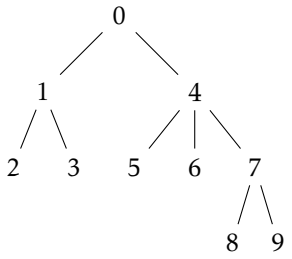
Plus proche ancêtre commun (X 2010)

Durée : 4 heures

Ce problème s'intéresse à la question suivante : étant donné un arbre et deux nœuds dans cet arbre, quel est le plus proche ancêtre commun de ces deux nœuds ? Une solution efficace à ce problème a de nombreuses applications, notamment en bio-informatique. Ainsi, dans l'arbre



(1)

le plus proche ancêtre commun des nœuds 5 et 8 est le nœud 4, et celui des nœuds 3 et 7 est la racine 0. De manière générale, on se donne un arbre quelconque, sur lequel on s'autorise un pré-traitement, puis on souhaite calculer le plus efficacement possible le plus proche ancêtre commun pour des couples de nœuds dans cet arbre.

Les arbres que l'on considère ici ont des nœuds étiquetés par des entiers distincts. Chaque nœud a un nombre fini de descendants immédiats, appelés ses *fils*. Un nœud sans descendant est appelé une *feuille*. Dans l'exemple ci-dessus, le nœud 4 a trois fils, a savoir 5, 6 et 7, et les feuilles sont 2, 3, 5, 6, 8 et 9. Le *sous-arbre* du nœud i est defini récursivement de la maniere suivante : c'est l'ensemble de nœuds contenant i et l'union de tous les sous-arbres des fils de i. On écrira simplement « sous-arbre i » pour désigner le sous-arbre du nœud i. Si un nœud j appartient au sous-arbre i, on dit que i est un *ancêtre* de j. En particulier, chaque nœud est son propre ancêtre.

Le plus proche ancêtre commun de deux nœuds i et j, noté $\mathrm{PPAC}(i,j)$, est defini formellement de la manière suivante :

- si i est un ancêtre de j, alors $\mathrm{PPAC}(i,j) = i$;
- symétriquement, si j est un ancêtre de i, alors $\mathrm{PPAC}(i,j) = j$;
- sinon, $\mathrm{PPAC}(i,j)$ est l'unique nœud a possédant au moins deux fils distincts f_1 et f_2 tels que i appartient au sous-arbre f_1 et j au sous-arbre f_2.

Les arbres seront représentés en Caml de la manière suivante :

```
type arbre = Noeud of int * arbre list ;;
```

Dans l'ensemble de ce problème, on se fixe une constante entière n, avec $n \geqslant 1$, et un arbre `A` contenant n nœuds numérotés de 0 à $n-1$. On suppose en outre que cette numérotation vérifie la propriété suivante :

pour tout i, les nœuds du sous-arbre i portent des numéros consécutifs à partir de i. (P)

On note que la racine de `A` est donc nécessairement le nœud 0.

La partie I propose une solution simple mais inefficace pour le problème du plus proche ancêtre commun. La partie II propose ensuite une solution plus efficace, avec un pré-traitement en $O(n \log n)$ et une réponse en temps $O(\log n)$. Les parties III et IV étudient le cas particulier des arbres binaires complets. Enfin, la partie V étudie une application du calcul du plus proche ancêtre commun.

Les parties peuvent être traitées indépendamment. Mais attention, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes. On rappelle que les tableaux sont indexés à partir de 0 et que la notation `t.(i)` est utilisée pour désigner l'élément d'indice i du tableau `t`. On appelle segment et on note `t`$[i..j]$ le sous-tableau du tableau `t` défini en considérant les indices compris entre i et j au sens large.

Partie I. Une solution simple

Dans cette partie, on considère une solution simple qui calcule le plus proche ancêtre commun en temps $O(n)$.

Question 1. On suppose avoir declaré un tableau global **`taille`** de taille n. Écrire une procédure **`remplir_taille`** qui stocke dans **`taille.(i)`** le nombre de nœuds du sous-arbre i, pour tout i entre 0 et $n-1$. On garantira une complexité $O(n)$.

```
remplir_taille : unit -> unit
```

Par la suite, on suppose avoir appelé cette procédure.

