TD4: Algorithmique et programmation C: I & II

(Version corrigée)

(Michaël Quisquater, UVSQ)

Corps finis tabulés & programmation d'un algorithme de chiffrement/déchiffrement mini-AES.

Dans la présente correction, beaucoup de variantes sont présentées. Il est clair qu'un travail aussi complet n'était pas demandé durant le TD, cela vous permet simplement de vérifier votre version et/ou de voir des alternatives à ce que vous avez fait.

Question 1: Génération de polynômes primitifs et affichage de polynômes.

1. Écrire une fonction renvoyant un polynôme primitif de $\mathbb{F}_{2^n}$, extension de $\mathbb{F}_2$ (sous le format d'un *unsigned int* avec la convention $2^i <-> X^i$). On rentrera les polynômes (voir ci-dessous) en dur dans la fonction.
2. Écrire une fonction permettant d'afficher un polynôme donné en argument sous la forme d'un *unsigned int* (avec la convention $2^i <-> X^i$)
3. (A faire à la fin). Écrire une fonction retournant une chaîne de caractères contenant le polynôme donné en argument sous la forme d'un *unsigned int*. On utilisera les fonctions de la librairie < string.h >.

Polynômes primitifs de $\mathbb{F}_{2^n} : \mathbb{F}_2$.

n	k ou (k_1, k_2, k_3)	n	k ou (k_1, k_2, k_3)	n	k ou (k_1, k_2, k_3)
2	1	12	7,4,3	22	1
3	1	13	4,3,1	23	5
4	1	14	12,11,1	24	4,3,1
5	2	15	1	25	3
6	1	16	5,3,2	26	8,7,1
7	1	17	3	27	8,7,1
8	6,5,1	18	7	28	3
9	4	19	6,5,1	29	2
10	3	20	3	30	16,15,1
11	2	21	2	31	3

Pour chaque n, un exposant k est donné pour lequel le trinôme x^n+x^k+1 est primitif. A défaut, un triplet (k_1, k_2, k_3) est donné pour lequel le pentanôme $x^n + x^{k_1} + x^{k_2} + x^{k_3} + 1$ est primitif.

Correction:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int Hamming(unsigned int x);

unsigned int Poly_Primitif(unsigned int n);
unsigned int Poly_Primitif_Bis(unsigned int n);
void Impression_Polynome(unsigned int P, unsigned int n);

void Allouer_Chaine(char **Output,unsigned int n);
void Liberer_Chaine(char **Output);
void Initialiser_Chaine(char *Output);
int Longueur_Chaine_Polynome(unsigned int n)

void Conversion_Polynome(char *Output, unsigned int P, unsigned int n);


int main()
{
  unsigned int n=9;
  unsigned int P;
  char *Output;


  /* P=Poly_Primitif(n); Poly_Primitif et Poly_Primitif_Bis ont le meme effet,
    c'est juste l'implementation qui est differente*/
  P=Poly_Primitif_Bis(n);

  printf("Le polynome primitif de degre %u est: ",n);
  Impression_Polynome(P,n);
  printf("\n");

  Allouer_Chaine(&Output,n);
  Initialiser_Chaine(Output);
  Conversion_Polynome(Output,P,n);
  printf("Le polynome primitif de degre %u est: %s \n",n,Output);


  Liberer_Chaine(&Output);

  exit(0);
}

int Hamming(unsigned int x)
{
   x=( x & 0x55555555 ) + ( (x>>1) & 0x55555555 );
   x=( x & 0x33333333 ) + ( (x>>2) & 0x33333333 );
   x=( x & 0x0F0F0F0F ) + ( (x>>4) & 0x0F0F0F0F );
   x=( x & 0x00FF00FF ) + ( (x>>8) & 0x00FF00FF );
   x=( x & 0x0000FFFF ) + ( (x>>16) & 0x0000FFFF );
   return x;
}

unsigned int Poly_Primitif(unsigned int n)
{
    int i;
    unsigned int P;
    int poly[32][4];
    /*poly[n][0] represente le nombre de monomes non constant et
          non dominant dans le polynome primitif de degre n.*/
    /*poly[n][i] avec i>=0 represente la presence du monome
              x^i dans le polynome primitif de degre n.*/

    poly[2][0]=1;  poly[2][1]=1;
    poly[3][0]=1;  poly[3][1]=1;
    poly[4][0]=1;  poly[4][1]=1;
    poly[5][0]=1;  poly[5][1]=2;
```

