Corrigé : épreuve d'informatique (Centrale 2007)

Partie I. Autour de la suite de Fibonacci

I.1 Questions préliminaires

Question 1.

a) Considérons la méthode de multiplication enseignée à l'école primaire : elle consiste à effectuer n multiplications d'un entier de n bits par un entier de un bit en décalant à chaque étape d'un cran vers la gauche, puis à additionner les n nombres obtenus.
Considérons deux nombres a et b codés sur n bits, et posons $a = (a_{n-1}\cdots a_1 a_0)_2$ et $b = (b_{n-1}\cdots b_1 b_0)_2$. La multiplication de a par b avec l'algorithme décrit ci-dessus revient à calculer $\displaystyle\sum_{i=0}^{n-1} a b_i 2^i = \sum_{b_i\neq 0} a 2^i$ donc à réaliser $p-1$ additions, où p est le nombre de bits non nuls de b. Chacune de ces additions se réalise en $O(n)$ opérations élémentaires sur les bits, donc le coût total est un $\Theta(pn)$; un $O(n^2)$ dans tous les cas.

b) L'algorithme de Karatsuba permet d'obtenir un meilleur coût : il consiste à poser $a = a_1 2^p + a_2$ et $b = b_1 2^p + b_2$, où $p = \lceil n/2\rceil$ (a_1 est donc constitué des p bits de poids forts de a et a_2 des $n-p$ bits de poids faible, idem pour b).
On réalise alors le produit de a par b de manière récursive en écrivant :

$$a\times b = (a_1\times b_1)2^{2p} + \Big((a_1+a_2)\times(b_1+b_2) - a_1\times b_1 - a_2\times b_2\Big)2^p + (a_2\times b_2).$$

Cet algorithme nécessite 3 appels récursifs sur des entiers codés sur p bits ainsi que 4 additions (de coût linéaire). Si $C(n)$ désigne le coût de cet algorithme on dispose donc de la relation : $C(n) = 3C(\lceil n/2\rceil) + \Theta(n)$ qui d'après le théorème maître donne : $C(n) = \Theta(n^{\log_2(3)}) \approx \Theta(n^{1,58})$.

Question 2. L'algorithme d'exponentiation rapide utilise les relations : $a^{2p} = (a^2)^p$ et $a^{2p+1} = a\times(a^2)^p$. Si on note $C(n)$ le nombre de multiplications requises par cet algorithme on dispose des relations :

$$C(0) = C(1) = 0, \quad C(2p) = 1 + C(p) \quad \text{et} \quad C(2p+1) = 2 + C(p)$$

soit $C(n) = C(\lfloor n/2\rfloor) + \Theta(1)$. Le théorème maître prouve que $C(n) = \Theta(\log n)$, ce qui est meilleur que les $n-1$ multiplications requises par l'algorithme naïf.

Question 3. Les solutions de l'équation caractéristique $x^2 = x+1$ sont $\phi = \dfrac{1+\sqrt{5}}{2} > 1$ et $\overline{\phi} = \dfrac{1-\sqrt{5}}{2} \in]-1,0[$; il existe donc A et B tels que $f_n = A\phi^n + B\overline{\phi}^n$ pour tout n.

Les conditions initiales $f_0 = 0$ et $f_1 = 1$ fournissent $A = -B = \dfrac{1}{\sqrt{5}}$, d'où : $f_n = \dfrac{\phi^n - \overline{\phi}^n}{\sqrt{5}}$.

On a $f_n \sim \dfrac{\phi^n}{\sqrt{5}}$ donc $\log f_n = \log\dfrac{\phi^n}{\sqrt{5}} + o(1) = n\log\phi - \dfrac{\log\sqrt{5}}{2} + o(1)$ et finalement $\log f_n \sim n\log\phi$.
Le nombre de bits nécessaire pour représenter f_n est donc proportionnel à n ; c'est un $\Theta(n)$. Tout algorithme calculant f_n va l'écrire au moins une fois en mémoire ; il aura donc un coût au moins en $\Theta(n)$.

