Concours Centrale MP 2007
Epreuve d'informatique : un corrigé

1 Autour de la suite de Fibonacci.

A.1. Dans l'algorithme classique de multiplication en base 10 de l'entier a par l'entier b, on multiplie a par le nombre des unités de b puis par celui des dizaine que l'on décale vers la gauche, puis par celui des centaines etc. On somme alors toutes les quantités calculées. C'est cet algorithme élémentaire que l'on adapte ici en travaillant en base 2.
Soient $a = a_{n-1} \dots a_0$ et $b = b_{n-1}b_0$ deux entiers codés sur n bits. On a

$$ab = \sum_{i=0}^{n-1} b_i 2^i a$$

Pour effectuer la multiplication classique de a par b, on fait la somme des quantités $2^i a$ pour les i tels que $b_i \neq 0$. L'algorithme a donc l'allure suivante :

```
r := 0 (*référence pour stocker le résultat *)
m := a (*référence pour stocker 2^i a *)
Pour i de 0 à n − 1, faire
  si b_i ≠ 0 alors r := r + m
  m := 2 ∗ m
renvoyer m
```

La mise à jour de r coûte de l'ordre de n opérations (seuls n bits sont concernés par l'addition) et celle de m se fait en temps constant (on ajoute un bit nul). Le coût total de l'algorithme est donc quadratique en fonction de n ($O(n^2)$).
Remarque : si $b = 2q + r$ est la division euclidienne de b par 2. On a alors $ab = (2a)q + ra$. ra vaut 0 ou a (car $r \in \{0, 1\}$ et on peut donc calculer ab avec un appel récursif et une somme (le cas de base est celui où b est nul, le résultat valant alors 0). Ceci est une description récursif de l'algorithme itératif précédent.

A.2. Je ne vois pas d'algorithme simple (i.e. adapté à une seconde question de problème) permettant de répondre à la question. Une solution aurait été de donner un algorithme beaucoup plus mauvais en question $A.1$. Par exemple pour effectuer le calcul de ab, on peut effectuer l'addition $a + \dots + a$ un nombre de fois égal à b. Chaque addition coûte de l'ordre de n opérations et on en fait de l'ordre de 2^n (puisque b est codé sur n bits). la complexité est alors de l'ordre de $n2^n$.

A.3. On veut calculer a^b où a et b sont des entiers.
- Si $b = 0$ alors le résultat voulu est 1.
- Sinon, soit $b = 2q + r$ la division euclidienne de b par 2. On a

$$a^b = (a * a)^q a^r$$

a^r vaut a ou 1 et s'obtient en temps constant. $(a * a)^q$ se calcule par appel récursif.
Remarque : de manière itérative, en notant $b_0, \dots, b_{n-1}$ les bits de b, on a

$$a^b = \prod_{i=0}^{n-1} a^{b_i 2^i} = \prod_{i=0}^{n-1} (a^{2^i})^{b_i}$$

On calcule au fur et à mesure les a^{2^i} (le passage du rang i au rang $i + 1$ se fait par une multiplication) et on gère une référence pour calculer le produit (l'élévation à la puissance b_i ne

coûte rien car $b_i \in \{0,1\}$).

On effectue alors de l'ordre de n multiplications (au maximum $2n$) alors que dans le calcul classique, on en effectue de l'ordre de b (en multipliant a par lui même b fois).

B.1. (f_n) est une suite récurrente l'inéaire d'ordre 2 à coefficients constants dont l'équation caractéristique est $r^2 - r - 1 = 0$. Les solutions de cette dernière sont $\lambda = \frac{1-\sqrt{5}}{2}$ et $\mu = \frac{1+\sqrt{5}}{2}$. Il existe donc des constantes a et b telles que $\forall n,\ f_n = a\lambda^n + b\mu^n$. Avec $f_0 = 0$ et $f_1 = 1$, on obtient $b = 1/\sqrt{5}$ et $a = -1/\sqrt{5}$ et donc

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

On a $\mu \in]-1, 0]$ et $\lambda \in]3/2, 2[$ et donc

- $f_n/2^n$ est de limite nulle ; le nombre de bits de f_n est donc asymptotiquement plus petit que n
- pour n assez grand, $f_n \geq \left(\frac{3}{2}\right)^n$ (le quotient tend vers l'infini) ; le nombre de bits de f_n est donc asymptotiquement plus grand que $n \log_2(3/2)$.

