Concours Centrale-Supélec 2007

Corrigé de l'épreuve d'informatique

Partie I - Autour de la suite de Fibonacci

I.A.1) Utilisons la méthode de multiplication apprise à l'école élémentaire. Ainsi, le calcul du produit de $a = 1001101$ par $b = 1101$ (en base 2) se fait de la façon suivante :

$$
\begin{array}{cccccccccc}
 & & & & & 1 & 0 & 1 & 0 & 1 \\
 & & & & \times & 1 & 1 & 1 & 0 & 1 \\
\hline
 & & & & & 1 & 0 & 1 & 0 & 1 \\
 & & & 1 & 0 & 1 & 0 & 1 & \cdot & \cdot \\
 & & 1 & 0 & 1 & 0 & 1 & \cdot & \cdot & \cdot \\
 & 1 & 0 & 1 & 0 & 1 & \cdot & \cdot & \cdot & \cdot \\
\hline
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Si on note i le nombre de bits non nul de b, le calcul de ab peut donc se faire en effectuant $i-1$ additions, les entiers calculés ne dépassant par $2n$ bits. Comme chaque entier correspondant à une nouvelle ligne se calcule en temps linéaire, le temps de calcul total sera de l'ordre de $(i-1)O(n)$, qui est un $O(n^2)$.

I.A.2) On peut penser à l'algorithme de Karatsuba, mais les candidats ne sont pas sensés le connaître (et encore moins avoir retenu les formules) : pour multiplier deux entiers a et b de n bits, on les coupe en tronçons de $m = \lceil n/2 \rceil$ bits : $a = a_1\, 2^m + a_2$ et $b = b_1\, 2^m + b_2$ avec a_1, a_2, b_1, b_2 codés sur m bits. On calcule alors (récursivement) les trois produits :

$$x = (a_1 + a_2)(b_1 + b_2),\ \ y = a_1 b_1 \text{ et } z = a_2 b_2$$

et on obtient

$$ab = y\, 2^{2m} + (x - y - z)\, 2^m + z$$

ce qui permet d'obtenir ab en effectuant :

- le découpage de a et b en a_1, a_2, b_1, b_2 (temps en $\Theta(n)$) ;
- les additions $a_1 + a_2$ et $b_1 + b_2$ (temps en $\Theta(n)$) ;
- 3 multiplication d'entiers de taille m (dans le pire des cas) ;
- 4 additions d'entiers de taille au plus $2n$ et deux décalages (de m et $2m$ bits) pour obtenir ab (temps en $\Theta(n)$).

Ceci conduit à un temps de calcul décrit par la récurrence :

$$T(n) = 3T(\lceil n/2 \rceil) + \Theta(n)$$

qui donne $T(n) = \Theta(n^\alpha)$ avec $\alpha = \ln_2 3 < 2$.

I.A.3) Le calcul de a^b (où a et b sont ici deux entiers naturels) par exponentiation rapide se fait par induction :

- si $b = 0$, $a^b = 1$;
- si b est pair non nul, avec $b = 2b'$, $a^b = \left(a^{b'}\right)^2$;
- si b est impair, avec $b = 1 + 2b'$, $a^b = a\left(a^{b'}\right)^2$.

En notant $T(b)$ le nombre de produits nécessaires au calcul de a^b, nous obtenons la récurrence :

$$\begin{cases} T(0) = 0 \\ \forall\, b \geq 1,\ T(2b) = 1 + T(b) \\ \forall\, b \geq 0,\ T(2b+1) = 2 + T(b) \end{cases}$$

Cette récurrence du type $T(b) = T(\lfloor b/2 \rfloor) + \Theta(1)$ est bien connue et donne $T(b) = \Theta(\ln(b))$. Ainsi, l'exponentiation rapide permet de calculer a^b en effectuant de l'ordre de $\ln b$ multiplication, contre $b-1$ multiplication pour la méthode naïve ($a^b = aa^{b-1}$ si $b \geq 1$).

I.B.1) En notant $\Phi = \dfrac{1+\sqrt{5}}{2}$ et $\overline{\Phi} = \dfrac{1-\sqrt{5}}{2}$ les deux racines de l'équation caractéristique $X^2 - X - 1 = 0$, nous savons qu'il existe deux constantes A et B telles que $f_n = A\Phi^n + B\overline{\Phi}^n$ pour tout $n \geq 0$. Les conditions $f_0 = 0$ et $f_1 = 1$ donnent ensuite les valeurs de A et B, d'où :

$$\forall\, n \in \mathbb{N},\ f_n = \frac{\sqrt{5}}{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right].$$

Le nombre de bits de f_n étant équivalent à $\ln_2(f_n)$, il est équivalent (calcul élémentaire) à $n \ln_2 \Phi$. Tout programme calculant f_n aura donc un temps de calcul au moins linéaire, puisqu'il faut déjà un temps linéaire pour stocker les bits de f_n.

