Autour du problème de la primalité

Epreuve orale

Sujet A Sujet B

1 Le problème de la primalité

On s'inspire du résultat de 2002 d'Agrawal, Kayal et Saxena qui décide la primalité d'un entier n en temps polynomial. Rappelons que, par définition, un entier $n > 1$ est premier s'il n'admet pas d'autre diviseur que 1 et lui même.

On rappelle ci-dessous l'énoncé du problème de la primalité :
INSTANCE : Un entier n ;
QUESTION : n est-il un nombre premier ?
Et on considère les deux codages suivants de ce langage :

$$L_{\text{Prem},1} = \{1^p : p \text{ premier}\}$$
$$L_{\text{Prem},2} = \{m \in \{0,1\}^\star : m \text{ est le codage binaire d'un entier premier}\}$$

Exercice 1.
On s'intéresse tout d'abord au langage $L_{\text{Prem},1}$ des nombres premiers en unaire.
1.1. Donnez les premiers éléments de $L_{\text{Prem},1}$. **Corrigé** : $L_{\text{Prem},1}$=$\{$1,11,111,11111,1111111,...$\}$

1.2. Montrez que $L_{\text{Prem},1}$ n'est pas rationnel. **Corrigé** : Supposons $L_{\text{Prem},1}$ rationnel. On fixe n et on considère le mot $z = 1^p$ pour p un nombre premier tel que $|p| \geq n$. Quel que soit le découpage en trois mots u, v et w on a $|u| + |w| = p - k$ et $|v| = k$. On considère alors la longueur du mot uv^iw : $|uv^iw| = (p-k) + ki = p + k(i-1)$. Pour la valeur de $i = p + 1$, on obtient le mot $uv^{p+1}w$ dont la longueur vaut $p(k+1)$ qui n'est plus un nombre premier, donc plus dans le langage ; une contradiction. On en déduit donc que $L_{\text{Prem},1}$ n'est pas rationnel.

1.3. Montrez que $L_{\text{Prem},1}$ n'est pas algébrique. **Corrigé** : Supposons $L_{\text{Prem},1}$ algébrique. On fixe n et on considère le mot $z = 1^p$ pour p un nombre premier tel que $|p| \geq n$. Quel que soit le découpage en cinq mots u, v, w, x et y on a $|uvy| = p - k$ et $|vx| = k$. On considère alors la longueur du mot uv^iwx^iy : $|uv^iwx^iy| = (p-k) + ki = p + k(i-1)$. Pour la valeur de $i = p + 1$, on obtient un mot dont la longueur vaut $p(k+1)$ qui n'est plus un nombre premier, donc plus dans le langage ; une contradiction. On en déduit donc que $L_{\text{Prem},1}$ n'est pas algébrique.

On rappelle ci-dessous l'énoncé du «lemme de la pompe» pour les langages rationnels :

Lemme 1. *Si L est un langage rationnel, il existe un entier n qui ne dépend que de L tel que pour tout mot w de L tel que $|w| \geq n$ il existe une factorisation $w = xyz$ telle que $|xy| \leq n$, $|y| > 0$ et pour tout $i \geq 0$, $xy^iz \in L$*

On rappelle ci-dessous l'énoncé du «lemme de la pompe» pour les langages algébriques :

Lemme 2. *Si L est un langage algébrique, il existe un entier n qui ne dépend que de L tel que pour tout mot $z \in L$, $|z| \geq n$, il existe une factorisation $z = uvwxy$ telle que $|vwx| \leq n$, $|vx| > 0$ et pour tout $i \geq 0$, $uv^iwx^iy \in L$*

2 Complexité des opérations arithmétiques

On rappelle dans le tableau ci-dessous la complexité des opérations arithmétiques les plus courantes pour des entiers sur N bits :

Opération	Notation	Complexité
multiplication	$M(N)$	$O(N \log N \log\log N)$
division	$D(N)$	$O(M(N))$
multiplication modulaire	$M_{\text{mod}}(N)$	$O(M(N))$

Pour la complexité asymptotique, un terme en $\log\log$ est négligeable devant un terme plus que logarithmique.