```
    poly[6][0]=1;  poly[6][1]=1;
    poly[7][0]=1;  poly[7][1]=1;
    poly[8][0]=3;  poly[8][1]=6;     poly[8][2]=5  ;poly[8][3]=1;
    poly[9][0]=1;  poly[9][1]=4;
    poly[10][0]=1;  poly[10][1]=3;
    poly[11][0]=1;  poly[11][1]=2;
    poly[12][0]=3;  poly[12][1]=7  ; poly[12][2]=4  ; poly[12][3]=3  ;
    poly[13][0]=3;  poly[13][1]=4  ; poly[13][2]=3  ; poly[13][3]=1  ;
    poly[14][0]=3;  poly[14][1]=12; poly[14][2]=11; poly[14][3]=1  ;
    poly[15][0]=1;  poly[15][1]=1  ;
    poly[16][0]=3;  poly[16][1]=5  ; poly[16][2]=3  ; poly[16][3]=2  ;
    poly[17][0]=1;  poly[17][1]=3  ;
    poly[18][0]=1;  poly[18][1]=7  ;
    poly[19][0]=3;  poly[19][1]=6  ; poly[19][2]=5  ; poly[19][3]=1  ;
    poly[20][0]=1;  poly[20][1]=3  ;
    poly[21][0]=1;  poly[21][1]=2  ;
    poly[22][0]=1;  poly[22][1]=1  ;
    poly[23][0]=1;  poly[23][1]=5  ;
    poly[24][0]=3;  poly[24][1]=4  ; poly[24][2]=3  ; poly[24][3]=1  ;
    poly[25][0]=1;  poly[25][1]=3  ;
    poly[26][0]=3;  poly[26][1]=8  ; poly[26][2]=7  ; poly[26][3]=1  ;
    poly[27][0]=3;  poly[27][1]=8  ; poly[27][2]=7  ; poly[27][3]=1  ;
    poly[28][0]=1;  poly[28][1]=3  ;
    poly[29][0]=1;  poly[29][1]=2  ;
    poly[30][0]=3;  poly[30][1]=16; poly[30][2]=15; poly[30][3]=1  ;
    poly[31][0]=1;  poly[31][1]=3  ;

    P=(0x1<<n)^0x1;

    for(i=1;i<=poly[n][0];i++)
        P^=(01<<poly[n][i]);
  /*P contient le polynome primitif permettant de construire le corps a 2^n elements.
    Il suffit de remplacer 2^i par X^i pour obtenir le polynome */

    return P;
}

unsigned int Poly_Primitif_Bis(unsigned int n)
{
 /*Cette fonction est identique a Poly_Primitif mais est implementee
   plus simplement, necessitant plus de travail en amont*/

unsigned int Primitif[]={0x0,0x0,0x7,0xb,0x13,0x25,0x43,0x83,
0x163,0x211,0x409,0x805,0x1099,0x201b,0x5803,0x8003,
0x1002d,0x20009,0x40081,0x80063,0x100009,0x200005,0x400003,0x800021,
0x100001b,0x2000009,0x4000183,0x8000183,0x10000009,0x20000005,0x40018003,0x80000009};

return Primitif[n];
}

void Impression_Polynome(unsigned int P,unsigned int n)
{
  /*a partir d'un unsigned int, affiche un polynome. La convention est 2^i <-> X^i.
    n represente le degre maximal que peut avoir le polynome P*/

    int i,temp;

    temp=Hamming(P)-1;

    if (P==0)
        printf("0 ");

    for(i=n;i>=0;i--)
        {
          if ((P>>i)&0x1==1)
            {
              if (i==0)
                  printf("1");
                else
                printf("X^%d",i);
```

```
                if (temp!=0)
                 {
                   printf("+");
                   temp--;
                 }
                 else
                   printf(" ");
              }
         }
}



void Allouer_Chaine(char **Output, unsigned int n)
{
   /*n represente le degre maximal que peut avoir le polynome P.
     Alloue 2(2n+1) char pour la chaine chaine output si n<10 et 5*n-8 sinon.. */

   int Taille_chaine_polynome_max;

   Taille_chaine_polynome_max=Longueur_Chaine_Polynome(n);
   *Output=(char*)malloc(Taille_chaine_polynome_max*sizeof(char));
}

void Liberer_Chaine(char **Output)
{  /*libere la memoire allouee a *Ouput et lui affecte NULL*/

   free(*Output);
   *Output=NULL;
}

void Initialiser_Chaine(char *Output)
{
   /*initialise la chaine Output. On suppose que la memoire a ete allouee*/

    Output[0]='\0';
}


int Longueur_Chaine_Polynome(unsigned int n)
{
    int Taille_chaine_polynome_max;

