Corrigé de l'épreuve d'informatique 2010
MP option informatique

I. Une solution simple

Q1 La fonction auxiliaire `remplir` s'applique à une liste d'arbres : elle met à jour les champs `taille` des nœuds contenus dans les arbres de la liste et renvoie le nombre de ces nœuds. La fonction principale se contente d'appliquer `remplir` à la liste contenant A pour unique élément (la dernière instruction évite le renvoi final de la taille de l'arbre).

```
let remplir_taille() =
   let rec remplir = function
      []                    -> 0
     |Noeud(i,fils)::queue -> t.(i)<- 1+remplir fils;
                              t.(i)+remplir queue in
   remplir [A];
   ();;
```

Q2 Le nœud j contient les nœuds $j, j+1, \ldots, j+t.(j)-1$. Il contient donc le nœud i si et seulement si $j \leq i < j + t.(j)$:

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

Q3 PPAC(i,j) est le plus grand entier k tel que i et j appartiennent à k. On peut donc calculer cet élément en testant l'appartenance de i et j à k pour k décroissant à partir de la valeur $\min(i,j)$:

```
let ppac1 i j = let k = ref (min i j) in
   while not(appartient i k && appartient j k) do
         k := (!k)-1
   done;
   !k;;
```

Comme i et j appartiennent à 0, cette boucle se termine après au plus $1 + \min(i,j)$ calcul de coût constant : le temps de calcul est bien en $O(n)$.

II. Une solution plus efficace

Q4 Pour i compris entre 0 et $n-1$, le nombre d'occurence de i dans le tour Eulérien est égal à $1 + f_i$ où f_i est le nombre de fils du nœud i. Comme $\sum_{i=0}^{n-1} f_i = n-1$ (chaque nœud est un fils, excepté la racine), nous obtenons :

$$m = \sum_{i=0}^{n-1} (1 + f_i) = n + \sum_{i=0}^{n-1} f_i = 2n - 1.$$

Q5 La définition récursive du tour Eulérien permet d'écrire facilement la fonction auxiliaire `tour` :

- l'écriture dans la table `euler` se fait en utilisant une référence `r` sur le numéro de la "première case libre" de `Euler` (elle est initialisée à 0 et est incrémentée chaque fois qu'une case de `Euler` est remplie) ;
- on remplit la case `index.(i)` à la première insertion de i dans la séquence ;
- la liste des fils est parcourue par le biais de la référence `pile`.

```
let remplir_euler() =
   let r = ref 0 in
   let rec tour = function
      Noeud(i,fils) -> euler.(!r) <- i;
                       index.(i) <- (!r);
                       r := (!r)+1;
                       let pile = ref fils in
                       while !pile <> [] do
                          tour (hd(!pile));
                          pile := tl(!pile);
                          euler.(!r) <- i;
                          r := (!r)+1;
                       done in
   tour A;;
```

Q6 Le tour Eulérien est obtenu lors d'un parcours en profondeur de l'arbre, chaque passage par un nœud étant suivi de l'insertion du numéro du nœud dans la séquence. La portion de la table `Euler` comprise entre les indices $r_i =$ `index.`(i) et $r_j =$ `index.`(j) code donc un parcours du nœud i au nœud j et on se persuade facilement que le plus petit nœud présent sur ce parcours est $\mathrm{PPAC}(i,j)$: reste à donner une preuve précise. Nous pouvons, pour fixer les idées, supposer que $r_i \leq r_j$ et noter k le plus petit élément du tableau `euler` compris entre ces deux indices. Plusieurs cas peuvent alors être distingués :

