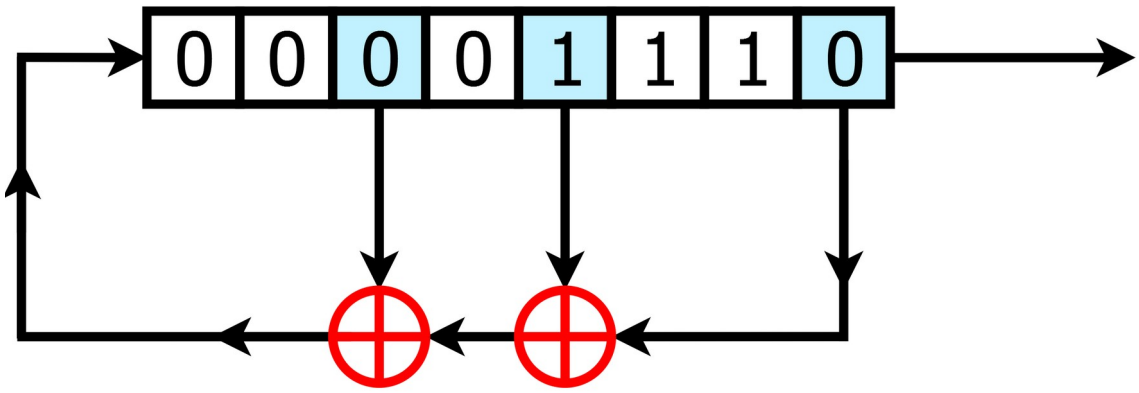
Calcul Symbolique – LFSR



Projet par Léo Lovicourt et Jordan Elie

I – Travail à réaliser

Nous avions pour objectif d'implanter des fonctions de manipulation des registres à décalage à rétroaction linéaire (LFSR : Linear Feedback Shift Register) en OCaml. Un LFSR est un dispositif permettant de produire une suite de bits ultimement périodique à coefficient dans F2, F2 étant un corps à deux éléments dont les lois de composition sont définies ainsi :

$\oplus$	0	1
0	0	1
1	1	0

$\otimes$	0	1
0	0	0
1	0	1

Les LFSR sont utilisés en cryptographie pour engendrer des suites de nombres pseudo-aléatoires. Ils sont définis par la suite récurrente :

$$\begin{cases} r_0, \ldots, r_{\ell-1} \in \mathbb{F}_2 \\ r_n = \alpha_1 \otimes r_{n-1} \oplus \cdots \oplus \alpha_\ell \otimes r_{n-\ell} \quad \text{si } n \geq \ell \end{cases}$$

où l > 0 est la longueur du LFSR. Les valeurs $\alpha_i \in$ F2 sont appelés branchements.

La finalité de ce projet était la réalisation de fonctions de chiffrement et de déchiffrement à l'aide de LFSR.

II – Développement et preuves

Préambule

Nous avons choisi de diviser notre projet en 4 modules distincts.
Le fichier **utils.ml** contient des fonctions utilisées régulièrement dans les fonctions principales du projet.
Le fichier **polynom.ml** contient les principales fonctions liées aux polynômes à coefficient dans F2 ainsi que quelques fonctions auxiliaires.
Le fichier **lfsr.ml** contient la quasi totalité des fonctions de manipulation des LFSR ainsi que différentes fonctions d'affichage, dont la spécification est donnée à la fin du code.
Enfin, le fichier **test.ml** contient uniquement une série de tests des fonctions réalisées au cours du projet.
Pour exécuter les tests, il faut interpréter les fichiers dans l'ordre suivant : **utils.ml**, puis **polynom.ml**, puis **lfsr.ml** et enfin **test.ml**.

Exercice 1

On montre d'abord que **(F2, ⊕)** est un groupe abélien.
La loi ⊕ doit donc être associative, commutative, posséder un élément neutre et chaque élément de F2 doit avoir son symétrique pour cette loi.

- Associativité :

A	B	C	A ⊕ B	B ⊕ C	(A ⊕ B) ⊕ C	A ⊕ (B ⊕ C)
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	0	0	0
1	0	0	1	0	1	1
1	0	1	1	1	0	0
1	1	0	0	1	0	0
1	1	1	0	0	1	1

⊕ est associative.

