Part I

Error Correcting Codes

Chapter 1

Finite fields

Finite fields are a very valuables source of combinatoric constructions, especially in coding theory and cryptography. In this first chapter we will construct these fields and review their properties. This chapter is meant as an overview and hence we only state results without proving them.

1.1 Prime fields

As we know a field $\mathbb{F}, +, \cdot$ is an algebraic structure where $\mathbb{F}, +$ is a commutative group with neutral element denoted by 0 (i.e associative, commutative, neutral element, inverses exist) and $\mathbb{F} \setminus \{0\}, \cdot$ is a commutative group with neutral element 1. Finally between the addition and the multiplication the distributive laws holds. This means that in a random field one can do exactly the same operations as one would do with the rational, real or complex numbers. The aim of this chapter is now to find examples of fields that contain only a finite number of elements.

The starting point of our expedition is the ring of the integers, $\mathbb{Z}$. $\mathbb{Z}$ has most properties of a field, except that there are not inverses for the multiplication. One can solve this problem by introducing fractions leading us to the field of rational numbers $\mathbb{Q}$. However in this way one obtains an infinite field.

There is a second possibility to turn $\mathbb{Z}$ into a field. Although we cannot always divide in $\mathbb{Z}$, there is a division algorithm that given numbers a and p produces a quotient q and a rest r such that

$$a = pq + r,\ 0 \leq r < |p|$$

If we take a fixed p, the trick is now to treat numbers with the same rest divided by p as the same, and define the addition and multiplication up to the rests;

Definition 1.1. We say that $a, b \in \mathbb{Z}$ are equivalent modulo $p \in \mathbb{N}$ if the differ only by a multiple of p.

$$a \equiv b \mod p \ \Leftrightarrow\ p|a - b$$

The equivalence class of a modulo p is the set of integers that are equivalent modulo p with a:

$$[a]_p := \{b \in \mathbb{Z} | a \equiv b \mod p\}$$

The ring $\mathbb{Z}_p := \{[a]_p | a \in \mathbb{Z}\}$ is the ring of rest classes modulo p together with the obvious addition and multiplication.

$$[a]_p + [b]_p := [a + b]_p,\ [a]_p \cdot [b]_p := [a \cdot b]_p.$$

For sake of simplicity we will abandon the notation $[a]_p$ in favor of a when it is obvious what we mean.

One can check that for every number in $\{0, \ldots, p-1\}$ there is a unique equivalence class so one can identify the classes with those numbers. Addition and multiplication are a piece of cake: one computes it in the ring of integers and then one takes the rest for division by p.

Computing inverses for the multiplication is more complicated. $[b]_p$ is the inverse of $[a]_p$ if

$$[a]_p[b]_p = [1]_p \text{ or } \exists q \in Z : ab + pq = 1.$$

For random a and p such a b can only exists if their greatest common divisor is 1 and in that case one can compute this b by using the algorithm of Euclid.

Algorithm 1.1 (The algorithm of Euclid)**.** Suppose a and b two integers, we compute their gcd and write it as a linear combination of a and b.

1. Set r_0 the biggest of a and b in absolute value, and r_1 the other one. Put $i = 1$ and define

$$\begin{aligned} t_0 &= 1, & t_1 &= 0 \\ s_0 &= 0, & s_1 &= 1 \end{aligned}$$

2. If $r_i \neq 0$ divide r_{i-1} by r_i and call the quotient q_i and the rest r_{i+1}. Put also

$$\begin{aligned} t_{i+1} &= -q_i t_i + t_{i-1} \\ s_{i+1} &= -q_i s_i + s_{i-1} \end{aligned}$$

   increment i by 1 and repeat this until $r_i = 0$. If this is the case r_{i-1} is the greatest common divisor.

3. Because of the definition of the s and t we have for each j the identity

$$r_j = t_j a + s_j b$$

   Especially for $j = i - 1$ we have expressed the greatest common divisor in terms of the original polynomials.

Because inverses only exist if the gcd is 1, $\mathbb{Z}_p$ will only be a field if p is a prime in this case every $a \neq [0]_p$ will have $gcd(a, p) = 1$.

Theorem 1.1. $\mathbb{Z}_p$ is a field if and only if p is a prime, in that case we also denote is by $\mathbb{F}_p$.

Exercise 1.1. Write source code that enable you to add, subtract, multiply and divide in $\mathbb{F}_p$ for a random prime p.

1.2 Polynomial rings

In the previous section we have already constructed a infinite number of finite fields, but there are more. In order to find the others we will introduce here polynomial rings over fields.

Definition 1.2. The ring of polynomials over a field $\mathbb{F}$ is

$$\mathbb{F}[X] := \{\sum_{i=0}^{n} a_i X^i | 0 \leq n < \infty, a_i \in \mathbb{F}\}$$

Addition and multiplication are the usual like in $\mathbb{R}[X]$. The degree of a polynomial is the highest coefficient

Example 1.1. Adding polynomials over $\mathbb{F}_2$ is like adding normal polynomials keeping in mind that $1 + 1 = 0$ and hence

$$\begin{aligned}(X^5 + X^3 + X + 1) + (X^4 + X^3 + 1) &= X^5 + X^4 + (1+1)X^3 + X + (1+1) \\ &= X^5 + X^4 + X.\end{aligned}$$

The product of two polynomials is also obvious

$$\begin{aligned}(X^2 + 1)(X^3 + X + 1) &= (X^5 + X^3 + X^2) + (X^3 + X + 1) \\ &= X^5 + X^2 + X + 1\end{aligned}$$

This structure is not a field because we still don't have inverses but we have overcome the problem of the zero divisors because the degree (the highest non-zero power of X) of the product of two polynomials is the sum of the degrees of those two polynomials. Nevertheless it is an important structure in the way that all possible finite fields are derived from such rings.

Although we don't have inverses in $\mathbb{F}[X]$, we can divide polynomials with the division algorithm. Given two polynomials $a(X), b(X)$ we can compute the quotient $q(X)$ and rest $r(X)$ such that

$$a(X) = b(X)q(X) + r(X) \text{ and } \deg r(X) < \deg b(X)$$

f.i. in $\mathbb{F}_2$

$$X^4 + X^2 + X + 1 = (X^2 + X + 1)(X^2 + X + 1) + X$$

Just as in normal division theory with the natural numbers, we can define prime (or more correct irreducible) polynomials which have no nontrivial divisors. Like with ordinary numbers we can decompose every polynomial in his prime components.

$$X^4 + X^2 + X + 1 = (X + 1)(X^3 + X^2 + 1), (\text{ in } \mathbb{F}_2).$$

We also can define te concept op greatest common divisor (gcd) and least common multiple (lcm) of two or more polynomials. An important property of the gcd is that we can express $gcd(a(X), b(X))$ as a combination of multiples of $a(X)$ and $b(X)$. For this we again use the algorithm of Euclid, adapted to polynomials. The *lcm* on the other hand can be calculated by dividing the product of the two polynomials by its *gcd*.

1.3 Quotient rings

As we have seen a polynomial ring behaves more or less like the ring of integers, $\mathbb{Z}$. Therefore it can be used in the same way to construct new finite fields.

Consider a polynomial $g(X)$ of degree n in $\mathbb{F}_p[X]$ Two polynomials $a(X)$ and $b(X)$ are said to be equivalent modulo $g(X)$ if they have the same rest if divided by $g(X)$ or equivalently if their difference divides $g(X)$. We write

$$a(X) = b(X) \mod g(X)$$

We construct a new ring by looking again only up to equivalence modulo $g(X)$. We consider a new ring $\mathbb{F}_2[X]/(g)$ which consists of elements

$$[a(X)]_g := \{b(X) \in \mathbb{F}_2[X] | a(X) = b(X) \mod g(X)\}$$

the addition and multiplication are defined as

$$[a(X)]_g + [b(X)]_g = [a(X) + b(X)]_g,\ [a(X)]_g \cdot [b(X)]_g = [a(X)b(X)]_g$$

Because rests always have a lower degree than g, every polynomial is equivalent with one of degree smaller than n. Therefore there are only p^n elements in this ring corresponding to all possible rests. The easiest way to work in this ring is to work only with the rests and each time you multiply you must calculate the rest of the product divided by g.

It is not necessarily true that this ring has no zero divisors because we could have the situation that the product of two rests gives us a multiple of $g(X)$ f.i.

$$[X^2]_{X^4}[X^3]_{X^4} = [X^5]_{X^4} = [0]_{X^4}$$

This is only possible when $g(X)$ is not a prime polynomial because if $g(X)$ were prime and $g(X)|a(X)b(X)$ then $a(X)$ or $b(X)$ must contain $g(X)$ in its prime decomposition and thus its rest will be zero. Computing inverses is also done by using the algorithm of Euclid for polynomials.

For sake of simplicity we will drop the notation $[\cdot]_g$ and denote $[X]_g$ by a Greek letter f.i. ξ. We easily have that $[a(X)]_g := a(\xi)$.

Example 1.2. If we take $g(X) := X^2+X+1$ the field $\mathbb{F}_2[X]/(g(X))$ will consist of four elements: 0, 1, ξ, $\xi+1$ with the following tables for multiplication and addition:

$+$	0	1	ξ	$\xi+1$
0	0	1	ξ	$\xi+1$
1	1	0	$\xi+1$	ξ
ξ	ξ	$\xi+1$	0	1
$\xi+1$	$\xi+1$	ξ	1	0

$\cdot$	1	ξ	$\xi+1$
1	1	ξ	$\xi+1$
ξ	ξ	$\xi+1$	1
$\xi+1$	$\xi+1$	1	ξ

Example 1.3. If we take $g(X) := X^2+1$ then $\mathbb{F}_2[X]/(g(X))$ is not a field anymore) because $X^2+1 = (X+1)^2$. it will consist again of four elements: 0, 1, ξ, $\xi+1$, but $\xi+1$ won't be invertible because $(\xi+1)^2 = 0$.

$\cdot$	1	ξ	$\xi+1$
1	1	ξ	$\xi+1$
ξ	ξ	1	$\xi+1$
$\xi+1$	$\xi+1$	$\xi+1$	0

The number of elements in a finite field will be exactly p^n with n the degree of the prime polynomial. For $p = 2$, $n = 2$ we just have one irreducible polynomial: X^2+X+1. For n bigger this is not true anymore. If n is 3 we have exactly 2 irreducibles:

$$X^3+X+1,\ X^3+X^2+1.$$

This does not mean that there are two different kinds of fields with 8 elements. Mathematically $\mathbb{F}_2[X]/(X^3+X+1)$ and $\mathbb{F}_2[X]/(X^3+X^2+1)$ are isomorphic.

This means that we can identify the elements of both fields. To put it more clearly take $\xi := [X]_{X^3+X+1}$ and $\eta := [X]_{X^3+X^2+1}$ than we see that

$$\begin{aligned}(\xi+1)^3+(\xi+1)^2+1 &= \xi^3+\xi^2+\xi+1+\xi^2+1+1\\ &= \xi^3+\xi+1=0\end{aligned}$$

So $\xi+1$ fulfills the same equation in the first field as η in the second field. Via this method we can identify all the elements of the two fields with each other:

$$\begin{array}{ccc|ccc} 0 & \longrightarrow & 0 & \xi^2 & \longrightarrow & \eta^2+1\\ 1 & \longrightarrow & 1 & \xi^2+1 & \longrightarrow & \eta^2\\ \xi & \longrightarrow & \eta+1 & \xi^2+\xi & \longrightarrow & \eta^2+\eta\\ \xi+1 & \longrightarrow & \eta & \xi^2+\xi+1 & \longrightarrow & \eta^2+\eta+1 \end{array}$$

One can prove that for every n there exists at least 1 irreducible polynomial of degree n and that if there exists more of them they all induce isomorphic fields.

Theorem 1.2. For every prime power $q := p^n$ there exists a unique finite field with q elements, this is called the *Galois field with q elements* and is denoted by $\mathbb{F}_q$.

Miniature 1: Evariste Galois (1811-32)

Mathematican, born in Bourg-la-Reine, France. He was educated privately and at the Collège Royal de Louis-le-Grand. Despite mathematical ability he failed the entrance for the Ecole Polytechnique to study maths, and settled for the Ecole normale Supérieure in 1829 to train as a teacher, but was expelled in 1830 for republican sympathies. He engaged in political agitation, was imprisoned twice, and was killed in a duel aged 21. His mathematical reputation rests on original genius in the branch of higher algebra known as group theory.

1.4 Galois fields

We will now investigate some properties of those fields. In what follows we will set q equal to p^n.

Theorem 1.3. for a finite field $\mathbb{F}_q$, $\mathbb{F}_q^* := \mathbb{F}_q \setminus \{0\}$ will be a commutative cyclic group with $q-1$ elements.

This theorem states that there exists an element $\alpha \in \mathbb{F}_q^*$ such that every other element of $\mathbb{F}_q^*$ can be expressed as a power of α. For example in $\mathbb{F}_8$ we can take ξ itself:

$$\begin{array}{rclcrcl} [0 & = & 0] & & \xi^2 & = & \xi^2 \\ 1 & = & \xi^0 & & \xi^2+1 & = & \xi^6 \\ \xi & = & \xi^1 & & \xi^2+\xi & = & \xi^4 \\ \xi & = & \xi^3 & & \xi^2+\xi+1 & = & \xi^5 \end{array}$$

Here we used all the powers of ξ until the sixth, if we compute the seventh power we will see that its again 1 so we have for every element in $\mathbb{F}_8^*$ an infinite number of possibilities to express it as a power of ξ:

$$\xi + 1 = \xi^3 = \xi^{10} = \xi^{17} = \dots$$

One can also take η because $\xi^3 = \eta$ thus $\xi = \xi^{15} = \eta^5$ and so f.i.

$$\xi^2 + \xi + 1 = \xi^5 = \eta^{25} = \eta^4.$$

However it is not true that one can always take the generator element of the Galois field. Consider $\mathbb{F}_{16} \cong \mathbb{F}_2[X]/(X^4+X^3+X^2+X+1)$ and take ξ to be the equivalence class of X. One can compute that

$$\begin{aligned} \xi^5 &= \xi(\xi^3+\xi^2+\xi+1) \\ &= (\xi^3+\xi^2+\xi+1)+\xi^3+\xi^2+\xi \\ &= 1. \end{aligned}$$

In general there is no algorithm to find a generator of the cyclic group, so one has to do some trial and error.

Consider an element $\alpha \in \mathbb{F}_q$. A polynomial $a(X) \in \mathbb{F}_2[X]$ such that $a(\alpha) = 0$ is called a *characteristic polynomial* for α, the characteristic polynomial of least degree is called the *minimal polynomial* of α. One can show that the minimal polynomial divides every other characteristic polynomial and that it is irreducible. F.i. In $\mathbb{F}_4$ $s(X) = X^3 + 1$ is a characteristic polynomial for ξ because $\xi^3 = (\xi + 1)\xi = \xi + 1 + \xi = 1$ but it is not its minimal polynomial since

$$X^3 + 1 = (X + 1)(X^2 + X + 1) \text{ and } \xi^2 + \xi + 1 = 0.$$

The degree of an element of a finite field is defined as the degree of its minimal polynomial. Not all the elements of a finite field have the same degree. In $\mathbb{F}_{16} = \mathbb{F}_2[X]/(X^4 + X + 1)$ ξ obviously has degree 4 but for ξ^5 we have that

$$\xi^{10} = \xi^2(\xi + 1)^2 = \xi^4 + \xi^2 = \xi + 1 + \xi^2 = \xi(\xi + 1) + 1 = \xi^5 + 1.$$

So ξ^5 has minimal polynomial X^2+X+1. This implies also that $\mathbb{F}_{16}$ contains a subfield generated by ξ^5. This subfield contains the 4 elements 0, 1, ξ^5 and $\xi^5 + 1$ and it is isomorphic to $\mathbb{F}_4$. Although $\mathbb{F}_{16}$ contains $\mathbb{F}_4$ it does not contain $\mathbb{F}_8$. This is because if it would there would exist an element α such that $\alpha^7 = 1$ suppose that this element can be written as ξ^j then $\xi^{7j} = 1 = \xi^{15}$ so 15 must divide $7j$ and thus also $15|j$ and $\alpha = 1$ which is a contradiction.

Theorem 1.4. Generally one can embed $\mathbb{F}_{p^m}$ in $\mathbb{F}_{p^n}$ if and only if $m|n$, moreover there is only one subfield of $\mathbb{F}_{p^n}$ isomorphic to $\mathbb{F}_{p^m}$. If ξ is a generator of $\mathbb{F}^*_{p^n}$ and then $\mathbb{F}^*_{p^m}$ will be generated by

$$\xi^{\frac{p^n-1}{p^m-1}}.$$

Some elements of a finite field have the same minimal polynomial, in $\mathbb{F}_4$ e.g. ξ and $\xi + 1$ both satisfy $X^2 + X + 1$, ξ by definition, $\xi + 1$ because

$$(\xi + 1)^2 + \xi + 1 + 1 = \xi^2 + 1 + \xi + 1 = \xi + 1 + 1 + \xi + 1.$$

One can prove that every minimal polynomial of an element in a finite field has as many roots as it's degree. In $\mathbb{F}_8$ for example the minimal polynomial of ξ is $X^3 + X + 1$, but one can compute also that but ξ^2 and ξ^4 are roots of the same polynomial.

$$(\xi^2)^3 + \xi^2 + 1 = (\xi + 1)^2 + \xi^2 + 1 = \xi^2 + 1 + \xi^2 + 1 = 0$$
$$(\xi^4)^3 + \xi^4 + 1 = (\xi + 1)^4 + \xi^4 + 1 = \xi^4 + 1 + \xi^4 + 1 = 0.$$

this is also a general rule if α is a root of a polynomial then $\alpha^p, \alpha^{p^2}, \alpha^{p^3}, \ldots$ will be also roots. Take care some, of those roots will be the same as previous ones. In $\mathbb{F}_8$ We have that $\xi^8 = \xi$, so there aren't any new roots apart from ξ, ξ^2 and ξ^4.

Because $\mathbb{F}_q^*$ is a cyclic group of $q-1$ elements, we have that $\alpha \in \mathbb{F}_q^*$

$$\forall \alpha \in \mathbb{F}_q^* : \alpha^{q-1} = 1$$

This implies that every element of $\mathbb{F}_q$, including the zero is a root of the polynomial $X^q - X$. So all the minimal polynomials divide $X^q + X$ so that $X^q + X$ is the product of all minimal polynomials of elements in $\mathbb{F}_q$.

$$X^9 - X = \underbrace{X}_{0}\underbrace{(X+1)}_{1}\underbrace{(X^3+X+1)}_{\xi,\xi^2,\xi^4}\underbrace{(X^3+X^2+1)}_{\xi^3,\xi^5,\xi^6}$$

As we considered polynomials over the prime field $\mathbb{F}_p$, we can also consider polynomials over the field $\mathbb{F}_q$. Most of the properties of these polynomials are the same as those of $\mathbb{F}_p[X]$. We again have a division algorithm and one can also define irreducible polynomials, and we have a unique factorization theorem. Notify however that a polynomial which is irreducible over $\mathbb{F}_2$ might not be irreducible for a bigger field. In $\mathbb{F}_4$

$$X^2 + X + 1 = (X + \xi)(X + \xi + 1).$$

In general when α is a root of a polynomial $a(X)$ over a field then $X - \alpha$ divides $a(X)$. To calculate the quotient $a(X)/(X + \alpha)$ one uses the method of Horner.

Example 1.4. Compute the quotient of $f(X) := X^4+\xi X^3+X^2+(\xi+1)X+\xi$ divided by $X - \xi$ over $\mathbb{F}_4$

	1	ξ	1	$\xi+1$	ξ
+		ξ	0	ξ	ξ
=	1	0	1	1	0

In the upper line we put the coefficients of $f(X)$. The third line is the sum of the two upper lines and the k^{th} element of the second line is ξ times the $(k-1)^{th}$ element of the third line. The coefficients of the quotient are all but the last element of the third line, so here the qoutient is $X^3 + X + 1$.

One could also try to produce fields taking polynomials over $\mathbb{F}_q$ but those fields will be isomorphic to the one we already found. For instance if we take the polynomial $X^2 + X + \xi$ over $\mathbb{F}_4$ and represent $\alpha = [X]_{X^2+X+\xi}$ than we can identify the new field with $\mathbb{F}_{16}$ because

$$\begin{aligned}\alpha^4 &= (\alpha + \xi)^2 \\ &= \alpha^2 + \xi^2 \\ &= \alpha^2 + \xi + 1 \\ &= \alpha^2 + \alpha^2 + \alpha + 1 \\ &= \alpha + 1\end{aligned}$$

So α satisfies the polynomial $X^4 + X + 1$ over $\mathbb{F}_2$ and so we can identify it with an element of $\mathbb{F}_{16}$.

Exercise 1.2.

- Write source code that can add, multiply and divide over a finite field $F_p[X]/(g(X))$ if p and $g(X)$ are given.
- Write source code that can find a generator element of $F_p[X]/(g(X))^*, \cdot$ and produces a conversion table between the additive and the exponential notation of the finite field.
- Write source code that generates prime polynomials over $\mathbb{F}_p[X]$

Chapter 2

Linear Codes

2.1 Error correcting codes

Almost all systems of communication are confronted with the problem of noise. This problem has many different causes both human or natural and imply that parts of the messages you want to send, are falsely transmitted. The solution to this problem is to transmit the message in an encoded form such that contains as certain amount of redundancy which enables us to correct the message if some errors have occurred.

Mathematically one can see a coded message as a sequence of symbols chosen out of a finite set $\mathcal{F}$. If an error occurs during the transmission, some of the symbols will be changed, and the sequence received will differ at some positions from the original. If one would allow to transmit all possible sequences, one could never correct any errors because one could always assume that the received message is the one sent and that no errors have occurred.

To avoid this problem we restrict ourself to transmit only special sequences which we call *code words*. If few errors occur during the transmission one will see that the received message is not a codeword. In this way the receiver knows that errors have occurred. To correct the message one searches for the code word which is the closest to the received message, i.e. which differs at the least number of positions with the received message.

Example 2.1. Suppose you have to navigate at distance a blind person through a maze. One could use a morse like coding system:

Action	Code
One step forward	..
One step Backward	−−
Turn left	.−
Turn right	−.

This code is useful as long as no errors occur, because even if only a single error occurs the blind person would not notice this, understand a different command and probably walk against a wall.

To improve the system one could use this code

Action	Code
One step forward	...
One step Backward	− − .
Turn left	. − −
Turn right	−.−

All the codewords here contain an even number of bars, if one error occurs during the transmission, there will be one bar more or less and the received message will no longer be a code word. The blind person will notice this and could ask for retransmission. However he will not be able to correct the message himself because if he receives −.., the original message could be ... or −.−. We call such a code a single error detecting code.

To do better we use the concatenation of the previous two codes.

Action	Code
One step forward	.....
One step Backward	− − − − .
Turn left	. − . − −
Turn right	−.. − −

If here one error occurs, the blind person can deduce what the original message was because if the error occurred in the first two symbols he will see that the last three digits form a code word of the second code and decode the last tree symbols. If the error occurs in the last tree digits these will no more be a code word of the second code, so he nows that the first two digits are correct and take these to decode. This code is a single error correcting code.

We 've seen that we can construct codes that are able to correct mistakes, but the prize for it is redundancy. As we saw above we had to use five symbols to transmit, symbols in stead of two, to produce a single error correcting code.

We will now formalize all this to get an exact definition of the concept code:

Definition 2.1. Suppose $\mathcal{F}$ is a finite set of symbols. A *n-code* C over $\mathcal{F}$ is a subset of $\mathcal{F}^n$. If $\mathcal{F} = \mathbb{F}_2$ we call the code *binary*.

So for the last example the used code

$$C := \{\mathtt{00000},\ \mathtt{11110},\ \mathtt{01011},\ \mathtt{10101}\}$$

is a binary 5-code.

Definition 2.2. An *error* occurred at the i^{th} place changes the i^{th} entry of a codeword. f.i.

$$11110 \mapsto 10110$$

is an error occurred at the second place.

Definition 2.3. The *Hamming distance* $d(\mathtt{x}, \mathtt{y})$ between two sequences of symbols $\mathtt{x}, \mathtt{y}$ is the number of places where they differ f.i.

$$d(\mathtt{11110}, \mathtt{01011}) = 3$$

This function is really a distance function meaning that it satisfies the following easily to prove properties

- $\forall \mathtt{x}, \mathtt{y} \in C : d(\mathtt{x}, \mathtt{y}) = d(\mathtt{y}, \mathtt{x})$
- $\forall \mathtt{x}, \mathtt{y} \in C : d(\mathtt{x}, \mathtt{y}) = 0 \Leftrightarrow \mathtt{x} = \mathtt{y}$

- $\forall \mathtt{x}, \mathtt{y}, \mathtt{z} \in C : d(\mathtt{x}, \mathtt{y}) \leq d(\mathtt{x}, \mathtt{z}) + d(\mathtt{z}, \mathtt{y})$

When i errors occur the Hamming distance between the original code word and the received message will be i. The normal procedure to correct the received message is to assume that the least possible errors occurred so the reconstructed codeword will be the one with the smallest Hamming distance from the received message.

Example 2.2. If we are still using code C and you receive as message `11010` you will decode it as 11110 because

$$d(\mathtt{11010}, \mathtt{00000}) = 3$$
$$d(\mathtt{11010}, \mathtt{11110}) = 1$$
$$d(\mathtt{11010}, \mathtt{01011}) = 3$$
$$d(\mathtt{11010}, \mathtt{10101}) = 4$$

and it is most likely that only 1 error had occurred. If the chance of transmitting a correct bit is 90% then the probability that only one error occurred is

$$C_5^1(.9)^4(.1)^1 = .32805$$

probability that 3 errors occurred:

$$C_5^3(.9)^2(.1)^3 = .0081$$

is 400 times less so it will be most obvious to decode it the way we did.

Definition 2.4. The distance of a code is the minimum of the distances between all codewords.

$$d(C) = \{d(\mathtt{x}, \mathtt{y}) | \mathtt{x}, \mathtt{y} \in C\}$$

The parameters of a code are formed by n the number of symbols used for a code word, $|C|$ the size of the code an d it's minimal distance. in This case we speak of an $(n, |C|, d)$-code

One can easily compute that in the example we are using $d(C) = 3$. If a code has distance n it can detect every possible $n-1$ error-pattern. If we change at most $n-1$ bits the received message can never be again a codeword because every codeword is at a distance at least n of the transmitted codeword.