    Taille_chaine_polynome_max=(n<10)? (((n<<1)+1)<<1): 5*n-8;

    return Taille_chaine_polynome_max;
}

void Conversion_Polynome(char *Output, unsigned int P, unsigned int n)
{
    /*a partir d'un unsigned int, retourne une chaine de caractere
          contenant le polynome. La convention est 2^i <-> X^i.
          n represente le degre maximal que peut avoir le polynome P.
          La chaine output est supposee allouee et de
          taille Longueur_Chaine_Polynome(n). On complete toute la
          chaine avec des espaces si le polynome n'est pas de plein degre.
          On suppose que $n<100$.*/

    int i, temp, longueur, accu;
    char tempor[5];
    int Taille_chaine_polynome_max;

    Taille_chaine_polynome_max=Longueur_Chaine_Polynome(n);

    temp=Hamming(P)-1;
    accu=0;
    Initialiser_Chaine(Output);

    if (P==0)
```

```
    {
     strcat(Output,"0");
     accu++;
    }

   for(i=n;i>=0;i--)
        {
            if ((P>>i)&0x1==1)
              {
                if (i==0)
                 {
                  strcat(Output,"1");
                  accu++;
                 }
                    else
                     {
                      sprintf(tempor,"X^%d",i);
                      strcat(Output,tempor);
                      accu+=3;
                     }

                if (temp!=0)
                 {
                   strcat(Output,"+");
                   temp--;
                   accu++;
                 }
              }
        }