Exercice 2.
On s'intéresse tout d'abord à la complexité de l'exponentiation modulaire, autrement dit au calcul de $a^b \mod n$.
2.1. Montrez que, étant donnés a, b et n, le coût du calcul de $a^b \mod n$ est de $\log(b)$ multiplications, chacune d'entiers d'au plus $\log(n)$ bits. Indication : On utilisera le calcul des carrés successifs.
2.2. Montrez que lorsque a, b et n sont donnés, le coût du calcul de $a^b \mod n$ est en $O(\log^2 n)$. Indication : On utilisera le calcul des carrés successifs. **Corrigé** : C'est l'algo classique du calcul de l'exponentielle modulaire où on calcule $a^{\sum b_i 2^i} \mod n = \prod a^{b_i 2^i} \mod n$ où l'on effectue $\log b$ multiplication d'au plus $\log n$ bits, d'où un coût global de l'algo en $O(\log^2 n)$.

3 La complexité du problème avant 2002

Exercice 3.
Rappelons que la méthode connue depuis l'antiquité, le *crible d'Eratosthène*, permet de décider la primalité d'un entier $n \geq 3$. Il suffit de vérifier qu'il n'existe pas de diviseur entier de n parmi les entiers impairs (autres que 1) inférieurs à la racine carrée de n.
3.1. Montrez que le temps de fonctionnement du crible d'Eratosthène n'est pas une fonction polynomiale de la taille de l'entrée pour le codage $L_{\text{Prem},2}$. **Corrigé** : Si n est composé, n a un diviseur différent de 1 et de n qui est inférieur à $\sqrt{n}$. Il suffit donc de faire $\sqrt{n}$ divisions dont le temps de calcul de chacune est en $O(\log n \log\log n)$, ce qui donne une

complexité en $O(\sqrt{n}\log n \log\log n)$ où n n'est pas la taille de l'entrée (qui est de $\log n$) mais l'entrée elle-même.

3.2. Montrez que le temps de fonctionnement du crible d'Eratosthène est une fonction polynomiale de la taille de l'entrée pour le codage $L_{\text{Prem},1}$. **Corrigé** : Dans ce cas, la taille de l'entrée est n et le temps de calcul du crible d'Eratosthène est toujours en $O(\sqrt{n}\log n \log\log n)$ où n est dorénavant la taille de l'entrée.

Exercice 4.
Le problème obtenu par la négation de la question d'un problème de décision Π est appelé le *complémentaire* de Π.

4.1. Définissez le problème complémentaire du problème de la primalité, appelé *problème de la composition*. **Corrigé** :

INSTANCE : Un entier n ;
QUESTION : n est-il un nombre composé ?

4.2. Donnez une instance positive et une instance négative du problème de la primalité. **Corrigé** : 7 est un nombre premier et 16 un nombre composé. En considérant leur écriture binaire, $111 \in L_{\text{Prem},2}$ est une instance positive de ce problème tandisque $10000 \notin L_{\text{Prem},2}$ et constitue une instance négative au problème de la primalité.

Avant que Agrawal, Kayal et Saxena proposent un algorithme de test de primalité en temps polynomial, Pratt avait montré le théorème suivant :

Théorème 1 (Pratt, 1975). *Le problème de la primalité est dans NP∩co-NP.*

Rappelons qu'un problème est dans la classe NP∩co-NP si toute instance positive de ce problème admet un certificat de taille polynomiale et si toute instance négative de ce problème admet un certificat de taille polynomiale. Il faut de plus qu'il existe un algorithme déterministe qui, sur l'entrée de l'entier n et d'un des certificats, décide en temps polynomial la primalité ou la composition (le fait que n est un nombre *composé*) de n.

Exercice 5.
On s'intéresse ici à la taille des certificats utilisés par l'algorithme de vérification.

5.1. Donnez un certificat de non-primalité de 12.

5.2. Donnez un certificat de composition de 12. **Corrigé** : Un certificat de composition de 12 est p.e. 2 ou 10 en binaire qui est un diviseur de 12.

