2025-2026

Manipulation des entiers multiprécision - Application à la cryptographie asymétrique

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 séance d'amphi en Info2 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.
Cette notion est fondamentale en programmation orientée objet (POO) et doit être parfaitement maîtrisée pour bien commencer le Semestre 3 de CiTiSE2 en informatique.

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".
Les opérations sur lequel le processeur peut travailler directement (ie : qui font partie de son jeu d'instructions) sont appelées 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 importante en cryptographie asymétrique : l'exponentiation modulaire. Cette opération permet en particulier de faire du chiffrement de donnée (système RSA) mais aussi et surtout de la signature de documents électroniques.

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 en base $r = 2^{32}$.
On a donc, pour un entier long de m chiffres de 32 bits chacun :

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

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 unsigned 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. Nous utiliserons un nouveau type `lentier` pour représenter un entier long.
   Ce type sera composé de :
   (a) un pointeur vers la zone mémoire contenant les chiffres de l'entier considéré
   (b) la taille pratique de cet entier : (dans le cas où il y aurait des chiffres nuls au niveau des poids forts.)

   La déclaration de ce nouveau type vous est donnée ci-dessous :

   ```
   typedef struct
   {
     unsigned int* p;
     unsigned int size;
   } lentier;
   ```

   Ainsi, la déclaration précédente `unsigned int *a=new unsigned int[m]();` peut être remplacée par (si on suppose `m` connu) :
   ```
   lentier a;
   a.size=m;
   a.p=new unsigned int[m];
   ```
3. Libération de la mémoire : une fois que la zone mémoire n'est plus utile, il faudra la libérer grâce à l'opérateur `delete`.
   Pour libérer la zone mémoire pointée par `a.p` (précédemment déclaré), on écrira : `delete[] a.p;`
   Cela rend la mémoire au système mais ne détruit pas le pointeur. Le pointeur `a.p` peut alors pointer vers une nouvelle zone de la mémoire qui sera soit existante, soit allouée de nouveau avec l'opérateur `new`.
   Rappel : Il est **indispensable** de libérer cette zone avant de faire pointer le pointeur vers une nouvelle zone.

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).
Il sera nécessaire de les réecrire avec le formalisme vu en Info1.

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.

Remarque1 : le dernier chiffre s_n est la dernière retenue et est forcément telle que $s_n \leq 1$

Remarque2 : si les deux opérandes n'ont pas le même nombre de chiffres, il est tout à fait possible de considérer que les chiffres de poids forts jusqu'à $n-1$ de l'opérande ayant le plus petit nombre de chiffres sont des 0. Il faudra peut être réecrire/adapter l'algorithme...

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 $(s_n \dots s_1 s_0)_r$

Soustraction 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 et tels que $a \geq b$

Sortie : $A - B = (s_{n-1} \dots s_1 s_0)_r$ 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 $(s_{n-1} \dots s_1 s_0)_r$

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

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

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

Algorithme :

1. Pour i de 0 à $n+t-1$ faire
   $w_i \leftarrow 0$
   FinPour
2. Pour i de 0 à $n-1$ faire
   (a) $c \leftarrow 0$
   (b) Pour j de 0 à $t-1$ faire
   $\texttt{temp} \leftarrow w_{i+j} + a_i \times b_j + c$
   $w_{i+j} \leftarrow \texttt{temp}$ reste r
   $c \leftarrow \texttt{temp}$ div r
   FinPour
   (c) $w_{i+t} \leftarrow c$
   FinPour
3. Retourne $(w_{n+t-1} \dots w_1 w_0)_r$

Remarque 2 `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*. Vous réferer à la section 6 pour gérer ce cas.*

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 $n \geq t \geq 2$, $B_{t-1} \neq 0$

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

$A \leftarrow A + B r^{i-t}$

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

FinSi

FinPour

4. $R \leftarrow A$
5. Retourne (Q, R)

Remarque 3 *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.