If $d = 2n + 1$ then the code can correct every possible n-error pattern. If we change at most n bits the Hamming distance between the received message $\mathtt{r}$ and the transmitted codeword $\mathtt{c}$ will be equal or less than n. Suppose now $\mathtt{x}$ is any other code word then

$$\begin{aligned} d(\mathtt{c},\mathtt{x}) &\leq d(\mathtt{c},\mathtt{r}) + d(\mathtt{r},\mathtt{x}) \\ -d(\mathtt{c},\mathtt{r}) + d(\mathtt{c},\mathtt{x}) &\leq +d(\mathtt{r},\mathtt{x}) \\ 2n + 1 - n &\leq d(\mathtt{x},\mathtt{r}) \\ n + 1 &\leq d(\mathtt{x},\mathtt{r}) \end{aligned}$$

so the original codeword will still be the codeword closest to the received message.

That the concept of error correcting codes is indeed very useful to transmit messages over a noisy channel can be shown by a little computation. Suppose again that the error rate is 10% and that we have to transmit messages, each chosen out of 4 possibilities. if we use the simplest code

$$C = \{\mathtt{00}, \mathtt{11}, \mathtt{01}, \mathtt{10}\}$$

the probability that one receives the correct message is here.

$$.9^2 = 81\%$$

If we use the code

$$C := \{\mathtt{00000},\ \mathtt{11110},\ \mathtt{01011},\ \mathtt{10101}\}$$

the probability of finding the correct message will be

$$.9^5 + C_5^1.9^4.1 = 91\%.$$

So the error rate is reduced by 50%.

Miniature 2: Richard W. Hamming

Richard W. Hamming received the 1968 Turing Award for his work on error-correcting codes at AT& T Bell Laboratories, where he worked for 30 years on a multitude of research projects. He joined the Naval Postgraduate School in Monterey, California, in 1976 as Adjunct Professor, where he is currently involved in teaching and writing books on probability and combinatorics. Hamming is a scientific generalist whose aims have included to teach his students an attitude of excellence toward science, not just technical skills.

2.2 Linear codes

In order to use the algebraical methods we introduced in the previous chapter, we will now define the notion of a linear code.

Definition 2.5. A *linear binary* $[n, m, d]$*-code* C over the finite field $\mathbb{F}_q$ is an m-dimensional subspace of $\mathbb{F}_q^n$, which has a minimal distance d. By subspace we mean that

$$\forall \mathtt{x}, \mathtt{y} \in C : \forall a, b \in \mathbb{F}_q : a\mathtt{x} + b\mathtt{y} \in C.$$

The fact of being m-dimensional makes that C contains q^m code words.

Example 2.3.

- $C := \{\mathtt{000}, \mathtt{110}, \mathtt{011}, \mathtt{101}\}$ is a binary linear $[3, 2, 1]$-code.
- $C := \{\mathtt{00000},\ \mathtt{11110},\ \mathtt{01011},\ \mathtt{10101}\}$ is a binary linear $[5, 2, 3]$-code.
- $C := \{\mathtt{0000},\ \mathtt{1110},\ \mathtt{0101},\ \mathtt{1001}\}$ isn't linear because $\mathtt{1110} + \mathtt{0101} = \mathtt{1011} \notin C$.
- $C := \{\mathtt{00}, \mathtt{1}\xi, \xi\xi^{\mathtt{2}}, \xi^{\mathtt{2}}\mathtt{1}\}$ is a linear $[2, 1, 2]$-code over $\mathbb{F}_4$.

Definition 2.6. The weight w of a codeword is the total number of non-zero elements in it's sequence:

$$w(\mathtt{01101101}) = 5$$

In a linear code C the minimal distance is also the least weight of the codewords in C because if $d = d(\mathtt{x}, \mathtt{y})$ then $\mathtt{x} - \mathtt{y}$ will be nonzero at exactly those places where $\mathtt{x}$ and $\mathtt{y}$ differ.

$$\forall \mathtt{x}, \mathtt{y} \in C : d(\mathtt{x}, \mathtt{y}) = w(\mathtt{x} - \mathtt{y}).$$

Because $\mathtt{x} - \mathtt{y}$ is in C there will be a vector which has weight equal to the minimal distance. Vice versa we have that $d(\mathtt{o}, \mathtt{x}) = w(x)$ so the minimal weight will be the minimal distance.

On the vectorspace C we can also define a scalar product $\cdot$. This maps every pair of codewords onto $\mathbb{F}_2$ in a bilinear way

Definition 2.7.

$$\forall \mathtt{x} := \mathtt{x}_1 \cdots \mathtt{x}_n, \mathtt{y} := \mathtt{y}_1 \cdots \mathtt{y}_n \in \mathbb{F}_q^n : \mathtt{x} \cdot \mathtt{y} = \mathtt{x}_1\mathtt{y}_1 + \cdots + \mathtt{x}_n\mathtt{y}_n \in \mathbb{F}_q$$

two sequences whose scalar product is 0 are said to be orthogonal. The orthogonal complement of a subset $S \subset \mathbb{F}_q^n$ is the set of all sequences that are orthogonal to each sequence in S:

$$S^\perp := \{\mathtt{x} \in \mathbb{F}_q^n | \forall \mathtt{y} \in S : \mathtt{x} \cdot \mathtt{y} = 0\}$$

Example 2.4. Here we first compute the scalar product of two sequences in $\mathbb{F}_2$.

$$\begin{aligned} \mathtt{11001} \cdot \mathtt{01101} &= 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 + 0 \cdot 0 + 1 \cdot 1 \\ &= 0 + 1 + 0 + 0 + 1 = 0 \end{aligned}$$

We can also compute the orthogonal complement of the code

$$\begin{aligned} C &:= \{\mathtt{00000},\ \mathtt{11110},\ \mathtt{01011},\ \mathtt{10101}\} \\ C^\perp &:= \{\mathtt{00000},\ \mathtt{11110},\ \mathtt{01010},\ \mathtt{10100}, \\ &\qquad \mathtt{11001},\ \mathtt{00111},\ \mathtt{10011},\ \mathtt{11001},\ \mathtt{01101}\} \end{aligned}$$

Which is a again a linear code because of the distributivity of the scalar product. this code is called the orthogonal code.

2.3 Basis and generator matrix

We will here review the concept of linearly dependence and see how it relates to linear codes. If we consider some codewords $\mathtt{x_1}, \cdots, \mathtt{x_k} \in C$ then they are said to be linear independent if there exists no numbers $a_i \in \mathbb{F}_2$ such that

$$a_1\mathtt{x_1} + \cdots + a_k\mathtt{x_k} = 0$$

and not all the a_i are zero. Otherwise $\{\mathtt{x_i}\}$ is said to be linear dependent.

Example 2.5.

$$\mathtt{0101}, \mathtt{0110}, \mathtt{0011}$$

are linear independent because

$$a_1\mathtt{0101} + a_2\mathtt{0110} + a_3\mathtt{0101} = (0, a_1 + a_2, a_2, a_3) = \mathtt{0000}$$

implies that $a_1 = a_2 = a_3 = 0$. On the other hand is

$$\mathtt{0111}, \mathtt{1111}, \mathtt{1110}, \mathtt{1001}$$

linear dependent because

$$\mathtt{0111} + \mathtt{1110} + \mathtt{1001} = \mathtt{0000}$$

A subset $S \subset C$ is said to be generating if all codewords in C can be expressed as a linear combination of codewords in S. S is said to be a base if it is both linear independent and generating. If this is the case any codeword in C can be written as *unique* combination of base elements.

Example 2.6. If we consider the binary code

$$C := \{\mathtt{00000},\ \mathtt{11110},\ \mathtt{01011},\ \mathtt{10101}\}$$

then there are three possible bases containing all 2 elements

$$B_1 := \{\mathtt{11110}, \mathtt{01011}\}$$
$$B_2 := \{\mathtt{10101}, \mathtt{01011}\}$$
$$B_3 := \{\mathtt{11110}, \mathtt{10101}\}$$

A very easy way to handle vector spaces is using matrices. matrices over binary fields multiply just the way they do over the real numbers, except that you must use the calculation rules of the field (f.i. in $\mathbb{F}_2$ $1 + 1 = 0$).

Example 2.7.

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1\cdot 1 + 0\cdot 1 + 1\cdot 1 & 1\cdot 0 + 0\cdot 1 + 1\cdot 1 \\ 1\cdot 1 + 1\cdot 1 + 0\cdot 0 & 1\cdot 0 + 1\cdot 1 + 0\cdot 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

Calculation the determinant of a matrix is also just in the ordinary way (remember that we can replace all the minus signs by plus signs).

$$\mathrm{Det} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = 1\cdot 1 + 1\cdot 0 = 1$$

Definition 2.8. When we have an m-dimensional code C with a base in it then we can make a $m \times n$-matrix. The rows of this matrix consists of the base-vectors of the considered base. This matrix is called a *generator matrix*.

We know that there are q^m code words so it is reasonable to consider the set of messages equal to the space $\mathbb{F}_q^m$ Encoding the message means that we assign to each message word a unique codeword. This can be done easily by considering the message as a row-matrix and multiplying it with the generator matrix. The encoding depends highly on the generator matrix, different generator matrices give different encodings but use the same codewords.

In fact if we multiply the generator matrix on the left by an invertible $m \times m$-matrix we will obtain a new generator matrix because multiplying on the left corresponds to a base change in the C.

One could also try to multiply on the right with an invertible matrix but doing this will alter the corresponding code, such that the decoding capacities will alter as well. However if we multiply on the right by a permutation matrix (i.e. a matrix which has in every row and every column exactly one 1 and all the other entries zero) this will correspond to a permutation of the columns of the generator matrix. This means that we just considered a new code by switching some of the bits around. This operation will not alter the minimal distance of the code, we call this code an *equivalent code*.

Example 2.8. If we consider the code

$$C := \{\texttt{00000},\ \texttt{11110},\ \texttt{01011},\ \texttt{10101}\}$$

then there are 6 possible generator matrices two for each base

$$B_1 \mapsto \begin{pmatrix}1&1&1&1&0\\0&1&0&1&1\end{pmatrix}, \begin{pmatrix}0&1&0&1&1\\1&1&1&1&0\end{pmatrix}$$

$$B_2 \mapsto \begin{pmatrix}1&0&1&0&1\\0&1&0&1&1\end{pmatrix}, \begin{pmatrix}0&1&0&1&1\\1&0&1&0&1\end{pmatrix}$$

$$B_3 \mapsto \begin{pmatrix}1&0&1&0&1\\1&1&1&1&0\end{pmatrix}, \begin{pmatrix}1&1&1&1&0\\1&0&1&0&1\end{pmatrix}$$

And if we take an invertible 2×2-matrix and multiply it with one of these generator matrices

$$\begin{pmatrix}1&1\\1&0\end{pmatrix}\begin{pmatrix}1&1&1&1&0\\0&1&0&1&1\end{pmatrix} = \begin{pmatrix}1&0&1&0&1\\1&1&1&1&0\end{pmatrix}$$

we obtain again one of the six generator-matrices. However is we multiply on the left with a permutation matrix

$$\begin{pmatrix}1&1&1&1&0\\0&1&0&1&1\end{pmatrix}\begin{pmatrix}0&1&0&0&0\\0&0&0&1&0\\1&0&0&0&0\\0&0&0&0&1\\0&0&1&0&0\end{pmatrix} = \begin{pmatrix}1&1&0&1&1\\0&0&1&1&1\end{pmatrix}$$

We obtain a generator matrix of a different code

$$C' := \{\texttt{00000},\ \texttt{11011},\ \texttt{00111},\ \texttt{11100}\}$$

This code is in fact equivalent to C and has thus the same mimimal distance: 3

Definition 2.9. A generator matrix G whose left part is the identity matrix, i.e. $G = [1_m X]$, is said to be in *normal position*. For the code we considered in the previous example, is

$$\begin{pmatrix}1&0&1&0&1\\0&1&0&1&1\end{pmatrix}$$

a generator matrix in normal position. A code that admits a such generator matrix is called a *systematic code*.

Theorem 2.1. Every code is equivalent to a systematic code.

2.4 The parity check matrix

When we have a code C then we can also consider it's orthogonal complement $C^\perp$. $C^\perp$ is again a code, so we can consider a generator matrix of this code. If we call G the generator matrix of C and H a generator matrix of $C^\perp$ then the following identity holds:

$$GH^t = HG^t = 0$$

where $\cdot^t$ stands for the transposed of a matrix. We will call H the parity check matrix of the code C. If G is a generator matrix is in standard form $[I_m X]$ we can also choose a special form of the parity check matrix

$$H^t = \begin{pmatrix} -X \\ 1_{n-m} \end{pmatrix} \text{ because } \begin{pmatrix} 1_m & X \end{pmatrix} \begin{pmatrix} X \\ 1_{n-m} \end{pmatrix} = (X - X) = 0$$

The parity check matrix is a very useful tool in decoding the code. Because H is a generator matrix of the orthogonal code, the following holds

$$\mathtt{c} \in C \;\Leftrightarrow\; H\mathtt{c}^t = \mathtt{o}$$

Definition 2.10. For a random word $\mathtt{x} \in \mathbb{F}_2^n$ we define its *syndrome* as

$$\mathtt{s}(\mathtt{x}) := H\mathtt{x}^t \in \mathbb{F}_q^{n-m}.$$

The codewords are the words with zero syndrome.

Suppose we receive a word $\mathtt{r}$ then we can assume that it is of the form $\mathtt{c} + \mathtt{e}$ where $\mathtt{c}$ is the transmitted codeword, and $\mathtt{e}$ is the error vector. So to remove the errors from $\mathtt{r}$ we have to find $\mathtt{e}$. If we look at the syndrome of $\mathtt{r}$ we see that it only depends on the error vector and not on the codeword. So instead of looking at the received word we should concentrate on the syndromes.

One should make a list of all possible syndromes and associate to each one the error vector with the least weight producing that syndrome, and thus the most likely to have occurred. It is not always true that there is such a unique error vector, more than one error vector can have the same weight and syndrome. If this is the case we denote this also in the list.

Algorithm 2.1 (Encoding and decoding a linear code)**.** Suppose that C is a systematic $[n, m, d]$-code and G is a generator matrix in normal position with H the corresponding parity check matrix.

Encoding

Take the message word $\mathtt{m} \in \mathbb{F}_2^m$ and multiply it with G and transmit it.

Decoding

1. Make a list of syndromes and corresponding error vectors
2. Compute the syndrome of the received codeword $\mathtt{s} := H\mathtt{r}^t$.
3. Look in the list of syndromes which error vector $\mathtt{e}$ corresponds to $\mathtt{s}$. If there is only one *code*, subtract it from $\mathtt{r}$. The message word $\mathtt{m}$ is most likely to be the first m bits of $\mathtt{r} + \mathtt{e}$. If there are more error vectors, all those give equally probable message words, so if possible one should better ask for retransmission.

Example 2.9. Consider the $[5, 2, 3]$-code with generator matrix

$$G := \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

The possible messages and there codewords are then

message	code word
00	00000
10	10110
01	01011
11	11101

The parity check matrix is

$$H := \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

The list of possible syndromes and error vectors:

Syndrome	error vectors
000	00000
100	00100
010	00010
110	10000
001	00001
101	11000, 00101
011	01000
111	10001, 01100

Suppose we receive 11011, computing the syndrome gives s := 110 so the error vector is 10000 and the corrected code word is 01011 and the message was 01.

Suppose we receive 00111, computing the syndrome gives s := 111. The are two equally probable error vector, so the code word could either be 10110 or 01011.

Chapter 3

Perfect codes

3.1 Introduction

One of the main challenges of coding theory is to find the best possible codes, but what are in fact the criteria for a good code? One wants to transmit as fast as possible, as many as possible information over a channel, such that there occur as few as possible mistakes.

So for a linear $[n, m, d]$-code over a field $\mathbb{F}_q$ one wants to increase both the ratio m/n and d/n. As is expected one can not improve them both as much as one wants, because there are inequalities that are satisfied between those parameters.

One of those inequalities is called the *sphere packing boundary*. If the minimum distance between two code words is d then one can draw around each codeword $\mathtt{c}$ in $\mathbb{F}_q^n$ a sphere with radius $(d-1)/2$.

$$B_{\mathtt{c}} := \{\mathtt{x} \in \mathbb{F}_q^n | d(\mathtt{x}, \mathtt{c}) \leq \frac{d-1}{2}\}$$

All those spheres are non-intersecting because otherwise there are two code words at a distance less than d which is impossible. In each of these spheres

there are exactly

$$\sum_{i=0}^{\frac{d-1}{2}} C_i^n (q-1)^i$$

words. This sum corresponds to the number of words with $0, 1, \ldots, (d-1)/2$ errors. For each sphere we have different elements and there are q^m spheres so

$$\sum_{i=0}^{\frac{d-1}{2}} C_i^n (q-1)^i \leq q^{n-m}.$$

In the ideal situation this inequality would be an identity.

Definition 3.1. An $(n, |C|, d)$-code over $\mathcal{F}$ is perfect if the all the B_c cover whole $\mathcal{F}^n$. For a linear $[n, m, d]$-linear code over $\mathbb{F}_q$ this implies that

$$\sum_{i=0}^{\frac{d-1}{2}} C_i^n (q-1)^i = q^{n-m}.$$

In the previous chapter we've seen how to decode a linear code by using a table of syndromes. To every syndrome corresponded one or more error vectors. When we had a syndrome with more than one error vector, we could not uniquely decode the received message.

In the case of perfect codes this last situation does not occur. In a perfect code the number of possible error vectors is equal to q^{n-m} which is the same as the number of possible syndromes, so there is a bijection between them.

As we've seen perfect codes are in many ways very good, but unfortunately these codes are very rare. In what follows we will describe all perfect linear codes.

First notice that a code can never be perfect if its minimal distance is even. If this were, there would exist words that are at distance $d/2$ of two code words. so the syndrome decoding can't be unique.

A trivial class of linear codes are the binary repetition codes

Definition 3.2. A binary repetion code is a code consisting of two code words

$$C := \{\mathtt{0} \ldots \mathtt{0}, \mathtt{1} \cdots \mathtt{1}\}$$

A repetition code has as parameters $[n, 1, n]$ and so it is easy to check whether such a code is perfect.

Theorem 3.1. a binary repetition code is perfect if and only if n is odd.

Proof. We already know that n has to be odd so $n = 2t + 1$. We can use the binomium of newton

$$\begin{aligned}\sum_{i=0}^{t} C_n^i (2-1)^i &= \sum_{i=0}^{t} \frac{1}{2}(C_i^n + C_{n-i}^n)(1)^i \\ &= \frac{1}{2}\sum_{i=0}^{n} C_i^n = 2^{n-1}.\end{aligned}$$

□

3.2 Hamming codes

In this section, we consider an important family of perfect codes which are easy to encode and decode.

We first start with binary hamming codes and afterward we define them for arbitrary finite fields.

Definition 3.3. A binary code is a *Hamming code* if it has a parity check matrix $H \in \mathsf{Mat}_{r\times 2^r-1}(\mathbb{F}_2)$ consisting of all possible column vectors of $\mathbb{F}_2^r$.

$$\begin{pmatrix} v_1 & \dots & v_n \end{pmatrix},\ v_i \in \mathbb{F}_2^r - \{0\}$$

We denote this Hamming code as $\mathtt{Ham}(r, 2)$.

Example 3.1. if r= 2 this means that

$$H := \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \Rightarrow G := \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}.$$

So $\mathtt{Ham}(2, 2)$ is the binary repetition [3,1,3]-code. Notice that this is the only hamming code that is a repetition code because $d = 3$ for a hamming code and $d = n$ for a repetition code.

Example 3.2. if r= 3 this means we can put H in standard form like

$$H := \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

This gives us as generator matrix:

$$G := \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}.$$

Theorem 3.2. $\mathtt{Ham}(r, 2)$ is a perfect $[2^r - 1, 2^r - r - 1, 3]$-code.

Proof. By the dimension of the parity check matrix we know that $n = 2^r - 1$ and $m = 2^r - 1 - r$. We now have to prove that the minimal weight of a codeword is 3. Suppose thus $\mathtt{c}$ is a codeword with weight one or two. Than we should have that $H\mathtt{c}^t = \mathtt{o}$. but $H\mathtt{c}^t$ is the sum of maximum 2 column vectors of H so this should mean that two columns of H are linearly dependent. This is impossible because two different vectors over $\mathbb{F}_2$ are always linearly independent. But the minimal weight is also equal to 3. H contains the columns $\mathtt{100\ldots0}^t$,$\mathtt{010\ldots0}^t$ and $\mathtt{110\ldots0}^t$ at the i, j, k^{th} position. construct the word $\mathtt{c}$ containing a 1 at those 3 places and zero's everywhere else this is a codeword because

$$cH^t = \mathtt{100\ldots0} + \mathtt{010\ldots0} + \mathtt{110\ldots0} = \mathtt{o}.$$

Finally we prove that $\mathtt{Ham}(r, 2)$ is perfect.

$$\sum_{i=0}^{1} C_n^i = 1 + C_1^{2^r-1} = 2^r = 2^{2^r-1-(2^r-r-1)} = 2^{n-m}$$

□

Encoding the Hamming code is done by using the generator matrix. Because $d = 3$ the code is one error correcting and the possible error vectors are of the form $\mathtt{e_j} = \mathtt{0\cdots010\cdots0}$ where the one is at position j. The syndrome of $\mathtt{e_j}$ is equal to the j^{th} column of H. To decode we proceed like this:

Algorithm 3.1 (Decoding the binary Hamming code)**.** Suppose we receive the word $\mathtt{r}$ and G and H are in standard position

1. calculate the syndrome $\mathtt{s} = H\mathtt{r}^t$
2. If $\mathtt{s} = \mathtt{o}$ the original code word was $\mathtt{r}$ and the message words are the first $2^r - r - 1$ symbols of $\mathtt{r}$.
3. If $\mathtt{s} \neq \mathtt{o}$ we suppose that one error has occurred. the position of this error is the position of the column vector of H equal to $\mathtt{s}$.

One can generalize binary Hamming codes to Hamming codes over arbitrary fields. However one must take care in the construction of the parity check matrix. One cannot take all possible vectors of $\mathbb{F}_q^r$ because then are two of them that can be linear dependent of each other (take $v \in F_q^r \setminus \{0\}$ and kv where $k \in \mathbb{F}_q \setminus \{0, 1\}$). So one has to take for each ray of vectors only one representative in H. the number of such rays is equal to the points of the $r-1$-dimensional projective space over $\mathbb{F}_q$ which is $\frac{q^r-1}{q-1}$.

Example 3.3. if r= 2 and $q = 3$ this means that

$$H := \begin{pmatrix} 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{pmatrix} \Rightarrow G := \begin{pmatrix} 1 & 0 & 2 & 1 \\ 0 & 1 & 2 & 2 \end{pmatrix}.$$

To compute G we use the fact that $-1 = 2 \mod 3$. So $\mathtt{Ham}(2, 3)$ is a [4,2,3]-code.

Example 3.4. if r= 2 and $q = 4$ this means that

$$H := \begin{pmatrix} 1 & 1 & 1 & 1 & 0 \\ \xi^2 & \xi & 1 & 0 & 1 \end{pmatrix} \Rightarrow G := \begin{pmatrix} 1 & 0 & 0 & 1 & \xi^2 \\ 0 & 1 & 0 & 1 & \xi \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

So $\mathtt{Ham}(2, 4)$ is a [5,3,3]-code.

Example 3.5. if r= 3 and $q = 3$ we can put H in standard form like

$$H := \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 & 0 & 1 & 0 \\ 1 & 2 & 1 & 2 & 0 & 1 & 2 & 0 & 1 & 2 & 0 & 0 & 1 \end{pmatrix}$$

Just like the binary codes we have this theorem

Theorem 3.3. $\texttt{Ham}(r,q)$ is a perfect $[\frac{q^r-1}{q-1}, \frac{q^r-1}{q-1} - r, 3]$-code over $\mathbb{F}_q$

Exercise 3.1. Prove the theorem above

Exercise 3.2. Design a decoding algorithm for non-binary hamming codes.

Exercise 3.3. Write source code to encode and decode both binary and non-binary Hamming codes.

3.3 The ternary Golay-code

Hamming-codes form an infinite series of codes that are perfect, apart from these codes there are also a limited number of special linear codes. These codes were discovered by Marcel Golay and hence they are called Golay codes.

First we construct the ternary Golay code. Consider the field $\mathbb{F}_3$ and take the matrix

$$S_5 := \begin{pmatrix} 0 & 1 & 2 & 2 & 1 \\ 1 & 2 & 2 & 1 & 0 \\ 2 & 2 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 2 \\ 1 & 0 & 1 & 2 & 2 \end{pmatrix}$$

Definition 3.4. The ternary Golay code $\texttt{Gol}(11,3)$ is a $[11,6]$ code over $\mathbb{F}_3$ with generator matrix

$$G := \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & & & & & & & & & & \\ 0 & & & & & & & & & & \\ 0 & & & \texttt{1}_5 & & & & & S_5 & & \\ 0 & & & & & & & & & & \\ 0 & & & & & & & & & & \end{pmatrix}$$

and parity check matrix

$$\begin{pmatrix} 2 & 0 & 2 & 1 & 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 2 & 2 & 0 & 2 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 2 & 1 & 2 & 0 & 2 & 1 & 0 & 0 & 1 & 0 & 0 \\ 2 & 1 & 1 & 2 & 0 & 2 & 0 & 0 & 0 & 1 & 0 \\ 2 & 2 & 1 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Theorem 3.4. The ternary golay code is a perfect 2 error correcting code.

Proof. One can compute that

$$GG^t = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 2 & 2 & 2 & 2 \\ 0 & 2 & 2 & 2 & 2 & 2 \\ 0 & 2 & 2 & 2 & 2 & 2 \\ 0 & 2 & 2 & 2 & 2 & 2 \\ 0 & 2 & 2 & 2 & 2 & 2 \end{pmatrix}$$

So if $\mathtt{x} := \mathtt{x_1} \cdots \mathtt{x_6}$ is a message word $\mathtt{x}G$ will be a code word and hence

$$\begin{aligned} \mathtt{x}G \cdot \mathtt{x}G &= \mathtt{x}GG^t\mathtt{x}^t \\ &= 2\left(\sum_{i=2}^{6} \mathtt{x_i}\right)^2. \end{aligned}$$

Because 1 is the only non zero square in $\mathbb{F}_3$, $\mathtt{x}G \cdot \mathtt{x}G \neq 1$. But in $\mathbb{F}_3$ $\mathtt{x}G \cdot \mathtt{x}G$ is equal to the weight modulo 3. So the weight cannot be $1, 4, 7$ or 10.

