X 2010 — Option informatique

Partie I — Une solution simple

Question I.1
On propose le programme 1. On procède à un parcours suffixe de l'arbre, en commençant donc par affecter les tailles des feuilles. Cela garantit une complexité en $O(n)$.
Pour un nœud i, on commence, en ligne 8, par parcourir ses fils dont on peut supposer que les tailles sont maintenant calculées, ce qui permet de calculer la taille du nœud i, en calculant la somme des tailles de ses fils, augmentée d'une unité (pour le nœuds i lui-même), ce dont se charge l'appel à la fonction classique `it_list`, en ligne 9.
La fonction `consulte_taille` permet de trouver la taille d'un nœud déjà visité.
On pourrait regrouper les deux itérations (`do_list` et `it_list`) en une seule, mais au prix d'une moindre lisibilité.

```
1   let remplir_taille () =
2     let consulte_taille = function
3       | Noeud(i,_) -> taille.(i)
4     in
5     let rec parcourir = function
6       | Noeud(i,[]) -> taille.(i) <- 1
7       | Noeud(i,fils) ->
8          do_list (function a -> parcourir a) fils ;
9          taille.(i) <- it_list (fun m a -> consulte_taille a + m) 1 fils
10    in
11      parcourir A ;;
```

Programme 1 la fonction `remplir_taille`

Question I.2
La proprité (P) impose que pour tout nœud j, ses fils sont les nœuds $j+1$, $j+2$, $\dots$, $j+\texttt{taille}[j]-1$. On en déduit le programme 2.

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

Programme 2 la fonction `appartient`

Question I.3
On propose le programme 3.

```
14  let ppac1 i j =
15    let rec cherche k =
16      if appartient i k && appartient j k then k
17      else cherche (k-1)
18    in
19      if appartient i j then j
20      else if appartient j i then i
21      else cherche ((min i j) - 1) ;;
```

Programme 3 la fonction `ppac1`

On commence ici par éliminer les cas triviaux où l'un des nœuds est ancêtre de l'autre (lignes 19 et 20). Dans le cas général, le premier nœud candidat pour être ancêtre commun des nœuds i et j est le nœud k où $k = \min(i, j) - 1$. On appelle donc la fonction auxiliaire `cherche` qui décrémente au besoin son argument k jusqu'à trouver l'ancêtre commun.
Au pire, il faudra décrémenter jusqu'à atteindre la racine $k = 0$: la complexité est donc bien en $O(n)$.

Partie II — Une solution plus efficace

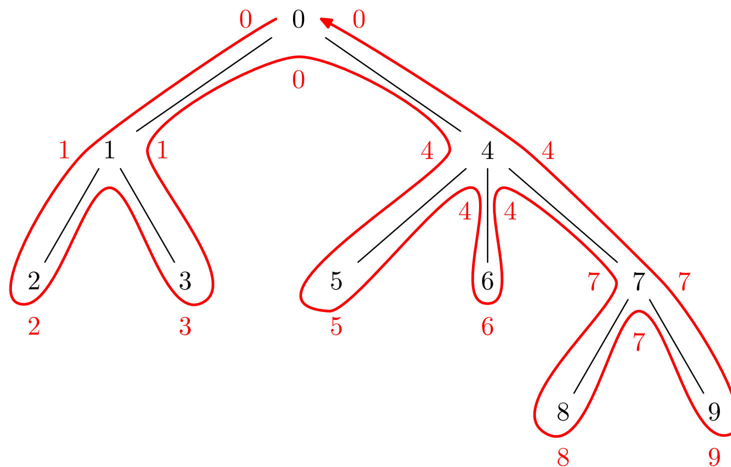

Figure 1 tour eulérien

Question II.4
On montre plus généralement que le tour eulérien du nœud i est de longueur $2 \times \texttt{taille}[i] - 1$.
Le parcours eulérien d'une feuille i est simplement i, de longueur $1 = 2 \times 1 - 1$.
Soit maintenant un nœud i dont on sait, par récurrence, que les tours eulériens à partir de chacun de ses fils j sont de longueur $2 \times \texttt{taille}[j] - 1$.
Le tour eulérien à partir de i a pour longueur :

$$1 + \sum_{j \text{ fils de } i} (2 \times \texttt{taille}[j] - 1 + 1) = 1 + 2 \sum_{j \text{ fils de } i} \texttt{taille}[j] = 1 + 2(\texttt{taille}[i] - 1) = 2 \times \texttt{taille}[i] - 1,$$

ce qui enclenche la récurrence et termine la démonstration.