Le nombre de bits de f_n est donc $\Theta(n)$ et le calcul de f_n prend donc au moins un temps $\Theta(n)$ (il convient de générer les bits de f_n).

B.2. La fonction récursive suit la définition. Elle est de type `int -> int`

```
let rec fibo n =
  if n=0 then 0
  else if n=1 then 1
  else (fibo (n-1)) + (fibo (n-2)) ;;
```

B.3. Notons α_n le nombre d'appels à `fibo` effectués dans le calcul de f_n par la fonction précédente. On a $\alpha_0 = \alpha_1 = 1$ et

$$\forall n \geq 2,\ \alpha_n = 1 + \alpha_{n-1} + \alpha_{n-2}$$

$(1 + \alpha_n)$ une nouvelle suite de Fibonacci (données initiales 2 et 2). Le même calcul qu'en $B.1$ donne $\alpha_n = \Theta(\lambda^n)$. La complexité est exponentielle en fonction de n.

B.4. On a besoin de garder une trace de deux termes consécutifs de la suite. C'est l'objet des références `premier` et `suivant`. La fonction est de type `int -> int`.

```
let fibo2 n =
let premier=ref 0 and suivant=ref 1 in
for i=1 to n do
   let aux = !premier in
   premier := !suivant ;
   suivant := aux + !suivant
   done ;
!premier ;;
```

On effectue une addition par boucle et donc n additions. A l'étape numéro i, on somme f_i et f_{i+1} ce qui coûte de l'ordre de i opérations (puisque f_i et f_{i+1} sont codés sur un nombre de bits de l'ordre de i). Les autres opérations se font en temps constant. Le nombre d'opérations est donc de l'ordre de $\sum_{i=1}^{n} i$ c'est à dire quadratique en fonction de n ($O(n^2)$). La place requise en mémoire est de l'ordre de n bits au maximum ($\leq 3n$ pour les deux référence et la variable auxiliaire).

B.5. Pour exploiter la relation $X_n = A^n X_0$, on calcule A^n. On gère donc une matrice M dans laquelle on stocke les puissances successives de A. Mieux, on exploite l'algorithme d'exponentiation rapide de n en notant $n = n_{k-1} \dots n_0$ la décomposition binaire de n, on écrit que

$$A^n = \prod_{i=0}^{k-1} (A_i)^{n_i} \quad \text{où} \quad \left\{ \begin{array}{l} A_0 = A \\ \forall i \geq 0,\ A_{i+1} = A_i^2 \end{array} \right.$$

- Comme $A_i X_0 = X_{2^i} = \begin{pmatrix} f_{2^i} \\ f_{1+2^i} \end{pmatrix}$, les coefficients (entiers naturels) de A_i sont codés sur un nombre de bits de l'ordre de 2^i. L'élévation au carré de A_i se fait à l'aide de 8 multiplications et quatre additions qui coûtent $O(2^{2i})$ opérations.
- Une étape lors du calcul du produit consistera à multiplier $A_0^{n_0} \dots A_{i-1}^{n_{i-1}}$ par $A_i^{n_i}$. La première matrice a des coefficients plus petits que $A_0 \dots A_{i-1} = A^{2^0+\dots+2^{i-1}} = A^{2^i-1}$ et donc plus petits que A_i. La multiplication prendra donc, comme pour le calcul de A_i^2, de l'ordre de $O(2^{2i})$ opérations au maximum.

Le nombre total d'opérations sera donc de l'ordre de

$$\sum_{i=0}^{k-1} 2^{2i} = 4^k - 1 = O(n^2)$$

Pour le calcul, on a besoin d'une matrice pour stocker les valeurs successives des A_i et une autre pour stocker les valeurs successives du produit. On a donc besoin de 8 entiers codés au maximum sur n bits.