Question 4.

a)

```
let rec fibo = function
  | 0 -> 0
  | 1 -> 1
  | n -> fibo (n-1) + fibo (n-2) ;;
```

b) Notons $C(n)$ le nombre d'appels récursifs requis pour calculer f_n avec la fonction **fibo**. On dispose des relations :

$$C(0) = C(1) = 0 \qquad \text{et} \qquad \forall n\geqslant 2,\quad C(n) = 2 + C(n-1) + C(n-2).$$

On résout l'équation $k = 2 + k + k \iff k = -2$ qui suggère de poser $u_n = C(n) + 2$. Alors :

$$u_0 = u_1 = 2 \qquad \text{et} \qquad \forall n \geqslant 2, \quad u_n = u_{n-1} + u_{n-2}.$$

Il existe donc A et B tel que pour tout $n \in \mathbb{N}$, $u_n = A\phi^n + B\overline{\phi}^n$, et les conditions initiales fournissent $A = \frac{2}{\sqrt{5}}\phi$ et $B = -\frac{2}{\sqrt{5}}\overline{\phi}$.

Ainsi, $C(n) = \frac{2}{\sqrt{5}}(\phi^{n+1} - \overline{\phi}^{n+1}) - 2$ et $C(n) \sim \frac{2}{\sqrt{5}}\phi^{n+1}$; le nombre d'appels récursifs est exponentiel.

Question 5. On définit maintenant la fonction :

```
let fibo2 n =
  let rec aux acc = function
    | 0 -> fst acc
    | 1 -> snd acc
    | n -> let u, v = acc in aux (v, u + v) (n-1)
  in aux (0, 1) n ;;
```

Le nombre d'additions requis par cette fonction est égal à 0 si $n < 2$, et à $n - 1$ sinon. Puisqu'on a pris soin de rédiger une fonction récursive terminale, le seul coût spatial est celui engendré par le stockage de l'accumulateur, composé de deux entiers longs. Ceux-ci ne dépassent pas en taille f_n, qui d'après la question 3 occupe un espace proportionnel à n. Le coût spatial est donc un $\Theta(n)$.
Le coût temporel est celui requis par la réalisation de n additions sur des entiers dont la taille est un $O(n)$; on peut donc affirmer qu'il s'agit d'un $O(n^2)$.

Question 6. Il s'agit bien entendu de calculer A^n en utilisant l'algorithme d'exponentiation rapide. Ce dernier utilise un nombre de multiplications matricielles en $\Theta(\log n)$. Chaque multiplication matricielle utilise 8 multiplications et 4 additions sur des entiers longs dont la taille n'excède pas celle de f_n. Avec l'algorithme de Karatsuba pour multiplier on obtient un coût temporel total en $O(n^{\log 3} \log n)$; avec l'algorithme de multiplication naïf un coût en $O(n^2 \log n)$.
Pour peu qu'on ait rédigé l'algorithme d'exponentiation rapide sous forme terminale, le coût spatial reste cantonné au stockage de la matrice A donc est en $O(n)$.

Question 7. Si k est un entier dont le nombre de bits requis pour le représenter est notablement plus petit que n, on peut considérer que les opérations d'additions et de multiplication modulo k sont de coûts constants. Dans ce cas, l'algorithme décrit à la question précédente permet le calcul de A^n et donc de f_n en temps $\Theta(\log n)$ avec un coût spatial borné.

I.2 Une généralisation

Question 8. Si on calcule g_n à l'aide de la fonction :

```
let rec g m = function
  | 0             -> 0
  | n when n < m -> 1
  | n             -> g m (n-1) + g m (n-m) ;;
```

le coût temporel de son exécution sera exponentiel, car le nombre d'appels récursif sera lui aussi exponentiel (même raisonnement qu'à la question 4a).