Question 2. Écrire une fonction **`appartient`** qui prend en arguments deux nœuds i et j et détermine, en temps constant, si i appartient au sous-arbre j.

```
appartient : int -> int -> bool
```

Question 3. Écrire une fonction **`ppac1`** qui prend en arguments deux nœuds i et j et determine $\mathrm{PPAC}(i,j)$ en temps $O(n)$.

```
ppac1 : int -> int -> int
```

Partie II. Une solution plus efficace

Dans cette partie, on va effectuer un pré-traitement de l'arbre **`A`** qui permettra de déterminer ensuite en temps logarithmique le plus proche ancêtre commun pour tout couple de nœuds.

On commence par construire une séquence de nœuds en effectuant un *tour Eulérien* de l'arbre **`A`** à partir de sa racine. D'une maniere générale, le tour Eulérien à partir du nœud i est defini comme agissant sur une sequence résultat, construite de la maniere suivante :

1. ajouter le nœud i à la fin de la séquence résultat ;
2. pour chaque fils j de i,
   (a) effectuer un tour Eulérien à partir de j,
   (b) ajouter le nœud i à la fin de la séquence résultat.

Ainsi, sur l'exemple de l'arbre (1), on obtient la séquence suivante :

$$0, 1, 2, 1, 3, 1, 0, 4, 5, 4, 6, 4, 7, 8, 7, 9, 7, 4, 0$$

Question 4. Montrer que le tour Eulérien de **`A`** contient exactement $2n-1$ éléments. Par la suite, on appelle m cette valeur et on suppose avoir declaré un tableau global **`euler`** de taille m destiné à contenir le resultat du tour Eulérien.

Question 5. On suppose avoir déclaré un tableau global **`index`** de taille n. Écrire une procédure **`remplir_euler`** qui remplit le tableau **`euler`** avec le résultat du tour Eulérien de **`A`** et, dans le même temps, le tableau **`index`** de telle sorte que **`euler[index[`**i**`]]`** $= i$ pour tout i entre 0 et $n-1$ (s'il existe plusieurs valeurs possibles pour **`index[`**i**`]`**, on pourra choisir arbitrairement).

```
remplir_euler : unit -> unit
```

Par la suite, on suppose avoir appelé cette procédure.

Question 6. Montrer que le plus proche ancêtre commun de i et j dans **`A`** est égal au plus petit élément du tableau **`euler`** compris entre les indices **`index[`**i**`]`** et **`index[`**j**`]`**.

Question 7. Écrire une fonction **`log2`** qui prend en argument un entier n, avec $n \geqslant 1$, et calcule le plus grand entier k tel que $2^k \leqslant n$.

```
log2 : int -> int
```

Question 8. On pose $k =$ `log2(`m`)`. On suppose avoir déclaré une matrice globale `M` de taille $m \times (k+1)$. Écrire une procédure `remplir_M` qui remplit la matrice `M` de telle sorte que, pour tout $0 \leqslant j \leqslant k$ et tout $0 \leqslant i \leqslant m - 2^j$, on ait

$$\texttt{M}[i][j] = \min_{i \leqslant l < i+2^j} \texttt{euler}[l]$$

On garantira une complexité $O(n \log n)$, c'est-à-dire au plus proportionnelle à la taille de la matrice `M`.

```
remplir_M : unit -> unit
```

Par la suite, on suppose avoir appelé cette procédure.

Question 9. Écrire une fonction `minimum` qui prend en arguments deux entiers i et j, avec $0 \leqslant i \leqslant j < m$, et qui détermine le plus petit élément du segment `euler`$[i..j]$ en temps logarithmique.

```
minimum : int -> int -> int
```

Question 10. Écrire une fonction `ppac2` qui prend en arguments deux nœuds i et j et qui détermine $\mathrm{PPAC}(i,j)$ en temps logarithmique.

```
ppac2 : int -> int -> int
```

Partie III. Opérations sur les bits des entiers primitifs

Dans cette partie, on introduit des notations et des fonctions qui seront utiles pour la partie IV. Il s'agit de fonctions opérant sur la représentation binaire des entiers primitifs (type `int` de Caml). On ne considère ici que des entiers positifs ou nuls, s'écrivant en binaire sur un nombre de bits N dont la valeur est non précisée et non connue (mais inférieure ou egale à 30). Un entier x s'écrivant en binaire sur N bits est noté $(x_{\mathrm{N}-1} \cdots x_1 x_0)_2$ où chaque bit x_i vaut 0 ou 1. Les chiffres les moins significatifs sont écrits à droite, de sorte que la valeur de x est donc $\sum_{i=0}^{\mathrm{N}-1} x_i 2^i$.