Remarque 4 *De plus cet algorithme est donné pour une taille de diviseur au moins égale à deux chiffres. Il est conseillé de réecrire l'algo dans le cas particulier où B est constitué d'un seul chiffre. De nombreuses simplifications sont possibles.*
Enfin, dans le cas où B_{t-1} est petit, la boucle 3.(b) peut être extrêmement longue (d'autant plus longue que la base r est grande). Dans ce cas, il est conseillé (faites le !) de normaliser les données et de travailler sur λB et λA pour un λ tel que :

— *B soit toujours sur t chiffres, et $B_{t-1} \geq \frac{r}{2}$.*

— *λ peut être pris comme une puissance de 2 pour que la normalisation soit rapide à calculer.*

Dans ce cas, la division de λA par λB donne le même quotient mais comme reste λR, il faut donc dénormaliser le reste en divisant le reste obtenu par λ.

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_lentier` : additionne deux `lentier a` et `b` de tailles respectives `t` et `n` et retourne un nouveau `lentier` résultat de l'addition `a+b`.
   Prototype :

   ```
   lentier Add_lentier(lentier a, lentier b);
   ```

   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. Ou réecrire l'algorithme de manière un peu plus astucieuse. Vous êtes libre de proposer ce que vous voulez, mais il faut gérer ce cas et justifier vos choix.
2. `Sub_lentier` : 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_lentier(lentier a, lentier b)
   ```

   Remarque : Il faut s'assurer que `a`$\geq$`b`. C'est l'objet de la fonction suivante.
3. `Cmp_lentier` : compare deux `lentier a` et `b` et retourne :

0 si ils sont égaux
1 si $a > b$
-1 si $a < b$
Prototype :

```
char Cmp_lentier(lentier a, lentier b)
```

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)
   ```

   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)
   ```

   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` (sinon il faudra d'abord les réduire modulo N en appelant la fonction `div_eucl`), donc seule la taille de `N` compte.
Prototype :

```
lentier Mul_mod(lentier a, lentier b, lentier 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 : $e = (e_{n_e-1} \dots e_1 e_0)_2$.
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 ← P² 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 : $19 = 2^4 + 2^1 + 2^0 = (10011)_2$

$P = a$ {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

$e_0 = 1 \Rightarrow P = \left(\left(\left((a)^2\right)^2 \times a\right)^2 \times a\right)^2 \times a$ 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);
```

avec :

`a` : l'entier long à élever à la puissance `x` ;
`N` : l'entier long représentant le module ;
`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.

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 `''123456789101112131415161 7''` en base 10, votre fonction devra retourner un `lentier` dont les **champs** auront les valeurs suivantes

`size` : valant 3
`unsigned int*` : pointeur d'`unsigned int` vers la zone `{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 :

$$66926 \times (2^{32})^2 + 255446423 \times 2^{32} + 3197516993 = 123456789101112131415161 7$$

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 `taille` 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);
```

Si `nombre_base_r` contient $\begin{array}{|c|c|}\hline \texttt{size} & 3 \\ \hline \texttt{p} & 0xC3FE \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|}\hline 3197516993 & 255446423 & 66926 \\ \hline \end{array}$ la fonction `lentier2dec` doit générer une chaîne de caractère correspondant à `"12345678910111213141516l7"`.

Résumé du découpage fonctionnel

Le découpage fonctionnel et les appels aux fonctions permettant de résoudre le problème du calcul de l'exponentiation modulaire sont résumés dans la figure 1.

