CCP 2015 - Informatique
un corrigé

1 Logique et calcul des propositions

Q.1 On doit être dans un parmi trois cas et la formule attendue est

$$(A_1 \wedge \overline{A_2} \wedge \overline{A_3}) \vee (\overline{A_1} \wedge A_2 \wedge \overline{A_3}) \vee (\overline{A_1} \wedge \overline{A_2} \wedge A_3)$$

Premier cas

Q.2 Les traductions sont immédiates

$$A_1 \equiv G \wedge \overline{L},\ A_2 \equiv G \Rightarrow \overline{L} \equiv \overline{G} \vee \overline{L},\ A_3 \equiv \overline{L}$$

Q.3 On en déduit que

$$A_1 \wedge \overline{A_2} \wedge \overline{A_3} \equiv (G \vee \overline{L}) \wedge (\overline{G} \wedge L) \wedge L \equiv 0 \wedge L \equiv 0$$
$$\overline{A_1} \wedge A_2 \wedge \overline{A_3} \equiv (\overline{G} \vee L) \wedge (\overline{G} \vee \overline{L}) \wedge L \equiv [\overline{G} \vee (L \wedge \overline{L})] \wedge L \equiv \overline{G} \wedge L$$
$$\overline{A_1} \wedge \overline{A_2} \wedge A_3 \equiv (\overline{G} \vee L) \wedge (\overline{G} \wedge L) \wedge \overline{L} \equiv (\overline{G} \vee L) \wedge 0 \equiv 0$$

L'un des trois devant s'évaluer à `true`, c'est $\overline{G} \wedge L$ et il faut manger des lipides mais pas de glucides.

Second cas

Q.4 La traduction est un peu plus douteuse ici. Pour A_2, c'est comme avant. J'interprète la réponse A_1 comme une condition nécessaire que je traduis par une implication

$$A_1 \equiv S \Rightarrow R,\ A_2 \equiv \overline{I} \Rightarrow \overline{R}$$

Quant à A_3, on ne sait si les activités à pratiquer sont sportives, intellectuelles ou les deux.

$$A_3 = R \vee S \ \text{ ou } \ A_3 \equiv R \vee I \ \text{ ou } \ A_3 \equiv R \vee (I \wedge S)$$

Q.5 On a la table de vérité suivante

S	I	R	A_1	A_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	1

Comme l'un seulement parmi A_1, A_2, A_3 est vrai, a fortiori on ne peut avoir deux des formules A_1, A_2, R vraies. Il reste les possibilités suivantes

S	I	R	A_1	A_2
1	0	0	0	0
1	1	0	0	1

La première ligne n'est compatible que si l'énoncé A_3 contient "faire du sport".
La seconde est à exclure puisque A_3 est vraie dans ce cas et A_2 aussi.
Il faut donc faire du sport uniquement.

2 Automates et langages

Automate fini complet déterministe

Q.1 On a

$$L(\mathcal{E}) = b(ab)^*(a + ba^*)$$

Q.2 L'énoncé tel qu'écrit n'a pour moi pas de sens puisque q intervient dans le membre de droite de l'équivalence logique mais pas dans celui de droite. On va plutôt montrer que pour tout mot m, on a la phrase suivante :

$$\forall n \in X^*,\ \forall o, d \in Q,\ (\exists q \in Q/\ q = \delta^*(o, m) \wedge d = \delta^*(q, n)) \iff d = \delta^*(o, m.n)$$

On procède par induction structurelle sur m.

- Initialisation : on traite le cas où $m = \Lambda$. Soient $n \in X^*$, $o, d \in Q$.
  S'il existe q tel que $q = \delta^*(o, \Lambda)$ et $d = \delta^*(q, n)$ alors $q = 0$ et $d = \delta^*(o, n) = \delta^*(o, \Lambda.n)$.
  Réciproquement, si $d = \delta^*(o, \Lambda.n)$ alors en posant $q = 0$ on a $q = \delta^*(o, \Lambda)$ et $d = \delta^*(q, n)$.
- Hérédité : on suppose le résultat vrai pour un certain mot m' et on le montre pour le mot $m = x.m'$ avec $x \in X$. Soient donc $n \in X^*$ et $o, d \in Q$.
  S'il existe q tel que $q = \delta^*(o, m)$ et $d = \delta^*(q, n)$ alors $q = \delta^*(o, x.m') = \delta^*(\delta(o, x), m')$ et d'après l'hypothèse de récurrence $d = \delta^*(\delta(o, x), m'.n)$ et, par définition de δ^*, ceci s'écrit $d = \delta^*(o, x.m'.n) = \delta^*(o, m.n)$.
  Réciproquement, supposons que $d = \delta^*(o, m.n)$. Par définition de δ^*, ceci s'écrit $d = \delta^*(\delta(o, x), m'.n)$. D'après l'hypothèse de récurrence, il existe q tel que $q = \delta^*(\delta(o, x), m')$ et $\delta^*(q, n) = d$. Toujours par définition de δ^*, ceci s'écrit $q' = \delta^*(o, m)$ et $\delta^*(q, n) = d$.

Racine carrée d'un langage

Q.3 Même si aucune justification n'est demandée, tentons un raisonnement. Il convient d'abord de trouver les mots de $L(\mathcal{E})$ qui sont composés de deux parties identiques. En particulier, ces mots sont de longueur paire. Il y a deux types de mots de longueur paire dans $L(\mathcal{E})$.

- Ceux du type $b(ab)^n a$. Si n est pair, la première partie du mot se termine par b et la seconde par a ; cela ne convient pas. Si n est impair, le mot se découpe en $b(ab)^p a$ puis $b(ab)^p a$ et cela convient.
- Ceux du type $b(ab)^n ba^{2p}$. Si $p \geq 1$, la seconde partie du mot termine par aa et pas la première. On a donc $p = 0$. Si $n \geq 1$, la seonde partie du mot se termine par bb et pas la première. Le mot est donc bb et il a bien la forme voulue.

