المدرسة العليا للتكنولوجيا - الداخلة
ⵜⵜⵉⵛⵏ ⵜⴰⵙⴷⴰⵡⵉⵜ ⵏ ⵜⵜⵣⵔⵉⵙⵜ - ⴷⴰⵅⵍⴰ
ÉCOLE SUPÉRIEURE DE TECHNOLOGIE - DAKHLA

Le Compte Rendu du TD3/TP3 Cryptographie

Filière : *Cybersécurité & Intelligence Artificielle*

Module : *Cryptographie*

Chiffrement à clé publique : RSA

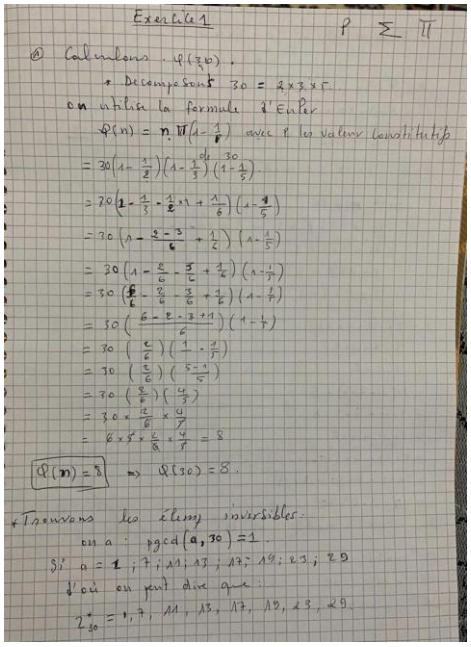
Année Universitaire : 2025 – 2026

Réalisé par : YAOGO Lucien

Encadré par : Pr. KHADIM

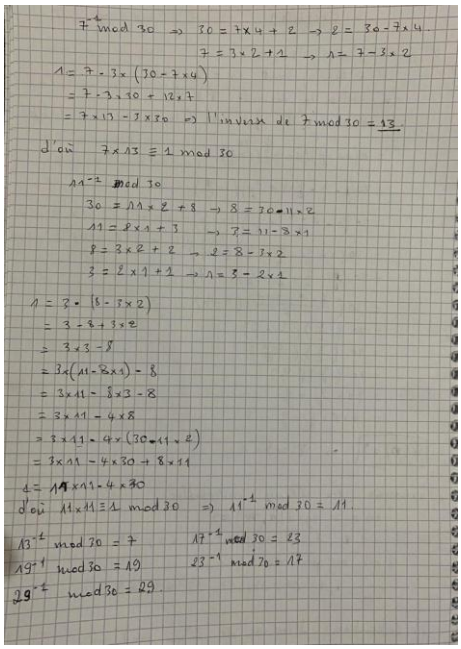
Exercice 1

1) Calcul de (φ (30))



Nous avons calculé **φ (30)** afin de déterminer le nombre d'éléments inversibles modulo 30, car la fonction d'Euler indique combien d'entiers sont premiers avec 30.

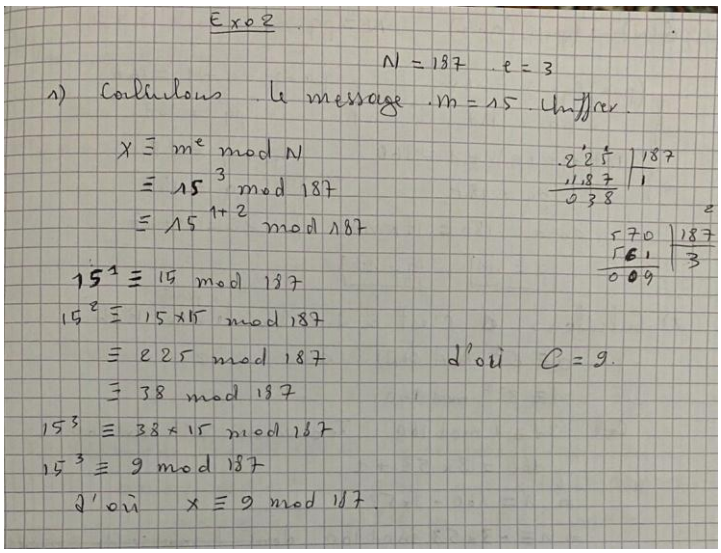
2) Détermination des éléments inversibles



Nous avons recherché les éléments de $\mathbf{Z_{30}}$ tels que **pgcd(a,30) = 1**, car seuls ces éléments possèdent un inverse multiplicatif modulo 30.

