Les deux parties sont complètement indépendantes.

Partie I - Autour de la suite de Fibonacci

L'objectif de ce problème est de comparer les complexités temporelles (et dans une moindre mesure spatiale) d'algorithmes de calcul numérique de suites entières vérifiant une relation de récurrence linéaire. Les entiers manipulés pourront être arbitrairement longs (ce qui n'est *a priori* le cas ni en Caml ni en Pascal). Dans les calculs de complexité, on demande des ordres de grandeur du nombre d'opérations élémentaires sur les bits.

Exemple 1 : l'algorithme naturel pour sommer deux entiers de n bits demande $O(n)$ opérations élémentaires (additions bit-à-bit, propagation de retenue).

On ne demande pas d'équivalents, mais des majorants asymptotiques.

Lorsque $u_n = O(v_n)$ et $v_n = O(u_n)$, on pourra noter $u_n = \Theta(v_n)$, ce qui est plus précis que $u_n = O(v_n)$.
Exemple 2 : Le "tri-bulle" d'un tableau de n valeurs effectue $\Theta(n^2)$ comparaisons de valeurs, et réalise $O(n^2)$ échanges dans le tableau.

Les candidats rédigeant en Caml devront donner le type de chaque fonction écrite.

Notation : a $[b]$ désignera a modulo b, c'est-à-dire le reste dans la division euclidienne de a par b.

I.A - Questions préliminaires

I.A.1) Évaluer le nombre d'opérations élémentaires sur des bits requises pour effectuer le produit de deux entiers de n bits avec une méthode élémentaire (que l'on décrira sommairement).

I.A.2) Donner un algorithme effectuant une multiplication d'entiers avec une meilleure complexité. Le décrire, et donner (sans justification) sa complexité temporelle.

I.A.3) Décrire très sommairement l'algorithme d'exponentiation rapide. Justifier son intérêt par rapport à un algorithme élémentaire.

I.B - Diverses façons de calculer f_n

La suite de Fibonacci est définie par les relations $f_0 = 0$, $f_1 = 1$, et pour tout $n \in \mathbb{N}$: $f_{n+2} = f_n + f_{n+1}$.

I.B.1) En utilisant les résultats standards sur les suites vérifiant une relation de récurrence linéaire d'ordre 2, à coefficients constants, donner la valeur de f_n, et l'ordre de grandeur de sa représentation binaire (écriture en base 2). En déduire un *minorant* pour le temps de calcul de *tout* programme calculant f_n.

I.B.2) Écrire une fonction récursive `fibo` prenant en entrée un entier n et retournant f_n en appliquant directement la définition de f.

I.B.3) Prouver que le nombre d'appels récursifs effectués pour calculer f_n avec la fonction précédente est exponentiel en n.

I.B.4) Écrire une nouvelle fonction `fibo2` réalisant le calcul de f_n de façon itérative, en calculant de proche en proche les f_k pour $k \leqslant n$. Donner le nombre d'additions d'entiers qui seront effectuées lors du calcul de f_n, et évaluer la dépendance du temps de calcul par rapport à n, ainsi que la place en mémoire requise (on supposera que les entiers calculés restent inférieurs au plus grand entier représentable en Caml ou en Pascal).

I.B.5) On observe qu'en notant $X_n = \begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix}$ et $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, on a $X_{n+1} = AX_n$ pour tout $n \in \mathbb{N}$, puis : $X_n = A^n X_0$. Estimer le temps de calcul et la place en mémoire requise pour calculer f_n en exploitant cette relation.

I.B.6) Si on cherche à calculer f_n modulo un entier k "petit" (dans un sens à préciser par le candidat), quelle méthode peut-on utiliser, et quels sont le temps et l'espace requis par cette méthode ?

I.C - Utilisation d'automates

I.C.1) Vérifier que pour tout $n \geqslant 1$, $A^n = \begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix}$.

I.C.2) En déduire des expressions simples de f_{2n}, f_{2n+1} et f_{2n+2} en fonction de f_n et f_{n+1}.

I.C.3) Construire et représenter un automate fini déterministe d'ensemble d'états Q et une fonction $\varphi : Q \to \mathbb{Z}/2\mathbb{Z}$ tels qu'en lisant depuis l'état initial la représentation binaire d'un entier n depuis le bit de poids fort jusqu'au bit de poids faible, on arrive

- version du 30 avril 2007 20h1

dans un état q tel que $\varphi(q) = f_n$ [2].

I.C.4) À l'aide de l'automate précédent, donner la valeur de f_{2050} [2].

I.C.5) Prouver que la suite f' définie par $f'_n = f_n$ [2] est 3-périodique et retrouver le résultat de la question précédente.

I.C.6) Construire et représenter un automate permettant de déterminer le reste modulo 3 d'un entier n, en lisant la représentation binaire de n, du bit de poids fort vers le bit de poids faible. Comparer avec l'automate de la question I.C.3??

I.D - Une généralisation

On considère maintenant, pour $m \geqslant 3$ fixé, la suite g de premiers termes $g_0 = 0$ et $g_1 = g_2 = \cdots = g_{m-1} = 1$, et vérifiant la relation de récurrence : $g_{n+m} = g_n + g_{n+m-1}$ pour tout $n \in \mathbb{N}$ (g_{n+m} est la somme des deux nombres g_n et g_{n+m-1}).