Autre démonstration : pour tout i, le nœud i est écrit exactement $d_i + 1$ fois dans le tour eulérien de A, si on note d_i le degré du nœud i, c'est-à-dire le nombre de ses fils. La longueur du tour est donc égale à $n + \sum_i d_i$. Mais la somme des degrés compte tous les nœuds qui ont un père, c'est-à-dire tous sauf la racine de A, et on retrouve que la longueur du tour eulérien vaut $2n - 1$.

Question II.5
On propose le programme 4, page 3.
On utilise une fonction auxiliaire `remplir_depuis_index : int -> arbre -> arbre` dont l'appel avec pour arguments un entier k et un sous-arbre a de A effectue le remplissage des tableaux `euler` et `index` dans le tour eulérien à partir de a, l'indice k désignant l'index à partir duquel on en est du remplissage du tableau `euler`. Elle renvoie l'index de la première cas pas encore remplie à l'issue de ce parcours.
Le cas d'une feuille est facile : il est traité en ligne 25.
Le cas d'un nœud i commence par l'écriture de l'index (on a choisit ici d'inscrire dans le tableau `index` la première occurrence de i dans le tableau `euler`) et par écrire i dans le tableau `euler`, dans la première case disponible, la case k.
On continue le processus à partir de la case d'indice $k + 1$ par un appel de `it_list` parcourant chaque fils tour à tour, qui commence par (a) le parcours eulérien du fils en ligne 30, puis (b) par l'écriture de i à la suite, en ligne 31.
La fonction `ignore` se contente d'ignorer son argument, comme son nom l'indique.
La complexité de `remplir_euler` est clairement un $O(m) = O(n)$.

```
22  let ignore _ = () ;;

23  let remplir_euler () =
24    let rec remplir_depuis_index k = function
25      | Noeud(i,[]) -> index.(i) <- k ; euler.(k) <- i ; k+1
26      | Noeud(i,fils) ->
27         index.(i) <- k ; euler.(k) <- i ;            (* étape 1 *)
28         it_list                                      (* étape 2 *)
29           (fun k a ->
30              let k' = remplir_depuis_index k a in    (* sous-étape (a) *)
31                euler.(k') <- i ;                     (* sous-étape (b) *)
32                k'+1) (k+1) fils
33    in
34      ignore (remplir_depuis_index 0 A) ;;
```

Programme 4 la fonction `remplir_euler`

Question II.6

Dans le tour eulérien d'un nœud i, on n'écrit dans **euler** que des entiers $j \geqslant i$, puisqu'on visite le sous-arbre i. Autrement dit, entre deux occurrences de i dans **euler**, on ne trouve que des entiers $j \geqslant i$ désignant des nœuds dont i est un ancêtre.
Par conséquent, si j est un descendant de i, j apparaît dans **euler** entre deux occurrences de i, et $i = \text{PPAC}(i, j)$ est le plus petit entier du sous-tableau de **euler** entre les indices $\texttt{index}[i]$ et $\texttt{index}[j]$.
Le cas où i est un descendant de j se traite symétriquement de la même manière.