The weight of a codeword $\mathtt{x}G$ is always at least the weight of $\mathtt{x}$ because the first 6 symbols of $\mathtt{x}G$ are $\mathtt{x}$.

If $\mathtt{x}$ has weight 1, $\mathtt{x}G$ has weight at least 5 because all rows of G have weights bigger than 4. Suppose that $\mathtt{x}$ has weight 2 then $\mathtt{x}G$ has weight bigger than 4. Take two rows G_i and G_j of G and look at the quotients of the last five entries: $G_{i6}/G_{j6}, \cdots, G_{i11}/G_{j11}$. There are at least 3 different quotients. Every linear combination will have at least 2 nonzero symbols in the last 5 digits, so the weight is at least 4 and hence 5 or more.

if $\mathtt{x}$ has weight 3 one can do a similar thing using the fact that the submatrix consisting of the last five digits of 3 rows of G has rank 3.

Now we've proved that the minimal weight is 5, we only have to check the sphere packing identity:

$$\sum_{i=0}^{2} C_i^{11} 2^i = 1 + 2C_1^{11} + 4C_2^{11}$$

$$1 + 22 + 220 = 243 = 3^5 = 3^{11-6}$$

□

3.4 Binary Golay-codes

In this section we describe both the extended binary Golay code and the perfect binary Golay code, because they are both of practical importance.

Let B be the 12×12 matrix over $\mathbb{F}_2$

$$B := \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

This matrix is symmetric and if one looks at the submatrix consisting op the first 11 rows and columns one can easily check that the rows are cyclic permutations of each other. another property of B is that $B^2 = \mathtt{1}_{12}$.

Definition 3.5. The extended Golay-code $\mathtt{Gol}(24, 2)$ is a binary code with generator matrix

$$G := [\mathtt{1}_{12} B]$$

Because B is its own inverse, not only $[B\mathtt{1}_{12}]$ is a parity check matrix but G itself as well. Also $[B\mathtt{1}_{12}]$ is a generator matrix for $\mathtt{Gol}(24, 2)$.

Theorem 3.5. The minimal weight of $\mathtt{Gol}(24, 2)$ is 8.

Proof. First we prove that the weight of a random codeword is a multiple of 4. Because $GG^t = 0$ the inproduct of two code words is zero. This means that the number of entries where they are both 1 is even. Suppose now that $\mathtt{x}$ and $\mathtt{y}$ are two codewords of which the weight is a multiple of four then

$$w(\mathtt{x} + \mathtt{y}) = w(\mathtt{x}) + w(\mathtt{y}) - 2(\#\text{common } 1's \text{ between } \mathtt{x} \text{ and } \mathtt{y})$$

This expression is again a multiple of four because all its terms are. Notice that the rows of G all have weight 8 or weight 12, so every code word has weight $4k,\ k \in \mathbb{N}$.

Now we prove that no code word can have weight four. Suppose $\mathtt{x}G = \mathtt{xx}B$ is a code word of weight four. Because $[B\mathtt{1}_{12}]$ is also a generator matrix, there exists a $\mathtt{y}$ such that

$$\mathtt{x}G = \mathtt{xx}B = \mathtt{y}[B\mathtt{1}_{12}] = \mathtt{y}B\mathtt{y} = \mathtt{xy}$$

Therefore either $\mathtt{x}$ or $\mathtt{y}$ must have a weight smaller or equal than 2. If this is the case for $\mathtt{x}$ we know that $\mathtt{x}G$ is the sum of at most two rows of G and can never have weight equal to 4. For $w(\mathtt{y}) \leq 2$ we proceed the same. □

We will now search for a decoding algorithm for $\mathtt{Gol}(24, 2)$. Because the minimal weight is 8 we will be able to correct all error vectors with weight smaller than 4. Take $H := G$ to be the parity check matrix and suppose we have an error vector $\mathtt{e}$ with weight at most 3. We will split up our error vector in two parts of length 12 $\mathtt{e} := [\mathtt{e}_1, \mathtt{e}_2]$. the syndrome of such and error vector is

$$\mathtt{s} = G\mathtt{e}^t = {\mathtt{e}_1}^t + B{\mathtt{e}_2}^t$$

Because the weight of $\mathtt{e}$ is smaller than 4 either $\mathtt{e}_1$ or $\mathtt{e}_2$ has weight smaller than 2

Suppose first that the weight of $\mathtt{e}_2$ is at most 1, then $\mathtt{s}$ the syndrome consist of either a word of weight at most 3 (if $\mathtt{e}_2 = \mathtt{o}$) or a row of B with at most two digits changed.

If the weight of $\mathtt{e}_1$ is at most 1 then one can do the same but now using the syndrome

$$\mathtt{t} = [B\mathtt{1}_{12}]\mathtt{e}^t = B{\mathtt{e}_1}^t + {\mathtt{e}_2}^t = B\mathtt{s}$$

Algorithm 3.2 (Decoding the extended Golay code)**.** We receive the word $\mathtt{r}$.

1. Compute the syndrome $\mathtt{s} = G\mathtt{r}^t$
2. If $w(\mathtt{s}) < 3$ then $\mathtt{e} := [\mathtt{s}, \mathtt{o}]$, stop.

3. If $w(\mathtt{s} + B_i) < 3$ then $\mathtt{e} := [\mathtt{s} + B_i, \delta_i]$ where δ_i stands for the vector with everywhere zero's except on the i^{th} place a $\mathtt{1}$. Stop.

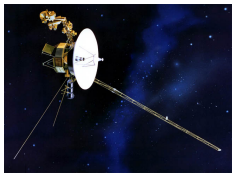
4. Compute $\mathtt{t} := B\mathtt{s}$.

5. If $w(\mathtt{t}) < 3$ then $\mathtt{e} := [\mathtt{o}, \mathtt{t}]$, stop.

6. If $w(\mathtt{t} + B_i) < 3$ then $\mathtt{e} := [\delta_i, \mathtt{t} + B_i]$, stop.

7. If $\mathtt{e}$ is not determined request retransmission.

Miniature 3: the voyager mission (1811-32)



In the late seventies NASA set up a mission to explore the outer planets of the solar system. This mission visited Jupiter, Saturn, Uranus and Neptune. The images of those planets and their moons had to be transmitted over several billions of kilometers. In order to achieve good quality NASA used the binary extended Golay code for encoding the photographs.

The last code we will see in this chapter is the binary Golay code.

Definition 3.6. The binary Golay code $\mathtt{Gol}(23, 2)$ has as generator matrix, the generator matrix of $\mathtt{Gol}(24, 2)$ except for the last column which is omitted. Because the minimal weight of $\mathtt{Gol}(24, 2)$ is 8 the minimal weight of this code will be 7.

Theorem 3.6. $\mathtt{Gol}(23, 2)$ is a perfect $[23, 12, 7]$-code.

Proof. We only have to calculate the sphere packing identity.

$$\sum_{i=0}^{3} C_i^{23} 2^i = 1 + C_1^{23} + C_2^{23} + C_3^{23}$$

$$1 + 23 + 253 + 1771 = 2048 = 3^{23-12}$$

□

How do we decode $\mathtt{Gol}(23, 2)$? We already have a decoding algorithm for $\mathtt{Gol}(24, 2)$ so we can use this. suppose we receive $\mathtt{r}$, we have to transform it into a word of 24 bit. We know that both $\mathtt{Gol}(23, 2)$ and $\mathtt{Gol}(24, 2)$ can

correct 3 errors. Because every error changes the weight of a word by 1 and codewords in $\texttt{Gol}(24, 2)$ have even weight, message words with an odd weight will contain an odd number of errors.

If r contains at most 3 errors we want to add one bit in order to have a message word for $\texttt{Gol}(24, 2)$. We do this by adding a 0 or a 1 such that $w(\texttt{r1})$ or $w(\texttt{r0})$ is odd. Call this new word $\texttt{r}'$. Because $\texttt{r}'$ has odd weight it contains an odd number of errors and we know that it contained at most 3 errors in the first 23 digits, so it contains as a whole also at most 3 errors. Knowing this we can decode r' by algorithm **??**.

In practice, the received word is almost always a code word, however $\texttt{r}'$ is never a codeword. In that case computing the syndrome will give us the last row of `B`. It is useful to check this at the start of the algorithm rather than to wait until step 3.

3.5 Fundamental theorems

Have we in fact considered all possible perfect codes or are there other ones? In the case of linear codes this is actually the case, but one can also define perfect non-linear codes. In general van Lint (and others) proved the following theorem

Theorem 3.7 (van Lint-Tietäväinen)**.** Every non-trivial perfect code over q symbols where q is a prime power, has the parameters of a Hamming or a Golay code ($\texttt{Gol}(23, 2)$ or $\texttt{Gol}(11, 3)$.

In 1975 Delsarte and Goethals proved that every code with the parameters of a Golay code is in fact a Golay code. On the other hand one can prove easily that every linear code with the parameters of a Hamming code is a Hamming code.

Theorem 3.8 (van Lint-Tietäväinen)**.** Every non-trivial perfect linear code over $\mathbb{F}_q$ is a Hamming or a Golay code. ($\texttt{Gol}(23, 2)$ or $\texttt{Gol}(11, 3)$.

There exist however perfect non-linear codes that have the parameters of a Hamming code. Those were constructed by Schönheim and Lindström in

1968-69. It remains an open question whether there exist perfect codes over alphabets where the number of symbols is not a prime power.

Chapter 4

Cyclic Codes

4.1 Introduction

In the previous chapter we saw examples of linear codes, that are practical to use and have good properties, but because these codes are, there one has little flexibility in choosing appropriate parameters for the code you need. In this chapter we will examine a vast variety of codes that are easy to construct and adapt to different situations.

Definition 4.1. A linear $[n, m]$-code C is called cyclic if and only if for every word $\mathtt{x} := \mathtt{x_1} \cdots \mathtt{x_n}$

$$\mathtt{x_1} \cdots \mathtt{x_n} \in C \Rightarrow \overrightarrow{\mathtt{x}} := \mathtt{x_2} \cdots \mathtt{x_n} \mathtt{x_1} \in C.$$

So every cyclic shift of a codeword is again a codeword.

This property seems very strict but in fact there are many codes that are cyclic.

Example 4.1. The hamming code $\mathtt{Ham}(3, 2)$ as we have seen it in the previous chapter is not cyclic but it is equivalent to a cyclic code if we take another parity check and generator matrix. Take

$$H := \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

One sees that the rows of this matrix are cyclic permutations of each other. So the code with H as generator matrix is cyclic.

Because $\overrightarrow{\mathtt{x}} \cdot \overrightarrow{\mathtt{y}} = \mathtt{x} \cdot \mathtt{y}$ the orthogonal complement of a cyclic code is cyclic as well.

Because we introduced an extra property to the linear codes, we can put an extra structure onto the vector space of code words we're working with.

With every element of $\mathbb{F}_q^n$ one can associate a polynomial of degree at most $n-1$ like this

$$\mathbb{F}_q^n \to \mathbb{F}_q[X] : \mathtt{c_0} \cdots \mathtt{c_{n-1}} \mapsto c_0 + c_1 X + \cdots + c_{n-1} X^{n-1}.$$

If we multiply a such a polynomial with X we see that all the coefficients shift one position to the right. However because c_{n-1} also shifts one to the right and not to the first position, the new polynomial does not correspond anymore to a codeword. This problem is solved by identifying X^n with 1. This means that one has to work in the quotient ring

$$\mathbb{F}_q[X]/(X^n - 1)$$

rather than in $\mathbb{F}_q[X]$. In this ring multiplying by X corresponds to a cyclic shift of the coefficients. A cyclic code corresponds to a subset of $\mathbb{F}_q[X]/(X^n - 1)$ closed under addition and multiplication by X and hence multiplication by every element of $\mathbb{F}_q[X]/(X^n - 1)$. Such a subset is called an ideal of $\mathbb{F}_q[X]/(X^n - 1)$.

Theorem 4.1. There is a bijective correspondence between cyclic n-codes over $\mathbb{F}_q$ and ideals in $\mathbb{F}_q[X]/(X^n - 1)$.

4.2 Generator polynomial and check polynomial

For linear codes we had a generator matrix, in the case of cyclic codes one can prove the following:

Theorem 4.2. For every ideal (or cyclic code) $\mathfrak{c} \lhd \mathbb{F}_q[X]/(X^n-1)$ there exists a polynomial $g(X) \in \mathbb{F}_q[X]/(X^n-1)$ such that:

$$\mathfrak{c} := \{a(X)g(X) | a(X) \in \mathbb{F}_q[X]/(X^n-1)\}$$

Proof. Define $g(X)$ to be the greatest common divisor of all code polynomials. By the theorem of Euclid one can write $g(x)$ as a linear combination of code polynomials. Because an ideal $\mathfrak{c}$ is closed under linear combinations, $g(x)$ is again a code polynomial. Because $g(x)$ is the gcd it is also the codeword with the lowest degree. □

Writing $g(X) = g_0 + g_1 X + \cdots + g_k X^k$ we see that $g(X), Xg(X), \ldots, X^{n-k-1}g(X)$ form a basis for $\mathfrak{c}$ and hence

$$G = \begin{pmatrix} g_0 & g_1 & \cdots & g_k & & \\ & g_0 & g_1 & \cdots & g_k & \\ & & \ddots & \ddots & \cdots & \ddots \\ & & & g_0 & g_1 & \cdots & g_k \end{pmatrix}$$

is a generator matrix for the code.

Theorem 4.3. For a non-trivial cyclic code $\mathfrak{c} \lhd \mathbb{F}_q[X]/(X^n-1)$ the generator polynomial divides $X^n - 1$.

Proof. If $g(X)$ doesn't divide $X^n - 1$ then one can look at the rest of $X^n - 1$ divided by $g(X)$. This rest $r(X)$ has lower degree than $g(X)$ but it is a linear combination of $g(X)$ and $X^n - 1$ so it is again an element of the code. But this is impossible because $g(X)$ is the element of the code with the least degree. □

Definition 4.2. Define $h(X) := \frac{X^n-1}{g(X)}$ to be the check polynomial of $\mathfrak{c}$.

One can easily prove that $c(X)$ is a code polynomial if and only if

$$h(X)c(X) \equiv 0 \mod X^n - 1.$$

If we go back to the vector space representation we can model multiplication by $h(X)$ as matrix multiplication with

$$H = \begin{pmatrix} h_{n-k} & h_{n-k-1} & \cdots & h_0 & & & \\ & h_{n-k} & h_{n-k-1} & \cdots & h_0 & & \\ & & \ddots & \ddots & \cdots & \ddots & \\ & & & h_{n-k} & h_{n-k-1} & \cdots & h_0 \end{pmatrix}.$$

So this H is the parity check matrix for our code.

Example 4.2. Because the parity check matrix of the Hamming code in the previous example

$$H := \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

We can deduce that $h(X) := X^4 + X^2 + X + 1$ and $g(X) := (X^7 - 1)/h(X) = X^3 + X + 1$

An interesting way of encoding a cyclic code is systematic encoding. If we encode just by multiplying with $g(X)$ the code word will not contain the original message because of multiplication with $g(X)$. To remedy this problem one tries to construct a codeword of our code that contains the original message plus some check digits. this is done as follows:

Suppose $u(X)$ is the message we want to encode and that $n-k$ is the degree of $g(X)$. $X^{n-k}u(X)$ will then be the message shifted to the last k of the n digits of the codewords. However $X^{n-k}u(X)$ itself is not a code word. If we want to make a code word without destroying the digits from $u(X)$ one has to change the first $n-k$ digits. Because of the division algorithm we have the identity

$$\exists q(X), r(X) : \ X^{n-k}u(X) = q(X)g(X) + r(X).$$

This implies that $X^{n-k}u(X) - r(X)$ is a codeword satisfying our demands, because the two terms of the sum do not overlap.

Another way of calculating a syndrome for a received word $r(X)$ is looking at its rest if we divide it by the generator polynomial. This has indeed the properties of a syndrome polynomial because it is zero if and only if $r(X)$ is a multiple of $g(X)$ and hence a codeword. In what follows we will always use this syndrome to decode cyclic codes.

4.3 Decoding Cyclic Codes

In our study of linear codes, we saw that maximum likelihood decoding, using syndrome-error pattern tables was complex, requiring a list that is

exponentially long compared to the codeword length. The syndromes for cyclic codes have a special property that yields simpler decoder algorithms. In particular, as the next theorem shows, if we know the syndrome for a received vector, we can very easily calculate the syndrome for a cyclic shift of the received vector, using the the original syndrome. This will allow us to design a decoder that essentially corrects the received bits one by one.

Theorem 4.4 (Meggit)**.** If $s(X)$ is the syndrome of a received vector $r(X) = r_0 + \cdots + r_{n-1}X^{n-1}$, then the syndrome $s'(X)$ of $\overrightarrow{r}(X)$ is the remainder left after dividing $Xs(X)$ by $g(X)$

Proof. We can write $Xr(X)$ as follows

$$Xr(X) = r_{n-1}(X^n - 1) + \overrightarrow{r}(X).$$

Rearranging, and using the fact that $X_n - 1 = g(X)h(X)$, we get

$$\overrightarrow{r}(X) = Xr(X) - r_{n-1}g(X)h(X).$$

Now divide by $g(X)$, to expand each term into a quotient-remainder form.

$$c(X)g(X) + d(X) = X(a(X)g(X) + s(X)) - r_{n-1}g(X)h(X),$$

where the remainder $d(X)$ is the syndrome for $\overrightarrow{r}(X)$. on the RHS all the terms are divisible by $g(X)$ except $Xs(X)$, so $d(X)$ is also the remainder of $Xs(X)$ divided by $g(X)$. □

This theorem gives a convenient way to calculate the syndromes of shifted versions of the received vector. In this way we can reduce the list of the syndromes of which we have to know the error vector. One only has to know 1 syndrome for 1 of the possible shifts of the error vector. i.e. for a 1-error correcting binary cyclic n-code one has to know 1 syndrome corresponding to the error polynomial $e(X) = 1$ instead of the n syndromes corresponding to the n possible error vectors $\cdot 10\cdots 0, \ldots, \cdot 0\cdots 10$. For a two error correcting codes there are up to cyclic shifts $n-1$ error polynomials

$$e_1(X) = 1 + X,\ e_2(X) = 1 + X^2, \cdots$$

and C_n^2 possible error vectors.

Algorithm 4.1 (Meggit decoding of a cyclic code)**.** Make a list of error vectors up to cyclic shifts and calculate the corresponding syndrome. Suppose now we receive a polynomial $r(X)$. Set $i := 0$.

1. calculate the syndrome of $r(X)$, i.e. $s(X) := r(X) \mod g(X)$.
2. If $s(X)$ is in the list subtract the corresponding error vector from $r(X)$. Goto step 4 If $i > n$ stop and ask for retransmission.
3. Shift $r(X)$, increase i by 1. Set $s(X) := Xs(X) \mod g(X)$. Goto step 2
4. Shift $r(X)$ i times backwards (or $n - i$ times forwards)

Chapter 5

BCH Codes

5.1 BCH-codes

The Bose-Chaudhuri-Hocquenghem (BCH) codes are a multiple error correcting generalization of Hamming codes. BCH codes are cyclic, and there are several efficient algorithms for decoding them, based on their algebraic structure. There exist both binary and non-binary BCH codes. Reed-Solomon codes are an important subclass of the non-binary BCH codes.

Let us start from the general problem of designing a t-error correcting cyclic code. Consider a cyclic code with generator polynomial $g(X)$ over $\mathbb{F}_q$. Now g has degree k, and the fundamental theorem of algebra says that g has k zeros in a field extension $\mathbb{F}_{q^m}$. Let these k zeros be $\beta_1, \ldots, \beta_k$. Then we can write $g(X)$ as follows.

$$g(X) = \mathsf{lcm}(m_{\beta_1}(X), \ldots, m_{\beta_k}(X))$$

where m_{β_i} is the minimal polynomial of β_i over $\mathbb{F}_q$ and lcm denotes least common multiple. Now let $r(X) = v(X) + e(X)$ be a received polynomial, corresponding to the transmitted code polynomial $v(X)$, and error polynomial $e(X)$. If we evaluate $r(x)$ at the elements $\beta_i \in \mathbb{F}_{q^m}$, corresponding to the zeros of $g(X)$, we find that $r(\beta_i) = e(\beta_i)$ because $v(X)$ is a multiple of $g(X)$ and hence zero in β_i. This leads to the following set of k simultaneous

equations, in the unknowns $e_0, e_1, \ldots, e_{n-1}$.

$$\begin{aligned} e_0 + e_1\beta_1^1 + ... + e_{n-1}\beta_1^{n-1} &= r(\beta_1) \\ e_0 + e_1\beta_2^1 + ... + e_{n-1}\beta_2^{n-1} &= r(\beta_2) \\ &\vdots \\ e_0 + e_1\beta_k^1 + ... + e_{n-1}\beta_k^{n-1} &= r(\beta_k) \end{aligned}$$

The design problem for a t-error correcting code can now be phrased as follows. Choose a generator polynomial $g(X)$ with zeros $\beta_1, \ldots, \beta_k$, such that the k simultaneous equations from above can be solved (hopefully in an efficient manner) whenever at most t of the e_i are non-zero.

As it turns out, if α is a primitive element of $\mathbb{F}_{q^m}$, a suitable set of zeros is $\beta_i = \alpha^i \ i = 1, \ldots, 2t$.

Theorem 5.1 (Primitive BCH Code)**.** Let m and $t < \frac{q^m-1}{2}$ be integers. Then there is a q-ary BCH code with parameters

$$\begin{aligned} n &= q^m - 1 \\ k &\geq n - 2mt \\ d &\geq 2t + 1. \end{aligned}$$

This code is generated by $g(X)$, the lowest degree polynomial over $\mathbb{F}_q$ having as roots $\alpha^{c+i}, \ i = 1, \ldots, 2t$ where α is a primitive element of $\mathbb{F}_{q^m}$ and $c \in \mathbb{Z}$. The parameter $d = 2t + 1$ is called the designed distance of the code.

Proof. If we take the special $g(X)$ as above and suppose $e(X) = \sum_{l=1}^{\nu} e_{i_l} X^{i_l}$ is a polynomial of weight $\nu < 2t + 1$ then this can never be a codeword because then we would have as equations

$$\begin{aligned} e_{i_1}(\alpha^{c+1})^{i_1} + ... + e_{i_\nu}(\alpha^{c+1})^{i_\nu} &= 0 \\ &\vdots \\ e_{i_1}(\alpha^{c+\nu})^{i_1} + ... + e_{i_\nu}(\alpha^{c+\nu})^{i_\nu} &= 0 \end{aligned}$$

These equations can be solved uniquely as long as the following determinant is not zero.

$$\det \begin{pmatrix} (\alpha^{c+1})^{i_1} & \cdots & (\alpha^{c+1})^{i_\nu} \\ \vdots & & \vdots \\ (\alpha^{c+\nu})^{i_1} & \cdots & (\alpha^{c+\nu})^{i_\nu} \end{pmatrix} = \alpha^{(c+1)(i_1+\cdots+i_\nu)} \det \begin{pmatrix} (\alpha^{i_1})^0 & \cdots & (\alpha^{i_\nu})^0 \\ \vdots & & \vdots \\ (\alpha^{i_1})^{\nu-1} & \cdots & (\alpha^{i_\nu})^{\nu-1} \end{pmatrix}$$

$$= \alpha^{(c+1)(i_1+\ldots i_\nu)} \prod_{\kappa \neq \lambda} (\alpha^{i_\kappa} - \alpha^{i_\lambda})$$

The last identity is obtained by using formula for the Vandermonde determinant. Because the $\alpha^{i_\kappa} \neq \alpha^{i_\lambda}$ whenever $i_\kappa \neq i_\lambda$ and both exponents are smaller than $q^m - 1$ the determinant is not zero and the only possible solution is the zero solution. So a polynomial of weight smaller then $2t + 1$ can never be a codeword.

The parameters can be obtained in the following way: we take n as big as we can because of the condition on the i_κ this must be $q^m - 1$. The degree of g is smaller than $2mt$ because the degree of α is m. □

Recall that in $\mathbb{F}_{2^m}$, the elements α and α^2 are conjugates, and have the same minimal polynomial. For a binary BCH code we can form the generator polynomial by considering only the odd powers of the primitive element α

$$g(X) = \mathsf{lcm}(m_{\alpha^1}(X), m_{\alpha^3}(X), \ldots, m_{\alpha^{2t-1}}(X))$$

You should be aware that BCH codes as defined are only one class of BCH codes, namely the primitive ones. The element α can also be selected as a non-primitive root of unity (i.e. $\alpha^n = 1$, where $n < q^m - 1$), yielding a length n code. When $n = q^m - 1$, we say that the code is primitive. If $c = 0$, the code is known as a narrow sense BCH code.

5.2 Code Construction

Given previous theorem, we have the following procedure for designing a t-error correcting narrow sense BCH code of length $q^m - 1$.

1. Find a degree m primitive polynomial $\pi(X)$ over $\mathbb{F}_q$.
2. Use $\pi(X)$ to construct $\mathbb{F}_{q^m}$.
3. Let α be a primitive element of $\mathbb{F}_{q^m}$
4. Find $m_{\alpha^i}(X)$, the minimal polynomial for α^i, for $i = 1, 2, \ldots, 2t$

5. Let $g(X)$ be the minimal degree polynomial with the $m_{\alpha^i}(X)$ as factors, $g(X) = \mathsf{lcm}(m_{\alpha^1}(X), m_{\alpha^2}(X), \ldots, m_{\alpha^{2t}}(X))$.

Example 5.1 (Single error correcting binary BCH code)**.** Let $\pi(X)$ be a degree m binary primitive polynomial, such that $\pi(X)$ is also the minimal polynomial of a primitive element, α in $\mathbb{F}_{2^m}$. This is not always the case f.i. the roots of $X^4 + X^3 + X^2 + X + 1$ are not primitive elements over $\mathbb{F}_{2^4}$.

If it is the case $\pi(X)$ generates a $(2^m - 1, 2^m - m - 1)$ code. This code is in fact the Hamming code because it has the same parameters.