I.B.2)

```
let rec fibo n = if n<2 then n else fibo(n-1)+fibo(n-2);;
```

I.B.3) Notons $T(n)$ le nombre d'appel réalisé par la fonction `fibo` quand on l'applique à l'entier n. Nous avons : $T(0) = T(1) = 0$ et $T(n+2) = 2 + T(n+1) + T(n)$ pour $n \geq 0$. La récurrence $y_{n+2} = 2 + y_{n+1} + y_n$ a pour solution générale :

$$y_n = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n - 2$$

(la suite constante égale à -2 est une solution particulière). Nous obtenons alors facilement :

$$\forall\, n \in \mathbb{N},\ T(n) = \left(1 + \frac{\sqrt{5}}{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(1 - \frac{\sqrt{5}}{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^n - 2$$

et $T(n)$ est équivalent à $\left(1 + \dfrac{\sqrt{5}}{5}\right)\Phi^n$ quand n tend vers l'infini ($1 < \Phi$ et $-1 < \overline{\Phi} < 0$). Le nombre d'appel est donc exponentiel en n.

I.B.4) Il suffit de calculer f_n de proche en proche, en ne retenant que les deux dernières valeurs calculées. Cela donne :

```
let fibo2 n = match n with
     0 -> 0
   | _ -> let a=ref 0 and b = ref 1 in
          for k=2 to n do
             let c=(!b) in
             b:=(!a)+(!b);
             a:=c
          done;
          !b;;
```

Dès que $n = 46$, f_n sort des limites du type entier de Caml. Il est donc maladroit de faire une étude asymptotique de la complexité en temps et en place dans le cadre du type `int`. Nous ferons donc cette étude en supposant que des opérations permettant de manipuler des entiers arbitrairement longs ont été implémentées. La complexité spatiale est évidente : l'espace mémoire utilisé est en $\Theta(n)$, puisque l'on stocke uniquement 3 entiers de longueur $\Theta(k)$ tout au long du calcul. La complexité temporelle est à peine plus délicate : pour $n > 1$, le passage dans la boucle à l'étape k demande un temps $\Theta(k)$ et la complexité totale est $\sum_{k=2}^{n} \Theta(k) = \Theta(n^2)$.

En travaillant sur des entiers longs, l'algorithme `fibo2` a donc une complexité spatiale linéaire et une complexité temporelle quadratique.

I.B.5) Pour améliorer la complexité temporelle, on calcule A^{n-1} par exponentiation rapide, en effectuant $\Theta(\ln n)$ additions d'entiers de longueurs $O(n)$. On obtient ainsi f_n en un temps $O(n \ln n)$. On pourrait bien sûr reprendre plus précisément l'analyse du temps de calcul de cette exponentiation, en tenant compte du fait que A^k contient des entiers de longueurs $\Theta(k)$, et donner un équivalent du temps de calcul.

La complexité en place est une nouvelle fois linéaire (à condition de programmer l'exponention rapide itérativement ou avec une récursivité terminale).

I.B.6) Si on reprend la méthode précédente, mais en faisant tous les calculs modulo k, on obtient un algorithme qui répond à la question en un temps logarithmique et à complexité spatiale constante. Plus précisément, le temps est un $\Theta(k \ln n)$ et l'espace mémoire utilisé est un $\Theta(k)$. L'énoncé laisse entendre que l'on peut encore améliorer le temps de calcul, peut-être au détriment de la complexité spatiale. La seule idée qui me vient est de remarquer qu'en notant g_n la valeur de f_n modulo k, la suite (g_n) est périodique à partir d'un certain rang. En effet, les couples (g_n, g_{n+1}) sont contenus dans l'ensemble fini $\{0, 1, \dots, k-1\}^2$. Il existe donc deux entiers n_1 et n_2 tels que $0 \leq n_1 < n_2$, $g_{n_1} = g_{n_2}$ et $g_{n_1+1} = g_{n_2+1}$. On a alors trivialement $g_i = g_{i+n_2-n_1}$ pour tout $i \geq n_1$. Une méthode possible est donc la suivante :

- on calcule (en un temps ne dépendant que de k) un couple n_1, n_2 minimal ainsi que le vecteur $G = [|g_0; g_1; \dots; g_{n_2-1}|]$ et on note p la période $n_2 - n_1$;
- pour tout n entier au plus égal à $n_2 - 1$, on lit directement g_n dans G ;
- pour tout entier $n \geq n_2$, on calcule en temps constant l'unique entier i égal à n modulo p et tel que $n_1 \leq i < n_2$: on a $g_n = g_i$.