Considérons maintenant le cas restant : $\text{PPAC}(i, j) = a$, et, par exemple, i est dans le sous-arbre du fils gauche k de a, et j dans le sous-arbre du fils droit ℓ de a.
Le tour eulérien du nœud k s'écrit $E_k = \langle k, \ldots, k\rangle$: toutes les occurrences de i y figurent, en particulier celle d'indice $\texttt{index}[i]$, et le plus petit entier de ce tour est bien sûr k.
De même, le tour eulérien du nœud ℓ s'écrit $E_\ell = \langle \ell, \ldots, \ell\rangle$: toutes les occurrences de j y figurent, en particulier celle d'indice $\texttt{index}[j]$, et le plus petit entier de ce tour est bien sûr ℓ.
Le tour eulérien de A s'écrit alors $\langle \ldots, E_k, a, E_\ell, \ldots\rangle$ et a est bien le plus petit élément de **euler** entre les indices $\texttt{index}[i]$ et $\texttt{index}[j]$.

Question II.7

Aucune difficulté ici.

```
35  let rec log2 = function
36    | 1 -> 0
37    | n -> log2 (n/2) + 1 ;;
```

Programme 5 la fonction `log2`

Question II.8

La première colonne ($j = 0$) de la matrice `M` se calcule facilement : $\texttt{M}[i][0] = \texttt{euler}[i]$, pour $0 \leqslant i \leqslant m-1$.
Montrons que chaque terme de la colonne $j \geqslant 1$ se calcule comme un minimum de deux termes de la colonne $j-1$. En effet, si $0 \leqslant i \leqslant m - 2^j$:

$$\begin{aligned}\texttt{M}[i][j] &= \min \texttt{euler}[i..i+2^j-1]\\ &= \min\big(\min \texttt{euler}[i..i+2^{j-1}-1], \min \texttt{euler}[i+2^{j-1}..i+2^j-1]\big)\\ &= \min(\texttt{M}[i][j-1], \texttt{M}[i+2^{j-1}][j-1]).\end{aligned}$$

Chaque terme de la matrice `M` se calcule donc au prix d'un seul appel à la fonction **min**, à condition de travailler par colonne. La complexité sera donc bien $O(m \times (k+1)) = O(n \lg n)$.

```
38  let remplir_M () =
39    for i = 0 to m - 1 do M.(i).(0) <- euler.(i) done ;
40    let d = ref 1 in
41      for j = 1 to k do
42        for i = 0 to m - !d * 2 do
43         M.(i).(j) <- min M.(i).(j - 1) M.(i + !d).(j - 1)
44        done ;
45        d := 2 * !d ;
46      done ;;
```

Programme 6 la fonction `remplir_M`

On a fait en sorte qu'en ligne 42, `!d` vaut toujours 2^{j-1}.

Question II.9
Soit i et j tels que $0 \leqslant i \leqslant j \leqslant m-1$.
Notons $\delta = j - i + 1$ et $p = \lfloor \lg \delta \rfloor$, de sorte que $2^p \leqslant \delta < 2^{p+1}$.
Alors $i \leqslant j + 1 - 2^p \leqslant i + 2^p - 1 \leqslant j$, de sorte que $[i..j] = [i..i + 2^p - 1] \cup [j + 1 - 2^p..j]$ et donc $\min \texttt{euler}[i..j] = \min(\texttt{M}[i][p], \texttt{M}[j + 1 - 2^p][p])$.
On en déduit le programme 7, où la fonction `log_et_puissance`, appelée sur un entier $\delta \geqslant 1$ renvoie le couple (p, d) où $d = 2^p \leqslant \delta < 2^{p+1}$.

```
47  let minimum i j =
48    let rec log_et_puissance = function
49      | 1 -> (0,1)
50      | k -> let (p,d) = log_et_puissance (k/2) in (p+1,2*d)
51    in
52    let (p,d) = log_et_puissance (j - i + 1)
53    in
54      min M.(i).(p) M.(j+1-d).(p) ;;
```

Programme 7 la fonction `minimum`

Question II.10
Bien sûr, d'après la question II.6, il suffit d'écrire :

```
55  let ppac2 i j =
56    let k = min index.(i) index.(j) and l = max index.(i) index.(j) in minimum k l ;;
```

