Réseau et sécurité

M1 informatique

Implémentation de RSA

Dans ce TP, nous allons implémenter l'algorithme RSA, à partir de notre implémentation d'une librairie de grands nombres.

Compte rendu à envoyer à frederic.hayek@uca.fr avant le jeudi 31 mars à 23:59 heure de Paris.

On préférera l'utilisation du langage C pour ce TP. À défaut, un autre langage bas niveau fera l'affaire.

Partie 1 – Manipulation des grands nombres (début)

Exercice 1.1 : Ecrivez une structure de données qui représente un nombre entier comme un tableau de x caractères, avec x connu à l'avance. Chaque caractère représente un bit du nombre. On utilisera généralement x=100 (ou une valeur inférieure pour tester les fonctions).

Exercice 1.2 : Ecrivez une fonction *initialiser0* qui crée un nombre valant 0.

Exercice 1.3 : Ecrivez une fonction *initialiser1* qui crée un nombre valant 1.

Exercice 1.4 : Ecrivez une fonction de libération de la mémoire associée à un nombre, appelée *libererNombre*.

Exercice 1.5 : Ecrivez une fonction *afficher* qui permet l'affichage (en binaire) du nombre.

Exercice 1.6 : Ecrivez une fonction *comparer* qui indique si deux nombres sont égaux ou non.

Exercice 1.7 : Ecrivez une fonction *estPair* qui détermine si un nombre n est pair ou non. Pour cela, il suffit de regarder la parité du dernier bit.

Exercice 1.8 : Ecrivez une fonction *diviserPar2* qui divise un nombre n par deux, en l'arrondissant à la valeur inférieure si besoin. Pour cela, il suffit de supprimer le dernier bit.

Exercice 1.9 : Ecrivez une fonction *reduireDe1* qui retire 1 au nombre.

Exercice 1.10 : Ecrivez une fonction *multiplierPar2* qui multiplie un nombre n par deux. Pour cela, il suffit d'ajouter un dernier bit à 0.

Exercice 1.11 : Ecrivez une fonction *ajouter* qui ajoute deux nombres a et b. Par exemple, pour calculer 11001+01100, il faut :

- Additionner les bits deux à deux, de droite à gauche : 1+0=1, 0+0=0, 0+1=1, 1+1=0 avec retenue, 1+0+retenue = 0 avec retenue, et 0+0+retenue=1.
- Reprendre tous ces bits du dernier au premier, soit 100101.

Exercice 1.12 : Ecrivez une fonction *multiplier* qui prend en paramètres deux nombres a et b, et qui les multiplie. Par exemple, pour calculer a*11010, il faut :

- Calculer a*0 (qui fait 0).
- Calculer a*10 ; 10 s'obtient en multipliant 1 fois le nombre 1 par 2.
- Calculer a*000 (qui fait 0).
- Calculer a*1000 ; 1000 s'obtient en multipliant 3 fois le nombre 1 par 2.
- Calculer a*10000 ; 10000 s'obtient en multipliant 4 fois le nombre 1 par 2.

- Ajouter tous ces résultats.

Exercice 1.13 : Ecrivez une fonction *exponentiationRapideSansModulo*, qui prend en paramètre un nombre a et un nombre b, et qui calcule a^b. Par exemple, pour calculer a^11001, il faut considérer les bits de la puissance de droite à gauche de la manière suivante :

- Partir de x=1 et de base=a.
- Calculer x=base*x pour le bit 1 et base=base^2.
- Ne pas changer x pour le bit 0 et base=base^2.
- Ne pas changer x pour le bit 0 et base=base^2.
- Calculer x=base*x pour le bit 1 et base=base^2.
- Calculer x=base*x pour le bit 1 et base=base^2.
- Le résultat est x.

Exercice 1.14 : Ecrivez une fonction *soustraire*, qui permet de calculer a-b.

Exercice 1.15 : Ecrivez une fonction *modulo* qui prend en paramètre un nombre a et un nombre n, et qui calcule a [n].

- Une solution consiste à soustraire de a la valeur n plusieurs fois, jusqu'à obtenir un nombre compris entre 0 et n-1. Mais, cette solution est trop lente.
- Une solution consiste à diviser a par n et obtenir q, puis calculer a-(n x q), mais cette solution nécessite d'implémenter la division.
- La solution que nous allons utiliser est hybride entre les deux premières solutions. Elle fait une approximation (d'une borne inférieure) du quotient de a par n, que nous allons noter k, et effectue des soustractions. En effet, on peut soustraire n x k de a sans changer la valeur du modulo. Par exemple, pour calculer 1111011b modulo 1101b, on peut :
    - Approximer 1101b en 10000b, et voire que 1111011b est supérieur à 10000b x 100b. On peut donc retirer 1101b x 100b de 1111011b. On obtient 1000111b.
    - On soustrait 1101b du résultat jusqu'à obtenir un nombre sur moins de bit qu'à l'étape précédente. On obtient 111010b (après une soustraction dans ce cas).
    - Approximer 1101b en 10000b, et voire que 111010b est supérieur à 1000b x 10b. On peut donc retirer 1101b x 10b de 111010b. On obtient 100000b.
    - On soustrait 1101b du résultat jusqu'à obtenir un nombre sur moins de bit qu'à létape précédente. On obtient 10011b (après une soustraction dans ce cas).
    - Approximer 1101b en 10000b, et voire que 10011b est supérieur à 10000b x 1b. On peut donc retirer 1101b x 1b de 10011b. On obtient 110b.
    - 110b est inférieur à 1101b, donc il s'agit du modulo recherché (en effet, on cherchait à faire le modulo de 123 par 13, ce qui donne bien 6).

