$\mathcal{L}$anguage $\mathcal{C}$ — PROJET/PROJECT S5

Équipe Pédagogique de l'ESIEA

Binary Algorithm for Calculating GCD: Oh "BigDigits"! !

Table des matières

1 Avant propos

Avant-propos : On décrit ici les éléments nécessaires aux Phases 2 et 3

1.1 (Rappel) Phase Deux : Fonctionnalités Avancées

Il y a un ajout : la Multiplication Égyptienne (oui, on en a besoin). La bonne nouvelle c'est que ce très ancien algorithme, encore plus vieux que l'Algorithme d'Euclide, est parfaitement adapté aux grands entiers binaires.

- Calcul du PGCD de deux "nombres binaires" par l'Algorithme Binaire d'Euclide.
- Calcul du modulo de A par N
- Multiplication Égyptienne (oui, on en a besoin)
- Calcul Exponentiation rapide modulaire

1.2 (Rappel) Phase Trois : RSA simplifié (Bonus)

Les "bonus" sont pour celles et ceux qui vont très vite.

1. Chiffrement RSA
2. Déchiffrement RSA

2 Calcul Modulaire "adapté" à nos besoins

Définissons la fonction *Modulo* par analogie avec le calcul du PGCD par soustraction dans la version d'Euclide [1] (le fameux *Algorithme d'Euclide*) :

1. On suppose a et b positifs
2. Si $a = b$ alors $Modulo(a, b) = 0$ car $a = a * 1 + 0$
3. Si $a > b$ alors : $Modulo(a, b) = Modulo(a - b, b)$ avec $a - b > 0$
4. Si $a > 2 * b$ alors : $Modulo(a, b) = Modulo(a - 2 * b, b)$ avec $a - 2 * b > 0$
5. Si $a > 3 * b$ alors : $Modulo(a, b) = Modulo(a - 3 * b, b)$ avec $a - 3 * b > 0$
6. etc.
7. Et pour finir : **Si** $a < b$ **alors** $Modulo(a, b) = a$ **car** $a = b * 0 + a$

Ce qui nous permet d'écrire : si $a = b * q + r$ avec $0 < 2^k \leq q < 2^{k+1}$ $(k \geq 0)$ alors

$$\forall i \in \{1, \cdots k\} : Modulo(a, b) = Modulo(a - 2^i * b, b) \tag{1}$$

Comment continuer ?

1. **Il suffit de voir que comme** $Modulo(a, b) = Modulo(a - 2^i * b, b)$,
2. Le meilleur choix de i est donc : k tel que $0 < 2^k \leq q < 2^{k+1}$
3. **On remplace donc** a **par** $a - 2^k * b$ **et on continue . . .**
4. On vous **laisse le soin de trouver le bon "test d'arrêt".**

Cet algorithme est certes *un peu moins efficace* dans certains cas que la *Division Euclidienne classique* passant par le calcul du quotient qui donne ensuite le modulo, car il nécessite plusieurs soustractions et additions pour chaque itération. **Cependant :**

1. Appelée *antiphérèse* par Euclide lui-même.

1. Il est plus facile que les méthodes dites "rapides" et qui souvent ne sont intéressantes que pour des entiers très très très longs (de quelques dizaines de milliers de chiffres à quelques millions, et nous en sommes loin)
2. Il est utile pour comprendre les principes de base du calcul du modulo et de la division
3. Mais surtout : **il est beaucoup plus facile à implémenter**
4. **c'est pourquoi nous vous demandons de l'utiliser**.
5. Remarque : cet algorithme devient vraiment intéressant si on voit et si on utilise le fait que : $2*b = b+b$, puis que $4*b = 2*b+2*b$ etc. en clair : on n'a que des additions à faire pour ce calcul, ce qui le rend assez rapide.