Example 5.2 (Double error correcting binary BCH code.)**.** Let $\pi(X) = 1 + X + X^4$. Let $t = 2$ and $m = 4$. Then

$$\begin{aligned} g(X) &= \mathsf{lcm}(m_\alpha, m_{\alpha^3}) \\ &= (X^4 + X + 1)(X^4 + X^3 + X^2 + X + 1) \\ &= X^8 + X^7 + X^6 + X^4 + 1. \end{aligned}$$

Hence we have designed a (15,7) BCH code with $d \geq 5$. Note however that $g(X)$ has weight 5, and since $g(X)$ is a codeword, the code has $d = 5$.

Example 5.3 (Triple error correcting binary BCH code)**.** Using again $\pi(X)$ from the previous example, we can let $t = 3$, $m = 4$ to design a triple error correcting code. This time the generator polynomial is given by

$$\begin{aligned} g(X) &= \mathsf{lcm}(m_\alpha, m_{\alpha^3}, m_{\alpha^5}) \\ &= (X^4 + X + 1)(X^4 + X^3 + X^2 + X + 1)(1 + X + X^2) \\ &= X^{10} + X^8 + X^5 + X^4 + X^2 + X + 1. \end{aligned}$$

The generator has weight 7, and hence the code has minimum distance exactly 7. This is a $(15, 5, 7)$ code.

BCH codes are cyclic, and we could use any general decoding technique designed for cyclic codes. The special structure of narrow sense BCH codes yields decoders of even lower complexity. The generator polynomial $g(X)$ for BCH codes was chosen specifically such that it has zeros at $\alpha, \alpha^2, \ldots, \alpha^{2t}$, where α is a primitive element of $\mathbb{F}_{q^m}$. In the introduction to the codes, we saw that this implies

$$r(\alpha^i) = e(\alpha^i), i = 1, 2, \ldots, 2t,$$

where $r(X)$ and $e(X)$ are the received polynomial and error polynomial respectively. For $j = 1, \ldots, 2t$, define

$$S_j = r(\alpha^j) = \sum_{l=1}^{n} e_l(\alpha^j)^l,$$

to be the j-th syndrome for the received polynomial $r(X)$. Suppose that a total of $\nu \leq t$ errors occurred, located at positions $i_1, i_2, , \ldots, i_\nu$,

$$e(X) = e_{i_1}X^{i_1} + e_{i_2}X^{i_2} + \cdots + e_{i_\nu}X^{i_\nu}.$$

Then the powers of X define the error locations, and the coefficients determine the error magnitudes. Note that in the case of binary codes, the error magnitudes are all equal to 1. We can now write

$$\begin{aligned}
S_1 &= e_{i_1}\alpha^{i_1} + e_{i_2}\alpha^{i_2} + \cdots + e_{i_\nu}\alpha^{i_\nu} \\
S_2 &= e_{i_1}(\alpha^{i_1})^2 + e_{i_2}(\alpha^{i_2})^2 + \cdots + e_{i_\nu}(\alpha^{i_\nu})^2 \\
&\vdots \\
S_{2t} &= e_{i_1}(\alpha^{i_1})^{2t} + e_{i_2}(\alpha^{i_2})^{2t} + \cdots + e_{i_\nu}(\alpha^{i_\nu})^{2t}
\end{aligned}$$

We need to solve this set of equations for the α_{i_l} and the e_{i_l}. Once these have been found, we can take logarithms (base α) in $\mathbb{F}_{q^m}$ to find the error location numbers i_l, and correct the symbols at these positions by subtracting the corresponding error magnitudes. In order to avoid a profusion of subscripts and superscripts, it is usual to introduce at this point the error location variables $X_l = \alpha^{i_l}$ and the error magnitude variables $Y_l = e_{i_l}$, Using this notation, we have

$$\begin{aligned}
S_1 &= Y_1X_1 + Y_2X_2 + \cdots + Y_\nu X_\nu \\
S_2 &= Y_1X_1^2 + Y_2X_2^2 + \cdots + Y_\nu X_\nu^2 \\
&\vdots \\
S_{2t} &= Y_1X_1^{2t} + Y_2X_2^{2t} + \cdots + Y_\nu X_\nu^{2t}
\end{aligned}$$

This system of $2t$ non-linear equations are symmetric functions of $X_1, X_2, \ldots, X_\nu$ known as power-sum symmetric functions. Any method of solving these equations is a decoding method for BCH codes. Note that if the error location variables are known, the above system of equations is linear in the Y_l, and can be solved using regular techniques from linear algebra. The hard part of the decoding process is to find the error locations.

5.3 The Peterson-Gorenstein-Zierler Decoder

Direct solution of this system is not easy, but by carefully choosing an intermediate variable, we can use a change of variables to linearize the system. This was first done for binary codes by Peterson, then extended to non-binary codes by Gorenstein and Zierler.

Let $\Lambda(X)$ be the polynomial which has as its roots the inverses of the error location variables X_l.

$$\begin{aligned}\Lambda(x) &= \prod_{l=1}^{\nu}(1 - X_l x) \\ &= \Lambda_\nu x^\nu + \Lambda_{\nu-1}x^{\nu-1} + \cdots + \Lambda_1 x + 1.\end{aligned}$$

This polynomial is known as the *error-locator polynomial*, since its roots are directly related to the error location variables. The coefficients Λ_l are however unknown, and must be determined (hopefully in some efficient manner) from the syndromes. Although finding the roots of polynomials is in general a hard problem (over the reals there is no general solution for degree 5 of higher), finite field polynomials have the advantage that we can find the roots by exhaustive search over $\{0, 1, \alpha, \alpha^2, \ldots, \alpha^{q^m-1}\}$. Such a search is known as a Chien search. So far, we have turned the problem of solving the power sum symmetric functions into determining the coefficients of $\Lambda(X)$. Setting $x = X_l^{-1}$, and multiplying by $Y_l X_l^{j+\nu}$, we get

$$\begin{aligned}0 &= Y_l X_l^{j+\nu}\Lambda(X_l^{-1}) \\ &= Y_l X_l^{j+\nu}(\Lambda_\nu X_l^{-\nu} + \Lambda_{\nu-1}X_l^{1-\nu} + \cdots + \Lambda_1 X_l^{-1} + 1) \\ &= Y_l\Lambda_\nu X_l^j + Y_l\Lambda_{\nu-1}X_l^{j+1} + \cdots + Y_l\Lambda_1 X_l^{j+\nu-1} + Y_l X_l^{j+\nu}.\end{aligned}$$

This equation hold for any l and j. Summing up these equations over $l = 1, \ldots, \nu$, we get

$$\begin{aligned}0 &= \sum_{l=1}^{\nu}(Y_l\Lambda_\nu X_l^j + Y_l\Lambda_{\nu-1}X_l^{j+1} + \cdots + Y_l\Lambda_1 X_l^{j+\nu-1} + Y_l X_l^{j+\nu}) \\ &= S_{j+\nu} + \Lambda_1 S_{j+\nu-l} + \cdots + \Lambda_\nu S_j.\end{aligned}$$

Provided that $1 \le j \le t$, these syndromes are known. Hence we have the

following set of linear equations for the X_l.

$$\begin{pmatrix} S_1 & S_2 & S_3 & \cdots & S_{\nu-1} & S_\nu \\ S_2 & S_3 & S_4 & \cdots & S_\nu & S_{\nu+1} \\ S_3 & S_4 & S_5 & \cdots & S_{\nu+1} & S_{\nu+2} \\ \vdots & & & & & \vdots \\ S_\nu & S_{\nu+1} & S_{\nu+2} & \cdots & S_{2\nu-2} & S_{2\nu-1} \end{pmatrix} \begin{pmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \Lambda_{\nu-2} \\ \vdots \\ \Lambda_1 \end{pmatrix} = \begin{pmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ -S_{\nu+3} \\ \vdots \\ -S_{2\nu} \end{pmatrix}$$

Provided that the matrix of syndromes on the left is non-singular, this system can be solved by matrix inversion. The following theorem (which we give without proof) is key for development of decoding process.

Theorem 5.2. The matrix of syndromes

$$M_\mu = \begin{pmatrix} S_1 & S_2 & S_3 & \cdots & S_{\mu-1} & S_\mu \\ S_2 & S_3 & S_4 & \cdots & S_\mu & S_{\mu+1} \\ S_3 & S_4 & S_5 & \cdots & S_{\mu+1} & S_{\mu+2} \\ \vdots & & & & & \vdots \\ S_\mu & S_{\mu+1} & S_{\mu+2} & \cdots & S_{2\mu-2} & S_{2\mu-1} \end{pmatrix}$$

is non-singular if μ is equal to ν, the number of errors that occurred. If $\mu > \nu$, the matrix is singular.

We can now describe the decoding algorithm.

Algorithm 5.1 (Peterson-Gorenstein-Zierler Decoder)**.**

1. Let $r(X)$ be the received polynomial. Calculate the syndromes $S_j = r(\alpha^j), j = 1, 2, \ldots, 2t$. Let $\mu = t$ and Construct the matrix M_μ.
2. Calculate $\Delta = \det M_\mu$.
3. If $\Delta = 0$ let $\mu = \mu - 1$. If $\mu = 0$ a detectable error pattern did not occur, so stop. Otherwise construct the matrix M_μ. This can be obtained from $M_{\mu+1}$ simply be deleting row $\mu + 1$ and column $\mu + 1$. Go back to step 2.
4. Determine the coefficients of the error location polynomial via

$$\begin{pmatrix} \Lambda_\mu \\ \Lambda_{\mu-1} \\ \vdots \\ \Lambda_1 \end{pmatrix} = M_\mu^{-1} \begin{pmatrix} -S_{\mu+1} \\ -S_{\mu+2} \\ \vdots \\ -S_{2\mu} \end{pmatrix}.$$

5. Find the error location variables X_l, by determining the inverses of the zeros of $\Lambda(X)$.

6. Calculate the error magnitudes according to

$$\begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_\mu \end{pmatrix} = \begin{pmatrix} X_1 & X_2 & \cdots & X_\mu \\ X_1^2 & X_2^2 & \cdots & X_\mu^2 \\ \vdots & & & \vdots \\ X_1^\mu & X_2^\mu & \cdots & X_\mu^\mu \end{pmatrix}^{-1} \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_\mu \end{pmatrix}.$$

7. For each $j = 1, 2, \ldots, \mu$, let i be the base α logarithm of X_j. Correct r_i by subtracting Y_j.

Note that for binary codes, Step 6 is not required, since all the error magnitudes are equal to 1. The complexity of this algorithm resides in the need to calculate two matrix inverses over $\mathbb{F}_{q^m}$. The complexity of this decoder increases with the cube of the number of errors corrected. We can simplify the decoding process a bit by using elementary properties of finite field

Theorem 5.3. For a BCH code over $\mathbb{F}_q$,

$$S_{qj} = S_j^q$$

Proof. Using the fact that $\cdot^q$ is an automorphism of $\mathbb{F}_{q^m}$ gives

$$S_{qj} = \sum_{l=1}^{v} Y_l^q X_l^{qj} = \left(\sum_{l=1}^{v} Y_l X_l^j \right)^q = S_j^q$$

□

Example 5.4 (Single error correction)**.** If a single error occurred,

$$S_1 \Lambda_1 = -S_2$$

and hence $\Lambda_1 = -S_2/S_1$. If the code is binary the previous theorem results in $\Lambda_1 = -S_1$, and hence $\Lambda(X) = 1 - S_1 X$, which has a single zero (equal to the inverse of the error locator) at $1/S_1$. Hence if $S_1 = \alpha^i$, the error locater is $X_1 = \alpha^i$ and the error is at location i.

Example 5.5 (Double Error Correction)**.** If errors occurred at two locations,

$$M_2 = \begin{pmatrix} S_1 & S_2 \\ S_2 & S_3 \end{pmatrix}$$

is a non-singular matrix and we can determine the coefficients of $\Lambda(X)$ according to

$$\begin{pmatrix} \Lambda_2 \\ \Lambda_1 \end{pmatrix} = \frac{1}{S_1 S_3 - S_2^2} \begin{pmatrix} S_3 & -S_2 \\ -S_2 & S_1 \end{pmatrix} \begin{pmatrix} -S_3 \\ -S_4 \end{pmatrix}$$

Thus

$$\Lambda_1 = \frac{S_2 S_3 - S_1 S_4}{S_1 S_3 - S_2^2}, \Lambda_2 = \frac{-S_3^2 - S_2 S_4}{S_1 S_3 - S_2^2}.$$

For binary codes, we can use the simplification to find

$$\Lambda_1 = S_1, \Lambda_2 = \frac{S_3 + S_1^3}{S_1}.$$

In either case, double error correction requires finding the roots of a quadratic equation over $\mathbb{F}_{q^m}$, which requires an exhaustive search.

5.4 Reed-Solomon Codes

Reed-Solomon codes are an important sub-class of the non-binary BCH codes. These codes are frequently used in applications. Reed-Solomon codes can be defined simply as follows.

Definition 5.1 (Reed-Solomon Code)**.** A Reed-Solomon code is a BCH code over the field $\mathbb{F}_q$ with generator polynomial

$$g(X) = \prod_{j=0}^{2t-1} (X - \alpha^j)$$

Where α is a primitive element of $\mathbb{F}_q$. The big difference with a general BCH code is that the roots of $g(X)$ are now all in the field where $g(X)$ is defined, so we need not to consider any field extensions. Notice as well that we start with the root $\alpha^0 = 1$ instead of with α. This is to apply some algebraic methods.

Theorem 5.4. An (n, k) Reed-Solomon code is maximal distance separable, i.e. it has the greatest possible minimal distance for a given n and k.

Proof. The minimal distance is at least $2t + 1$ by construction. Because the degree of $g(X)$ is $2t$ $n - k = 2t$ so $d \geq n - k + 1$. On the other hand for a general linear (n, k)-code $d \leq n - k + 1$. This is because the parity check matrix has $n - k$ rows it has rank at most $n - k$ So there is a linear combination of $n-k+1$ column of H that gives us $\mathtt{0}$. This linear combination can be written as $H\mathtt{x}^t = \mathtt{0}$ for a certain vector $\mathtt{x}$ with weight $n - k + 1$. This implies that $\mathtt{x}$ is a codeword and hence $d \leq n - k + 1$. □

This does not however imply that RS codes are the best codes that exist. There are other combinations of n and k that do not yield RS codes, but have better minimum distance. Combining the above results, we have proved the following theorem, concerning the parameters of Reed-Solomon codes.

Theorem 5.5. A t-error correcting Reed Solomon code over $\mathbb{F}_q$ has the following parameters:

$$\begin{aligned} n &= q - 1 \\ k &= q - 1 - 2t \\ d &= 2t + 1 \end{aligned}$$

In the previous section we made an explicit algorithm to decode BCH codes, in this section we will describe more efficient algorithms to decode Reed Solomon codes. We define the *syndrome polynomial* as

$$S(X) = \sum_{l=0}^{2t-1} S_l X^l$$

Consider the ring $\mathbb{F}_q[X]/(X^{2t})$. The polynomial $1 - \alpha^j X$ is not divisible by X and hence invertible. Its inverse is by Taylor expansion chopping of at $X^{2t} \equiv 0$

$$\frac{1}{1 - \alpha^j X} \equiv \sum_{l=0}^{2t-1} (\alpha^j X)^l \mod X^{2t}.$$

Using this identity we get

$$\begin{aligned} S(X) &= \sum_{l=0}^{2t-1} S_l X^l \\ &= \sum_{l=0}^{2t-1} \sum_{k=1}^{\nu} e_{i_k} (\alpha^{i_k} X)^l \\ &= \sum_{k=1}^{\nu} e_{i_k} \left(\sum_{l=0}^{2t-1} (\alpha^{i_k} X)^l \right) \\ &\equiv \sum_{k=1}^{\nu} \frac{e_{i_k}}{1 - \alpha^{i_k} X} \mod X^{2t}. \end{aligned}$$

We define the *error evaluator polynomial* as

$$\Omega(X) = \sum_{k=1}^{\nu} e_{i_k} \prod_{l \neq k} (1 - \alpha^{i_l} X)$$

this polynomial has no zeros in common with the error locater polynomial $\Lambda(X)$ because

$$\Omega(\alpha^{-i_k}) = e_{i_k} \prod_{l \neq k} (1 - \alpha^{i_l - i_k})$$

So $\gcd(\Lambda(X), \Omega(X)) = 1$. The degree of $\Lambda(X)$ is equal to the number of errors $\nu \leq t$, the degree of $\Omega(X)$ is smaller.

We can relate $\Lambda(X)$, $S(X)$ and $\Omega(X)$ in the following way

$$\Omega(X) = \Lambda(X) \sum_{k=1}^{\nu} \frac{e_{i_k}}{1 - \alpha^{i_k} X} \equiv \Lambda(X) S(X) \mod X^{2t}.$$

To find $\Omega(X)$ and $\Lambda(X)$ out of $S(X)$ we will use the algorithm of Euclid. As we saw in the first chapter this algorithm enables us to find the gcd elements by division.

Suppose thus that $a(X)$ and $b(X)$ are polynomials of $\mathbb{F}_q$ then the algorithm of Euclid supplies us with series $r_i(X)$, $s_i(X)$ and $t_i(X)$ $(i = 1, \ldots, \kappa + 1)$ such that

$$s_i(X)a(X) + t_i(X)b(X) = r_i(X) \text{ and } \deg t_i(X) + \deg r_{i-1}(X) = \deg a(X)$$

With $r_0(X) = a(X)$, $r_\kappa(X) = \gcd(a(X), b(X))$ and $r_{\kappa+1}(X) = 0$.

Theorem 5.6. Suppose $t(X)$ and $r(X)$ are nonzero polynomials over $\mathbb{F}_q$ satisfying the following conditions:

1. $\gcd(t(X), r(X)) = 1$,
2. $\deg t(X) + \deg r(X) < \deg a(X)$,
3. $t(X)b(X) = r(X) \mod a(X)$.

Then there exists an index $h \in \mathbb{N}$ and a constant $c \in \mathbb{F}_Q$ such that

$$t(X) = ct_h(X) \text{ and } r(X) = cr_h(X).$$

Where the $r_h(X)$ and the $t_h(X)$ are coming from the algorithm of Euclid on $t(X)$ and $a(X)$.

Proof. First observe that the $\deg r_i(X)$ strictly decreases when i increases. by condition 2 we have that $\deg r < \deg a$ and hence there is an index h such that

$$\deg r_h(X) \leq \deg r(X) < \deg r_{h-1}(X).$$

From condition 3 and Euclid's algorithm we have that

$$\begin{aligned} \exists s(X) \in \mathbb{F}_q[X] : s(X)a(X) + t(X)b(X) &= r(X) \\ s_h(X)a(X) + t_h(X)b(X) &= r_h(X) \end{aligned}$$

Multiplying the equations by $t_h(X)$ and $t(X)$ and subtracting the two results we obtain

$$(t(X)s_h(X) - t_h(X)s(X))a(X) = t(X)r_h(X) - t_h(X)r(X)$$

By the conditions on the degrees, the right equation has a degree strictly smaller then $\deg a(X)$. Therefore both sides must be zero and

$$t(X)r_h(X) = t_h(X)r(X).$$

Because $\deg t_h(X) + \deg r_{h-1}(X) = \deg a(X)$ $t_h \neq 0$ and by condition 1 $r(X)$ divides $r_h(X)$ but it has the same or a higher degree so it is a scalar multiple of $r_h(X)$. Dividing the previous equation by $r_h(X)$, we see that $t(X)$ is also a multiple of $t_h(X)$. □

Based on this theorem, we can find $\Lambda(X)$ and $\Omega(X)$ because they satisfy necessary the conditions if we identify

$$a(X) := X^{2t},\ b(X) := S(X),\ t(X) := \Lambda(X),\ r(X) := \Omega(X)$$

The constant c must be chosen such that $ct_h(0) = \Lambda(0) = 1$. We claim that h is the unique index such that

$$\deg r_h < t \leq \deg r_{h-1}.$$

Indeed, smaller values of i would result in a polynomial $\Omega(X) = cr_i(X)$ whose degree is larger than $t-1$. On the other hand we have for every $i > h$

$$\deg t_i \geq \deg t_{h+1} = \deg a - \deg r_h > t$$

So then $\Lambda(X)$ will have a degree larger than t.

After we have found $\Lambda(X)$ and $\Omega(X)$ we proceed by searching the error locations α^{i_l}, whose inverses are the roots of $\Lambda(X)$.

To find the error magnitudes we will have to recall formal derivations in $\mathbb{F}_q$ For a polynomial $a(X) = \sum_{i=0}^{s} a_i X^i$ we define

$$a'(X) = \sum_{i=1}^{s} i a_i X^{i-1}$$

Where we take i modulo the characteristic of $\mathbb{F}_q$. For these formal derivatives the same properties, like the product rule, hold as in the normal case. We can now calculate that

$$\Lambda'(X) = \sum_{k} -\alpha^{i_k} \prod_{l \neq k} (1 - \alpha^{i_l} X),$$

so by the definition of $\Omega(X)$

$$\Lambda'(\alpha^{-i_k}) = -\alpha^{i_k} \prod_{l \neq k} (1 - \alpha^{i_l - i_k}) = \frac{-\alpha^{i_k}}{e_{i_k}} \Omega(\alpha^{-i_k})$$

This gives us the following expression for the error magnitudes

$$e_{i_k} = \frac{-\alpha^{i_k} \Omega(\alpha^{-i_k})}{\Lambda'(\alpha^{-i_k})}.$$

If we put every thing together we get the following algorithm

Algorithm 5.2 (Berlekamp-Massy-Forney)**.**

1. Compute the syndrome polynomial $S(X)$ out of the received word $m(X)$.

2. Use the algorithm of Euclid for X^{2t} and $S(X)$ to find an index h such that $\deg r_h < t \leq \deg r_{h-1}$.

3. Define $\Lambda(X) = t_h(X)/t_h(0)$ and $\Omega(X) = r_h(X)/t_h(0)$. Find the i_k such that α^{-i_k} is a root of $\Lambda(X)$. There has to be $\deg \Lambda(X)$ distinct roots otherwise to many errors have occured and you must ask for retransmission.

4. Define
$$e_{i_k} = \frac{-\alpha^{i_k}\Omega(\alpha^{-i_k})}{\Lambda'(\alpha^{-i_k})}.$$
The corrected polynomial is
$$m(X) - \sum_k e_{i_k} X^{i_k}.$$

Miniature 4: Elwyn E. Berlekamp (1940-)

Elwyn R. Berlekamp, professor at Berkeley was born in Dover, Ohio on September 6, 1940. In the early 1970s, Dr. Berlekamp founded Cyclotomics, Inc., a research and engineering firm specializing in the development and implementation of high-performance error control systems for digital communications and mass data storage. Cyclotomics designed and developed a variety of innovative electronic subsystems and full-custom integrated circuits that implement novel algorithms for error-correcting codes, deskewing, and synch acquisition for aerospace and commercial applications. In 1984, Cyclotomics "Bit-Serial" Reed Solomon encoders were formally adopted as the NASA standard for deep space communications. On the commercial side, all compact disk players use RS Codes with Berlekamp decoding.

Part II

Cryptography

Chapter 6

History and Background

6.1 The Main Problem

Cryptography is the art or science of secret writing, or more exactly, of storing information (for a shorter or longer period of time) in a form which allows it to be revealed to those you wish to see it yet hides it from all others. A cryptosystem is a method to accomplish this. Cryptanalysis is the practice of defeating such attempts to hide information. Cryptology includes both cryptography and cryptanalysis. The original information to be hidden is called "plaintext". The hidden information is called "ciphertext". Encryption is any procedure to convert plaintext into ciphertext. Decryption is any procedure to convert ciphertext into plaintext.

A cryptosystem is designed so that decryption can be accomplished only under certain conditions, which generally means only by persons in possession of both a decryption engine (these days, generally a computer program) and a particular piece of information, called the decryption key, which is supplied to the decryption engine in the process of decryption. Plaintext is converted into ciphertext by means of an encryption engine (again, generally a computer program) whose operation is fixed and determinate (the encryption method) but which functions in practice in a way dependent on a piece of information (the encryption key) which has a major effect on the output of the encryption process. In this process the encryption key and the decryption key may or

may not be the same. When they are the cryptosystem is called a "symmetric key" system; when they are not it is called an "asymmetric key" system. The most widely-known instance of a symmetric cryptosystem is DES (the so-called "Data Encryption Standard").

6.2 Time Line

In this section we give a short review of the main cryptographic developments in history.

- About 1900 BC: an Egyptian scribe used non-standard hieroglyphs in an inscription.
- 50-60 BC: Julius Caesar (100-44 BC) used a simple substitution with the normal alphabet (just shifting the letters a fixed amount) in government communiciations. This cipher was less strong than a reversed alphabet substitute cipher system, by a small amount, but in a day when few people read in the first place, it was good enough. He also used transliteration of Latin into Greek letters and a number of other simple ciphers.
- 1518: Johannes Trithemius wrote the first printed book on cryptology. He invented a steganographic cipher in which each letter was represented as a word taken from a succession of columns. The resulting series of words would be a legitimate prayer. He also described polyalphabetic ciphers in the now-standard form of rectangular substitution tables. He introduced the notion of changing alphabets with each letter.
- 1918: The ADFGVX system was put into service by the Germans near the end of WW-I. This was a cipher which performed a substitution (through a keyed array), fractionation and then transposition of the letter fractions. It was broken by the French cryptanalyst, Lieutenant Georges Painvin.
- 1933-45: The Enigma machine was not a commercial success but it was taken over and improved upon to become the cryptographic work-

horse of Nazi Germany. [It was broken by the Polish mathematician, Marian Rejewski, based only on captured ciphertext and one list of three months worth of daily keys obtained through a spy. Continued breaks were based on developments during the war by Alan Turing, Gordon Welchman and others at Bletchley Park in England.] Enigma was mainly used in the U-Boats and it was the job of Turing and other mathematicians of the time to break the communication cipher code to prevent further losses in Allied shipping.

- 1976: FIPS PUB-46; a design by IBM, based on the Lucifer cipher and with changes (including both S-box improvements and reduction of key size) by the US NSA, was chosen to be the U.S. Data Encryption Standard. It has since found worldwide acceptance, largely because it has shown itself strong against 20 years of attacks. Even some who believe it is past its useful life use it as a component – e.g., of 3-key triple-DES.

- 1977: One night in April, Ron Rivest was laid up with a massive headache and the RSA algorithm came to him. He wrote it up and sent it to Shamir and Adleman the next morning. RSA is a practical public-key cipher for both confidentiality and digital signatures, based on the difficulty of factoring large numbers. They submitted this to Martin Gardner on April 4 for publication in Scientific American. It appeared in the September, 1977 issue. The Scientific American article included an offer to send the full technical report to anyone submitting a self-addressed, stamped envelope. There were thousands of such requests, from all over the world.

