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

#define MODULO 37

typedef struct {
    int rows;
    int cols;
    int **data;
} Matrice;

// Vérifie si un caractère est dans Z37
int est_caractere_valide(char c) {
    return (toupper(c) >= 'A' && toupper(c) <= 'Z') || (toupper(c) >= '0'
&& toupper(c) <= '9') || toupper(c) == ' ';
}

// Convertit un caractère Z37 en nombre
int caractere_vers_nombre(char c) {
    c = toupper(c);
    int nombre;
    if (c >= 'A' && c <= 'Z') {
        nombre = c - 'A';
        printf("Conversion du caractère '%c' en nombre: %d\n", c,
nombre);
        return nombre;
    }
    if (c >= '0' && c <= '9') {
        nombre = c - '0' + 26;
        printf("Conversion du caractère '%c' en nombre: %d\n", c,
nombre);
        return nombre;
    }
    if (c == ' ') {
        nombre = 36;
        printf("Conversion du caractère '%c' en nombre: %d\n", c,
nombre);
        return nombre;
    }
    return -1;
}

// Convertit un nombre en caractère
char nombre_vers_caractere(int n) {
    char caractere;
    if (n >= 0 && n <= 25) {
        caractere = 'A' + n;
        printf("Conversion du nombre %d en caractère: '%c'\n", n,
caractere);
        return caractere;
    }
    if (n >= 26 && n <= 35) {
        caractere = '0' + (n - 26);
        printf("Conversion du nombre %d en caractère: '%c'\n", n,
caractere);
        return caractere;
```

```
    }
    if (n == 36) {
        caractere = ' ';
        printf("Conversion du nombre %d en caractère: '%c'\n", n,
caractere);
        return caractere;
    }
    return '\0';
}

// Alloue une matrice
Matrice* creer_matrice(int rows, int cols) {
    printf("Allocation de mémoire pour une matrice %dx%d...\n", rows,
cols);
    Matrice *mat = malloc(sizeof(Matrice));
    if (!mat) {
        printf("Erreur d'allocation de mémoire pour la structure
Matrice.\n");
        return NULL;
    }
    mat->rows = rows;
    mat->cols = cols;
    mat->data = malloc(rows * sizeof(int *));
    if (!mat->data) {
        printf("Erreur d'allocation de mémoire pour les lignes de la
matrice.\n");
        free(mat);
        return NULL;
    }
    for (int i = 0; i < rows; i++) {
        mat->data[i] = malloc(cols * sizeof(int));
        if (!mat->data[i]) {
            printf("Erreur d'allocation de mémoire pour les éléments de
la ligne %d.\n", i);
            for (int j = 0; j < i; j++) free(mat->data[j]);
            free(mat->data);
            free(mat);
            return NULL;
        }
    }
    printf("Matrice %dx%d allouée avec succès.\n", rows, cols);
    return mat;
}

// Libère une matrice
void liberer_matrice(Matrice *mat) {
    if (!mat) return;
    printf("Libération de la mémoire de la matrice %dx%d...\n", mat-
>rows, mat->cols);
    for (int i = 0; i < mat->rows; i++) {
        free(mat->data[i]);
    }
    free(mat->data);
    free(mat);
    printf("Mémoire de la matrice libérée.\n");
}

// Lit une matrice clé n x n
```

```
Matrice* lire_matrice_cle_n(int n) {
    Matrice *mat = creer_matrice(n, n);
    if (!mat) return NULL;
    printf("Entrez les %d éléments de la matrice clé (%dx%d):\n", n * n,
n, n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("Entrez l'élément [%d][%d]: ", i, j);
            if (scanf("%d", &mat->data[i][j]) != 1) {
                printf("Erreur lors de la lecture de l'élément
[%d][%d].\n", i, j);
                liberer_matrice(mat);
                while (getchar() != '\n');
                return NULL;
            }
            mat->data[i][j] %= MODULO;
        }
    }
    while (getchar() != '\n'); // Nettoyer le tampon d'entrée
    printf("Matrice clé lue avec succès.\n");
    return mat;
}

