Université Jean Monnet SAINT-ETIENNE

2014-2015

Manipulation des entiers multiprécision - Application à la cryptographie asymétrique

telecom Saint-Etienne

CiTiSE1 - Projet info2

1 Introduction

L'objectif de ce travail est d'apprendre la manipulation de l'espace mémoire accessible à un programme à l'aide des pointeurs. Un pointeur est une variable qui contient l'adresse d'une case mémoire. La fin du cours d'Info1 donne quelques éléments pour comprendre et manipuler les pointeurs mais ne remplace en aucun cas la pratique. C'est l'objectif principal de ce projet que de s'approprier la notion de pointeurs par la pratique.

Le processeur d'un système informatique travaille avec des mots machines de taille fixée (8 bits, 16 bits, 32 bits, 64 bits). Or dans certaines applications, en cryptographie par exemple, les entiers manipulés peuvent atteindre plusieurs milliers de bits et sont donc d'une taille bien supérieure à la taille d'un mot machine sur lequel le processeur peut travailler "directement". Dans ce cas on parle d'opérations en *simple précision* alors que dans le cas de grands entiers, on parlera d'opérations *multiprécisions*.

On peut commencer par faire ici une analogie avec la façon dont vous avez appris à calculer dans les classes primaires. Dans un premier temps vous avez appris des tables d'addition et de multiplication sur des chiffres entre 0 et 9 (vos mots machines en quelques sortes et l'ensemble de ces opérations de base font partie de votre "jeu d'instructions").
Puis vous avez ensuite appris à utiliser vos instructions de base pour poser et effectuer des additions, soustractions, multiplications et divisions sur des nombres de plusieurs chiffres.

Un autre objectif de ce projet et d'apprendre à (programmer) l'ordinateur à faire ces calculs (additions, soustractions, multiplications, divisions) sur des entiers très grands à partir des opérations de base.
Comme la taille des entiers qui seront manipulés ne sera pas connue au départ, il faudra réserver une certaine quantité de l'espace mémoire pour stocker ces entiers et effectuer les calculs. Cette réservation mémoire (on parlera d'allocation mémoire) se fera à l'aide de pointeurs et de fonctions d'allocation/libréation dynamique de la mémoire.

Une fois les opérations sur les grands entiers réalisés, on pourra les utiliser pour effectuer une opération clé en cryptographie asymétrique : l'exponentiation modulaire.

2 Représentation des entiers longs

La cryptographie asymétrique requiert des entiers d'une grande taille (plusieurs milliers de bits).
En langage C, un entier `unsigned int` est usuellement représenté sur 32 bits et n'est donc pas suffisant pour représenter les entiers utilisés en cryptographie asymétrique.
Un *entier long* ou *entier multiprécision* sera donc représenté par un ensemble de chiffres en base $r = 2^{32}$, c'est à dire que chaque chiffre sera codé sur 32 bits.
Par exemple, un entier de 1024 bits sera représenté par 32 chiffres de 32 bits, c'est à dire un tableau de 32 `unsigned int`.
Comme on ne connait pas à l'avance la taille des entiers qu'on va manipuler, on travaillera avec des pointeurs sur des `unsigned int` et on utilisera les fonctions d'allocation et de libération dynamique de la mémoire du langage C++ : `new` et `delete`.

Ainsi une variable `a` représentant un entier de taille 32 chiffres de 32 bits sera déclaré comme :
`unsigned int * a= new unsigned int[32];`
`a[0]` (ou `*(a+0)`) représentera le chiffre de poids le plus faible et de manière générale, `a[i]` (ou `*(a+i)`) représentera le chiffre de poids r^i où $r = 2^{32}$.
Il est possible, si on ne veut pas se limiter aux entiers de 1024 bits, d'utiliser une variable m représentant le nombre de chiffres.