Programme 8 la fonction `ppac2`

Partie III — Opérations sur les bits des entiers primitifs

Question III.11
On écrit facilement le programme 9.

```
57 let bit_fort n =
58   let rec cherche n =
59     if n = 1 then 0
60     else cherche (decalage_droite(n,1)) + 1
61   in
62     cherche n ;;
```

Programme 9 la fonction `bit_fort`

Question III.12
On suppose donc qu'un tableau `bits_forts` contient les valeurs de `bit_fort` pour les $256 = 2^8$ premiers entiers.
On écrit alors le programme 10, où on effectue au plus deux décalages pour chaque entrée n.

```
63 let bit_fort n =
64   let haut = decalage_droite(n,15) in
65     if haut = 0 then
66       let bas_haut = decalage_droite(n,8) in
67        if bas_haut = 0 then bits_forts.(n)
68        else bits_forts.(bas_haut) + 8
69     else
70       let haut_haut = decalage_droite(haut,8) in
71        if haut_haut = 0 then bits_forts.(haut) + 15
72        else bits_forts.(haut_haut) + 8 + 15 ;;
```

Programme 10 la fonction `bit_fort`

La figure 2 explique mieux qu'un long discours les décalages utilisés.

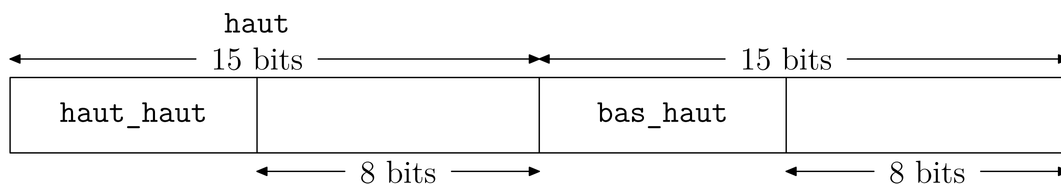

Figure 2 découpage d'un entier sur $N \leqslant 30$ bits

Partie IV — Cas particulier d'un arbre binaire complet

Question IV.13
On propose le programme 11, dont la complexité est clairement $O(n)$ puisque l'on visite tour à tour chaque nœud de l'arbre pour un coût constant.
Ce programme suppose que l'arbre A est bien formé : il doit s'agir d'un arbre binaire complet.

```
73  let remplir_B () =
74    let rec hauteur chemin = function
75      | Noeud(i,[]) -> let x = ou_bits(1,decalage_gauche(chemin,1)) in
76         B.(i) <- x ; Binv.(x) <- i ;
77         0
78      | Noeud(i,[g;d]) ->
79         let xg = decalage_gauche(chemin,1) in
80         let xd = ou_bits(xg,1) in
81         let h = hauteur xg g + 1 and _ = hauteur xd d in
82         let x = decalage_gauche(xd,h) in
83           B.(i) <- x ; Binv.(x) <- i ;
84           h
85    in
86      ignore (hauteur 0 A) ;;
```

Programme 11 la fonction `remplir_B`

La fonction auxiliaire (et récursive) **`hauteur`** renvoie la hauteur du sous-arbre fourni en deuxième argument, après avoir écrit les valeurs convenables dans les tableaux `B` et `Binv`. Le premier argument qui lui est passé est **`chemin`**, qui est l'entier décrivant le chemin jusqu'à la racine du sous-arbre concerné.
Dans le cas d'une feuille (ligne 75), on se contente d'ajouter un 1 à la fin de l'écriture binaire du chemin, d'affecter les cases concernées des deux tableaux et on renvoie une hauteur nulle.
Dans le cas d'un nœud possédant 2 fils `g` et `d`, on commence par calculer les chemins `xg` et `xd` correspondants (en adjoignant un 0 ou un 1 à l'écriture binaire du chemin), puis on procède aux appels récursifs, ce qui remplit les cases des tableaux `B` et `Binv` relatives à tous les descendants, et on récupère ainsi la hauteur du nœud i, qui est le décalage à gauche à appliquer au chemin courant.