- Premier cas : $r_i = r_j$; on a alors $i = j = k$ et $\mathrm{PPAC}(i,j) = k$.
- Deuxième cas : $r_i < r_j$ et $i = k$. Notons r l'indice de la dernière occurence de i dans la table `euler` : nous avons nécessairement $r_j < r$. En effet, ou bien $i = 0$ et $r = m - 1 > r_j$, ou bien i possède un père p et `euler.`$(r+1) = p < i$, ce qui impose $r > r_j$ par minimalité de k. Par définition du parcours Eulérien, la case d'indice r_j a été remplie pendant le parcours des fils de i : j est un descendant de i et $\mathrm{PPAC}(i,j) = i = k$.
- Le troisième cas $r_i < r_j$ et $j = k$ se traite comme le précédent, en échangeant les rôles de i et j et en considérant l'indice de la première occurence de j.
- Quatrième cas : $r_i < r_j$, $k < i$ et $k < j$. Pour la même raison que précédemment, k apparaît dans `euler` avant l'indice r_i et après l'indice r_j. Cela signifie que i et j ont été insérés en position r_i et r_j lors des parcours de deux fils différents du nœud k : k est donc bien le plus proche ancêtre commun de i et j.

Q7 L'écriture récursive est évidente :

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

Q8 Nous allons remplir les colonnes de `M` les une après les autres, en utilisant les relations :

$$\begin{aligned}
\forall i \in [\![0, m-1]\!],\ \texttt{M.}(i).(0) &= \texttt{euler.}(i) \\
\forall j \in [\![1, k]\!],\ \forall i \in [\![0, m-2^j]\!],\ \texttt{M.}(i).(j) &= \min_{i \le l < i+2^j} \texttt{euler.}(l) \\
&= \min\left(\min_{i \le l < i+2^{j-1}} \texttt{euler.}(l), \min_{i+2^{j-1} \le l < i+2^{j-1}+2^{j-1}} \texttt{euler.}(l) \right) \\
&= \min\left(\texttt{M.}(i).(j-1), \texttt{M.}(i+2^{j-1}).(j-1) \right)
\end{aligned}$$

Nous pouvons donc construire `M` en temps $O(n \ln n)$, puique chaque valeur $M.(i).(j)$ est calculée en temps constant (on crée une référence `puiss` qui contient 2^{j-1}).

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

Q9 La matrice `M` contient tous les minima des différents segments de `euler` de taille une puissance de 2. Une idée est de recouvrir `euler`$[i..j]$ par deux tels segments de même taille 2^k, l'un commençant en i et l'autre se terminant en j. Il faut donc calculer un entier k tel que

$$[\![i, i+2^k-1]\!] \cup [\![j-2^k+1, j]\!] = [\![i, j]\!],$$

i.e. tel que $2^k \le j-i+1$ et $j-2^k+1 \le i+2^k$. Ces deux conditions sont vérifiées par $k = \texttt{log2}(j-i)$. La méthode est donc la suivante : si $i = j$, on renvoie `euler`$.(i)$; sinon, on calcule $k = \texttt{log2}(j-i)$ et on renvoie le minimum des valeurs `M`$.(i).(k)$ et `M`$.(j-2^k+1).(k)$. On calcule 2^k par le biais de la fonction :

```
let puissance2 = function
   0 -> 1
   |n -> 2*(puissance2 (n-1));;
```

mais on pourrait également utiliser la fonction `decalage_gauche`, i.e. la fonction `lsl`, définie dans le III). Il y a toutefois un petit problème, puisque les calculs de k et de 2^k se font en un temps de l'ordre de k, i.e. de l'ordre $\log(j-i)$: je pense que l'on peut dire que ces temps de calcul sont constants, puisque majorés par le nombre N de bits utilisés pour représenter les entiers, mais l'énoncé n'a pas fait cette hypothèse à la question 8. Il y a peut-être une astuce qui permet d'éviter le calcul de k ...

```
let minimum i j =
   if i=j then
      euler.(i)
   else let k = log2 (j-i) in
      min M.(i).(k) M.(j-(puissance2 k)+1).(k);;
```

Q10 Il suffit de distinguer selon la position des entiers `index`$.(i)$ et `index`$.(j)$:

```
let ppac2 i j =
   if index.(i)>index.(j) then
      minimum index.(j) index.(i)
   else
      minimum index.(i) index.(j);;
```

III. Opérations sur les bits des entiers primitifs

On a par exemple :

```
let decalage_gauche x k = x lsl k;;

let decalage_droite x k = x lsr k;;
```

Nous noterons bf(n) le bit fort d'un entier non nul n.