On a donc, pour un entier long de m chiffres de 32 bits chacun :

$$a = \sum_{i=0}^{m-1} a[i](2^{32})^i \text{ qui a donc } 32m \text{ bits.}$$

La déclaration de cet entier de m chiffres (où m peut être fixé dynamiquement au cours du programme) se déclarera comme :
`unsigned int *a=new unsigned int[m];`

Remarques complémentaires :

1. En utilisant l'allocation `new`, les cases mémoires ne sont pas forcément initialisées. Pour les initialiser à 0, il faudra utiliser une déclaration du type :
   `unsigned int * a=new int[32]();` qui déclare une zone en mémoire de 32 `unsigned int` consécutifs ET les initialise à 0. L'adresse de la première case de cette zone est renvoyée et correspond à la valeur stockée par `a`.
2. Il est souvent plus lisible de déclarer un nouveau type pour représenter un entier long (qui sera un `unsigned int *`), mais ce n'est pas une obligation.
   Pour ceux qui souhaiteraient procéder ainsi, voici comment déclarer un nouveau type appelé `lentier` :
   ```
   typedef unsigned int * lentier;
   ```
   Ainsi, la déclaration précédente `unsigned int *a=new unsigned int[m];` peut être remplacée par `lentier a=new unsigned int[m];`
3. Libération de la mémoire : une fois la zone mémoire non utilisée, on pourra la libérer grâce à l'opérateur `delete`.
   Pour libérer la zone mémoire pointée par `a` (précédemment déclaré), on écrira : `delete[] a;`

3 Algorithmes génériques sur les entiers longs

Ces algorithmes sont issus du *Handbook of Applied Cryptography*, accessible en ligne ici (sections 14.2.2 à 14.2.5).

Addition multiprécision

Entrées : $A = (a_{n-1} \dots a_1 a_0)_r$ et $B = (b_{n-1} \dots b_1 b_0)_r$ deux entiers positifs ayant chacun n chiffres en base r

Sortie : $A + B = (s_n \dots s_1 s_0)_r$ la somme sur $n + 1$ chiffres en base r.

Remarque : le dernier chiffre s_n est la dernière retenue et est forcément telle que $s_n \leq 1$

Algorithme :

1. $c \leftarrow 0$
2. Pour i de 0 à $n - 1$ faire
   (a) $s_i \leftarrow (a_i + b_i + c)$ reste r
   (b) Si $a_i + b_i + c < r$
       Alors
           $c \leftarrow 0$
       Sinon
           $c \leftarrow 1$
       FinSi
   FinPour
3. $s_n \leftarrow c$
4. Retourne $(s_n \dots s_1 s_0)_r$

Remarque 1 *Il faudra faire attention à ce que les nombres additionnés aient le même nombre de chiffres. Si ce n'est pas le cas, il est tout à fait possible de rajouter des* 0 *sur les poids les plus forts du nombre ayant le plus petit nombre de chiffres pour atteindre une taille identique.*

Soustraction multiprécision

Entrées : $A = (a_{n-1} \ldots a_1 a_0)_r$ et $B = (b_{n-1} \ldots b_1 b_0)_r$ deux entiers positifs ayant chacun n chiffres en base r et tels que $a \geq b$

Sortie : $A - B = (s_{n-1} \ldots s_1 s_0)_r$ la somme sur n chiffres en base r.

Algorithme :

1. $c \leftarrow 0$
2. Pour i de 0 à $n-1$ faire
   (a) $s_i \leftarrow (a_i - b_i + c)$ reste r
   (b) Si $a_i - b_i + c \geq 0$
       Alors
         $c \leftarrow 0$
       Sinon
         $c \leftarrow -1$
      FinSi
   FinPour
3. Retourne $(s_{n-1} \ldots s_1 s_0)_r$

Remarque 2 *Il est très important que a soit plus grand que b. Cet algorithme doit donner un résultat forcément positif.*
Il faudra donc proposer un algorithme qui permet de comparer a et b avant d'effectuer la soustraction (voir section 4).
De plus comme pour l'addition, il faudra également faire attention à ce que les nombres additionnés aient le même nombre de chiffres. Si ce n'est pas le cas, il est tout à fait possible de rajouter des 0 *sur les poids les plus forts du nombre ayant le plus petit nombre de chiffres pour atteindre une taille identique.*

Multiplication multiprécision

Entrées : $A = (a_{n-1} \ldots a_1 a_0)_r$ et $B = (b_{t-1} \ldots b_1 b_0)_r$ deux entiers positifs ayant respectivement n et t chiffres en base r.

