Tests de primalité

Les nombres premiers sont d'usage fréquent en algorithmique, entre autres en cryptographie, en calcul formel et en calcul multiprécision. Un problème algorithmiquement difficile est de déterminer si un entier n particulier est ou non premier. On ne connaît pas à l'heure actuelle, pour les nombres de grande taille, de solution complètement satisfaisante d'un point de vue algorithmique : il existe des algorithmes dits déterministes testant la primalité d'un entier de façon certaine mais au prix d'un temps de calcul prohibitif et d'autres algorithmes dits probabilistes s'exécutant beaucoup plus rapidement mais donnant une réponse parfois inexacte avec une « probabilité d'erreur » considérée comme faible. L'objet du TP est d'implémenter les algorithmes les plus simples dans ces deux catégories. On utilisera pour simplifier les nombres entiers ordinaires du Caml bien qu'ils soient limités à $\pm 10^9$, mais l'extension de ces algorithmes aux « grands entiers » ne pose pas de problème particulier, si ce n'est le temps de calcul pour les algorithmes déterministes.

1) Méthodes déterministes

$n \in \mathbb{N}$ est premier si et seulement si $n \geqslant 2$ et s'il n'existe pas d'entier $p \in [\![2, n-2]\!]$ divisant n. Pour déterminer si n est premier, il suffit donc d'essayer les entiers successifs $p = 2, 3, 4, \ldots, n-1$ en arrêtant dès qu'on en trouve un qui divise n. p est appelé un « diviseur potentiel » dans ce type d'algorithme. La complexité de cet algorithme est de l'ordre de n divisions lorsque n est effectivement premier, moins si n est composé car on arrête dès qu'on a trouvé un diviseur.

a) Programmer cette méthode naïve. On écrira une fonction `test_naïf : int` $\longrightarrow$ `bool` qui retourne `true` quand son argument est premier et `false` sinon.

b) Une amélioration évidente consiste à arrêter la recherche lorsque p dépasse $\sqrt{n}$ car si n n'a pas de diviseur inférieur ou égal à $\sqrt{n}$ alors il n'en n'a pas non plus supérieur à $\sqrt{n}$. Programmer cette amélioration *en évitant de recalculer la racine carrée de n à chaque itération.*

c) Une autre possibilité d'amélioration consiste à éviter certains diviseurs potentiels p pour lesquels on est certain que la division ne tombera pas juste. Par exemple si on a constaté que n n'est pas divisible par 2 alors on peut sauter tous les entiers p pairs ce qui divise par 2 le temps de calcul. De même si on aussi vérifié que n n'est pas divisible par 3 alors on peut sauter tous les entiers p multiples de 3 d'où un gain en temps de calcul de 33%. En combinant ces deux remarques on voit que les seuls entiers p à tester sont de la forme $6k \pm 1$ en plus de 2 et 3. Programmer cette méthode. On peut continuer la réduction de la liste des diviseurs potentiels en retirant aussi les multiples de 5 ce qui limite p aux entiers de la forme $30k \pm a$ avec $a \in \{1, 7, 11, 13\}$. Au delà de 5 le gain en vitesse devient trop faible pour justifier une complication supplémentaire du programme.

2) Méthode de Fermat

Fermat a prouvé que si n est premier impair alors pour tout entier $a \in \left[\!\left[2, \frac{n-1}{2} \right]\!\right]$ on a

$$a^{n-1} \equiv 1 \bmod n. \tag{1}$$

Le test de Fermat consiste à choisir quelques entiers a au hasard dans l'intervalle $\left[\!\left[2, \frac{n-1}{2} \right]\!\right]$ et à vérifier (1). Si l'un au moins des a testés rend une réponse fausse alors n n'est pas premier ; si tous les a tirés passent le test alors on peut penser que a est premier.

La mise en œuvre du test de Fermat suppose qu'on puisse calculer facilement $a^{n-1} \bmod n$, en tout état de cause en moins de $\sqrt{n}$ multiplications, sinon autant prendre une méthode déterministe ! On verra en cours une méthode de calcul rapide de la puissance k-ème d'un nombre fondée sur la stratégie « diviser pour régner ». Voici en avant-première le code d'une telle fonction :

```
(* calcule x^k mod n, k >= 0 et n > 0 *)
let rec puiss_mod x k n =
  if p < 1 then 1
  else let y = puiss_mod x (k/2) n in
       let z = y*y mod n           in
       if p mod 2 = 0 then z else z*x mod n
;;
```

Recopier ce code et l'utiliser pour implémenter le test de Fermat. Le choix d'un entier aléatoire dans l'intervalle $[\![a, b[\![$ s'écrit : `a + random__int(b-a)`. Certains entiers, appelés « nombres de Carmichaël », bien que non premiers passent ce test avec succès si tous les a tirés sont premiers avec n ce qui peut arriver par « malchance ». Le plus petit nombre de Carmichaël est $561 = 3 * 11 * 17$ donc il suffit que les cinq nombres aléatoires tirés ne soient divisibles ni par 3 ni par 11 ni par 17 pour conclure à tort que 561 est premier et la probabilité de cette conclusion erronée est de l'ordre de 1.4% ce qui n'est pas négligeable.

3) Méthode de Rabin-Miller

C'est une amélioration de la méthode de Fermat. On écrit $n - 1 = p * 2^q$ avec p impair et on calcule modulo n les entiers :

$$a^p, a^{2p}, a^{4p}, \ldots, a^{p2^q}$$

où a est un entier aléatoire entre 2 et $(n-1)/2$. Remarquer que chaque entier est le carré, modulo n, du précédent. n passe le test si cette suite est constituée uniquement de 1 ou si elle se termine par $n - 1, 1, 1, \ldots, 1$. Dans les autres cas il est démontré que n n'est pas premier. On peut prouver, au moyen de calculs compliqués, que la probabilité pour que le test conclue à tort à la primalité de n est inférieure ou égale à $\frac{1}{4}$ donc en effectuant ce test pour cinq entiers a aléatoires, la probabilité d'une conclusion erronée est inférieure à 0.1%. Les logiciels de calcul formel utilisent en général cette méthode en tirant 25 valeurs aléatoires pour a et il n'existe pas, à l'heure actuelle, d'exemple d'entier non premier ayant passé avec succès ces 25 tests. Programmer cette méthode.