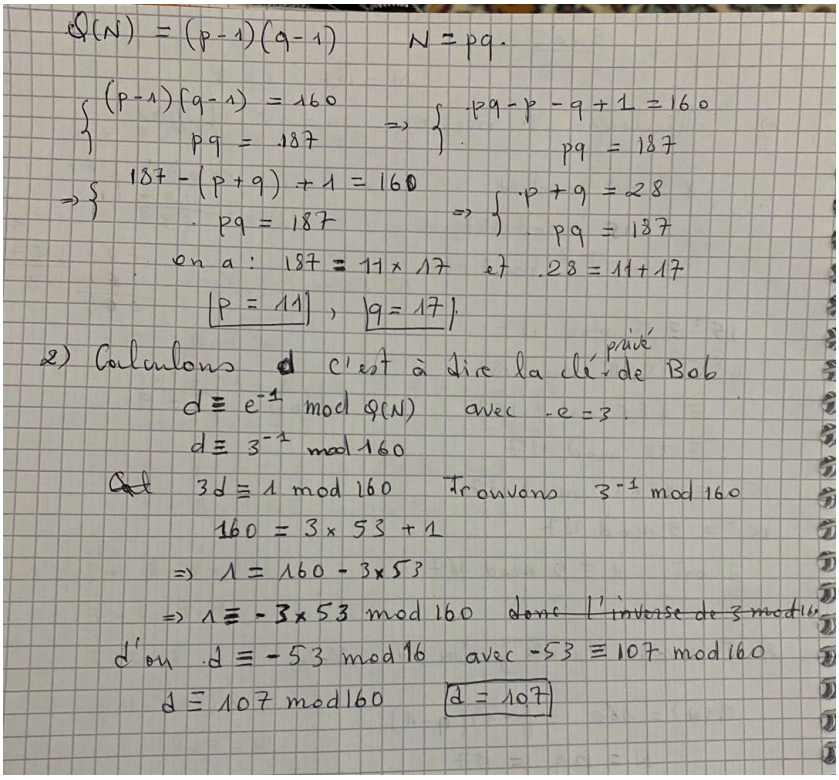
Exercice 2

1) Chiffrement du message



Nous avons appliqué la formule **c = m^e mod N** afin de transformer le message clair en message chiffré en utilisant la clé publique de Bob.

2) Retrouver la factorisation de N



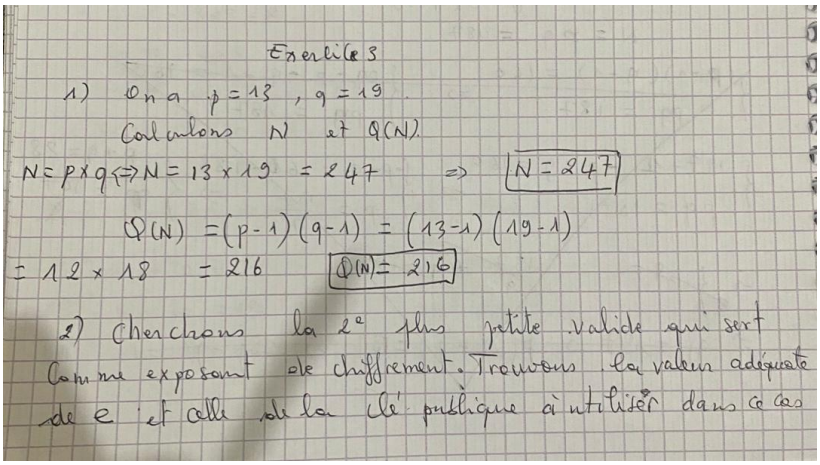
Nous avons utilisé la relation **φ (N) = (p-1) (q-1)** pour retrouver les facteurs premiers de N, car connaître **φ (N)** permet de casser la sécurité du système RSA.