- 1991: Phil Zimmerman released his first version of PGP (Pretty Good Privacy) in response to the threat by the FBI to demand access to the cleartext of the communications of citizens. Although PGP offered little beyond what was already available in products like Mailsafe from RSADSI, PGP is notable because it was released as freeware and has become a worldwide standard as a result.

- 2000: NSA accepted Rijndael, an encryption system by the Belgians Rijmen and Daemen, as the new standard for cryptography replacing DES.

6.3 Basic Methods

The two main historical methods of encryption are substitution and transposition and most known modern methods are a mixture of both.

Substitution: When individual letters or n-grams of plaintext are replaced by letters or n-grams of ciphertext.

e.g. replace any letter in a string with its following letter in the alphabet.

$$\texttt{hello} \longrightarrow \texttt{ifmmp}$$

One can of course shift further in the alphabet, ciphers obtained in this way are called Caesar substitutions

Transposition: when the characters of the original message are rearranged according to some particular pattern.

e.g. reverse the order of all the characters in the string

$$\texttt{hello} \longrightarrow \texttt{olleh}$$

Monoalphabetic Substitution

A monoalphabetic substitution is one where a letter of plaintext always produces the same letter of ciphertext. The simplest example of monoalphabetic substitutions is probably the Caesar Cipher. These are special cases, of the more general substitution, so you may like to read the description of these first. In general, an example of a monoalphabetic substitution is shown below.

Example 6.1 (Monoalphabetic Substitution)**.**

alphabet	a	b	c	d	e	f	g	h	i	j	k	l	m
cipher	Q	R	S	K	O	W	E	I	P	L	T	U	Y
alphabet	n	o	p	q	r	s	t	u	v	w	x	y	z
cipher	A	C	Z	M	N	V	D	H	F	G	X	J	B

PLAINTEXT: there is a house in new orleans

```
CIPHERTEXT: DIONO PV Q ICHVO PA AOG CNUOQAV
```

You may naively think that this cipher is secure, after all there are 26! different cipher alphabets ($4 \cdot 10^{26}$) to choose from, however the letter frequencies and underlying patterns will be unchanged - and as such the cipher can be solved by pen and paper techniques. In English the 10 most common letters ordered by their frequency are `ETAIN SHRDL`. To solve a monoalphabetic substitution, one tries to match the most common letters in the ciphertext to this sequence of letters. The match is however not perfect because of statistical deviations in the letter frequencies.

Monoalphabetical substitution can also be applied on other alphabets such as the set of all digraphs (i.e. pairs of letters). An interesting example of this sort is the Playfair cipher.

Example 6.2. The Playfair cipher was invented by a rather clever person by the name of Wheatstone - but Playfair's name is attached to it as he is the one who was a vocal supporter of it in government circles. Playfair first demonstrated this cipher at a dinner in 1854. The dinner was given by a lord Granville, and a notable guest was Lord Palmerston. The cipher is a form of monoalphabetic substitution, but relies on digraphs rather than single letters - and it is simple to master. The Playfair cipher is believed to be the first digraphic system. We start with a keyword - for instance using "Palmerston" as a keyword we obtain: `PALMERSTONBCDFGHIKQUVWXYZ` and then place the remaining letters in a 5x5 square Note that I and J are considered as the same letter. This system of generating the square degenerated into simply entering the keyword directly into the 5 by 5 square (this is the method we shall use for demonstration purposes, however you should be aware that any method of placing letters into the grid may be used).

P	A	L	M	E
R	S	T	O	N
B	C	D	F	G
H	I	K	Q	U
V	W	X	Y	Z

To encipher some text, that text must first be split into digraphs - double letters are separated, here I've used an `x` - so each digraph will consist of

different letters. If it turns out that the last letter is on its own an `x` is added to the end of the message. So the message `Lord Granville's dinner party`, when split into digraphs will become `lo rd gr an vi lx le sd in ne rp ar ty`. Now the text is ready to encipher. For example, in order to encipher `ay` we must locate `a` and `y` in the square, and find the letter which is in the same row as `a` and the same column as `y`.

P	A	L	M	E
.	.	.	O	.
.	.	.	F	.
.	.	.	Q	.
.	.	.	Y	.

Hence the first letter of the enciphered digraph is `M`, the second letter is found by examining the column containing the first letter and the row containing the second. So in this case the second letter is `W`. Therefore `ay` becomes `MW`. You may like to think of this by imagining the plaintext letters as being one corner of a rectangle, and the ciphertext letters as being the other corners of the rectangle.

What happens if the two letters fall in the same row or column? If they fall in the same row then the letters to the right are taken, and if they fall in the same column then the letters underneath are taken. Note that the table has the topology of a torus, so `Y` is to the right of `X`, `Z` is to the right of `Y`, and `V` is to the right of `Z`. Thus `el` becomes `PM`. Note that the order of the letters in the digraph is important and should be preserved. Using these rules the message is encoded as follows:

```
lo rd gr an vi lx le sd in ne rp ar ty
MT TB BN ES WH TL MP TC US GN BR PS OX
```

and becomes `"MTTBBNESWHTLMPTCUSGNBRPSOX"`. (Note the encoding of `LX` to avoid a double letter). To decode the same rules are used in reverse.

What are the advantages of such a system? The prime reason is that one of the main weapons of the cryptanalyst is weakened. You will have noticed for example, that the letter "e" does not always encipher to the same letter - how it enciphers depends upon what it is paired with - much more ciphertext

must be obtained in order to make use of digraphic frequency analysis (and there are many more digraphs than single letters). In other words it COULD be broken using the same techniques as a single-letter monoalphabetic substitution, but we'd need more text. (Note that this is not the best way to crack playfair!) Also we now have less elements available for analysis in a 100 letter message enciphered using a single letter substitution we have 100 message elements (from a choice of 26) for analysis - if the message had been enciphered using digraphs then we'd only have 50 message elements (from a choice of 676). The cipher had many advantages, no cumbersome tables or apparatus was required, it had a keyword which could be easily changed and remembered and it was very simple to operate. These considerations lend the system well to use as a 'field cipher'. Apparently Wheatstone and Playfair presented this system to the Foreign office for diplomatic use, but it was dismissed as being too complex. Wheatstone countered by claiming that he could teach three schoolboys out of four to use the system in less that fifteen minutes - the under secretary at the FO replied "That is very possible, but you could never teach it to attachés." The cipher was mentioned at Granville's party with a view to its use in the Crimean War. The system was not used in the Crimean war, but there are reports that it served in the Boer war.

Polyalphabetic substitution

In polyalphabetic substitution the cleartext letters are enciphered differently depending upon their placement in the text. As the name polyalphabetic suggests this is achieved by using several cryptoalphabets instead of just one, as is the case in most of the simpler cryptosystems. Which cryptoalphabet to use at a given time is usually guided by a key of some kind, or the agreement can be to switch alphabet after each word encrypted (which, of course, presumes that the word boundaries are kept intact or indicated in some way), but the latter is seldom practiced in real life. Several systems exists, and I shall try and explain some of the more common.

Example 6.3 (Gronsfeld's system)**.** One of the simplest polyalphabetic substitution ciphers is Gronsfeld's system. Gaspar Schott, a German 17-century cryptographer, tells that he was taught this cipher during a trip between Mainz and Frankfurt by count Gronsfeld, hence the name. Gronsfeld's sys-

tem uses a numeric key - usually quite short - e.g. 7341, and this key is repeated, one figure at a time, above the individual letters of the cleartext, like this:

Key:	7	3	4	1	7	3	4	1	7
Text:	G	R	O	N	S	F	E	L	D

To encrypt, one simply count forwards in the alphabet from the letter to be encrypted, the number of steps given by the key figure above, the resulting letter being the crypto. If one happens to reach the last letter of the alphabet, still having remaining steps to count, one begins from the beginning of the alphabet. It helps to think of the alphabet as a ring of letters, instead of a row.

Key:	7	3	4	1	7	3	4	1	7
Text:	G	R	O	N	S	F	E	L	D
Crypto:	N	U	S	O	Z	I	I	M	K

Decryption is the reverse process. One writes out the key figures above the letters of the cryptogram and counts backwards in the alphabet instead to reach the cleartext. Gronsfeld's system can be made more secure against enemy decryption by using a differently ordered alphabet instead of the normal sequence. There are numerous ways to design an unordered alphabet, I will show only one method. Using the key (or, preferably, another key of one's own choosing) from the example above, the following table is constructed:

7	A	E	I	M	Q	U	Y
3	B	F	J	N	R	V	Z
4	C	G	K	O	S	W	
1	D	H	L	P	T	X	

Writing the letters out, row by row, and starting with the row having the lowest keyfigure gives the following unordered sequence:

DHLPTXBFJNRVZCGKOSWAEIMQUY.

The encryption example from above will, when counting in this unordered alphabet, look like this:

Key:	7	3	4	1	7	3	4	1	7
Text:	G	R	O	N	S	F	E	L	D
Crypto:	I	C	E	R	U	R	U	P	F

Example 6.4 (Vigenère)**.** Instead of using a sequence of numbers to add we can also use a codeword a which add the letter values. If this is the case such a system is called a Vigenère cipher.

- **Standard Vigenère**
  In standard Vigenère the value letter of the key word minus 1 are add up by the letters of the message. Here is an example with the codeword PYRAMID.

Key letters:	P	Y	R	A	M	I	D	P	Y	R	A	M	I	D	P
Cleartext:	A	T	T	A	C	K	A	T	S	U	N	D	O	W	N
Crypto:	P	R	K	A	O	S	D	I	Q	L	N	P	W	Z	C

- **Vigenère with unordered alphabets**
  As is the case with Gronsfeld's system, the cipher produced by Vigenère will be harder to break by the enemy if unordered alphabets are used instead of the normal alphabetic sequence. Either the mixed sequence is written down in a fashion similar to the Vigenère table above, or the mixed sequence is written on the top line and down, forming a new leftmost column, which are then used when locating the key-, cleartext, and cryptoletters.

Exercise 6.1. The standard Vigenère was the main cryptographic system used by the Confederated States during the American Civil War, and the following four key phrases used by the Confederates have survived to this day: `IN GOD WE TRUST`, `COMPLETE VICTORY`, `MANCHESTER BLUFF` and, as the war-luck turned: `COME RETRIBUTION`.

Decipher the following example of a real Confederate message (the key being one of the four mentioned above):

```
Jackson, May 25th, 1863 Lieut.  Genl.  Pemberton:
My XAFV. USLX was VVUFLSJP by the BRCYIJ 200,000 VEGT.
SUAJ. NERP. ZIFM. It will be GFOECSZQD as they NTYMNX.
Bragg MJ TPHINZG a QKCMKBSE. When it DZGJX N will YOIG.
AS. QHY. NITWM do you YTIAM the IIKM. VFVEY. How and
where is the JSQML GUGSFTVE. HBFY is your ROEEL.
J. E. Johnston
```

Transposition ciphers

Transposition ciphers are rarely encountered nowadays. They differ from both code systems and substitution ciphers; in a transposition cipher the letters of the cleartext are shifted about to form the cryptogram. This can be done in a number of ways and some systems exist where even whole words are transposed, rather than individual letters. To encrypt Chinese, for instance, one can use a transposition cipher operating on the individual signs of written Chinese (using a substitution cipher for a language like Chinese would be awkward if not impossible).

- **Single columnar transposition**
  One of the easiest ways to achieve transposition is the single columnar transposition cipher. To use it one needs a keyword or phrase, whose letters are numbered according to their presence in the alphabet. The keyword 'Osymandias' is numbered in the following way:

O	S	Y	M	A	N	D	I	A	S
7	8	10	5	1	6	3	4	2	9

  That is, the first occurrence of the letter A is numbered 1, the second 2. There are no B's or C's so the next letter to be numbered are the D followed by I, and so on.

  Next the plaintext are written in rows under the numbered keyword, one letter under each letter of the keyword. Let's say that the plaintext to be encrypted is 'Company has reached primary goal'. It will look like this:

O	S	Y	M	A	N	D	I	A	S
7	8	10	5	1	6	3	4	2	9
c	o	m	p	a	n	y	h	a	s
r	e	a	c	h	e	d	p	r	i
m	a	r	y	g	o	a	l		

Now the letters of the plaintext are copied down by reading them off columnwise in the order stated by the enumeration of the keyword and the result is the finished cryptogram, which - of course - are put into groups of five letters, like this:

AHGAR YDAHP LPCYN EOCRM OEASI MAR

To decrypt a received message enciphered by this method one first must calculate the number of letters present in the cryptogram. This is done to see how many letters there originally were in the last row. As can be seen above, the last two columns - the ones numbered 8 and 9 - only contain two letters and this is important. Now the cryptogram above contains 28 letters and as a legitimate user of the cryptosystem one knows that the keyword is ten letters wide, therefore the last row must consist of eight letters only, the last position being empty. Keeping that in mind - or better still, marking the last two positions of row three in some way to indicate that it should not be used - one numbers the keyword letters just as when encrypting and then start by writing the first three letters of the cryptogram under keyword letter number one, thus:

O	S	Y	M	A	N	D	I	A	S
7	8	10	5	1	6	3	4	2	9
.	.	.	.	a	.	.	.	.	.
.	.	.	.	h	.	.	.	.	.
.	.	.	.	g	.	.	.	*	*

Continue in the same way by writing the next two! letters under keyword letter number two, and so on up to keyword letter ten. Now the cleartext can be read in the normal way, row by row.

Usually when employing a transposition cipher like the above, one adds dummy letters to make the final group five letters long if it isn't already

full. It is important to do this before transposing the letters, otherwise the receiver can't calculate the columns that haven't a full number of letters if the last row isn't complete. In some cases the last row is always made complete by adding dummy letters, but that reduces the security of the cipher and isn't recommended (now, this cipher is quite easy to break anyhow...).

- **Double columnar transposition**

  Double columnar transposition is similar to single columnar transposition, but the process is repeated twice. One either uses the same keyword both times or, preferably, a different one on the second occasion. Let's encrypt the text 'Send armoured car to headquarters' using the keywords 'Agamemnon' and 'Mycenae':

A	G	A	M	E	M	N	O	N
1	4	2	5	3	6	7	9	8
s	e	n	d	a	r	m	o	u
r	e	d	c	a	r	t	o	h
e	a	d	q	u	a	r	t	e
r	s	j						

  (Note dummy letter `j` added at the end to make the total number of letters a multiple of five) This first encryption gives:

  `srer-nddj-aau-eeas-dcq-rra-mtr-uhe-oot`.

  These letters are written under the second keyword, thus:

M	Y	C	E	N	A	E
5	7	2	3	6	1	4
s	r	e	r	n	d	d
j	a	a	u	e	e	a
s	d	c	q	r	r	a
m	t	r	u	h	e	o
o	t					

  And, finally this gives the cryptogram:

```
DEREE ACRRU QUDAA OSJSM ONERH RADTT
```

Double columnar transposition is substantially safer against cryptanalysis than single columnar transposition (not impossible, though).

Product ciphers

Product ciphers are combinations of substitution and transposition techniques chained together. In general they are far too hard to do by hand. However one famous product cipher, the ADFGVX cipher, was used in WW1 by the German army. More generally, wider used use had to wait for the cipher machines.

Example 6.5 (ADFGVX Product Cipher)**.** The cipher is named this way since only letters ADFGVX are used. These letters are chosen since have very distinct morse codes. The code was used by the German's for the great offensive in 1918. It was broken for the Allies by a French cryptoanalist Georges Painvin. It uses a fixed substitution table to map each plaintext letter to a letter pair (row-col index).

	A	D	F	G	V	X
A	K	Z	W	R	1	F
D	9	B	6	C	L	5
F	Q	7	J	P	G	X
G	E	V	Y	3	A	N
V	8	O	D	H	0	2
X	U	4	I	S	T	M

Then uses a keyed block transposition to split letter pairs up ciphertext then written in blocks and sent

Plaintext: `PRODUCTCIPHERS`

Key: `DEUTSCH`

Intermediate Text: `FGAGVDVFXADGXVDGXFFGVGGAAGXG`

Keyed Block Columnar Transposition Matrix

D	E	U	T	S	C	H
2	3	7	6	5	1	4
F	G	A	G	V	D	V
F	X	A	D	G	X	V
D	G	X	F	F	G	V
G	G	A	A	G	X	G

Ciphertext: `DXGX FFDG GXGG VVVG VGFG CDFA AAXA`

Chapter 7

Enigma

The efficiently of the German's armed forces in the second world war was only made possible by the use of radio communications. Messages sent this way had to be enciphered, and the encryption system they used was developed from one that was commercially available before the war. The Enigma was a portable cipher machine used to encrypt and decrypt secret messages. More precisely, Enigma was a family of related electro-mechanical rotor machines there were a variety of different models.

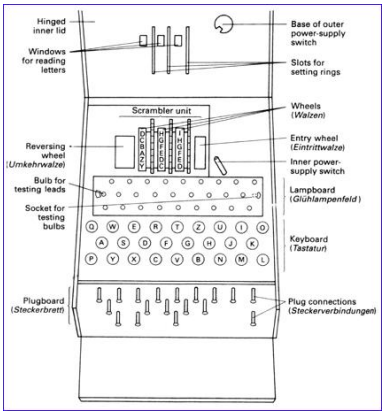
The Enigma was used commercially from the early 1920s on, and was also adopted by the military and governmental services of a number of nations most famously by Nazi Germany before and during World War II.

The German military model, the Wehrmacht Enigma, is the version most commonly discussed. The machine has gained notoriety because Allied cryptologists were able to decrypt a large number of messages that had been enciphered on the machine. The intelligence gained through this source codenamed ULTRA was a significant aid to the Allied war effort. The exact influence of ULTRA is debated, but a typical assessment is that the end of the European war was hastened by two years because of the decryption of German ciphers.

7.1 Description of the Enigma machine

Plain text messages were enciphered and deciphered using a machine called Enigma. This consisted of the following components:

- A 26 letter keyboard.
- A 26 letter lamp board.
- A device called a "scrambler" that was made of three rotating wheels on a common spindle.
- A plugboard known as a "Steckerboard" that added an additional level of security.



These were wired up in the following way. The keyboard and lampboard were both connected to a common cabling bus containing 26 wires. When no keys were being held down all the lamps were connected to the bus and current present on any particular wire would cause it's lamp to glow. When a key was pressed current was placed on it's wire, but the corresponding lamp was disconnected from the bus.

The nature of Enigma was that no letter could ever be enciphered to itself, so this common keyboard/lampboard bus worked well. When a key was pressed one wire on the bus would have current applied to it, and the other 25 could

respond with the enciphered letter. The one that did respond was the result of the other components.

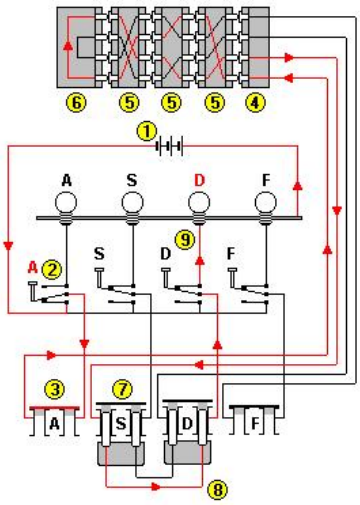
The original commercial Enigma did not have the plugboard, so let us start by considering the machine without this. The 26 wire bus was connected to a circular set of contacts that sat to the right hand side of the three rotor scrambler. These three rotors were identical except in their internal wiring. Each rotor had two sets of 26 connectors one on the left side, and the other on the right. The current from the keyboard would flow along one of the wires, for example let us say `A`, this would arrive at the circular set of 26 contacts that were next to the rightmost rotor. The internal wiring of this rotor would change this input setting on the right to a different output setting on the left. For reasons that will soon become clear this first rotor was known as the "fast" rotor. The current would then exit from the fast rotor on a different contact, let us say `G`. This would enter the middle (medium) rotor and be changed again from `G` to let us say `W`. Again the current would be translated by the third (slow) rotor on the left from `W` to perhaps `C`. When the current exited from the left hand side of the slow rotor it entered into a device called the "reflector". Here the current was returned back to the left hand side of the slow rotor but on a different contact. This mapping provided a further level of enciphering, but, unlike the rotors, because there are only one set of contacts in the reflector the mapping was always reciprocal. So our `C` might now be a `K`. This now passed through the slow rotor to become perhaps an `A`, and then the medium rotor to become perhaps an L, and finally through the fast rotor to become another letter, say a `Z`. This would go into the final set of contacts and back onto the keyboards/lampboard bus and light the `Z` lamp.

All three rotor wheels could be rotated by hand in the Enigma machine. On each rotor there were a set of 26 letters on the outer edge that indicated which way the rotor was set, and because each rotor could be set independently there were $26 \times 26 \times 26$ possible rotor settings. This in itself would not have provided a secure ciphering system because the letter frequency of the ciphered output would quickly have revealed the message. To prevent this every time a key was pressed the fast rotor was advanced one position, so if one were to encipher the same character three times the result would be three different characters. When the fast wheel got to a certain position it would cause the middle wheel to be rotated one position, which could in turn

cause the slow wheel to move, like in an odometer of a car (in fact there was a subtle difference but we will not go into this).

There were two additional complications with the rotors. The first was that their position could be changed in the Enigma machine, allowing six different orderings. The second was that there were actually five rotors, any three of which could be used at one time, giving ten possible selections. As each of these could be placed into the machine six different ways this made sixty different possible rotor configurations. (This is in fact an over simplification because there were several different versions of Enigma, but the differences are not hugely important here.) So basically we now have $26^3 \times 60$ possible initial machine settings.

The Enigma plugboard or steckerboard was a simple additional device which sat between the keyboard/lampboard bus, and the scrambler. Its function was to swap letter pairs. So, for instance, `A` could be swapped with `H` (which would also mean that `H` was swapped with `A`). It is useful in describing Enigma to refer to this process as "Steckering", so one might say that `A` was steckered to `B` (which means the same as `A` being steckered to `A`). Letters that were not plugged would remain unchanged. Each Enigma box had several plug leads (I am not exactly sure how many, but more than 5 and less than the maximum 13) and using these the number of Steckerboard combinations was in the trillions.



Sitting between the keyboard/lampboard bus, and the scrambler caused the Steckerboard to be used twice, just as the rotors were. So now there are a huge number of possible settings. I am simplifying things a little, but the basic way Enigma was used was to have a monthly code book which contained a daily setting that was used for all messages in each 24 hour period. If just one message could be broken the daily setting would be know which could enable all the other messages that day to be read.

With the level of sophistication highlighted above Enigma should have been unbreakable, but the Germans had a number of procedural flaws which allowed the British (and the Polish before them) to break the cipher. Some of these were plain stupid, like reusing the monthly code book settings, but the others were less obvious and this allowed several decryption tricks.

7.2 Work by the Poles to break Enigma

During the 1930s, Polish mathematicians worked to determine the inner wiring of the rotors without having the rotors themselves. The names of some of these mathematicians were Marian Rejewski, Henryk Zygalski and Jerzy Rozycki.

Here we summarize the mathematical reasoning that was done to deduce the wiring of the rotors.

Mathematically we can describe all the operations that occur by permutations on the alphabet Define $\mathcal{A} := \{\mathtt{A}, \cdots, \mathtt{Z}\}$ and $\mathfrak{S}_{\mathcal{A}}$ the set of all permutations of $\mathcal{A}$. We will use the cycle convention to write down permutations so the permutation $\sigma := (ADCG)(TEL)(XS)$ will map

Letter :ABCDEFGHIJKLMNOPQRSTUVWXYZ

Image :DBGCLFAHIJKTMNOPQRXEUVWSYZ

Every rotor of the enigma machine can be represented by a permutation which we we will denote by $\rho_1, \ldots, \rho_5$. Also the steckerbox and the reflector can be seen as permutation σ and τ. τ and σ are involutions (i.e. $\sigma^2 = \tau^2 = 1$) because they are products of cycles of length 2 (switching 2 letters). At

last we take π to be the cyclic shift $(\mathtt{AB}\cdots\mathtt{YZ})$. π^k cyclicly shifts the letters by k positions.

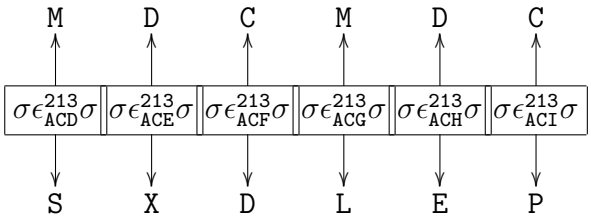
If a rotor is in its original position i.e. `A`, it will act like its permutation ρ_i. If it is rotated to position f.i. `K`, everything will be shifted 10 times so we have to conjugate the permutation with π^{10}.

So if we take the example of rotors `3-4-1` in positions `B-G-S` and for the steckerboard we interchange `D/F, G/M, U/I, A/E, H/Y, K/Q`, the permutation corresponding to the enigma machine will be $\sigma\epsilon^{341}_{\mathtt{BGS}}\sigma$ with

$$\begin{aligned}\sigma &:= (\mathtt{DF})(\mathtt{GM})(\mathtt{UI})(\mathtt{AE})(\mathtt{HY})(\mathtt{KQ})\\ \epsilon^{341}_{\mathtt{BGS}} &:= (\pi^{-18}\rho_1^{-1}\pi^{18})(\pi^{-6}\rho_4^{-1}\pi^{6})(\pi^{-1}\rho_3^{-1}\pi)\tau(\pi^{-1}\rho_1\pi)(\pi^{-6}\rho_1\pi^{6})(\pi^{-18}\rho_1\pi^{18})\end{aligned}$$

We see that every such a permutation is also an involution so $(\epsilon^{341}_{\mathtt{BGS}})^2 = 1$.

