TP #3 — Autour d'algorithmes antiques

Le but de ce TP est d'implémenter plusieurs algorithmes de calcul du PGCD (étendu) de deux entiers naturels : ultimement, soit a, b ces entiers on veut calculer trois entiers $d > 0$, u, v, tels que :

$$au + bv = d$$

et d est le plus grand entier naturel divisant à la fois a et b.

Algorithme d'Euclide non étendu

On va dans un premier temps utiliser l'algorithme « d'Euclide », qui est basé sur les observations suivantes (où gcd est la fonction PGCD que l'on cherche à calculer, et r le reste (positif) de la division entière (« euclidienne ») de a par b) :

— $\gcd(a, b) = \gcd(b, a)$
— $\gcd(a, 0) = a$
— $\gcd(a, b) = \gcd(b, r)$ pour $a \geq b > 0$

Les deux premières observations sont par définition du PGCD, et la dernière se démontre de la façon suivante : on note q le quotient de la division entière de a par b, qui est tel que $a - bq = r$. Si l'on note $a = a'd$ et $b = b'd$ avec d entier strictement positif, il est clair que d divise $r = a - bq = (a' - b'q)d$; dans l'autre sens, si d' strictement positif divise $r = r''d'$ et $b = b''d'$, alors d' divise $a = (r'' + b''q)d'$. Les ensembles des diviseurs communs de (a, b) et (b, r) étant les mêmes et finis, on a donc bien égalité entre leurs plus grands éléments respectifs.

1. Écrivez une fonction C *non récursive* de signature :
   ```
   uint64_t egcd(uint64_t a, uint64_t b)
   ```
   qui utilise les observations ci-dessus pour calculer et renvoyer le PGCD de a et b.
2. Montrez qu'aucune des opérations de votre fonction n'entraîne de *wraparound* (de « réduction modulo 2^{64} »).
3. Écrivez une fonction C de signature :
   ```
   bool test_egcd(void)
   ```
   qui teste votre fonction `egcd` sur des entrées variées. Ces tests devront notamment vérifier que `egcd` est correcte quand un ou plusieurs arguments sont nuls ; satisfait $\gcd(a, b) = \gcd(b, a)$ dans des cas variés ; renvoie un résultat correct quand a et b sont premiers entre eux ou non, etc.
4. Testez.

Algorithme du PGCD binaire

On souhaite maintenant implémenter un autre algorithme de calcul de PGCD, *viz.* l'algorithme du PGCD « binaire » (également d'origine antique, plus orientale). Celui-ci est basé sur les observations suivantes :

- $\gcd(a, b) = \gcd(b, a)$
- $\gcd(a, 0) = a$
- $\gcd(a, b) = \gcd(a - b, b)$ pour $a \geq b$
- Soit $a = 2^{e_a} o_a$, $b = 2^{e_b} o_b$ avec o_a, o_b impairs > 0:

$$\gcd(a, b) = 2^{\min(e_a, e_b)} \gcd(o_a, o_b)$$

- Si o_a et $o_b < o_a$ sont impairs, alors $o_a - o_b > 0$ est pair et s'écrit $2^{e_{b'}} o_{b'}$ avec $e_{b'} > 0$ et $o_{b'}$ impair

La troisième observation se prouve de façon analogue à « $\gcd(a, b) = \gcd(r, b)$ » : si l'on note $a = a'd$ et $b = b'd$, il est clair que d divise $a - b = (a' - b')d$; dans l'autre sens, si d' divise $a - b = x''d'$ et $b = b''d'$, alors il divise $a = (x'' + b'')d'$.

La quatrième suit de la décomposition unique en facteurs premiers et de la définition d'un nombre pair.

5. Soit `a`, `b` les écritures en base deux de a, b entiers naturels, montrez que les écritures en base deux de e_a, e_b, o_a, o_b (dans le même sens que ci-dessus) peuvent s'obtenir essentiellement « sans calcul ».
6. En admettant le fait que calculer une division coûte plus cher que calculer une soustraction, quel peut être l'avantage de l'algorithme du PGCD « binaire » relativement à l'algorithme « d'Euclide » quand les arguments sont représentés en base deux ? Cet avantage est-il conservé si les arguments sont représentés en base dix ?
7. Écrivez une fonction C *non récursive* de signature :
   ```
   uint64_t bgcd(uint64_t a, uint64_t b)
   ```
   qui utilise les observations ci-dessus pour calculer et renvoyer le PGCD de a et b.

   Il est possible (mais ce n'est pas nécessaire) d'écrire cette fonction comme une unique boucle « infinie » avec un unique point de sortie. Dans tous les cas, assurez vous que chaque itération de la boucle « progresse », et qu'elle terminera bien un jour (par exemple, si la valeur d'une de vos variables peut ne *jamais* changer, c'est mauvais signe).
8. Écrivez une fonction C de signature :
   ```
   bool test_bgcd(void)
   ```
   qui applique les tests de `test_egcd` à votre fonction `bgcd`.
9. Testez.
10. Écrivez une fonction C de signature :
    ```
    bool test_compare_e_b(uint64_t bnd)
    ```
    qui compare les résultats renvoyés par `egcd` et `bgcd` pour tous les arguments dans l'intervalle $[\![0, bnd[\![$.

11. Testez.

Algorithme d'Euclide étendu

On peut « étendre » l'algorithme « d'Euclide » pour également calculer les coefficients « de Bézout ». Le résultat d'un calcul de PGCD étendu est alors formé de trois entiers : le PGCD d et les coefficients u et v. Afin de représenter ce résultat en C, on utilisera le type `struct gcof` suivant (en cas de besoin, consultez les notes de cours à propos des `struct`) :

```
struct gcof
{
    uint64_t d;
    int64_t u;
    int64_t v;
};
```

Les champs `u` et `v` sont de type signé `int64_t`, car même pour un PGCD de nombres naturels, l'un des coefficients u et v peut être négatif.