On suppose que le langage fournit les trois operations `et_bits`, `ou_bits` et `ou_excl_bits` calculant respectivement les operations ET, OU et OU-exclusif sur les representations binaires de deux entiers, en temps constant. Plus précisément, si $x = (x_{\mathrm{N}-1} \cdots x_1 x_0)_2$ et $y = (y_{\mathrm{N}-1} \cdots y_1 y_0)_2$, alors `et_bits(`$x$`, `$y$`)` est l'entier $z = (z_{\mathrm{N}-1} \cdots z_1 z_0)_2$ defini par $z_i = \mathrm{ET}(x_i, y_i)$ pour tout i ; de même pour les operations `ou_bits` et `ou_excl_bits`. On rappelle que les operations ET, OU et OU-exclusif sont définies par les tables de vérité suivantes :

ET

	0	1
0	0	0
1	0	1

OU

	0	1
0	0	1
1	1	1

OU-exclusif

	0	1
0	0	1
1	1	0

On suppose que le langage fournit également deux opérations `decalage_gauche` et `decalage_droite` décalant respectivement les bits d'un entier vers la gauche et vers la droite, en insérant des bits 0 respectivement à droite et à gauche. Plus précisément, si $x = (x_{\mathrm{N}-1} \cdots x_1 x_0)_2$ et si $0 \leqslant k \leqslant \mathrm{N}$ alors

$$\texttt{decalage\_gauche(}x\texttt{, }k\texttt{)} = (x_{\mathrm{N}-1-k} \cdots x_1 x_0 \underbrace{0 \cdots 0}_{k})_2$$

$$\texttt{decalage\_droite(}x\texttt{, }k\texttt{)} = (\underbrace{0 \cdots 0}_{k} x_{\mathrm{N}-1} \cdots x_{k+1} x_k)_2$$

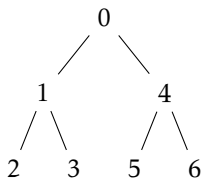
Question 11. Écrire une fonction `bit_fort` qui prend en argument un entier $x = (x_{\mathrm{N}-1} \cdots x_1 x_0)_2$, supposé non nul, et détermine le plus grand indice i tel que $x_i = 1$, c'est-à-dire la position du bit le plus significatif de x. On garantira une complexité $O(\mathrm{N})$.

```
bit_fort : int -> int
```

Question 12. Pour plus d'efficacité, on peut pré-calculer et ranger dans un tableau les valeurs de **`bit_fort`** pour les 256 premiers entiers. Écrire une nouvelle version de **`bit_fort`** qui exploite cette idee pour n'effectuer pas plus de deux décalages. (On rappelle qu'on a supposé $N \leqslant 30$.)

Partie IV. Cas particulier d'un arbre binaire complet

Dans cette partie, on considère le problème du plus proche ancêtre commun dans le cas particulier où l'arbre **`A`** est un arbre binaire complet, c'est-a-dire que tout nœud de **`A`** a exactement zero ou deux fils et toutes les feuilles de **`A`** sont à la meme distance de la racine. (On continue cependant d'utiliser le meme type *arbre*.) Voici un exemple d'arbre binaire complet de hauteur 2 :

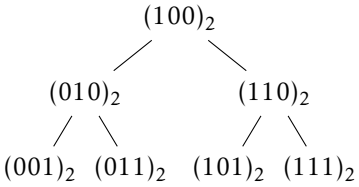


D'une maniere générale, si on note d la hauteur de **`A`**, alors **`A`** possède 2^d feuilles et $n = 2^{d+1} - 1$ nœuds au total. La hauteur d'un nœud dans l'arbre **`A`** est définie comme sa distance aux feuilles. Dans l'exemple ci-dessus, les feuilles 2, 3, 5, 6 ont pour hauteur 0, les nœuds 1 et 4 ont pour hauteur 1 et la racine 0 a pour hauteur 2.