Principe : $A \times B = \sum_{i=0}^{t-1} \left(\sum_{j=0}^{n-1} a_i \cdot b_j r^{i+j} \right)$

Sortie : $A \times B = (w_{n+t-1} \ldots w_1 w_0)_r$

Algorithme :

1. Pour i de 0 à $n+t-1$ faire
       $w_i \leftarrow 0$
   FinPour
2. Pour i de 0 à $t-1$ faire
   (a) $c \leftarrow 0$
   (b) Pour j de 0 à $n-1$ faire
         `temp` $\leftarrow w_{i+j} + a_i \times b_j + c$
         $w_{i+j} \leftarrow$ `temp` reste r
         $c \leftarrow$ `temp` div r
      FinPour

(c) $w_{i+n} \leftarrow c$

FinPour

3. Retourne $(w_{n+t-1} \dots w_1 w_0)_r$

Remarque 3 `temp` *désigne un nombre de deux chiffres en base* r*. L'opération* `temp` *reste* r *consiste à prendre le chiffre de poids faible en base* r*. L'opération* `temp` *div* r *consiste à prendre le chiffre de poids fort en base* r*.*

Division multiprécision

Entrées : $A = (A_{n-1} \dots A_1 A_0)_r$, $B = (B_{t-1} \dots B_1 B_0)_r$ deux entiers représentés respectivement par n et t chiffres en base r avec $n \geq t \geq 1$, $B_{t-1} \neq 0$

Sortie : Quotient $Q = (Q_{n-t} \dots Q_1 Q_0)_r$ et reste $R = (R_{t-1} \dots R_1 R_0)$ tel que $A = BQ + R$ et $0 \leq R < B$

Algorithme :

1. Pour j de 0 à $n - t$

   $Q_j \leftarrow 0$

   FinPour

2. Tant que $A \geq Br^{n-t}$ faire

   $Q_{n-t} \leftarrow Q_{n-t} + 1$

   $A \leftarrow A - Br^{n-t}$

3. Pour i de $n - 1$ à t faire

   (a) Si $A_i = B_{t-1}$

   Alors

   $Q_{i-t} \leftarrow r - 1$

   Sinon

   $Q_{i-t} = \left\lfloor \frac{A_i r + A_{i-1}}{B_{t-1}} \right\rfloor$

   FinSi

   (b) Tant que $Q_{i-t}(B_{t-1}r + B_{t-2}) > A_i r^2 + A_{i-1} r + A_{i-2}$ faire

   $Q_{i-t} \leftarrow Q_{i-t} - 1$

   FinTantque

   (c) $A \leftarrow A - Q_{i-t} B r^{i-t}$

   (d) Si $A < 0$

   Alors

   $A \leftarrow A + Br^{i-t}$

   $Q_{i-t} \leftarrow Q_{i-t} - 1$

   FinSi

   FinPour

4. $R \leftarrow A$

5. Retourne (Q, R)

Remarque 4 *Cet algorithme est le plus compliqué des algorithmes multiprécision. Il vous est conseillé de l'appliquer une fois à la main en base 10 pour comprendre son fonctionnement.*
Cet algorithme fait appel à de nombreuses fonctions (multiplications multiprécisions, additions multiprécisions, soustractions multiprécisions, comparaisons multiprécisions) qui doivent impérativement être fiables (et donc avoir été testées).
Cet algorithme sera à adapter en fonction des choix que vous aurez fait pour l'implantation des autres fonctions.

4 Prototypes des fonctions à définir

Se référer et s'inspirer des algorithmes de la section 3 pour définir ces fonctions.
Vous respecterez rigoureusement le prototype imposé.

