d. Then all of the α^i for $i \in [\![0..d]\!]$ satisfy $(\alpha^i)^d = 1$ and are therefore all the roots of the polynomial $Y^d - 1$. Hence all element of order d is of the form α^i. Laslty $\beta = \alpha^i$ has order d iff it can generate α: $\exists j, \beta^j = \alpha^{ij} = \alpha$. This is equivalent to $ij = 1 \mod d$. Hence there is exactly $\varphi(d)$ such elements. □

Lemma 3.3.3

$\sum_{d|n} \varphi(d) = n$

Proof. For every integers $x \in [\![0..n-1]\!]$, let $d = \text{GCD}(x, n)$.Then $x = yn/d$ where $y \in [\![0..d-1]\!]$ is coprime with d. There are exactly $\varphi(d)$ such integers having a GCD d with n. The n integers can then be counted $\sum_{d|n} \varphi(d) = n$. □

Proof of Theorem 3.3.3. Let $n = \#K^*$. From Corollary 3.2.4, every element in K has an order dividing n. Hence $\sum_{d|n} n_d = n$. On the other hand, $\sum_{d|n} n_d \leq \sum_{d|n} \varphi(d) = n$. Consequently, $n_d = \varphi(d)$ for all $d|n$. In particular, $n_n = \varphi(n) \neq 0$ which implies that there exist a generator of order n. □

Definition 3.3.3

The elements of order $q - 1$ in $(\mathbb{F}_q)^*$ are called primitive elements. They are primitive

$(q-1)$-th root of unity.

This leads to another point of view on the extension fields: $\mathbb{F}_{p^k}$ is the field $\mathbb{F}_p$ to which a primitive (p^k-1)-th root of unity α has been added (and all elements induced by applying the $+$ and $\times$ laws), i.e. the smallest field containing $\mathbb{F}_p$ and α. This field is also denoted by $\mathbb{F}_p(\alpha)$.

If $f \in \mathbb{F}_p[X]$ is the minimal polynomial of α, namely the least degree monic polynomial vanishing in α, then $\mathbb{F}_p(\alpha) = \mathbb{F}_p[X]/(f)$. In such cases, $X = \alpha$ generates the multiplicative group of the extension field.

However the converse is not always true: in an extension field $\mathbb{F}_p[X]/(Q)$, the element X may not necessarily generate the multiplicative group $(\mathbb{F}_p[X]/(Q))^*$.

Definition 3.3.4

A polynomial $f \in K[X]$ is primitive if it is irreducible over K and X generates the multiplicative group $(K[X]/(Q))^*$.

Galois fields in practice

These three viewpoints on Galois fields, each useful ingredient to implement their arithmetic efficiently.

Modular polynomial arithmetic. The representation of $\mathbb{F}_{p^k}$ as $\mathbb{F}_p[X]/(Q)$ where $Q \in \mathbb{F}_p[X]$ is irreducible of degree k translates naturally to an implementation using polynomial arithmetic. Field elements are polynomials of degree $< k$ over $\mathbb{F}_p$ and can be represented by array of k elements in $\mathbb{F}_p$.

- Field additions are polynomial additions, i.e. coefficient-wise addition over the base field $\mathbb{F}_p$ for which the techniques of Sections 3.1 and 3.3.1 apply. Note that in the case where $p = 2$ and $k \in \{8, 16, 32, 64\}$, field elements are represented by machine words of $8, 16, 32, 64$ bits for which the bitwise XOR operation is available via the `^` operator in `C`.

- Field multiplications are polynomial multiplications followed by a modular division performed by a Euclidean division. Alternatively, it can be achived by successive applications of a multiplication-by-X algorithm. If $Q = X^k - \sum_{i=0}^{k-1} q_i X^i$, then $X^k = \sum_{i=0}^{k-1} q_i X^i \mod Q$. Multiplying by X a degree $< k$ polynomial reduces to shifting the position of each of its coefficients by 1 position, and replacing the overflowing $p_{k-1}X^k$ term by $p_k \sum_{i=0}^{k-1} q_i X^i$. For instance, this operation for $\mathbb{F}_{256} = \mathbb{F}_2[X]/(X^8+X^4+X^3+X^2+1)$ can be implemented in `C` as follows

```
char mulByX (char a){ // in F256=F2[X]/(X^8+X^4+X^3+X^2+1)
  char b = a<<1;
  if (a & 128) // a is of degree 7
      b ^= 29;   // X^8 = X^4+X^3+X^2+1 = 29
  return b;
}
```

The multiplication algorithm then write as follows:

```
char mul (char a, char b){//in F256=F2[X]/(X^8+X^4+X^3+X^2+1)
  char r = 0;
  for (int i=0;i<8;i++){
     if (a & 1) r ^= b
     a >>= 1;
     b = mulByX (b);
  }
  return r;
}
```

Zech log representation. This alternative representation relies on the property of Theorem 3.3.3, that all non zero elements of a finite field can be generated by a unique element. Hence, a generator g of the multiplicative group is chosen once for all, and every element $x \in (\mathbb{F}_q)^*$ is represented by its discrete logarithm $i \in [\![0..q-1]\!]$ such that $x = g^i$. A special value is used to represent 0, for instance $i = q-1$.

