Centrale 2007. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

Partie I. Autour de la suite de Fibonacci.

I.A. Questions préliminaires.

Question I.A.1.

En développant directement $c = ab$ avec $a = \sum_{i=0}^{n-1} a_i 2^i$ et $b = \sum_{j=0}^{n-1} b_j 2^j$ on obtient :

$c = \sum_{i=0}^{n} \Big(\underbrace{\sum_{j=0}^{n} a_i b_j 2^{i+j}}_{d_i} \Big)$ ce qui conduit à additionner n entiers d_i.

Le calcul de chaque entier d_i demande n multiplications de bits donc le calcul de tous les entiers d_i demande n^2 multiplications de bits.
Comme chaque entier d_i ne dépasse pas $2n$ bits, chaque addition requiert $O(n)$ additions de bits et donc au total $O(n^2)$ additions de bits.
Ainsi le coût du produit de deux entiers codés sur n bits est-il de $O(n^2)$ additions et multiplications de bits. □

Remarque : on a aussi $c = \sum_{i \in A} \Big(\underbrace{\sum_{j=0}^{n} b_j 2^{i+j}}_{e_i} \Big)$ en notant A l'ensemble des indices i tels que $a_i \neq 0$.

Il n'y a alors plus de multiplications et le nombre d'entiers e_i à ajouter peut être significativement moindre que n donc le nombre d'additions de bits moindre que n^2. On rajoute seulement n tests de nullité des bits de a.

Question I.A.2.

Á part le principe de la multiplication rapide de Knuth (mais non au programme !!) je ne vois pas.
Quitte à compléter a et b par un bit fort nul, on suppose a et b codés sur $2n$ bits. On écrit alors $a = \alpha_1 + 2^n \alpha_2$ et $b = \beta_1 + 2^n \beta_2$ avec $\alpha_1 = \sum_{i=0}^{n-1} a_i 2^i$ et $\alpha_2 = \sum_{i=0}^{n-1} a_{n+i} 2^i$ et idem pour la décomposition de b.
Alors $ab = \alpha_1\beta_1 + 2^n\Big((\alpha_1+\alpha_2)(\beta_1+\beta_2) - \alpha_1\beta_1 - \alpha_2\beta_2\Big) + 2^{2n}\alpha_2\beta_2$.
Ainsi le calcul de ab se ramène au calcul du produit de 3 entiers sur n bits : $\alpha_1\beta_1$, $\alpha_2\beta_2$ et $(\alpha_1+\alpha_2)(\beta_1+\beta_2)$ et au calcul de 4 sommes d'entiers codés sur au plus $2n$ bits (les multiplications par 2^n et 2^{2n} se traduisent par $\Theta(n)$ décalages de bits).
Notons $T(2n)$ la complexité temporelle de cet algorithme de type diviser pour régner.
La création des 4 entiers α_1 et β_i se fait en un temps proportionnel à n ($4n$ recopies de bits).
Le calcul de $(\alpha_1+\alpha_2)$ et $(\beta_1+\beta_2)$ en $O(n)$. De même que le calcul des 3 autres sommes et les multiplications par 2^n et 2^{2n} une fois les 3 produits d'entiers de taille n effectués.
Ainsi $T(2n) = 3T(n) + O(n^1)$. Comme $1 < \omega = \log_2 3$, il vient $T(n) = \Theta(n^\omega)$. □

Question I.A.3.

En notant $f(p) = a^p$, f satisfait la récursion $f(0) = 1$, $f(2p) = x*x$ et $f(2p+1) = a*x*x$ avec $x = f(n)$. On effectue ainsi un nombre de multiplications de l'ordre de $\log_2(p)$ (c'est à dire de l'ordre du nombre n de chiffres de l'écriture de p en base 2) au lieu de $p-1$ par l'algorithme "naïf". □

Remarque : cela débouche naturellement sur un algorithme récursif non terminal que l'on peut facilement dérécursifier en un algorithme itératif car il s'agit d'une récursion sur $\mathbb{N}$ liée à la fonction de descente du "diviser pour régner" : $n \longmapsto \mathrm{Int}(n/2)$ qui se traduit par un simple décalage à droite des bits de l'entier n.
L'intérêt de la dérécursification est que la complexité spatiale est alors en $\theta(N)$ bits où N est la taille du résultat et évidemment on ne peut espérer mieux !

I.B. Diverses façons de calculer f_n.

Question I.B.1.