1. `add_classique` : additionne deux `lentier` `a` et `b` de tailles respectives `t` et `n` et retourne un `lentier` résultat de l'addition `a+b`.
   Prototype :
   ```
   lentier
   add_classique(lentier a, lentier b, unsigned int t, unsigned int n);
   ```
   Remarque : il faut définir une taille commune pour les deux entiers (la plus grande des deux tailles) et compléter par des 0 en poids fort pour l'entier le plus petit.
2. `sub_classique` : Pour $a \geq b$, effectue l'opération de soustraction entre `lentier a` et `lentier b` de tailles respectives `t` et `n` et retourne un `lentier` résultat de la soustraction `a-b`.
   Prototype :
   ```
   lentier
   sub_classique(lentier a, lentier b, unsigned int t, unsigned int n)
   ```
   Remarque : Il faut s'assurer que a≥b. C'est l'objet de la fonction suivante.
3. `cmp_lentier` : compare deux `lentier` `a` et `b` de même taille et retourne :
   0 si ils sont égaux
   1 si $a > b$
   -1 si $a < b$
   Prototype :
   ```
   char
   cmp_lentier(lentier a,lentier b,unsigned int t)
   ```
4. `mult_classique` : multiplie deux `lentier` `a` et `b` de tailles respectives `t` et `n` et retourne un `lentier` résultat de la multiplication a×b.
   Prototype :
   ```
   lentier
   mult_classique(lentier a, lentier b, unsigned int t, unsigned int n)
   ```
   Remarque : la taille du résultat doit être $n + t$. Voir l'algorithme de multiplication multiprécision section 3.
5. `div_eucl` : effectue la division euclidienne de `lentier a` par `lentier b` de tailles respectives `n` et `t` et retourne un `lentier` correspondant au reste : a reste b.
   Prototype :
   ```
   lentier
   div_eucl(lentier a, lentier b, unsigned int n, unsigned int t)
   ```
   Remarque : la taille du résultat doit être au maximum t puisque le reste est nécessairement $< b$. Voir l'algorithme de division multiprécision section 3.
   Cet algorithme utilise les fonctions précédentes.

Important : pensez à bien tester chacune de vos fonctions avant d'aller plus loin.

Des vecteurs de test seront mis à disposition sur l'ENT.

5 Applications

L'objectif est maintenant d'utiliser les fonctions multiprécisions définies dans les parties précédentes pour effectuer des calculs de cryptographie asymétrique.
Une opération principale est au coeur de la cryptographie asymétrique, la multiplication modulaire.

Multiplication modulaire

La multiplication modulaire est définie comme étant l'opération $A \times B \mod N$ où A et B sont les opérandes et N un nombre spécial appelé *module*. A, B et N sont des entiers multiprécisions.
Cette opération se décompose en deux étapes :

1. Calcul du produit : $P \leftarrow A \times B$
2. Réduction du produit modulo N, c'est à dire calcul de :
   $R \leftarrow P$ reste N
   à l'aide d'un algorithme de division euclidienne.

Remarque 5 *Le résultat d'un calcul de ce type est donc nécessairement* $< N$.

`mul_mod` : effectue la multiplication modulaire de `lentier a` par `lentier b` modulo un `lentier N` et retourne un `lentier` correspondant au reste : $a \times b$ reste N.
`a` et `b` sont supposés être <`N`, donc seule la taille de `N` compte.
Cette taille est donnée par un `unsigned int n`.
Prototype :

```
lentier
mul_mod(lentier a, lentier b, lentier N, unsigned int n);
```

Exponentiation modulaire

L'exponentiation modulaire est une opération très utilisée en cryptographie asymétrique, chiffrement RSA ou encore pour signer des documents électroniques.
Par définition, l'exponentiation modulaire consiste à calculer : a^e reste N où a, e et N sont de très grands entiers.
Calculer $\underbrace{a \times a \times \cdots \times a}_{e \text{ fois}}$ pour e très grand (e sur 1024 bits, soit 2^{1024} opérations de multiplication modulaire) n'est pas réalisable.
Pour donner un ordre de grandeur, il y a environ 2^{266} atomes dans l'univers visible. Si un ordinateur idéal était capable de calculer une multiplication modulaire en 1ns, soit environ 2^{30} multiplications modulaires par seconde, il faudrait 2^{994} secondes pour faire ce calcul, ce qui représente un peu plus du cube du nombre d'atomes dans l'univers...
Pourtant nous allons effectuer cette opération mais avec un algorithme plus efficace qui ne nécessite qu'un nombre de multiplications modulaires de l'ordre du millier.