- Commutativité :

0 ⊕ 0 = 0
0 ⊕ 1 = 1
1 ⊕ 0 = 1
0 ⊕ 0 = 0

⊕ est commutative.

Le neutre de la loi ⊕ est 0, car 0 ⊕ 1 = 1 et 0 ⊕ 0 = 0.
Enfin, le symétrique de 1 est 1 et celui de 0 est 0.
(F2, ⊕) est donc un groupe abélien de neutre 0.

Montrons maintenant que **(F2\{0}, ⊗)** est un groupe.
La loi ⊗ doit donc être associative, posséder un élément neutre et chaque élément de F2\{0} doit avoir son symétrique pour cette loi.

- Associativité

A	B	C	A ⊗ B	B ⊗ C	(A ⊗ B) ⊗ C	A ⊗ (B ⊗ C)
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

⊗ est associative.

Le neutre de ⊗ est 1 car 0 ⊗ 1 = 0 et 1 ⊗ 1 = 1.
Le symétrique de 1 est 1 car 1 ⊗ 1 = 1.

(F2\{0}, ⊗) est donc un groupe.

Il reste à montrer que ⊗ est distributive par rapport à ⊕ :

On a d'une part :

A	B	C	A ⊗ B	A ⊗ C	A ⊗ (B ⊕ C)	A ⊗ B ⊕ A ⊗ C
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	0	1	1	1
1	1	0	1	0	1	1
1	1	1	1	1	0	0

Donc A ⊗ (B ⊕ C) = A ⊗ B ⊕ A ⊗ C.

Et d’autre part :

A	B	C	A ⊕ B	A ⊗ C	B ⊗ C	(A ⊕ B) ⊗ C	(A ⊗ C) ⊕ (B ⊗ C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	1	1	1
1	0	0	1	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	0	0	0	0	0
1	1	1	0	1	1	0	0

Donc (A ⊕ B) ⊗ C = (A ⊗ C) ⊕ (B ⊗ C).

F2 est donc un corps commutatif.

On constate également que la loi ⊕ s'apparente à l'opération logique "ou exclusif" et que la loi ⊗ s'apparente à l'opération logique "et".
En effet, ces opérations logiques ont la même table de vérité.

Exercice 2

La structure de donnée que nous avons choisi afin de représenter les polynômes à coefficient dans F2 est une liste d’entier, et nous avons nommé ce type **f2poly**.
Comme en TP, nous utilisons la représentation creuse, et les valeurs contenues dans la liste représentent les degrés où le coefficient vaut 1.
Ces valeurs sont rangées dans l’ordre croissant usuel sur les entiers.

Par exemple, le polynôme $1 \oplus X \oplus X^3$ sera représenté par la liste [0 ; 1 ; 3].
Nous avons estimé que cette structure était la plus optimale, car d’une part nous n’avons plus à nous préoccuper du coefficient : si une valeur **i** est présente dans la liste, alors le coefficient du monôme $\mathbf{X^i}$ est forcément 1. En revanche, si cette valeur est absente, alors ce coefficient vaut 0.

Nous avons par la suite adapté les fonctions réalisant la somme, le produit par la méthode de Karatsuba et la division rapide par la méthode de Newton de polynômes afin que ces opérations soient réalisés avec des coefficients dans F2. Nous

avons également implanté l'algorithme d'Euclide afin qu'il puisse calculer le PGCD de deux polynômes à coefficient dans F2.

Ensuite, nous avons développé une fonction permettant de calculer l'ordre d'un polynôme. Celle-ci renvoie une exception si le terme constant n'est pas 1, si le polynôme est nul ou si le degré est inférieur ou égal à 1. Sinon, elle effectue la division du monôme $\mathbf{X^n}$ par le polynôme désiré. Si le reste est égal à 1, alors l'ordre de ce polynôme est n. Sinon, on effectue un appel récursif à ce procédé avec le monôme $\mathbf{X^{n+1}}$.