Calcul de la clé privée

Nous avons calculé l'inverse modulaire de e modulo **φ (N)** afin d'obtenir la clé privée **d**, qui permet le déchiffrement des messages. D'où la clé privée de Bob est : **(107, 247)**

Exercice 3

1) Calcul de n


Exercice 3

1) On a p = 13 , q = 19.

Calculons N et φ(N).

N = p×q ⇔ N = 13 × 19 = 247 ⇒ N = 247

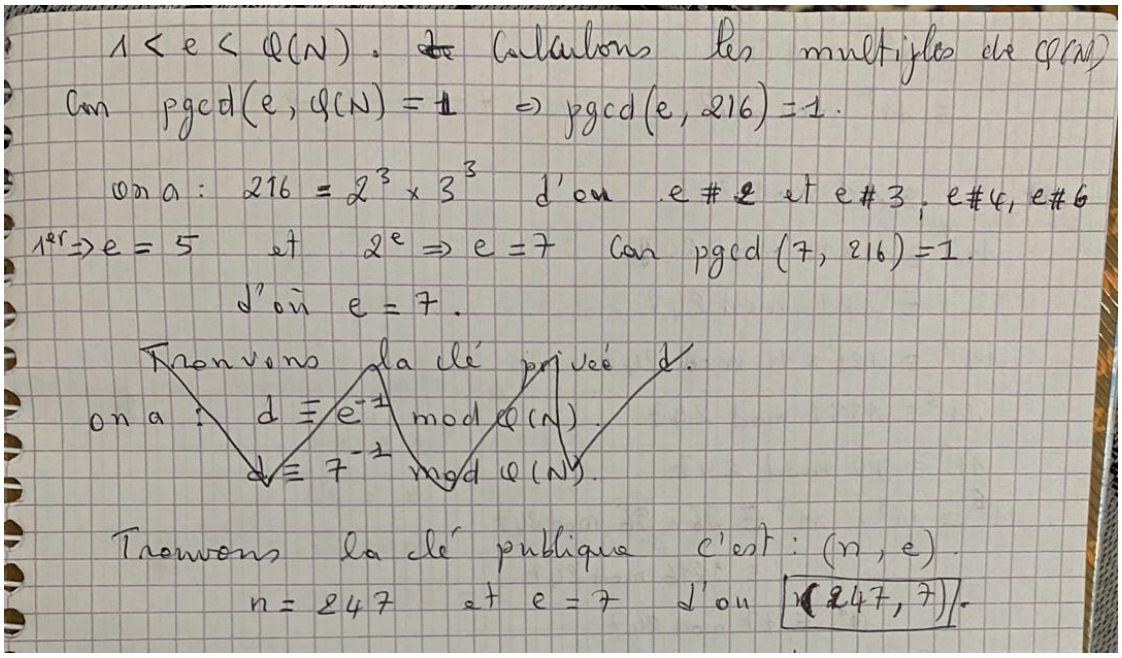
φ(N) = (p-1)(q-1) = (13-1)(19-1)

= 12 × 18 = 216 φ(N) = 216

2) Cherchons la 2e plus petite valide qui sert comme exposant de chiffrement. Trouvons la valeur adéquate de e et celle de la clé publique à utiliser dans ce cas

Nous avons multiplié p et q pour obtenir n, car le module n est utilisé dans les opérations de chiffrement et de déchiffrement.

2) Calcul de l'exposant e


1 < e < φ(N). Calculons les multiples de φ(N)

Car pgcd(e, φ(N)) = 1 ⇒ pgcd(e, 216) = 1.