B.6. Si on veut seulement calculer $f_n[k]$ où k est un entier fixé supposé "petit", on procède de la même façon à la différence près que l'on fait les calculs modulo k (ce qui est licite car la congruence modulo k est compatible avec l'addition et la multiplication). L'élévation au carré de A_i se fait alors en temps proportionnel au carré de la taille (nombre de bits) de k ainsi que la multiplication dans une étape du calcul (on suppose ici que le calcul modulo k ne coûte pas plus cher que le calcul simple). En négligeant le taille de k (on considère la quantité comme constante), la complexité temporelle est alors $O(\log_2(n))$. La complexité spatiale est elle réduite à 8 entiers de taille plus petite que celle de k.

C.1. On prouve le résultat par récurrence sur n.
- C'est vrai par définition de A si $n = 1$ ($f_0 = 0$ et $f_1 = f_2 = 1$).
- Soit $n \geq 1$ tel que $A^n = \begin{pmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{pmatrix}$. On a alors

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

ce qui prouve le résultat au rang $n+1$.

C.2. On en déduit que

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

et, par identification des coefficients (et $f_{n-1} = f_{n+1} - f_n$)

$$f_{2n} = f_{n-1}f_n + f_nf_{n+1} = 2f_nf_{n+1} - f_n^2$$

$$f_{2n+1} = f_n^2 + f_{n+1}^2$$

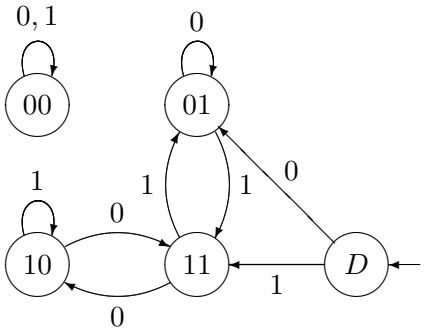
On en déduit alors que

$$f_{2n+2} = f_{2n+1} + f_{2n} = 2f_nf_{n+1} + f_{n+1}^2$$

C.3. Le tableau suivant donne les congruences modulo 2 de $f_{2n}, f_{2n+1}, f_{2n+2}$ en fonction de celles de f_n et f_{n+1}.

f_n	f_{n+1}	f_{2n}	f_{2n+1}	f_{2n+2}
0	0	0	0	0
0	1	0	1	1
1	0	1	1	0
1	1	1	0	1

On crée alors quatre états 00, 01, 10 et 11. On veut qu'après lecture de l'entier n, on soit dans l'état ij si $f_n = i[2]$ et $f_{n+1} = j[2]$. Le tableau précédent nous donne les transitions entre les états (après avoir lu n : si on lit un 0, on passe à l'état qui correspond à (f_{2n}, f_{2n+1}) et si on lit un 1, on passe à l'état qui correspond à (f_{2n+1}, f_{2n+2})). Il reste à créer un état initial D ; la lecture de 0 nous amène à $(0, 1)$ car f_0 est pair et f_1 est impair ; la lecture de 1 nous amène à $(1, 1)$ car f_1 et f_2 sont impairs.



La fonction φ envoie 01 et 00 sur 0 et 10 et 11 sur 1.

C.4. L'écriture en base 2 de $2050 = 2 * (1024 + 1) = 2^(11) + 2$ est

$$100000000010$$

En appliquant l'automate, on voit que f_{2050} et f_{2051} sont impairs (on tombe dans l'état 11) et donc

$$f_{2050}[2] = 1$$

C.5. On a

$$f_{n+3} = f_{n+2} + f_{n+1} = 2f_{n+1} + f_n$$

En passant au reste modulo 2, on a donc

$$f'_{n+3} = f'_n$$

et (f'_n) est 3-périodique. Comme $2050 = 3 * 683 + 1$, on a

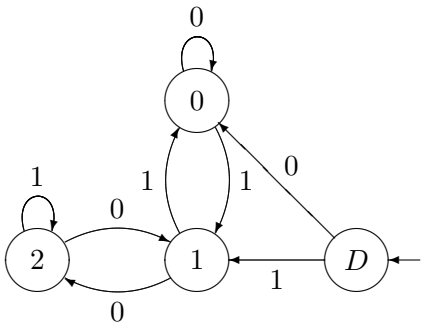
$$f'_{2050} = f'_1 = 1$$

et on retrouve le résultat précédent.