Nous avons ensuite développé une fonction permettant de vérifier si un polynôme est irréductible. Pour cela, nous avons tout d'abord 2 fonctions auxiliaires (situées dans **polynom.ml** car elles font appel à des fonctions définies dans ce fichier).
La fonction **apply_sum** parcourt les éléments de la liste **list**, et ajoute le monôme **x** à chacun d'eux avec la fonction **sum_poly** (somme de deux polynômes à coefficient dans F2).

Celle-ci est utilisée par la seconde fonction auxiliaire **all_poly_below** qui renvoie tous les polynômes de F2[X] de degré inférieur à **n**.
C'est cette fonction qui sera utilisée dans la fonction **est_irreductible** qui renvoie **true** si et seulement si le polynôme **p** est irréductible.

Afin de générer un polynôme irréductible de degré **n**, on commence par appeler dans notre fonction **irreductible** la fonction **all_poly_below** avec comme paramètre **n+1** afin d'obtenir tous les polynômes de degré inférieur ou égal à n. On parcourt ensuite cette liste, en ne gardant que ceux dont le degré est **n** et on teste alors leur irréductibilité avec la fonction créée précédemment.

Enfin, pour générer un polynôme primitif de degré **n**, on effectue dans la fonction **primitif** les mêmes opérations que pour la fonction **irreductible**, avec pour seule différence une condition supplémentaire : en effet, le polynôme doit être de degré **n**, irréductible, mais également avoir pour ordre $2^n - 1$.