Algorithme Square and Multiply

Cet algorithme est basé sur la décomposition en base 2 de l'exposant e.
On désigne par e_i le **bit** d'indice i de e et par n_e la taille **en bits** de e (ce qui implique que $x_{n_e-1} = 1$ et que pour tout $i \geq n_e$, $x_e = 0$).
e peut donc s'écrire : $e = (e_{n_e-1} \ldots e_1 e_0)_2$.
Pour calculer a^e reste N selon la méthode Square and Multiply, le principe est le suivant :

```
P ← a
Pour i de n_e − 2 à 0 faire
   P ← P^2 reste N {square}
   Si e_i = 1
      Alors
         P ← P × a reste N {multiply}
   FinSi
FinPour
Retourner P
```

Exemple :
Calcul de : a^{19} reste N
Décomposition de l'exposant en binaire : $19 = 2^4 + 2^1 + 2^0 = (10011)_2$

$P = a \quad \{\text{initialisation}\}$

$e_3 = 0 \Rightarrow P = (a)^2$ reste N

$e_2 = 0 \Rightarrow P = \left((a)^2\right)^2$ reste N

$e_1 = 1 \Rightarrow P = \left(\left((a)^2\right)^2\right)^2 \times a$ reste N

$e_0 = 1 \Rightarrow P = \left(\left(\left((a)^2\right)^2\right)^2 \times a\right)^2 \times a$ reste N

Cela représente 6 multiplications modulaires au lieu de 19 sur cet exemple.
Plus généralement, il y a au maximum 2 multiplications par bit d'exposant, donc sur un exposant de 1024 bits, il y a au plus 2048 multiplications, au minimum 1024 et en moyenne 1536 pour un exposant ayant autant de 0 que de 1 dans son écriture binaire.

Voici le prototype de la fonction à réaliser :

```
lentier
exp_mod(lentier a, lentier x, lentier N, unsigned int n);
```

avec :
- `a` : l'entier long à élever à la puissance `x` ;
- `N` : l'entier long représentant le module ;
- `n` : la taille (en chiffres de 32 bits) de N ;
- `x` : l'exposant (dont la taille est nécessairement $\leq$ `n`).

6 Compléments

Important :
Vous aurez à utiliser des variables permettant de stocker le résultat d'un produit de deux nombres de 32 bits. Ce résultat se représente sur 64 bits et peut être codé simplement avec le type `unsigned long long int`.
Pour indiquer qu'un nombre doit être considéré comme un entier de 64 bits, vous pourrez utiliser la conversion forcée de type.
Exemple : calcul d'un produit entre deux `unsigned int a` et `b` stockés sur un `unsigned long long int` :

```
//déclaration du résultat sur 64 bits
unsigned long long int p;
p=((unsigned long long int)a)*b;/*considère a comme un unsigned long long int
avant de le mutliplier par b*/
```

Si cette conversion de type n'est pas faite, `a*b` est réalisé sur 32 bits (on perd donc les 32 bits de poids les plus forts) puis convertie sur 64 bits avec 32 bits de poids forts à 0...

Une fois ce nombre généré sur 64 bits, vous pourrez utiliser des opérations de décalages/masquages des données pour extraire les bits de poids forts ou les bits de poids faibles.