// Lit un message et le convertit en majuscules
char* lire_message() {
    char buffer[1024];
    printf("Entrez le message (a-z, A-Z, 0-9, espace): ");
    fgets(buffer, sizeof(buffer), stdin);
    buffer[strcspn(buffer, "\n")] = '\0';

    printf("Message entré: \"%s\"\n", buffer);
    for (int i = 0; buffer[i]; i++) {
        buffer[i] = toupper(buffer[i]); // Convertit en majuscule
        if (!est_caractere_valide(buffer[i])) {
            printf("Caractère invalide détecté: '%c'. Veuillez utiliser
A-Z, 0-9 ou espace.\n", buffer[i]);
            free(strdup(buffer)); // Libérer la copie potentielle
            return NULL;
        }
    }
    printf("Message converti en majuscules: \"%s\"\n", buffer);
    return strdup(buffer);
}
// Complète le texte avec des 'X'
char* completer_texte_n(char *texte, int taille_bloc) {
    int len = strlen(texte);
    int reste = len % taille_bloc;
    char *complet;
    if (reste == 0) {
        printf("Le message n'a pas besoin d'être complété.\n");
        complet = strdup(texte);
    } else {
        int new_len = len + (taille_bloc - reste);
        complet = malloc(new_len + 1);
        strcpy(complet, texte);
        for (int i = len; i < new_len; i++) {
            complet[i] = 'X';
        }
```

```
        complet[new_len] = '\0';
        printf("Le message a été complété avec %d 'X': \"%s\"\n",
taille_bloc - reste, complet);
    }
    return complet;
}

// Multiplication matrice * vecteur
void multiplier_matrice_vecteur(Matrice *mat, int *vecteur, int
*resultat) {
    printf("Multiplication de la matrice %dx%d par le vecteur...\n", mat-
>rows, mat->cols);
    for (int i = 0; i < mat->rows; i++) {
        resultat[i] = 0;
        printf("Calcul de resultat[%d]:\n", i);
        for (int j = 0; j < mat->cols; j++) {
            printf("  %d * %d = %d\n", mat->data[i][j], vecteur[j], mat-
>data[i][j] * vecteur[j]);
            resultat[i] += mat->data[i][j] * vecteur[j];
        }
        printf("  Somme = %d\n", resultat[i]);
        resultat[i] %= MODULO;
        printf("  resultat[%d] mod %d = %d\n", i, MODULO, resultat[i]);
    }
    printf("Multiplication matrice-vecteur terminée.\n");
}

// Chiffrement de Hill
char* chiffrer_hill_n(char *texte_clair, Matrice *cle) {
    int n = cle->rows;
    printf("\n--- Début du Chiffrement (taille de clé %dx%d) ---\n", n,
n);
    char *texte = completer_texte_n(texte_clair, n);
    int len = strlen(texte);
    char *chiffre = malloc(len + 1);

    printf("Texte clair préparé pour le chiffrement: \"%s\"\n", texte);

    for (int i = 0; i < len; i += n) {
        printf("\nTraitement du bloc %d-%d: ", i, i + n - 1);
        int vecteur[n], res[n];
        printf("Vecteur bloc (caractères -> nombres): ");
        for (int j = 0; j < n; j++) {
            vecteur[j] = caractere_vers_nombre(texte[i + j]);
            printf("%c(%d) ", texte[i + j], vecteur[j]);
        }
        printf("\n");
        multiplier_matrice_vecteur(cle, vecteur, res);
        printf("Bloc chiffré (nombres -> caractères): ");
        for (int j = 0; j < n; j++) {
            chiffre[i + j] = nombre_vers_caractere(res[j]);
            printf("%d(%c) ", res[j], chiffre[i + j]);
        }
        printf("\n");
    }
    chiffre[len] = '\0';
    printf("--- Fin du Chiffrement ---\n");
    free(texte);
```

```
    return chiffre;
}

// Matrice mineure
Matrice* calculer_matrice_mineure(Matrice *mat, int row, int col) {
    int n = mat->rows;
    printf("Calcul de la matrice mineure pour l'élément [%d][%d]...\n",
row, col);
    Matrice *minor = creer_matrice(n - 1, n - 1);
    if (!minor) return NULL;
    for (int i = 0, mi = 0; i < n; i++) {
        if (i == row) continue;
        for (int j = 0, mj = 0; j < n; j++) {
            if (j == col) continue;
            minor->data[mi][mj++] = mat->data[i][j];
        }
        mi++;
    }
    printf("Matrice mineure calculée:\n");
    for (int i = 0; i < minor->rows; i++) {
        printf("  ");
        for (int j = 0; j < minor->cols; j++) {
            printf("%d ", minor->data[i][j]);
        }
        printf("\n");
    }
    return minor;
}

