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

Équipe pédagogique de l'ESIEA

L'Algorithme Binaire de Calcul du PGCD etc.: Oh les "BigDigits" !

Table des matières

PROPOSITION PREMIÈRE
Deux nombres inégaux étant proposés, le plus petit étant toujours retranché du plus grand, si le reste ne mesure celui qui est avant lui que lorsque l'on a pris l'unité, les nombres proposés seront premiers entr'eux.
— Euclide (vers 325-225 av. J.-C.)
We might call Euclid's method the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day. (The chief rival for this honour is perhaps the ancient Egyptian method of multiplication, which was based on doubling and adding, and which forms the basis for the efficient calculation of nth powers...)
— D. E. Knuth.

1 Avant-propos

Avant-propos : Ce qu'on appelle un "*grand entier*" est un outil fondamental dans la cybersécurité, notamment dans la cryptographie dite "à clé publique" ou encore "asymétrique". Malheureusement, le langage $\mathbb{C}$, de même que beaucoup d'autres langages, ne permet pas de manipuler efficacement des "grands entiers". Le projet a donc pour objectif de vous faire réaliser une "bibliothèque" de fonctions permettant la manipulation d'entiers arbitrairement grands, **en binaire**. Une définition simple de ce qu'est un "grand entier" : c'est un entier qui ne peut être exactement représenté sur 32 bits ou sur 64 bits, ni même sur 128 bits. Ainsi : $2^{101} = 2535301200456458802993406410752$ ne peut être représenté sur 64 bits vu qu'on aurait besoin de 102 bits.

2 Le projet ? Trois "Phases"

Quelques remarques :

1. Le projet est structuré en 3 phases (pour mieux vous aider)
2. Planification conseillée : voir plus haut dans le texte.

3 Euclide et ses algorithmes

L'algorithme d'Euclide est un des rares algorithmes qui se trouve enseigné à la fois dans des cours de mathématiques et dans des cours d'informatique. Cet algorithme calcule de manière vraiment efficace le **plus grand commun diviseur (PGCD) de deux entiers** $a \geq 0$ **et** $b \geq 0$. Le $\mathrm{PGCD}(a, b)$ est le plus grand entier qui divise les deux entiers, c'est-à-dire qu'ils sont tous les deux multiples de celui-ci. Comme le dit Wikipedia : *C'est un des plus anciens algorithmes connus, mais il reste toujours d'actualité.*

3.1 Pourquoi le PGCD ?

Pourquoi l'algorithme d'Euclide est important ? On peut lister plusieurs raisons (liste non exhaustive) :

1. C'est un des plus "vieux" algorithmes existants (cf la citation de D. KNUTH), il est de plus "prouvé" dans l'ouvrage d'origine, les fameux "*Éléments*" d'Euclide (publié vers 300 avant J.C).
2. Il existe de très nombreuses variantes, car il s'adapte particulièrement bien aux différents processeurs et à leurs limites.
3. Il se programme "facilement" sur des nombres machines (32 bits ou 64 bits) que ce soit en itératif, en récursif non terminal ou en récursif terminal (il est d'ailleurs "naturellement" récursif terminal.
4. Enfin, c'est le seul algorithme capable de calculer une "clé RSA", ce qui en fait de cet algorithme un des piliers de la cybersécurité des données des réseaux et des applications.
5. C'est un des algorithmes "cachés" parmi les plus exécutés au monde et ce chaque jour qui passe. Si vous êtes sur un site web en `https`, le "s" pour "secure", il y a de fortes chances que une ou plusieurs des clés de chiffrement, de signature numérique soient calculées/générées par l'algorithme d 'Euclide ou une de ses variantes.
6. Il est au coeur de nombre d'algorithmes dit "post-quantique", d'une manière directe ou indirecte, tant sur les entiers que sur les polynômes (et oui l'algorithme d'Euclide permet aussi de calculer le "PGCD de deux polynômes". L'application la plus importante de cet variante polynomiale c'est : la détection et la correction d'erreurs, qui protègent vos disques durs, vos clés USB etc. (On le trouvait aussi dans la technologie des CD et DVD !).
7. Enfin, ce qui vous concerne peut-être plus en tant que future stagiaire et futur.e ingénieur.e : de nombreux entretiens d'embauche portent sur cet algorithme.

