Arithmétique et cryptographie

Dans ce TP, nous utiliserons le module `big_int` de Caml Light qui implémente des entiers en précision arbitraire. Ce module définit un type abstrait `big_int` des entiers. Afin d'afficher convenablement les valeurs de ce type, on commencera par installer le "pretty-printer" correspondant :

```
#open "big_int";;
let pprinter b = format__print_string (string_of_big_int b);;
install_printer "pprinter";;
```

Les fonctions de ce module nécessaires à ce TP sont données à la fin de ce sujet.

1 Algorithme d'Euclide

1.1 Algorithme d'Euclide originel

Soient $u, v \in \mathbb{N}$. L'algorithme suivant, dit *algorithme d'Euclide*, calcule le pgcd de u et v :

A1 Si $v = 0$ alors la réponse est u.

A2 Faire $(u, v) \leftarrow (v, u \bmod v)$. Retourner en **A1**.

• Justifier cet algorithme (on posera $0 \wedge 0 = 0$ par convention). Ecrire une fonction `euclide : big_int -> big_int -> big_int` implémentant l'algorithme d'Euclide.

Complexité. La complexité de l'algorithme d'Euclide est donnée par le résultat suivant :

Théorème 1 (G. Lamé, 1845) *Si $0 \leq u, v < N$, le nombre de divisions dans l'algorithme d'Euclide appliqué à u et v est au plus $\lceil \log_\phi(\sqrt{5}N) \rceil - 2$, où ϕ est le nombre d'or $\frac{1+\sqrt{5}}{2}$.*

1.2 Algorithme d'Euclide étendu

Soient $u, v \in \mathbb{N}$. L'algorithme d'Euclide peut-être adapté pour calculer, en même temps que le pgcd de u et v, les coefficients de Bezout. L'algorithme suivant calcule un vecteur (u_1, u_2, u_3) tel que $uu_1 + vu_2 = u_3 = u \wedge v$.

B1 $(u_1, u_2, u_3) \leftarrow (1, 0, u)$, $(v_1, v_2, v_3) \leftarrow (0, 1, v)$.

B2 Si $v_3 = 0$ alors la réponse est (u_1, u_2, u_3).

B3 Soit $q = \lfloor u_3/v_3 \rfloor$. Faire

$(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - q(v_1, v_2, v_3)$,

$(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3)$

$(v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3)$.

Retourner en **B2**.

• Justifier cet algorithme (on déterminera l'invariant de boucle). Ecrire une fonction `bezout : big_int -> big_int -> big_int * big_int * big_int` qui implémente cet algorithme.

1.3 Application : division modulo m

Soient $u, v, m \in \mathbb{N}^\star$ tels que $v \wedge m = 1$. On appelle quotient de u par v modulo m tout entier w tel que $0 \leq w < m$ et $u \equiv vw \pmod{m}$.

• Ecrire une fonction calculant le quotient de u et v modulo m.

2 Décomposition en facteurs premiers

Etant donné $n \in \mathbb{N}$, n s'écrit de manière unique sous la forme

$$n = p_1 p_2 \ldots p_k \qquad \text{avec} \quad p_1 \leq p_2 \leq \ldots \leq p_k \quad \text{et} \quad p_i \text{ premier} \tag{1}$$

On se propose ici de déterminer les facteurs p_i.

2.1 Méthode par division

La méthode la plus simple est la suivante : si $n > 1$ on teste la divisibilité de n par les nombres premiers successifs $p = 2, 3, 5, \ldots$ jusqu'à ce que $n \equiv 0 \pmod{p}$. On remplace alors n par n/p et on reprend à p. Lorsque $n \not\equiv 0 \pmod{p}$ avec $\lfloor n/p \rfloor \leq p$ on s'arrête, avec n premier.

Cette méthode a l'inconvénient qu'il faut déterminer la séquence des nombres permiers. Mais on peut simplifier cette méthode de la façon suivante : Soit $n \in \mathbb{N}^\star$ dont on souhaite déterminer la décomposition en facteurs premiers. Soit (d_i) une séquence d'entiers

$$2 = d_0 < d_1 < d_2 < \ldots$$

qui inclut tous les nombres premiers $\leq \sqrt{n}$ et au moins un élément $d_k \geq \sqrt{n}$. Alors l'algorithme suivant détermine la décomposition de n en facteurs premiers :

A1 $t \leftarrow 0$, $k \leftarrow 0$.

A2 Si $n = 1$ c'est terminé.

A3 On divise n par d_k : $n = qd_k + r$ avec $(0 \leq r < d_k)$.

A4 Si $r = 0$ alors $t \leftarrow t + 1$, $p_t \leftarrow d_k$, $n \leftarrow q$. Aller en **A2**.

A5 Si $q > d_k$ alors $k \leftarrow k + 1$ et aller en **A3**.

A6 $t \leftarrow t + 1$, $p_t \leftarrow n$. C'est terminé.

• Justifier cet algorithme. Ecrire une fonction `decomp : big_int list -> big_int -> big_int list` prenant en argument une séquence (d_k) pour n, n et rendant la séquence des p_i telle que (**??**).

On pourrait prendre pour (d_k) la séquence $2, 3, 5, 7, \ldots$, c'est-à-dire 2 puis tous les impairs à partir de 3. Mais on peut prendre plutôt la séquence $2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \ldots$, c'est-à-dire ajouter alternativement 2 et 4 à partir de 5 (on supprime ainsi tous les multiples de 2 et 3).

• Ecrire une fonction `dk` prenant n en argument et renvoyant une telle séquence $d_0, \ldots, d_k$ avec $d_k \geq \sqrt{n}$.