Q11 On calcule le bit fort en décalant l'entier vers la droite jusqu'à trouver la valeur 1. Cela donne récursivement :

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

Q12 Nous commençons par créer la table `T` des 256 premières valeurs (pour i compris entre 1 et 255, `T.`$(i) =$ bf(i)) :

```
let T = make_vect 256 0;;

let j=ref 2 in   (* j contient la valeur 2^k *)
   for k=1 to 7 do
      for i=(!j) to (2*(!j)-1) do
         T.(i) <- k
      done;
      j := 2*(!j);
   done;;
```

Pour calculer le bit fort d'un entier n compris entre 1 et 2^{30}, nous commençons par écrire $n = a \times 2^{16} + b$ avec $a, b < 2^{16}$. Si a est non nul, bf$(n) = 16 +$ bf(a) ; sinon, bf$(n) =$ bf(b). La méthode fonctionne ensuite pour a ou b : on se ramène donc, après deux décalages, au calcul du bit fort d'un entier strictement plus petit que 10^8, qu'il suffit de lire dans la table `T`.

```
let bf n = match decalage_droite n 16 with
   0 -> begin
        match decalage_droite n 8 with
            0 -> T.(n)
           |c -> 8+T.(c)
        end
   a -> begin
        match decalage_droite a 8 with
            0 -> 16+T.(a)
           |c -> 24+T.(c)
        end;;
```

IV. Cas particulier d'un arbre binaire complet

Q13 Dans l'arbre `A`, le nœud i situé à la hauteur h a pour fils gauche $i+1$ et pour fils droits $i+2^h$. Il est donc possible de remplir `B` grâce à une fonction récursive `remplir_noeud` prenant en argument l'entier i, la hauteur h et l'entier codant le chemin de 0 à i. Celle-ci fonctionne de la façon suivante :

- on calcule $B(i)$ que l'on copie dans le tableau ;
- si $h \neq 0$, on applique récursivement la fonction aux fils de i : si `chemin` code le chemin de 0 à i, `decalage_gauchechemin` 1 et 1 + `decalage_gauchechemin` 1 codent respectivement les chemins de 0 aux fils gauche et droit de i.

Le code du chemin $x_d \dots x_{h+1}$ de 0 à i est en fait l'entier écrit $0 \dots 0 x_d \dots x_{h+1}$ sur N bits. Ainsi, des nœuds différents peuvent avoir le même code de chemin (comme par exemple, les nœuds de 0 à d, situés sur la branche la plus à gauche de l'arbre complet, qui sont codés par l'entier 0) : la distinction se fait entre ces différents chemins grâce à l'ajout des h occurrences du bit 0.

Cela donne :

```
let remplir_B() =
   let rec remplir i chemin h =
      let m=decalage_gauche (1+decalage_gauche chemin 1) h in    (* m=B(i) *)
         B.(i)<- m;
         Binv.(m)<- i;
         if h<>0 then
            begin
            remplir (i+1) (decalage_gauche chemin 1) (h-1);
            remplir (i+decalage_gauche 1 h) (1+decalage_gauche chemin 1) (h-1)
            end in
      remplir 0 0 d;;
```