- Multiplications become additions of the exponents: $x \times y = g^i g^j = g^{i+j} = g^{i+j \mod q-1}$ and are therefore very efficient.

- Unfortunately, additions are in contrast more complicated to handle. A table look-up approach is often used. Instead of storing a 2 dimensional addition table, one can only store 1-dimensional conversion tables: an addition is viewed as a multiplication: Since $g^i + g^j = g^i(1 + g^{j-i \mod q-1})$, one first compute $k = j - i \mod q-1$, then look-up in a pre-computed table, the discrete logarithm ℓ such that $g^\ell = 1 + g^k$, and lastly perform the addition $i + \ell \mod q-1$.

Chapter 4

Coding Theory

Coding theory aims to ensure or at least increase the integrity of numerical data, in a context where errors may alter it. This is motivated by a wide range of applications :
- long distance communication through radio waves for satellite of spatial probes;
- shorter distance radio communications for mobile devices (phones, bluetooth devices, etc)
- electric communication on of computer networks (ethernet)
- optic storages (audio CD, DVD, blue-ray)
- mass storage on hard drives disks, some of which may fail

As an abstraction for all these settings, one can consider the following setting: a *sender* wants to transmit a *message* in the form of a sequence of *symbols* to a *receiver* through a *canal* which may introduces errors on some symbols. The goal is to either *detect* the errors (in order to ask for a re-send) or *correct* them (which may be mandatory when interaction is not an option). This goal is achieved by the introduction of redundancy to the message. For instance, the NATO phonetic alphabet associates to each letter of the alphabet a word starting by this letter *Alpha, Bravo, Charlie, Delta*, so as to remove the phonetic ambiguities, such as B and D or F and S. The information can be handled as a stream of symbols or block by block.

4.1 Linear codes

From now on we will only consider block codes for which the treatment is only defined on fixed length vectors of symbols. Furthermore, some algebraic structure is also introduced, namely that of vector spaces.

Definition 4.1.1

A linear code is a linear subspace $\mathcal{C} \subseteq \mathcal{A}$ of a finite dimensional space $\mathcal{A}$ over a field K. The dimension of the code is the dimension of $\mathcal{C}$, the length of the code is the dimension of $\mathcal{A}$. The elements of K are the symbols of the code. Any vector $c \in \mathcal{C}$ is a code-word.

Example 4.1.1

1. *The parity check code is defined as $\mathcal{C}_{\text{par}} = \{x \in K^n : x_n = -\sum_{i=1}^{n-1} x_i\}$*
2. *The n-repetition code is defined as $\mathcal{C}_{rep} = \{x \in K^n : x_1 = x_2 = \cdots = x_n\}$*

As a linear subspace, a code can be defined as the image of linear application $E : K^k \to K^n$, or as the kernel of another linear application $V : K^n \to K^{n-k}$. The application E corresponds to the encoding function, changing any source message of k symbols $m \in K^k$ into a code-word $c \in \mathcal{C} \subseteq K^n$. The application V is such $V(x) = 0 \in K^{n-k}$ if and only if $x \in \mathcal{C}$, and acts therefore as a verification: it can detect the presence of errors.

However, introducing some redundancy can not ensure that any error will be detected or corrected. In order to quantify the detection and correction capacities, one need to introduce a metric to measure the distance from a received word to a code word.

4.1.1 The Hamming metric

Definition 4.1.2

1. *The Hamming weight of a vector $x \in K^n$ is the number of its non-zero coefficients: $w_H(x) = \{i \in [\![1..n]\!] : x_i \neq 0\}$.*
2. *The Hamming distance between two vectors $x, y \in K^n$ is the number of position where they differ: $d_H(x, y) = \{i \in [\![1..n]\!] : x_i \neq y_i\} = w_H(x - y)$.*
3. *The minimum distance of a code $\mathcal{C}$ is the least distance between two distinct code words: $d_\mathcal{C} = \min_{x,y\in\mathcal{C},x\neq y} d_H(x, y)$*

Proposition 4.1.1

The minimum distance of linear code is the smallest Hamming weight of its non-zero code words: $d_\mathcal{C} = \min_{x\in\mathcal{C}\setminus\{0\}} w_H(x)$.

