Corrigé : plus proche ancêtre commun

Partie I. Une solution simple

Question 1. On définit la fonction suivante :

```
let remplir_taille () =
  let rec aux = function
    | Noeud (k, []) -> taille.(k) <- 1
    | Noeud (k, lst) -> do_list aux lst ;
                        taille.(k) <- list_it (fun (Noeud (a, _)) b -> taille.(a) + b) lst 1
  in aux A ;;
```

La fonction auxiliaire **aux** est de type *arbre -> unit*. Appliquée à un nœud k sans descendance, elle stocke la valeur 1 dans **taille.(k)** ; appliquée à un nœud avec descendance, elle agit récursivement sur chacun des fils (à l'aide de la fonctionnelle **do_list**) puis range dans **taille.(k)** la somme des tailles de ses fils plus 1 (calculée à l'aide de la fonctionnelle **list_it**).

Question 2. Compte tenu de la propriété (P), i appartient au sous-arbre j si et seulement si $i \in [\![j, j+k-1]\!]$, où k est la taille du sous-arbre j. D'où la fonction :

```
let appartient i j = (j <= i) && (i < j + taille.(j)) ;;
```

Question 3. D'après la propriété (P), si $k \leqslant j \leqslant i$ et si i appartient au sous-arbre k, alors k est un ancêtre commun à i et j. Le plus proche ancêtre commun à i et à j est donc le plus grand de ces entiers k.

```
let ppac1 i j =
  let ii = max i j and jj = min i j in
  let rec aux = function
    | k when appartient ii k -> k
    | k                      -> aux (k - 1)
  in aux jj ;;
```

Partie II. Une solution plus efficace

Question 4. Montrons par récurrence sur k que le tour eulérien d'un sous-arbre i de taille k contient $2k-1$ éléments :

- si $k = 1$, le sous-arbre i n'a pas de fils donc son tour eulérien ne contient que lui-même ;
- si $k > 1$, on suppose le résultat acquis pour tous les sous-arbres de tailles inférieures, et on considère la liste $(j_1, \ldots, j_p)$ de ses fils, ainsi que les tailles $k_1, \ldots, k_p$ des sous-arbres associés. On a : $k_1 + \cdots + k_p = k-1$.
  Le tour eulérien du sous-arbre i prend la forme suivante :
  $$i, (\text{tour eulérien de } j_1), i, (\text{tour eulérien de } j_2), i, \ldots, i, (\text{tour eulérien de } j_p), i$$
  donc il contient : $(2k_1 - 1) + \cdots + (2k_p - 1) + p + 1 = 2(k_1 + \cdots + k_p) + 1 = 2k - 1$ éléments.

Il en résulte que le tout eulérien de l'arbre **A** contient exactement $2n-1$ éléments.

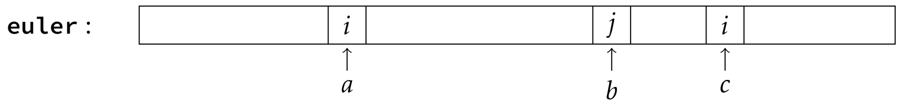
Question 5. La fonction qui suit utilise une référence qui indique le premier emplacement non encore rempli de la séquence résultat.

```
let remplir_euler () =
  let k = ref 0 in
  let ajoute i = euler.(!k) <- i ; index.(i) <- !k ; incr k in
  let rec aux = function
    | Noeud (i, lst) -> ajoute i ; do_list (function j -> aux j ; ajoute i) lst
  in aux A ;;
```

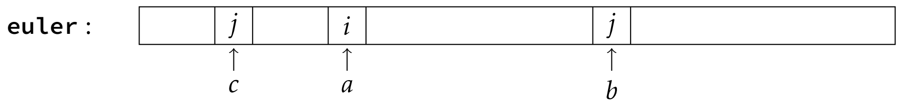
Question 6. Commençons par une observation : entre deux occurrences *consécutives* d'un nœud i dans le tableau **euler** se trouve l'ensemble des descendants d'*un* de ses fils. Entre la première et la dernière occurrence d'un nœud i se trouve donc l'*ensemble* des descendants de i plus éventuellement lui-même, qui ont tous des indices *supérieurs ou égaux* à i.