Question 9. Comme à la question 5 on utilise un accumulateur pour « transporter » g_n, g_{n+1} et g_{n+2}.

```
let g3 n =
  let rec aux acc = function
    | 0 -> let (s, _, _) = acc in s
    | 1 -> let (_, s, _) = acc in s
    | 2 -> let (_, _, s) = acc in s
    | n -> let u, v, w = acc in aux (v, w, u + w) (n-1)
  in aux (0, 1, 1) n ;;
```

Question 10.

a) L'accumulateur est maintenant remplacé par un tableau de longueur m. S'agissant d'une structure mutable il n'est pas nécessaire de le passer en paramètre de la fonction auxiliaire.

```
let g m n =
  let v = make_vect m 1 in
  v.(0) <- 0 ;
  let rec aux = function
    | n when n < m -> v.(n)
    | n            -> let a = v.(0) in
                      blit_vect v 1 v 0 (m-1) ;
                      v.(m-1) <- a + v.(m-2) ;
                      aux (n-1)
  in aux n ;;
```

`blit_vect v i w j len` copie **`len`** éléments du vecteur **`v`** à partir de l'élément d'indice **`i`** dans le vecteur **`w`** à partir de l'élément d'indice **`j`**. Dans la fonction ci-dessus, son usage peut être remplacé par :

```
  for i = 0 to m-2 do v.(i) <- v.(i+1) done
```

b) La fonction ci-dessus n'effectue pas d'addition si $n < m$, et $n - m$ additions sinon. En revanche, le nombre d'affectations dans le tableau **`v`** est plus important : de l'ordre de $m(n-m)$, dues au décalage complet du tableau **`v`** qu'on effectue à chaque appel récursif. Pour en diminuer le nombre, il faut travailler avec un tableau circulaire, en passant en paramètre de la fonction auxiliaire l'indice p de la dernière case du tableau ; la première case est alors celle d'indice $(p+1) \bmod m$.

```
let g m n =
  let v = make_vect m 1 in
  v.(0) <- 0 ;
  let rec aux p = function
    | n when n < m -> v.((n+p+1) mod m)
    | n            -> let q = (p+1) mod m in
                      v.(q) <- v.(q) + v.(p) ;
                      aux q (n-1)
  in aux (m-1) n ;;
```

Question 11. Notons X_n le vecteur colonne de coefficients $g_n, g_{n+1}, \ldots, g_{n+m-1}$, et considérons la matrice

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

Sachant que $X_{n+1} = AX_n$, il s'agit de calculer $X_n = A^n X_0$ (on pourrait même se contenter de X_{n-m+1}).
Le calcul de A^n peut se faire à l'aide de l'algorithme d'exponentiation rapide, en effectuant tous les calculs dans $\mathbb{Z}/3\mathbb{Z}$. De cette façon, toute opération (addition ou multiplication) sur les entiers est de coût constant, et il est légitime de considérer qu'un produit matriciel se réalise en $\Theta(m^3)$ opérations élémentaires [1]. Le calcul de A^n a donc un temps d'exécution en $O(m^3 \log n)$.
Avec $m = 1000$ et $n = 10^{20}$ on a $m^3 \log n = 10^9 \times 20 \log 10 \approx 6,6.10^{10}$, ce qui fait de l'ordre de 66 milliards d'opérations.
En 2007, un processeur standard était capable de réaliser une opération arithmétique en environ 10^{-6} seconde ; on peut donc estimer que le temps de calcul est de l'ordre de $6,6.10^4$ secondes, soit environ 18 heures.