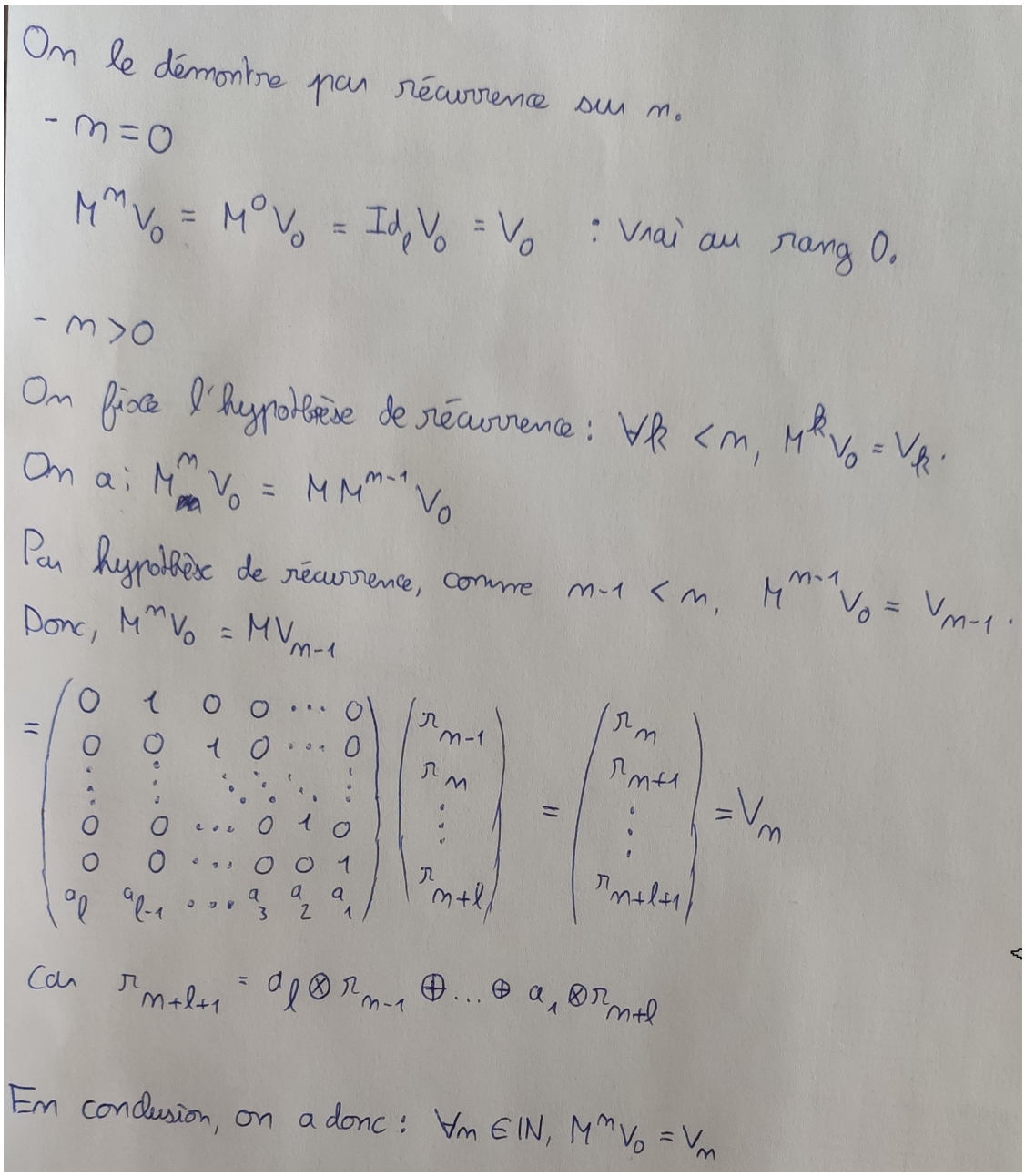
Exercice 3

Nous avons opté pour une structure de donnée contenant 3 champs. Le champ **length** est un entier représentant la longueur du LFSR. Le champ **base** représente les valeurs initiales de la suite r_i du LFSR sous forme de liste d'entiers. Enfin, le champ **branch** représente les branchements du LFSR, sous forme de liste d'entiers également.
Pour cette structure, nous avons fait le choix de garder des liste de coefficients 0 et 1 au lieu d'opter pour des listes similaires à cette utilisée pour les polynômes car elles

semblaient plus faciles à manipuler pour effectuer des opérations sur des bits. Nous avons cependant créé deux fonctions (**binary_to_poly** et **poly_to_binary**) afin de passer d'un type de liste à l'autre.
Enfin, nous avons défini les opérations +! et *!, représentant les lois de composition de F2.

Par la suite, nous avons implanté le calcul de la n-ième valeur r_n d'un LFSR. Pour cela, nous avons d'abord la fonction **calc** qui calcule la valeur r_i selon les valeurs de branchements ainsi qu'une liste contenant les valeurs de la suite nécessaires au calcul de r_i. Cette fonction est utilisée dans la fonction **lfsr_value**, qui calcule la **n**-ième valeur du LFSR **lfsr**.

On établit la preuve que $M^nV_0 = V_n$ par récurrence sur n.



On montre maintenant qu'un LFSR produit une suite ultimement périodique, de période inférieure à $2^L - 1$ avec L la longueur du LFSR.
Tout d'abord, on sait que chaque état d'un LFSR quelconque de longueur L est représenté par un vecteur de L éléments. Il y a donc un total de 2^L états possibles.
On remarque que la suite des valeurs produites par le LFSR n'est pas périodique dans le cas général, puisque les valeurs produites dépendent d'une part des valeurs initiales, et des branchements. Cette suite est donc, dans le cas général, au plus ultimement périodique.

On veut donc montrer que cette suite est ultimement périodique, de période inférieure à $2^L - 1$.
Cela revient à prouver qu'il existe $p \in \mathbb{N}$, $1 \leq p \leq 2^L - 1$, tel que pour tout $i \in \mathbb{N}$ et $i \geq n_0$ (borne à partir de laquelle la suite est périodique), $V_i = V_{i+p}$.
On suppose dans un premier temps que p n'est pas une période : pour tous k, $p \in \mathbb{N}$, $V_{k+p} \neq V_k$. Or, ceci est impossible puisqu'un LFSR possède un nombre d'états fini de 2^L. p est donc une période : il reste à prouver que $p \leq 2^L - 1$.
Soit $k \in \mathbb{N}$, et V_k un état du LFSR. Il reste alors $2^L - 1$ états possibles du LFSR différents de V_k. Si V_k est le vecteur nul, alors $V_{k+1} = V_{k+2} = V_{k+n} = V_k$ ($n \in \mathbb{N}$). Donc, la période p vaut 1 et $1 \leq 2^L - 1$ car $L \geq 1$.
Sinon, il reste alors $2^L - 2$ états possibles car si jamais l'état représenté par le vecteur nul est atteint, on ne peut atteindre un autre état par celui-ci : on revient donc au cas précédent. Donc, p vaut au plus $2^L - 1$ puisqu'on a éliminé la possibilité d'atteindre l'état nul sans en « ressortir ».
On a donc montré que la suite de valeurs produites par un LFSR de longueur L est ultimement périodique, de période inférieure à $2^L - 1$.