Plus précisément, on utilise une matrice A de taille $k \times k$, dont les entrées contiennent -1 au début du calcul, et un vecteur d'entiers G de taille $k^2 + 1$, tel que $G.(0) = 0$ et $G.(1) = 1$ au début du calcul. On utilise ensuite un compteur n initialisé à la valeur 0 puis on effectue la boucle :

```
while A.(G.(!n)).(G.(!n+1))=-1 do
   A.(G.(!n)).(G.(!n+1)) <- !n ;
   G.(!n+2) <- G.(!n)+G.(!n+1) mod k ;
   n := !n+1 ;
done;
```

À la sortie de cette boucle, $n_1 = \texttt{A.(G.(!n), G.(!n + 1))}$ et $n_2 = \texttt{!n}$ sont les deux entiers cherchés.
Ainsi, avec un temps de calcul initial de l'ordre de $\Theta(k^2)$ et en utilisant un espace mémoire en $\Theta(k^2)$, il est possible de calculer en quelques opérations élémentaires chaque valeurs g_n.

Il faut évidemment que k soit assez petit pour que le temps en $\Theta(k^2)$ soit négligeable devant le temps en $k \ln n$ obtenu à la question précédente. Par exemple, avec $k = 6$, les calculs sont intéressants à condition d'avoir n grand par rapport à 10^{11}. Il semble donc que $k = 6$ soit déjà une valeur trop grande !

I.C.1) Cela se prouve par récurrence immédiate sur n :

- $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} f_0 & f_1 \\ f_1 & f_2 \end{pmatrix}$;
- soit $n \geq 1$ et supposons l'égalité vérifiée au rang n. Alors :

$$A^{n+1} = \begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} f_n & f_{n-1} + f_n \\ f_{n+1} & f_n + f_{n+1} \end{pmatrix} = \begin{pmatrix} f_n & f_{n+1} \\ f_{n+1} & f_{n+2} \end{pmatrix}$$

I.C.2) Nous avons ensuite, pour $n \geq 1$:

$$\begin{pmatrix} f_{2n} & f_{2n+1} \\ f_{2n+1} & f_{2n+2} \end{pmatrix} = A^{2n+1} = A^{n+1}A^n = \begin{pmatrix} f_n & f_{n+1} \\ f_{n+1} & f_{n+2} \end{pmatrix} \begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix}$$

donc

$$\begin{array}{rcl} f_{2n} & = & f_n(f_{n-1} + f_{n+1}) = f_n(2f_{n+1} - f_n) \\ f_{2n+1} & = & f_n^2 + f_{n+1}^2 \\ f_{2n+2} & = & f_{n+1}(f_n + f_{n+2}) = f_{n+1}(2f_n + f_{n+1}) \end{array}$$

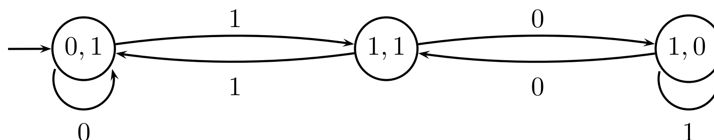
I.C.3) En particulier, nous obtenons grâce aux formules précédentes :

$$\begin{array}{rcl} f_{2n}[2] & = & f_n[2] \\ f_{2n+1}[2] & = & f_n[2] + f_{n+1}[2] \bmod 2 \\ f_{2n+2}[2] & = & f_{n+1}[2] \end{array}$$

Nous allons construire un automate déterministe sur l'ensemble d'états $Q = \{(0,0), (0,1), (1,0), (1,1)\}$ tel que l'état atteint en lisant le code binaire d'une entier n soit le couple $(f_n[2], f_{n+1}[2])$. Les relations précédentes donnent la fonction de transition :

q	(0,0)	(0,1)	(1,0)	(1,1)
$q.0$	(0,0)	(0,1)	(1,1)	(1,0)
$q.1$	(0,0)	(1,1)	(1,0)	(0,1)

L'état initial est $(f_0[2], f_1[2]) = (0,1)$ et l'état $(0,0)$ n'est pas accessible, ce qui donne l'automate :