  for(i=Taille_chaine_polynome_max; i>accu ; i--)
   {
      strcat(Output," ");
   }
}
```

Question 2: Corps finis: tabulation polynôme/expo.

Génération des "tables" de la fonction logarithme et exponentielle

1. Écrire une fonction allouant la mémoire pour les "tables" de la fonction logarithme et exponentielle en fonction du degré de l'extension du corps fini (c'est-à-dire le degré du polynôme primitif le definissant).
2. Écrire une fonction libérant la mémoire pour les "tables" des fonctions logarithme et exponentielle.
3. Écrire une fonction remplissant les "tables" des fonctions logarithme et exponentielle en fonction d'un polynôme primitif donné en argument.

Utilisation des "tables" de la fonction logarithme et exponentielle

4. Écrire une fonction permettant de calculer l'addition de deux éléments d'un corps fini $\mathbb{F}_{2^n}$ donnés sous forme polynomiale.
5. Écrire une fonction permettant de calculer le produit de deux éléments d'un corps fini $\mathbb{F}_{2^n}$ donnés sous forme polynomiale.
6. Écrire une fonction permettant de calculer l'inverse d'un élément d'un corps fini $\mathbb{F}_{2^n}$ donné sous forme polynomiale.

Affichage des tables de Cayley d'addition et de la multiplication

7. Écrire une fonction permettant d'allouer un tableau bidimensionnel dont le nombre de lignes et de colonnes est la taille du corps.

8. Écrire une fonction permettant de désallouer un tableau bidimensionnel dont le nombre de lignes et de colonnes est la taille du corps.

9. Écrire une fonction permettant de calculer (sous forme polynomiale) les tables de Cayley pour l'addition et la multiplication en se basant sur les tables exponentielle et logarithme.

10. Écrire une fonction permettant d'afficher (sous forme hexadécimale) un tableau bidimensionnel dont le nombre de lignes et de colonnes est la taille du corps.

Correction:

(voir plus loin).

Question 3: Corps finis: table du logarithme de Zech

Génération de la "table" du logarithme de Zech

1. Écrire une fonction allouant la mémoire pour la "table" du logarithme de Zech en fonction du du degré de l'extension du corps fini (c'est-à-dire le degré du polynôme primitif le definissant).

2. Écrire une fonction libérant la mémoire pour la "table" du logarithme de Zech.

3. Écrire une fonction remplissant la "table" du logarithme de Zech en fonction des tables de la fonction logarithme et exponentielle données en argument.

Utilisation de la "table" du logarithme de Zech

4. Écrire une fonction permettant de calculer l'addition de deux éléments d'un corps fini $\mathbb{F}_{2^n}$ donnés sous forme exponentielle.

5. Écrire une fonction permettant de calculer le produit de deux éléments d'un corps fini $\mathbb{F}_{2^n}$ donnés sous forme exponentielle.

6. Écrire une fonction permettant de calculer l'inverse d'un élément d'un corps fini $\mathbb{F}_{2^n}$ donné sous forme exponentielle.

7. Écrire une fonction permettant de calculer (sous forme polynomiale) les tables de Cayley pour l'addition et la multiplication en se basant sur le logarithme de Zech.

Correction:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>

typedef unsigned int uint;

/*Fonction(s) annnexe(s)*/
int Hamming(uint x);
void Impression_Tableau(uint *t,int lg);

/*Gestion de la memoire pour les chaines de caracteres*/
void Allouer_Chaine(char **Output, uint n);
void Liberer_Chaine(char **Output);
void Initialiser_Chaine(char *Output);

/*Transformation d'un polynome sous forme unsigned int
en un sa forme chaine de caracteres*/
int Longueur_Chaine_Polynome(uint n);
void Conversion_Polynome(char *Output, uint P, uint n);

/*Calcul du polynome primitif*/
unsigned int Poly_Primitif_Bis(uint n);

/*Partie specifique a la methode polynomiale/exponentielle */
void Initialisation_Tab_Exp_Log(uint **Exp, uint **Log, uint Taille_corps);
void Liberer_Tab_Exp_Log(uint **Exp, uint **Log);
void Corps_Fini_Pol_Exp(uint *Exp, uint *Log, uint P, uint Taille_corps);
void Inverse_Exp_Log(uint *x, uint *Exp, uint *Log, uint Taille_corps);
uint Multiplication_Exp_Log(uint x, uint y, uint *Exp, uint *Log, uint Taille_corps);
uint Addition_Exp_Log(uint x, uint y);
void Calcul_Cayley_Exp_Log(int Flag, uint **Tableau, uint *Exp, uint *Log, uint Taille_corps);


/*Partie specifique a la methode logarithme de Zech/Jacobi*/
void Initialisation_Tab_Log_Zech(uint **Log_Zech, uint Taille_corps);
void Liberer_Tab_Log_Zech(uint **Log_Zech);