5.3. Montrez qu'un certificat de non-primalité est polynomial en la taille de l'entrée (i.e. que si n n'est pas premier, il existe C_n un certificat de non-primalité dont la taille est majorée par $\lceil \log_2(n) \rceil^k$ pour $k \geq 1$). **Corrigé** : Si n est composé, il possède un diviseur autre que 1 et n qui est inférieur à $\sqrt{n}$. La taille de ce diviseur est donc majorée par $\log n$, donc polynomiale (et même linéaire) en la taille de l'entrée.

Il est alors facile de construire un algorithme déterministe de vérification qui travaille en temps polynomial. On en déduit que le problème de la primalité est dans co-NP. Un

certificat de non-primalité consiste en la donnée d'un entier qui est un diviseur de n, dont la taille est majorée par $\log n$.

5.4. Donnez le fonctionnement d'un algorithme déterministe de vérification de la non-primalité de n et estimez-en la complexité. **Corrigé** : Sur l'entré de n et de C_n, l'algorithme vérifie tout d'abord la conformité syntaxique puis que $C_n|n$ en un temps $O(M(\log n))$.

Exhiber un certificat de primalité succint est une affaire plus difficile et repose sur un corollaire du petit Théorème de Fermat (1640) que l'on admettra :

Théorème 2. *Un entier $n > 1$ est premier ssi il existe un élément $1 < a < n$ tel que $a^{(n-1)} \equiv 1 \mod n$ et $a^{(n-1)/q} \not\equiv 1 \mod n$ pour tout q premier diviseur de $n-1$.*

Ainsi, un certificat de primalité de n va contenir a et une factorisation de $(n-1)$ en produit de facteurs premiers, $(n-1) = q_1^{e_1}.....q_k^{e_k}$. Cette factorisation de $(n-1)$ va permettre de tester la condition $a^{(n-1)/q} \not\equiv 1 \mod n$ pour tout q premier diviseur de $(n-1)$.

5.5. Montrez que la deuxième condition est nécessaire ; on peut en effet trouver un entier n non premier qui vérifie que $a^{(n-1)} \equiv 1 \mod n$, par exemple lorsque a est un carré modulo n.

5.6. Montrez que la deuxième condition est nécessaire ; on peut en effet trouver un entier n non premier qui vérifie que $a^{(n-1)} \equiv 1 \mod n$, par exemple lorsque $a \equiv -1 \mod n$.

Il nous faut donc aussi donner un certificat de la primalité des q_i et notre certificat de primalité de n va donc être récursif.

Soit P_n le certificat de primalité de l'entier n ; par définition $P_2 = \langle\langle 1, 1\rangle, \varepsilon\rangle$ et pour $n \geq 3$, on a

$$P_n = \langle\langle a, q_1, ^{e_1}, q_2, ^{e_2}, \ldots, q_k, ^{e_k}\rangle, (P_{q_1}, P_{q_2}, \ldots, P_{q_k})\rangle$$

pour a un générateur[1] de $\{1, \ldots, n-1\}$ et $(n-1) = q_1^{e_1}.....q_k^{e_k}$.

5.7. Donnez un certificat de primalité de 13. **Corrigé** : $P_{13} = \langle\langle 3, 2, ^2, 3, ^1\rangle, (P_2, P_3)\rangle$, $P_3 = \langle\langle 2, 2, ^1\rangle(P_2)\rangle$.

5.8. Montrez par une minoration grossière que le nombre de facteurs premiers de $(n-1)$ est majoré par $\log n$, i.e. si $(n-1) = \prod_{i=1}^{k} q_i^{e_i}$, alors $k < \log n$. **Corrigé** : $n-1 = \prod_{i=1}^{k} q_i^{e_i}$ et chaque $q_i \geq 2$. On a donc $n-1 \geq \prod_{i=1}^{k} 2^{e_i}$ et comme $e_i \geq 1$, $n > \prod_{i=1}^{k} 2^{e_i} = 2^{\sum_{i=1}^{k} 1} = 2^k$; d'où $k < \log n$.