On peut gagner encore 20% sur cette séquence en supprimant les entiers de la forme $30m \pm 5$, et encore 14% en supprimant les multiples de 7, etc. Si n est petit on peut utiliser une table des nombres premiers (pour $n \leq 10^6$ il n'y a que 168 nombres premiers dans une telle table).

Complexité. La complexité de cet algorithme est un problème très difficile, mais en pratique l'algorithme n'est plus guère utilisable au delà de 10^6.

2.2 Méthode de Fermat

On se propose ici d'implémenter un autre algorithme de décomposition en facteurs premiers dû à Pierre de Fermat (1643), plus adapté à la recherche de grands facteurs premiers.

Soit N impair s'écrivant sous la forme uv, avec $u \leq v$. On définit alors

$$x = (u+v)/2, \qquad y = (v-u)/2$$

et on a

$$N = x^2 - y^2, \qquad 0 \leq y < x \leq N \tag{2}$$

La méthode de Fermat consiste à rechercher des valeurs de x et y satisfaisant (**??**). Etant donné N impair, l'algorithme suivant détermine le plus grand facteur de N inférieur ou égal à $\sqrt{N}$.

A1 Initialiser : $x' \leftarrow 2\lfloor\sqrt{N}\rfloor + 1$, $y' \leftarrow 1$ et $r \leftarrow \lfloor\sqrt{N}\rfloor^2 - N$ (dans les étapes suivantes, x' correspond à $2x+1$, y' à $2y+1$ et r à x^2+y^2-N).

A2 Si $r > 0$ alors faire $r \leftarrow r - y'$, $y' \leftarrow y' + 2$. Aller en **A2**.

A3 Si $r < 0$ alors faire $r \leftarrow r + x'$, $x' \leftarrow x' + 2$. Aller en **A2**.

A4 Si $r = 0$ alors c'est terminé : on a

$$N = ((x'-y')/2)((x'+y'-2)/2)$$

et $(x'-y')/2$ est le plus grand facteur de N inférieur ou égal à $\sqrt{N}$.

• Justifier cet algorithme. Ecrire une fonction `fermat : big_int -> big_int * big_int` implémentant l'algorithme ci-dessus et renvoyant la décomposition en deux facteurs obtenue. En déduire une nouvelle fonction `decomp2` de décomposition en facteurs premiers.

3 Application à la cryptographie : la méthode RSA

Parmi les procédés cryptographiques, la méthode RSA, découverte par R. Rivest, A. Shamir et L. Adleman en 1978, est l'une des plus utilisée actuellement. Elle fait partie des méthodes dite *à clé publique* : chaque personne possède une clé P *publique* que dont tout le monde peut avoir connaissance (par exemple dans un annuaire), et une clé *secrète* S qu'elle seule connait. Lorsque que je veux envoyer un message M à une personne A je le code avec sa clé publique P_A. Le receveur A le décode alors avec sa clé secrète S_A et peut le lire.

Un tel système fonctionne donc si on a les propriétés suivantes :

1. $S(P(M)) = M$ pour tout message M ;
2. Les paires (S, P) sont toutes distinctes ;
3. Découvrir la clé secrète S à partir de la clé publique P est aussi difficile que déchiffrer le message codé ;
4. Une paire (S, P) peut se calculer facilement.

La méthode RSA fonctionne de la manière suivante : soient x, y et s trois grands nombres premiers, tels que $x, y \leq s$. Soit $N = xy$, et p tel que $ps \bmod (x-1)(y-1) = 1$. On peut alors montrer que pour tout M on a $M^{ps} = M \pmod{N}$.

La clé publique est alors le couple (N, p) et la clé secrète le couple (N, s). Pour coder un message on commence par le découper en entiers inférieurs à N, et on élève alors ces entiers à la puissance p modulo N. Pour le décoder on élève les entiers composant le message à la puissance s modulo N.

• En supposant que l'on dispose d'un générateur de grands nombres premiers, comment engendrer un couple clé publique-clé secrète pour la méthode RSA ?

• J'ai codé un message en le découpant par groupes de 3 caractères, lesquels sont représentés par leur code ASCII sur 2 chiffres. (Exemple : le groupe de trois caractères TAS est représenté par l'entier 846583). Le message est ensuite codé en utilisant la clé publique (N, p) où $N = 49808911$ et $p = 5685669$, et le résultat est

```
41021358   1169160  31809925  17484908  32136910  30925474  22255997
48565640  37770004  48144453   6582428  39919725
```

"Craquer" ce codage RSA pour découvrir le message en clair.

Références

D. E. KNUTH. *The art of computer programming, Volume 2: Seminumerical Algorithms*, Addison-Wesley, 1981.

R. SEDGEWICK. *Algorithms*, Addison-Wesley, 1988.

Annexe : le module `big_int`

Le module `big_int` fournit, entre autres, les valeurs et fonctions suivantes :

```
(* 0 et 1 : *)

value zero_big_int : big_int
value unit_big_int : big_int

(* conversions : *)

value big_int_of_int : int -> big_int
value string_of_big_int : big_int -> string
value big_int_of_string : string -> big_int

(* égalité et comparaisons : *)

value eq_big_int : big_int -> big_int -> bool
value le_big_int : big_int -> big_int -> bool
value ge_big_int : big_int -> big_int -> bool
value lt_big_int : big_int -> big_int -> bool
value gt_big_int : big_int -> big_int -> bool
```

```
(* opérations + - x / et modulo : *)

value add_big_int : big_int -> big_int -> big_int
value sub_big_int : big_int -> big_int -> big_int
value mult_big_int : big_int -> big_int -> big_int
value quomod_big_int : big_int -> big_int -> big_int * big_int
value div_big_int : big_int -> big_int -> big_int
value mod_big_int : big_int -> big_int -> big_int

value power_big_int_positive_big_int: big_int -> big_int -> big_int
```