$\sqrt{L(\mathcal{E})}$ est constitué du mot b et des mots $b(ab)^p a$ avec $p \in \mathbb{N}$. On a donc

$$\sqrt{L(\mathcal{E})} = b + b(ab)^* a$$

Q.4 Si $m \in L$ alors $m.m \in L^2$ ce qui signifie que $m \in \sqrt{L^2}$. La réciproque est fausse en général puisque si $L = \{a, bab\}$ alors $abab = a(bab) \in L^2$ ce qui indique que $ab \in \sqrt{L^2}$ alors que $ab \notin L$.

$$L \subset \sqrt{L^2}$$

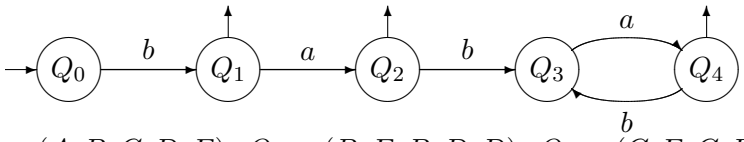
Si l'on reprend l'exemple $L = \{a, bab\}$, on a $\sqrt{L} = \emptyset$ et donc $(\sqrt{L})^2 = \emptyset$. Ceci montre qu'en général L n'est pas inclus dans $(\sqrt{L})^2$. Si on pose $L = \{aa, bb\}$ on a $a, b \in \sqrt{L}$ et donc $ab \in (\sqrt{L})^2$ alors que $ab \notin L$. L'autre inclusion est tout aussi fausse.

Q.5 La construction est assez délicate pour deux raisons : il n'est pas simple de décider si un état est ou non co-accessible et les états ne sont pas numérotés comme dans la définition mais nommés par des lettres. J'ai procédé de la manière suivante pour la construction :

- Je suis parti de l'état initial (A, B, C, D, E) puis ai construit les états accessibles à partir de celui-ci en recommençant avec chaque état construit jusqu'à ce que le processus stationne. J'ai obtenu un gros automate à 12 états. Par exemple, à partir de (A, B, C, D, E) en lisant un a on arrive à (D, C, D, D, E) (A amène à D, B amène à C etc.).

- Je me suis demandé quels états étaient inutiles de façon claire. Comme dans $\mathcal{E}$ on ne quitte plus l'état D quand on l'atteint, un quintuplet atteint à l'étape précédente aura toujours une quatrième coordonnée égale à D. Un état du type $(D, ?, ?, D, ?)$ est alors non co-accessible car on n'atteindra pas un état terminal de $\sqrt{\mathcal{E}}$ car les deux D ne disparaîtront pas (avec les notations de l'énoncé, $t_0 = q_4$ et t_4 n'est pas terminal). De même, (E, D, E, D, D) est inutile car il mène à lui même où à l'état piège (D, D, D, D, D) et aucun des deux n'est terminal.

Bref, après un patient travail d'élagage, j'obtiens l'automate suivant



$$Q_0 = (A, B, C, D, E),\ Q_1 = (B, E, B, D, D),\ Q_2 = (C, E, C, D, D),$$
$$Q_3 = (B, D, B, D, D),\ Q_4 = (C, D, C, D, D)$$

Q.6 On a ainsi

$$L(\sqrt{\mathcal{E}}) = b + ba + baba(ba)^* = b + ba(ba)^*$$

Notons au passage que (même si ce n'est pas la même expression) on a

$$L(\sqrt{\mathcal{E}}) = \sqrt{L(\mathcal{E})}$$

Q.7 $\sqrt{A}$ a un unique état initial et est définie par une fonction de transition complète. C'est donc un automate déterministe.

Q.8 On prouve le résultat demandé par induction structurelle sur le mot m. L'hypothèse pour m est ainsi

$$\forall (o_0, \ldots, o_n) \in Q_{\mathcal{A}}^{n+1},\ \delta^*_{\sqrt{\mathcal{A}}}((o_0, \ldots, o_n), m) = (\delta^*_{\mathcal{A}}(o_0, m), \ldots, \delta^*_{\mathcal{A}}(o_n, m))$$

- Initialisation : le résutat est vrai quand $m = \Lambda$ par définition de la fonction de transition itérée.
- Hérédité : supposons le résultat vrai pour un mot m' et montrons le pour $m = x.m'$ avec $x \in X$. Soit $(o_0, \ldots, o_n) \in Q_{\mathcal{A}}^{n+1}$. On a

$$\begin{aligned}
\delta^*_{\sqrt{\mathcal{A}}}((o_0, \ldots, o_n), m) &= \delta^*_{\sqrt{\mathcal{A}}}(\delta_{\sqrt{\mathcal{A}}}((o_0, \ldots, o_n), x), m') \text{ par définition de } \delta^*_{\sqrt{\mathcal{A}}} \\
&= \delta^*_{\sqrt{\mathcal{A}}}((o'_0, \ldots, o'_n), m') \text{ avec } o'_i = \delta_{\mathcal{A}}(o_i, x) \text{ par def de } \delta_{\sqrt{A}}. \\
&= (\delta^*_{\mathcal{A}}(o'_0, m'), \ldots, \delta^*_{\mathcal{A}}(o'_n, m')) \text{ par hyp. de réc.} \\
&= (\delta^*_{\mathcal{A}}(o_0, m), \ldots, \delta^*_{\mathcal{A}}(o_n, m)) \text{ par def de } \delta_A.
\end{aligned}$$

ce qui montre le résultat pour m.

Q.9 Par définition du langage reconnu si $m \in L(\sqrt{A})$ alors $\delta^*_{\sqrt{\mathcal{A}}}((q_0, \ldots, q_n), m) \in T_{\sqrt{A}}$ et donc (avec la question précédente), en notant $q_j = \delta^*_{\mathcal{A}}(q_0, m)$, on a $\delta^*_{\mathcal{A}}(q_j, m) \in T_{\mathcal{A}}$. Avec la question **II.2** on e déduit que $\delta^*_{\mathcal{A}}(q_0, m.m) \in T_{\mathcal{A}}$ et donc que $m.m \in L(\mathcal{A})$.