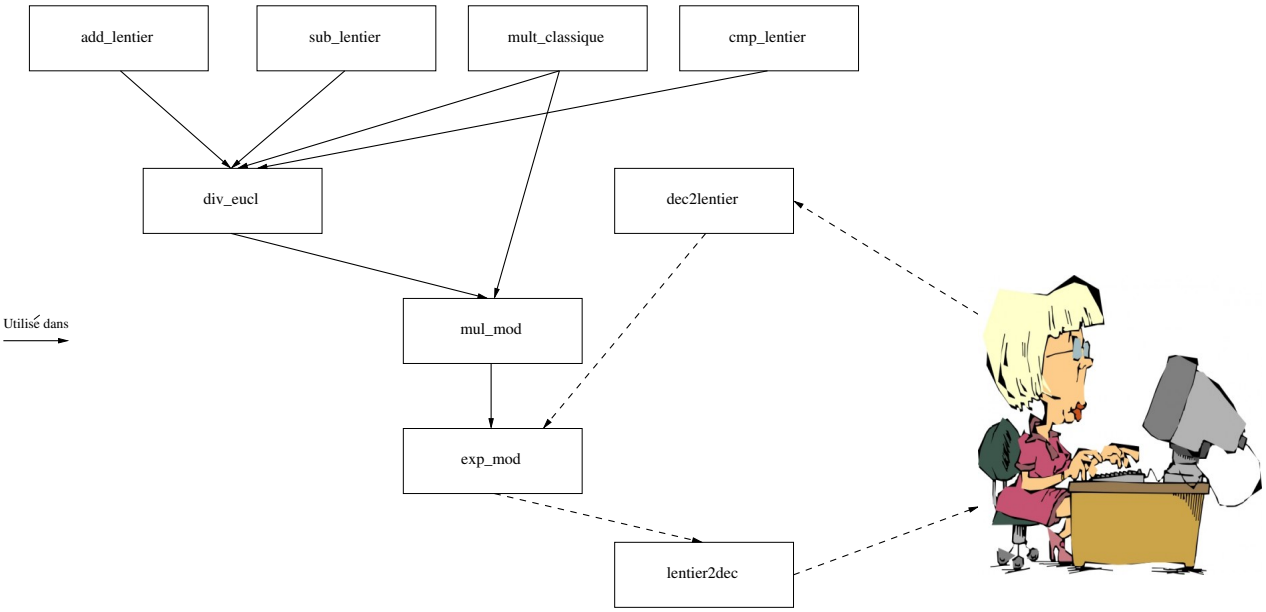

FIGURE 1 – Découpage fonctionnel et lien entre les fonctions

Remarque 6 *Il est tout à fait possible (et même souhaitable pour certaines fonctions) qu'une fonction du découpage nécessite un sous-découpage fonctionnel. Il faudra alors le préciser et faire un schéma comme celui de la figure 1 afin de visualiser les liens entre vos sous-fonctions et celle dont vous avez la tâche.*

En plus de ces fonctions, deux fonctions supplémentaires seront utiles :

1. La fonction `Affiche_lentier` dont le rôle est d'afficher l'entier long passé en paramètre. Cette fonction sera particulièrement utile pour comparer les résultats de vos fonctions avec les vecteurs de test qui seront fournis. Son prototype est le suivant :

   ```
   void Affiche_lentier(lentier a);
   ```

2. La fonction `lAdjust` dont le rôle est de redimensionner un entier long dont les chiffres de poids forts seraient à 0 (dans le cas d'une soustraction par exemple qui donnerait un résultat

beaucoup plus petit que les opérandes de départ). On propose de coder deux versions de cette fonction :

(a) Version1 (prototype : `void lAdjust(lentier &a);`) : Seule la taille est ajustée, les mots machines non utilisés restent en mémoire et ne sont pas utilisés (évite de réallouer l'espace)

(b) Version2 (prototype : `void lAdjust_realloc(lentier &a);`) : La taille est réajustée et une nouvelle zone de cette taille doit être allouée dans laquelle seront copiés tous les chiffres significatifs du `lentier`, en commençant par les poids faibles, la zone précédente devant alors être supprimée.

Le `&` indique un passage par référence (ou par adresse), au lieu de passer une copie de la variable `a`, on lui passe l'adresse de cette variable. Toute modification faite sur `a` (champ `size` ou champ `p`) sera transmise à la fonction appelante. Cela permet d'éviter de passer un pointeur vers la variable `a` à la fonction et de manipuler `a` comme si c'était une variable.

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.

Travail à effectuer

Pour **chaque fonction** (donc pour chaque trinôme), il faudra réaliser un mini rapport consistant à :

1. Réecrire l'algorithme au formalisme vu en Info1 (RES, déclaration fonction, Lexique local, Algo local).
2. Préciser un éventuel redécoupage fonctionnel pour réaliser votre fonction en argumentant vos choix.
3. Expliquer les choix qui sont faits pour implanter l'algorithme en C en faisant apparaître les extraits de codes commentés.
4. Tester votre fonction dans un algorithme principal à l'aide des vecteurs de tests fournis et fournir les captures d'écran montrant les tests effectués.

Tout le monde participe donc au bon déroulement du projet. Ceux qui ont des fonctions un peu plus faciles (ou codage quasi direct des algorithmes) ne doivent pas oublier que leurs fonctions servent de base aux autres et doivent être parfaitement bien écrites et testées ! Par ailleurs, tous les groupes auront à travailler avec la notion de pointeur et se l'approprier, c'est un des objectifs principaux du projet.