Mais, car il y un mais, l'opération de base de l'algorithme d'Euclide c'est la *Division Euclidienne* **et cette opération est coûteuse lorsque les entiers sont très grands, de l'ordre de plusieurs centaines à plusieurs milliers de digits.**
Après un bref rappel de quelques variantes de l'Algorithme d'Euclide, nous présenterons le sujet de ce projet.

3.2 Algorithme d'Euclide par soustraction

Dans le texte d'origine d'Euclide (voir la citation), l'algorithme repose sur la relation fondamentale suivante : (notation moderne)

$$\text{Si a } > \text{b } : \text{PGCD}(a,b) = \text{PGCD}(a-b,b) == \text{PGCD}(b,a-b). \tag{1}$$

Ce que Wikipedia résume par *l'algorithme part du constat suivant : le* PGCD *de deux nombres n'est pas changé si on remplace le plus grand d'entre eux par leur différence. Autrement dit,* $\text{PGCD}(a,b) = \text{PGCD}(a-b,b)$. Et la propriété suivante est intéressante :

$$\text{Si a } < \text{b } : \text{PGCD}(a,b) = \text{PGCD}(b,a). \tag{2}$$

On ajoute la propriété suivante

$$\text{PGCD}(a,0) = a \tag{3}$$

Ce qui donne finalement comme algorithme :

Algorithme 1 : « Euclide par Soustraction (antiphérèse) »
Données : a, b ;
Sortie : $\text{PGCD}(a,\ b)$;
Début :
$u = \max(a,\ b)$; $v = \min(a,\ b)$;
Tant Que *Non Convergence* **Faire**

Si $u > v$ **alors** $u = u - v$;
sinon $v = v - u$;
Fin du Si ;
Fin du Tant Que ;
Retourner (u) ;
Fin.

Donnons de suite un exemple de cet algorithme par soustraction :

$$\text{PGCD}(741, 715) = \text{PGCD}(741 - 715, 715) = \text{PGCD}(715, 26) = \text{PGCD}(715 - 26, 26) \quad (4)$$

$$\text{PGCD}(715 - 26, 26) = \text{PGCD}(715 - 2 * 26, 26) \cdots = \text{PGCD}(13, 26) = \text{PGCD}(26, 13) \quad (5)$$

Et donc finalement :

$$\text{PGCD}(741, 715) = \text{PGCD}(26, 13) = \text{PGCD}(26 - 13, 13) = \text{PGCD}(13, 13) = 13 \quad (6)$$

Cette technique de calcul par "soustraction" se nomme **antiphérèse**[1]. Nous verrons pourquoi cette technique, lente dans le cas où a est très grand devant b, peut dans certains cas s'avérer très intéressante. Tout ceci donne l'algorithme présenté à la figure (1) qui vient de Wikipedia et équivalente à la version de l'algorithme (1).

Algorithme d'Euclide original

Entrée = Deux entiers a et b
Sortie = Le PGCD de a et b

```
fonction euclide(a, b)
  tant que a ≠ b
    si a > b alors
      a := a − b;
    sinon
      b := b − a;
  renvoyer a;
```

Figure 1 – Wikipedia : Algorithme d'Euclide "Original", par simple soustraction

3.3 Algorithme d'Euclide avec Division Euclidienne

Mais que se passe-t-il pour l'algorithme d'Euclide par soustraction si par exemple $n = 10^{200} + 5$ et $m = 3$? L'algorithme devient alors **(vraiment) très lent**. C'est certainement ce qui a entraîné Euclide à proposer, de manière implicite dans sa "démonstration", l'algorithme 2. En effet, qans la version moderne de l'algorithme d'Euclide on utilise la **Division Euclidienne** pour "éliminer" les soustractions nombreuses, l'opération de base devient ainsi :

$$\text{PGCD}(a, b) = \text{PGCD}(b, a \bmod b) \quad (7)$$

1. Wikipedia :En histoire des mathématiques, on appelle **antiphérèse** (ou "anthyphérèse") la méthode qu'Euclide utilise pour calculer le PGCD de deux nombres ou démontrer que deux longueurs sont incommensurables. Anthyphèrese vient du grec "*anthyphairesis*" qui signifie soustraire alternativement.

