PROGRAMMATION C - PROJET D'ALGÈBRE : FACTORISATION ET GRANDS ENTIERS

1. Factorisation

Soit N un entier positif. On sait que N s'écrit de manière unique dans la forme

$$N = \prod_{i=1}^{n} p_i^{e_i},$$

où les p_i sont des premiers distincts, et les e_i sont des entiers positifs.
Toutefois, il est en pratique très difficile d'écrire cette factorisation en un temps raisonnable.
Par conséquent, il y a beaucoup d'intêret pour les algorithmes de factorisation, et les problèmes de factorisation sont aussi utilisés en cryptographie comme problèmes difficiles.
On va considérer le problème suivant :

Problème 1. *Soit $N \in \mathbb{N}_{>1}$ un entier composé. On cherche à trouver $1 < M < N$ tel que $M \mid N$.*

Il est évident que si on sait résoudre ce problème on pourra, par récurrence, résoudre le problème de factorisation.

On va présenter dans cette section, deux algorithmes très faciles pour chercher à résoudre le Problème 1. Il en existe, bien sûr, beaucoup d'autres plus compliqués (mais plus rapides et efficaces !)...

1.1. **Divisions successives.** L'algorithme le plus trivial pour trouver un facteur de N consiste à essayer de diviser par tous les premiers $p \leq \sqrt{N}$.
→ Implementer l'algorithme des divisions successives (on pourra utiliser l'exercice du crible d'Erathostene pour avoir la liste des premiers).

1.2. **L'algorithme "rho" de Pollard.** Cet algorithme est un algorithme probabiliste dû à Pollard (1975) et prend son nom de la forme de la lettre grèque ρ.
L'idée est la suivante : on considère une fonction $f : \mathbb{Z}_N \to \mathbb{Z}_N$ (par exemple : $f(x) = x^2 + 1$) et un point initial $x_0 \in \mathbb{Z}_N$. On définit récursivement $x_i \in \mathbb{Z}_N$ par $x_i = f(x_{i-1})$ pour tout $i > 0$.
On va supposer que la suite $(x_i)_{i \geq 0}$ se comporte comme une suite aléatoire d'éléments indépendants de $\mathbb{Z}_N$.
Soit p un diviseur premier de N (que nous ne connaissons pas encore). Supposons qu'il existe une collision $\bmod p$, c'est-à-dire deux entiers m et n tels que $x_m \equiv x_n \bmod p$
Si N n'est pas de la forme $N = p^k$ $(k \geq 1)$ et que q est un autre diviseur premier de N, alors $x_i \bmod p$ et $x_i \bmod q$ sont deux variables aléatoires indépendantes (par le théorème des Restes Chinois) donc il est très probable que $x_m \not\equiv x_n \bmod q$ et donc $\text{pgcd}(x_m - x_n, N)$ soit un facteur non trivial de N.
Une amélioration (Floyd) consiste à considérer une deuxième suite $y_i = x_{2i}$ et à tester seulement les congruences de la forme $x_i \equiv y_i$. Cela nous permet de ne pas devoir stocker tous les x_i.
Grâce au paradoxe des anniversaires on s'attend à une collision après $O(\sqrt{p})$ étapes (heuristiquement).

(TOURNEZ SVP)

Date: 17 novembre 2010.

Algorithme 1. input : $N \geq 3$ pas de la forme p^k pour p premier, $k \geq 1$.
output : soit un facteur propre de N, soit "echec"

(1) Initialisation : on fixe x_0, $y_0 = x_0$, $i = 0$.
(2) Répéter :
- $i = i + 1$, $x_i = x_{i-1}^2 + 1 \mod N$, $y_i = (y_{i-1}^2 + 1)^2 + 1 \mod N$
- $g = \text{pgcd}(x_i - y_i, N)$
- Si $1 < g < N$ retourner g
- Si $g == N$ retourner "echec".