La réciproque se fait de même en utilisant l'implication dans l'autre sens de **II.2**.

Q.10 On a ainsi montré que

$$L(\sqrt{\mathcal{A}}) = \sqrt{L(\mathcal{A})}$$

3 Algorithmique et programmation

Le tri à bulles

III.1 L'élément i de la la liste l est l'élément $i - 1$ de la queue de l, sauf si $i = 1$.

```
let rec lire i l =
   match i with
   1 -> hd l
  |_ -> lire (i-1) (tl l);;
```

III.2 La fonction affiche les valeurs de i, j et t à la fin de chaque étape de boucle (il y a donc autant d'affichage que d'itérations c'est à dire $\sum_{i=0}^{n-1}(n-i-1)$). Dans l'exemple proposé, on voit s'afficher

```
0 0 [1, 3, 4, 2]
0 1 [1, 3, 4, 2]
0 2 [1, 3, 2, 4]
1 0 [1, 3, 2, 4]
1 1 [1, 2, 3, 4]
2 0 [1, 2, 3, 4]
```

Après exécution, on a donc

`resultat=[1,2,3,4]`

Si on n'utilise pas l'instruction `copy`, `t` et `p` sont physiquement égaux. Ainsi, les modification sont opérées au niveau de la liste argument et, après exécution, on perd la valeur initiale de la liste.

III.3 Il y a un gros problème de numérotation dans cette question puisque les listes Python sont numérotées à partir de 0 alors que les séquences d'entier sont numérotées à partir de 1. Pour éviter de jongler avec les indices ($s[i]$ représentant s_{i-1} avec la numérotation de l'énoncé), je propose de traiter cette question en considérant que les séquences sont, comme en Python, numérotées à partir de 0.

En continuant de noter m la taille de la séquence argument, je note donc $s_0, \dots, s_{m-1}$ ses éléments.

La fonction définit un tableau `t` de taille m et ne procède ensuite qu'à des échanges d'éléments. Il en résulte qu'en fin d'itération, les éléments de `t` sont globalement les mêmes que ceux de la séquence argument. On a ainsi

$$dom(\mathtt{r}) = dom(\mathtt{s}) \wedge codom(\mathtt{r}) = codom(\mathtt{s})$$

La boucle extérieure est effectuée pour les valeurs $i = 0, \dots, m-1$. Montrons qu'au début de la boucle numérotée i, les i dernières cases de `t` sont triées et sont les i plus grandes de la liste `t`, c'est à dire que

$$\forall k \leq m-1-i,\ \mathtt{t}[k] \leq \mathtt{t}[m-i] \ \text{ et } \ \mathtt{t}[m-i] \leq \cdots \leq \mathtt{t}[m-1]$$

- Initialisation : il n'y a rien à vérifier pour le cas $i = 0$.
- Hérédité : on suppose le résultat vrai à un rang $i \leq m-1$. On effectue alors la boucle de numéro i. Dans celle-ci, on effecue la boucle intérieure pour de valeurs $j = 0, \dots, m-i-2$. Montrons qu'au début de la boucle numéro j, l'élément $\mathtt{t}[j]$ est inférieur à ceux qui le précèdent.
  - Initialisation : il n'y a rien à vérifier au rang $j = 0$.
  - Hérédité : supposons le résultat vrai à un rang $j \leq m-i-2$. On effectue la boucle numéro j. Après l'instruction conditionnelle, $\mathtt{t}[j+1]$ est plus grand que $\mathtt{t}[j]$ lui même plus grand que les précédents éléments. Le résultat est donc vrai au rang $j+1$.

  La boucle interne s'arrête au début de l'itération $j = m-i-1$. On sait alors que $\mathtt{t}[m-i-1]$ est plus grand que les éléments qui le précédent. Mais, par hypothèse de récurrence, ill est aussi plus petit que $\mathtt{t}[m-i]$. On a ainsi le résultat au rang $i+1$.

Quiand la boucle externe s'arrête, on a $i = m$ et la liste est triée d'après le résultat prouvé.

III.4 Les boucles étant inconditionnelles, il n'y a aucun problème de terminaison.

III.5 On note cette fois n la taille de la liste argument. Quelle que soit la liste argument, les boucles sont effectuées le même nombre de fois égal à $\sum_{i=0}^{n-1}(n-i-1) = O(n^2)$. Une itération

se fait en temps constant et les boucles ont un coût $O(n^2)$. Hors de la boucle, la copie de liste a un coût $O(n)$. La complexité est donc $O(n^2)$.
Il n'y a pas vraiment de "cas le pire" particulier. Si on veut en distinguer un, il faudrait un cas où le test de la conditionnelle est toujours égal à `True`. Ceci a lieu quand la liste est initialement triée par ordre décroissant.