Partie II. Un calcul de ppcm

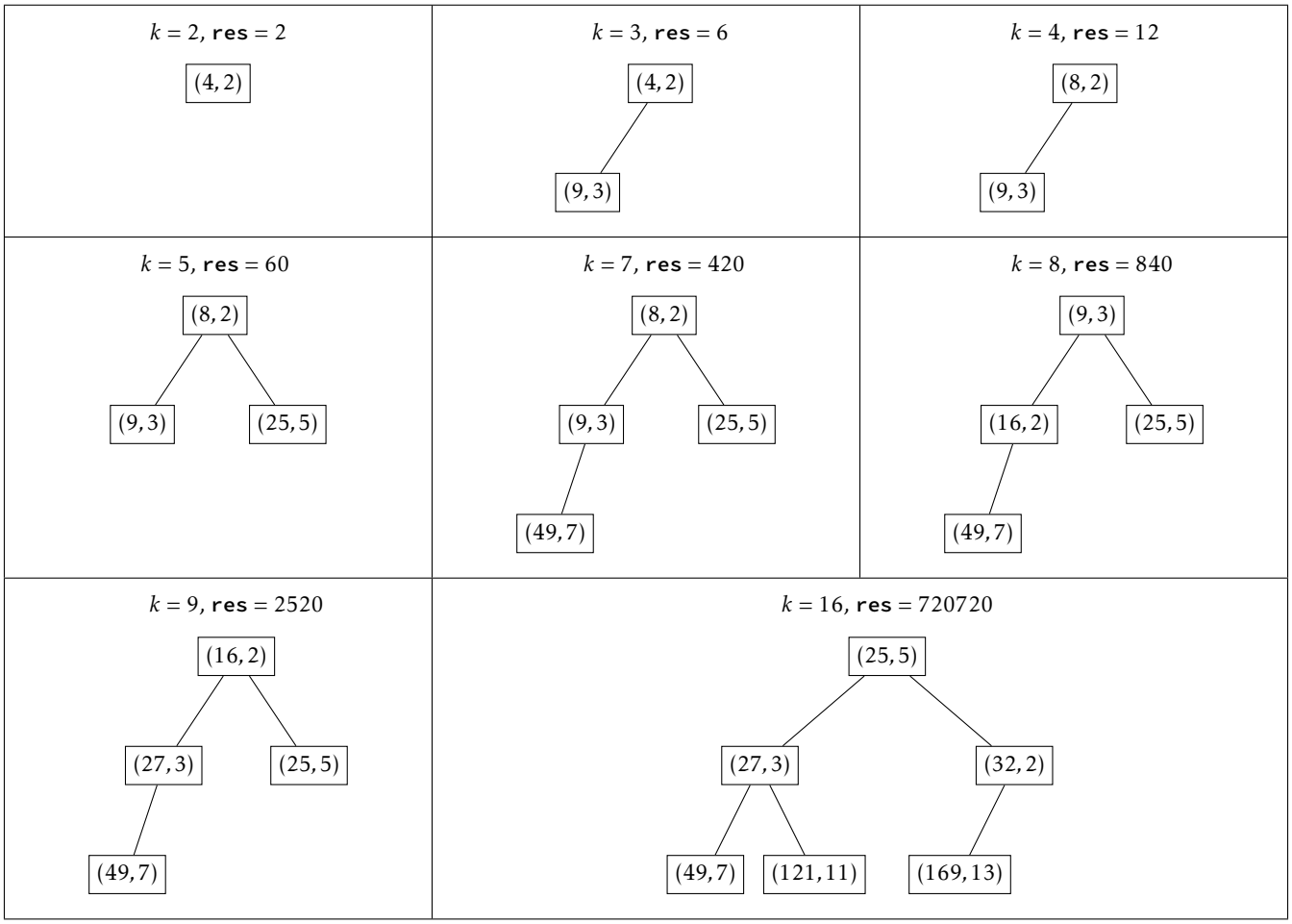
Question 12.

a) On insère un nouvel élément dans un tas en le positionnant au premier emplacement libre disponible (le premier nœud libre de la profondeur h ou le premier nœud de la profondeur $h+1$) puis en le permutant avec son père jusqu'à retrouver une structure de tas.

b) De même, si la valeur de la racine change il faut permuter cette dernière avec le plus petit de ses fils jusqu'à reconstituer la structure de tas.