On a : 216 = $2^3 \times 3^3$ d'où e ≠ 2 et e ≠ 3 ; e ≠ 4, e ≠ 6

1er ⇒ e = 5 et 2e ⇒ e = 7 Car pgcd(7, 216) = 1.

d'où e = 7.

Trouvons la clé privée d.

on a : $d \equiv e^{-1} \bmod \varphi(n)$.

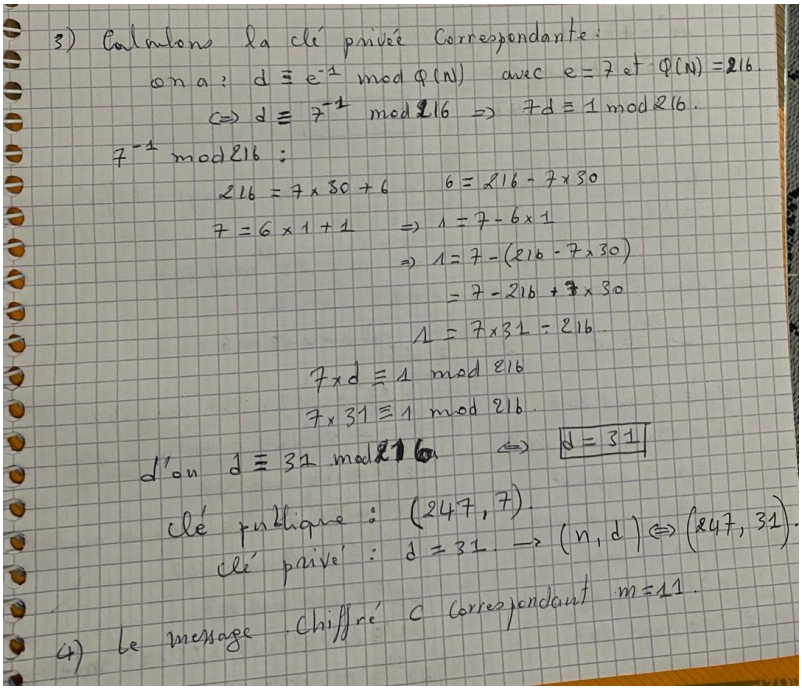
$d \equiv 7^{-1} \bmod \varphi(N)$.

Trouvons la clé publique c'est : (n, e)

n = 247 et e = 7 d'où (247, 7).

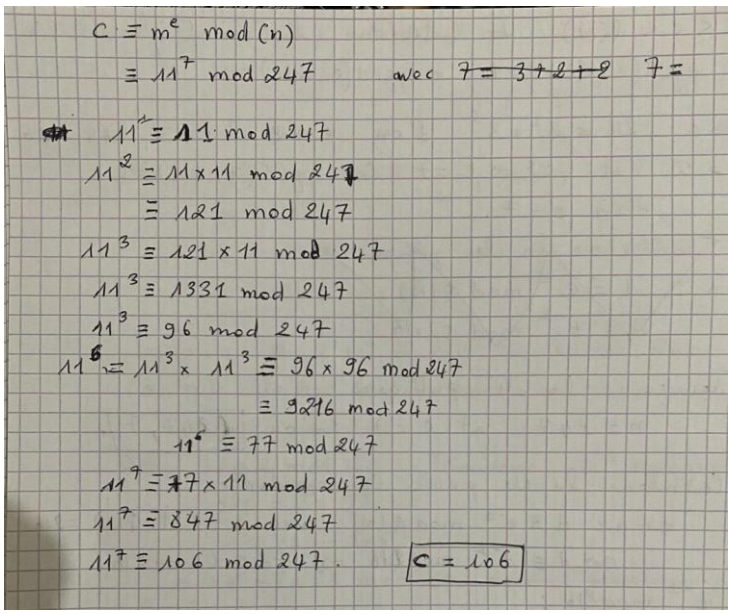
Nous avons calculé **φ (n) = (p-1) (q-1)** afin de pouvoir déterminer une clé privée valide.
Nous avons choisi un entier e premier avec **φ (n)** afin de garantir l'existence d'un inverse modulaire nécessaire au calcul de la clé privée.