Remarque 6 *En plus des fonctions imposées, il peut être souhaitable de prévoir des fonctions permettant d'afficher un* `lentier`*, notamment pour comparer avec les vecteurs de tests fournis.*

Interfaçage

Pour entrer un grand nombre, il faut connaître chacun de ses "chiffres" en base $r = 2^{32}$. On aimerait pouvoir entrer directement un nombre sous la forme du chaîne de caractères contenant tous les chiffres en base 10 de notre grand entier. Le problème est que les fonctions de calcul précédentes ne peuvent pas manipuler directement les entiers représentés en base 10 comme des chaînes de caractères.
Il faudrait donc écrire une fonction qui permet de convertir cette chaîne de caractères représentant votre grand entier en une structure mémoire correspondant à un `lentier` utilisables par les fonctions précédentes.
Exemple :
Si l'utilisateur saisit la chaîne constituées des chiffres `''1234567891011121314151617''` en base 10, votre fonction devra retourner un pointeur vers la structure constituée de :
`{3197516993, 255446423, 66926}` où les mots de 32 bits de poids faibles sont à gauche et les mots de 32 bits de poids fort sont à droite.
Cela signifie que :

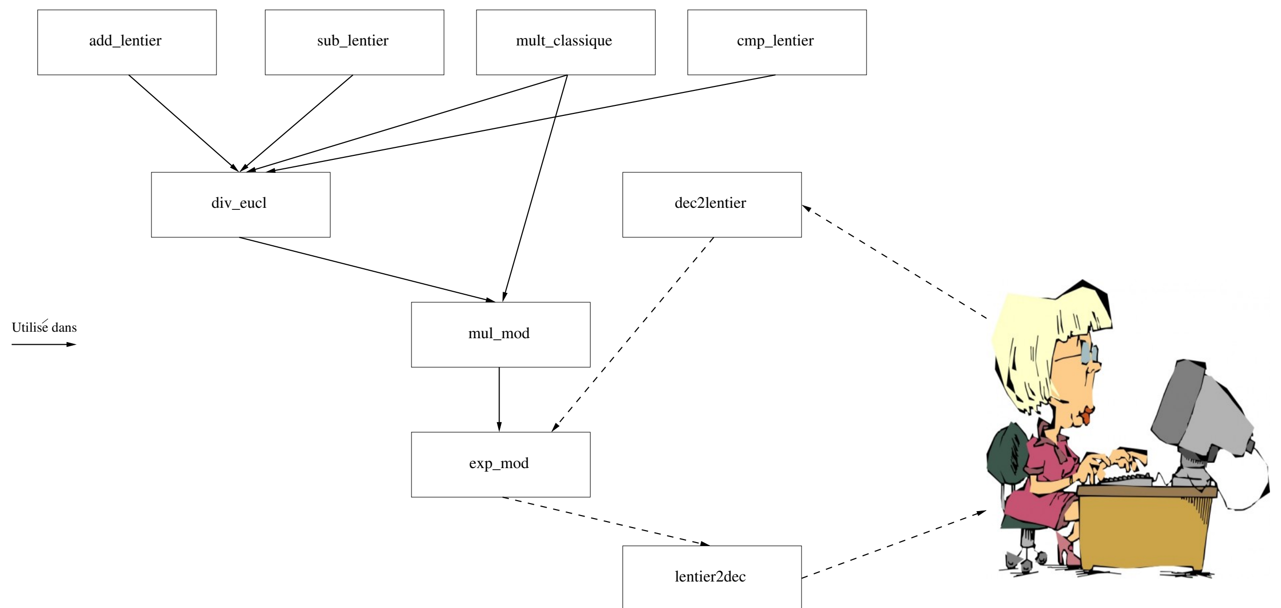
$$66926 \times (2^{32})^2 + 255446423 \times 2^{32} + 3197516993 = 1234567891011121314151617$$

Le prototype de cette fonction est le suivant :

```
lentier
dec2lentier(char* nombre_dec);
```

Il faudra définir également la fonction réciproque. Son rôle est de convertir un entier long représenté par un ensemble de chiffres en base $r = 2^{32}$ en une chaîne de caractères composées des chiffres en base 10 de ce même entier.
Son prototype est le suivant :

```
char*
lentier2dec(lentier nombre_base_r);
```



Remarque importante

Même si des fonctions sont requises pour en définir/coder d'autres, rien n'empêche les groupes concernés d'écrire puis de coder leurs algorithmes en supposant que les fonctions appelées existent et donnent les bons résultats.
Il faudra juste veiller à respecter le prototype des fonctions appelées et je fournirai des valeurs adaptées au cas par cas pour que vous puissiez tester vos fonctions. N'hésitez pas à me solliciter, surtout en début de projet.