***Attention/Pay attention** [1]:*

Donnons un exemple, avec tous les calculs (en base 10 pour simplifier) : soient $A = 192$ et $B = 33$, on veut calculer $Modulo(A, B)$ avec "notre" algorithme soit $A \bmod B = 192 \bmod 33 = 27$:

1. Première étape : on part avec $A = 192$ et $B = 33$
2. On "calcule" $A - B = 192 - 33$ qui est est positif, donc ...
3. On "calcule" $A - 2*B = 192 - 2*33$ qui est positif, donc ...
4. On "calcule" $A - 4*B = 192 - 4*33$ qui est positif, donc ...
5. On "calcule" $A - 8*B = 192 - 8*33$ qui est **négatif** donc ...
6. On remplace $Modulo(A, B)$ par $Modulo(A-4*33, B)$ c'est à dire qu'on "écrase" la valeur de A par affectation : $A = A - 4*B$
7. Ce qui donne $Modulo(A - 4*33, B) = Modulo(192 - 4*33, 33) = Modulo(60, 33)$
8. Deuxième étape : on part cete fois-ci avec $A = 60$ et $B = 33$
9. Comme $60 - 33 > 0$ et que $60 - 2*33 < 0$, on aura donc
10. Finalement : $Modulo(192, 33) = Modulo(60, 33) = 27$ sans aucune division et sans multiplications !

***Conseil/Advice** [1]*

Cet algorithme a un nom : *Division-free mod* soit *modulo sans division*. Voir le magnifique ouvrage de Crandall et Pommerance *Prime Numbers : A computational Perspective* (page 452 : Algorithm 9.2.10). Techniquement la version qui vous est proposée ici devrait s'appeler *Division-And-Multiplication free mod* soit *modulo sans division et sans multiplications* grâce à l'astuce $2*x$ remplacé par $x+x$! Le pdf de la version 2 de l'ouvrage cité est facile à trouver de "manière légale", avec une requête dans votre moteur de recherche préféré.

Indice(s):

L'élève attentif aura remarqué que cet algorithme revient en fait à calculer $Modulo(A, B)$ en "estimant" le quotient de A par B "bit par bit" et à retrancher successivement ce qu'il faut pour obtenir le reste de A par B.

3 Exponentiation modulaire

Maintenant, nous avons tout ce dont nous avons besoin pour programmer la fonction Exponentiation modulaire :

1. Soient A un GrandEntier Binaire ("*BigBinary*")
2. Soit N un GrandEntier Binaire ("*BigBinary*")
3. Et e **un entier** (pour simplifier ! e sera donc un `unsigned int`)
4. On suppose que $0 < A < N$
5. On suppose que $0 < e < N$
6. **On désire calculer A^e mod N**
7. On vous propose de programmer cette fonction en utilisant l'un des algorithmes que nous avons déjà vus pour le calcul de l'exponentiation (dite) rapide
8. Il vous suffit juste de l'adapter à la struct "GrandEntier Binaire ("*BigBinary*")" ...
9. À vous de jouer.

3.1 Voici un jeu de données pour tester vos fonctions :

— $N = 115792089237342390393211096473342943949313232746809673270778931389365205055153 =$
$(1000000000000000000000000000000000000000000000011111110101101001010110110110010110010101110101$

— $A = 184370326397549441185721294313630638906474130100850129613517930461593382384 38 =$
$(1010001100001011111101100111001010110110010011111000110100000100111100000011000011101011$

— $e = 17$

— $A^e \bmod N = 2$ [oui, ce n'est pas une erreur]

4 Multiplication Égyptienne

La multiplication égyptienne, aussi appelée multiplication par duplication, est une méthode ancienne très bien adaptée aux nombres binaires. Elle repose sur l'idée que toute multiplication peut être réalisée par une série d'additions et de décalages (doublages), ce qui correspond exactement aux opérations naturelles en binaire.

4.1 Principe général (en base 10 pour commencer)

4.2 Application en binaire

L'algorithme est encore plus simple en binaire car :

1. Doubler un nombre revient à un décalage à gauche.
2. Tester si un bit est à 1 se fait facilement.

3. Additionner est naturel.

Avantages en binaire :

1. Pas de table de multiplication : uniquement des additions et des décalages/shifts/"division par 2".
2. Particulièrement efficace donc pour les grands entiers binaires.
3. Très utilisée dans les circuits logiques et le calcul modulaire (ex : cryptographie RSA).