Exercice 1.15 : Ecrivez une fonction *exponentiationRapide*, qui prend en paramètre trois nombres a, b et n, et qui calcule a^b [n]. La fonction calcule le modulo à chaque étape de l'exponentiation rapide.

Partie 2 – Chiffrement et déchiffrement

Exercice 2.1 : Avec OpenSSL, créez un couple de clés publique-privé sur 100 bits, où (e,n) est la clé publique et (d,n) est la clé privée. Stockez dans votre programme les valeurs correspondantes.

Le chiffrement d'un message m avec RSA se fait en calculant m^e [n].

Exercice 2.2 : Ecrivez un message m d'au plus 12 octets (12 octets = 96 bits < 100 bits), et sauvegardez-le dans un fichier texte.

Exercice 2.3 : Ecrivez la fonction qui calcule m^e [n], en utilisant l'exponentiation rapide et en effectuant le calcul du modulo n à chaque étape de l'exponentiation rapide.

Le déchiffrement d'un message c avec RSA se fait en calculant c^d [n].

Exercice 2.4 : Ecrivez la fonction qui calcule c^d [n], en utilisant l'exponentiation rapide et en effectuant le calcul du modulo n à chaque étape de l'exponentiation rapide.

Exercice 2.5 : Chiffrez votre message m, puis déchiffrez-le, et vérifiez que vous obtenez bien le message initial.

Exercice 2.6 : Modifiez votre programme pour qu'il puisse chiffrer et déchiffrer un fichier texte de taille quelconque. Pour cela, il faudra scinder le fichier texte en blocs de bits, et chiffrer indépendemment chaque bloc.

Partie 3 – Manipulation des grands nombres (fin)

Exercice 3.1 : Le calcul de l'inverse de a modulo b se fait au moyen de l'algorithme d'Euclide étendu.

- Implémentez l'algorithme *inverserAvecModulo* (sa description se trouve sur Wikipedia par exemple).
- Vous aurez besoin de calculer le quotient de la division euclidienne : il s'agit du nombre de soustractions réalisées dans l'exercice 1.15 (100b soustractions à l'étape 1 + 1 soustraction à l'étape 2 + 10b soustractions à l'étape 3 + 1 soustraction à l'étape 4 + 1b soustraction à l'étape 5 = 100b+1+10b+1+1b = 4+1+2+1+1 = 9, ce qui est bien le quotient).

Exercice 3.2 : Pour déterminer si deux nombres a et b sont premiers entre eux, il suffit de déterminer si le plus grand diviseur commun entre a et b est égal à 1 ou non. A partir de l'algorithme d'Euclide étendu, implémentez la fonction *sontPremiersEntreEux*.

Exercice 3.3 : Pour implémenter un nombre premier aléatoire, nous allons utiliser les nombres de Mersenne. Un nombre de Mersenne est un nombre de la forme Mn = 2^n-1. Un nombre premier de Mersenne est un nombre de Mersenne qui est premier. Pour que Mn soit premier, il faut (mais il ne suffit pas) que n soit premier. Par exemple, M4 n'est pas premier car n=4 n'est pas premier. Implémentez une fonction nombrePremierMersenne qui retourne un nombre premier de Mersenne, avec n compris aléatoirement entre 80 et 700 (ce qui génèrera des nombres premiers ayant entre 27 et 183 chiffres). On fera l'hypothèse que si n est premier alors Mn est premier.

Partie 4 – Création des clés

La création des clés dans RSA se fait de la manière suivante :

- Choisir deux nombres p et q distincts, aléatoirement et tous les deux premiers.
- Calculer n = p x q.
- Calculer phi(n) = (p-1) x (q-1).
- Choisir arbitrairement un nombre e, compris entre 1 et phi(n), et premier avec phi(n).
- Calculer d, l'inverse de e modulo phi(n).

Exercice 4.1 : Implémentez la génération des clés.

Exercice 4.2 : Remplacez les valeurs de votre couple de clés de l'exercice 2.1 par des valeurs calculées par votre programme, et vérifiez que le chiffrement/déchiffrement fonctionne toujours.

Exercice 4.3 : Effectuez des tests de performance, en calculant le temps nécessaire pour chiffrer un message donné en fonction de la taille de la clé.

Bonus : Que se passe-t-il si les nombres générés, p et q, ne sont pas premiers ? Et que se passe-t-il si le nombre généré, e, n'est pas premier avec phi(n) ?

Remarque : C'est devenu courant aujourd'hui, de d'abord choisir e, puis de choisir p et q tel que e ne divise ni (p-1) ni (q-1).