The method the Poles used to break the enigma code was based on a weakness in the decoding system of the Germans. Every day there was a daily code stating the order of the rotors used (f.i. `2-1-3`), the start positions of the rotor (f.i. `A-C-D`) and the ways the steckerboard was plugged (f.i. $\sigma =$`(AH)(MU)(LP)(EQ)(IR)(DF)`). For every code of that day the same rotors and steckerboard were used but the starting positions were different and were sent at the beginning of each message and coded in the daily code. The three letters of the start position were transmitted twice so for example



Because the first and the forth letter, etc are the same we can conclude the following identities which depend only on the coded text

$$\begin{aligned}\sigma\epsilon^{213}_{\mathtt{ACG}}\epsilon^{213}_{\mathtt{ACD}}\sigma(\mathtt{S}) &= \mathtt{L}\\ \sigma\epsilon^{213}_{\mathtt{ACH}}\epsilon^{213}_{\mathtt{ACE}}\sigma(\mathtt{X}) &= \mathtt{E}\\ \sigma\epsilon^{213}_{\mathtt{ACI}}\epsilon^{213}_{\mathtt{ACF}}\sigma(\mathtt{D}) &= \mathtt{P}\end{aligned}$$

If we receive enough messages for the day we can construct the three permutations $\sigma\epsilon^{213}_{\mathtt{ACG}}\epsilon^{213}_{\mathtt{ACD}}\sigma, \ldots$. These permutations still depend on both the the

steckerboard and the rotor arraignment. We want to exclude the steckerboard because it has to much possibilities to check, and hence we want to look at something that is independent of σ. Because the action of sigma is by conjugation we can look at the conjugation classes of the three permutations.

Theorem 7.1. Every conjugation class in $\mathfrak{S}_n$ is completely determined by its Young partition (which is a decompsition of n in an unordered sum of natural numbers e.g. $7 = 1+1+2+3$). The young partition of a permutation corresponds to the length of its cycles:

$$(1456)(379) \in \mathfrak{S}_{10} \quad \longrightarrow \quad 10 = 4+3+1+1+1$$

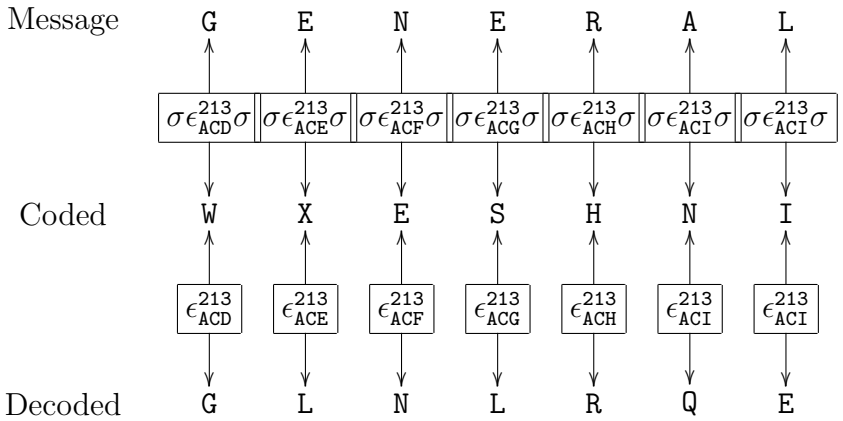
Exercise 7.1. Write a program to compute how many young partitions there are for a given n.

The number of young partitions of 26 is 2436. Up to conjugation with σ the three permutations we deduced give us an ordered sequence of 3 young partitions. There are 2436^3 such triples which is far more than the 6×26^3 (because at that time there were only 3 rotors) so we probably can determine the start positions by its triple.

Therefore the Polish cryptographers, which had a copy of the enigma machine by espionnage, made a list of all possible starting positions and their corresponding triples. every day they used the messages to derive the three young partitions and then looked them up in their table to find the starting position of the day.

Once the starting positions of a message were known, you can try to decode it by typing the encrypted text in the enigma machine with the rotor in the

exact position and without any steckers. This decoding will give you



Because σ is made of 6 swaps, 14 letters will remain unchanged, the same holds for $\epsilon^{213}_{\mathtt{ACD}}\sigma\epsilon^{213}_{\mathtt{ACI}}$. If we consider the $\epsilon^{213}_{\mathtt{ACD}}$ as random permutations then we have $14/26 > 50\%$ chance that a message letter L is decoded as $\sigma(\mathtt{L})$. Of these $14/26$ there are again 14 letter out of 26 for which $\sigma(\mathtt{L}) = \mathtt{L}$. About one third of letters are decoded correct, and one can with some luck find comprehensible phrases and use these to determine the swaps of σ.

It took the Poles about a year to construct the table, once they had it it was very usefull but by the end of the '30s the Germans had introduced 2 new rotors which multiplied the possible rotor combination by a factor 10 so with the same velocity the poles would have had their new table finished by the year 1949 which was rather late. More drastic methods were needed.

7.3 The Turing bomb

Due to the work of the Poles, along with captured Enigma machines, the English were aware of the wirings of all of the rotors. Their work concerned trying to determine the initial rotor setting (initial state) of the machine, along with the plugboard settings, for a given day. Many techniques were used, but we will mainly focus on one that was made possible by a machine called the Turing Bomb.

Another weakness in the communication system of the Germans was the rigidity of the messages sent. The officers frequently used the same phrases

in their messages, f.i. a message to general ... would always start with ANGENERAL... so one could always find a piece of text of which one knows the decryption, such a sample is called a crib. In this section we will use the following example

Nr	1	2	3	4	5	6	7	8	9	10	11	12	13
Text	K	E	I	N	E	Z	U	S	A	E	T	Z	E
Code	D	A	E	D	A	Q	O	Z	S	I	Q	M	M
Nr	14	15	16	17	18	19	20	21	22	23	24	25	
Text	Z	U	M	V	O	R	B	E	R	I	Q	T	
Code	K	B	I	L	G	M	P	W	H	A	I	V	

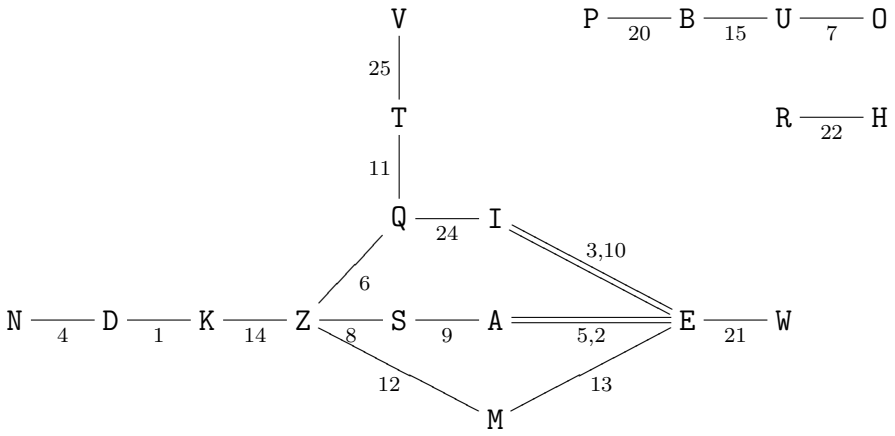
Take a rotor combination 2-3-5 and define $\epsilon_1 := \epsilon^{245}_{\mathtt{AAA}}$, $\epsilon_2 := \epsilon^{245}_{\mathtt{AAB}}$, $\epsilon_3 := \epsilon^{245}_{\mathtt{AAC}}, \ldots$.

If the rotor settings are correct then there exists a $k \in \mathbb{N}$. Such that

$$\begin{aligned} \sigma\epsilon_{k+1}\sigma(\mathtt{K}) &= D \\ \sigma\epsilon_{k+2}\sigma(\mathtt{E}) &= A \\ \sigma\epsilon_{k+3}\sigma(\mathtt{I}) &= E \\ &\vdots \end{aligned}$$

where σ is again the steckerboard permutation, The problem is now to find k without knowing σ.

Using the crib we can construct a graph having as vertices the letters of the alphabet and as edges the different couples of a plaintext letter and its encryption.



For a given k the vertex with number i, corresponds to the permutation $\sigma\epsilon_{k+i}\sigma$. To every path p in the graph (f.i. $\mathtt{S} \overset{8}{\text{---}} \mathtt{Z} \overset{6}{\text{---}} \mathtt{Q}$) we can associate a permutation ϕ_p (by combining the permutations of the edges ($\sigma\epsilon_{k+8}\sigma\sigma\epsilon_{k+6}\sigma = \sigma\epsilon_{k+8}\epsilon_{k+6}\sigma$)

We now take as base letter, the letter with the most cycles through so in this case E. We then make a list of the cycles through E and the corresponding permutations

$$\begin{array}{l|l}
\mathtt{E} \overset{13}{\text{---}} \mathtt{M} \overset{12}{\text{---}} \mathtt{Z} \overset{8}{\text{---}} \mathtt{S} \overset{9}{\text{---}} \mathtt{A} \overset{2}{\text{---}} \mathtt{E} & \sigma\epsilon_{k+13}\epsilon_{k+12}\epsilon_{k+8}\epsilon_{k+9}\epsilon_{k+2}\sigma \\
\mathtt{E} \overset{13}{\text{---}} \mathtt{M} \overset{12}{\text{---}} \mathtt{Z} \overset{8}{\text{---}} \mathtt{S} \overset{9}{\text{---}} \mathtt{A} \overset{5}{\text{---}} \mathtt{E} & \sigma\epsilon_{k+13}\epsilon_{k+12}\epsilon_{k+8}\epsilon_{k+9}\epsilon_{k+5}\sigma \\
\mathtt{E} \overset{13}{\text{---}} \mathtt{M} \overset{12}{\text{---}} \mathtt{Z} \overset{6}{\text{---}} \mathtt{Q} \overset{24}{\text{---}} \mathtt{I} \overset{3}{\text{---}} \mathtt{E} & \sigma\epsilon_{k+13}\epsilon_{k+12}\epsilon_{k+6}\epsilon_{k+24}\epsilon_{k+3}\sigma \\
\mathtt{E} \overset{13}{\text{---}} \mathtt{M} \overset{12}{\text{---}} \mathtt{Z} \overset{6}{\text{---}} \mathtt{Q} \overset{24}{\text{---}} \mathtt{I} \overset{10}{\text{---}} \mathtt{E} & \sigma\epsilon_{k+13}\epsilon_{k+12}\epsilon_{k+6}\epsilon_{k+24}\epsilon_{k+10}\sigma \\
\vdots & \vdots
\end{array}$$

Every cycle implies an equality:

$$\sigma\epsilon_{k+13}\ldots\epsilon_{k+2}\sigma(\mathtt{E}) = \mathtt{E} \text{ or } \epsilon_{k+13}\ldots\epsilon_{k+2}\sigma(\mathtt{E}) = \sigma(\mathtt{E})$$

So if we define $\mathrm{Fix}(\pi) := \{p|\pi(p) = p\}$, we have that

$$\sigma(\mathtt{E}) \in \bigcap_{c \text{ is a cycle through } \mathtt{E}} \mathrm{Fix}(\epsilon_{k,c})$$

if k is the correct initial setting. If k is not equal to the initial setting, the chance that this intersection in non-empty is not bigger than $26^{1-\#\text{cycles}}$ because a random permutation has a chance $1/26$ for fixing a given letter and there are 26 possible letters that can be fixed.

The permutation $\epsilon_{k,c}, k = 0, \ldots$ could be easily constructed by connecting different enigmas in the different rotor positions without a steckerboard in the same way as the graph. To vary k one had to rotate all enigmas once at the same time.

The first Turing bomb consisted of such cycles of connected enigmas and a device that checked whether there was one letter that was mapped onto itselve by all the cycles. If this was the case the machine stopped rotating and the position was written down. Of course there could be more than one

position that had this property so out of the list that was written down the crackers had to find manually the correct rotor positions.

This version of the Turing bomb had some disadvantages. First of all it was useles if the crib didn't contain any cycles and secondly it didn't give any clues about the steckerboard except for $\sigma(E)$. In order to take these problems in account Turing and this mates constructed a more powerfull version.

The idea was to construct a machine that could deduce the steckerboard permutation out of the rotor positions and some other assumptions. If one had some initial assumptions about σ then one could deduce conclusions using the different paths in the graph of the crib. F.i. if we suppose that $\sigma\mathtt{E} = \mathtt{P}$ then we also know that

$$\sigma\mathtt{M} = \epsilon_{k+13}\sigma\mathtt{E} = \epsilon_{k+13}\mathtt{P}.$$

In general we have a set of 26×26 statements of the form $\sigma(\mathtt{L}_1) = \mathtt{L}_2$ which can be ordered in a table

	A	$\cdots$	Z
A	$\sigma(\mathtt{A}) = \mathtt{A}$	$\cdots$	$\sigma(\mathtt{A}) = \mathtt{Z}$
$\vdots$	$\vdots$		$\vdots$
A	$\sigma(\mathtt{Z}) = \mathtt{A}$	$\cdots$	$\sigma(\mathtt{Z}) = \mathtt{Z}$

For every edge $\mathtt{L}_1 \overset{l}{-} \mathtt{L}_2$ of the graph of the crib we have implications

$$\sigma\mathtt{L}_1 = \mathtt{L}_3 \Rightarrow \sigma\mathtt{L}_2 = \epsilon_{k+l}\mathtt{L}_3$$

Apart from that we also have implications

$$\sigma\mathtt{L}_1 = \mathtt{L}_3 \Rightarrow \sigma\mathtt{L}_3 = \mathtt{L}_1$$

because σ is an involution. If we now make a graph Γ_k with as vertices the couples of letters $\mathcal{A} \times \mathcal{A}$ and as edges

$$\begin{cases} (\mathtt{L}_1, \mathtt{L}_3) - (\mathtt{L}_2, \epsilon_{k+1}\mathtt{L}_3) & \text{if } \mathtt{L}_1 \overset{l}{-} \mathtt{L}_2 \text{ is in the crib graph} \\ (\mathtt{L}_1, \mathtt{L}_2) - (\mathtt{L}_2, \mathtt{L}_1) & \forall \mathtt{L}_1, \mathtt{L}_2 \in \mathcal{A} \end{cases}$$

If for a certain k we can deduce the statement

$$\sigma \mathtt{L}_1 = \mathtt{L}_3 \Rightarrow \sigma \mathtt{L}_2 = \mathtt{L}_4$$

this means that in the graph Γ_k , the vertices $(\mathtt{L}_1, \mathtt{L}_3)$ and $(\mathtt{L}_2, \mathtt{L}_4)$ will be connected.

If k is the correct rotor setting and $\sigma \mathtt{L}_1 = \mathtt{L}_2$ there will be no contradictions in the deductions so none of the vertices $(\mathtt{L}_1, \mathtt{L}_3),\ \mathtt{L}_1 \neq \mathtt{L}_3$ will be connected with $(\mathtt{L}_1, \mathtt{L}_2)$. So there will be a connected component of Γ_k that hits every horizontal row vertices $\{\mathtt{L}_1\} \times \mathcal{A}$ in at most 1 (not exactly one because it is possible that you can not deduce σ completely out of the crib. Turing applied this graph theoretical result to construct his machine. The trick was to turn this graph into an electrical circuit. If this was done one chooses the letter $\mathtt{L}_1$ where the crib graph has the most edges and chooses a couple $(\mathtt{L}_1, \mathtt{L}_2)$ and puts an electric potential on it, then all the vertices that are connected with $(\mathtt{L}_1, \mathtt{L}_2)$ will also have this potential.

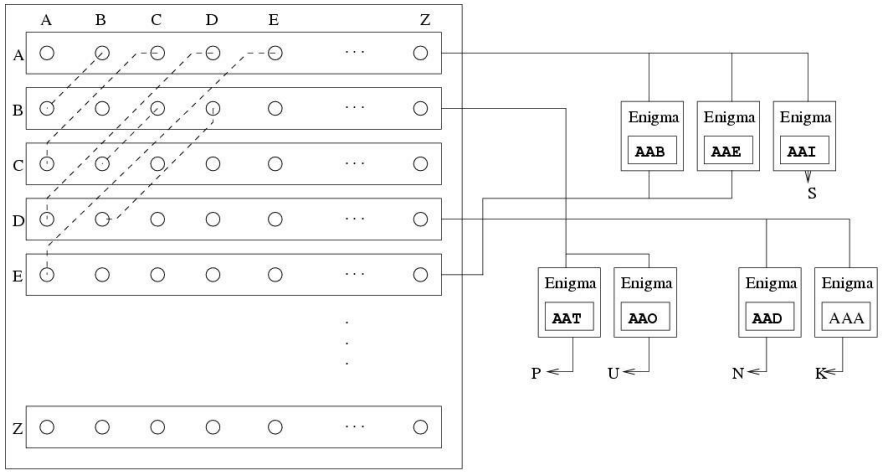
If k was the correct position and $(\mathtt{L}_1, \mathtt{L}_2)$ was part of the steckerboard, the horizontal line of $\mathtt{L}_1$ had no other vertex with this potential.

If k was correct but $(\mathtt{L}_1, \mathtt{L}_2)$ was not part of the steckerboard, the implication $\sigma \mathtt{L}_1 = \mathtt{L}_2$ probably led to contradictions and in almost all cases everything of the horizontal line would be under potential except the correct steckerboard plug.

If k was not a correct rotor setting every possible $\sigma \mathtt{L}_1 = \mathtt{L}_2$ would lead to contradictions and in almost all cases everything of the horizontal would be under potential.

The graph could be easily constructed by attaching enigma machines to a square board with 26^2 contacts corresponding to the letter couples on it. For every edge of the crib graph two horizontal lines of contacts were connected by an enigma machine. Apart from that two contacts are also connected by a wire if their couples are swaps of each other.

On the row of one letter there was an electric circuit that put a potential on one of the contacts and that could detect whether only one or all but one contact was under potential. Now the enigma machines rotated all at the same time and the electric circuit stopped the rotation whenever such an incident happened.



If this was the case the code cracker read of the rotor and steckerboard positions, and tried to decode the whole coded message using this information.

Chapter 8

Modern Cryptography and Complexity

Cryptography is about communication in the presence of an adversary. It encompasses many problems (encryption, authentication, key distribution to name a few). The field of modern cryptography provides a theoretical foundation based on which we may understand what exactly these problems are, how to evaluate protocols that purport to solve them, and how to build protocols in whose security we can have condence. We introduce the basic issues by discussing the problem of encryption.

8.1 The problem

The most ancient and basic problem of cryptography is secure communication over an insecure channel. Party A wants to send to party B a secret message over a communication line which may be tapped by an adversary. The traditional solution to this problem is called private key encryption. In private key encryption A and B hold a meeting before the remote transmission takes place and agree on a pair of encryption and decryption algorithms $\mathcal{E}$ and $\mathcal{D}$, and an additional piece of information S to be kept secret. We shall refer to S as the common secret key. The adversary may know the encryption and decryption algorithms $\mathcal{E}$ and $\mathcal{D}$ which are being used, but does not know

S.

After the initial meeting when A wants to send B the cleartext or plaintext message $\mathtt{m}$ over the insecure communication line, A encrypts $\mathtt{m}$ by computing the ciphertext $\mathtt{c} = \mathcal{E}(\mathtt{S}, \mathtt{m})$ and sends $\mathtt{c}$ to B. Upon receipt, B decrypts c by computing $\mathtt{m} = \mathcal{D}(\mathtt{S}, \mathtt{c})$. The line-tapper (or adversary), who does not know $\mathtt{S}$, should not be able to compute $\mathtt{m}$ from $\mathtt{c}$.

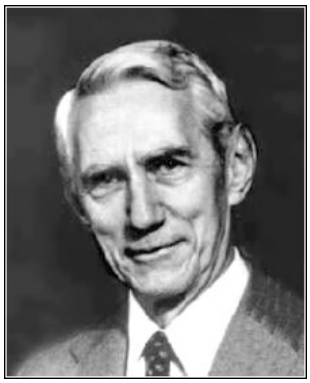
Let us illustrate this general and informal setup with an example familiar to most of us from childhood, the substitution cipher. In this method A and B meet and agree on some secret permutation $f : \Sigma \to \Sigma$ (where Σ is the alphabet of the messages to be sent). To encrypt message $\mathtt{m} = \mathtt{m}_1 \dots \mathtt{m}_n$ where $\mathtt{m}_i \in \Sigma$, A computes $\mathcal{E}(f, \mathtt{m}) = \mathtt{f}(\mathtt{m}_1) \dots \mathtt{f}(\mathtt{m}_n)$. To decrypt $\mathtt{c} = \mathtt{c}_1 \dots \mathtt{c}_n$ where $\mathtt{c}_i \in \Sigma$, B computes $\mathcal{D}(f, \mathtt{c}) = \mathtt{f}^{-1}(\mathtt{c}_1) \dots \mathtt{f}^{-1}(\mathtt{c}_n) = \mathtt{m}_1 \dots \mathtt{m}_n = \mathtt{m}$. In this example the common secret key is the permutation f . The encryption and decryption algorithms $\mathcal{E}$ and $\mathcal{D}$ are as specified, and are known to the adversary. We note that the substitution cipher is easy to break by an adversary who sees a moderate (as a function of the size of the alphabet Σ) number of ciphertexts.

A rigorous theory of perfect secrecy based on information theory was developed by Shannon in 1943. In this theory, the adversary is assumed to have unlimited computational resources. Shannon showed that secure (properly defined) encryption system can exist only if the size of the secret information S that A and B agree on prior to remote transmission is as large as the number of secret bits to be ever exchanged remotely using the encryption system.

An example of a private key encryption method which is secure even in presence of a computationally unbounded adversary is the *one time pad*. A and B agree on a secret bit string $\mathtt{pad} = \mathtt{b}_1\mathtt{b}_2 \dots \mathtt{b}_n$, where $\mathtt{b}_i \in \{0, 1\} = \mathbb{F}_2$ with uniform probability. This is the common secret key. To encrypt a message $\mathtt{m} = \mathtt{m}_1\mathtt{m}_2 \dots \mathtt{m}_n \in \mathbb{F}_2^n$, A computes $\mathcal{E}(\mathtt{pad}, \mathtt{m}) = \mathtt{m} + \mathtt{pad}$ (i.e. bitwise exclusive or). To decrypt ciphertext $\mathtt{c} \in \mathbb{F}_2^n$, B computes $\mathcal{D}(\mathtt{pad}, \mathtt{c}) = \mathtt{pad} + \mathtt{c} = \mathtt{pad} + (\mathtt{m} + \mathtt{pad}) = \mathtt{m}$. As the pad was randomly chosen the the adversary cannot decrypt $\mathtt{c}$ without knowing $\mathtt{pad}$ because for every possible message $\mathtt{m}' \in \mathbb{F}_2$ one can decrypt $\mathtt{c}$ as $\mathtt{m}'$ using the pad $\mathtt{c} + \mathtt{m}'$. From this, it can be argued that seeing $\mathtt{c}$ gives no information about what has been sent.

Now, suppose A wants to send B an additional message $\mathtt{n}$. If A were to simply send $\mathtt{c} = \mathcal{E}(\mathtt{pad}, \mathtt{n})$, then the sum of the lengths of messages $\mathtt{m}$ and $\mathtt{n}$ will exceed the length of the secret key pad, and thus by Shannon's theory the system cannot be secure. Indeed, the adversary can compute $\mathcal{E}(\mathtt{pad}, \mathtt{m}) + \mathcal{E}(\mathtt{pad}, \mathtt{n}) = \mathtt{m} + \mathtt{n}$ which gives information about $\mathtt{m}$ and $\mathtt{n}$ (e.g. can tell which bits of $\mathtt{m}$ and $\mathtt{n}$ are equal and which are different). To fix this, the length of the pad agreed upon a-priori should be the sum total of the length of all messages ever to be exchanged over the insecure communication line.

Miniature 5: Claude Shannon



Claude Elwood Shannon is considered as the founding father of information theory. He joined the staff of Bell Telephone Laboratories in 1942. While working at Bell Laboratories, he formulated a theory explaining the communication of information and worked on the problem of most efficiently transmitting information. The mathematical theory of communication was the climax of Shannon's mathematical and engineering investigations. The concept of entropy was an important feature of Shannon's theory, which he demonstrated to be equivalent to a shortage in the information content in a message.

8.2 Complexity theory

Modern cryptography abandons the assumption that the Adversary has available infinite computing resources, and assumes instead that the adversary's computation is resource bounded in some reasonable way. In particular, in these notes we will assume that the adversary is an algorithm that runs in polynomial time. Similarly, the encryption and decryption algorithms designed run in polynomial time.

Computational complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps does it take to solve a problem) and space (how much memory does it take to solve a problem).

In an intuitive way we can say that an algorithm A is a computer program which transforms a binary input string $\mathtt{i} \in \cup_{\mathtt{n}\in\mathbb{N}}\mathbb{F}_2^{\mathtt{n}} := \mathbb{F}^\bullet$ to an output $\mathtt{u} =\in \mathbb{F}^\bullet$. The length of the input or output is denoted by $|\mathtt{i}|$ or $|\mathtt{u}|$ and is equal to its number of binary digits. A defines a function $f_A : \mathbb{F}^\bullet \to \mathbb{F}^\bullet$ which assigns to every input its output.

Note that we could also allow more than one input or output string, is this case f_A is a function from $(\mathbb{F}^\bullet)^k$ to $(\mathbb{F}^\bullet)^l$ for some fixed k, l. Denote the set of all such algorithms with $\mathcal{A}_{k,l}$.

The time that an algorithm A takes to compute the output from a given input $\mathtt{i}$ is defined as the number of operations (binary logical operations as AND, OR, NOT) that the program has to perform on the bits and this is denoted by $T_A(\mathtt{i})$. An algorithm A is in polynomial time if $\forall i : T_A(\mathtt{i}) \leq \mathtt{g}(|\mathtt{i}|)$ for some polynomial $g(X)$.

Exercise 8.1. Show that the standard algorithms for addition, multiplication and division are in polynomial time.

A problem $\mathtt{P}$ is a function $f : \mathbb{F}^\bullet \to \mathbb{F}^\bullet$ and it is called a *decision problem* if the output is in $\mathbb{F}_2 \subset \mathbb{F}^\bullet$.

In this theory, the class $\mathcal{P}$ consists of all those decision problems that can be solved by a computer program in an amount of time that is polynomial in the size of the input. Formally

$$\mathtt{P} \in \mathcal{P} \Leftrightarrow \exists A \in \mathcal{A}_{1,1} : \exists g(X) \in \mathbb{R}[X] : f_A = \mathtt{P} \text{ and } \forall \mathtt{i} \in \mathbb{F}^\bullet : \mathtt{T_A(i)} \leq \mathtt{g}(|\mathtt{i}|).$$

The class $\mathcal{NP}$ consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine. Formally

$$\mathtt{P} \in \mathcal{NP} \Leftrightarrow \exists A \in \mathcal{A}_{2,1} : \exists g(X) \in \mathbb{R}[X] :$$

$$\forall \mathtt{i} : \begin{cases} \mathtt{P(i)} = 1 & \Rightarrow \exists \mathtt{y} : f_A(\mathtt{i}, \mathtt{y}) = 1 \text{ and } T_A(\mathtt{i}, \mathtt{y}) \leq g(|\mathtt{i}|) \\ \mathtt{P(i)} = 0 & \Rightarrow \forall \mathtt{y} : f_A(\mathtt{i}, \mathtt{y}) = 0 \end{cases}$$

Analogously we define say that a problem P is in co$\mathcal{NP}$ if $\neg$P $\in \mathcal{NP}$.