Q14 Soit $a = \text{PPAC}(i,j)$. Notons h, h_i et h_j les hauteurs des nœuds a, i et j. En notant $x_d \dots x_{h+1}$ le chemin de 0 à a, les chemins de 0 à i et de 0 à j sont de la forme $x_d \dots x_{h+1} y_h \dots y_{h_i+1}$ et $x_d \dots x_{h+1} z_h \dots z_{h_j+1}$ avec $y_h \neq z_h$. Nous avons donc

$$\begin{cases} B(a) = x_d \dots x_{h+1} 10 \dots\dots\dots\dots\dots 0 \\ B(i) = x_d \dots x_{h+1} y_h \dots y_{h_i+1} 10 \dots\dots 0 \\ B(j) = x_d \dots x_{h+1} z_h \dots\dots z_{h_j+1} 10 \dots 0 \end{cases}$$

Comme ces trois codes sont de même longueur, $k = \text{bf}\,(\texttt{ou\_excl\_bits}(B(i), B(j)))$ est la position du dernier 1 du code de a : on a donc $k = h$. Ainsi, k est la hauteur de $\text{PPAC}(i,j)$ et on obtient le code $B(a)$ en décalant le code $B(i)$ à droite de $k+1$ bits (on retrouve le code du chemin de 0 à a) puis en concaténant à droite le mot $1\underbrace{0\dots0}_{k}$.

Q15 Il est assez curieux d'utiliser la fonction `appartient` de la question 2, alors que l'arbre `A` n'est pas construit. Il est par contre facile de tester si i est un ancêtre "strict" de j, puisqu'il suffit d'avoir $i < j < i - 1 + 2^h$ où h est la hauteur du nœud i, i.e. le nombre de zéro qui terminent l'écriture binaire de $B(i)$. L'analyse de cet algorithme est la suivante :

- si $i = j$, on, renvoie i ;
- si $i < j$, on teste si j est un descendant de i ; on calcule pour cela la hauteur h de i et on compare j à $i - 1 + 2^{h+1}$, qui est l'indice du premier nœud qui ne descend pas de i. Si j descend de i, on

renvoie i ; sinon, on applique la méthode détaillée à la question précédente : `chemin` est le chemin de 0 à PPAC(i, j), que l'on complète en ajoutant $10 \ldots 0$ pour obtenir $B.(a)$, qui nous donne a par le biais de la table `Binv` ;

- si $i > j$, on échange les rôles de i et j.

Cela donne :

```
let rec nb_zero n = match n mod 2 with          (* calcule le nombre de z\'eros   *)
   1 -> 0                                       (* qui terminent l'\'ecriture     *)
  |_ -> 1 + nb_zero (decalage_droite n 1);;     (* d'un entier non nul          *)

let ou_exc_bits i j =  i lxor j ;;

let ppac3 i j = match i=j,i<j with
   true,_     -> i                                        (* i=j *)
  |false,true -> let h=nb_zero B.(i) in
                 if j<i-1+decalage_gauche 1 (h+1) then
                    i                                     (* i est un anc\^etre de j *)
                 else
                    let k=bf (ou_exc_bits B.(i) B.(j)) in
                       let chemin = decalage_droite B.(i) (k+1) in
                       Binv.(decalage_gauche (1+decalage_gauche chemin 1) k)
  |false,false -> let h=nb_zero B.(j) in
                 if i<j-1+decalage_gauche 1 (h+1) then
                    j                                     (* j est un anc\^etre de i *)
                 else
                    let k=bf (ou_exc_bits B.(i) B.(j)) in
                       let chemin = decalage_droite B.(i) (k+1) in
                       Binv.(decalage_gauche (1+decalage_gauche chemin 1) k);;
```

V. Application

Q16 En notant $T(n)$ le temps de construction de l'arbre pour un tableau strictement croissant de taille n, nous avons $T(1) = 1$ et $T(n) = \Theta(n) + T(n-1)$: $T(n)$ est de l'ordre de n^2. D'autre part, l'algorithme effectue exactement n appels récursifs, puisqu'un nœud est créé à chaque appel. Au cours d'un appel, deux cas se présentent :

- si $i = j$, on crée simplement une feuille, ce qui prend un temps constant ;
- si $i < j$, on calcule le minimum de $T[i..j]$ et on crée une feuille

ce qui prend dans tous les cas un temps $O(n)$. L'algorithme s'exécute donc en un temps $O(n^2)$: le pire des cas donne bien un temps en $\Theta(n^2)$.