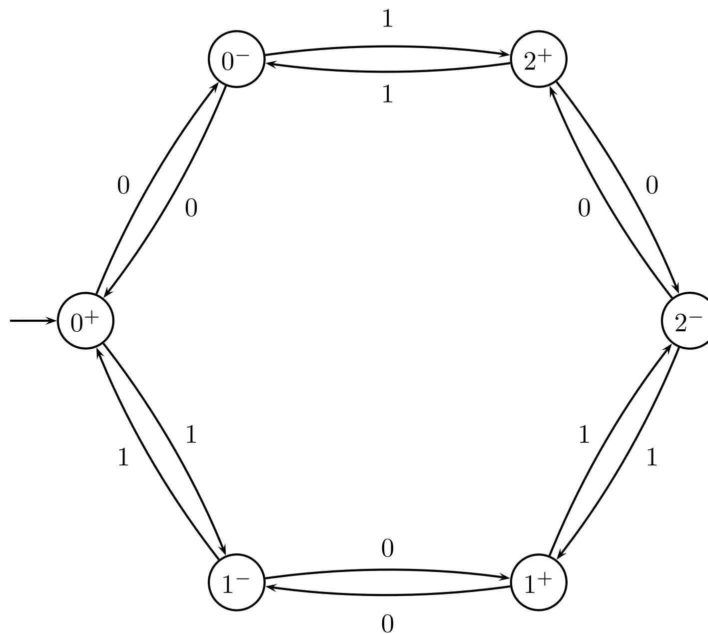
qui répond à la question posée, avec $\varphi(0,1) = 0$, $\varphi(1,1) = 1$ et $\varphi(1,0) = 1$.

I.C.4) Comme $2050 = 2^{11} + 2$, l'écriture binaire de 2050 est 100000000010, la lecture de ce mot par l'automate conduit à l'état $(1,0)$ et $f_{2050}[2] = 1$.

I.C.5) Comme $f'_0 = f'_3 = 0$ et $f'_1 = f'_4 = 1$, la suite $(f'_n)_{n\geq 0}$ est 3-périodique. On en déduit que $f'_{2050} = f'_7 = f'_1 = 1$: on retrouve que le 2050-ème nombre de Fibonacci est impair.

I.C.6) On peut penser à deux méthodes naturelles.

Première idée : comme $2 \equiv -1 \bmod 3$, l'entier n dont l'écriture binaire est $a_k a_{k-1} \dots a_0$ est congru modulo 3 à la somme alternée $a_0 - a_1 + \dots + (-1)^k a_k$. Nous allons donc construire un automate qui calcule, en lisant le mot $a_k a_{k-1} \dots a_0$, le signe $(-1)^{k+1}$ et la valeur modulo 3 de la somme alternée $a_k - a_{k-1} + \dots + (-1)^k a_0$. Cet automate possède 6 états, notés 0^+, 0^-, 1^+, 1^-, 2^+ et 2^- et est représenté par le graphe :



Quand on lit le mot $a_k a_{k-1} a_0$, on atteint l'état S^s où $s = (-1)^{k+1}$ et $S = a_k - a_{k-1} + \dots + (-1)^k a_0$ [3]. Autrement-dit, le signe placé en exposant est le signe qui sera attribué au chiffre suivant. Cet automate calcule donc le reste modulo 3, par le biais de la fonction ψ :

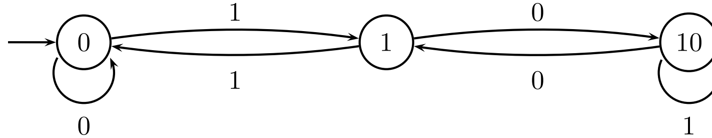
$$\psi(0^+) = \psi(0^-) = 0,\ \psi(1^-) = \psi(2^+) = 1 \text{ et } \psi(1^+) = \psi(2^-) = 2.$$

Ce n'est pas la méthode imaginée par l'auteur, car cet automate ne ressemble pas à celui de la question 3).

Seconde idée : on peut construire un automate qui effectue la division de n par 3 comme à l'école primaire, mais en base 2. Comme le quotient ne nous intéresse pas, on retient simplement le reste, qui est soit 0, soit 1, soit 10. La lecture d'un bit correspond à la "descente d'un chiffre". Le tableau suivant donne le nouveau reste, en fonction du reste précédent et du chiffre descendu :

	0	1
0	0	1
1	10	0
10	1	10

Nous obtenons ainsi directement l'automate



qui calcule le reste modulo 3, par le biais de la fonction ψ :

$$\psi(0) = 0,\ \psi(1) = 1 \text{ et } \psi(10) = 2.$$

On retrouve l'automate de la question 3).