III.6 La gestion avec Caml est un peu plus complexe car les liste Caml ne sont pas mutables. Le tri à bulles est en fait une suite de "passes" dans la liste : on commence par scanner le tableau entier (étape $i = 0$ de la fonction Python) pour mettre le plus grand élément à sa place puis on recommence avec les $n-1$ premiers éléments puis les $n-2$ premiers etc.
J'écris une première fonction `passe : int list → int → int list` telle que l'appel `passe l k` renvoie la liste après une passe sur les k premiers éléments. Cette fonction correspond à la boucle interne de la fonction Python.

```
let rec passe l k =
    if k=1 then l
    else begin
         let a=hd l and b=hd (tl l) in
         if b<a then b::(passe (a::(tl (tl l))) (k-1))
         else a::(passe (tl l) (k-1));
         end ;;
```

Il convient ensuite de faire se succéder les passes. Pour cela, j'écris une seconde fonction auxiliaire `etape : int → int list → list`. Dans l'appel `etape i l`, on renvoie la liste obtenue après avoir effectué des passes avec $i, i-1, \ldots, 1$ élément.

```
let rec etape i liste =
    if i=1 then liste
    else etape (i-1) (passe liste i)
```

Il reste à appeler cette fonction avec le nombre d'éléments de la liste.

```
let tribul l=
 let n=list_length l in
 etape n l;;
```

ATTENTION : tout devient beaucoup plus simple si on ne suit pas exactement le schéma de la fonction Python et que l'on se permet d'effectuer une passe sur toute la liste à chaque étape (on n'a alors plus besoin de deux arguments pour la fonction `passe`). On n'y perd pas grand chose en complexité. On pourrait aussi s'arrêter quand une passe ne change rien (pour cela, on lui fait renvoyer un couple booléen,liste, le booléen indiquant si un échange a été effectué.

Le tri par tas

III.7 L'arbre dessiné est le suivant.

```
let t4=Noeud(
          Noeud (
             Noeud (Vide,1,Vide),
             3,
             Vide) ,
          4,
          Noeud (Vide,2,Vide) );;
```

III.8 Ici, la hauteur est le maximum de nombre de noeuds rencontrés (et non le nombre d'arêtes) dans un chemin de la racine vers une feuille. On a alors la formule suivante (facile à comprendre quand les arbres sont vides et pour laquelle la convention choisie pour l'arbre vide est cohérente) :

$$\forall a \neq \emptyset,\ \eta(a) = 1 + \max(\eta(G(a)), \eta(D(a)))$$

III.9 La hauteur est maximale quand il y a un minimum de noeud sur chaque niveau. Comme il

y en a au moins un sur chaque niveau existant, la hauteur est maximale quand chaque noeud à part les feuilles ont un seul fils non vide. On a alors un arbre filiforme de hauteur n (nombre d'étiquettes). Ce peut-être, par exemple, un peigne à droite ou à gauche.
De même, la hauteur est minimale quand il y a un maximum de noeuds par niveau. Or, chaque noeud ayant au plus deux fils, le nombre maximal de noeuds par niveau double à chaque niveau. Il y en a 1 au niveau 1 (la racine), au plus 2 ou niveau 2 et, plus généralement, au plus 2^{h-1} au niveau h. Pour un arbre de hauteur h, il y a donc au maximum $1+2+\cdots+2^{h-1} = 2^h-1$ noeuds. La hauteur minimale est donc le plus petit h tel que $2^h - 1 \geq n$ c'est à dire $h \geq \log_2(n+1)$. Cette hauteur minimale est égale à $\lceil \log_2(n+1) \rceil$. Ce minimum est atteint pour un arbre où tous les noeuds sauf éventuellement ceux des niveaux le plus bas ont deux fils non vides.
Remarque : la notion de niveau est en fait presque celle de profondeur introduite juste après.

III.10 Il est ici question de parcours en largeur et on a donc naturellement envie d'utiliser la structure de file. On pourrait utiliser le module `queue` de Caml mais les files disponibles sont alors mutables et contreviennent aux directives de l'énoncé. Pour faire simple, je vais modéliser une file d'attente avec une liste. La tête de la file est la tête de la liste et quand il faut ajouter un élément à une file, il faut le faire en bout de liste et donc grâce à une concaténation.
Remarque : on se permet cette implémentation car l'énoncé ne nous impose pas de complexité. Il serait possible de modéliser une file d'attente avec deux listes, l'une des éléments prêts à être utilisés et l'autre avec les éléments en, attente qu'on a ajouté. Quand on a besoin d'un élément de lal file et qu'il n'y en a plus de prêt, on déverse la file des éléments en attente dans la liste des éléments prêts.
On écrit ainsi une fonction auxiliaire `lire_aux : int → arbre list → int`. Dans l'appel `lire_aux i file`, le second argument est notre file d'attente. Elle nous permet d'appliquer l'algorithme du cours de parcours en largeur, qu'on se permet de ne pas redétailler.

```
let lire i a =
   let rec lire_aux i file =
       if i=1 then begin
          let (Noeud(g,e,d))=(hd file) in
           e
          end
       else begin
          let (Noeud(g,e,d))=(hd file) in
          lire_aux (i-1) ((tl file)@[g;d])
          end
   in lire_aux i [a];;
```

III.11 Pour éviter de multiples parcours dans l'arbre, on écrit une fonction auxiliaire `verif_aux : arbre → bool*int` renvoyant un couple indiquant si l'arbre est partiellement ordonné ainsi que la maximum de ses étiquettes. Pour inclure le cas de l'arbre vide, on choisit la classique convention $\max(\emptyset) = -\infty$, $-\infty$ étant l'entier minimal Caml `min_int`. Il suffit alors, dans la fonction principale, de ne garder que la première coordonnée.

```
let rec verif_aux a =
   match a with
   Vide -> true,min_int
  |Noeud(g,e,d) ->
       let (bg,mg) = verif_aux g in
       let (bd,md) = verif_aux d in
       (bg && bd && e>=mg && e >= md , max (max mg md) e);;

let verifier a = fst (verif_aux a);;
```

III.12 Chaque noeud ayant au plus deux fils, le nombre maximal de noeuds double à chaque profondeur. Il y en a 1 à profondeur 0 (la racine), au plus 2 à profondeur 1 et, plus généralement,

au plus 2^p à profondeur p.

III.13 Pour un arbre complet de hauteur $p = \eta(a)$, il y a des noeuds à profondeurs $0, \ldots, p-1$ (par exemple, si la hauteur est égale à 1, il y a seulement la racine et donc seulement des noeuds à profondeur 0). Il y a donc $1 + 2 + \cdots + 2^{\eta(a)-1} = 2^{\eta(a)} - 1$ noeuds.