L'équation caractéristique $r^2 - r - 1 = 0$ admettant pour racines $\Phi = (1+\sqrt{5})/2$ et $\overline{\Phi} = (1-\sqrt{5})/2$, il existe deux constantes α et β telles que $f_n = \alpha\Phi^n + \beta\overline{\Phi}^n$.
Les conditions initiales $f_0 = 0$ et $f_1 = 1$ fournissent $\alpha = -\beta = 1/\sqrt{5}$.
ainsi $f_n = \frac{1}{\sqrt{5}}\Big(\frac{1+\sqrt{5}}{2}\Big)^n - \frac{1}{\sqrt{5}}\Big(\frac{1-\sqrt{5}}{2}\Big)^n \sim \frac{1}{\sqrt{5}}\Big(\frac{1+\sqrt{5}}{2}\Big)^n$.
Comme les logarithmes de deux infiniment grands équivalents sont équivalents, le nombre de bits dans l'écriture en base 2 de f_n qui est équivalent à $\log_2(f_n)$ est donc équivalent à λn avec $\lambda = \log_2(\Phi)$.
Ainsi tout programme calculant f_n aura donc un temps au moins linéaire : le temps de générer les bits de f_n. □

Question I.B.2.

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

Question I.B.3.

Notons R_n le nombre d'appels récursifs nécessaires au calcul de f_n par la fonction précédente.
Il vient $R_n = 2 + R_{n-1} + R_{n-2}$ et $R_0 = R_1 = 0$. Ainsi la suite $R'_n = R_n - 2$ vérifie la même récurrence que la suite (f_n) avec $R'_0 = R'_1 = -2$.
Il existe donc deux constantes a et b telles que $R_n = 2 + a\Phi^n + b\overline{\Phi}^n \sim a\Phi^n$. □

Question I.B.4.

```
let fibo2 n = let x0 = ref 0 and x1 = ref 1 in
  for k=2 to n do let x = !x1 + !x0 in
    x0 := !x1;
    x1 := x
  done;
  !x1
;;
fibo2 : int -> int = <fun>
```

Ce programme calcule f_n pour $n \geqslant 1$.
Il effectue $n-2$ additions. De manière plus précise lors de la boucle d'indice k il ajoute f_{k-1} et f_{k-2} dont le nombre de bits est équivalent à λk (question précédente) donc est un $\Theta(k)$.
Ainsi la complexité temporelle en additions de bits est $\sum_{k=2}^{n} \theta(k) = \Theta(n^2)$ □

À chaque instant il consomme trois cases mémoires dont la plus grande taille est celle utilisée pour codée f_n à la fin soit $\Theta(n)$ bits. □

Question I.B.5.

On calcule A^n par l'algorithme d'exponentiation rapide (dérécursifié) donc avec $\Theta(\ln n)$ multiplications de matrices.
Chaque multiplication de deux matrices demande 8 multiplications et 4 additions d'entiers d'au plus n bits. (car les matrices sont remplies avec des f_k avec $k \leqslant n+1$ (Cf question I.C.1) ou avec des 0 et des 1 (cas de la matrice A pour une puissance impaire)).
En effectuant les multiplication d'entiers avec la multiplication rapide de Knuth, chaque multiplication de matrices a une complexité temporelle en $O(n^\omega)$ avec $\omega = \log_2 3$.
Ainsi la complexité temporelle totale du calcul de f_n est $O(n^\omega \ln n) = o(n^2)$. □

En utilisant la version dérécursifiée de l'exponentiation rapide (Cf I.A.3), la complexité spatiale est celle du résultat A^n c'est à dire $\Theta(n)$ bits. □

Question I.B.6.

Soient un entier k fixé, $\overline{X}_n = \begin{pmatrix} \overline{f}_n \\ \overline{f}_{n+1} \end{pmatrix}$ où $\overline{f}_i$ est la classe de f_i modulo k et $\overline{A^n}$ la matrice dont les éléments sont les classes modulo k des éléments de A^n. Alors, la relation de congruence étant compatible avec la multiplication et la multiplication, $\overline{X}_n = \overline{A^n}\,\overline{X}_o$ et $\overline{A^n} = \overline{A}^n$.
On calcule alors $\overline{A}^n$ par l'algorithme d'exponentiation rapide dérécursifié d'où comme précédemment $\Theta(\ln n)$ multiplications de matrices dans $\mathbb{Z}/k\mathbb{Z}$.
Chaque multiplication de matrices demande 8 multiplications et 4 additions dans $\mathbb{Z}/kZ$.
Pour multiplier deux éléments de $\mathbb{Z}/kZ$ codés par leur représentant entre 0 et $k-1$ on effectue déja la multiplication dans $\mathbb{Z}$ de ces deux codes qui sont sur $\Theta(\ln k)$ bits. D'où pour cela une complexité temporelle de $\Theta(\ln^2 k)$. Puis le calcul du représentant entre 0 et $k-1$ de la classe de ce produit inférieur à k^2 se fait par au plus $k-1$ soustractions de l'entier k donc par au plus $k-1$ additions d'entiers codés sur au plus $\log_2(k^2) = \Theta(\ln k)$ bits d'où une complexité temporelle de $\Theta(k \ln k)$. Ainsi la complexité temporelle du produit de deux éléments dans $\mathbb{Z}/k\mathbb{Z}$ est de $\Theta(\ln^2 k + k\ln(k)) = \Theta(k \ln k)$.
C'est aussi la complexité du produit de deux matrices dans $\mathbb{Z}/k\mathbb{Z}$.
Finalement la complexité temporelle du calcul de la classe de f_n modulo k est de $\Theta(k \ln k)\Theta(\ln n)$
Donc pour k fixé, lorsque $n \to +\infty$ on a une complexité temporelle en $\Theta(\ln n)$. □

Par le même argument que dans la question précédente, la complexité spatiale est ici bornée. □

I.C. Utilisation d'automates.