I.D.1) Écrire une fonction `g` prenant en entrée m et n, et retournant g_n en appliquant simplement la relation définissant g. Donner un ordre de grandeur du temps d'exécution (on ne demande pas une preuve formelle du résultat).

I.D.2) On se place *pour cette question seulement* dans le cas $m = 3$. Écrire une fonction `g3` prenant en entrée n et retournant g_n.
On écrira un programme itératif calculant les différents termes de proche en proche, en ne stockant que "quelques" termes de la suite.

I.D.3) Écrire maintenant une fonction itérative `g` prenant en entrée m et n, et retournant g_n en faisant $O(n)$ additions.
On utilisera un tableau pour stocker des valeurs successives de g.

I.D.4) Si ce n'est pas déjà le cas dans la question précédente, écrire une fonction calculant g_n en $O\big(\max(n, m)\big)$ additions et affectations de variables (y compris dans des tableaux). Cette fonction utilisera un tableau dont la taille est de l'ordre de m.

I.D.5) Proposer une façon raisonnable de calculer $g_{10^{20}}$ modulo 3, avec $m = 1000$.

Par "raisonnable" on entend : retournant le résultat en moins d'une journée de calcul sur un ordinateur de bureau de puissance et de capacité de stockage moyens en 2007 !

Partie II - Un calcul de ppcm

L'objectif de cette partie est d'analyser un algorithme permettant de calculer, pour $n \in \mathbb{N}$, le plus petit multiple commun de tous les entiers $\leqslant n$.

Il s'agit de $P_n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$, avec $p_1, p_2, ..., p_k$ la suite (strictement croissante) des nombres premiers compris au sens large entre 2 et n et pour chaque i, α_i est l'unique entier tel que $p^{\alpha_i} \leqslant n < p^{\alpha_{i+1}}$. Par exemple, $P_9 = 2^3.3^2.5.7$.

Plus précisément, on souhaite calculer tous les P_k, pour $k \in [\![1, n]\!]$, en exploitant au maximum les calculs précédents. Pour cela, on va utiliser une structure de tas.

Un tas est un arbre binaire dont les nœuds sont étiquetés par des éléments distincts d'un ensemble complètement ordonné. Chaque nœud possède une étiquette strictement plus *petite* que les étiquettes de ses éventuels fils. Les adjonctions et éventuelles suppressions de nœuds doivent préserver cette propriété, ainsi que le fait que « *tous les niveaux de l'arbre sont remplis, sauf éventuellement celui de profondeur h (hauteur de l'arbre), qui est lui-même rempli de la gauche vers la droite* ». Par exemple, un tas possédant six nœuds sera constitué d'une racine, deux nœuds de profondeur 1, et 3 nœuds de profondeur 2. La structure de données utilisée en machine pour stocker et manipuler ces tas n'importe pas ici : on ne demande pas de programmer effectivement les algorithmes.

Ici, les nœuds sont étiquetés par des couples (p^α, p), avec p premier. Après le calcul de P_k, sont stockés dans le tas tous les couples d'entiers (p^α, p) tels que p est un nombre premier inférieur ou égal à k et α est le plus petit entier tel que $k < p^\alpha$. Par exemple, après avoir calculé P_9, on trouve dans le tas les couples $(16, 2)$, $(27, 3)$, $(25, 5)$ et $(49, 7)$.

Les couples sont ordonnés de la façon suivante : $(a, b) <_1 (a', b')$ si et seulement si $a < a'$. Pour cet algorithme, le tas ne contient jamais deux couples ayant la même première composante, de sorte que les étiquettes sont toujours comparables.

L'algorithme est itératif. On stocke dans une variable `Res` le ppcm calculé jusque là. Au départ, `Res=2` est le ppcm des entiers $\leqslant 2$ et le tas est constitué d'un seul nœud indexé par $(4, 2)$. Après avoir calculé P_{k-1} (qui est alors présent dans `Res`), on calcule P_k de la façon suivante :

- Si k est premier, on multiplie `Res` par k, et on insère un nouveau nœud indexé par (k^2, k) dans le tas.
- Sinon, si la racine (p^α, p) est telle que $k = p^\alpha$, alors on multiplie `Res` par p, on change la racine en $(p^{\alpha+1}, p)$ et on reconstitue la structure de tas.

II.A - Comment peut-on insérer un nouveau nœud dans un tas (en préservant la structure de tas) ?

II.B - Comment reconstituer la structure de tas après avoir changé la valeur de la racine ? Une telle opération sera appelée *percolation* dans la suite.

II.C - Représenter les différentes valeurs du tas après le calcul de P_k, pour k allant de 3 jusqu'à 10, puis la valeur du tas pour $k = 16$.

II.D - Montrer que pour un tas de hauteur h constitué de n nœuds, on a $h = \Theta(\ln n)$.

II.E - Quel est le coût d'une percolation ? Montrer que le nombre de percolations effectuées lors du calcul de P_n est négligeable devant n.

II.F - Proposer un algorithme élémentaire permettant de déterminer si un entier est premier. Évaluer grossièrement son temps d'exécution. Existe-t-il des algorithmes plus efficaces ?

II.G - On peut prouver que $\ln P_n \sim n$. Évaluer grossièrement en fonction de n le temps nécessaire pour calculer P_n en utilisant l'algorithme présenté dans ce problème.

• • • FIN • • •