Posons $a = \texttt{index.}(i)$ et $b = \texttt{index.}(j)$. Sans perte de généralité nous pouvons supposer $a < b$, le cas d'égalité étant trivial. Notons alors μ le plus petit élément du tableau **euler** compris entre les indices a et b.

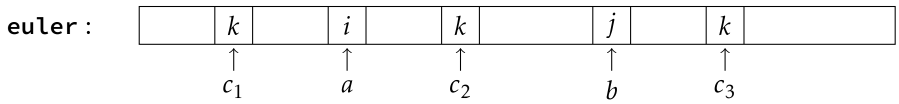
– S'il existe un entier $c > b$ tel que $\texttt{euler.}(c) = i$, alors j est un descendant de i ainsi que tout nœud présent entre les indices a et b donc $\mu = i = \mathrm{PPAC}(i,j)$.



– S'il existe un entier $c < a$ tel que $\texttt{euler.}(c) = j$, alors i est un descendant de j ainsi que tout nœud présent entre les indices a et b donc $\mu = j = \mathrm{PPAC}(i,j)$.



– Dans les autres cas i ne peut être un descendant de j ni j un descendant de i ; ils ont donc un ancêtre commun k, et il existe trois indices c_1, c_2 et c_3 tels que $\texttt{euler.}(c_1) = \texttt{euler.}(c_2) = \texttt{euler.}(c_3) = k$ avec $c_1 < a < c_2 < b < c_3$.



(Si l'entier c_2 n'existait pas, i et j appartiendraient à la descendance d'un fils de k, ce qui contredirait la définition de k.) Tous les éléments du tableau compris entre les indices a et c_2 sont donc supérieurs ou égaux à k, tout comme ceux compris entre les indices c_2 et b. On a donc bien $\mu = k = \mathrm{PPAC}(i,j)$.

Question 7. La fonction qui suit est basée sur l'observation suivante :

$$2^k \leqslant n < 2^{k+1} \iff 2^{k-1} \leqslant \frac{n}{2} < 2^k \iff 2^{k-1} \leqslant \left\lfloor \frac{n}{2} \right\rfloor < 2^k$$

```
let rec log2 = function
  | 1 -> 0
  | n -> 1 + log2 (n/2) ;;
```

Question 8. La construction de la matrice `M` repose sur les égalités suivantes :

(i) $\forall i \in [\![0, m-1]\!]$, $\quad \texttt{M.}(i)\texttt{.}(0) = \texttt{euler.}(i)$;

(ii) $\forall j \geqslant 1, \forall i \in [\![0, m-2^j]\!]$, $\quad \texttt{M.}(i)\texttt{.}(j) = \min\Big(\texttt{M.}(i)\texttt{.}(j-1),\ \texttt{M.}(i')\texttt{.}(j-1)\Big)$ avec $i' = i + 2^{j-1}$.

En effet, $\Big[\!\!\Big[i, i+2^j \Big[\!\!\Big[= \Big[\!\!\Big[i, i+2^{j-1} \Big[\!\!\Big[\cup \Big[\!\!\Big[i', i'+2^{j-1} \Big[\!\!\Big[$.

```
let remplir_M () =
  for i = 0 to m - 1 do
    M.(i).(0) <- euler.(i)
  done ;
  let p = ref 1 in
  for j = 1 to k do
    p := 2 * !p ;
    for i = 0 to m - !p do
      let iprime = i + !p / 2 in
      M.(i).(j) <- min M.(i).(j-1) M.(iprime).(j-1)
    done
  done ;;
```

La référence **p** calcule la valeur de 2^j.