C.6. On a le tableau suivant

$n[3]$	$2n[3]$	$(2n+1)[3]$
0	0	1
1	2	0
2	1	2

On crée trois états 0, 1, 2 tels que la lecture de n nous amène en l'état $n[3]$. Le tableau précédent donne les transitions entre les trois états. Il reste à créer un état initial pour débuter la lecture.



On retrouve le même automate sans l'état 00 dont il est clair qu'il était inutile (non accessible).

D.1. La fonction est similaire à `fibo`. Elle est de type `int -> int -> int`

```
let rec g m n=
   if n=0 then 0
   else if n<m then 1
   else (g m (n-1)) + (g m (n-m)) ;;
```

On obtient encore une complexité exponentielle en fonction de n (ce qui se voit en formant l'arbre des appels récursifs).

D.2. L'idée est la même que pour `fibo2`. On gère ici trois référence `un`, `deux` et `trois` représentant respectivement g_i, g_{i+1} et g_{i+2}. La fonction est de type `int -> int`.

```
let g3 n =
let un=ref 0 and deux=ref 1 and trois=ref 1 in
for i=1 to n do
  let aux = !un in
  un:= deux ;
  deux := !trois ;
  trois := aux + !trois ;
  done ;
!un ;;
```

D.3. On procède de même mais on a besoin de m références. On choisit donc d'utiliser un tableau `stock` tel qu'à l'étape i, `stock.(k)` contient g_{i+k}. Initialement, le tableau contient donc $0, 1, \ldots, 1$. La fonction est de type `int -> int -> int`.

```
let g m n =
let stock=make_vect m 1 in
stock.(0)<-0 ;
for i=1 to n do
  let aux = stock.(0) in
  for k=0 to m-2 do stock.(k)<-stock.(k+1) done ;
  stock.(m-1) <- stock.(m-1)+aux
  done ;
stock.(0) ;;
```

D.4. La fonction précédente utilise trop d'affectation de variable (de l'ordre de mn). On peut améliorer la situation car la majorité de ces affectations sont de simples décalages de cases. En plus du tableau `stock`, on gère une référence `p`. A l'étape i, on veut que `stock.(p)` contienne g_i, les valeurs $g_{i+1}, \ldots, g_{i+m-1}$ étant contenues dans les cases $p+1, \ldots, m-1, 0, \ldots, p-1$. Seule la cas numéro p doit être modifiée (ainsi que la valeur de p). Les valeurs $p = 0$ et $p = m-1$ sont un peu particulière. La fonction est toujours de type `int -> int -> int`.

```
let g m n =
let stock=make_vect m 1 in
stock.(0)<-0 ;
let p=ref 0 in
for i=1 to n do
  if !p=0 then begin
       stock.(!p)<-stock.(!p)+stock.(m-1) ;
       p:=1
       end
  else if !p=m-1 then  begin
       stock.(!p)<-stock.(!p)+stock.(!p-1) ;
       p:=0
       end
  else begin
       stock.(!p)<-stock.(!p)+stock.(!p-1) ;
```

```
        p:=!p+1
        end
  done ;
stock.(!p) ;;
```

Remarque : on pourrait unifier les trois cas en travaillant avec des restes modulo m.