I.D.1) La procédure est évidente :

```
let rec g n m = match n with
   0 -> 0
   |_ where n<m -> 1
   |_ -> (g (n-m) m)+(g (n-1) m);;
```

Je ne vois pas comment répondre à cette question sans donner une preuve formelle du résultat. Le nombre d'appels récursifs $T(n)$ vérifie la récurrence :

$$T(0) = T(1) = \ldots = T(m-1) = 0 \text{ et } T(n+m) = T(n+m-1) + T(n) + 2.$$

Notons P le polynôme $X^m - X^{m-1} - 1$. Si m est impair, il possède une unique racine réelle λ_1, qui est strictement supérieure à 1. Sinon, il possède deux racines réelles λ_1, λ_2, avec $-1 < \lambda_2 < 0 < 1 < \lambda_1$. Si λ est une racine complexe non réelle de P, on a :

$$|\lambda|^m = |\lambda^{m-1} + 1| < |\lambda|^{m-1} + 1$$

et donc $|\lambda| < \lambda_1$ (car $P > 0$ sur $]\lambda_1, +\infty[$). On en déduit que λ_1 est l'unique racine de module maximal de P. Comme P possède m racines complexes λ_i deux à deux distinctes (puisque 0 et $1 - 1/m$ sont les seules racines du polynôme dérivé), on en déduit qu'il existe des scalaires A_i tels que :

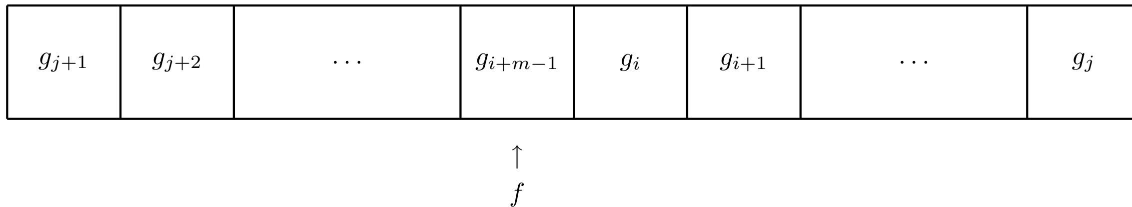
$$T(n) = -2 + \sum_{i=1}^{m} A_i\, \lambda_i^n$$

En admettant que A_1 est non nul, $T(n) = \Theta(\lambda_1^n)$ avec $\lambda_1 > 1$ et le temps de calcul est exponentiel.

I.D.2) Il suffit de stocker 3 termes consécutifs de la suite. Cela donne :

```
let g3 n = match n with
   0 -> 0
   | _ when n<3 -> 1
   | _ -> let a=ref 0 and b = ref 1 and c= ref 1 in
          for k=3 to n do
             let d=(!c) in
              c:=(!a)+(!c);
              a:=(!b);
              b:=d;
          done;
          !c;;
```

I.D.3-4) Nous devons donc stocker, pour i variant par pas de 1, les valeurs $(g_i, g_{i+1}, \ldots, g_{i+m-1})$ dans un tableau. Pour éviter de décaler les valeurs déjà calculées, nous utiliserons un tableau circulaire, représenté par un vecteur G de longueur m, initialisé à la valeur $[|0; 1; 1; \ldots; 1|]$, et un indice f initialisé à la valeur $m-1$, indiquant la fin de lecture (circulaire) du tableau :



Ainsi, le passage de k à $k+1$ se fera en modifiant une entrée du tableau G et en indentant (modulo m) le compteur f. Cela donne :

```
let g m n = match n with
   0 -> 0
   | _ when n<m -> 1
   | _ -> let G = make_vect m 1 in let f = ref (m-1) in
          G.(0)<- 0;
          for k=m to n do
             let ff = if !f = m-1 then 0 else (!f)+1 in
             G.(ff) <- G.(!f)+G.(ff);
             f := ff;
          done;
          G.(!f);;
```