Question I.C.1.

Vérification immédiate par récurrence. □

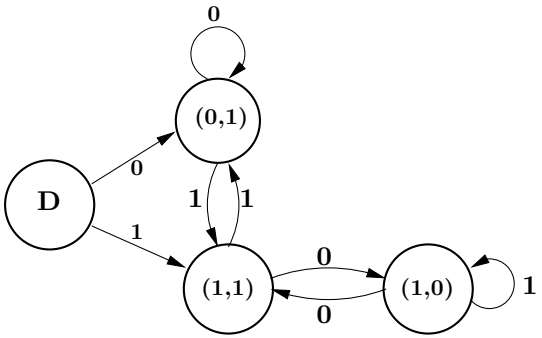
Question I.C.2.

En remarquant que $A^{2n+1} = A^n A^{n+1}$ et en utilisant la question précédente (ainsi que la récurrence de Fibonacci), on obtient par identification des termes des deux matrices :

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

Question I.C.3.

Notons $f'_k = f_k\ [2]$ et $M_n = (f'_n, f'_{n+1})$.
Il résulte facilement de la question précédente que :
$f'_{2n} = f'_n \quad f'_{2n+1} = f'_n + f'_{n+1} \quad f'_{2n+2} = f'_{n+1} \quad (1)$
Remarquons par ailleurs que si $a_n a_{n-1} \ldots a_k$ est le codage d'un entier p alors $a_n a_{n-1} \ldots a_k 0$ est le codage de 2p et $a_n a_{n-1} \ldots a_k 1$ celui de $2p+1$.
Soit Q l'ensemble des valeurs possibles de M_n.
Si l'on connait M_p, l'état à la suite de la lecture des bits de p : $a_n a_{n-1} \ldots a_k$ alors on connait l'état à la suite de la lecture de $a_n a_{n-1} \ldots a_k 0$ et de $a_n a_{n-1} \ldots a_k 1$. En effet il s'agit respectivement de M_{2n} et M_{2n+1} calculables grâce à (1).
Ce qui fournit l'automate déterministe suivant en notant D l'état initial correspondant à une suite de bits nuls :



Question I.C.4.

$2050 = 2048 + 2 = 2^{11} + 2 = \overline{100000000010}$ et la lecture de l'automate conduit à l'état $(1,1)$ donc $f_{2050} \equiv f_{2051} \equiv 1 \mod(2)$ □

Question I.C.5.

On a $f_{n+3} = f_{n+2} + f_{n+1} = 2f_{n+1} + f_n$ donc $f'_{n+3} = f'_n$. □

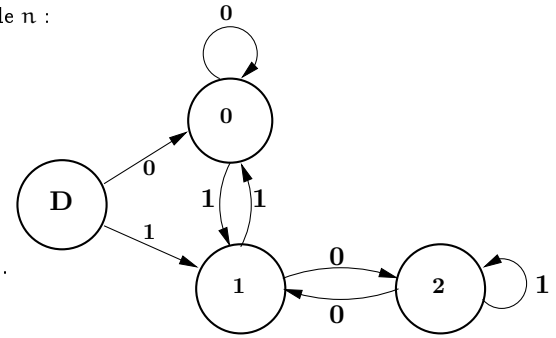
On a $2050 = 3 \times 683 + 1$ donc $f'_{2050} = f'_1 = 1$ □

Question I.C.6.

Modulo 3 on a $2n \equiv -n$ et $2n+1 \equiv -n+1$. D'où le tableau suivant des congruences de $2n$ et $2n+1$ en fonctions de celle de n :

n	0	1	2
$2n$	0	2	1
$2n+1$	1	0	2

Ce qui fournit l'automate ci-contre identique au précédent.



I.C. Une généralisation.

Question I.D.1.