Question IV.14
Le ou-exclusif sélectionne les bits distincts : $k =$ `bit_fort`(`ou_excl_bits`($B(i), B(j)$)) est donc l'index du premier bit qui distingue $B(i)$ et $B(j)$. La partie commune (depuis la gauche), de longueur $d - k$, représente le chemin du plus proche ancêtre commun a.
Autrement dit $B(a)$ peut s'obtenir en mettant à 0 les k bits de poids faible du OU de $B(i)$ et $B(j)$.

Question IV.15
On obtient le programme 12. Pour effacer les k bits de poids faible, il suffit de composer un décalage à droite suivi d'un décalage à gauche !

```
87  let ppac3 i j =
88    if appartient i j then j
89    else if appartient j i then i
90    else
91     let k = bit_fort (ou_excl_bits(B.(i),B.(j))) in
92     let x = decalage_droite(ou_bits(B.(i),B.(j)),k) in
93       Binv.(decalage_gauche(x,k)) ;;
```

Programme 12 la fonction `ppac3`

Partie V — Application

Question V.16
Notons C_δ le coût de construction, dans le pire des cas, de l'arbre associé à $\texttt{T}[i..j]$ quand $\delta = j - i$.
On a $C_0 = 1$.
Si $\delta > 0$, on recherche le minimum $\texttt{T}[m]$ de la tranche : le pire cas vient quand $m = j$, car la recherche du minimum coûte $\delta - 1$ et ensuite la construction du sous-arbre coûte $C_{\delta-1}$, donc $C_\delta = \delta - 1 + C_{\delta-1}$, ce qui induit un coût quadratique.
Finalement, la complexité dans le cas le pire de l'algorithme récursif proposé est $O(n^2)$.

Question V.17
On propose le programme 13.

```
94   let rec recompose fd = function
95     | [] -> hd fd
96     | Noeud(v,t) :: suite -> recompose [Noeud(v, t @ fd)] suite ;;
97   let rec accroche v noeud_v = function
98     | [] -> [ noeud_v ]
99     | Noeud(vk,tk) :: suite when v > vk -> noeud_v :: Noeud(vk,tk) :: suite
100    | Noeud(vk,tk) :: suite ->
101        match noeud_v with
102         | Noeud(v,tv) -> accroche v (Noeud(v,[Noeud(vk,tk @ tv)])) suite ;;
103  let construire_A t =
104    let n = vect_length t in
105    let rec itère i branche =
106      if i = n then recompose [] branche
107      else let v = t.(i) in
108        itère (i+1) (accroche v (Noeud(v,[])) branche)
109    in
110      itère 0 [] ;;
```

Programme 13 la fonction `construire_A`

On représente, suivant l'indication de l'énoncé, la branche droite par la suite des arbres $\langle v_k, t_k \rangle$ en ordre inverse.
`recompose : arbre list -> arbre list -> arbre` permet de recomposer l'arbre final par l'appel `recompose branche []`.
`accroche : int -> arbre -> arbre list -> arbre list` réalise le traitement d'une étape, son appel s'écrit `accroche v (Noeud(v,[])) branche`, où `branche` est la branche droite de l'arbre partiel à ce moment, `v` est la clé à insérer. La fonction renvoie la nouvelle valeur de la branche droite.
`construire_A : int vect -> arbre` se contente d'itérer les appels à `accroche`.
On aura noté que les appels à l'opération de concaténation de listes `@` ne pose pas de problème, car les listes utilisées sont limitées à 2 éléments maximum.

Question V.18
Notons $\beta_0 = 0$, et, pour $1 \leqslant i \leqslant n$, β_i la longueur de la branche droite de l'arbre après insertion de $v = \texttt{T}[i]$.
L'étape d'insertion de cet élément a un coût $\beta_i - \beta_{i-1}$, donc, au total, le coût de l'algorithme est égal à $\displaystyle\sum_{i=1}^{n}(\beta_i - \beta_{i-1}) = \beta_n = O(n)$.