3) Calcul de la clé privée d



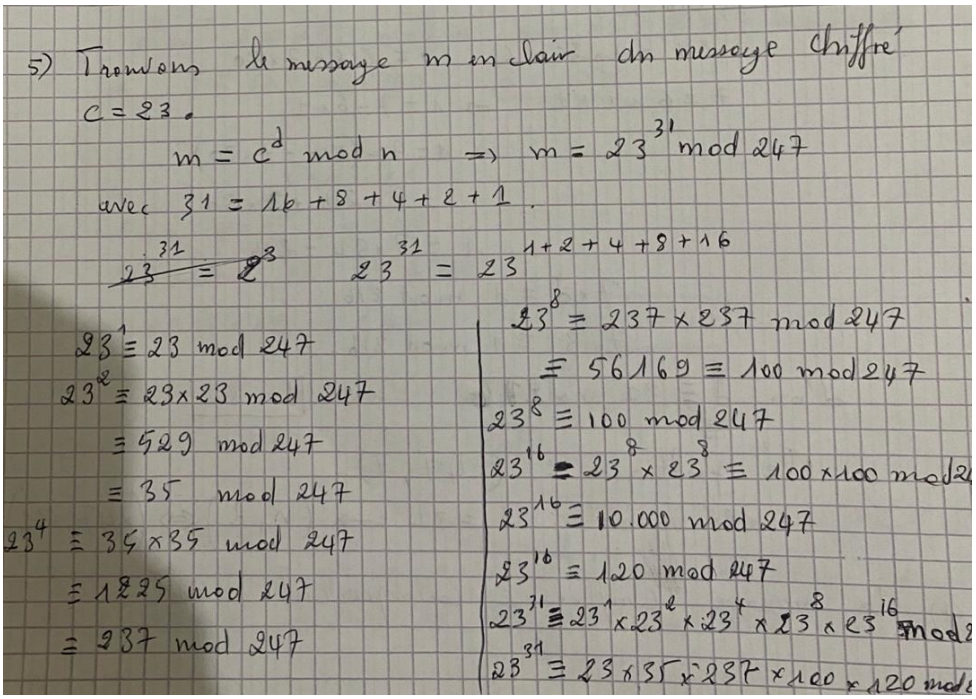
Nous avons déterminé **d** tel que : **ed ≡ 1 mod φ (n)** afin que le déchiffrement annule l'effet du chiffrement.

4) Chiffrement du message m = 11



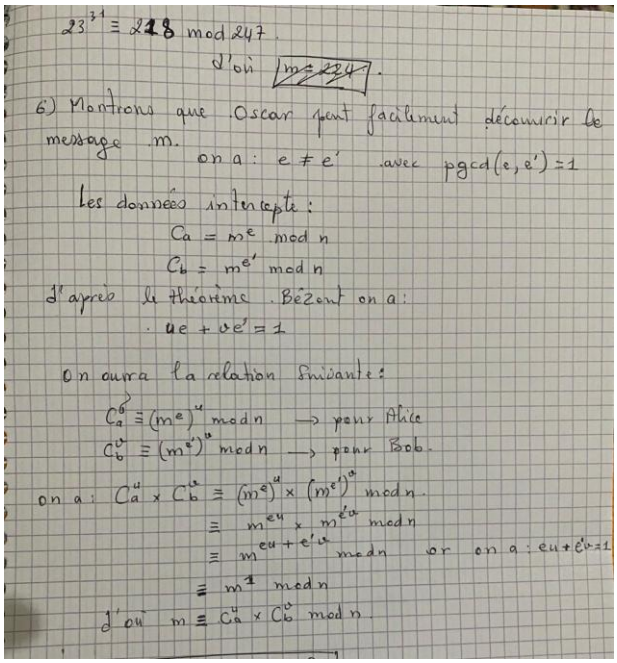
Nous avons calculé **c = m^e mod N** afin de produire le message chiffré à transmettre.

5) Déchiffrement du message c = 23



Nous avons calculé **m = c^{d} mod n** afin de retrouver le message clair à partir du message chiffré.