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

Le temps de calcul T_n de g_n vérifie la relation $T_n = T_{n-m} + T_{n-1} + A_n$ en désignant par A_n le temps de l'addition de g_{n-m} et de g_{n-1}.
L'équation caractéristique est $P(r) = r^m - r^{m-1} - 1 = 0$. Elle n'admet aucune racine multiple car les seules racines de P' sont 0 et $(m-1)/m$ non racines de f (les seules racines rationnelles possibles a priori de P sont 1 et -1 par le théorème de Gauss).
Comme P est strictement croissant sur $[1, +\infty[$ il admet une unique racine $\lambda_0 > 1$. Une étude facile des variations montre que si m est impair, λ_0 est la seule racine réelle et que si m est pair il y a une seule autre racine réelle : μ avec $-1 < \mu < 0$.
En outre si z est une racine non réelle alors $|z|^m = |z^{m-1} - 1| \leqslant |z|^{m-1} + 1 < 1$ (en effet z non nul car non réel).
En conclusion toutes les racines (réelles ou pas) de P sauf λ_0 sont en module strictement inférieures à 1 et toutes les racines sont simples.

Donc il existe $(a_0, a_1, \ldots, a_{m-1}) \in \mathbb{R}^m$ tel que $g_n = a_0\lambda_0^n + \sum_{i=1}^{m-1} a_i\lambda_i$ avec $\lambda > 1$ et $|\lambda_i| < 1$.

Or $a_0 \neq 0$ sinon la suite (g_n) convergerait vers 0 ce qui n'est pas car il est immédiat que $g_n \geqslant 1$ pour $n \geqslant m$. Ainsi $g_n \sim a_0\lambda_0^n$.

Il en découle que le nombre de bits dans le code de g_n (donc également de g_{n-m} et g_{n-1}) est $\Theta(n)$ donc $A_n = \Theta(n)$.
Ainsi $T_n = T_{n-m} + T_{n-1} + \Theta(n)$ et $T_0 = \ldots = T_{m-1} = \alpha$ (α le temps de traitement d'un cas de base de la récursion).
Sans tenir compte du temps de l'addition on a $T'_n = T'_{n-m} + T'_{n-1}$ donc (Cf étude précédente) $T'_n \sim b_0\lambda_0^n$.
La présence de $\Theta(n)$ dans la récurrence de la suite (T_n) rajoute un terme polynômial.
Finalement $T_n \sim b_0\lambda_0^n$ et la complexité temporelle, comme on pouvait s'en douter, est exponentielle. □

Question I.D.2.

```
let g3 n = match n with
  | 0     -> 0
  | 1 | 2 -> 1
  | _     -> let x0 = ref 0 and x1 = ref 1 and x2= ref 1 in
             for k=3 to n do let x = !x0 + !x2 in
               x0 := !x1;
               x1 := !x2;
               x2 := x
             done;
             !x2
;;
g3 : int -> int = <fun>
```


Question I.D.3. et I.D.4.

On dispose déjà de la version élémentaire suivante :

```
let g_iter m n = match n with
  | 0            -> 0
  | _ when n < m -> 1
  | _            -> let t = make_vect m 1 in
                    t.(0) <- 0;
                    for k=m to n do let x = t.(0) + t.(m-1) in
                      for i=0 to m-2 do t.(i) <- t.(i+1) done;
                      t.(m-1) <- x
                    done;
                    t.(m-1)
;;
g_iter : int -> int -> int = <fun>
```

Pour calculer g_n ce programme met un temps constant si $n < m$ et sinon il effectue $n - m$ tours de boucles. À chaque tour, il effectue une addition et m affectations.

Ainsi ce programme effectue $\Theta(n)$ additions d'entiers. □
Remarque : Notons là une ambiguïté dans l'énoncé : on compte ici en additions d'entiers et non de bits. En fait compte-tenu d'une étude précédente, on sait que le code de g_n comporte $\Theta(n)$ bits donc au tour k de la boucle (il calcule g_{m+k-1}) il effectue $\Theta(k)$ additions de bits. D'ou finalement la complexité en nombre d'additions de bits est $\Theta(n^2)$. □

Ce programme effectue un nombre d'affectations équivalent à mn.
En fait à chaque tour $m - 1$ affectations sont de simples décalages et il n'y a qu'une seule valeur nouvelle nv dans le tableau qui se place en dernière position.

On peut gérer cela de manière plus économe en évitant les recopies dues aux décalages. Pour cela on ne modifie pas physiquement les cases en question et on fait une lecture circulaire du tableau en gérant un pointeur debut qui indiquera l'indice physique du début effectif du tableau.
On initialise bien sûr debut à 0. Lors d'un tour avec debut à l'entrée :
1/ on charge la case d'indice debut par la somme de celle d'indice debut et de celle d'indice $debut - 1 \mod(m)$;
2/ on remplace debut par $debut + 1 \mod(m)$;

La complexité en affectations de ce nouveau programme est alors $\Theta(n)$ □

```
let g_iter_opt m n = match n with
  | 0 -> 0
  | _ when n < m -> 1
  | _ -> let t = make_vect m 1 and debut = ref 0 in
           t.(0) <- 0;
           for k = m to n do
             t.(!debut) <- ( t.(!debut) + t.((!debut + m - 1) mod m) );
             debut := ((!debut + 1) mod m)
           done;
           t.((!debut + m - 1) mod m)
;;
g_iter_opt : int -> int -> int = <fun>
```

Remarques :

1/ Dans le calcul modulo m de $debut - 1$ on remplace $debut - 1$ par $debut + m - 1$ car la fonction mod de CAML ne gère pas les entiers négatifs.