// Déterminant par expansion
int calculer_determinant_n(Matrice *mat) {
    int n = mat->rows;
    printf("Calcul du déterminant de la matrice %dx%d...\n", n, n);
    if (n == 1) {
        printf("Déterminant (matrice 1x1): %d mod %d = %d\n", mat-
>data[0][0], MODULO, mat->data[0][0] % MODULO);
        return mat->data[0][0] % MODULO;
    }

    int det = 0;
    for (int j = 0; j < mat->cols; j++) {
        printf("\n--- Calcul du terme pour l'élément [%d][%d] (%d) ---
\n", 0, j, mat->data[0][j]);
        Matrice *minor = calculer_matrice_mineure(mat, 0, j);
        if (!minor) return 0;
        int det_minor = calculer_determinant_n(minor);
        int sign = ((j % 2 == 0) ? 1 : -1);
        int term = sign * mat->data[0][j] * det_minor;
        printf("Signe: %d\n", sign);
        printf("Déterminant de la mineure: %d\n", det_minor);
        printf("Terme: %d * %d * %d = %d\n", sign, mat->data[0][j],
det_minor, term);
        det = (det + term) % MODULO;
        printf("Déterminant partiel: %d mod %d = %d\n", det, MODULO, (det
+ MODULO) % MODULO);
        liberer_matrice(minor);
    }
```

```
    printf("Déterminant final de la matrice: %d mod %d = %d\n", det,
MODULO, (det + MODULO) % MODULO);
    return (det + MODULO) % MODULO;
}

// Matrice des cofacteurs
Matrice* calculer_matrice_cofacteurs_n(Matrice *mat) {
    int n = mat->rows;
    printf("Calcul de la matrice des cofacteurs %dx%d...\n", n, n);
    Matrice *cof = creer_matrice(n, n);
    if (!cof) return NULL;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("\n--- Calcul du cofacteur pour l'élément [%d][%d] ---
\n", i, j);
            Matrice *minor = calculer_matrice_mineure(mat, i, j);
            if (!minor) {
                liberer_matrice(cof);
                return NULL;
            }
            int det_minor = calculer_determinant_n(minor);
            int sign = ((i + j) % 2 == 0) ? 1 : -1;
            cof->data[i][j] = (sign * det_minor + MODULO) % MODULO;
            printf("Signe: %d\n", sign);
            printf("Déterminant de la mineure: %d\n", det_minor);
            printf("Cofacteur [%d][%d]: %d * %d mod %d = %d\n", i, j,
sign, det_minor, MODULO, cof->data[i][j]);
            liberer_matrice(minor);
        }
    }
    printf("Matrice des cofacteurs calculée:\n");
    for (int i = 0; i < n; i++) {
        printf("  ");
        for (int j = 0; j < n; j++) {
            printf("%d ", cof->data[i][j]);
        }
        printf("\n");
    }
    return cof;
}

// Transposée
Matrice* transposer_matrice(Matrice *mat) {
    printf("Calcul de la transposée de la matrice %dx%d...\n", mat->rows,
mat->cols);
    Matrice *t = creer_matrice(mat->cols, mat->rows);
    if (!t) return NULL;
    for (int i = 0; i < mat->rows; i++)
        for (int j = 0; j < mat->cols; j++) {
            t->data[j][i] = mat->data[i][j];
        }
    printf("Transposée calculée:\n");
    for (int i = 0; i < t->rows; i++) {
        printf("  ");
        for (int j = 0; j < t->cols; j++) {
            printf("%d ", t->data[i][j]);
        }
        printf("\n");
```

```
    }
    return t;
}

// Adjointe = transposée des cofacteurs
Matrice* calculer_matrice_adjointe_n(Matrice *mat) {
    printf("Calcul de la matrice adjointe...\n");
    Matrice *cof = calculer_matrice_cofacteurs_n(mat);
    if (!cof) return NULL;
    Matrice *adj = transposer_matrice(cof);
    liberer_matrice(cof);
    printf("Matrice adjointe calculée:\n");
    for (int i = 0; i < adj->rows; i++) {
        printf("  ");
        for (int j = 0; j < adj->cols; j++) {
            printf("%d ", adj->data[i][j]);
        }
        printf("\n");
    }
    return adj;
}