Oscar peut retrouver le message m



Nous avons montré que si deux utilisateurs partagent le même module n avec des exposants premiers entre eux, il est possible de combiner les deux cryptogrammes pour retrouver directement le message original en utilisant le théorème de Bézout.

Exercice 4

1) Fonction `premier (int n)`

```c
#include <stdio.h>

// 1) Fonction pour tester si un nombre est premier

int premier(int n) {

    if (n <= 1) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;

    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return 0;
    }
    return 1;
}

// La fonction principale pour le teste.

int main(){

    int a;
    printf("%d Entrez la valeur de a : ");
    scanf("%d", &a);
    int p = premier(a);

    if(p){
        printf("%d est un nombre premier ", a);
    }else{
        printf("%d n'est pas un nombre premier ", a);
    }

    return 0;
}
```

Nous avons écrit une fonction permettant de vérifier si un entier est premier, car RSA nécessite la génération de nombres premiers pour construire le module n.

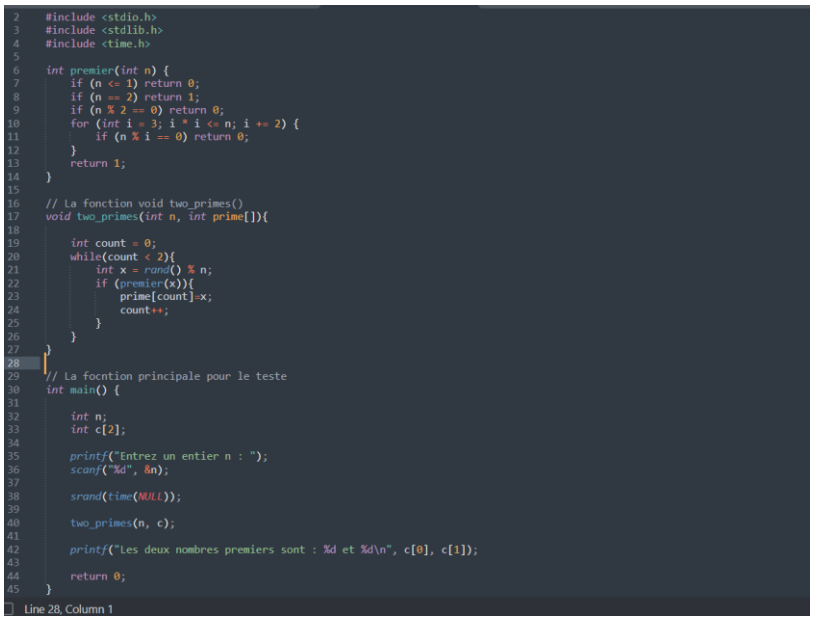
2) Fonction `factorisation (int n)`

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

int premier(int n) {

    if (n <= 1) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;

    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return 0;
    }
    return 1;
}

int factorisation(int n) {

    for (int i = 2; i <= n / 2; i++) {
        if (n % i == 0) {
            int j = n / i;
            if (premier(i) && premier(j)) {
                return i;
            }
        }
    }
    return n;
}

// La fonction principale pour le teste.
int main() {

    int n;
    printf("Entrez un entier n : ");
    scanf("%d", &n);

    int p = factorisation(n);
    if (p != n) {
        printf("n = %d x %d avec n premiers\n", p, n / p);
    } else {
        printf("n ne peut pas etre factorise en deux nombres premiers.\n");
    }
    return 0;
}
```

Line 30, Column 41

Nous avons implémenté une fonction de factorisation afin de comprendre que la sécurité de RSA repose sur la difficulté de décomposer **n** en produit de deux nombres premiers.

3) Fonction `two_primes (int n, int prime [])`



```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int premier(int n) {
    if (n <= 1) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;
    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return 0;
    }
    return 1;
}

// La fonction void two_primes()
void two_primes(int n, int prime[]){

    int count = 0;
    while(count < 2){
        int x = rand() % n;
        if (premier(x)){
            prime[count]=x;
            count++;
        }
    }
}