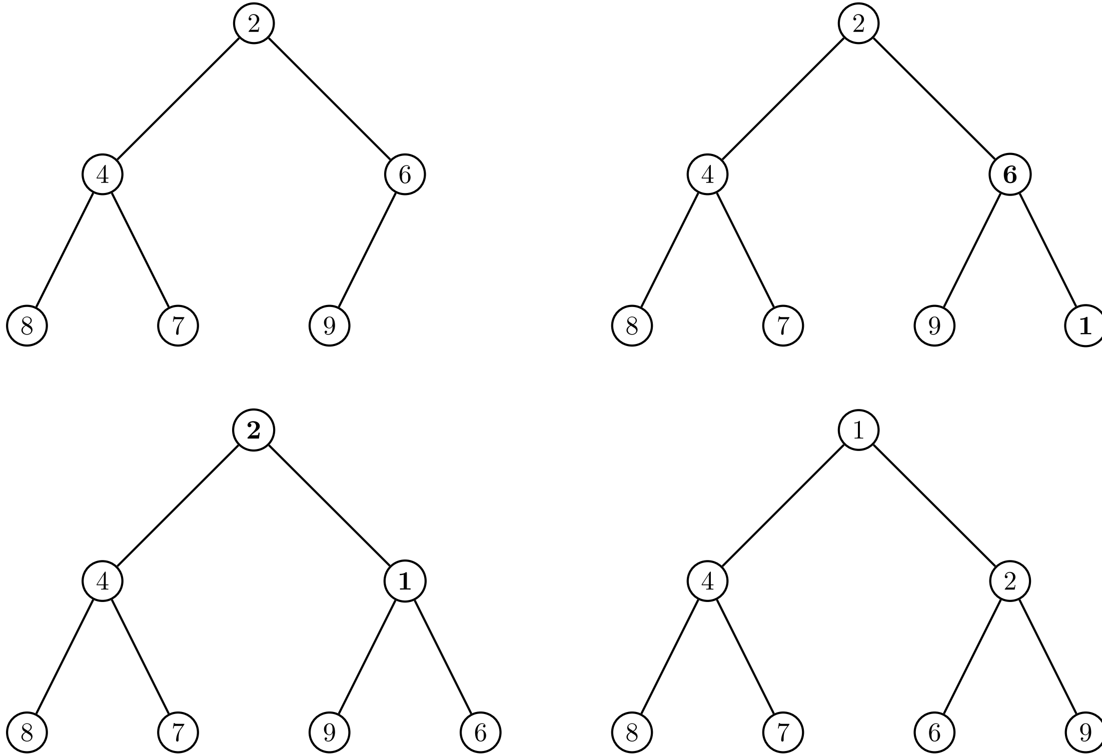
Si $n < m$, le temps de calcul est constant (indépendant de m). Si $n \geq m$, l'algorithme effectue $2(n-m+1)$ affectations et $2(n-m+1)$ additions dans le corps de la boucle.

Remarque : la distinction entre le $O(n)$ et le $O(\max(n,m))$ des questions 3) et 4) n'a pas de sens (pour $n < m$, on peut calculer g_n en temps constant), sauf si l'auteur parle de complexité par rapport à n (à m fixé) dans la question 3), et de complexité par rapport au couple (n,m) à la question 4).

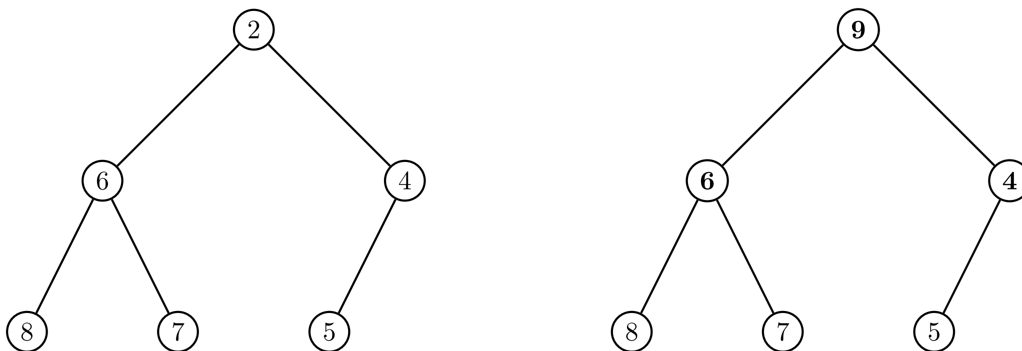
Partie II - Un calcul de ppcm

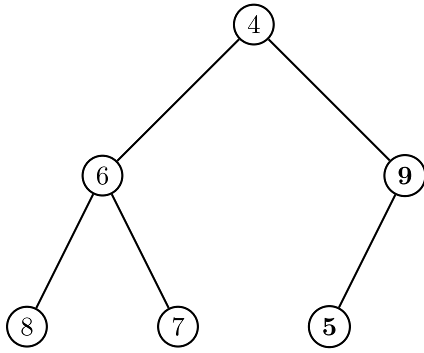
Dans cette partie, je considère que les opérations sur les entiers se font à temps constant, l'énoncé ne donnant pas de précision à ce sujet, mais il serait intéressant de reprendre les calculs de temps d'exécution en travaillant sur des entiers longs.