void Corps_Fini_Log_Zech(uint *Log_Zech, uint *Exp, uint *Log, uint Taille_corps);
uint Addition_Log_Zech(uint x, uint y, uint *Log_Zech, uint Taille_corps);
uint sum(uint x, uint y, uint modulo);
uint diff(uint x, uint y, uint modulo);
uint Multiplication_Log_Zech(uint x, uint y, uint Taille_corps);
void Inverse_Log_Zech(uint *x, uint Taille_corps);
void Calcul_Cayley_Zech(int Flag, uint **Tableau, uint *Log_Zech, uint *Exp, uint Taille_corps);


/*Gestion de tableaux bi-dimensionnels d'unsigned int*/
void Allouer_Tableau_Bi(uint ***Tableau, uint Taille);
void Liberer_Tableau_Bi(uint ***Tableau, uint Taille);

/*Fonction d'affichage Cayley*/
void Affichage_Cayley(int Flag, uint **Tableau, uint Taille_corps, uint n);

int main()
{

    unsigned int n=2;
    uint Taille_corps;
    uint P,Arg,Inv,Add,Multi;
    uint *Exp,*Log,*Log_Zech;
    uint **Tableau;
    char *Output,*Input1,*Input2,*Primitif;

    Taille_corps=0x1<<n;

        /*construction des tables*/
    P=Poly_Primitif_Bis(n);
    Initialisation_Tab_Exp_Log(&Exp,&Log,Taille_corps);
    Corps_Fini_Pol_Exp(Exp,Log,P,Taille_corps);
```

```
    /*printf("Tableau Exp\n");
Impression_Tableau(Exp,Taille_corps);
printf("Tableau Log\n");
Impression_Tableau(Log,Taille_corps);*/

Allouer_Chaine(&Primitif,n);
Conversion_Polynome(Primitif,P,n);
printf("Le polynome primitif de degre %u est: %s \n",n,Primitif);
Liberer_Chaine(&Primitif);

/*----------------Partie Specifique a Log_Exp----------------------------------------*/
printf("Exp_Log\n");

Arg=0x2; /*il faudrait verifier que "degre" de Arg est bien < P*/
/*Inverse*/
Inv=Arg;
Inverse_Exp_Log(&Inv,Exp,Log,Taille_corps);
/*Multiplication*/
Multi=Multiplication_Exp_Log(Arg,Inv,Exp,Log,Taille_corps);
/*Addition*/
Add=Addition_Exp_Log(Arg,Inv);

/*Allocation des chaines de caracteres*/
Allouer_Chaine(&Input1,n-1);  /*n-1 car les representants minimaux sont de */
Allouer_Chaine(&Input2,n-1);  /*degre strictement inferieur a n.           */
Allouer_Chaine(&Output,n-1);

/*Inverse*/
Conversion_Polynome(Input1,Arg,n-1);
Conversion_Polynome(Output,Inv,n-1);
printf("L'inverse de %s est: %s\n",Input1,Output);
/*Multiplication*/
Conversion_Polynome(Input2,Inv,n-1);;
Conversion_Polynome(Output,Multi,n-1);
printf("Le produit de %s et %s est: %s\n",Input1,Input2,Output);
/*Addition*/
Conversion_Polynome(Output,Add,n-1);
printf("La somme de %s et %s est: %s\n",Input1,Input2,Output);

Liberer_Chaine(&Input1);
Liberer_Chaine(&Input2);
Liberer_Chaine(&Output);

/*Calcul et affichage des tables de Cayley. Remarque: on calcule toute la table avant
de l'afficher.  Si la table est grande ce n'est pas une bonne idee, dans ce cas
il est preferable de calculer ligne la ligne et d'afficher a la volee. */

Allouer_Tableau_Bi(&Tableau,Taille_corps);
Calcul_Cayley_Exp_Log(0,Tableau,Exp,Log,Taille_corps);
Affichage_Cayley(0,Tableau,Taille_corps,n);
Calcul_Cayley_Exp_Log(1,Tableau,Exp,Log,Taille_corps);
Affichage_Cayley(1,Tableau,Taille_corps,n);
Liberer_Tableau_Bi(&Tableau,Taille_corps);

/*----------------------Partie Specifique a Log_Zech---------------------------------*/
printf("Zech\n");

Initialisation_Tab_Log_Zech(&Log_Zech,Taille_corps);
Corps_Fini_Log_Zech(Log_Zech,Exp,Log,Taille_corps);

/*printf("Tableau Log_Zech\n");
Impression_Tableau(Log_Zech,Taille_corps);*/

/*Calculs locaux*/
Allouer_Chaine(&Input1,n-1);
Allouer_Chaine(&Input2,n-1);
Arg=0x3; /*il faudrait verifier que "degre" de Arg est bien < P*/
Conversion_Polynome(Input1,Arg,n-1);
Inv=Log[Arg];
Inverse_Log_Zech(&Inv,Taille_corps);
```

```
    Inv=Exp[Inv];
    Conversion_Polynome(Input2,Inv,n-1);
    printf("L'inverse de %s est %s\n",Input1,Input2);
    Liberer_Chaine(&Input1);
    Liberer_Chaine(&Input2);

    /*Calcul et affichage des tables de Cayley. Remarque: on calcule toute la table avant
    de l'afficher.  Si la table est grande ce n'est pas une bonne idee, dans ce cas
    il est preferable de calculer ligne la ligne et d'afficher a la volee. C'est cependant
    plus modulaire et permet plus facilement d'afficher les elements forme polynomiale/hexa et
    dans le "bon ordre".*/

    Allouer_Tableau_Bi(&Tableau,Taille_corps);
    Calcul_Cayley_Zech(0,Tableau,Log_Zech,Exp,Taille_corps);
    Affichage_Cayley(0,Tableau,Taille_corps,n);
    Calcul_Cayley_Zech(1,Tableau,Log_Zech,Exp,Taille_corps);
    Affichage_Cayley(1,Tableau,Taille_corps,n);
    Liberer_Tableau_Bi(&Tableau,Taille_corps);

    /*Liberation de la memoire*/
    Liberer_Tab_Exp_Log(&Exp,&Log);