Cette égalité provient de l'égalité suivante :

$$a = q * b + r \tag{8}$$

Avec $a > b$: r est le *reste* de la Division Euclidienne de a par b et q est le *quotient* de la Division Euclidienne de a par b.
Pourquoi cela fonctionne t-il ? Et bien si $a = q * b + r$ et si $d = \text{PGCD}(a, b)$ alors d doit aussi diviser r et $\text{PGCD}(a, b) = \text{PGCD}(a, r)$, qui est équivalente à l'égalité (7).
Cet algorithme repose sur l'égalité $\text{PGCD}(u,\, v) = \text{PGCD}(v,\, u \bmod v)$, vraie dès lors que $u \geq v$. C'est cette version que l'on nomme habituellement l'*algorithme d'Euclide*. Comme le souligne Knuth , Euclide — après avoir proposé l'algorithme 1 (page 4) — propose en fait sur des exemples l'algorithme 2 qui utilise la Division Euclidienne dans la suite de son ouvrage.

Algorithme 2 : « Euclide avec Division Euclidienne »
Données : a, b ;
Sortie : $\text{PGCD}(a,\, b)$;
Début :
$u = \max(a,\, b)$; $v = \min(a,\, b)$; $r = 1$;
Tant Que $r > 0$ **Faire**
$r = u \bmod v$;
$u = v$;
$v = r$;
Fin du Tant Que ;
Retourner (u) ;
Fin.

Reprenons l'exemple précédent :

$$\text{PGCD}(741, 715) = \text{PGCD}(715, 26) = \text{PGCD}(715, 26) = \text{PGCD}(26, 13) = \text{PGCD}(13, 13) \tag{9}$$

On peut remarquer que la dernière égalité est obtenue par une *antiphérèse* car $26 = 13 + 13$. Ainsi, l'algorithme d'Euclide avec Division Euclidienne présenté à la figure (2), équivalente à l'algorithme (2) peut lui aussi comporter des "étapes" d'antiphérèse lorsque le quotient est égal à 1.

Algorithme d'Euclide itératif

Entrée = Deux entiers a et b
Sortie = Le PGCD de a et b

```
fonction euclide(a, b)
  tant que b ≠ 0
    t := b;
    b := a modulo b;
    a := t;
  renvoyer a;
```

Figure 2 – **Wikipedia : Algorithme d'Euclide "avec Division Euclidienne"**

3.4 L'algorithme Binaire d'Euclide

3.5 Les principes

L'algorithme standard nécessite à chaque itération une Division Euclidienne, ce qui peut s'avérer coûteux voire difficile pour des "grands entiers" sur certains processeurs de faible puissance.
Mais comme on peut exécuter l'algorithme d'Euclide dans n'importe quelle base, il existe une base plus intéressante que d'autres : c'est la **base binaire**. Pourquoi ? Parce que la Division Euclidienne "disparait". En binaire on peut exécuter l'algorithme sans Division Euclidienne, juste avec des additions et soustractions, ce qui le rend particulièrement intéressant pour des processeurs de faible puissance comme celui d'une carte à puce. On appelle cette variante **l'algorithme Binaire d'Euclide**.
Donnons les propriétés qui permettent d'exécuter l'algorithme Binaire d'Euclide. Supposons que a, b soient deux entiers > 0 et tels que $a > b$. On a alors les propriétés suivantes :

$$\text{pour tout a et b=0 :} \quad \text{PGCD}(a, 0) = a \tag{10}$$
$$\text{Si a est pair et b pair :} \quad \text{PGCD}(a, b) = 2 \times \text{PGCD}(a/2, b/2) \tag{11}$$
$$\text{Si a est pair et b est impair :} \quad \text{PGCD}(a, b) = \text{PGCD}(a/2, b) \tag{12}$$
$$\text{Si a est impair et b pair :} \quad \text{PGCD}(a, b) = \text{PGCD}(a, b/2) \tag{13}$$
$$\text{Si a est impair et b impair :} \quad \text{PGCD}(a, b) = \text{PGCD}(a - b, b) \tag{14}$$

Et dans ce dernier cas : si a est impair et b impair alors (a-b) devient pair !
Mais à quoi peut bien servir cet algorithme "binaire" ? En fait, si a et b sont tous deux *représentés en binaire* il est alors **facile de "diviser" a ou b par deux : on** *shifte/décale* et du coup, on "élimine" la Division Euclidienne !
On résume pourquoi l'algorithme Binaire d'Euclide est intéressant :