4.3 Un exemple simple

Objectif : multiplier $A = 13 = (1101)_2$ et $B = 11 = (1011)_2$
Ce qui donne :

Étapes de la multiplication égyptienne (binaire)

Étape	Multiplier B (binaire)	Bit LSB	Multiplicand A (binaire)	Action	Résultat partiel
1	1011	1	1101	Ajouter 1101	1101
2	0101	1	11010	Ajouter 11010	100111
3	0010	0	110100	Ignorer	100111
4	0001	1	1101000	Ajouter 1101000	100111 + 1101000 = 10001111

Figure 1 – Multiplication Égyptienne pour 13×11

Explications :

Étapes de la multiplication égyptienne (binaire)

Étape	Multiplier B (binaire)	Bit LSB	Multiplicand A (binaire)	Action	Résultat partiel
1	1011	1	1101	Ajouter 1101	1101
2	0101	1	11010	Ajouter 11010	100111
3	0010	0	110100	Ignorer	100111
4	0001	1	1101000	Ajouter 1101000	100111 + 1101000 = 10001111

Figure 2 – Détails des calculs de 13×11

5 BONUS : Phase 3 : (enfin) RSA un vrai algorithme de chiffrement . . .

***Attention/Pay attention** [2]:*

Si tout s'est bien passé, vous disposez maintenant de toutes les fonctions pour programmer une fonction de chiffrement/déchiffrement RSA, avec un exposant E ou D de type `unsigned int` *c'est-à-dire une fonction qui implémente l'algorithme RSA.*
Mais, **pour avoir un "vrai" RSA il nous faut aller . . . un peu plus loin**. Pas d'inquiétude, on explique pas à pas ce qu'il vous reste à faire. Mais d'abord, décrivons ce célèbre algorithme RSA.

***Attention/Pay attention** [3]:*

Cette section est un "Bonus". Pour la commencer et espérer la finir, vous devez absolument avoir terminé TOUT ce qui est demandé dans les phases UN et DEUX.

5.1 RSA ?

La sécurité informatique est un domaine clé en ingénierie, et l'algorithme RSA (Rivest-Shamir-Adleman) est l'un des piliers de ce domaine.

- **RSA** est un système de *cryptographie à clé publique*, appelée aussi plus proprement *cryptographie asymétrique* qui est largement utilisé pour sécuriser les communications *sensibles*.
- C'est un des algorithmes de chiffrement (et de signature) les plus célèbres.
- Malgré son âge (publié en 1978) il est encore très utilisé, surtout pour HTTPS (TLS etc.) mais il se trouve aussi par exemple dans toute carte à puce bancaire en France ou encore dans PGP.

5.2 Cryptographie à clé publique (cryptographie asymétrique) ?

Dans ce système, deux clés sont utilisées qui sont habituellement différentes :

1. une clé publique pour le chiffrement
2. et une clé privée pour le déchiffrement

L'avantage de ce système est que la clé publique peut être distribuée librement, tandis que la clé privée reste secrète.

5.3 Arithmétique de RSA

L'algorithme **RSA** est basé sur l'arithmétique des (grands) nombres entiers et utilise un produit de deux grands nombres premiers pour générer les clés. L'aspect le plus important à comprendre ici est que, bien que la clé publique (le produit des deux nombres premiers) soit connue, il est

pratiquement impossible[2] de retrouver les deux nombres premiers à partir de leur produit, ce qui rend le déchiffrement sans la clé privée extrêmement difficile. L'algorithme **RSA** utilise deux clés différentes : une *clé publique pour chiffrer les messages* et une *clé privée pour les déchiffrer*.
Le processus de génération de clés **RSA** comprend trois étapes principales :

— **Génération des clés**[3] :