    Liberer_Tab_Log_Zech(&Log_Zech);

    exit(0);
}



/*Fonction(s) annnexe(s)*/
int Hamming(uint x)
{
   x=( x & 0x55555555 ) + ( (x>>1) & 0x55555555 );
   x=( x & 0x33333333 ) + ( (x>>2) & 0x33333333 );
   x=( x & 0x0F0F0F0F ) + ( (x>>4) & 0x0F0F0F0F );
   x=( x & 0x00FF00FF ) + ( (x>>8) & 0x00FF00FF );
   x=( x & 0x0000FFFF ) + ( (x>>16) & 0x0000FFFF );
   return x;
}


void Impression_Tableau(uint *t, int lg)
{
/*imprime les elements du tableau d'unsigned int, tous sur la meme ligne.
Un message d'erreur si celui-ci est vide  */

   int i;
   int elem;

   if (t==NULL)
    {
       fprintf(stderr,"tableau vide\n");
    }
   else
    {
      for(i=0;i < lg;i++)
       {
         printf("t[%d]=%u  ",i,t[i]);
         if (i==(lg-1)) printf("\n");
       }
    }
}

void Allouer_Chaine(char **Output, uint n)
{
   /*n represente le degre maximal que peut avoir le polynome P.
     Alloue 2(2n+1) char pour la chaine chaine output si n<10 et 5*n-8 sinon.. */

   int Taille_chaine_polynome_max;

   Taille_chaine_polynome_max=Longueur_Chaine_Polynome(n);
```

```
   *Output=(char*)malloc(Taille_chaine_polynome_max*sizeof(char));
}

void Liberer_Chaine(char **Output)
{  /*libere la memoire allouee a *Ouput et lui affecte NULL*/

   free(*Output);
   *Output=NULL;
}

void Initialiser_Chaine(char *Output)
{
   /*initialise la chaine Output. On suppose que la memoire a ete allouee*/

    Output[0]='\0';
}


int Longueur_Chaine_Polynome(uint n)
{
    int Taille_chaine_polynome_max;

    Taille_chaine_polynome_max=(n<10)? (((n<<1)+1)<<1): 5*n-8;

    return Taille_chaine_polynome_max;
}

void Conversion_Polynome(char *Output, uint P, uint n)
{
        /*a partir d'un unsigned int, retourne une chaine de caractere
          contenant le polynome. La convention est 2^i <-> X^i.
          n represente le degre maximal que peut avoir le polynome P.
          La chaine output est supposee allouee et de
          taille Longueur_Chaine_Polynome(n). On complete toute la
          chaine avec des espaces si le polynome n'est pas de plein degre.
          On suppose que $n<100$.*/

    int i, temp, longueur, accu;
    char tempor[5];
    int Taille_chaine_polynome_max;

    Taille_chaine_polynome_max=Longueur_Chaine_Polynome(n);

    temp=Hamming(P)-1;
    accu=0;
    Initialiser_Chaine(Output);

    if (P==0)
     {
      strcat(Output,"0");
      accu++;
     }

    for(i=n;i>=0;i--)
         {
            if ((P>>i)&0x1==1)
              {
                if (i==0)
                 {
                  strcat(Output,"1");
                  accu++;
                 }
                    else
                     {
                      sprintf(tempor,"X^%d",i);
                      strcat(Output,tempor);
                      accu+=3;
                     }

                if (temp!=0)
```

```
                {
                  strcat(Output,"+");
                  temp--;
                  accu++;
                }
              }
          }

   for(i=Taille_chaine_polynome_max; i>accu ; i--)
    {
       strcat(Output," ");
    }
}


/*Calcul du polynome primitif*/
unsigned int Poly_Primitif_Bis(uint n)
{

uint Primitif[]={0x0,0x0,0x7,0xb,0x13,0x25,0x43,0x83,
0x163,0x211,0x409,0x805,0x1099,0x201b,0x5803,0x8003,
0x1002d,0x20009,0x40081,0x80063,0x100009,0x200005,0x400003,0x800021,
0x100001b,0x2000009,0x4000183,0x8000183,0x10000009,0x20000005,0x40018003,0x80000009};

return Primitif[n];
}


/*Partie specifique a la methode polynomiale/exponentielle */

void Initialisation_Tab_Exp_Log(uint **Exp, uint **Log, uint Taille_corps)
{
   *Exp=(uint*) calloc(Taille_corps,sizeof(uint));
   *Log=(uint*) calloc(Taille_corps,sizeof(uint));
}

void Liberer_Tab_Exp_Log(uint **Exp, uint **Log)
{
    free(*Exp);
    *Exp=NULL;
    free(*Log);
    *Log=NULL;
}

void Corps_Fini_Pol_Exp(uint *Exp, uint *Log, uint P, uint Taille_corps)
{
  /*Construit la table de la fonction exponentielle (Exp) et du logarithme (Log) d'un corps fini
 de caracteristique 2 et de degre n (2^n=Taille_corps), les tables sont supposees allouees.
 P contient le polynome primitif utilise pour construire ces tables*/

   uint i;

  /*initialisation*/
   Exp[0]=1;
   Log[Exp[0]]=0;

   for(i=1; i < Taille_corps-1 ; i++)
    {
      Exp[i]=Exp[i-1]<<1;
      if ((Exp[i]&Taille_corps)!=0)
            Exp[i]^=P;
      Log[Exp[i]]=i;
    }
   Exp[Taille_corps-1]=0;  /*exponentiel en moins l'infini est zero*/