III.14 La hauteur $\eta(a)$ d'un arbre complet non vide à n noeuds est donc $\log_2(n+1)$.

III.15 Il suffit de suivre les définitions. Je note cependant une "astuce". Quand on a fait l'appel récursif, par exemple sur le fils gauche, on obtient un hauteur `hg` et une catégorie `cg`. `g` est complet si `cg=Complet` MAIS il est parfait si `cg` vaut `Complet` OU `Parfait` (puisque tout arbre complet est parfait).

```
let rec analyser a =
   match a with
   Vide -> (0,Complet)
  |Noeud(g,e,d) ->
      let (hg,cg) = analyser g in
      let (hd,cd) = analyser d in
      if cg=Complet && cd=Complet && hg=hd then (hg+1,Complet)
      else if cd=Complet && (cg=Parfait||cg=Complet) && hg=1+hd then (hg+1,Parfait)
      else if cd=Complet && (cg=Parfait||cg=Complet) && hg=1+hd then (hg+1,Parfait)
      else if cg=Complet && (cd=Parfait||cd=Complet) && hg=hd then (hg+1,Parfait)
      else (1+(max hg hd),Quelconque);;
```

III.16 Les noeuds de la profondeur $\leq p-1$ sont ceux dont le numéro est inférieur ou égal à $1 + 2 + \cdots + 2^{p-1} = 2^p - 1$. Ceux de la profondeur p sont ceux de numéro compris (au sens large) entre 2^p et $2^{p+1} - 1$. A gauche de celui numéro n, il y en a donc $n - 2^p + 1$ (si on prend ceux à gauche au sens large, c'est à dire en incluant le noeud lui même) ou $n - 2^p$ si on compte ceux qui sont strictement à gauche.

III.17 A la profondeur p, les noeuds qui sont strictement à gauche des fils du noeud de numéro n (supposé à profondeur p) sont les fils des noeuds strictement à gauche de n à la profondeur p. Il y en a donc $2n - 2^{p+1}$ (deux fois plus de fils que de pères).

III.18 Le premier noeud à profondeur $p+1$ a pour numéro 2^{p+1}. Les k premiers noeuds de ce niveau ont donc des numéros entre 2^{p+1} et $2^{p+1} + k - 1$. Avec la question précédente, les deux fils du noeud numéro n ont donc pour numéros $2^{p+1} + (2n - 2^{p+1})$ et $2^{p+1} + (2n - 2^{p+1}) + 1$ c'est à dire $2n$ et $2n+1$.

III.19 Si les fils de n sont $2n$ et $2n+1$, le père de n est $\lfloor n/2 \rfloor$.

III.20 Considérons le noeud de numéro n et notons $b_k \ldots b_0$ son écriture binaire (bit de poids faible à droite). D'après ce qui précède, $b_0 = 0$ si le noeud est fils gauche et $b_0 = 1$ sinon. De plus, $b_k \ldots b_1$ est l'écriture binaire du numéro du père.

Il est alors facile de localiser le noeud de numéro $n \geq 1$ dans un arbre `t=Noeud(g,e,d)` : on décompose n en base 2 sous la forme $b_k b_{k-1} \ldots b_0$ avec $b_k = 1$. Les bits $b_{k-1}, \ldots, b_0$ (dans cet ordre) nous indiquent le chemin à prendre pour atteindre n : un 0 et on part à gauche, un 1 et on par à droite. Par exemple, $11 = \overline{1011}$ et on atteint le noeud numéro 11 par le chemin gauche-droit-droit.

Une façon d'envisager la question est de commencer par obtenir la liste $[b_{k-1}, \ldots, b_0]$ associée à n (avec les notations précédentes). C'est l'objet de la première fonction auxiliaire. Ici, on veut une liste avec le bit de poids faible à droite et il est plus naturel d'obtenir celle avec le poid faible à gauche. Pour éviter une concaténation, j'utilise une technique accumulative. J'écris donc une fonction `decompose : int → int list → int list`. Dans l'appel `decompose n accu`, on envoie la liste $b_{k_1} :: \ldots :: b_0$@`accu` (avec les notations précédentes).

```
let rec decompose n accu=
   if n=1 then accu
   else if n mod 2 = 0 then decompose (n/2) (0::accu)
   else decompose (n/2) (1::accu);;
```

La seconde fonction `parcours : int list → arbre → int` est telle que `parcours l t` renvoie l'étiquette de `t` obtenue en suivant la brache définie par la liste `l` supposée composée de 0 et de 1. La lecture d'un 0 nous amène à gauche et celle d'un 1 à droite. On suppose que la branche en question existe.

```
let rec parcours l (Noeud(g,e,d)) =
   match l with
   [] -> e
  |0::q -> parcours q g
  |1::q -> parcours q d;;
```

Il reste alors à combiner les deux fonctions précédentes.

```
let lire i t =
   parcours (decompose i []) t;;
```

III.21 Il y a beaucoup d'appels à la première fonctions auxiliaire. Pour s'en sortir, il faut bien distinguer l'endroit d'où vient l'appel récursif. Je reprends pour cela le code :

```
let construire l =
   let rec aux1 l a =
      match l,a with
     | ([],_) -> (a,l)
     | (t::q,Vide) -> (Noeud(Vide,t,Vide),q)
     | (_,Noeud(g,v,d)) ->
        match (aux1 l g) with     (* POSITION 1*)
        | (rga,[]) ->
            ((Noeud(rga,v,d)),[])
        | (rga,rgl) ->
            (match (aux1 rgl d) with    (* POSITION 2 *)
              |(rda,rdl) ->
                ((Noeud(rga,v,rda)),rdl))
   in
   let rec aux2 l a =
     match (aux1 l a) with   (* POSITION 3 *)
       | (ra,[]) -> ra
       | (ra,rl) -> (aux2 rl ra)
   in
   (aux2 l Vide) ;;
```

L'appel initial de `aux 2` se fait avec $[1;2;3;4;5;6]$ et `Vide`

- En position 3, on appelle `aux1` avec $[1;2;3;4;5;6]$ et `Vide`.
- L'appel se termine et renvoie `Noeud(Vide,1,Vide)` et $[2;3;4;5;6]$.