Question 9. Posons $k = $ `log2`$(j-i+1)$. Alors $2^k \leqslant j-i+1 < 2^{k+1}$ donc $i \leqslant j+1-2^k < i+2^k \leqslant j+1$.
Ainsi, $[\![i,j]\!] = \left[\!\left[i, i+2^k\right[\!\right[\cup \left[\!\left[j+1-2^k, j+1\right[\!\right[$ et la fonction doit retourner le minimum de `M.`(i)`.`(k) et de `M.`$(j+1-2^k)$`.`(k), ce qui se réalise en coût constant si on néglige le coût du calcul de k et de 2^k.

Le calcul de 2^k peut se faire à l'aide de l'algorithme d'exponentiation rapide ou mieux en réécrivant la fonction **`log2`** pour que cette fonction retourne le couple $(k, 2^k)$:

```
let rec log2 = function
  | 1 -> 0, 1
  | n -> let k, p = log2 (n/2) in k+1, 2*p ;;
```

On obtient alors :

```
let minimum i j =
  let k, p = log2 (j - i + 1) in
  min M.(i).(k) M.(j+1-p).(k) ;;
```

Question 10. Compte tenu de la question 6, il reste à définir :

```
let ppac2 i j =
  let ki = index.(i) and kj = index.(j) in
  minimum (min ki kj) (max ki kj) ;;
```

Partie III. Opérations sur les bits des entiers primitifs

Remarque. Les opérateurs binaires sur le type *int* existent sous forme infixe : il s'agit des opérateurs **`land`** (*logical and*), **`lor`** (*logical or*) et **`lxor`** (*logical xor*). Il en est de même des opérations de décalage gauche (**`lsl`**, *logical shift left*) et droit (**`lsr`**, *logical shift right*). Si on souhaite tester les fonctions écrites on peut donc définir :

```
let et_bits (x, y) = x land y ;;
let ou_bits (x, y) = x lor y ;;
let ou_excl_bits (x, y) = x lxor y ;;
```

```
let decalage_gauche (x, k) = x lsl k ;;
let decalage_droite (x, k) = x lsr k ;;
```

Question 11. L'entier i est caractérisé par l'encadrement : $2^i \leqslant x < 2^{i+1}$; il s'agit donc de la même fonction que **`log2`**, sauf que cette fois nous allons l'écrire à l'aide des opérations binaires sur le type *int* :

```
let rec bit_fort = function
  | 1 -> 0
  | n -> 1 + bit_fort (decalage_droite (n, 1)) ;;
```

Question 12. Supposons posséder un tableau **`T`** contenant les valeurs de **`bit_fort`** pour les entiers compris entre 1 et 255. Ce tableau peut par exemple être construit à l'aide du script :

```
let T = make_vect 256 0 ;;
let p = ref 1 in
    for j = 0 to 7 do
      for i = !p to 2 * !p - 1 do
        T.(i) <- j
      done ;
      p := 2 * !p
    done ;;
```

(Notez qu'on peut remplacer **`2 * !p`** par **`decalage_gauche (!p, 1)`** pour rester dans l'esprit de cette partie.)

Considérons alors un entier x dans $[\![1, 2^{31}-1]\!]$. Cet entier se code sur 30 bits. Notons y_1 et y_2 les deux entiers codés sur 15 bits tels que $x = 2^{15}y_1 + y_2$:

x : | ← y_1 → | ← y_2 → |

puis posons $y = \begin{cases} y_1 & \text{si } y_1 \neq 0 \\ y_2 & \text{sinon} \end{cases}$. Notons ensuite z_1 et z_2 les entiers codés respectivement sur 7 et 8 bits tels que $y = 2^8 z_1 + z_2$.

$$y : \boxed{\longleftarrow \; z_1 \; \longrightarrow}\boxed{\longleftarrow \; z_2 \; \longrightarrow}$$