1. L'utilisateur (Alice[4] dans la suite) choisit deux nombres premiers, p et q, "assez grands" et calcule leur produit $N = p * q$.
2. Alice **garde soigneusement secret** p et q !
3. Ce nombre N est utilisé comme module de chiffrement et souvent appelé *Modulus RSA*.
4. Ensuite, Alice calcule la fonction d'Euler de N , notée $\varphi(N)$, qui représente le nombre d'entiers positifs inférieurs à N et premiers avec N et qui vaut en fait $\varphi(N) = (p-1)(q-1)$.
5. Ensuite, Alice choisit un exposant de chiffrement E , qui est un entier premier à $\varphi(N)$, *i.e.* : $PGCD(E, \varphi(N)) = 1$ et qui vérifie $0 < E < \varphi(N)$.
6. La clé dite *publique* d'Alice est constituée de N et E , on note souvent (N, E) cette clé publique
7. Tandis que la clé dite *privée* d'Alice est composée de N et d'un exposant de déchiffrement D , on note souvent (N, D) cette cél privée.
8. Évidemment, Alice doit à tout prix garder secret D en plus de garder secret p et q !

— **Chiffrement** : Pour chiffrer un message, Bernard le convertit en un nombre[5] entier positif M, Ensuite, Bernard utilise la clé publique (N, E) pour élever le nombre M à la puissance E (évidemment) modulo N pour obtenir le chiffré C, ce qui s'écrit :

$$C = M^E \bmod N.$$

— **Déchiffrement** : Pour déchiffrer le message chiffré C, Alice utilise sa clé privée (N, D). Alice élève C le "nombre/message" chiffré à la puissance D modulo N , ce qui nous donne[6] le "nombre/message" original M :

$$C^D \bmod N = (M^E)^D \bmod N = M^{E*D} \bmod N M^1 \bmod N.$$

5.4 En résumé :

L'algorithme **RSA** utilise deux exposants E (public : connu de tout le monde) et D (privé : connu d'Alice seulement) et un module N (public : connu de tout le monde), choisis de telle sorte que $E * D$ soit congru à 1 modulo $\varphi(N) = (p-1)(q-1)$, avec p et q "assez grands" et le tout doit vérifier :

$$E * D = 1 \bmod \varphi(N).$$

Ce qui peut aussi s'écrire : il existe un entier k tel que :

2. *Computationaly Difficult*
3. On ne vous demande pas de "générer" les clés vous-même !
4. Le floklore de la cryptographie moderne parle de Alice et de Bernard : Bernard aime envoyer des messages chiffrés à Alice, qui aime les déchiffer !
5. Cela non plus ne vous est pas demandé !
6. Magiquement ? Non, c'est de l'arithmétique modulaire !

$$E * D - k * \varphi(N) = 1$$

Cette équation est appelée **équation RSA** mais en fait ce n'est rien d'autre que **l'identité de Bézout**, appliquée aux entiers E et $\varphi(N)$) qui se résoud par *l'algorithme dEuclide étendu, qui généralise l'algorithme d'Euclide* et qui est bien connue des élèves en France [7].
Ainsi :

1. La **clé publique** est le couple (N, E)
2. La **clé privée** est le couple (N, D).

La sécurité de l'algorithme dépend de la difficulté de factoriser N en p et q avec les moyens de calcul disponibles. En effet, si l'algorithme **RSA** est (encore) considéré comme sûr c'est qu'il repose sur la difficulté de factoriser de grands nombres non premiers. La sécurité du système dépend donc de la taille des nombres premiers utilisés pour générer les clés. Comment calculer une "bonne" clé RSA ? En fait c'est devenu un *art* que de savoir calculer une "bonne" clé RSA, mais ceci est une autre histoire ...
L'algorithme **RSA** est souvent utilisé pour transmettre une clé de chiffrement symétrique [8], qui permet ensuite d'échanger des données de façon plus rapide et efficace [9].