Dans toute la suite du TP, on supposera qu'on ne calcule que des PGCDs de nombres suffisamment petits pour ne jamais causer de dépassement de capacité ou de wrap-around.

L'extension de l'algorithme « d'Euclide » se fait en maintenant deux couples de variables `ua`, `va` et `ub`, `vb` tels que pour les variables `a` et `b` utilisées lors du calcul du PGCD, on maintienne (en un point bien choisi) les égalités :

- `a*ua + b*va == a`
- `b*ub + b*vb == b`

au cours du calcul.

Ceci peut se faire en utilisation l'initialisation triviale `ua = 1`, `va = 0` et `ub = 0`, `vb = 1`, et l'égalité (garantie en C) entre `a % b` et `a - b*(a/b)`.

EXEMPLE. Soit $a = 15$, $b = 12$, le calcul de leur PGCD étendu par l'approche ci-dessus fait successivement intervenir les égalités :

```
// 1
a == 15 == 1*15 + 0*12
b == 12 == 0*15 + 1*12
// 2
a == 12 == 0*15 + 1*12
b == 3 == 1*15 + -1*12 // b = a - b*(a/b) == 15 - 12*1
                       //  == (1*15 + 0*12) - (0*15 + 1*12)
// 3
a == 3 == 1*15 + -1*12
b == 0 == -4*15 + 5*12
```

d'où l'on déduit $d = 3$, $u = 1$, $v = -1$.

12. Écrivez une fonction C *non récursive* de signature :
```
struct gcof xgcd(uint64_t a, uint64_t b)
```
qui utilise l'approche ci-dessus pour calculer et renvoyer le PGCD étendu de a et b.
ATTENTION : les coefficients renvoyés doivent permettre de vérifier l'égalité $au + bv = d$ pour a et b les arguments *effectivement* fournis (pensez à correctement traiter le cas $a < b$).

Le résultat d'un calcul de PGCD étendu fournit suffisamment d'informations pour pouvoir valider sa correction : il suffit de vérifier que d divise effectivement a et b et que l'on a bien $au + bv = d$, ce qui se fait aisément.

13. Écrivez une fonction C de signature :
```
bool is_valid_xgcd(uint64_t a, uint64_t b,
                   struct gcof g)
```
qui effectue une telle vérification. Pensez à correctement traiter le cas d'un PGCD nul.

14. Écrivez une fonction C de signature :
```
bool test_xgcd(uint64_t bnd)
```
qui teste votre fonction `xgcd` pour tous les arguments dans l'intervalle $[\![0, bnd[\![$.

15. Testez.

Inversion modulaire

L'une des (nombreuses) applications du calcul du PGCD étendu est de permettre d'efficacement calculer les *inversions modulaires* (quand elles existent).
On définit l'*inverse* d'un entier naturel $x > 0$ *modulo* $N > x$ (ou juste *inverse modulaire* si le module N est clair dans le contexte) comme l'unique entier $y \in [\![0, N-1]\!]$ tel que $xy \equiv 1\ [N]$, si il existe. On a alors que si x est premier avec N (c'est à dire, si leur PGCD vaut 1), le calcul du PGCD étendu de x et N renverra u et v entiers tels que $xu = 1 + (-v)N$, ce qui permet d'en déduire l'inverse de x ; sinon, x n'a pas d'inverse modulo N (pouvez-vous le prouver ?).

16. Écrivez une fonction C de signature :
```
uint64_t modinv(uint64_t x, uint64_t n)
```
qui pour des arguments bien définis (essayez de rejeter les arguments non définis de façon explicite) calcule et renvoie l'inverse modulaire de x modulo n si il existe, et `0` sinon.
On suppose toujours que cette fonction ne sera appelée que pour des arguments n'entraînant aucun dépassement de capacité ou *wraparound*.

REMARQUE. Le fait que `0` n'est inverse d'aucun nombre (sauf éventuellement lui-même, si l'on veut) fait qu'on peut ici utiliser cette valeur comme valeur « spéciale » pour signifier qu'un argument n'est pas inversible. C'est une

approche valable mais qui n'est pas très robuste, car on ne peut pas forcer un ou une utilisatrice de la fonction à vérifier et correctement traiter une telle valeur. Certains langages de programmation fournissent des mécanismes imposant une telle vérification, par exemple *via* l'utilisation de *type* `Option` ; c'est notamment le cas de Rust ou OCaml.

17. Écrivez une fonction C de signature :

```
bool is_valid_modinv(uint64_t x, uint64_t xi, uint64_t n)
```

qui vérifie que *xi* est bien inverse modulaire de *x* modulo *n*. On pourra se contenter d'une vérification partielle, c'est à dire que si *xi* vaut `0`, il n'est pas nécessaire de vérifier que *x* est bien non inversible modulo *n*.
(Ceci n'est pas tout à fait sans problèmes : comment pourriez-vous faire pour trivialement passer tous ces tests ?)

18. Écrivez une fonction C de signature :

```
bool test_modinv(uint64_t bnd)
```

qui vérifie (partiellement) que votre fonction `modinv` calcule correctement tous les inverses modulo tous les modules plus petits que *bnd*.

19. Testez.

N.B. Dans certains cas (notamment celui des modules premiers (pourquoi ?)), les inverses modulaires peuvent être calculés par exponentiation modulaire, et l'utilisation d'un algorithme d'exponentiation rapide est alors une alternative valable au calcul du PGCD étendu.