Posons enfin $b = \begin{cases} \texttt{T.}(z_1) & \text{si } z_1 \neq 0 \\ \texttt{T.}(z_2) & \text{sinon} \end{cases}$. Alors $\texttt{bit\_fort}(x) = \begin{cases} 23+b & \text{si } y_1 \neq 0 \text{ et } z_1 \neq 0 \\ 15+b & \text{si } y_1 \neq 0 \text{ et } z_1 = 0 \\ 8+b & \text{si } y_1 = 0 \text{ et } z_1 \neq 0 \\ b & \text{si } y_1 = 0 \text{ et } z_1 = 0 \end{cases}$. D'où la fonction :

```
let bit_fort x =
  let y1 = decalage_droite (x, 15) in
  let y = if y1 = 0 then x else y1 in
  let z1 = decalage_droite (y, 8) in
  let b = if z1 = 0 then T.(y) else T.(z1) in
  match y1, z1 with
  | 0, 0 -> b
  | 0, _ -> 8 + b
  | _, 0 -> 15 + b
  | _, _ -> 23 + b ;;
```

Partie IV. Cas particulier d'un arbre binaire complet

Question 13. On peut observer que la numérotation décrite munit l'arbre **A** d'une structure d'arbre binaire de recherche : le numéro de chaque nœud est supérieur aux numéros de tous les descendants de son fils gauche et inférieur aux numéros de tous les descendants de son fils droit. Dès lors, il suffit d'un parcours infixe de **A** pour numéroter par ordre croissant chacun des nœuds.

```
let remplir_B () =
  let p = ref 1 in
  let rec aux = function
    | Noeud (x, [])      -> B.(x) <- !p ; Binv.(!p) <- x ; incr p
    | Noeud (x, [a; b]) -> aux a ;
                           B.(x) <- !p ; Binv.(!p) <- x ; incr p ;
                           aux b
    | Noeud (_, _ )      -> failwith "arbre non binaire"
  in aux A ;;
```

Question 14. Notons a le PPAC de i et j. Puisque $a \neq i$ et $a \neq j$, on peut supposer sans perte de généralité que i est un descendant du fils gauche de a et j un descendant du fils droit.
Posons $\mathrm{B}(a) = (x_d \cdots x_{h+1} 10 \cdots 0)_2$, où h est la hauteur de a. Alors :

$$\mathrm{B}(i) = (x_d \cdots x_{h+1} 0\,?\ldots?\,10 \cdots 0)_2 \qquad \text{et} \qquad \mathrm{B}(j) = (x_d \cdots x_{h+1} 1\,?\ldots?10 \cdots 0)_2$$

donc $x = \texttt{ou\_excl\_bits}(\mathrm{B}(i), \mathrm{B}(j)) = (0 \cdots 01\,?\ldots?)_2$ et $k = \texttt{bit\_fort}(x) = h$.
L'entier k représente donc la hauteur de l'ancêtre commun à i et j. Pour obtenir le numéro de a, il suffit de considérer les $d+1-k$ premiers bits de $\texttt{ou\_bits}(\mathrm{B}(i), \mathrm{B}(j))$ et de les faire suivre de k 0, ce qu'on obtient par deux décalages successifs à droite puis à gauche.

Question 15. Tout ceci nous donne :

```
let ppac3 i j =
  if appartient i j then j
  else if appartient j i then i
  else
    let k = bit_fort (ou_excl_bits (B.(i), B.(j))) in
    let a = decalage_gauche (decalage_droite (ou_bits (B.(i), B.(j)), k), k) in
    Binv.(a) ;;
```

Partie V. Application

Question 16. Notons $\mathrm{C}(n)$ le coût de la construction de l'arbre binaire à partir d'un segment `T`$[i..j]$ de longueur n. Le coût du calcul du minimum du tableau est en $\mathrm{O}(n)$ donc on dispose de la relation :

$$\mathrm{C}(n) = \mathrm{C}(p) + \mathrm{C}(q) + \mathrm{O}(n) \quad \text{avec} \quad p + q = n - 1 \text{ et } \mathrm{C}(0) = 0.$$