5.9. Montrez (par récurrence sur n) que P_n peut être codé au moyen de $O(\log^2 n)$ bits. On pourra utiliser le fait que le nombre de facteurs premiers de $(n-1)$ est majoré par $\log n$. **Corrigé** : Par récurrence sur n ; c'est vrai pour la base. On a de plus les majorations suivantes : $k < \log n$, $q_i \geq 2$, $\sum e_i < \log n$. Il faut $\log n$ bits pour coder a, on a k q_i et besoin de $\log \frac{3n-1}{2}$ pour les coder et k e_i à coder en $O(\log n)$ bits. Comme $n-1$ est pair,

[1] On dit que a est un *générateur* de $\mathbb{Z}_n^\star = \{1, \ldots, n-1\}$ si les puissances successives de a modulo n permettent de retrouver tous les ´el´ements de $\mathbb{Z}_n^\star$.

il faut aussi un nombre constant de bits pour coder le certificat de 2, P_2. Pour les P_{q_i} on applique HR, d'où $|P_{q_i}| \leq \log^2 q_i$. Soit en tout

$$|P_n| \leq O(\log n) + cte + O(\sum_{i=2}^{k} \log^2 q_i) \tag{1}$$

Il s'agit maintenant de majorer $\sum_{i=2}^{k} \log^2 q_i$. On a $\sum_{i=2}^{k} \log q_i \leq \log \prod_{i=1}^{k} q_i - \log 2 \leq \log(n-1) - 1 < \log n - 1$. De plus, $\sum_{i=2}^{k} \log^2 q_i < (\log n - 1)^2 = \log^2 n + 1 - 2\log n$. On peut donc réécrire l'équation (**??**) comme $|P_n| \leq O(\log n) + cte + O(\log^2 n)$ qui se comportera comme $O(\log^2 n)$ pour n assez grand. Il est alors relativement facile d'en déduire un algorithme déterministe de vérification de la primalité de n à l'aide des certificats par un algorithme déterministe qui travaille en temps $O(\log^4 n)$

5.10. Montrez que la vérification de la primalité de n à l'aide des certificats peut être réalisée en temps polynomial par un algorithme déterministe. **Corrigé** : Pour vérifier P_n, il faut : (1) calculer $a^{n-1} \mod n$ en $O(log^2 n)$, (2) calculer $(n-1)/q_i$ en $\log n.O(M(\log n))$, (3) calculer $a^{\frac{n-1}{q_i}} \mod n$ en $\log n.O(\log^2 n)$ et calculer $\prod q_i^{e_i}$ les opération (2) et (3) à répéter $\log n$ fois, donc un coût de l'ordre de $O(\log^4 n)$.

4 Un algorithme polynomial de test de primalité

En s'appuyant sur l'identité (1) (un dérivé du petit théorème de Fermat) Agrawal, Kayal et Saxena ont pu construire un *algorithme déterministe de test de primalité*.

$$\text{si pgcd}(a,n) = 1, \qquad [(x-a)^n \equiv (x^n - a) \mod n \Leftrightarrow n \text{ premier}] \tag{2}$$

Exercice 6.

6.1. Prouvez que si n est premier, n divise les coefficients binomiaux $\binom{n}{i}$ pour $i \neq 1, n$. **Corrigé** : n premier ; $\binom{n}{i} = \frac{n!}{i!(n-i)!} = \frac{n(n-1)!}{i!(n-i)!}$ et $n | \frac{n(n-1)!}{i!(n-i)!}$.

En revanche, si n est composé, il existe q premier qui divise n. Soit q^k la plus grande puissance de q tel que q^k divise n. q^k ne divise pas $\binom{n}{q}$ Il est assez facile de voir que si n est premier, n divise les coefficients binomiaux $\binom{n}{i}$ pour $i \neq 1, n$.

6.2. Si n est composé, il existe q premier diviseur de n. Soit q^k la plus grande puissance de q qui divise n. Montrez que q^k ne divise pas $\binom{n}{q}$. **Corrigé** : $n = q^k \alpha$ et q ne divise pas α ; $\binom{n}{q} = \frac{n!}{q!(n-q)!} = \frac{n(n-1)!}{q(q-1)!(n-q)!}$; Il s'ensuit que, si $q^k | \binom{n}{q}, q^{k+1} | n$ contredisant q ne divise pas α ;

6.3. Déduisez-en l'identité (1). **Corrigé** : De la question 1, on déduit que si n premier, $\binom{n}{i} \equiv 0 \mod n$ et que tous les coefficients du développement sont nuls excepté le premier et le dernier et de la question 2 que si n est composé, les coefficients que x^q dans le développement ne sont pas nuls, donc que $(x-a)^n - x^n + a$ n'est pas nul modulo n.