The biggest open question in theoretical computer science concerns the relationship between those two classes:

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}.$$

Most people think that the answer is probably "no"; some people believe the question may be undecidable from the currently accepted axioms. A $\$1,000,000$ prize has been offered for a correct solution.

In essence, the $\mathcal{P} = \mathcal{NP}$ question asks: if positive solutions to a YES/NO problem can be verified quickly, can the answers also be computed quickly? Here is an example to get a feeling for the question. Given two large numbers X and Y, we might ask whether Y is a multiple of any integers between 1 and X, exclusive. For example, we might ask whether 69799 is a multiple of any integers between 1 and 250. The answer is YES, though it would take a fair amount of work to find it manually. On the other hand, if someone claims that the answer is YES because 223 is a divisor of 69799, then we can quickly check that with a single division. Verifying that a number is a divisor is much easier than finding the divisor in the first place. The information needed to verify a positive answer is also called a certificate. So we conclude that given the right certificates, positive answers to our problem can be verified quickly (i.e. in polynomial time) and that's why this problem is in $\mathcal{NP}$. It is not known whether the problem is in $\mathcal{P}$. The special case where $X = Y$ was first shown to be in $\mathcal{P}$ in 2002.

8.3 $\mathcal{NP}$-complete problems

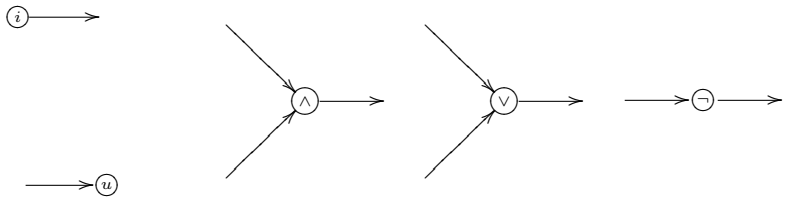
An important role in this discussion is played by the set of $\mathcal{NP}$-complete problems (or $\mathcal{NPC}$) which can be loosely described as those problems in $\mathcal{NP}$ that are the least likely to be in $\mathcal{P}$.

A decision problem P is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and if every other problem in $\mathcal{NP}$ is reducible to it. *Reducible* here means that for every $\mathcal{NP}$-problem L, there is a polynomial-time algorithm which transforms instances of L into instances of P, such that the two instances have the same values. As

a consequence, if we had a polynomial time algorithm for $\mathtt{P}$, we could solve all $\mathcal{NP}$-problems in polynomial time.

To prove that a problem is $\mathcal{NP}$-complete, one shows that one can reduce another problem that is known to be $\mathcal{NP}$-complete, to this problem. A standard problem that is known to be $\mathcal{NP}$-complete is the boolean circuit problem

A boolean circuit is a directed graph without oriented cycles of which the vertices of the forms below such that there is onle one output vertex $\textcircled{u}$.



The evaluation of a circuit given values in $\mathbb{F}_2$ for every input vertex $\textcircled{i}$, assigns to every other vertex a binary value such that for the operator vertices the value is the result of the operation on the incoming vertices and the vertex $\textcircled{u}$ has the same value as the one it is connected to.

It is easy to encode every circuit as a binary string. Given an appropriate encoding method, the circuit decision problem $\mathtt{CIRC} : \mathbb{F}_2^\bullet \to \mathbb{F}_2$ the maps a string to 1 if and only if the string corresponds to a circuit and there exist input values such that the circuit evaluates the output vertex as one.

Theorem 8.1. $\mathtt{CIRC}$ is $\mathcal{NP}$-complete.

Proof. (Sketch) First of all it is in $\mathcal{NP}$, because evaluation of a circuit given a certain input is an algorithm in polynomial time.

Now suppose that we have a decision problem $\mathtt{P}$ in $\mathcal{NP}$, then there exists an algorithm $A \in \mathcal{A}_{2,1}$ in polynomial time satisfying the requirements of the $\mathcal{NP}$-definition. We can transform this algorithm into a new algorithm $\tilde{A}$ that gives as output a boolean circuit that has for each input bit (of the second argument) an input vertex and such that the evaluation of the circuit with the input bit gives the same value as A. This new algorithm is in polynomial time because the old one was. The proof finishes with the observation that $\mathtt{P} = \mathtt{CIRC} \circ f_{\tilde{A}}$. □

As we already said, the easiest way to prove that some new problem is $\mathcal{NP}$-complete is to reduce some known $\mathcal{NP}$-complete problem to it. Therefore, it is useful to know a variety of $\mathcal{NP}$-complete problems. Here are a few:

- `SAT`: given a boolean expression (e.g. $(x_1 \wedge x_2) \vee \neg x_3$), does there exist an assignment to the variables such that the expression is true.
- `3-SAT`: given a boolean expression of the form

$$\underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{\text{3 terms}} \wedge \cdots \wedge \underbrace{(x_i \vee \neg x_j \vee \neg x_3)}_{\text{3 terms}}$$

  does there exist an assignment to the variables such that the expression is true.
- `CLIQUE`: A clique in a graph is a set of pairwise adjacent vertices. The k-Clique Problem is simply the problem of deciding if a graph has a clique of size k.
- `VCOVER`: A vertex cover of a graph is a subset of the vertices of the graph which contains at least one of the two endpoints of each edge. The vertex cover problem is to decide for a given graph and given k whether there is a cover with k or less vertices.
- `HAM`: a hamiltonian cycle in a graph is a path that runs through every vertex once. `HAM` decides whether a given graph has a hamiltonian cycle.
- The knapsack problem: given a set of items, each with a cost and a value, determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible. The decision problem form of the knapsack problem is the question "can a value of at least V be achieved without exceeding the cost C?"

The special status of $\mathcal{NP}$-complete problems in complexity theory is the following: to settle the $\mathcal{P} = \mathcal{NP}$ question one either has to prove that an $\mathcal{NP}$-complete problem is in $\mathcal{P}$ or that it cannot be in $\mathcal{P}$.

We end this section with some remarks on the factorization problem. We can turn the factorization question into a decision problem by defining

$$\texttt{FACTOR}(p, s) = 1 \Leftrightarrow \exists d \in [2, s[: d|p.$$

A related problem is PRIMES:

$$\texttt{PRIMES}(p) = 1 \Leftrightarrow \forall d \in [2, p[: d \not| p.$$

As already mentioned, FACTOR is in $\mathcal{NP}$ but more surprising is that also $\neg\texttt{PRIMES} \in \mathcal{NP}$. To prove this we need a fact from number theory.

$$p \text{ is prime } \Leftrightarrow \mathbb{Z}_p^*, \times \text{ is a cyclic group}$$

This means that there exists a $1 < k < p$ such that $k^{p-1} = 1 \mod p$ and $k^a \neq 1 \mod p$ for all divisors of $p - 1$.

So to provide a certificate to check that p is a prime we can supply a k and a factorization of $p - 1$. However to check the factorization we need to show that the prime factors of $p - 1$ are indeed primes. Therefore we need to provide for every prime factor again an r and a factorization and so on. Now the total number of prime factors we need, will not exceed $\log_2 p$ and as the checking for every prime is polynomial in $\log_2 p$ as well as the exponentiation modulo p, we know that the checking algorithm as a whole is polynomial in $\log_2 p$.

In a similar way one can prove that $\neg\texttt{FACTOR} \in \mathcal{NP}$. Experts in complexity think therefore that is unlikely that factorization is an $\mathcal{NP}$-complete problem: an $\mathcal{NP}$-complete problem in $\mathcal{NP} \cap \text{co}\mathcal{NP}$ would imply that $\mathcal{NP} = \text{co}\mathcal{NP}$.

8.4 Real Life Computation

All of the above discussion has assumed that $\mathcal{P}$ means "easy" and "not in $\mathcal{P}$" means "hard". While this is a common and reasonably accurate assumption in complexity theory, it is not always true in practice, for several reasons:

- It ignores constant factors. A problem that takes time $101000n$ is $\mathcal{P}$ (in fact, it's linear time), but is completely intractable in practice. A

problem that takes time $10^{0,0000002n}$ is not $\mathcal{P}$ (in fact, it's exponential time), but is very tractable for values of n up into the thousands.

- It ignores the size of the exponents. A problem with time n^1000 is P, yet intractable. A problem with time $2^n/10^10$ is not $\mathcal{P}$, yet is tractable for n up into the thousands.
- It only considers worst-case times. There might be a problem that arises in the real world. Most of the time, it can be solved in time n, but on very rare occasions you'll see an instance of the problem that takes time $2n$. This problem might have an average time that is polynomial, but the worst case is exponential, so the problem wouldn't be in $\mathcal{P}$.
- It only considers deterministic solutions. There might be a problem that you can solve quickly if you accept a tiny error probability, but a guaranteed correct answer is much harder to get. The problem would not belong to P even though in practice it can be solved fast. This is in fact a common approach to attack $\mathcal{NP}$-complete problems.
- New computing models such as quantum computers, which also work probabilistically, may be able to quickly solve some problems not known to be in $\mathcal{P}$.

8.5 Trapdoor functions

In secure encryption schemes, the legitimate user is able to decipher the messages (using some private information available to him: the key) So there exists a polynomial time decryction algorithm $\mathcal{D}$ which has as input the coded text and the key. Yet for an adversary (not having this private information) the task of decrypting the ciphertext (i.e., breaking the encryption) should be infeasible. Clearly, the breaking task can be viewn as an $\mathcal{NP}$ problem where the extra information is the key. Yet, the security requirement states that breaking should not be feasible, namely could not be performed by a polynomial-time algorithm.

Hence, the existence of secure encryption schemes implies that there are tasks performed by non-deterministic polynomial-time machines yet cannot

be performed by deterministic polynomial-time machines, which corresponds more or less to $\mathcal{NP}$-problems that are not in $\mathcal{P}$.

However, the above mentioned necessary condition (e.g., $\mathcal{P} \neq \mathcal{NP}$) is not a sufficient one. $\mathcal{P} \neq \mathcal{NP}$ only implies that the encryption scheme is hard to break in the worst case. It does not rule-out the possibility that the encryption scheme is easy to break in almost all cases. In fact, one can easily construct encryption schemes for which the breaking problem is $\mathcal{NP}$-complete and yet there exist an efficient breaking algorithm that succeeds on 99% of the cases. Hence, worst-case hardness is a poor measure of security. Security requires hardness on most cases or at least average-case hardness. Hence, a necessary condition for the existence of secure encryption schemes is the existence of languages in $\mathcal{NP}$ which are hard on the average. Furthermore, $\mathcal{P} \neq \mathcal{NP}$ is not known to imply the existence of languages in $\mathcal{NP}$ which are hard on the average.

Therefore instead of using the class of polynomial time algorithms it is sometimes more interesting to look at probabilistic polynomial time algorithms but this is beyond the scope of the course.

The mere existence of problems which are hard on the average does not suffice. In order to be able to use such problems we must be able to generate such hard instances together with auxiliary information which enable to solve these instances fast. Otherwise, the hard instances will be hard also for the legitimate users and they gain no computational advantage over the adversary. Hence, the existence of secure encryption schemes implies the existence of an efficient way (i.e. probabilistic polynomial-time algorithm) of generating instances with corresponding auxiliary input so that

1. it is easy to solve these instances given the auxiliary input,
2. it is hard on the average to solve these instances (when not given the auxiliary input).

In order to construct secure encryption scheme one can start from special functions meeting these conditions.

The most basic primitive for cryptographic applications is a one-way function. Informally, this is a function which is easy to compute but hard to invert. It is

a function which can be computed using a polynomial time algorithm but any (probabilistic) polynomial time algorithm attempting to invert the one-way function on a element in its range, will succeed with negligible probability. Formalizing this definition requires a lot more statistics and complexity and will be beyond the scope of this course.

Informally, a trapdoor function f is a one-way function with an extra property. There also exists a secret inverse function (the trapdoor) that allows its possessor to efficiently invert f at any point in the domain of his choosing. It should be easy to compute f on any point, but infeasible to invert f on any point without knowledge of the inverse function . Moreover, it should be easy to generate matched pairs of f's and corresponding trapdoor. Once a matched pair is generated, the publication of f should not reveal anything about how to compute its inverse on any point.

Formally, A trapdoor function is a one-way function f such that there exists a polynomial $g(X) \in \mathbb{R}[X]$ and a (probabilistic) polynomial time algorithm I such that for every $k \in \mathbb{N}$ there exists a string $t_k \in \cup_i \{0,1\}^i$, $|t_k| \leq g(k)$ such that $I(f(x), t_k) = x$ for all strings x of length smaller than k.

8.6 A Short List of Candidate One Way and trapdoor functions

As we said above, the most basic primitive for cryptographic applications is a one-way function which is easy to compute but hard to invert. (For public key encryption, it must also have a trapdoor.) By easy, we mean that the function can be computed by a probabilistic polynomial time algorithm, and by hard that any probabilistic polynomial time (PPT) algorithm attempting to invert it will succeed with small probability (where the probability ranges over the elements in the domain of the function.) Thus, to qualify as a potential candidate for a one-way function, the hardness of inverting the function should not hold only on rare inputs to the function but with high probability over the inputs. Several candidates which seem to posses the above properties have been proposed.

Algebra and Number theory provides a source of candidates for one way and

trapdoor functions.

1. The discrete log problem.
2. Factoring numbers and RSA.
3. Word problems in discrete groups like braid groups.
4. DES with fixed message.

In the next chapters we will study those functions and their corresponding cryptosystems.

Chapter 9

The discrete log problem

9.1 The discrete log problem for finite fields

From the theory of finite fields we know that $\mathbb{F}_p^*$ is the cyclic group in $p-1$ elements. This means that there exists a $g \in \mathbb{F}_p^*$ such that every element in $\mathbb{F}_p^*$ can be expressed as a power of g.

The function $f : x \mapsto g^x \mod p$ is a good candidate for being a one-way function. It can be calculated in polynomial time using repeated squaring: to calculate g^x we can look at the binary expansion of $x = x_0 2^0 + \ldots x_{k-1} 2^{k-1}$ and calculate $g^2, g^4, \ldots, g^{2^{k-1}} \mod p$ by repeated squaring g^x is than the product of the g^{2^j} where $x_j \neq 0$. So one need to do less than $2k$ multiplications modulo p. The number op steps to perform a multiplication mod p of 2 numbers smaller than p is polynomial in p so smaller than a given function $C(|p|)$. So the amount of time to perform f is smaller than $2C(|p|)k = 2C(|p|)|x|$ or polynomial.

Given a y and a g in $\mathbb{Z}_p$ finding the u such that $y = g^u \mod p$ is a lot more complicated and this problem is called the discrete logarithm problem or shortly DLP. There are several ways to tackle this problem:

1. **Exhaustive search:** The most obvious algorithm for DLP is to successively compute $g^0, g^1, g^2, \ldots$ until y is obtained. This method takes

$\mathcal{O}(n)$ multiplications, where n is the order of g, and is therefore inefficient if n is large (i.e. in cases of cryptographic interest).

2. **Pollards algorithm:** Construct the following sequence $(x_i)_{i\in\mathbb{N}}$: $x_0 = 1$ and

$$x_{i+1} = \begin{cases} yx_i & \text{if } x_i \mod 3 = 0 \\ gx_i & \text{if } x_i \mod 3 = 1 \\ x_i^2 & \text{if } x_i \mod 3 = 2. \end{cases}$$

x_i can be written as a product of powers of g and y: $x_i = g^{a_i}y^{b_i}$ where $(a_0, b_0) = (0, 0)$ and

$$(a_{i+1}, b_{i+1}) = \begin{cases} (a_i, b_i + 1) & \text{if } x_i \mod 3 = 0 \\ (a_i + 1, b_i) & \text{if } x_i \mod 3 = 1 \\ (2a_i, 2b_i) & \text{if } x_i \mod 3 = 2. \end{cases}$$

Instead of looking at $x_i \mod 3$ we kan take whatever subdivision of $\mathbb{F}_p^*$ in three set of more or less the same size (such that the last one doesn't contain 1.

Find the first i such that $x_i = x_{2i}$. Looking at the exponentials we have that

$$a_i + b_i x = a_{2i} + b_{2i} x \mod p - 1$$

so we can calculate

$$u = \frac{a_i - a_{2i}}{b_{2i} - b_i} \mod p - 1$$

provided $\gcd(b_{2i} - b_i, p - 1) = 1$. If the latter is not the case we simply restart the algorithm with a different $x_0 = g^{a_0}x^{b_0}$. However the probability this happens is small ($\approx \frac{1-\mu(p-1)}{p-1}$) where $\mu(k)$ is the number of invertible elements in the ring $\mathbb{Z}/(k\mathbb{Z})$.

What is the expected size of this i? The sequence (x_i) starts of more or less randomly but repeats itself as soon as one value occurs twice:

$$\exists k, \ell : \forall i \geq k : x_i = x_{i+\ell}.$$

We will call k the headlength of the sequence and ℓ the period. The first i such that $x_i = x_{2i}$ will be $j\ell$ where $(j-1)\ell \leq k \leq j\ell$ so $i \leq k + \ell$.

The probability that $k + \ell > n$ is

$$(1 - \frac{0}{p-1}) \cdot (1 - \frac{1}{p-1}) \cdots (1 - \frac{n-1}{p-1})$$

if $p - 1 \gg n \gg 1$ we can approximate this by

$$e^{-\frac{1}{p-1}} \cdots e^{-\frac{n-1}{p-1}} = e^{-\frac{1}{2}\frac{n(n-1)}{p-1}}$$

Therefore there is around 95% chance that the algorithm will stop before $\sqrt{2 \ln .95(1-p)}$. We can say that the complexity of the algorithm is $\mathcal{O}(\sqrt{p})$.

Example: Pollard's rho algorithm for logarithms in a subgroup of $\mathbb{F}_{383}$. The element $\alpha = 2$ is a generator of the subgroup G order $n = 191$. Suppose $\beta = 228$ The following table shows the values of x_i, a_i, b_i , x_{2i}, a_{2i}, b_{2i} at the end of each iteration. Note that $x_{14} = x_{28} = 144$. Finally, compute $r = b_{14} - b_{28} \mod 191 = 125$, $r^{-1} = 125^{-1} \mod 191 = 136$, and $r^{-1}(a_{28} - a_{14}) \mod 191 = 110$. Hence, $\log_2 228 = 110$.

i	x_i	a_i	b_i	x_{2i}	a_{2i}	b_{2i}
1	228	0	1	279	0	2
2	279	0	2	184	1	4
3	92	0	4	14	1	6
4	184	1	4	256	2	7
5	205	1	5	304	3	8
6	14	1	6	121	6	18
7	28	2	6	144	12	38
8	256	2	7	235	48	152
9	152	2	8	72	48	154
10	304	3	8	14	96	118
11	372	3	9	256	97	119
12	121	6	18	304	98	120
13	12	6	19	121	5	51
14	144	12	38	144	10	104

3. **Index calculus algorithm:** The indexcalculus algorithm is the most powerful method known for computing discrete logarithms. Denote by S the set of the first t primes. Make a list k_i such that all $g^{k_i} \mod p$

products are of factors from S. From this list one can determine the discrete logarithms of the elements in S.

Now we search for an l such that $y^l \mod p$ is also composed of factors from S. Using the discrete logarithms for the elements in S we can determine the discrete logarithm of y.

Example: Let $p = 229$. The element $g = 6$ is a generator of $\mathbb{Z}_{229}^*, \cdots$ of order $n = 228$. Consider $y = 13$. Then $\log_6 13$ is computed as follows, using the indexcalculus technique.

(a) The factor base is chosen to be the first 5 primes: $S = \{2, 3, 5, 7, 11\}$.

(b) The following six relations involving elements of the factor base are obtained (unsuccessful attempts are not shown):

$$\begin{aligned} 6^{100} &\mod 229 = 180 = 2^2 \cdot 3^2 \cdot 5 \\ 6^{18} &\mod 229 = 176 = 2^4 \cdot 11 \\ 6^{12} &\mod 229 = 165 = 3 \cdot 5 \cdot 11 \\ 6^{62} &\mod 229 = 154 = 2 \cdot 7 \cdot 11 \\ 6^{143} &\mod 229 = 198 = 2 \cdot 32 \cdot 11 \\ 6^{206} &\mod 229 = 210 = 2 \cdot 3 \cdot 5 \cdot 7. \end{aligned}$$

These relations yield the following six equations involving the logarithms of elements in the factor base:

$$\begin{aligned} 100 &= 2\log_6 2 + 2\log_6 3 + \log_6 5 (\mod 228) \\ 18 &= 4\log_6 2 + \log_6 11 (\mod 228) \\ 12 &= \log_6 3 + \log_6 5 + \log_6 11 (\mod 228) \\ 62 &= \log_6 2 + \log_6 7 + \log_6 11 (\mod 228) \\ 143 &= \log_6 2 + 2\log_6 3 + \log_6 11 (\mod 228) \\ 206 &= \log_6 2 + \log_6 3 + \log_6 5 + \log_6 7 (\mod 228). \end{aligned}$$

(c) Solving the linear system of six equations in five unknowns (yields the solutions $\log_6 2 = 21, \log_6 3 = 208, \log_6 5 = 98, \log_6 7 = 107$ and $\log_6 11 = 162$.

(d) Suppose that the integer $k = 77$ is selected. Since $y \cdot g^k = 13 \cdot 677 \mod 229 = 147 = 3 \cdot 7^2$, it follows that $\log_6 13 = (\log_6 3 + 2\log_6 7 - 77) \mod 228 = 117$.

Computing the complexity of this algorithm is a boring and complicated task which we will not do in this course, but one can prove that this algorithm is not in polynomial time. As this is the fastest algorithm known we can conclude that $x \to g^x \mod p$ is a valable candidate for a one way function.

4. **Pohlig Hellman algorithm**

   However we have to take care that $p-1$ have some large factors because otherwise we can use the fact that $p-1 = {p_1}^{e_1} \cdots {p_k}^{e_k}$ with $p_k \ll p$ to solve the DLP in a faster way.

   If $x = \log_\alpha \beta$ then x is determined modulo $n = p-1$ and if we know all the $x_i := x \mod p_i^{e_i}$ we can reconstruct x using the chinese remainder theorem.

   Each integer x i is determined by computing the digits $l_0, l_1, \ldots, l_{e_{i-1}}$ in turn of its $p_i$ary representation: $x_i = l_0 + l_1 p_i + \cdots + l_{e_{i-1}} p_i^{e_{i-1}}$ where $0 \le l_j \le p_i - 1$.

   First we calculate $\alpha' = \alpha^{n/p_i}$, this is a generator of the subgroup of order p_i. to to calculate l_0 we can compute

$$\log_{\alpha'} \beta^{n/p_i} = \log_{\alpha'} \alpha^{xn/p_i} = \log_{\alpha'} (\alpha')^{l_0 + k p_i} = l_0.$$

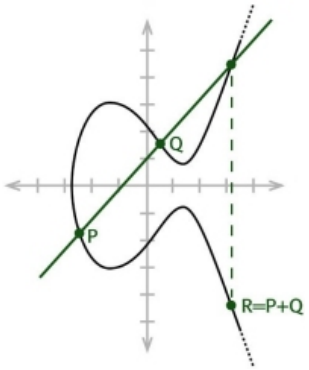
   Knowing $l_0, \ldots, l_{j-1}$ we can compute l_j as

$$\log_{\alpha'} \left(\frac{\beta}{\alpha^{l_0 + l_1 p_i + \cdots + l_j p_i^{j-1}}}\right)^{n/p_i^{j+1}}.$$

   Because $p_i \ll p$ computing these logarithms is much faster for p_i than for p and therefore this algorithm works efficiently when $p-1$ is *smooth* i.e. has only small factors.

9.2 The discrete log problem for elliptic curves

Consider a finite field $\mathbb{F}_q$. An elliptic curve over $\mathbb{F}_q$ is the set of solutions (x, y) to an equation of the form $y^2 = x^3 + Ax + B$, together with an extra point O which is called the point at infinity. This point corresponds to the point on the line at infinity corresponding to the vertical direction $x = 0$.



We usually write E for the equation $y^2 = x^3 + Ax + B$ and use the notation $E(\mathbb{F}_q)$ for the set of points together with O. For the sake of simplicity we will assume that $p \mod 2, 3 \neq 0$.

E is called nonsingular if $x^3 + Ax + B = 0$ has three distinct roots (possibly in a field extension of $\mathbb{F}_q$). This is the same as the condition $4A^3 + 27B^2 \neq 0$. To prove this say that α is a double root so $\alpha^3 + A\alpha + B = 0$ and $(x^3 + Ax + B)'(\alpha) = 3\alpha^2 + A = 0$ these equations imply that $4A^3 + 27B^2 = 0$.

The set of points on an elliptic curve forms a commutative group under a certain addition rule, which we write using the notation $+$. The point O is the identity element of the group. The addition rule can be constructed using the thumb rule that if three points P_1,P_2 and P_3 lie on a line the sum of the points is zero. remember that vertical lines contain the point at infinity and therefore the inverse of a point $P = (x, y)$ is the intersection of the elliptic curve with the vertical line through P and hence $-P = (x, -y)$.

So to add two different points P and Q we first draw the line through them and intersect it with $E(\mathbb{F}_q)$. The point obtained is than $-(P + Q)$ so to get $P + Q$ we simply switch the second coordinate. If $P = Q$ we have to draw the tangent line through P (because this intersects the curve twice in P).

The construction of the sum is always defined because a line that intersects the curve already twice will intersect it also a third time because the degree of E is 3.

The construction is obviously commutative and it is also associative (try this as an exercise).

We give an example: Let E be $y^2 = x^3 + 1$ find $E(\mathbb{F}_5)$. It helps to know the squares in $\mathbb{F}_5$: $0^2 = 0, 1^2 = 1, 2^2 = 4, 3^2 = 4, 4^2 = 1$.

x	x^3+1	y	p
0	1	± 1	$(0,1),(0,4)$
1	2	no	
2	4	± 2	$(2,2),(2,3)$
3	3	no	
4	0	0	$(4,0)$.