L'idée consiste alors à associer à chaque nœud i un entier $\mathrm{B}(i)$ dont la representation en binaire sur $d+1$ bits encode sa position dans l'arbre. On procède ainsi : le chemin qui mene de la racine au nœud est representé par une suite de 0 et de 1, avec la convention que 0 représente un déplacement vers le sous-arbre gauche et 1 un déplacement vers le sous-arbre droit. Pour un nœud de hauteur h, on obtient donc un chemin de longueur $d-h$, soit $x_d \cdots x_{h+1}$. À ce chemin, on concatene alors à droite la représentation binaire de 2^h, c'est-a-dire un bit 1 suivi de h bits 0. Au final, on a donc pour tout nœud i de hauteur h dans **`A`** :

$$\mathrm{B}(i) = (\underbrace{x_d \cdots x_{h+1}}_{\text{chemin de 0 à } i}\ \underbrace{1\,0 \cdots 0}_{h})_2$$

On note que le chemin est vide pour la racine et que le suffixe de zeros est vide pour les feuilles.
Pour l'arbre ci-dessus, on obtient les valeurs suivantes de $\mathrm{B}(i)$, données en binaire :



On note que les $\mathrm{B}(i)$ prennent toutes les valeurs entre 1 et n, et qu'il y a donc bijection entre les nœuds et les valeurs que leur associe la fonction B.

Question 13. On suppose avoir declaré deux tableaux globaux, **`B`** de taille n et **`Binv`** de taille $n+1$. Écrire une procédure **`remplir_B`** qui remplit le tableau **`B`** avec les valeurs de B et le tableau **`Binv`** avec les valeurs de B^{-1} (l'élément d'indice 0 de **`Binv`** étant inutilisé). On garantira une complexité $\mathrm{O}(n)$.

```
remplir_B : unit -> unit
```

Par la suite, on suppose avoir appelé cette procédure.

Question 14. Soient i et j deux nœuds de **`A`** tels que i n'est pas un ancêtre de j et j n'est pas un ancêtre de i. Soit x le résultat de **`ou_excl_bits(`**$\mathrm{B}(i)$**`,`** $\mathrm{B}(j)$**`)`**, puis k le résultat de **`bit_fort(`**x**`)`**. Que représente k ? Comment en déduire la valeur de $\mathrm{B}(a)$, où a est le plus proche ancêtre commun de i et j dans **`A`** ?

Question 15. Déduire de la question précédente une fonction **`ppac3`** qui prend en arguments deux nœuds i et j et qui détermine $\mathrm{PPAC}(i,j)$ en temps constant. On prendra soin de traiter différemment le cas où i est un ancêtre de j ou j un ancêtre de i ; on pourra réutiliser à cet effet la fonction **`appartient`** de la question 2.

```
ppac3 : int -> int -> int
```

Partie V. Application

Dans cette partie, on considère une application du calcul du plus proche ancêtre commun au problème suivant : étant donné un tableau `T` de n entiers, on souhaite calculer, pour des paires d'indices i et j tels que $0 \leqslant i \leqslant j < n$, l'élément minimum du segment $\texttt{T}[i..j]$. Comme pour le calcul du plus proche ancêtre commun, on s'autorise un pré-traitement du tableau `T` permettant ensuite de répondre en temps constant pour tout segment. L'idée est de construire un arbre `A` contenant les n éléments de `T` et de se ramener au problème du plus proche ancêtre commun.