- Plus besoin de Division Euclidienne
- La "multiplication par 2" devient une addition car naïvement : $2 \times n = n + n$
- On utilise donc que des additions et soustraction

3.6 Un Exemple détaillé

Soient $a = 3 * 17 = 51$ et $b = 3 * 19 = 57$ deux entiers.
Les étapes sont présentées dans l'image (3)

4 Objectifs : Calcul sur des entiers en précision arbitraire (binaire)

4.1 Description du projet :

Ce projet de programmation vise à développer une (petite) bibliothèque en langage C permettant la manipulation en binaire de "grands entiers" [2] stockés dans des tableaux, tableaux dynamiques pour mieux gérer la mémoire. *In Fine* vous serez amené.e.s à programmer une version simple du célèbre algorithme de chiffrement à clé publique RSA (en Bonus) !

2. De quelques dizaines de bits à quelques centaines si nécessaire !

Étape	a (déc)	a (bin)	b (déc)	b (bin)	Action
0	51	110011	57	111001	$b > a$, impairs → PGCD((57 - 51)/2 = 3, 51)
1	3	000011	51	110011	$b > a$, impairs → PGCD((51 - 3)/2 = 24, 3)
2	24	011000	3	000011	`a` pair → PGCD(12, 3)
3	12	001100	3	000011	`a` pair → PGCD(6, 3)
4	6	000110	3	000011	`a` pair → PGCD(3, 3)
5	3	000011	3	000011	`a == b` → PGCD = 3 ✅

Figure 3 – Les différentes étapes de l'Algorithme Binaire d'Euclide

4.2 Problème : comment représenter et stocker un "Grand Entier"

— Le plus simple[3] est de stocker les bits dans un tableau !

— On peut aussi stocker le signe (ce que nous vous demanderons de faire)

— Et pour alléger les calculs on peut aussi avoir envie de stocker le nombre de bits nécessaires (ce que nous vous demanderons aussi de faire)...

Remarques :

1. Il est utile de voir, pour gagner du temps et simplifier le code : qu'il vaut mieux stocker les bits en plaçant les bits de poids les plus forts "à gauche" et donc les bits de poids les plus faible "à droite"
2. Exemple avec 24 qui en binaire donne $(11000)_2$: diviser par deux ce nombre revient à "supprimer" le bit 0 de poids le plus faible, ce qui réduit de un la "taille" du nombre binaire (en nombre de bits évidemment).
3. donc, dans un tableau on préférera avoir $\{1, 1, 0, 0, 0\}$
4. Cela sera plus clair quand vous coderez.

4.3 Fonctionnalités attendues :

Voici ce que nous attendons de votre part[4] :

— **Opérations de base** :

1. Une fonction `Addition` (somme) (*)
2. Une fonction `Soustraction` (différence) (*)
3. Des fonctions usuelles de comparaison[5] (*) :
   — `Egal` : *`Egal`(A,B) renvoit `true` si A == B sinon on renvoit `false`*
   — `Inferieur` : *`Inferieur`(A,B) renvoit `true` si A < B sinon on renvoit `false`*

3. Ne pas utiliser svp (sauf si on vous dit le contraire) de chaînes de caractères pour faire les calculs comme on le voit souvent sur certains ... sites "d'aide" qui donnent parfois de (très) mauvais conseils par moment !
4. Le nombre de caractères "*" indique un niveau de "difficulté" (relatif).
5. ... prenant en argument deux "grands entiers" [on les compare et on renvoit `true` ou `false` selon que la comparaison sera vraie ou fausse].

4. Une fonction `Affichage` **que nous vous donnons**

— **Opérations non Triviales** (pour aller plus loin) :

1. Exponentiation rapide (méthode d'exponentiation modulaire) (**)
2. Calcul du modulo (***) : nous vous proposerons un algorithme qui sans être optimal sera disons plus facile à programmer que l'un des algorithmes optimaux.
3. Chiffrement et Déchiffrement RSA (** mais en BONUS) [une fois que vous aurez toutes les fonctionnalités précédentes cette partie est en fait facile].

5 Tâches proposées

Voici ce qui serait idéal comme "planning" de votre projet.