So we have 6 points in $E(\mathbb{F}_5)$ and hence the group structure will be $\mathbb{Z}_6$. Over a finite field you can add points using lines or addition formulas. If $G = (2,3)$ then $2G = (0,1)$, $3G = (4,0)$, $4G = (0,4)$ (note it has same x coordinate as $2G$ so $4G = -2G$), $5G = (2,2)$, $6G = O$. So $G = (2,3)$ is a generator of $E(\mathbb{F}_5)$.

Given a generating point $G = (2,3)$ and a public key $(0,4)$ find n such that $nG = (0,4)$ (of course you wouldn't work with such small numbers). This problem is called the elliptic curve discrete logarithm problem (ECDLP) and it is currently harder to solve than the discrete log problem in the nonzero elements of a finite field. Another advantage here is that for a given finite field there can be lots of associated elliptic curves. Take care, it is sometimes possible that $E(\mathbb{F}_q)$ is not a cyclic group, so one has to restrict to a subgroup .

Consider $y^2 = x^3 + 1$ over $\mathbb{F}_7$. $0^2 = 0, (\pm 1)^2 = 1, (\pm 2)^2 = 4, (\pm 3)^2 = 2$.

x	x^3+1	y	p
0	1	± 1	$(0,1),(0,6)$
1	2	± 3	$(1,3),(1,4)$
2	2	± 3	$(2,3),(2,4)$
3	0	0	$(3,0)$
4	2	± 3	$(4,3),(4,4)$
5	0	0	$(5,0)$
6	0	0	$(6,0)$

So $E(\mathbb{F}_7)$ has 12 points.

$$\begin{array}{ll} R = (5,0) & 2R = O \\ Q = (1,3) & Q + R = (2,3) \\ 2Q = (0,1) & 2Q + R = (4,4) \\ 3Q = (3,0) & 3Q + R = (6,0) \\ 4Q = (0,6) & 4Q + R = (4,3) \\ 5Q = (1,4) & 5Q + R = (2,4) \\ 6Q = O, & \end{array}$$

All points are of the form $nQ + mR$ with $n \in \mathbb{Z}_6$ and $m \in \mathbb{Z}_2$. Note that the coefficients of $y2 = x^3 + 1$ and the coordinates of the points are all defined modulo 7, whereas the points add up modulo 6. In this case, two points together generate. You could still use discrete log with $G = (1,3)$ for example. It wouldn't generate all of $E(\mathbb{F}_7)$ but half of it. If q is small Curves sometimes have few points $y2 = x_3 + 4$ over F_7 has only $(0,2), (0,5)$ and O.

In general $E(\mathbb{F}_q), +$ is a commutative group. The main theorem of commutative finite groups is that they can be written as a direct sum of cyclic groups $E(\mathbb{F}_q) = \mathbb{Z}_{p_1^{k_1}} \oplus \cdots \oplus \mathbb{Z}_{p_k^{k_k}}$ with p_i not necessarily different prime numbers.

How many elements $E(\mathbb{F}_q), +$ has, depends highly on the field and on the equation used however we can determine upper and lower bounds on the number of elements. As for any x the number $x^3 + Ax + B$ has at most 2 squer roots we know that $|E(\mathbb{F}_q)| \leq 2q + 1$ (the one comes from the point at infinity). Getting a lower bound for the number of

points is a far more difficult task. Hasse proved that the number of points in an elliptic curve satisfy

$$|q + 1 - |E(\mathbb{F}_q)|| \leq 2\sqrt{q}.$$

So in the case of $q = 7$, $3 \leq E(\mathbb{F}_7) \leq 13$.

To solve the Elliptic curve discrete logarithm problem one can use similar techniques as for the ordinary DLP. We can easily generalize Pollard's algorithm by choosing a subdivision of $E(\mathbb{F}_q)$ in 3 parts. However we can not apply the methods based on factor bases because we don't have something similar like small primes in $E(\mathbb{F}_q)$. Therefore ECDLP is a more complex problem than ordinary DLP.

9.3 Applications in Cryptography

9.3.1 Diffie Helman Key Exchange

Many cryptosystems depend on a secret key shared by the transmitter (A) and the receiver (B) of the message. This key must be exchanged and during this exchange a third party (E) might intercept this key and will then be enabled to decode the secret messages that will be sent later on. To overcome this problem one has to design a method for two people to agree on a common key such that a third party cannot retrieve this key. The Diffie-Hellman secret key exchange protocol enables us to do just this.

We fix a prime p and a generator $g \in \mathbb{F}_p^*$. These are public, and known not only to all parties but also to the adversary. Then

- A picks $x \in \mathbb{N}$ at random and lets $X = g^x \mod p$. She sends X to B,
- B picks $y \in \mathbb{N}$ at random and lets $Y = g^y \mod p$. He sends Y to A.

Now notice that

$$X^y = (g^x)^y = g^{xy} = (g^y)^x = Y^x$$

the operations being in the field $\mathbb{F}_p$. Let's call this common quantity K. The crucial fact is that both parties can compute it! Namely A computes Y^x , which is K, and B computes X^y , which is also K, and now they have a shared key.

Is this secure? Consider an adversary that is sitting on the wire and sees the flows that go by. She wants to compute K. What she sees is X and Y . But she knows neither x nor y. How could she get K? The natural attack is to find either x or y (one of them will do!) from which she can easily compute K. However, notice that computing x given X is just the discrete logarithm problem in $\mathbb{F}_p$ which is widely believed to be intractable (for suitable choices of the prime p). Similarly for computing y from Y . Accordingly, we would be justified in having some confidence that this attack would fail.

A number of issues now arise. The first is that computing discrete logarithms is not the only possible attack to try to recover K from X, Y . Perhaps there

are others. To examine this issue, let us formulate the computational problem the adversary is trying to solve. It is the following:

The DH Problem: Given g^x and g^y for x, y chosen at random from $\mathbb{F}_p^*$, compute $g^x y$.

Thus the question is how hard is this problem? We saw that if the discrete logarithm problem in $\mathbb{F}_p^*$ is easy then so is the DH problem, ie. if we can compute discrete logs we can solve the DH problem. Is the converse true? That is, if we can solve the DH problem, can we compute discrete logarithms? This remains an open question. To date it seems possible that there is some clever approach to solve the DH problem without computing discrete logarithms. However, no such approach has been found. The best known algorithm for the DH problem is to compute the discrete logarithm of either X or Y. This has lead cryptographers to believe that the DH problem, although not known to be equivalent to the discrete logarithm one, is still a computationally hard problem, and that as a result the DH secret key exchange is secure in the sense that a computationally bounded adversary can't compute the key K shared by the parties.

These days the size of the prime p is recommended to be at least 512 bits and preferably 1024. As we have already seen, in order to make sure the discrete logarithm problem modulo p is intractable, $p-1$ should have at least one large factor. In practice we often take $p = 2q+1$ for some large prime q. The relationship between the DH problem and the discrete logarithm problem is the subject of much investigation.

Miniature 6: Diffie, Hellman, Merckle

The first researchers to discover and publish the concepts of PKC were Whitfield Diffie and Martin Hellman from Stanford University, and Ralph Merkle from the University of California at Berkeley. As so often happens in the scientific world, the two groups were working independently on the same problem – Diffie and Hellman on public-key cryptography and Merkle on public key distribution – when they became aware of each other's work and realized there was synergy in their approaches.

9.3.2 El Gamal and Digital Signature Algorithm

Digital signatures are a method of authenticating digital information analogous to ordinary physical signatures on paper, but implemented using techniques from the field of cryptography.

Digital signature schemes rely on two keys for each user: one public and one private. The public key is distributed freely, but the private key is kept secret and confidential; another requirement is that it should be infeasible to derive the private key from the public key.

Consider the situation in which Bob sends a message to Alice and wants to be able to prove it came from him. In this case, Bob sends a message to Alice, with a digital signature attached. The digital signature is generated using Bob's private key, and takes the form of a simple numerical value, normally represented as a string of bits. On receipt, Alice can then check whether the message really came from Bob by running a verification algorithm on the message together with the signature and Bob's public key. If they match, the message was really from Bob, because the private key was needed to create the signature and no one but Bob has it.

More usually, for efficiency reasons, Bob first applies a cryptographic hash function to the message before signing. This makes the signature much shorter and thus saves time since hashing is generally much faster than signing in implementations. However, if the message digest algorithm is insecure (for example, if it is possible to generate collisions), then it might be feasible to forge digital signatures.

A general digital signature scheme was devised in 1984 by T. El Gamal based on discrete logarithms. The scheme is closely related to the Diffie-Hellman technique. Below we outline the protocol:

1. *Global Public Elements.* As with Diffie-Hellman, to generate a key pair, first choose a prime number q and g, a generator of $\mathbb{F}_q^*$.

2. *Key Generation.* Alice selects a private key $x_a < q$ and calculate a public key y_a as in the Diffie Hellman protocol: $y_a = g^{x_a}$.

   Independently, Bob also generates his public key and private key, x_b, y_b.

3. *User A Signs a Message.* Alice encrypts a plaintext $M < q$ intended for Bob as follows:

   (a) Choose a random integer k, $1 < k < q$,

   (b) Compute: $K = (y_b)^k \mod q$ and (C_1, C_2) where:

   $$C_1 = g^k \mod q,\ C_2 = KM \mod q$$

   These two numbers together make up the signature.

4. *User B Verifies the Signature* Bob verifies the signature by recovering the plaintext M as follows:

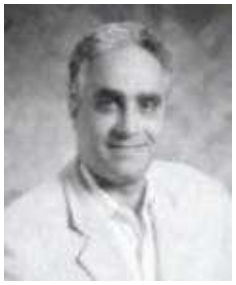
   (a) Compute $K = C_1^{x_b} \mod q$. Which is $(g^k)^{x_b} \mod q = y_b^k \mod q$.

   (b) Compute $M = (C_2 K^{-1}) \mod q$, where K^{-1} is the multiplicative inverse of K. Therefore: $(C_2 K^{-1}) \mod q = (KMK^{-1}) \mod q = M \mod q$.

This scheme is sometimes referred to as DSA stands for Digital Signature Algorithm.

The plaintext M is usually a digest of a message. It is seen that DSS does not encrypt the digest. The input to the algorithm is the digest of the data to sign, M, the key, y_b and a random number, k. The output is a pair of numbers C_1, and C_2. There will be many ciphertexts that are encryptions of the same digest, since the output depends on both the digest M and on the random value k chosen by Alice.

Miniature 7: Taher El-Gamal



Taher El-Gamal developed the above algorithm in the early 1980's while at Stanford. El-Gamal algorithms use similar principles to those used in the Diffie-Hellman protocols. El-Gamal schemes are simple, efficient, and not patented. This is why theDSS - the US government standard for signatures - is based on El-Gamal.

Chapter 10

Braid group Cryptography

Braids arose as a combinatorial tool in the study of knots and since then have been applied in many branches of mathematics. In this chapter we will give an introduction to this subject together with its applications to cryptography.

10.1 knots

A knot is in fact a smooth embedding of the circle into the three dimensional real space, where both are considered as manifolds. The set of knots is denoted by $\mathcal{K} := \{K : \mathbb{S}_1 \to \mathbb{R}^3 | K \text{ is a knot}\}$.

Two knots $K_1, K_2 \in \mathcal{K}$ are equivalent if they can be deformed into each other

$K_1 \cong K_2 \Leftrightarrow \exists \phi : \mathbb{S}_1 \times [0,1] \to \mathbb{R}^3$ such that $\phi|_0 = K_1, \phi|_1 = K_2, \forall i \in [0,1] \phi|_i \in \mathcal{K}$.

Using the equivalence relation allows us to look at the image of the map instead of the map itself: if K_1 and K_2 have the same image $K_1 \cong K_2$ or $K_1 \cong -K_2$ where $-K_2$ is the map K_2 composed with $\mathbb{S}_1 \to \mathbb{S}_1 : \theta \mapsto -\theta$. Therefore to define a knot one simply needs its image and its orientation (i.e. the direction in which θ increases).

The main question in knot theory is then: given two knots how can one find out wether they are equivalent or not. In general this question is to general

to solve easily, so one can also ask first whether a knot is really a knot or not. A knot is called trivial if it is equivalent to the standard embedding of the unit circle $C : \theta \mapsto (\cos\theta, \sin\theta, 0)$.

The tactic to solve this question is to find a knot invariant. This is a map $\mathcal{I} : \mathcal{K} \to A$ where A is an algebra, or a group or something like that. $\mathcal{I}$ must be easy to compute from the knot and $K_1 \cong K_2 \Rightarrow \mathcal{I}(K_1) = \mathcal{I}(K_2)$. So a knot K is definitely not trivial if $\mathcal{I}(K) \neq \mathcal{K}(C)$. The reverse implication that K is trivial if $\mathcal{I}(K) = \mathcal{K}(C)$ is however not always true.

The construction of knot invariants can be seen as a purely algebraic problem using the theory of braids.

10.2 Braids

An n-braid is a set of n continuous maps $s_i : [0,1] \to \mathbb{R}^2$ such that $s_i(0) = (i, 0), s_i(1) = (\pi(i), 0)$ (where π is a random permutation) and $\forall i, j : \forall t \in [0,1] : s_i(t) \neq s_j(t)$. A braid can be represented in three dimensions by taking the interval $[0,1]$ as third coordinate.

Two braids (s_i) and (t_i) are called equivalent if their exist n continuous maps $\varpi_i : [0,1] \times [0,1] \to \mathbb{R}^2$ such that $\varpi_i|_0 = s_i$, $\varpi_i|_1 = t_i$ and $(\varpi_i|_\mu)$ is a braid for every $\mu \in [0,1]$.

One can also compose two braids (s_i) and (t_i) to obtain a third braid

$$(s \cdot t)_i : [0,1] \to \mathbb{R}^2 : \lambda \mapsto \begin{cases} s_i(2\lambda) & \lambda \leq \frac{1}{2} \\ t_i(2\lambda - 1) & \lambda > \frac{1}{2} \end{cases}$$

Let B_n be the set of n-braids up to equivalence. Because the composition of braids will be compatible with the equivalence relation, $\cdot$ can be considered as an operation on B_n. We have even that:

Theorem 10.1. $\mathsf{B}_n, \cdot$ is a non-commutative group.

Proof. The group is associative because if (s), (t), (u) are braids then

$$\varpi_i : [0,1]\times[0,1] \to \mathbb{R}^2 : (\lambda,\mu) \begin{cases} s_i(\frac{4}{1+\mu}\lambda) & \lambda \leq \frac{1}{4}(1+\mu) \\ t_i(4(\lambda - \frac{1}{4}(1+\mu))) & \frac{1}{4}(1+\mu) < \lambda \leq \frac{1}{4}(2+\mu) \\ u_i(\frac{4}{2-\mu}(\lambda - \frac{1}{4}(2+\mu))) & \lambda > \frac{1}{4}(2+\mu) \end{cases}$$

defines an equivalence between $(s \cdot t) \cdot u$ and $s \cdot (t \cdot u)$.

The identity element are the constant maps $e_i(\lambda) = (i, 0)$ and the inverse of (s) is the braid $s_i^{-1}(\lambda) = s_i(1-\lambda)$. (try these as an exercise) □

The group B_n is finitely generated by $n-1$ standard braids:

$$(\sigma_k)_i(\lambda) = \begin{cases} (i+\lambda, \sin(\pi\lambda)) & i = k \\ (i-\lambda, -\sin(\pi\lambda)) & i = k \\ i & i \neq k, k+1 \end{cases}$$

For all $1 \leq k \leq n-1$

Two braids σ_k and σ_l commute as soon as $|k-l| > 1$. For σ_k and $\sigma_k + 1$ we have the relation $\sigma_k\sigma_{k+1}\sigma_k = \sigma_{k+1}\sigma_k\sigma_{k+1}$.

A lenghty computation shows that every other relation can be derived from the relations above:

Theorem 10.2. $\mathsf{B}_n, \cdot = \langle \sigma_k,\ 1 \leq k \leq n-1 | \sigma_k\sigma_l = \sigma_l\sigma_k,\ \sigma_k\sigma_{k+1}\sigma_k = \sigma_{k+1}\sigma_k\sigma_{k+1}, |k-l| > 1 \rangle$

As every braid s defines a permutiation on $\{1, \ldots, n\}$ simply by mapping i to the first coordinate of $s_i(1)$. Therefore we have a canonical group morphism from $\mathsf{B}_n \to \mathfrak{S}_n$.

How can we use the theory of braids to construct knot invariants? The trick to do this is the concept of the closure of a braid.

If s is a braid we define its closure the following subset of $\mathbb{R}^3$

$$\bar{s} = \cup_i \{(e^y \cos\theta, e^y \sin\theta, x) | s_i(\theta) = (x, y)\}$$

In general this is not the image of a knot because it can have different connected components (which is called a link, equivalence of links if defined analoguously). If the permutation of s is cyclic (i.e. $\forall j : \exists m : \pi^m(1) = j$) $\bar{s}$ has only one component can indeed be considered as a knot by giving it the orientation anti-clockwise around the z-axis.

Different braids can have equivalent closures and there are two possibilities to construct braids with equivalent closures.

1. As the closure connects the end points of the braids $s \bar{\cdot} t$ will be equivalent to $t \bar{\cdot} s$ and similarly $t^{-1} \bar{\cdot} s \cdot t \cong s \cdot t \bar{\cdot} t^{-1} \cong \bar{s}$. The closure is thus invariant under conjugation.

2. The braid group B_n is canonically included in B_{n+1} by identifying the first $n-1$ σ_i. B_{n+1} has then one extra generator σ_n. If $s \in \mathsf{B}_n$ then $s\bar{\sigma}_n \cong \bar{\sigma}$.

These two operations are called markov moves.

Theorem 10.3. If $s \in B_n$ and $t \in B_m$ and $\bar{s} \cong \bar{t}$ then s can be transformed to t using the two markov moves and their inverses.

In this way the knot invariants problem can be transformed to the question. Find maps $\phi_n : B_n \to A$ such that the ϕ_n are invariant under the two markov moves.

A usefull tool to do this are group representations:

A group representation of G is a group morphism

$$\phi : \mathsf{G} \to \mathsf{GL}_n(R)$$

Where R is a commutative ring and $\mathsf{GL}_n(R)$ is the group of invertible $n \times n$-matrices.

If ϕ is a representation then we can take the trace of the representation which is the sum of the diagonal elements of the matrix. As the trace is invariant under conjugation, the map $\mathtt{Tr}\phi : \mathsf{B}_n \to R$ will be invariant under

the first markov move. So we only need to find a sequence of representations $\phi_n : \mathsf{B}_n \to R$ such that

$$\mathtt{Tr}\phi_{n+1}(s\sigma_n) = \mathtt{Tr}\phi_n(s).$$

Instead of using the trace one can also use other conjugation invariant function like determinants and traces of powers. However the map $\mathtt{Det}\phi_n$ will not be interesting because invariance under the second markov move implies that $\mathtt{Det}\phi = 1$. $\mathtt{Det}(f(\phi))$ where $f(X)$ is a polynomial will be more usefull.

Here are two interesting constructions.

- Let Λ denote the ring $\mathbf{Z}[t, t^{-1}]$. The Burau representation $B_n \to GL_n(\Lambda)$ sends the i-th generator $\sigma_i \in B_n$ into the matrix

$$I_{i-1} \oplus \begin{pmatrix} 1-t & t \\ 1 & 0 \end{pmatrix} \oplus I_{n-i-1}$$

  where I_k denotes the identity $(k \times k)$-matrix and the non-trivial (2×2)-block appears in the i-th and $(i+1)$-th rows and columns. Substituting $t = 1$, we obtain the standard representation of the symmetric group S_n by permutation matrices. The Burau representation is reducible: it splits as a direct sum of an $(n-1)$-dimensional representation (the reduced burau representation) and the trivial one-dimensional representation $B_n \mapsto 1$. the trivial 1 dimensional subrepresentation corresponds to the invariant subspace $\mathbb{C}(1, \ldots, 1)$ and the reduced representation Ψ_n acts on the $n-1$-dimensional subspace $V = \{(a_1, \ldots, a_n) | a_1 + ta_2 + \cdots + t^{n-1}a_n = 0\}$.

  This reduced representation can be used to construct a knot invariant called the alexander polynomial

$$\Delta := \mathtt{Det}(1 - \Psi_n)/(1 + t + t^2 + \cdots + t^{n-1})$$

- We denote by $\mathtt{Ref} = \mathtt{Ref}_n$ the set of pairs of integers (i, j) such that $1 \leq i < j \leq n$. Clearly, $\#(\mathtt{Ref}) = n(n-1)/2$.

  Let R be a commutative ring with unit and $q, t \in R$ be two invertible elements. Let $V = \oplus_{s \in \mathtt{Ref}} Rx_s$ be the free R-module of rank $n(n-1)/2$

with basis $\{x_s\}_{s\in \mathtt{Ref}}$. Krammer [Kr2] defines an R-linear action of B_n on V by

$$\sigma_k(x_{i,j}) = \begin{cases} x_{i,j} & k < i-1 \text{ or } j < k, \\ x_{i-1,j} + (1-q)x_{i,j} & k = i-1, \\ tq(q-1)x_{i,i+1} + qx_{i+1,j} & k = i < j-1, \\ tq^2 x_{i,j} & k = i = j-1 \\ x_{i,j} + tq^{k-i}(q-1)^2 x_{k,k+1} & i < k < j-1, \\ x_{i,j-1} + tq^{j-i}(q-1)x_{j-1,j} & k = j-1, \\ (1-q)x_{i,j} + qx_{i,j+1} & k = j. \end{cases}$$

where $1 \leq i < j \leq n$ and $k = 1, ..., n-1$. That the action of σ_k is invertible and that the braid relations are satisfied should be verified by a direct computation.

This representation is interesting because it is faithfull for every n. This means that if two braids $s, t \in B_n$ are different the corresponding $\mathtt{Ref}(s)$ and $\mathtt{Ref}(t)$ will also be different. In the next sections we will see how one can use this to analyse cryptosystems based on braid groups.

10.3 Cryptosystems based on braid groups

To work with braids on a computer one first has to establish a method to write down the elements of the braid group in a unique normal form. It should also be easy to multiply two braids in normal form and then calculate the normal form of the product.

The solution to the word problem in braid groups arises from the existence of a unique normal form for each word in B n and this in turn

makes the representation of braids by computers easy. In each B_n we can define a fundamental braid to be $W_n = W_{n-1}\sigma_{n-1}\sigma_{n-2}\ldots\sigma_1$ with $W_0 = 1$

For example, the fundamental braid $W_4 = \sigma_1\sigma_2\sigma_1\sigma_3\sigma_2\sigma_1$. With W_n (or just when no confusion arises), we can express any word w in B_n as a product of W_n to some integer exponent and braids $F_1, F_2, \ldots, F_p$, each of which is a subword of W_n. It should be noted that the positive integer p generally differs

from word to word in B_n . We can then represent w on a computer by the tuple $[u, f_1, f_2, \ldots, f_p]$ where u is the integer exponent of W_n and each f_i is the permutation that results from mapping F_1 via the group homomorphism from B_n onto the group of permutations on the set $\{1, 2, , n\}$ given by $\sigma_i \mapsto (i, i{+}1)$.

Note this homomorphism is equivalent to adding the relations $\sigma_i^2 = 1$ for each generator σ_i in B_n . Thus if $n = 4$ and w has normal form $W_4^3\sigma_1\sigma_3\sigma_1$ then $u = 3$, $\sigma_1\sigma_3\sigma_1$ is not a subword of W_4 but $\sigma_1\sigma_3$ and σ_1 are so $F_1 = \sigma_1\sigma_3$ and $F_2 = \sigma_1$ (so $p = 2$.) Thus f_1 is the permutation $(12)(34)$, i.e. the permutation that interchanges 1 with 2 and 3 with 4, and f_2 is the permutation (12). The braid w can then be represented on computer as $[3, (12)(34), (12)]$ and for any word, v say, in B_n , $v = w$ if and only if v has the same representation.

The ease of representation of braids on computers also allows for fast algorithms to perform group operations in B_n. The product of words w and v in B_n is just the concatenation wv. Furthermore, by reducing wv to its normal form the factors w and v are then hard to recover; much the same way the prime factors of large integers are hard to find. This further implies that functions on B_n such as f: $B_n^2 \to B_n^2$ defined by $f(a, x) = (x, axa^{-1})$ are one-way functions because when axa^{-1} is reduced to its normal form it is hard to recover the group element a even when we know x.

As n grows in the braid group B_n , the computation of group operations becomes hard in $\mathcal{O}(n \ln n)$. On the other hand, a naive computation of one-way function seems to be at least $\mathcal{O}(n!)$. Consequently, n plays a reliable role of a security parameter.

One final property of B_n that makes the cryptosystem possible are the relations $\sigma_i\sigma_j = \sigma_j\sigma_i$, if $|i - j| > 1$. This means for any positive integers $l >= 2$ and $r >= 2$, such that $l + r = n$, the generators $\sigma_1, \sigma_2, \ldots, \sigma_{l-1}$ will commute with generators $\sigma_{l+1}, \sigma_{i+2}, \ldots, \sigma_{i+r-1}$. Thus for B_n , we can define the subgroups LB_l , and RB_r of words in $\{\sigma_1, \sigma_2, \ldots, \sigma_{l-1}\}$ and $\sigma_{l+1}, \ldots, \sigma_{l+r-1}\}$ respectively and so every word in LB_l commutes with every word in RB_r . This property allows for the following protocol for key exchange as described in chapter three.

Let Alice choose words x from B_n and a from LB_l, compute axa^{-1} , and transmit (x, axa^{-1}) to Bob. Next let Bob choose b from RB_r , compute bxb^{-1} and transmit bxb^{-1} to Alice. Alice can then compute the key $K =$

$a(bxb^{-1})a^{-1}$ and Bob can compute $b(axa^{-1})b^{-1} = baxa^{-1}b^{-1} = abxb^{-1}a^{-1} = K$. Thus both have the common key K but each retains their private keys a and b, respectively.

10.4 Attacks on the Braid Cryptosystem

Since the braid cryptosystem is relatively new, only a few attacks on the cryptosystem have been conceived. The most obvious negative property, for cryptographic purposes, of braid groups which could possibly be exploited in an attempt to crack the cryptosystem is the fact proven in that braid groups are linear. This means that braids can also be represented by matrices over some number field so that the considerable research into properties of matrix groups could be used break the code.