2/ Le $O(\max(n, m))$ de l'énoncé est ridicule à plusieurs titres.
Ici m est un entier fixé et la variable (sous entendu grande) est n. Donc $\max(n, m) = n$!
D'ailleurs avec la version simple où le nombre d'affectations est équivalent mn on a aussi un nombre d'affectations qui est $O(n) = O(\max(n, m))$.

Question I.D.5.

- Première idée :

  On modifie simplement le programme précédent en remplacant la ligne :
  `t.(!debut) <- ( t.(!debut) + t.((!debut + m - 1) mod m) );`
  par
  `t.(!debut) <- ( ( t.(!debut) + t.((!debut + m - 1) mod m) ) mod 3);`

  À chaque tour :
  1/ Calcul de $debut + m - 1 \mod(m)$. Comme m est fixé de taille "raisonnable" on considère que cela se fait à temps constant, celui d'une opération dite élémentaire.
  2/ Calcul de la somme $t.(debut) + t.(debut - 1 \mod(m))$. Il s'agit de la somme de deux entiers entre 0 et 2 donc opération à temps constant.
  3/ Calcul de la classe modulo 3 de cette somme c'est à dire de la classe modulo 3 d'un entier entre 0 et 4. Opération élémentaire.
  4/ Gestion du pointeur ; là encore opération élémentaire.

  Donc au total $4(n - m + 1)$ opérations élémentaires pour le corps de la boucle.

  Le temps de création du tableau initial rempli par des 0 et des 1 est de m opérations élémentaires.

  Bref pour m petit devant n on peut considérer que le temps de calcul est celui de $4n$ opérations élémentaires.

  Avec $n = 10^{20}$ et une machine effectuant 10^9 opérations élémentaires par seconde, on a un temps de calcul de l'ordre de 10^{11} secondes soit de l'ordre de 3200 années !

- Deuxième idée :

  En notant A la matrice carrée d'ordre m associée à la récurrence vérifiée par la suite de vecteurs colonne $G_n = {}^t(g_n, g_{n+1}, \ldots, g_{n+m-1})$: $G_n = AG_{n-1}$ il vient $G_n = A^n G_0$ et on se ramène, comme dans la question I.B.6, au calcul de $\overline{A^n} = \overline{A}^m$ (la congruence est modulo 3).
  On utilise l'algorithme d'exponentiation rapide (dérécursifié) qui demande $\Theta(\ln n)$ multiplications de matrices de $\mathbb{Z}/3\mathbb{Z}$.
  On effectue déjà la multiplications dans $\mathbb{Z}$ des coefficients qui sont des entiers entre 0 et 2. Chaque multiplication est donc élémentaire. De même que le calcul de la classe de ce produit modulo 3.
  Bref le calcul de $\overline{A^n}$ demande de l'ordre de $\log_2(n) \times 2m^2$ opérations élémentaires.

  Avec la machine évoquée précédemment pour le même exemple on a un temps de calcul de l'ordre de $2 \times 10^{-3} \times 20 \log_2(10) < 1$ seconde !!

Partie II. Un calcul de ppcm.

Question II.A à II.E.

- **Remarque** : Le fait que la structure de donnée utilisée en machine pour stocker et manipuler les tas n'importe pas me laisse perplexe. En effet comment décrire un tant soit peu précisément l'insertion et la percolation (et surtout étudier leur coût -Cf plus loin-) sans cela ?

- Par exemple pour l'insertion, on insère le nouveau nœud à la première position libre (lecture gauche-droite du dernier niveau) puis on fait des échanges successifs du nœud courant avec son père (en partant du nœud introduit) tant que l'étiquette du nœud courant est inférieure à celle de son père.

  Or atteindre la première position libre et connaitre le père d'un nœud n'est pas facile (mais bien sûr possible) si la structure de donnée est récursive : $tas = vide$ ou $noeud \times tas \times tas$.
  Par contre cela est immédiat si l'on représente un tas par un vecteur avec l'indicage classique d'un arbre binaire parfait (sauf éventuellement le dernier niveau) : les nœuds sont numérotés en partant de 1 pour la racine et le fils gauche (si non vide) du nœud numéroté n a pour numéro $2n$ et le fils droit : $2n + 1$. Le père d'un nœud de numéro n est celui de numéro $n/2$ (division entière)
  On peut alors utiliser pour représenter un tas en mémoire un "vecteur variable" c'est à dire un enregistrement avec un premier champ (appelé "objet") qui est un vecteur de couples d'entiers et un deuxième champ mutable (appelé "lg") qui est un entier. La longueur physique du vecteur "objet" sera choisie de manière à être suffisante pour manipuler les tas que l'on rencontrera et le champ "lg" indiquera le nombre de nœuds "effectifs" de l'arbre. Par exemple pour un vecteur "objet" de longueur 100 et lg de valeur 10 cela correspondra à un arbre ayant 10 nœuds, la racine correspondant à la case 1, la racine de son fils gauche à la case 2, celle de son fils droit à la case 3, etc.... Le dernier nœud correspond à la case 10 et c'est le fils gauche de la case 5. La première position libre est la case 11 qui correspond au fils droit de la case 5.