Cette relation est semblable à celle de l'algorithme de tri rapide, dont le coût dans le pire des cas est un $\mathrm{O}(n^2)$. Nous allons le justifier rigoureusement, à partir de l'inégalité : $\mathrm{C}(n) \leqslant \mathrm{C}(p) + \mathrm{C}(q) + \beta n$, où β est une constante.
Supposons avoir trouvé une constante $\alpha > 0$ (qui sera déterminée plus loin) telle que pour tout entier $p < n$ on ait $\mathrm{C}(p) \leqslant \alpha p^2$. Alors :

$$\mathrm{C}(n) \leqslant \alpha(p^2 + q^2) + \beta n = \alpha(p^2 + (n-1-p)^2) + \beta n \leqslant \alpha(n-1)^2 + \beta n = \alpha n^2 + (\beta - 2\alpha)n + \alpha$$

car la fonction $x \mapsto x^2 + (n-1-x)^2$ est maximale aux extrémités de l'intervalle $[0, n-1]$.
On constate alors qu'il suffit de choisir $\alpha > \beta$ pour terminer ce calcul : $\mathrm{C}(n) \leqslant \alpha n^2 - \alpha n + \alpha \leqslant \alpha n^2$ et conclure par récurrence.

Question 17. L'indication de l'énoncé suggère lors de la construction de représenter l'arbre (2) à l'aide du type *arbre list* sous la forme : `[Noeud (vk, [tk]); ... ; Noeud (v2, [t2]); Noeud (v1, [t1])]`.
Une fois la construction achevée, il va donc être nécessaire de transformer cette représentation en un élément de type *arbre*, ce qui va être réalisé par la fonction :

```
let reconstruire lst =
  let rec aux acc = function
    | []              -> hd acc
    | Noeud (v, t)::q -> aux [Noeud (v, t @ acc)] q
  in aux [] lst ;;
```

Comme il n'est pas prévu de pouvoir représenter l'arbre vide dans le type *arbre*, l'accumulateur est représenté par le type *arbre list* , mais cette liste ne comporte durant toute l'exécution que 0 ou 1 élément.

La fonction d'insertion d'un nœud v se réalise ainsi :

```
let rec insere v lst =
  let rec aux acc = function
    | []                          -> Noeud (v, [])::acc
    | Noeud (w, t)::q when v < w -> aux [Noeud (w, t @ acc)] q
    | lst                         -> Noeud (v, acc)::lst
    in aux [] lst ;;
```

Dans les deux fonctions ci-dessus nous utilisons la concaténation `@`, mais celle-ci est de coût constant puisqu'à chaque fois la liste `t` ne comporte que 0 ou 1 élément.
La construction de l'arbre insère un par un les éléments du tableau `T` puis reconstruit l'arbre final :

```
let construire_A T =
  let n = vect_length T in
  let rec aux acc = function
    | i when i = n -> reconstruire acc
    | i            -> aux (insere T.(i) acc) (i+1)
  in aux [] 0 ;;
```

Question 18. À chaque étape de la construction, notons ℓ_i la longueur de la branche droite de l'arbre après insertion de `T.(i)` (dans l'exemple de l'énoncé on a donc $\ell_{i-1} = k$ et $\ell_i = j + 1$).
Le coût de la fonction `insere` appliquée à `T.(i)` est alors un $\mathrm{O}(1)$ si $\ell_i = \ell_{i-1} + 1$, un $\mathrm{O}(\ell_{i-1} - \ell_i)$ sinon.
Puisque $\displaystyle\sum_{i=0}^{n-2} (\ell_{i-1} - \ell_i) = 1 - \ell_{n-1} \in [\![1-n, 0]\!]$, le coût total des insertions successives est un $\mathrm{O}(n)$.
Le coût de la reconstruction est ensuite un $\mathrm{O}(\ell_{n-1}) = \mathrm{O}(n)$, donc le coût total de cette construction est un $\mathrm{O}(n)$.