On souhaite maintenant montrer que si $\alpha_L = 1$, la suite est périodique. Pour cela, on étudie le déterminant de la matrice M. On constate après calcul, que pour un LFSR de longueur L, le déterminant **d** de la matrice M vaut $(-1)^{L+1}$. Comme $\mathbf{d} \neq 0$, M est inversible et donc on peut lui associer une application bijective. Soit V_k un état du registre, $k \in \mathbb{N}$. Lorsqu'on multiplie M par cet état, on a 2 cas possibles.
- Si $MV_k = V_k$, on a une période de 1, et ce dès l'état V_0.
- Sinon, il existe $j \in \mathbb{N}$ tel que $MV_k = V_j$.

Comme l'application associée est bijective, $MV_j \neq V_j$ car V_j aurait 2 antécédents, ce qui n'est pas possible. Le nombre d'états possible du LFSR étant fini, on reviendra forcément à l'état V_k.
Pour conclure, si $\alpha_L = 1$, la suite est périodique.

Pour la question 5, on détermine facilement à l'aide de l'expression de la suite r_n du premier LFSR que $\alpha_1 = \alpha_3 = \alpha_4 = \alpha_7 = \alpha_{10} = 1$, et que les autres valeurs de α_i valent 0.
De la même façon pour le second LFSR, on constate que $\alpha_3 = 1$, et que donc $\alpha_1 = \alpha_2 = 0$.
On constatera surtout que les 20 premières valeurs produites par ces LFSR sont identiques : il s'agit de séquences de « 1, 0, 0 ». Nous avons par ailleurs dans le fichier **test.ml** les 20 premières valeurs produites par ces deux LFSR, et les séquences produites sont bien identiques.

Exercice 4

On montre que S(X)R(X) = G(X) par récurrence sur la longueur L du LFSR.

- Base : L = 1

On a $S(X)R(X) = r_0\alpha_0 \oplus X(r_0\alpha_1 + \alpha_0 r_1) \oplus X^2(r_1\alpha_1 + r_2\alpha_0) \oplus \dots$
Or, Pour L = 1, $G(X) = r_0\alpha_0$.
Il faut donc montrer que $X(r_0\alpha_1 + \alpha_0 r_1) \oplus X^2(r_1\alpha_1 + r_2\alpha_0) \oplus \dots = 0$.

Si $\alpha_1 = 0$: $S(X)R(X) = G(X) \oplus r_1X \oplus r_2X^2 \oplus \dots$

Si $r_0 = 0$: S(X)R(X) = G(X) car $r_1 = r_0\alpha_1 = r_0 \otimes 0 = 0 = r_2 = r_n$

Si $r_0 = 1$: S(X)R(X) = G(X) car $r_1 = r_0\alpha_1 = r_0 \otimes 0 = 0 = r_2 = r_n$

Si $\alpha_1 = 1$:

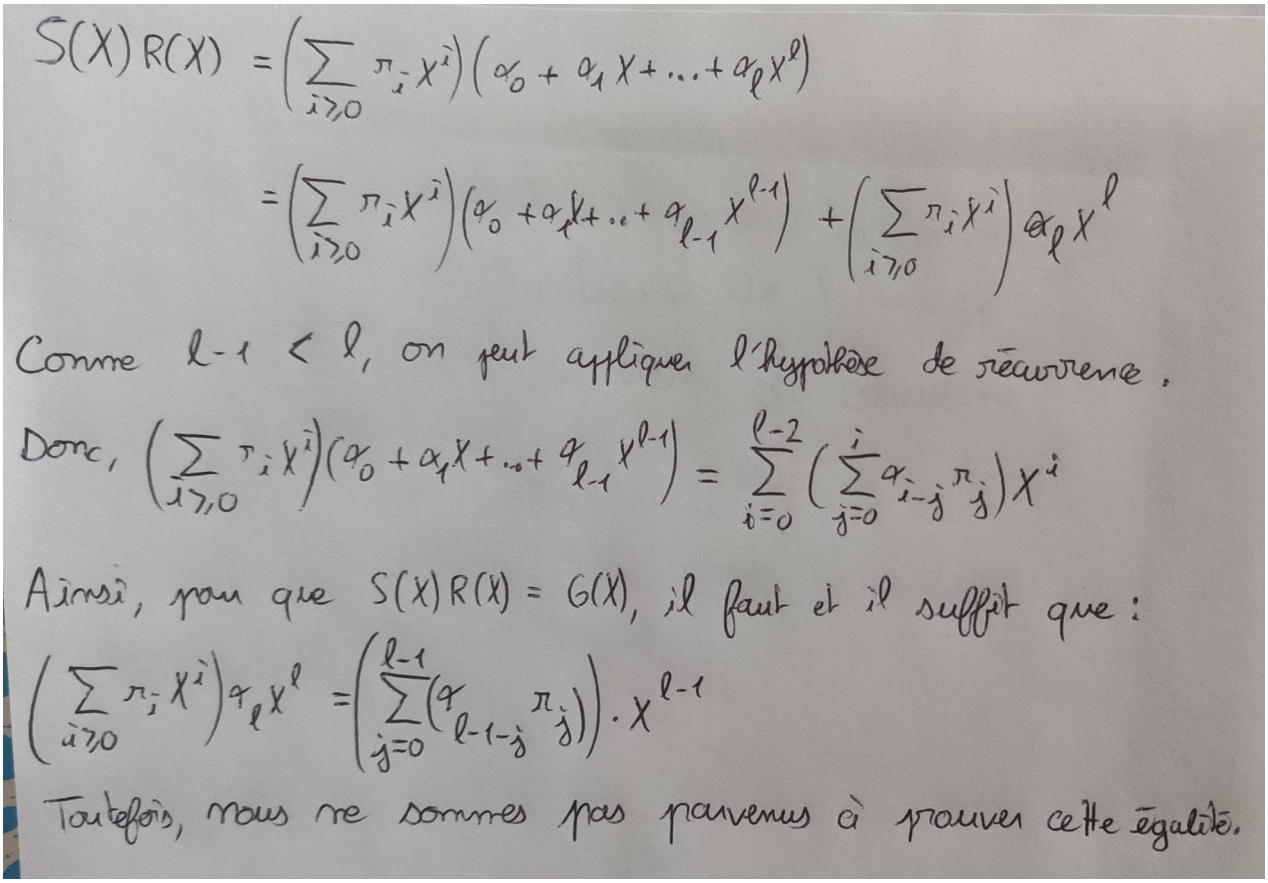
Si $r_0 = 0$: S(X)R(X) = G(X) car $r_1 = r_0\alpha_1 = 0 \otimes \alpha_1 = 0 = r_2 = r_n$

Si $r_0 = 1$: $S(X)R(X) = G(X) \oplus (r_0 \oplus r_1)X \oplus (r_1 \oplus r_2)X^2 \oplus \dots$
Or, $r_1 = r_2 = r_n = 1$ donc pour tout entier naturel i, $(r_i \oplus r_{i+1}) = 0$
d'où S(X)R(X) = G(X)

Pour un LFSR de longueur L = 1, on a S(X)R(X) = G(X).

- Hérédité : L > 1

On pose notre hypothèse de récurrence : Pour tout LFSR de longueur strictement inférieure à L, S(X)R(X) = G(X).
On montre maintenant que pour un LFSR de longueur L, on a toujours S(X)R(X) = G(X).



Nous avons par la suite développé la fonction permettant de convertir un LFSR en triplet (l, G(X), R(X)).
Nous avons dans un premier temps la fonction **rx_calc**, qui calcule trivialement le polynôme R(X) à partir du LFSR puisqu'il n'est constitué que des branchements.
Ensuite, nous avons 2 fonctions auxiliaires, **until_i** et **f2_sum_prod**, qui servent au calcul du polynôme G(X).