   Log[0]=Taille_corps-1;  /*logarithme en zero est moins l'infini*/
}

void Inverse_Exp_Log(uint *x, uint *Exp, uint *Log, uint Taille_corps)
{  /**x doit etre non nul*/
    *x=Exp[Taille_corps-1-Log[*x]];
```

```
}

uint Multiplication_Exp_Log(uint x, uint y, uint *Exp, uint *Log, uint Taille_corps)
{
    int Ordre;
    Ordre=Taille_corps-1;
    if ((x==0)||(y==0))
       return 0x0;
    else
       return Exp[(Log[x]+Log[y])%Ordre];
}

uint Addition_Exp_Log(uint x, uint y)
{
    return x^y;
}


void Calcul_Cayley_Exp_Log(int Flag, uint **Tableau, uint *Exp, uint *Log, uint Taille_corps)
{
        /*Calcule la table de Cayley d'addition (Flag=0) ou de Multiplication (Flag!=0) sous forme
    hexadecimale uniquement a partir des tables Exp et Log. On suppose que Tableau est allouee*/

        uint i,j;

    for(i=0; i<Taille_corps; i++)
     {
       for(j=0; j<Taille_corps; j++)
        {
          if (Flag==0)
                Tableau[i][j]=Addition_Exp_Log(i,j);
          else
            Tableau[i][j]=Multiplication_Exp_Log(i,j,Exp,Log,Taille_corps);
        }
    }
}




/*Partie specifique a la methode logarithme de Zech/Jacobi*/
void Initialisation_Tab_Log_Zech(uint **Log_Zech,uint Taille_corps)
{
   *Log_Zech=(uint*) calloc(Taille_corps,sizeof(uint));
}

void Liberer_Tab_Log_Zech(uint **Log_Zech)
{
    free(*Log_Zech);
    *Log_Zech=NULL;
}


void Corps_Fini_Log_Zech(uint *Log_Zech, uint *Exp, uint *Log, uint Taille_corps)
{
        int i;

    /*Log_Zech[0]=2^n-1; Log_Zech[2^n-1]=0; -infini est donc 2^n-1*/
         for(i=0;i<Taille_corps;i++)
              Log_Zech[i]=Log[1^Exp[i]];
}



uint Addition_Log_Zech(uint x, uint y, uint *Log_Zech, uint Taille_corps)
{
    int Ordre;

    Ordre=Taille_corps-1;
```

```
    return (x==Ordre)? y: sum(x,Log_Zech[diff(x,y,Ordre)],Ordre);
}

uint sum(uint x, uint y, uint modulo)
{
    return ((x==modulo)||(y==modulo))? modulo: (x+y)%modulo;
}

uint diff(uint x, uint y, uint modulo)
{
    if (y==modulo)
        return modulo;
    else
        return (y>=x)? y-x: modulo+(y-x);
}

uint Multiplication_Log_Zech(uint x, uint y, uint Taille_corps)
{
    int Ordre;

    Ordre=Taille_corps-1;
    return sum(x,y,Ordre);
}

void Inverse_Log_Zech(uint *x, uint Taille_corps)
{
    int Ordre;

    Ordre=Taille_corps-1;

        *x=((*x)==Ordre)? Ordre : Ordre-*x;
}

void Calcul_Cayley_Zech(int Flag, uint **Tableau, uint *Log_Zech, uint *Exp, uint Taille_corps)
{
    /*Calcule la table de Cayley d'addition (Flag=0) ou de Multiplication (Flag!=0) sous forme
    hexadecimale a partir du Log de Zech. On suppose que Tableau est allouee*/

   uint i,j;

   for(i=0; i<Taille_corps; i++)
    {
      for(j=0; j<Taille_corps; j++)
        {
          if (Flag==0)
                Tableau[Exp[i]][Exp[j]]=Exp[Addition_Log_Zech(i,j,Log_Zech,Taille_corps)];
          else
            Tableau[Exp[i]][Exp[j]]=Exp[Multiplication_Log_Zech(i,j,Taille_corps)];
        }
    }
}


/*Gestion de tableaux bi-dimensionnels d'unsigned int*/

void Allouer_Tableau_Bi(uint ***Tableau, uint Taille)
{
   int i;

   *Tableau=(uint**)malloc(Taille*sizeof(uint*));
   for(i=0; i<Taille ; i++)
        (*Tableau)[i]=(uint*)malloc(Taille*sizeof(uint));
}

void Liberer_Tableau_Bi(uint ***Tableau, uint Taille)
{
   int i;

   for(i=0; i<Taille ; i++)
     {
```

```
            free((*Tableau)[i]);
        }
       free((*Tableau));
     *Tableau=NULL;
}

/*Fonction d'affichage Cayley*/

void Affichage_Cayley(int Flag, uint **Tableau, uint Taille_corps, uint n)
{
   /*Calcule la table de Cayley d'addition (Flag=0) ou
   de Multiplication (Flag!=0) sous forme hexadecimale.*/

   uint i,j;
   uint Taille_hexa;

   Taille_hexa=(n>>2)+((n&0x3)>0);

   if (Flag==0)
    {
      printf("La table de Cayley d'addition est: \n");
      printf("+");
        for(i=0;i<Taille_hexa;i++)
          printf(" ");
      printf("| ");
    }
   else
    {
      printf("La table de Cayley de multiplication est: \n");
      printf("*");
         for(i=0;i<Taille_hexa;i++)
           printf(" ");
      printf("| ");
    }

   for(i=0; i<Taille_corps; i++)
     {
      printf("%.*x ",Taille_hexa,i);
     }
   printf("\n");
   printf("--");
   for(i=0; i<Taille_corps*(1+Taille_hexa)+Taille_hexa; i++)
      printf("-");
   printf("\n");
   for(i=0; i<Taille_corps; i++)
    {
      printf("%.*x | ",Taille_hexa,i);
      for(j=0; j<Taille_corps; j++)
          printf("%.*x ",Taille_hexa,Tableau[i][j]);
      printf("\n");
    }
}
```