// Fonction PGCD étendu
int pgcd_etendu(int a, int b, int *x, int *y) {
    printf("Calcul du PGCD étendu de %d et %d...\n", a, b);
    if (a == 0) {
        *x = 0;
        *y = 1;
        printf("PGCD étendu(%d, %d) = %d, x = %d, y = %d\n", a, b, b, *x,
*y);
        return b;
    }
    int x1, y1;
    int pgcd = pgcd_etendu(b % a, a, &x1, &y1);
    *x = y1 - (b / a) * x1;
    *y = x1;
    printf("PGCD étendu(%d, %d) = %d, x = %d, y = %d\n", a, b, pgcd, *x,
*y);
    return pgcd;
}

// Inverse du déterminant
int calculer_inverse_determinant_n(int det_K) {
    printf("Calcul de l'inverse modulaire du déterminant %d modulo
%d...\n", det_K, MODULO);
    int x, y;
    int pg = pgcd_etendu(det_K, MODULO, &x, &y);
    if (pg == 1) {
        int inv = (x % MODULO + MODULO) % MODULO;
        printf("Inverse modulaire du déterminant: %d (car PGCD(%d, %d) =
1)\n", inv, det_K, MODULO);
        return inv;
    } else {
        printf("Le déterminant %d n'a pas d'inverse modulaire modulo %d
(car PGCD(%d, %d) = %d != 1).\n", det_K, MODULO, det_K, MODULO, pg);
        return -1;
    }
}
```

```
// Multiplication matrice * scalaire
Matrice* multiplier_matrice_scalaire_n(Matrice *mat, int scalaire) {
    printf("Multiplication de la matrice par le scalaire %d...\n",
scalaire);
    Matrice *res = creer_matrice(mat->rows, mat->cols);
    if (!res) return NULL;
    for (int i = 0; i < mat->rows; i++)
        for (int j = 0; j < mat->cols; j++) {
            printf("  %d * %d = %d\n", mat->data[i][j], scalaire, mat-
>data[i][j] * scalaire);
            res->data[i][j] = (mat->data[i][j] * scalaire) % MODULO;
            printf("  Résultat[%d][%d] mod %d = %d\n", i, j, MODULO, res-
>data[i][j]);
        }
    printf("Multiplication scalaire terminée.\n");
    return res;
}

// Inverse matrice
Matrice* calculer_inverse_matrice_cle_n(Matrice *mat) {
    printf("\n--- Calcul de l'inverse de la matrice clé ---\n");
    int det = calculer_determinant_n(mat);
    if (det == 0) {
        printf("Le déterminant est 0, la matrice n'est pas
inversible.\n");
        return NULL;
    }
    int inv_det = calculer_inverse_determinant_n(det);
    if (inv_det == -1) return NULL;

    Matrice *adj = calculer_matrice_adjointe_n(mat);
    if (!adj) return NULL;
    Matrice *inv = multiplier_matrice_scalaire_n(adj, inv_det);
    liberer_matrice(adj);
    printf("--- Inverse de la matrice clé calculée ---\n");
    if (inv) {
        printf("Inverse de la matrice clé:\n");
        for (int i = 0; i < inv->rows; i++) {
            printf("  ");
            for (int j = 0; j < inv->cols; j++) {
                printf("%d ", inv->data[i][j]);
            }
            printf("\n");
        }
    }
    return inv;
}