Definition 4.1.3

1. *A code $\mathcal{C}$ is t-detector if any error of weight $\leq t$ can be detected, i.e. if $\forall x \in \mathcal{C}, \forall e \in K^n$ with $w_H(e) \leq t$, then $x + e \notin \mathcal{C}$.*
2. *A code C is t-corrector, if any error of weight $\leq t$ can be uniquely corrected, i.e. if $\forall x \in K^n, \#\{x \in \mathcal{C} : d_H(c, x) \leq t\} \leq 1$.*

Example 4.1.2

1. *The parity check code $\mathcal{C}_{par}$ is 1-detector, and 0-corrector.*
2. *The n-repetition code $\mathcal{C}_{rep}$ is $n-1$-detector, and $\lfloor \frac{n-1}{2} \rfloor$-corrector.*

Proposition 4.1.2

A code with minimum distance d is $\lfloor \frac{d-1}{2} \rfloor$-corrector.

Proof. Let $t = \lfloor \frac{d-1}{2} \rfloor$ For any $x \in K^n$, suppose there is are two codewords $c, c' \in \mathcal{C}$ such that $d_H(x, c) \leq t$ and $d_H(x, c') \leq t$. Then $d_H(c, c') \leq d_H(c, x) + d_H(x, c') \leq 2t \leq d - 1$. Hence $c = c'$ by the minimality of d. □

4.1.2 Generating and parity check matrices

In coding theory, as generally in linear algebra, matrices arise naturally to represent linear applications and basis of vector spaces, given a basis for their representation.

Definition 4.1.4

The generating matrix of a code is an $k \times n$ matrix G that defines the code as its rowspace: $\mathcal{C} = \{x^T G : x \in K^k\}$.

Definition 4.1.5

A generating matrix is in systematic form if the submatrix formed by its first k columns is the identity matrix.

The encoding through a systematic matrix, then consists in copying the first k symbols and computing the $n-k$ redundancy symbols by linear combinations of these symbols.

Definition 4.1.6

A parity check matrix of a code is a matrix H which kernel is the code: $\mathcal{C} = \{x \in K^n : Hx = 0\}$.

The parity check matrix can be used as a tool to detect errors by simply applying this matrix to the received word, producing an $n-k$ vector called *syndrom*. Any received word not in the code will generate a non-zero syndrom proving the existence of an error. This is true for any error up to the detection capacity $d-1$.

An error correction algorithm can be derived by precomputing all possible syndroms $s = He$ for any possible error pattern $e \in K^n$ with $w_H(e) \leq t$, where t is the correction capacity. Upon receiving a word $r = c + e$, where $c \in \mathcal{C}$ and e is the error vector, one can compute its syndrom $s = Hy = H(c+e) = He$ since $Hc = 0$. Hence a look in the table, will return which error e generates this syndrom, and the correction $c = r - e$ is then possible.

4.1.3 Bounds on the correction capacity

Obviously, the correction capacity can not be arbitrarily large, and depends on the amount of redundancy available, namely the difference between the length of the code and its dimension.

Theorem 4.1.1 (*Singleton bound*)

A linear code of length n and dimension k has a minimum distance d verifying $d \leq n-k+1$.

Proof. The parity check matrix H has $n-k$ rows and full rank. Hence any subset of $n-k+1$ columns of H is linearly dependent. If $c = (c_1, \dots, c_n)$ is the coefficients for this linear dependency relation, then $Hc = 0$ for a non-zero vector c of Hamming weight $n-k+1$. □

Definition 4.1.7

Codes attaining the Singleton bound are called Maximum Distance Separable (MDS).

4.2 Reed-Solomon codes

Reed-Solomon codes rely on a change of representation provided by the evaluation-interpolation scheme of Lagrange interpolation Theorem 2.2.3. There, a degree $\leq k$ polynomial is equivalently represented by the value of k evaluations in distinct points $x_1, \dots x_k$. In this conversion process, some redundancy can be added by oversampling the polynomial in more evaluation points, which will be exploited to detect and correct errors on some of these evaluations.

Consequently, the Reed-Solomon code can be defined as the set of evaluation vectors of all polynomials with bounded degrees.

Definition 4.2.1

Given n distinct field elements $x_1, \dots, x_n \in K$, the Reed-Solomon code of parameters $[n, k]$ is defined by

$$\mathcal{C}[n, k] = \{(f(x_1), \dots, f(x_n)) \in K^n \mid f \in K[x]_{<k}\}.$$

Proposition 4.2.1

A Reed-Solomon code $\mathcal{C}[n, k]$ is a linear code of length n and dimension k.

Proof. By Lagrange interpolation Theorem 2.2.3, it is isomorphic with the vector subspace $K[x]_{<k}$. □