D'une manière générale, on définit un arbre binaire à partir d'un segment non vide $\texttt{T}[i..j]$ en procédant récursivement de la manière suivante :

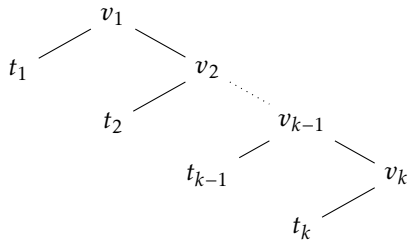
1. si $i = j$, l'arbre est réduit a la feuille $\texttt{T.(}i\texttt{)}$;
2. sinon, sa racine est le plus petit élément de $\texttt{T}[i..j]$; soit $\texttt{T.(}m\texttt{)}$ cet élément, avec $i \leqslant m \leqslant j$ (s'il y a plusieurs valeurs possibles de m, on choisit arbitrairement). On distingue alors trois cas :
   (a) si $m = i$, on construit un unique sous-arbre à partir de $\texttt{T}[m+1..j]$,
   (b) si $m = j$, on construit un unique sous-arbre à partir de $\texttt{T}[i..m-1]$,
   (c) sinon, on construit deux sous-arbres, respectivement à partir de $\texttt{T}[i..m-1]$ et de $\texttt{T}[m+1..j]$.

L'arbre `A` est alors defini en partant du segment complet $\texttt{T}[0..n-1]$. On note que les nœuds de `A` ont directement pour valeurs les éléments de `T`, i.e. on ne se soucie pas ici de numéroter les nœuds de `A` de 0 à $n-1$ afin de verifier la propriété (P).

> On admet le résultat suivant : le plus proche ancêtre commun des nœuds correspondant à i et j dans `A` est égal à l'élément minimum du segment $\texttt{T}[i..j]$.

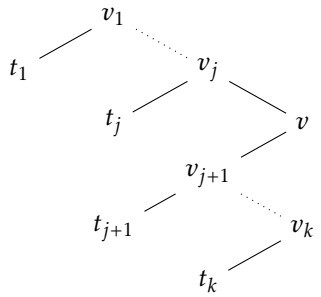
Question 16. Donner la complexité dans le pire des cas de l'algorithme récursif ci-dessus construisant `A` à partir de `T`, en fonction de n (en supposant le minimum calculé en temps linéaire).

On propose un algorithme plus efficace pour construire l'arbre `A`. Cet algorithme construit incrementalement l'arbre correspondant aux elements de $\texttt{T}[0..i]$, pour i allant de 0 à $n-1$. Initialement, l'arbre est réduit à l'unique nœud $\texttt{T.(}0\texttt{)}$. Supposons avoir déjà construit l'arbre correspondant aux i premiers elements de `T`, c'est-à-dire au segment $\texttt{T}[0..i-1]$. Cet arbre a la forme suivante



(2)

avec une « branche droite » formée de k nœuds tels que $v_1 \leqslant v_2 \leqslant \cdots \leqslant v_{k-1} \leqslant v_k$. Chaque arbre t_i est éventuellement vide et ne contient que des éléments strictement plus grands que v_i. Pour insérer l'élément suivant, soit $v = \texttt{T.(}i\texttt{)}$, on cherche la position de v dans la liste $v_1, v_2, \ldots, v_k$, c'est-à-dire le plus grand j tel que $v_j \leqslant v < v_{j+1}$ (si $v < v_1$ on pose $j = 0$ et si $v_k \leqslant v$ on pose $j = k$). On crée alors un nouveau nœud v que l'on insère à la place de v_{j+1} et le sous-arbre v_{j+1} est accroché sous le nœud v. On obtient le nouvel arbre



dont la nouvelle branche droite est donc $v_1, \ldots, v_j, v$. Si $j = k$, alors on ajoute v comme fils droit de v_k, et la nouvelle branche droite est simplement l'ancienne étendue par v (à la fin). On ne demande pas de montrer la correction de cet algorithme.

Question 17. Écrire une fonction **`construire_A`** qui construit l'arbre **`A`** à partir du tableau **`T`** en suivant ce nouvel algorithme.

```
construire_A : int array -> arbre
```

Indication : on pourra représenter la branche droite par une liste d'arbre (du type *arbre list* en CAML), dans l'ordre inverse, c'est-à-dire $v_k, \ldots, v_1$. Chacun de ces arbres a une racine v_i et au plus un fils t_i.

Question 18. Montrer que la complexité de cet algorithme est $O(n)$.

Note : En 1984, Harel et Tarjan ont montré qu'il existe un pré-traitement de complexité lineaire sur un arbre qui permet d'obtenir ensuite le plus proche ancêtre commun en temps constant. On a donc le même résultat pour le problème du minimum du segment.

♦ ♦
♦