// Déchiffrement
char* dechiffrer_hill_n(char *texte_chiffre, Matrice *cle_inverse) {
    int n = cle_inverse->rows;
    printf("\n--- Début du Déchiffrement (taille de clé inverse %dx%d) --
-\n", n, n);
    int len = strlen(texte_chiffre);
    char *clair = malloc(len + 1);

    printf("Texte chiffré à déchiffrer: \"%s\"\n", texte_chiffre);
```

```
    for (int i = 0; i < len; i += n) {
        printf("\nTraitement du bloc chiffré %d-%d: ", i, i + n - 1);
        int vecteur[n], res[n];
        printf("Vecteur bloc chiffré (caractères -> nombres): ");
        for (int j = 0; j < n; j++) {
            vecteur[j] = caractere_vers_nombre(texte_chiffre[i + j]);
            printf("%c(%d) ", texte_chiffre[i + j], vecteur[j]);
        }
        printf("\n");
        multiplier_matrice_vecteur(cle_inverse, vecteur, res);
        printf("Bloc déchiffré (nombres -> caractères): ");
        for (int j = 0; j < n; j++) {
            clair[i + j] = nombre_vers_caractere(res[j]);
            printf("%d(%c) ", res[j], clair[i + j]);
        }
        printf("\n");
    }
    clair[len] = '\0';
    printf("--- Fin du Déchiffrement ---\n");
    return clair;
}

// Fonction principale
int main() {
    int choix;
    do {
        printf("\n=== MENU HILL CHIFFREMENT / DECHIFFREMENT ===\n");
        printf("1. Chiffrer un message\n");
        printf("2. Déchiffrer un message\n");
        printf("3. Quitter\n");
        printf("Votre choix : ");
        if (scanf("%d", &choix) != 1) {
            printf("Erreur: Veuillez entrer un nombre.\n");
            while (getchar() != '\n'); // Nettoyer le tampon d'entrée
            choix = -1; // Pour continuer la boucle
            continue;
        }
        while (getchar() != '\n'); // Nettoyer le tampon d'entrée

        switch (choix) {
            case 1: {
                printf("\n--- Chiffrement ---\n");
                int n;
                printf("Entrez la taille de la matrice clé (n x n) : ");
                if (scanf("%d", &n) != 1 || n <= 0) {
                    printf("Erreur: Veuillez entrer un entier positif
pour la taille de la clé.\n");
                    while (getchar() != '\n');
                    break;
                }
                while (getchar() != '\n');

                Matrice *cle = lire_matrice_cle_n(n);
                if (cle == NULL) {
                    printf("Erreur lors de la lecture de la matrice
clé.\n");
                    break;
```

```
                }

                char *texte = lire_message();
                if (texte == NULL) {
                    printf("Erreur lors de la lecture du message.\n");
                    liberer_matrice(cle);
                    break;
                }

                char *texte_complet = completer_texte_n(texte, n);
                char *texte_chiffre = chiffrer_hill_n(texte_complet,
cle);
                printf("Message chiffré : %s\n", texte_chiffre);

                free(texte);
                if (texte_complet != texte) free(texte_complet);
                free(texte_chiffre);
                liberer_matrice(cle);
                break;
            }

            case 2: {
                printf("\n--- Déchiffrement ---\n");
                int n;
                printf("Entrez la taille de la matrice clé (n x n) : ");
                if (scanf("%d", &n) != 1 || n <= 0) {
                    printf("Erreur: Veuillez entrer un entier positif
pour la taille de la clé.\n");
                    while (getchar() != '\n');
                    break;
                }
                while (getchar() != '\n');

                Matrice *cle = lire_matrice_cle_n(n);
                if (cle == NULL) {
                    printf("Erreur lors de la lecture de la matrice
clé.\n");
                    break;
                }

                Matrice *cle_inverse =
calculer_inverse_matrice_cle_n(cle);
                if (cle_inverse == NULL) {
                    printf("La matrice clé n'est pas inversible modulo 37
(PGCD(det, 37) != 1). Déchiffrement impossible.\n");
                    liberer_matrice(cle);
                    break;
                }

                char *texte_chiffre = lire_message();
                if (texte_chiffre == NULL) {
                    printf("Erreur lors de la lecture du texte
chiffré.\n");
                    liberer_matrice(cle);
                    liberer_matrice(cle_inverse);
                    break;
                }
```

```
                char *texte_dechiffre = dechiffrer_hill_n(texte_chiffre,
cle_inverse);
                printf("Message déchiffré : %s\n", texte_dechiffre);

                free(texte_chiffre);
                free(texte_dechiffre);
                liberer_matrice(cle);
                liberer_matrice(cle_inverse);
                break;
            }

            case 3:
                printf("Fin du programme.\n");
                break;

            default:
                printf("Choix invalide. Veuillez réessayer.\n");
        }

    } while (choix != 3);

    return 0;
}
```