On alors un appel à `aux2` avec $[2;3;4;5;6]$ et `Noeud(Vide,1,Vide)`.

- En position 3, on appelle `aux1` avec $[2;3;4;5;6]$ et `Noeud(Vide,1,Vide)`.
  - En position 1, on appelle `aux1` avec $[2;3;4;5;6]$ et `Vide`.
  - L'appel se termine et renvoie `Noeud(Vide,2,Vide)` et $[3;4;5;6]$.
  - En position 2, on appelle `aux1` avec $[3;4;5;6]$ et `Vide`.
  - L'appel se termine et renvoie `Noeud(Noeud(Vide,2,Vide),1,Vide,Noeud(Vide,3,Vide))` et $[4;5;6]$.
- L'appel se termine et renvoie `Noeud(Noeud(Vide,2,Vide),1,Vide,Noeud(Vide,3,Vide))` et $[4;5;6]$.

On alors un appel à `aux2` avec $[4;5;6]$ et `Noeud(Noeud(Vide,2,Vide),1,Vide,Noeud(Vide,3,Vide))`.

- En position 3, on appelle `aux1` avec $[4;5;6]$ et `Noeud(Noeud(Vide,2,Vide),1,Vide,Noeud(Vide,3,Vide))`.
  - En position 1, on appelle `aux1` avec $[4;5;6]$ et `Noeud(Vide,2,Vide)`.
    - En position 1 on appelle `aux1` avec $[4;5;6]$ et `Vide`

+ L'appel se termine et renvoie `Noeud(Vide,4,Vide)` et $[5;6]$
+ En position 2 on appelle `aux1` avec $[5;6]$ et `Vide`
+ L'appel se termine et renvoie `Noeud(Vide,5,Vide)` et $[6]$

∗ L'appel se termine et renvoie `Noeud(Noeud(Vide,4,Vide),2,Noeud(Vide,5,Vide))` et $[6]$.

∗ En position 2, on appelle `aux1` avec $[6]$ et `Noeud(Vide,3,Vide)`.
+ En position 1, on appelle `aux1` avec $[6]$ et `Vide`
+ L'appel se termine et renvoie `Noeud(Vide,6,Vide)` et $[]$

∗ L'appel se termine et renvoie `Noeud(Noeud(Vide,6,Vide),3,Vide)` et $[]$.

- L'appel se termine et renvoie le résultat final qui est un arbre parfait à 6 noeuds étiquétés par leurs numéros respectifs.

III.22 Tentons une récurrence sur p (hauteur de l'arbre complet argument). On garde les notations de l'énoncé **MAIS** on change la propriété iv) où on doit lire $r_i = s_{I+2^p}$.

- Initialisation : dans le cas $p = 0$, l'argument a est l'arbre vide. On regarde donc le résultat de l'appel `aux1 s Vide` avec `s` liste de taille m.
  Si $m = 0$ alors $b =$ `Vide` et $r = []$ (et donc $n = 0$). Seule la propriété ii) est pertinente et elle est vérifiée.
  Si $m \geq 1$ alors $b =$ `Noeud`$(\texttt{Vide}, s_1, \texttt{Vide})$ et $r = [s_2, \ldots, s_m]$ (et donc $n = m - 1$). Il n'y a rien à vérifier pour i) et ii) et iii) (car $m \geq 2^p = 1$). Pour iv), on vérifie que b est complet de hauteur 1, que $n = m - 1$, et que $r_i = s_{i+1}$.
- Hérédité : on suppose le résultat vrai jusqu'à un rang $p - 1 \geq 0$. On se donne un arbre a complet de hauteur p et une liste s de taille m. On note $a =$ `Noeud`(g, v, d).
  Si la liste est vide ($m = 0$), alors $b = a$ et $r = []$ (et donc $n = 0$). Il n'y a rien à vérifier pour iii) et iv). i) est immédiat ainsi que ii).
  Sinon $m \geq 1$, il y a un appel `aux1 s g`; notons `(rga,rgs)` le résultat de cet appel. Comme a est complet de hauteur p, g est complet de hauteur $p-1$ on peut appliquer l'hypothèse de récurrence qui nous indique que le résultat renvoyé vérifie les hypothèses "au rang $p - 1$". On distingue alors deux cas.
  - Si $rgs = []$, on a alors $b =$ `Noeud`(rga, v, d) et $r = []$ (et donc $n = 0$). Comme rga et g ont les mêmes étiquettes au même endroit, il en va de même pour a et b (propriété i)). Comme $m \geq 1$, la propriété ii) est vérifiée. Ici, $rgs = []$ et on est dans le cas $m \leq 2^{p-1} < 2^p$ (par l'hypothèse de récurrence) et la propriété iv) est vraie. On doit vérifier iii) :
    - $n = 0$, est vrai
    - b est parfait de profondeur $p + 1$ car d est complet de profondeur $p - 1$ et rga est parfait de profondeur p par hypothèse de récurrence
    - soit $i \in [1, m]$; par hypothèse de récurrence, s_i sera égal à l'étiquette du noeud numéroté $i + 2^{p-1} - 1$ dans rga et ce même noeud a pour numéro $i + 2^p - 1$ dans b.
  - Sinon rgs est non vide et on est dans le cas $m > 2^{p-1}$ par hypothèse de récurrence. Il y a un appel `aux1 rgs d` ; notons `(rda,rds)` le résultat de cet appel. Comme a est complet de hauteur p, d est complet de hauteur $p - 1$ on peut appliquer l'hypothèse de récurrence qui nous indique que le résultat renvoyé vérifie les hypothèses "au rang $p - 1$". Que s'est-il passé jusque là ,
    - On a un arbre rga qui est complet et obtenu à partir de g en ajoutant les élément $s_1, \ldots, s_{2^{p-1}}$.
    - On a un arbre rda qui est parfait et obtenu à partir de d en ajoutant les élément $s_{2^{p-1}+1}, \ldots, s_m$ si $m \leq 2^p$ (et dans ce cas rds est vide) ou les éléments $s_{2^{p-1}+1}, \ldots, s_{2^p}$ sinon (et dans ce cas $rds = [s_{2^p+1}, \ldots, s_m]$).