  L'insertion du nœud new dans le tas T est alors immédiate :

  ```
  Incrémenter T.lg
  T.objet.(T.lg) <- new
  temp <- T.lg
  Tant que temp >= 2 et T.objet.(temp) <= T.objet.(temp/2)
    échanger T.objet.(temp) et T.objet.(temp/2)
    temp <- temp/2
  Fin tant que
  ```

  Voir programmation effective en CAML à la fin du corrigé en annexe.

- Si n est le nombre de nœuds du tas initial, on a immédiatement $2^h \leqslant n < 2^{h+1}$ soit $h = \Theta(\log_2 n)$ en notant h la hauteur de l'arbre.

- La complexité de l'insertion est au pire de h échanges. Ainsi complexité en $O(\ln n)$ affectations (un échange valant 3 affectations).

  Par contre avec l'autre structure de donnée on a un algorithme en $\Theta(n)$ (déjà pour la recherche par un parcours en largeur de la première position libre).

- La percolation n'est pas plus difficile : tant que la structure de tas est contredite et que le nœud courant a deux fils dont l'un au moins est plus petit on échange le nœud courant avec le plus petit de ses deux fils. Puis si le nœud courant est ainsi descendu au maximum et qu'il possède un fils (il ne peut alors en avoir 2 mais que 0 ou 1) on l'échange avec son unique fils (le gauche) si besoin.

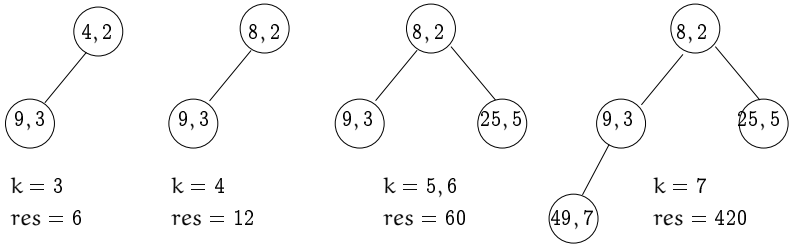
Cela fournit l'algorithme suivant :

```
T.objet.(1) <- new
temp <- 1
Tant que 2*temp+1 <= T.(lg) et
         ( T.objet.(temp) > T.objet.(2*temp) ou
           T.objet.(temp) > T.objet.(2*temp+1) )
  k <- indice de min (T.objet.(2*temp), T.objet.(2*temp+1))
  échanger T.objet.(temp) et T.objet.(k)
  temp <- k
Fin tant que
Si 2*temp = T.(lg) et T.objet.(temp) > T.objet.(2*temp)
  alors échanger T.(temp) et T.(2*temp)
Fin si
```

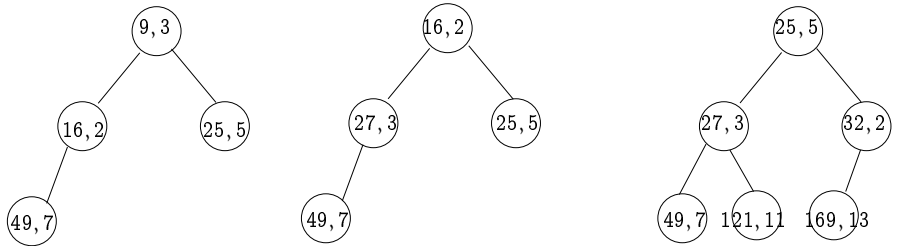
- Le coût d'une percolation est au pire de h échanges et de $2h$ comparaisons donc en $O(\ln n)$.
- On effectue une percolation lors du calcul de P_n pour les entiers $k \leqslant n$ tels que $k = p^\alpha$ avec $\alpha \geqslant 2$ et p premier. Un tel nombre premier est inférieur ou égal à $\sqrt{n}$ puisque $\alpha \geqslant 2$. Donc il y en a au plus $\sqrt{n}$. Par ailleurs pour un tel p on a $\alpha = \dfrac{\ln k}{\ln p} \leqslant \dfrac{\ln n}{\ln p} \leqslant \dfrac{\ln n}{\ln 2}$.

  Ainsi on effectue au plus $\dfrac{1}{\ln 2}\sqrt{n}\ln n$ percolations pour le calcul de P_n ce qui est bien négligeable devant n.