5.5 Un exemple "jouet" :

1. Soient $p = 1009$ et $q = 1201$ deux nombres premiers
2. Soit $N = p * q = 1211809$ et donc $\varphi(N) = (p-1)(q-1) = 1209600$
3. Soit E un entier > 0 tel que $PGCD(E, \varphi(N)) = 1$, par exemple $E = 101$
4. On calcule D vérifiant [10]
$$E * D = 1 \bmod \varphi(N)$$
5. Ce qui donne $D = 251501$

Le *chiffrement du message* $M = 99999$ se fait alors de la manière suivante :

$$C = M^E \bmod N = 99999^{101} \bmod 1211809 = 561752 \tag{2}$$

Tandis que le *déchiffrement* du message chiffré [11] C se fait de la manière suivante :

$$M = C^D \bmod N = 561752^{251501} \bmod 1211809 = 99999. \tag{3}$$

5.6 Un "vrai" RSA

Pour finir ce projet, si on a E ou D stocké avec le type `unsigned int` c'est déjà fait si vous avez codé la fonction d'exponentiation rapide modulaire avec ces arguments. Mais, dans le cas où E ou D sont eux même des GrandEntiers(s) Binaire il nous faut alors une fonction de chiffrement/déchiffrement qui soit capable de faire les calculs suivants ;

— Soit N un modulus RSA, de type "GrandEntier Binaire ("*BigBinary*")"

7. Enfin, on espère pour vous
8. C'est de cette manière que RSA est utilisé dans la célèbre application PGP : *Pretty Good Privacy*.
9. *On peut aussi signaler qu'il peut aussi servir à signer des messages en utilisant la clé privée pour chiffrer et la clé publique pour vérifier la signature*
10. On calcule D en utilisant la version de l'algorithme d'Euclide appelé *algorithme d'Euclide étendu*. Qu'on ne vous demande pas d'implémenter.
11. Le *déchiffrement* étant en fait la fonction "inverse" du le textcolormagenta]*chiffrement*, vous suivez ?

— Soit E un exposant de chiffement public de type "GrandEntier Binaire ("*BigBinary*")", donc E est aussi stocké dans un tableau de Digits en base `Base`

— Soit M un message de type "GrandEntier Binaire ("*BigBinary*")"

— Il faut calculer $M^E \bmod N$

Pour ce faire, il y a de nombreux algorithmes possibles, en voici 3, en supposant qu'on code l'un des algorithmes d'exponentiation rapide que nous avons vus et qui reposent sur la parité de E (et de ses "successeurs") :

1. On implémente la Division Euclidienne d'un GrandEntier Binaire ("*BigBinary*") A par un GrandEntier Binaire ("*BigBinary*") B avec $A \geq B$: assez difficile en fait, on vous le déconseille.
2. On implémente la Division Euclidienne d'un GrandEntier Binaire ("*BigBinary*") A par l'entier 2 avec $A \geq 2$: beaucoup plus facile même si cela reste technique
3. On implémente une fonction qui calcule $M^E \bmod N$ et qui utilise astucieusement la `struc` que nous utilisons. Nous appellerons cet algorithme : **Algorithme d'Exponentiation Rapide Modulaire "Maison"**

5.7 Algorithme d'Exponentiation Rapide Modulaire "Maison"

Donnons un exemple : soient N un module RSA (GrandEntier Binaire ("*BigBinary*")), M un message (GrandEntier Binaire ("*BigBinary*")) et $E = 5234$ (GrandEntier Binaire ("*BigBinary*")).
Techniquement, E est stocké dans un tableau, avec $Base = 10$ cela donne :

$$"E" = \{4, 3, 2, 5, 0 \cdots$$

Si on veut calculer MÍ$E \bmod N$, voici un algorithme plus simple qui évite la Division Euclidienne :