Travail à effectuer

Pour chaque groupe de TD, un rapport final sera à rendre à la fin du semestre.
Pour chaque fonction, il faudra :

1. Réecrire l'algorithme au formalisme vu en Info1 (déclaration fonction, RES, Lexique local, Algo local).
   ⇒ Dans le rapport final
2. Expliquer les choix qui sont faits pour implanter l'algorithme en C.
   ⇒ Dans le rapport final en faisant apparaître les extraits de codes commentés.
3. Tester cette fonction dans un algorithme principal à l'aide des vecteurs de tests fournis.
   ⇒ Dans le rapport final sous forme de copie d'écran.

Les groupes sont imposés :

Groupe A	Fonctions	Groupe B
Charles Royer Clément Fayolle Nour Hassoun Alexis Chaudier	`add_lentier` Difficulté : 3.5/5 (0.7)	Sami Mazzi Sébastien Bouvard Klément Rouchouse Dylan Muiras
Mohamed Oudouni Sébastien Cizeron Marc Cwiklinski Grégoire Bouland	`sub_lentier` Difficulté : 3.5/5 (0.7)	Charles Le Houedec Mathieu Henry Fabien Zanoletti Nicolas Crouzet
Hugo Morais Costa Nicolas Galley Charles Silvestre Damien Ciprelli	`mult_classique` Difficulté : 3.5/5 (0.7)	Mélanie Goncalves Alexis Defranoux Lilian Bernard Zo Andria.

Bastien Chabal
Alix Jeannerot
Fabien Bouteyre
Romain Velu
	`mul_mod`
	`exp_mod`
	Difficulté : 4/5 (0.8)
		Julie Dutems
		Axel Arnaud
		Hanae Perez
		Jean Gosse

Antoine Peyrard Manon Portay Corentin Faure Louis Poy Tardieu	`div_eucl` Difficulté : 4.5/5 (0.9)	Raphael Luciano Quentin Perret Nicolas Chalencon Léo Rousseau
Geoffrey Dumon Clément Frade Théo Basty Paul Elian Tabarant	Conversion chaîne ↔ `lentier` `cmp_lentier` Difficulté : 5/5 (1)	Auriane Brialon Charles Mure Yanis Maanane Maxime Desaintjean

Calendrier

03/03/2015 Présentation et lancement du projet

05/05/2015 Réunion Intégration (Evaluation) : chaque groupe doit présenter son avancement et l'intégration de son travail dans les autres fonctions.
Cette réunion en petits groupes sera l'occasion de demander à chacun (évaluer) plus précisément sa participation dans la tâche attribuée, ainsi que de définir le travail restant à faire pour mener à bien le projet.
Une note sera attribuée à chacun à l'issue de cette réunion et comptera pour au moins la moitié de la note finale.

01/06/2015 Rendu final : Rapport + projet en C++ [1] sous la forme de 3 fichiers sources :

(a) `lentier.h` : contenant les déclarations de vos fonctions

(b) `lentier.cpp` : contenant les définitions des fonctions

(c) `main.cpp` : contenant le programme principal avec le test de **chacune** des fonctions suivant les vecteurs de test fournis.

Notation

— Une note globale de projet pour chaque groupe de TD, basée sur le rendu final : N_f

— Une note d'intégration individuelle, à l'issue de la réunion d'intégration : N_i

La note individualisée du projet est donnée par la formule :

$$N_{\text{finale}} = \frac{N_f \times \text{Coeff_difficulté} + N_i \times (2 - \text{Coeff_difficulté})}{2}$$

1. Vous pouvez travailler sous Visual Studio ou tout autre environnement permettant de développer en C++. Seuls les 3 fichiers sources sont à rendre.