Q17 Il y a une difficulté dans la manipulation des arbres. Dans la construction de la branche droite, dans les arbres représentés par le graphique (2), il faut, quand un fils est vide, savoir si ce fils est le gauche ou le droit, ce que ne permet pas le type `arbre` (si un nœud a un fils unique, sa liste des fils est de longueur 1 et rien ne nous permet d'indiquer que ce fils unique est le fils gauche ou le fils droit). La subtilité, c'est que seul l'ordre des fils des nœuds contenant $v_1, \ldots, v_{k-1}$ nous intéresse dans la construction, puisqu'une fois l'arbre final obtenu, le PPAC de deux nœuds est défini sans faire la distinction entre les fils gauche et droit. En manipulant les branches droites, on peut donc se contenter du type `arbre`, puisque dans chaque sous-arbre de racine v_i, l'ordre dans les filiations ne joue aucun rôle (cette propriété se montre facilement par récurrence).

La méthode utilisée consiste à construire la "branche droite" en insérant les valeur `T.`(i) les unes après les autres, et de terminer par la construction de l'arbre en "concaténant" les nœuds de la branche droite finale. La fonction `inserer` prend en argument un entier v, une branche droite et un arbre `fg`, qui permet de construire récursivement ce qui sera le fils gauche du nœud d'étiquette v après insertion. Il faudrait alors appeler `inserer` avec pour `fg` un arbre vide, qui n'existe pas dans la structure `arbre`. J'ai donc choisi ici de remplacer `fg` par la liste `fils=[fg]` (qui devient la liste vide si `fg` est l'arbre vide). L'opérateur `@` ne pose pas de problème de temps d'exécution, puisqu'il ne sert qu'à concaténer des listes de longueur 0 ou 1.

```
let rec inserer  v branche fils = match branche with
    [] -> [Noeud(v,fils)]                                          (* cas a *)
   |Noeud(vk,tk)::queue -> if vk<=v then
                              Noeud(v,fils)::branche               (* cas b *)
                           else
                              inserer v queue [Noeud(vk,tk@fils)];;  (* cas c *)
```

La fonction `construire_branche` construit la branche droite à partir du tableau `T` :

```
let construire_branche t = let branche = ref [] in
   for i=0 to (vect_length t) -1 do
      branche := inserer t.(i) (!branche) []
   done;
   !branche;;
```

et la fonction `concatener` transforme une branche droite en arbre :

```
let rec concatener = function
    [a] -> a
   |Noeud(v1,fils)::queue -> Noeud(v1,(concatener queue)::fils);;
```

ce qui donne enfin :

```
let construire_A t = concatener (construire_branche t);;
```

Q18 Quand on applique `insertion`, chaque "opération élémentaire" contient la création d'un nœud. Lors de l'appel `construire _branche`, pour chaque j compris entre 0 et $n-1$, il y a création de 1 ou 2 nœud(s) contenant v_j. En effet, en insérant $T.(j)$, on crée le nœud `Noeud(T.(j),fils)` (cas a ou b). Ensuite, il est possible de créer un nouveau nœud contenant $T.(j)$ (cas c), mais cette opération ne se fera qu'une seule fois, puisque la branche ne contient plus d'arbre dont la racine est $T.(j)$. Plus précisément, le nombre de création de nœuds, i.e. le nombre d'opérations élémentaires, est égal à $2n - \ell$ où ℓ est la longueur de la branche droite finale. Comme ces opérations élémentaires prennent un temps constant, le coût total de `construire_branche()` est un $\Theta(n)$. La fonction `concatener` a ensuite un coût de l'ordre de la longueur de la branche droite, donc en $O(n)$: `construire_A()` a bien une complexité linéaire.