D.5. En notant X_n la matrice unicolonne de taille m de coefficients $g_n, \ldots, g_{n+m-1}$, on a

$$X_{n+1} = AX_n \quad \text{avec} \quad A = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ 0 & \ldots & \ldots & 0 & 1 \\ 1 & 0 & \ldots & 0 & 1 \end{pmatrix} \in \mathcal{M}_m(\mathbb{R})$$

Pour calculer $g_n[3]$, il nous suffit de calculer A^n en faisant toutes les opérations modulo 3. En utilisant l'exponentiation rapide, ce calcul prend au maximum $2\log_2(n)$ multiplications et chaque multiplication coûte m^2 multiplications de nombres parmi $\{0, 1, 2\}$. Le nombre d'opérations est de l'ordre de $m^2.log_2(n)$ ce qui semble "raisonnable" pour les données fournies (comme $10^{20} \leq 2^{80}$ on a $\log_2(10^{20}) \leq 80$ et on a un maximum de 8 millions d'opérations). On pourrait améliorer le calcul en utilisant une multiplication rapide pour les matrices (algorithme de Strassen).

2 Un calcul de ppcm.

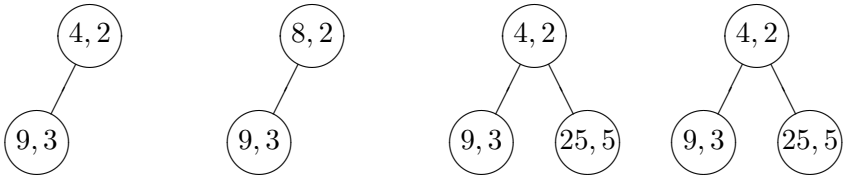
A. On commence par placer le nouveau couple (a, b) en bas de l'arbre dans un nouveau noeud N (qui est d'ailleurs une feuille) de manière à ce que l'arbre garde la bonne forme (on l'insère donc au dernier niveau le plus à gauche possible). On effectue alors les opérations suivantes :
- si (a, b) est à la racine ou si le père du noeud N a une étiquette inférieure à (a, b), on arrête ;
- sinon, on échange N et son père et on recommence.

On obtiendra toujours des arbres ayant la bonne forme et dans lesquels la propriété "d'ordre" ne peut être contredite que pour un unique noeud. De plus, la profondeur de ce noeud diminue au fur et à mesure de l'itération jusqu'à ce que le problème disparaisse (éventuellement en atteignat la racine). Ceci justifie sommairement la validité de la statégie.

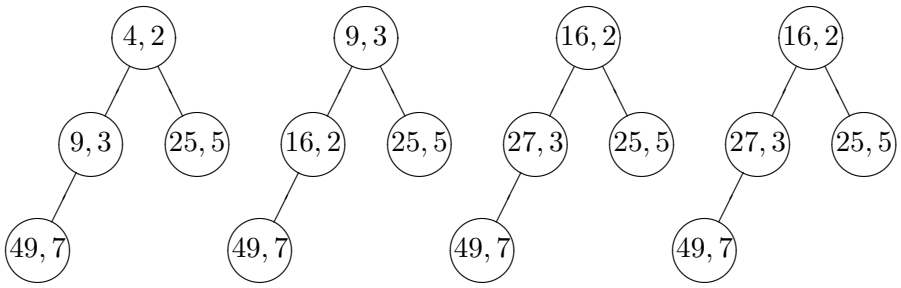
B. On cherche ici à faire "l'inverse" de la remontée dans la question précédente. Il s'agit de faire prendre à une étiquette, initialement placée à la racine, sa bonne place dans l'arbre. Soit N l'unique noeud présentant un conflit éventuel avec l'un de ses fils (initialement, la racine).
- Si N est une feuille ou si l'étiquette de N est plus grande que celle de chacuns de ses fils existants, on ne fait rien.
- Sinon, on échange N avec celui de ses fils qui a la plus petite étiquette et on recommence.

La justification de la validité de l'algorithme est similaire à celle de la question précédente.