// La focntion principale pour le teste
int main() {

    int n;
    int c[2];

    printf("Entrez un entier n : ");
    scanf("%d", &n);

    srand(time(NULL));

    two_primes(n, c);

    printf("Les deux nombres premiers sont : %d et %d\n", c[0], c[1]);

    return 0;
}
```

Line 28, Column 1

Nous avons créé une fonction générant deux nombres premiers aléatoires inférieurs à n afin d'automatiser la génération des paramètres p et q du système RSA.

4) Fonction `keys (int Pk [], int Sk [])`

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>


int pgcd(int a, int b) {
    int temp;
    while (b != 0) {
        temp = b;
        b = a % b;
        a = temp;
    }

    return a;
}

int premier(int n) {

    if (n <= 1) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;

    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) return 0;
    }
    return 1;

}

void two_primes(int n, int prime[]){
    int count = 0;
    while(count < 2){
        int x = rand() % n;
        if (premier(x)){
            prime[count]=x;
            count++;
        }
    }
}
```

```c
void key(int pk[], int sk[]){
    int p, q, n, phi, e, d;
    int prime[2];

    two_primes(50, prime);
    p=prime[0];
    q=prime[1];
    n = p * q;
    phi = (p-1) * (q-1);
    e=3;

    while(pgcd(e, phi)!= 1) e++;
    for(d=1; d<phi; d++){
        if ((e * d) % phi == 1) break;
    }

    pk[0]=n;
    sk[0]=n;

    pk[1]=e;
    sk[1]=d;

}


int main() {
    int pk[2];
    int sk[2];

    srand(time(NULL));
    key(pk, sk);

    printf("=== Cle publique ===\n");
    printf("n = %d\n", pk[0]);
    printf("e = %d\n", pk[1]);

    printf("\n=== Cle secrete ===\n");
    printf("n = %d\n", sk[0]);
    printf("d = %d\n", sk[1]);

    return 0;
}
```

Nous avons développé une fonction permettant de générer automatiquement la clé publique et la clé privée afin de simuler le fonctionnement complet du protocole RSA.

5) Fonction `to_bin(int bin[], int n)`

```c
#include <stdio.h>

void to_bin(int bin[], int n){

    int i = 0;

    while(n > 0){
        bin[i] = n % 2;
        n/= 2;
        i++;
    }
}


int main() {

    int n;
    int bin[32];
    int i = 0;

    printf("Entrez un entier : ");
    scanf("%d", &n);

    to_bin(bin, n);

    printf("Representation binaire : ");

    while(n > 0){
        printf("%d", bin[i]);
        i++;
        n /= 2;
    }

    return 0;
}
```

Nous avons implémenté cette fonction pour convertir un entier en représentation binaire, ce qui facilite l'implémentation efficace de l'exponentiation modulaire.

6) Fonction `Expo_modulaire(int a, int b, int n)`

```c

#include <stdio.h>

int expo_modulaire(int a, int b, int n){

    int rest = 1;
    a = a % n;

    while(b > 0){

        if(b % 2 == 1){
            rest = (rest * a) % n;
            b/= 2;
            a = (a * a) % n;
        }
    }

    return rest;
}

int main(){


    int a, b, n;

    printf("Entrez la valeur de a : ");
    scanf("%d", &a);

    printf("Entrez la valeur de b : ");
    scanf("%d", &b);

    printf("Entrez la valeur de n : ");
    scanf("%d", &n);

    int p = expo_modulaire(a, b, n);

    printf("a^b est : %d", p);
    return 0;
}
```

Nous avons programmé l'algorithme d'exponentiation modulaire rapide afin de calculer efficacement $\mathbf{a^b \bmod n}$, opération essentielle pour le chiffrement et le déchiffrement RSA.

Conclusion générale

Ce TP nous a permis de comprendre que chaque calcul effectué dans RSA, du choix des nombres premiers au calcul de l'inverse modulaire, joue un rôle essentiel dans la sécurité et le bon fonctionnement du système.