5.1 Phase Une : Fonctionnalités de base

— Une fonction `Addition` (somme) : utilisez l'algorithme naïf qui est en fait le plus efficace [algorithme de "l'école primaire"]

— Une fonction `Soustraction` (difference) : idem utilisez l'algorithme naïf qui est en fait le plus efficace [algorithme de "l'école primaire"]

— des fonctions usuelles de comparaison :

  — `Egal`,

  — `Inferieur`

— Une fonction `Affichage` vous sera donnée.

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

Quelques réponses à des questions reçues :

1. Addition et Soustraction : Récursif ou itératif ? Il faut simplifier ici, la récursivité n'apporterait que des ennuis, les deux problèmes ayant chacun un algorithme dit "naïf" mais linéaire, on reste **en itératif**, beaucoup plus rapide.
2. Pour l'addition : on ne considère que le cas "$A + B$" avec A et B "BigBinary"(s) positifs ou nuls éventuellement.
3. Pour la soustraction : on ne considère que le cas "$A - B$" avec A et B "BigBinary"(s) positifs et $A \geq B$.

5.2 Phase Deux : Fonctionnalités Avancées

— Calcul du PGCD de deux "nombres binaires" par l'Algorithme Binaire d'Euclide.

— Calcul du modulo

— Calcul Exponentiation rapide modulaire

5.3 Phase Trois : RSA simplifié (Bonus)

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

1. Chiffrement RSA
2. Déchiffrement RSA

6 Phase Une : quelques éléments pour démarrer correctement

6.1 Structure de données

Nous proposons la structure **BigBinary** suivante :

```
#include <stdio.h>
#include <stdbool.h>  // Permet d'utiliser des variables de type booléen ('false" ou "true
#define BASE 2
```

Pourquoi stocker les bits Avec comme `struct` pour représenter un BigBinary :

```
#define BASE 2  // La base du nombre (2 pour binaire)

typedef struct {
    int *Tdigits;    // Tableau de bits : Tdigits[0] = bit de poids fort,
       Tdigits[Taille - 1] = bit de poids faible
    int Taille;      // Nombre de bits significatifs
    int Signe;       // +1 pour positif, -1 pour négatif, 0 pour nul
} BigBinary;
```

La même sans les commentaires :

```
#define BASE 2  // La base du nombre (2 pour binaire)

typedef struct {
    int *Tdigits;
    int Taille;
    int Signe;
} BigBinary;
```

Une fonction pour créer un "BigBinary" vide :

```
BigBinary createBigBinary(int size) {
    BigBinary bb;
    bb.Tdigits = malloc(sizeof(int) * size);
    bb.Taille = size;
    bb.Signe = 0; // Par défaut nul
    return bb;
}
```

6.2 Bit de poids fort ou faible

Il faut choisir , le plus simple pour la suite (division par 2) c'est de faire le choix suivant, qu'on vous impose :

1. Tdigits[0] = bit le plus à gauche (MSB)
2. Tdigits[size - 1] = bit le plus à droite (LSB)

6.3 Un exemple de fonction d'affichage

Pour "afficher" un **BigBinary** nous vous proposons la fonction suivante, avec un exemple d'initialisation "naif" suivi de l'affichage : :