La fonction **until_i** renvoie la liste qui lui est donnée en argument privée de ses éléments dépassant l'indice **i**, lui aussi donné en argument. Elle aurait pu être définie dans **utils.ml**, mais comme elle ne sert qu'ici il est plus logique de la laisser juste au-dessus de l'endroit où elle sert.
La fonction **f2_sum_prod** calcule la somme dans F2 des produits des éléments de même indice deux listes. Par exemple, pour les listes [0;1;0] et [1;1;0], on obtiendrait le résultat $r = 0 \otimes 1 \oplus 1 \otimes 1 \oplus 0 \otimes 0 = 1$. Cette fonction a pour but de reproduire la somme contenue dans la somme de G(X). Enfin, la fonction **gx_calc** procède au calcul effectif du polynôme G(X) en faisant appel aux deux fonctions définies précédemment.

Finalement, les fonctions **rx_calc** et **gx_calc** sont utilisées dans la fonction **lgxrx_from_lfsr** qui permet de convertir un LFSR en triplet (l, G(X), R(X)).

Nous avons ensuite programmé la fonction **lfsr_from_lgxrx**, qui réalise l'opération inverse. Nous avons tout d'abord défini trois fonctions servant à récupérer les différents éléments du triplet : **lgxrx_length** pour obtenir la longueur, **lgxrx_gx** pour obtenir le polynôme G(X) et **lgxrx_rx** pour obtenir le polynôme R(X).

Nous avons ensuite développé la fonction **branch_calc** qui permet, à partir du polynôme R(X), de déterminer les branchements du LFSR et ainsi obtenir le troisième champ de notre structure de donnée modélisant un LFSR.
La fonction **base_calc** sert à déterminer les valeurs initiales du LFSR : il s'agit donc du calcul du deuxième champ de la structure LFSR. On utilise pour cela le résultat prouvé précédemment, à savoir que S(X)R(X) = G(X) avec S(X) contenant les valeurs produites par le LFSR et donc les valeurs initiales.

L'idée initiale était de réaliser le quotient de G(X) par R(X) , mais le résultat n'était pas celui escompté : nous avons donc, en plus du quotient, multiplié le reste de ce quotient par l'inverse modulo L (longueur du LFSR) de R(X). Nous lui avons ensuite ajouté avec la fonction **sum_poly** le quotient de G(X) par R(X) (premier élément du couple renvoyé par la division). Enfin, nous avons appliqué la fonction **moduloXn** à ce résultat avec comme second argument la longueur du LFSR afin d'obtenir uniquement les valeurs initiales.

Pour la question 4, on constate tout d'abord à l'aide de notre fonction **euclide** qui calcule le PGCD de deux polynômes selon l'algorithme d'Euclide que les polynômes R(X) et G(X) obtenus à partir du premier LFSR de la question 5 de l'exercice 3 possèdent un PGCD supérieur à 1. En effectuant les calculs décrits dans l'énoncé, on obtient un LFSR plus petit, qui produit la même séquence de valeur : il s'agit du second LFSR de la question 5 de l'exercice 3. On peut donc conclure que ces deux LFSR produisent le même flux de valeurs.

Nous avons développé à partir de ce résultat une fonction permettant de calculer le triplet (L', G'(X), R'(X)) minimal à partir d'un triplet (L, G(X), R(X)). Dans un premier temps, la fonction **smallest_lgxrx** calcule le PGCD des polynômes G(X) et R(X). Si celui-ci est supérieur à 1, on effectue les calculs de l'énoncé afin d'obtenir le triplet (L', G'(X), R'(X)) minimal.
Enfin, la fonction **smallest_lfsr** qui calcule le LFSR minimal à partir d'un LFSR donné en argument convertit seulement le LFSR en triplet (L, G(X), R(X)), en détermine le triplet minimal avec la fonction **smallest_lgxrx** et convertit ce triplet en LFSR.