  Différentes valeurs successives du tas et du ppcm :



$k = 3$ res $= 6$ | $k = 4$ res $= 12$ | $k = 5, 6$ res $= 60$ | $k = 7$ res $= 420$



$k = 8$ res $= 840$ | $k = 9, 10$ res $= 2560$ | $k = 16$ res $= 720720$

Question II.F.

Un algorithme élémentaire consiste à tester si n est divisible par k en partant de 2 tant que la réponse est négative et que $k^2 \leqslant n$.
Dans le pire des cas (n premier) la complexité est en $\sqrt{n}$ à condition d'admettre que tous les tests se font à temps constant. Ce qui est le cas si l'on reste dans le domaine de validité des entiers CAML (codés en mode signé à complément à 2 sur un vecteur de 31 bits). Si l'on travaille avec des entiers de taille arbitraire (donc codés par une liste chaînés de bits) on peut voir que la complexité est exponentielle en nombre d'opérations élémentaires sur les bits.

On peut diminuer un peu le nombre de tests en éliminant déjà les multiples de 2 et de 3 (en somme le début du crible d'Ératosthène) :

```
Si (n = 2) ou (n = 3) ou (n = 5) alors VRAI; exit
Si n mod 2 = 0  alors VRAI; exit
Si n mod 3 = 0 alors VRAI; exit
prem <- VRAI; k <- 1
Tant que ((6k+1)(6k+1) <= n) et (n mod (6k+1) <> 0)
  incrémenter k
Fin tant que
Si (6k+1)(6k+1)) <= n
  alors prem <- FAUX
  sinon
    k <-1
    Tant que ((6k+5)(6k+5)) <= n et (n mod (6k+5) <> 0)
      incrémenter k
    Fin tant que
    Si (6k+5)(6k+5)) <= n alors prem <- FAUX
Fin si
prem
```

Il existe des algorithmes plus efficaces mais (à ma connaissance) ils sont probabilistes (et cela dépasse le niveau du cours de nos classes) comme le test de Rabbin-Miller. Ils sont basés sur le théorème de Fermat et permettent d'affirmer de manière certaine qu'un nombre est non premier mais seulement avec une forte probabilité qu'il est premier.

Question II.G.

- Pour calculer P_n on commence par remarquer que l'arbre final comportera un nombre de nœuds égal à $\pi(n)$ le nombre des entiers premiers inférieurs ou égaux à n. On supposera connu que $\pi(n)$ est négligeable devant n ce que nous utiliserons dans la suite. En fait $\pi(n) \sim \dfrac{n}{\ln n}$ ("théorème des nombres premiers" établi par Hadamard et de la Vallée Poussin).
- En créant un vecteur variable de longueur n initialisé par des couples $(0, 0)$ sauf la case 1 par $(4, 2)$, on pourra représenter (largement !) l'arbre final.
  La complexité spatiale de l'algorithme sera alors de $\Theta(n)$.

  Remarque 1 : on peut l'améliorer en dressant déjà la liste $\mathscr{P}$ des entiers premiers inférieurs à n avec l'algorithme élémentaire vu précédemment et en créant un vecteur variable de longueur optimale $\text{longueur}(\mathscr{P}) + 1$ ce qui fournit une complexité spatiale $\sim \dfrac{n}{\ln n}$.

  La complexité temporelle de la création de la liste $\mathscr{P}$ est majorée par $\sum_{k=2}^{n} \sqrt{k} \sim n^{3/2}$ par comparaison classique à une intégrale.

Remarque 2 : On peut relativement facilement démontrer (Cf par exemple Mines-Ponts 2002 maths 2) que pour $n \geqslant 4$ on a $\pi(n) \leqslant \alpha(n) = 2\ln 2\Big(\frac{n}{\ln n} + \frac{4n}{\ln^2 n}\Big)$.

Donc en créant un vecteur variable de longueur $E(\alpha(n)) + 1$ on aura une taille suffisante et une complexité spatiale encore en $\frac{n}{\ln n}$ sans avoir à créer auparavant la liste $\mathscr{P}$.

En fait pour des valeurs de $n \leqslant 30$ on a $\alpha(n) \geqslant n$ et il vaut mieux créer un tableau de longueur n. Par contre c'est de plus en plus avantageux ensuite bien sûr.

- Puis on fait une boucle inconditionnelle pour k variant de 3 à n et
  - si k est premier (test par l'algorithme élémentaire et non par recherche dans la liste $\mathscr{P}$) :
    on fait une insertion et on ajuste res,
  - sinon si k est égal à la première composante de la case 1 :
    on fait une percolation et on ajuste rest,
  - sinon on ne fait rien.
- Étudions la complexité temporelle.
  - Comme pour la création éventuelle de la liste $\mathscr{P}$ le coût des recherches de primalité est $O(n^{3/2})$.
  - Le coût des recherche d'égalité avec la première composante de la case 1 est $n - \pi(n) \sim n$.
  - Le coût de l'insertion de (k^2, k) lorsque k est premier est (Cf précédemment) $O\Big(\ln(\pi(k))\Big)$ (le nombre de nœuds de l'arbre est alors de $\pi(k)$) donc a fortiori $O(\ln k)$ et donc $O(\ln n)$. Ainsi le coût total des insertions est $O(\pi(n)\ln n) = O(n)$.
  - De même le coût d'une percolation est $O(\ln n)$ et comme leur nombre est évidemment majoré par n, le coût total des percolations est $O(n \ln n)$.