1. On veut donc calculer $M^E \bmod N$, pour simplifier la présentation nous allons supprimet le $\bmod N$ dans la suite
2. Donc comme $E = 5 * 10^3 + 2 * 10^2 + 3 * 10^1 + 4 * 10^0$
3. On peut écrire que

$$M^E = M^{5*10^3+2*10^2+3*10^1+4} = M^{5*10^3} * M^{2*10^2} * M^{3*10^1} * M^4$$

La dernière égalité peut se **réécrire** de manière plus pratique :

$$M^E = M^{5*10^3} * M^{2*10^2} * M^{3*10^1} * M^4 = M^{10^{5*3}} * M^{10^{2*2}} * M^{10^{3*1}} * M^4 \tag{4}$$

Soit encore

$$M^E = (M^{10})^{15} * (M^{10})^4 * (M^{10})^3 * M^4 \tag{5}$$

Ou mieux, si on respecte la parcours des Digits de E :

$$M^E = M^4 * (M^{10})^3 * (M^{10})^4 * (M^{10})^{15} \tag{6}$$

Et avec le modulo on a donc :

$$M^E \bmod N = [M^4 \bmod N] * [(M^{10})^3 \bmod N] * [(M^{10})^4 \bmod N] * [(M^{10})^{15} \bmod N] \tag{7}$$

Et si on veut être précis et exact, on doit évidemment prendre le modulo du "tout" :

$$M^E \bmod N = \{[M^4 \bmod N] * [(M^{10})^3 \bmod N] * [(M^{10})^4 \bmod N] * [(M^{10})^{15} \bmod N]\} \bmod N. \tag{8}$$

Et on "voit" l'**Algorithme d'Exponentiation Rapide Modulaire "Maison"**. Pourquoi réécrit-on le calcul de M^E, parce que cela est plus efficace algorithmiquement et qu'on évite ainsi (avec le choix de `Base` que nous avons fait) les fameux **Integer Overflow** , les fameux *"Dépassement de capacité entière"* ou *"Débordement d'entier"*.
On généralise aisément en base 10 avec $E = \sum_{i=0}^{i=k} e_i 10^i$:

$$M^E \bmod N = M^{\sum_{i=0}^{i=k} e_i 10^i} = \prod_{i=0}^{i=k} (M^{10^i})^{e_i} = \prod_{i=0}^{i=k} (M^{10})^{e_i * k} \bmod N. \tag{9}$$

Et on généralise tout aussi aisément en base B avec $E = \sum_{i=0}^{i=k} e_i B^i$:

$$M^E \bmod N = M^{\sum_{i=0}^{i=k} e_i B^i} = \prod_{i=0}^{i=k} (M^{B^i})^{e_i} = \prod_{i=0}^{i=k} (M^{B})^{e_i * k} \bmod N. \tag{10}$$

Où encore une fois on ne doit calculer **qu'une seule fois** M^B.

6 Que devez vous faire pour finir ce projet ?

1. Avoir programmé une fonction `Addition`
2. Avoir programmé une fonction `Soustraction`
3. Avoir programmé une fonction `Multiplication Égyptienne`
4. Avoir programmé toutes les fonctions `Utiles` de comparaison etc.
5. Avoir programmé une fonction `Modulo` qui utilise les fonctions de la phase 1.
6. Avoir programmé une fonction `Exponentiation rapide modulaire avec` E `un "unsigned int"`
7. Avoir programmé une fonction `Chiffrement RSA` un "unsigned int" : facile !
8. Avoir programmé une fonction `Dechiffrement RSA` un "unsigned int" : facile !
9. Avoir programmé une fonction `Exponentiation rapide modulaire avec` E ou D un "GrandEntier Binaire ("*BigBinary*")" ! Avec l'algorithme de votre choix (i.e. l'un des trois présentés ici.

7 Quelques références

1. Algorihmes d'Euclide `https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide`
2. Algorihmes Binaire d'Euclide `https://fr.wikipedia.org/wiki/Algorithme_binaire_de_calcul_du_PGCD`
3. Wikipedia (diverss algorithmes de Division et de calcul d'inverse) : `https://en.wikipedia.org/wiki/Division_algorithm`