1. L'algorithme de Strassen permet même de réaliser le produit matriciel en $O(m^{\log 7}) \approx O(m^{2,81})$

Question 13. On représente ci-dessous le tas pour les premières valeurs de k (le tas ne change pas pour $k = 6$ et $k = 10$; les changements des étapes $k = 11$ et $k = 13$ ne sont pas représentés) :



Question 14. Pour un tas de hauteur h le nombre de nœuds vérifie l'encadrement :

$$\sum_{k=0}^{h-1} 2^k < n \leqslant \sum_{k=0}^{h} 2^k \iff 2^h - 1 < n \leqslant 2^{h+1} - 1 \iff 2^h \leqslant n < 2^{h+1} \qquad \text{soit} \qquad \log n - 1 < h \leqslant \log n.$$

On en déduit que $h = \Theta(\log n)$, où $h = \Theta(\ln n)$ si on préfère le logarithme népérien au logarithme en base 2.

Question 15. Le coût d'une percolation ne peut excéder la hauteur de l'arbre ; il s'agit donc d'un $O(\log n)$.
Notons N le nombre de percolations effectuées lors du calcul de P_n. Avec les notations de l'énoncé,

$$N = \alpha_1 + \alpha_2 + \cdots + \alpha_k.$$

On a $p_i \geqslant 2$ donc $\sum_i \alpha_i \leqslant \sum_i \alpha_i \log p_i \sim n$. Le nombre total des percolations est donc un $O(n)$.

Question 16. On peut déterminer si un entier n est premier en cherchant s'il possède un diviseur non trivial inférieur ou égal à n, ce qui donne par exemple en Caml :

```
let is_prime n =
  let rec aux = function
    | k when k * k > n -> true
    | k                -> n mod k <> 0 && aux (k+1)
  in n > 1 && aux 2 ;;
```

Si le calcul du reste modulo n se réalise en coût constant (ce qui est le cas pour les entiers Caml) alors le coût de cette fonction est un $O(\sqrt{n})$.
Il existe des algorithmes plus efficaces, mais ceux-ci débordent largement du cadre de l'informatique enseignée en classes préparatoires. Sachez seulement que ces algorithmes sont de nature probabiliste : si la réponse de l'algorithme est négative

c'est que l'entier testé n'est pas premier ; en revanche si la réponse est positive il n'y a qu'une probabilité (très forte mais non égale à 1) que ce dernier soit premier.

Remarque. Pour le problème qui nous intéresse (obtenir l'*ensemble* des nombres premiers inférieurs ou égaux à n) il serait sans doute plus judicieux (?) d'appliquer un crible d'Érathostène plutôt que cette fonction.

Question 17. Compte tenu des résultats de la question 15, le coût de la construction du tas est un $O(n \log n)$.

Par ailleurs, Le coût des tests de primalité de tous les entiers inférieurs ou égaux à n a un coût en $O\Big(\sum_{k \leqslant n} \sqrt{k}\Big) = O(n^{3/2})$. Une fois ces nombres premiers déterminés, il faut encore trouver ceux des entiers de $[\![2, n]\!]$ qui sont de la forme p^α avec $\alpha \geqslant 2$. Il y en a $(\alpha_1 - 1) + (\alpha_2 - 1) + \cdots + (\alpha_k - 1)$ donc ce calcul s'exécute en $O(n)$ (cf question 16).

Enfin, le calcul des différentes mise à jour de la variable `res` utilisent un nombre de multiplications égal à $\alpha_1 + \cdots + \alpha_k$ qui est, on l'a vu, un $O(n)$.
Au final, on peut évaluer grossièrement le coût de cet algorithme comme étant un $O(n^{3/2})$, le coût principal étant constitué des tests de primalité.