Exercice 5

On souhaite à présent programmer la construction d'un bon LFSR, de longueur L. Dans notre fonction **bon_lfsr**, on commence par générer un polynôme primitif de degré (L – 1), ce polynôme possède donc L coefficients valant 0 ou 1 : ce sera le polynôme de rétroaction de notre LFSR. On génère ensuite un polynôme de F2 de degré (L - 1) pseudo-aléatoire à l'aide d'une fonction auxiliaire **random_poly**. On calcule alors le produit du polynôme de rétroaction et de ce polynôme, afin d'obtenir le polynôme G(X) : il ne reste alors qu'à appliquer la fonction **moduloXn** avec comme argument la longueur du LFSR souhaité au polynôme obtenu afin d'obtenir G(X). On peut alors constituer notre triplet de la forme (L, G(X), R(X)) et ainsi calculer notre LFSR à partir de ce triplet.

Exercice 6

Nous devions ici programmer les fonctions de chiffrement et de déchiffrement d'un texte en clair. Nous avons également prévu de conserver le dernier état connu du LFSR comme état initial, afin d'éviter de générer plusieurs fois les valeurs déjà générées.
La fonction de chiffrement **plaintext_coding** permet de réaliser ce chiffrement en appliquant le calcul $c_i = t_i \oplus r_i$ avec t_i l'élément à l'indice i du texte en clair et c_i la i-ème produite par le LFSR.
La fonction de déchiffrement **plaintext_decoding** est exactement la même que la fonction précédente, puisque la relation $t_i \oplus r_i \oplus r_i$ est vérifiée.

Exercice 7

Pour finir, nous devions, en connaissant les 2n premières valeurs d'une suite produite par un LFSR inconnu, réaliser l'attaque programmant le calcul du plus petit LFSR générant une suite commençant avec ces valeurs à l'aide de l'algorithme de Berkelamp-Massey. Nous avons donc créé une fonction **ciphertext_attack** permettant de réaliser cette opération. Dans un premier temps, nous vérifions que la liste de valeurs que notre LFSR doit produire est de longueur paire (car nous avons besoin de 2n valeurs), puis nous appliquons l'algorithme d'Euclide étendu avec les valeurs de base données dans l'énoncé.
On extrait ensuite de cet algorithme les polynômes Bm et Rm pour ensuite calculer le polynôme de rétroaction. Enfin, on calcule le polynôme G(X) en multipliant le

polynôme R(X) calculé par S(X), qui est le polynôme issu de la conversion de la liste binaire d'entrée en polynôme de F2[X].
On applique enfin la fonction permettant de traduire un triplet (L, G(X), R(X)) en LFSR pour obtenir le LFSR attendu. Cependant, notre fonction ne fonctionne que dans certains cas : nous n'avons pas trouvé la cause de ce dysfonctionnement.

III – Difficultés rencontrées

Nous avons rencontré plusieurs difficultés lors de la réalisation de ce projet. La première a été de comprendre comment fonctionnait un LFSR, notamment le fonctionnement des branchements. Ensuite, nous n'avons pas eu du mal à établir la preuve sur la périodicité d'un LFSR lorsque α_L vaut 1 et nous n'avons pas réussi la preuve de la relation S(X)R(X) = G(X).
La dernière fonction a réaliser a également été problématique, puisque nous n'avons pas trouvé le problème de notre fonction malgré de nombreuses vérifications sur papier.
Enfin, le dernier problème rencontré n'était pas une difficulté à proprement parler mais plutôt un frein à la réalisation du projet puisqu'il s'agit des multiples coquilles présentes dans l'énoncé. Par exemple, l'erreur sur la valeur du degré du triplet obtenu après la division lors du calcul du LFSR minimal nous a beaucoup ralenti. Les erreurs sur le dernier exercice ont également été un léger frein, comme celle sur la valeur de Bm ou une valeur « l » qui semble se transformer en « n » dans le dernier exercice.

IV - Conclusion

La réalisation de ce projet était intéressante, et nous a également permis de voir une application concrète des polynômes dans la cryptographie à travers l'utilisation de LFSR. Une extension possible à ce projet serait, comme expliqué à la fin du sujet de projet, de créer un petit générateur de clé avec l'algorithme de Berkelamp-Massey, en générant une clé aléatoire assez grande puis en utilisant l'algorithme de Berkelamp-Massey pour trouver un LFSR générant cette clé.