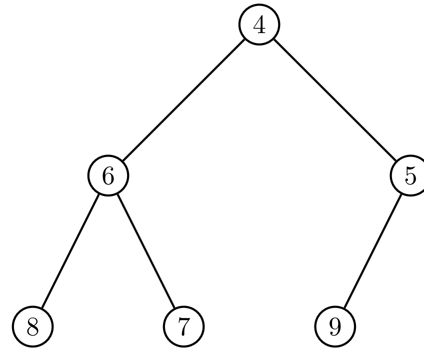
II.A On crée un nouveau nœud au seul endroit possible auquel on affecte la valeur voulue. On rétablit ensuite la structure de tas en échangeant la nouvelle valeur et la valeur affectée au nœud père, et en remontant le long d'une branche tant que la structure de tas est bafouée, comme dans l'exemple ci-dessous où l'on insère la valeur 1 :



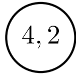
II.B Une fois changé la valeur de la racine, tous les nœuds, sauf peut-être la racine, respectent la structure de tas. On compare la valeur stockée α à la racine aux valeurs β et γ stockées dans ses fils (c'est un "match à trois") : si la structure de tas est bafouée, on échange α et le minimum de β et γ. Ainsi, tous les nœuds respectent la propriété de tas, sauf peut-être le nœud de profondeur 1 qui a été modifié. En itérant le procédé, on "descend" la valeur α dans l'arbre, jusqu'à faire disparaître le problème, en un nombre d'étape au plus égal à la hauteur de l'arbre (quand on arrive au bas de l'arbre, il est possible que l'on n'ait à comparer α qu'à une seule "valeur fille", en effectuant un "match à deux"). Cela donne, en remplaçant dans l'arbre initial la valeur 2 par 9 :

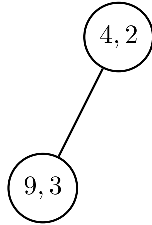


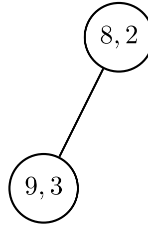


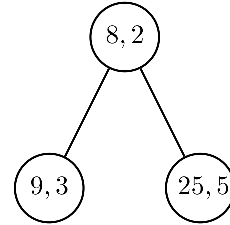


II.C On obtient successivement :

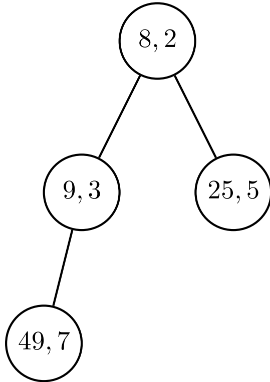


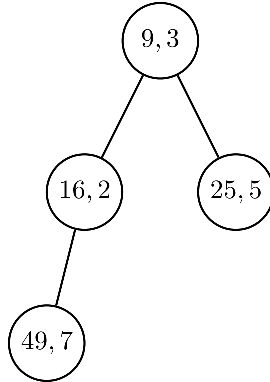


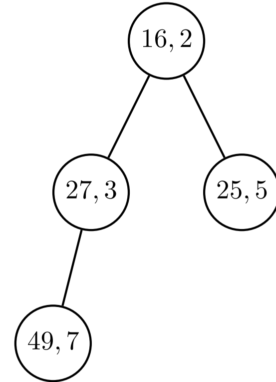




$res = 2$ $\quad$ $res = 2 \times 3 = 6$ $\quad$ $res = 6 \times 2 = 12$ $\quad$ $res = 12 \times 5 = 60$

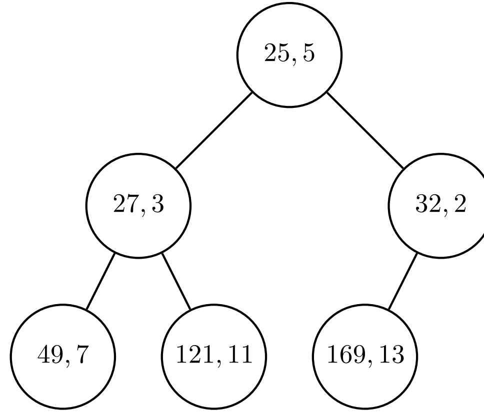






$res = 60 \times 7 = 420$ $\quad$ $res = 420 \times 2 = 840$ $\quad$ $res = 840 \times 3 = 2520$