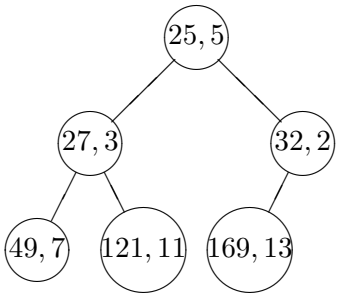
C. Voici les tas après calcul de P_3, P_4, P_5, P_6 :



Et ceux après calcul de P_7, P_8, P_9, P_{10} :



Voici enfin l'état du tas après calcul de P_{16} :



D. Soit T un tas et h sa hauteur. Pour tout $k \in [0..h-1]$, il y a 2^k noeud de profondeur k. Il y a donc au maximum $1+2+\ldots+2^h = 2^{h+1}-1$ noeuds dans un tas de hauteur h et au minimum $(1+2+\cdots+2^{h-1})+1 = 2^h$ (puisque tous les noeuds envisageables de profondeur $\leq h-1$ sont présents et qu'il en existe au moins un de profondeur h).
La hauteur h d'un arbre parfait à n noeuds vérifie donc $2^h \leq n < 2^{h+1}$ et on obtient

$$h = E\left(\log_2(n)\right) = \Theta(\ln(n))$$

E. Dans l'étape de percolation, un unique noeud viole la propriété de l'ordre et au fur et à mesure de l'itération, sa profondeur augmente. Il y a donc au maximum h étapes où h est la hauteur de l'arbre. Chaque étape se fait en temps constant et le coût est linéaire en fonction de la hauteur de l'arbre c'est à dire $\Theta(\ln(n))$.
Quand on effectue une percolation, l'entier k considéré est une puissance ≥ 2 de nombre premier. Il suffit donc de montrer que le nombre d'entiers du type $p^\alpha \leq n$ avec p premier et $\alpha \geq 2$ est négligeable devant n. Pour un nombre premier fixé p, le nombre de α convenables est de l'ordre de $\frac{\ln(n)}{\ln(p)}$. Comme on ne s'intéresse qu'aux puissances plus grandes que 2, seuls les nombres premiers $\leq \sqrt{n}$ sont concernés. En notant $p_1, \ldots, p_k$ la suite des nombres premiers compris entre 2 et $\sqrt{n}$, le nombre de percolations est de l'ordre de

$$\ln(n) \sum_{i=1}^{k} \frac{1}{\ln(p_i)}$$

Comme $k \leq \sqrt{n}$, le nombre de percolation est au plus $\ln(n)\sqrt{n}$ et est négligeable devant n.

F. Pour déterminer si un entier n est premier, on peut tester si n est divisible par $2, 3, \ldots, E(\sqrt{n})$ en s'arrêtant si on trouve un diviseur car alors n n'est pas premier. En considérant que le test de divisibilité se fait en temps constant, la complexité de l'algorithme est $O(\sqrt{n})$. Si on exprime la complexité en fonction du nombre de bits, on obtient une complexité exponentielle.
Il existe des algorithmes plus efficaces mais le problème est fondamentalement difficile. Il existe un algorithme polynomial (en fonction du nombre de bits) en théorie mais pas en pratique. Les tests les plus efficaces sont probabilistes.

G. Chaque percolation ou insertion se fait en $O(\ln(n))$ (majorant la hauteur de l'arbre). Il y a, d'après la question E, un nombre négligeable de telles opérations. Tout ceci a donc un coût

négligeable devant $n\ln(n)$. A chaque étape, on teste si k est premier ce qui coûte de l'ordre de $\sqrt{k}$ opérations avec l'algorithme élémentaire. Les tests de primalité coûtent donc de l'ordre de

$$\sum_{k=1}^{n} \sqrt{k} \underset{n\to+\infty}{\sim} \int_1^n \sqrt{x}\, dx = O(n^{3/2})$$

Ce coût excède celui des percolations et insertions.
Il reste à prendre en compte la mise à jour de `Res`. Le nombre de multiplications à effectuer est $\sum_{i=1}^{k} \alpha_i$. Or, $\ln(P_n) = \sum_{i=1}^{k} \alpha_i \ln(p_i)$ est équivalent à n. Le nombre des multiplications a donc un ordre de grandeur au plus égal à n. Une multiplication se faisant en $\ln(n)^2$ opérations sur les bits de n, on obtient là encore un coût négligeable devant $n^{3/2}$.
Le coût total semble donc être $O(n^{3/2})$.