  Avec la changement de numérotation évoqué plus haut quand on passe d'un noeud de la dernier ligne de rga ou rda considéré comme un noeud de b, on obtient les formules voulues (la vérification est VRAIMENT du même type que celle menée plus haut).

III.23 Il faut ici expliquer le comportement de `aux2` (dont on déduit celui de `construire`). Dans l'appel `aux2 l a`, on suppose que `a` est un arbre complet. L'appel `aux1 l a` ajoute une couche à `a` et enlève les éléments ajoutés de la liste. On recommence jusqu'à épuisement de la liste. Ceci montre que l'on construit un arbre parfait dont les étiquettes des noeuds successifs (dans l'ordre en largeur) sont les éléments de la liste.

III.24 La preuve de terminaison des fonctions est en fait inclus dans les preuves précédentes (on sous-entend dans la preuve "les appels se terminent et donnent un résultat tel que...").

III.25 Notons $A_{n,r}$ un majorant du nombre d'appels récursif effectués dans l'appel `aux2 l a` quand `a` est complet de hauteur r et quand `l` contient n éléments. Notons aussi B_k le nombre d'appels récursifs effectués dans l'appel `aux1 l a` quand a est complet de hauteur k. Lors de l'appel `aux2 l a` avec a de hauteur r, on fait un appel à `aux1` qui renvoie un arbre de hauteur $r+1$ et une liste avec $n-2^r$ éléments. On a donc

$$A_{n,r} = 1 + B_r + A_{n-2^r,r+1}$$

En itérant le processus, on obtient

$$A_{n,r} = 2 + B_r + B_{r+1} + A_{n-2^r-2^{r+1},r+2}$$

puis, plus généralement

$$A_{n,r} = k + B_r + \cdots + B_{r+k-1} + A_{n-2^r-\cdots-2^{r+k-1},r+k}$$

Si n est de l'ordre de $1 + \cdots + 2^{p-1}$, c'est à dire de l'ordre de 2^p, on obtiendra $A_{n,r}$ de l'ordre de

$$(p-r) + B_r + \cdots + B_{p-1}$$

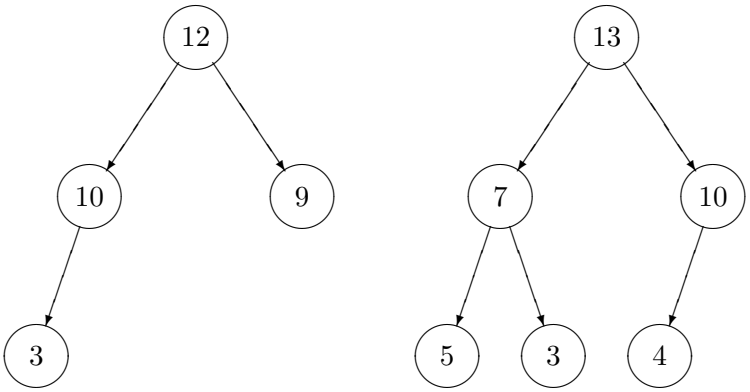
Par ailleurs, B_r est de l'ordre de $2B_{r-1}$ (au pire deux appels récursifs) et donc B_r est de l'ordre de 2^r. Ainsi, $A_{n,r}$ est de l'ordre de

$$(p-r) + 2^r + \cdots + 2^{p-1}$$

Si on veut construire un arbre à partir d'une liste de taille n, on fait un appel à `aux2` avec la liste et l'arbre vide. Le nombre d'appels est alors de l'ordre de $p + 2^0 + \cdots + 2^{p-1}$ c'est à dire de l'ordre de n.

Remarque : on obtient mieux que le résultat demandé.

III.26 L'énoncé d'origine est manifestement faux. Si l'étiquette v à insérer est strictement plus grande que celles des racines de g et d, on doit donner l'étiquette v à la racine de l'arbre résultat. Mais il est tout à fait possible qu'il faille transférer des étiquettes de l'arbre d vers l'arbre d pour garder une forme parfaite. Prenons l'exemple des arbres g et d suivants :



Pour ajouter l'étiquette 15, il faut transférer celles $5, 3, 4$ de b vers a. Il me semble impossible de faire cela sans un parcours de tout l'arbre.
Au vu de la suite du sujet, et en particulier des dernières questions, il est légitime de penser qu'il faut supposer que $\eta(g) = \eta(d)$ et g complet OU QUE $\eta(g)+1 = \eta(d)$ et que d est complet. Autrement dit, que `Noeud(g,v,d)` est un arbre parfait qui est "presque" en tas, le seul problème pouvant provenir de v et des étiquettes de g et d.
On peut alors proposer la fonction suivante (dans le cas général, on regarde qui doit venir à la racine de v, de l'étiquette de g ou de celle de d, en agissant alors en conséquence).

```
let rec placer g v d =
   match (g,d) with
   Vide,Vide -> Noeud(Vide,v,Vide)
  |Noeud(Vide,xg,Vide),Vide -> Noeud(Noeud(Vide,min xg v,Vide),max xg v,Vide)
  |Noeud(gg,xg,dg),Noeud(gd,xd,dd) ->
          if v>=(min xg xd) then Noeud(g,v,d)
          else if xg>=xd then Noeud(placer gg v dg,xg,d)
          else Noeud(g,xd,placer gd v dd);;
```

III.27 Il suffit de trier la liste argument avant de faire l'appel à `aux2`. En utilisant la fonction `sort` du module du même nom, on obtient

```
let construire_bis l=
   construire (sort__sort (fun x y -> x>=y) l);;
```

L'énoncé d'origine est totalement incompréhensible. De plus, il ne respecte pas la numérotation des listes en Python. Pour respecter les notations des séquences qui commencent à 1, on ignorera royalement la case numéro 0 des listes. Le nombre d'éléments d'une séquence représentée par une liste `a` est donc $n = \texttt{len(a)} - 1$.