Quand $k = 16$, on obtient l'arbre :



$$res = 16 \times 9 \times 5 \times 7 \times 11 \times 13 = 720720$$

II.D Pour i compris entre 0 et $h-1$, l'arbre contient 2^i nœuds à la profondeur i, et le nombre de nœuds de profondeur h est compris entre 1 et 2^h. Nous en déduisons :

$$2^h = 1 + 2 + 2^2 + \cdots + 2^{h-1} + 1 \leq n \leq 1 + 2 + 2^2 + \cdots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

et $h = \Theta(\ln n)$.

II.E Le coût d'une perculation, dans le pire des cas, est de l'ordre de la hauteur de l'arbre. En reprenant les notations du début de la partie II, le nombre N de percolations effectuée par l'algorithme est égal à la somme des $\alpha_i - 1$. Pour chaque i tel que $\alpha_i > 1$, on a

$$2 \leq p_i \leq \sqrt{n} \text{ et } \alpha_i - 1 \leq \alpha_i \leq \frac{\ln n}{\ln p_i} \leq \ln_2 n.$$

On en déduit :

$$N \leq \sum_{i\ /\ \alpha_i > 1} \ln_2 n \leq \sum_{2 \leq p \leq \sqrt{n}} \ln_2 n \leq (\sqrt{n} - 1)\ \ln_2 n = o(n).$$

II.F L'algorithme élémentaire usuel consiste à tester la divisibilité de n par les entiers k compris entre 2 et $\sqrt{n}$. en admettant que le calcul du reste de n modulo k se fait en temps contant, il est possible de tester la primalité de n en un temps (dans le pire des cas) de l'ordre de $\sqrt{n}$. Pour le calcul qui nous concerne, il serait plus habile d'utiliser le crible d'Ératosthène qui donne directement tous les nombres premiers inférieurs à n.

Il existe des tests de primalité plus rapides, mais cela dépasse largement le cadre du cours d'informatique de nos classes. Retenons que les seuls tests utilisables sur de grands entiers sont probabilistes : s'ils renvoient la valeur `false`, on est certain que l'entier n'est pas premier ; par contre, s'ils renvoient la valeur `true`, il y a seulement "de bonnes chances" que l'entier testé soit effectivement premier.

II.G L'énoncé a sans doute oublié de parler à la question E du nombre d'insertions (une insertion demande un temps, dans le pire des cas, de l'ordre de la hauteur de l'arbre). Le nombre M d'insertions est égal à l'entier k (en comptant l'insertion du nœud $(4, 2)$ dans l'arbre vide). On supposera connu que $k = \pi(n)$ est négligeable devant n.

Ainsi, le nombre total d'insertions-percolations est négligeable devant n, donc le temps utilisé pour ces opérations est négligeable devant $n \ln n$ (comme l'arbre contient toujours moins de n nœuds, sa hauteur est un $O(\ln n)$).

Les tests de primalité de chaque entier k demandent un temps de l'ordre de $\sum_{k=3}^{n} \sqrt{k} = \Theta(n^{3/2})$.

Enfin, les mises à jour de la variable `res` demandent $O(n)$-multiplication.

En sommant ces temps d'exécution, on obtient un temps en $\Theta(n^{3/2})$, que l'on pourrait sans doute améliorer en utilisant le crible d'Ératosthène pour calculer directement tous les nombres premiers p_i inférieurs à n, mais le temps de calcul n'est pas facile à majorer :

- on initialise un vecteur P de longueur $n+1$ à la valeur $[|false; false; true; true; \ldots; true|]$; 0 et 1 ne sont pas premiers mais les entiers qui suivent le sont *a priori* ;

- un référence p est initialisée à la valeur 2 : p est le dernier nombre premier reconnu ;

- tant que $p \leq n$:
  - on met à la valeur `false` les entrée $P.(k)$ pour k multiple (strict) de p (chaque modification coûte le prix d'une addition) ;
  - on indente ensuite p jusqu'à trouver une case contenant la valeur `true` : si cette case existe, p contient l'entier premier suivant. Sinon, on s'arrête avec $p = n + 1$.

Le seul point délicat serait de calculer le nombre d'addition effectuées par l'algorithme (certaines cases du tableau sont modifiées plusieurs fois, puisqu'un entier est en général multiple de plusieurs nombres premiers).