Question 4: (Mini-AES)

Le but de cet exercice est d'implémenter l'algorithme mini-AES en chiffrement et en déchiffrement.

Considérons le corps fini $F_{2^4} \cong \mathbb{Z}_2[x]/(P(x))$, où $P(X) = X^4 + X + 1$. Notons que $P(X)$ est un polynôme primitif. Chaque classe de ce corps fini de représentant minimal $a_3X^3 + a_2X^2 + a_1X + a_0$ est identifiée au vecteur colonne

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} \in \mathbb{Z}_2^4 .$$

Chaque représentant minimal $a_3X^3 + a_2X^2 + a_1X + a_0$ est également identifié au chiffre $a_3 2^3 + a_2 2^2 + a_1 2 + a_0$ que l'on notera en notation hexadécimale.

a. Description des tours de l'algorithme de chiffrement
La taille du bloc est de 16 bits et le vecteur d'état représentant le message "chiffré" à chaque endroit de l'algorithme de chiffrement est noté $(s_0, s_1, s_2, s_3) \in (\mathbb{F}_{2^4})^4$. Ce vecteur est également identifié à la matrice d'état:

$$S = \begin{pmatrix} s_0 & s_2 \\ s_1 & s_3 \end{pmatrix} \in (\mathbb{F}_{2^4})^{2\times 2}.$$

De même, la sous-clé au $K_i = (k_0^i, k_1^i, k_2^i, k_3^i) \in (\mathbb{F}_{2^4})^4$ est représentée par la matrice:

$$K_i = \begin{pmatrix} k_0^i & k_2^i \\ k_1^i & k_3^i \end{pmatrix} \in (\mathbb{F}_{2^4})^{2\times 2}.$$

Le chiffrement d'un message consiste à appliquer les étapes qui suivent.

On commence par appliquer l'étape AddRoundKey au moyen de la sous clé K_0 puis on effectue pour $i = 1..16$

- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey K_i

Décrivons à présent les différentes étapes en détails:

SubBytes:

Cette étape consiste donc à appliquer à chaque élément de la matrice d'état la fonction

$$S : \mathbb{F}_{2^4} \to \mathbb{F}_{2^4} : X \mapsto \begin{cases} A \cdot X^{-1} + b & \text{si } x \neq 0 \\ b & \text{si } x = 0 \end{cases}$$

où X^{-1} est la fonction inverse dans $\mathbb{F}_{2^4}$.
La matrice A est donnée par

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

Le vecteur b est donné par:

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

ShiftRows:

Cette étape consiste à modifier la matrice d'état S de la façon suivante:

$$S = \begin{pmatrix} s_0 & s_2 \\ s_1 & s_3 \end{pmatrix} \Rightarrow \begin{pmatrix} s_0 & s_2 \\ s_3 & s_1 \end{pmatrix}.$$

MixColumns:

Cette étape consiste à modifier la matrice d'état S de la façon suivante:

$$S = \begin{pmatrix} s_0 & s_2 \\ s_1 & s_3 \end{pmatrix} \Rightarrow C \cdot S\,,$$

où

$$C = \begin{pmatrix} '2' & '3' \\ '3' & '2' \end{pmatrix} \in (\mathbb{F}_{2^4})^{2\times 2}\,.$$

où $'i'$ est la notation hexadécimale de l'élément du corps fini $\mathbb{F}_{2^4}$.

AddRoundKey:

Il s'agit d'additionner la matrice d'état S et la matrice des sous-clés K_i.

b. Description de l'algorithme de cadencement des clés

La clé maître K est un élément de $(\mathbb{F}_{2^4})^4$. Par définition, $K_0 = (k_0^0, k_1^0, k_2^0, k_3^0) = K$. Les autres sous-clés $K_i = (k_0^i, k_1^i, k_2^i, k_3^i)$ se calculent via le schéma ci-dessous; (k_2^i, k_3^i) (considéré comme un vecteur de $\mathbb{F}_2^8$) est décalé à gauche de de c_i bits. On applique ensuite l'Sbox S sur chaque composante de 4 bits. On additionne $'2'^i$ à la composante de gauche. (k_0^{i+1}, k_1^{i+1}) est obtenu en effectuant le XOR du résultat obtenu et de (k_0^i, k_1^i). (k_2^{i+1}, k_3^{i+1}) est obtenu en effectuant le XOR de (k_0^{i+1}, k_1^{i+1}) et de (k_2^i, k_3^i).
Les c_i sont donnés par le tableau:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
c_i	1	1	2	2	1	2	2	1	1	1	1	2	1	1	1	1

Le schéma représentant l'algorithme de cadencement des clés est (i=0...15):