Dès lors que l'on dispose de l'identité (1), étant donné un entier n dont on veut décider la primalité, il suffit de choisir $P(x) = (x - a)$ et de vérifier si l'identité (1) est satisfaite. Cependant, cette technique est coûteuse en temps dans le pire des cas.
L'idée qui permet d'améliorer cette complexité est d'évaluer la congruence modulo un polynôme $x^r - 1$ pour un premier r bien choisi[2]. Une itération de l'algorithme sera alors

$$(x - a)^n \equiv (x^n - a) \mod (x^r - 1, n)$$

On vérifie cette nouvelle congruence pour un petit nombre de valeurs de a (de l'ordre de $O(\sqrt{r} \log n)$).
On dispose alors de l'algorithme suivant dont on se propose d'étudier la ligne 1.
Algorithme 1.

Entrée : un entier $n > 1$
1. **Si** n est de la forme $a^b, b > 1$ **alors Retourne** (n est `COMPOSE`) **fsi**
2. Trouver le plus petit r tel que l'ordre de n dans $\mathbb{Z}_r^\star$ $o_r(n) > 4\lceil \log^2 n \rceil$
   ($o_r(n)$ est le plus petit k t.q. $n^k \equiv 1 \mod r$)
3. **Pour** $a \leftarrow 1$ **jusqu'à** $2\lceil \sqrt{r} \log n \rceil$ **faire**
4. **Si** $(x - a)^n \not\equiv (x^n - a) \mod (x^r - 1, n)$ **alors Retourne** (n est `COMPOSE`) **fsi**
   fpour
   Retourne (n est `PREMIER`)

Exercice 7.
Il n'est pas évident de décider en temps polynomial si n est une puissance pure (i.e. de la forme a^b pour $b > 1$ et a non fixé). On déduit facilement de l'algorithme d'exponentiation modulaire un algorithme de calcul de a^b en $O(\log n)$ lorsque a et b sont fixés.
Corrigé : si a et b sont donnés, le coût du calcul de a^b par la méthode des carrés est de $\log(b)$ multiplications, chacune d'entiers d'au plus $\log(n)$ bits. En utilisant une mutiplication par transformée de Fourier rapide, chaque multiplication coûte $\log(n)\log(\log(n))$ opérations. Comme $b \leq n$, on a $\log(b) \leq \log(\log(n))$ donc le coût du calcul de a^b est $\log(n)\log(\log(n))^2$. Il est classique de "jeter" les termes $\log(\log)$ pour la complexité asymptotique, donc on obtient un coût de $O(\log(n))$ pour vérifier avec a et b donnés ;

7.1. Montrez que si a n'est pas donné, il est possible de retrouver sa valeur $O(\log^2 n)$.
Corrigé : On suppose donc b fixé. On calcule a^b en cherchant la valeur de a pour $2 \leq a \leq n$ par dichotomie , il n'y a que $\log(n)$ valeurs de a à essayer, chacune coûte $O(\log(n))$, donc on obtient un coût de $O(\log(n)^2)$ pour chercher avec seulement b donné ;.

[2]Un th´eor`eme de th´eorie des nombres garantit qu'il existe un tel r de l'ordre de $O(\log^6 n)$

7.2. Donnez une majoration de b en fonction de $\log(n)$ et déduisez-en le nombre de valeurs de b à tester si b n'est plus fourni en entrée. **Corrigé** : comme $b \leq \log(n)$, il n'y a que $\log(n)$ valeurs de b à essayer, on obtient bien un coût de $O(\log(n)^3)$ pour vérifier que n n'est pas une puissance pure.

7.3. Déduisez-en le coût d'un algorithme pour vérifier que n n'est pas une puissance pure. **Corrigé** : Par ce qui précède, on a un coût total de $O(\log(n)^3)$ pour vérifier que n n'est pas une puissance pure.