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

#define BASE 2  // La base du nombre (2 pour binaire)
```

```
5
6  typedef struct {
7      int *Tdigits;
8      // Tableau de bits : Tdigits[0] = bit de poids fort, Tdigits[Taille -
           1] = bit de poids faible
9      int Taille;      // Nombre de bits significatifs de l'entier
10     int Signe;       // +1 pour positif, -1 pour négatif, 0 pour nul
11 } BigBinary;
12
13 // Initialisation manuelle d'un BigBinary vide (tout est a zero)
14 BigBinary initBigBinary(int taille, int signe) {
15     BigBinary nb;
16     nb.Taille = taille;
17     nb.Signe = signe;
18     nb.Tdigits = malloc(sizeof(int) * taille);
19     for (int i = 0; i < taille; ++i) {
20         nb.Tdigits[i] = 0;
21     }
22     return nb;
23 }
24
25 // Affichage du nombre binaire
26 void afficheBigBinary(BigBinary nb) {
27     if (nb.Signe == -1) printf("-");
28     if (nb.Signe == 0 || nb.Taille == 0) {
29         printf("0\n");
30         return;
31     }
32     for (int i = 0; i < nb.Taille; ++i) {
33         printf("%d", nb.Tdigits[i]);
34     }
35     printf("\n");
36 }
37
38 // Division par 2
39 void divisePar2(BigBinary *nb) {
40     // A vous ...
41 }
42
43 // Libération de la mémoire
44 void libereBigBinary(BigBinary *nb) {
45     free(nb->Tdigits);
46     nb->Tdigits = NULL;
47     nb->Taille = 0;
48     nb->Signe = 0;
49 }
50
51 int main() {
52     // Représentation manuelle de 83 en binaire : 1010011
53     // Tdigits[0] = bit de poids fort -> ici 1
54     int bits83[] = {1, 0, 1, 0, 0, 1, 1};
55     int taille = sizeof(bits83) / sizeof(bits83[0]);
56
57     BigBinary nb = initBigBinary(taille, +1);
58     for (int i = 0; i < taille; ++i) {
59         nb.Tdigits[i] = bits83[i];
60     }
61
```

```
62      printf("Valeur initiale : ");
63      afficheBigBinary(nb);
64
65      divisePar2(&nb);
66      printf("Après division par 2 : ");
67      afficheBigBinary(nb);
68
69      libereBigBinary(&nb);
70      return 0;
71  }
```

Explications :

1. La fonction `afficheBigBinary` vérifie le signe du nombre pour décider d'afficher un signe moins (-) devant les bits si le nombre est négatif.
2. Elle parcourt le tableau de bits pour imprimer chaque bit. Notez que cette fonction assume que les bits dans le tableau sont stockés du chiffre le plus significatif (bit de poids le plus for, MSB) au moins significatif (bit de poids le plus faible, LSB) (comme dans l'exemple de l'entier 83).
3. À la fin, la fonction termine par imprimer un saut de ligne pour nettoyer l'affichage.

6.4 Le nombre 0 ?

Pour initialiser une structure BigBinary à "zéro", il est plus simple de concevoir une fonction spécifique qui configure correctement tous les champs de la structure pour représenter le nombre zéro. Dans le cas d'un BigBinary, le nombre zéro est généralement représenté par un seul chiffre, qui est 0, avec un signe positif.

6.5 Initialiser un "BigBinary"

Si vous voulez faire des tests avec de vrais BigBinarys(s) cela peut être assez pénible d'initialiser à la main avec un exemple du type "1111....1" (imaginez 1000 chiffres décimaux) etc.
Voici donc une fonction[6] qui prend en entrée une **chaine de caractères** composées uniquement de bits et qui initialise un "**BigBinary**" à partir de la chaine de caractères passée en argument :

```
// Création depuis une chaîne binaire
BigBinary creerBigBinaryDepuisChaine(const char *chaine) {
    BigBinary nb;
    int n = strlen(chaine);
    nb.Taille = 0;

    // Comptons uniquement les caractères valides ('0' ou '1')
    for (int i = 0; i < n; ++i) {
        if (chaine[i] == '0' || chaine[i] == '1') {
            nb.Taille++;
        }
    }

    nb.Tdigits = malloc(sizeof(int) * nb.Taille);
    nb.Signe = +1;
    int index = 0;
    int tousZeros = 1;
```

6. **Ce code se trouve sur Moodle**

```
    for (int i = 0; i < n; ++i) {
        if (chaine[i] == '0' || chaine[i] == '1') {
            nb.Tdigits[index] = chaine[i] - '0';
            if (nb.Tdigits[index] == 1) tousZeros = 0;
            index++;
        }
    }

    if (tousZeros) nb.Signe = 0;

    return nb;
}
```

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

Vous pouvez utiliser cette fonction telle quelle mais attention : elle ne gère pas de bases autres que $Base = 2$.

6.6 Le nombre de digits

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

Il est dangereux d'initialiser "à la main" le nombre de bits d'un "BigBinary", surtout dans une fonction qui calcule ce "BigBinary". Il est donc très utile et presque nécessaire d'avoir une fonction qui effectue ce travail !

6.7 "scanf" et "printf" ? ? ? ?

Comme il y a toujours des confusions au sujet des règles quant à l'utilisation des "scanf" et des "printf", voici la règle :

***Règle** [1]*

1. **Pendant la phase de développement de votre code : vous pouvez faire absolument TOUT CE QUE VOUS VOULEZ**
2. **Mais, pour le code "rendu" : votre code devra faire EXACTEMENT CE QUI SERA DEMANDÉ ! Ni plus ni moins. Et quand nous disons "EXACTEMENT" cela veut dire au caractère près.**

***Règle** [2]*

Des fichiers de test vous seront donnés. Voici un exemple : le caractère # désigne des commentaires