→ Implementer l'algorithme d'Euclide pour le pgcd.
→ Implementer l'algorithme rho de Pollard (x_0 peut être un nombre aléatoire ou un nombre fixé).
→ Tester les deux algorithmes sur des nombres composés (exemples : $N = 82123$, $N = 39621943$, $N = 316592561$).
Remarque : Pour des nombres aussi petits que ceux qu'on vient de tester il est possible que l'algorithme trivial soit plus rapide, ce qui n'est pas vrai dès que la taille des nombres à factoriser augmente.

2. Grands entiers

Le langage C ne nous permet que d'utiliser des entiers de taille très petite. En particulier, sur une machine à 32 bits, on ne pourra pas dépasser $n = 2^{32} - 1$ pour un `unsigned int`.
Cela est très limitant si on veut tester nos fonctions de factorisation sur des gros entiers, et en voir la rapidité.
On va donc construire un nouveau type `big_int` qui nous permettra de manipuler des entiers de taille quelconque.
On va répresenter un `big_int` comme un tableau de `size` éléments de type `unsigned int` (bornés par une valeur `B`).
Suggestion : On peut prendre `size`= 20 et `B`= 10^4 (attention : pour éviter d'avoir des problèmes avec les retenues on va supposer $B^2 < 2^{32} - 1$).
On pourra, par exemple, réaliser les `big_int` comme un objet de type `struct` (ce qui nous permettrait d'ajouter aussi un champs `ok` pour voir quand le nombre a depassé la taille permise) ou comme un pointeur à un tableau alloué dynamiquement.
→ Programmer les fonctions suivantes (pour travailler avec des entiers de taille quelconque) :

(1) Une fonction qui initialise un objet de type `big_int` (avec tous les éléments à zéro, sauf `ok` qui doit valoir 1);
(2) Une fonction `print_big_int` pour imprimer un `big_int`. Attention aux zéros : on pourra virer ceux tout à gauche, mais on devra imprimer ceux à l'interieur du nombre avec la bonne multiplicité (exemple : $100001111 \neq 10111$);
(3) Une fonction `equal` qui compare deux `big_int` a et b et vérifie si $a = b$;
(4) Une fonction `bigger` qui compare deux `big_int` a et b et vérifie si $a > b$;
(5) Une fonction `somme` (attention aux retenues et aussi à signaler une erreur si on dépasse la taille maximale);
(6) Une fonction `soustraction` (qui ne marche que si le premier élément est plus grand ou égal que le deuxième);
(7) Une fonction `produit` (attention aux retenues);
(8) Une fonction `division` (Euclidienne) d'un `big_int` par un `unsigned int` <`B`;
(9) Une fonction `division` (Euclidienne) d'un `big_int` par un `big_int`;
(10) Une fonction `modulo` pour calculer la réduction d'un `big_int` modulo un autre `big_int`;
(11) (Facultatif) Une fonction pour rentrer un `big_int` (à partir de l'input standard ou encore d'un fichier) : il faudra le rentrer comme chaîne de caractère et puis initialiser correctement le tableau;
(12) Ceux qui n'auront pas fait la fonction ci-dessus devront rentrer à la main les `big_int` à tester dans le code.

3. Factorisation de grands entiers

On va maintenant pouvoir appliquer les fonctions construites à la Section 2 pour factoriser des grands entiers.
→ Re-écrire l'algorithme d'Euclide pour les `big_int`.
→ Re-écrire l'algorithme rho de Pollard pour les `big_int`.
→ Tester sur des exemples ($N = 324352361371$, $N = 58800637410731$, ...).

Remarque concernant le choix des nombres à factoriser : Si tout marche et que vous voulez tester des exemples plus grands, un choix intéressant de nombre composé N est $N = p * q$ avec p et q premiers d'à peu près la même taille. En effet, c'est sur la factorisation de nombres de ce type que se base la difficulté de RSA. En outre, de cette manière, on est sûr de ne pas tomber sur un nombre avec un tout petit facteur (factorisable donc facilement par divisions successives) ni d'un nombre de la forme $N = p^k$.

4. Documents à rendre pour le projet (date limite 15 décembre)

- Il faudra, bien sûr, rendre tous les fichiers .c (et .h, éventuellement) du projet.
- Il est recommandé, aussi, de rendre un petit rapport, pour expliquer vos fonctions et afficher quelques calculs d'exemples.