Il en découle que la complexité de l'algorithme est $O(n^{3/2})$ avec une complexité spatiale équivalente à $\frac{n}{\ln n}$.

Annexe : programmation en caml.

```
type 'a var_vect = {objet : 'a vect; mutable lg : int};;

let echange v i j = let temp = v.(i) in
  v.(i) <- v.(j);
  v.(j) <- temp
;;

type comparaison = PP | PG | E;;

let compare (a1,b1) (a2,b2) = match 0 with
  | _ when a1 < a2 -> PP
  | _ when a1 > a2 -> PG
  | _              -> E
;;
```



```
let insere obj a = match 0 with
  | _ when a.lg >= vect_length (a.objet) -> failwith "Taille insuffisante"
  | _ ->
    a.lg <- succ a.lg;
    a.objet.(a.lg) <- obj;
    let rang = ref a.lg in
      while (!rang > 1) &
            compare a.objet.(!rang)  a.objet.(!rang / 2) = PP do
        echange a.objet !rang (!rang / 2);
        rang := !rang / 2
      done
;;

let percolation obj a =
  a.objet.(1) <- obj;
  let rg = ref 1 in
    while (2*(!rg)+1 <= a.lg) &
          ((compare a.objet.(!rg)  a.objet.(2*(!rg)) = PG) ||
          (compare a.objet.(!rg)  a.objet.(2*(!rg)+1) = PG)) do
      let k = if (compare a.objet.(2*(!rg))  a.objet.(2*(!rg)+1)) = PP
      then 2*(!rg) else 2*(!rg)+1 in
        echange a.objet !rg k;
        rg := k
    done;
    if (2*(!rg) = a.lg) & (compare a.objet.(!rg)  a.objet.(2*(!rg)) = PG)
    then echange a.objet !rg (2*(!rg))
;;

let premier n = match n with
  | 2 | 3 | 5 -> true
  | _ when n mod 2 = 0 -> false
  | _ when n mod 3 = 0 -> false
  | _ -> let prem = ref true and k = ref 1 in
           while ((6*(!k)+1)*(6*(!k)+1) <= n) &
                 (n mod (6*(!k)+1)*(6*(!k)+1) <> 0) do
             incr k
           done;
           if (6*(!k)+1)*(6*(!k)+1) <= n then prem := false
           else
             begin
               let k = ref 1 in
                 while ((6*(!k)+5)*(6*(!k)+5) <= n) &
                       (n mod (6*(!k)+5)*(6*(!k)+5) <> 0) do
                   incr k
                 done;
                 if (6*(!k)+5)*(6*(!k)+5) <= n then prem := false
             end;
           !prem
;;
```

```
let alpha n = let x = float_of_int n in
  int_of_float (2.*.log(2.)*.(x/.log(x)+.4.*.x/.log(x)/.log(x)))
;;
```

Le programme suyivant renvoie alors le ppcm du produit des entiers inférieurs ou égaux à n ainsi que le tas correspondant :

```
let ppcm n = let m = if n <=30 then n else alpha n in
  let res = ref 2 and a = {objet = make_vect m (0,0); lg = 1} in
    a.objet.(1) <-(4,2);
    for k=3 to n do
      if premier k then
        begin
          insere (k*k,k) a;
          res := !res * k
        end;
      let (r,p) = a.objet.(1) in
        if r = k then
          begin
            percolation (r*p,p) a;
            res := !res * p
          end;
    done;
    a,!res
;;
```

Pour $n = 16$ on retrouve bien le résultat calculé à la main précédemment :

```
#ppcm 16;;
- : (int * int) var_vect * int =
 {objet =
   [|0, 0; 25, 5; 27, 3; 32, 2; 49, 7; 121, 11; 169, 13; 0, 0; 0, 0;
     0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0|];
  lg = 6},
 720720
```

Avec $n = 24$ on atteint la limite calculable avec les entiers normaux en CAML (ce qui rend a posteriori inutile la comparaison entre $\alpha(n)$ et n !)

```
#ppcm 24;;
- : (int * int) var_vect * int =
 {objet =
   [|0, 0; 25, 5; 27, 3; 32, 2; 49, 7; 121, 11; 169, 13; 289, 17; 361, 19;
     529, 23; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0; 0, 0;
     0, 0; 0, 0; 0, 0; 0, 0; 0, 0|];
  lg = 9},
 1059261584
```

FIN