```
# Fichier de test pour BigBinary - Version "longue" (200+ bits)
# === PHASE 1 ===
# A1 + B1
A1:  10101010101001110011101000111110100001011101001000101100011010110000111000000110100011101001
B1:  11001011011111111000110000111010000000110101011000000100000010110101101001100011110001011000
# A == B2, A2 > C2
A2:  11001110111000001101101111110100110000001010101010001000000110010010001110000100001110010011
B2:  10110100101100011101100011111010010111011110000010011100011000000100100010011011010010011000
C2:  11100001000100000000100110010110010010100100111011011100001100000110011101101000101010010000
# === PHASE 2 ===
# PGCD
PGCD_A: 11110001111000100111100100001010110100110010110000100111001100110010111001010111101101110
PGCD_B: 11100110011111101100000011111111011000111000001100111110011101101101010111011011100011011
# Modulo
MOD_A:  11001110000111010001010110000101101001111000010001001110011100110100000110000010011011010
MOD_B:  11001001100000010001111000000101111110010111111011101010001000010100010001111111010011010
# Exponentiation modulaire
EXP_M: 10110011000000110001101100000011001001110110010010100101011101001000101111011000100110110C
EXP_EXP:  1001001000010111     # exposant raisonnable
EXP_MOD:  10100111000111110110000011100011000111010000011111101000101110010110110010110110101100C
```

7 Planification (suggérée)

7.1 Phase 1 : Fonctionnalités de base

Objectifs : Poser les bases du projet : représentation binaire d'un grand entier, opérations élémentaires.

- Implémenter la structure BigBinary (avec Tdigits, Taille, Signe).
- Implémenter la fonction d'initialisation (vide et depuis une chaîne binaire).
- Implémenter la fonction d'affichage (afficheBigBinary).
- Ajouter la fonction libereBigBinary pour éviter les fuites mémoire.
- Ajouter la fonction d'addition (algorithme "naïf").
- Ajouter la fonction de soustraction (avec $A \geq B$ uniquement, algorithme "naïf").
- Implémenter les fonctions de comparaison : Egal(A, B) et Inferieur(A, B)

7.2 Phase 2 : Fonctionnalités avancées

Objectifs : Implémenter des algorithmes de calcul efficaces pour les grands entiers binaires.

- Implémenter l'algorithme binaire d'Euclide pour le PGCD : BigBinary_PGCD(BigBinary A, BigBinary B) **et tester**
- Implémenter le modulo : BigBinary_mod(BigBinary A, BigBinary B) **et tester**

— Implémenter l'exponentiation modulaire rapide : **et tester**

— BigBinary_expMod(BigBinary M, BigBinary exp, BigBinary mod) **et tester**. On se limitera à des valeurs de l'exposant e de moins de 64 bits ; ce qui permettra de le stocker sur mon mot machine (en clair : le type `int` suffira).

7.3 Phase 3 : RSA simplifié (Bonus)

Objectifs : Objectifs pédagogiques :

1. Soit terminer les phases 1 et 2
2. Soit, si vous avez vraiment terminé : Comprendre les grandes étapes du chiffrement RSA. et appliquer les outils développés pour implémenter un chiffrement RSA simplifié.

— Implémenter le chiffrement RSA`BigBinary_RSA_encrypt(message, e, n)` : $C = M^e \bmod N$. On se limitera à des valeurs de l'exposant e de moins de 64 bits ; ce qui permettra de le stocker sur mon mot machine (en clair : le type `int` suffira).

— Implémenter le déchiffrement BigBinary_RSA_decrypt(cipher, d, n) : $M = C^d \bmod N$, là plus difficile si d est lui même grand ...

7.4 Que rendre et quand ... ?

Très bientôt ...

8 Quelques références

1. Algorithmes d'Euclide `https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide`
2. Algorithmes 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`