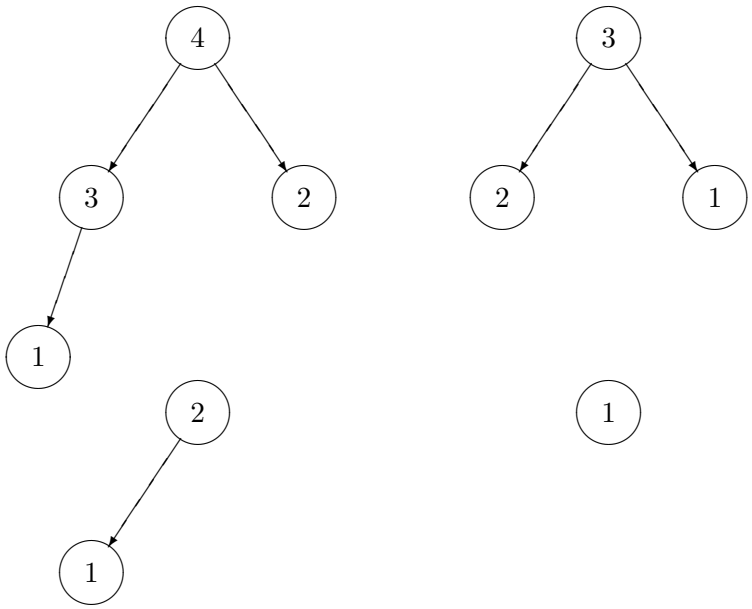
III.28 Dans cette question, il s'agit sans doute d'écrire l'équivalent de la fonction `placer` de la question **26**. Dans l'appel `placer a r f`, on suppose que le sous-tableau avec les indices entre `r` et `f` a une structure de tas SAUF éventuellement pour ce qui concerne le noeud numéro `r` (qui est la racine du tas considéré) dans sa liaison avec ses fils. On veut régler ce problème. L'idée est, s'il y a problème, d'échanger l'étiquette du noeud numéro r avec celle du plus grand de ses fils (s'ils sont dans le tas, c'est à dire de numéro $\leq$ `f`).

```
def placer(r,a,f):
    n=r
    while 2*n+1<=f and (a[2*n]>a[n] or a[2*n+1]>a[n]):
        if a[2*n]>a[2*n+1]:
            a[n],a[2*n]=a[2*n],a[n]
            n=2*n
        else:
            a[n],a[2*n+1]=a[2*n+1],a[n]
            n=2*n+1
    if f==2*n and a[2*n]>a[n]:
        a[n],a[2*n]=a[2*n],a[n]
```

III.29 On part du sous-tableau ne comportant que les deux dernier élément. Il correspond à un arbre parfait qui est presque un tas et que `placer` peut transformer. On prend en compte alors l'avant dernier élément etc.

```
def entasser(a):
    f=len(a)-1
    for i in range(f,0,-1):
        placer(i,a,f)
```

III.30 On obtient successivement



Les tableaux successifs étant

$$[4,3,2,1],\ [3,1,2,4],\ [2,1,3,4],\ [1,2,3,4]$$

III.31 On transforme l'argument `s` en un tas (fonction d'entassage). Le maximum est en position 1, et on l'échange avec la dernière case. Le bout de tableau entre 1 et $f-1$ est alors presque un tas, que l'on peut réorganiser avec `placer`. On recommence en ne prenant plus en compte le dernier élément (qui est sorti du tas). La variable `f` correspond au numéro de la dernière case utile du tableau. On a fini quand cette variable vaut 1.

```
def trier(s):
    f=len(a)-1
    entasser(s)
    while f>1:
        s[f],s[1]=s[1],s[f]
        f-=1
        placer(1,s,f)
```

III.32 Grâce à la fonction `lire`, on peut aller récupérer la valeur de la dernièrre étiquette. On modifie la fonction pour quelle nous renvoie non seulement cette valeur mais aussi un arbre similaire à celui de départ mais sans cette dernière étiquette (la fonction `parcours2` ainsi que `lire2` renvoient donc un couple.

```
let rec parcours2 l (Noeud(g,e,d)) =
   match l with
   [] -> e,Vide
  |0::q -> let x,newg=parcours q g in x,Noeud(newg,e,d)
  |1::q -> let x,newd=parcours q d in x,Noeud(g,e,newd);;

let lire2 i t =
   parcours2 (decompose i []) t;;
```

Il suffit alors d'utiliserla fonction `placer` de la question **26**.

```
let rec remonter i a =
    let x,t=lire2 i a in
    if t=Vide then Vide
    else let (Noeud(g,e,d))=t in
         placer g x d;;
```

III.33 C'est la synthèse de ce qui précède. `construire_bis` permet d'obtenir un tas avec les étiquettes de la liste. On va ensuite démonter ce tas petit à petit pour obtenir le plus grand élément, puis le suivant etc. La fonction `demonter : arbre` $\rightarrow$ `int` $\rightarrow$ `int list` fait le travail. L'argument entier correspond au nombre d'élément de l'arbre argument (nécessaire pour utiliser `remonter`).

```
let trier l =
   let rec demonter a i=
      match a with
      Vide -> []
     |Noeud(g,e,d) -> e::(demonter (remonter i a) (i-1))
   in demonter (construire_bis l) (list_length l);;
```