CCP 2018 - Option informatique
Un corrigé

I. Logique et calcul des propositions

I.1 Première épreuve

Q.1 $B_1 = P_1 \vee P_2$.

Q.2 $B_2 = \overline{P_1}$.

Q.3 On note F la proposition résultant de la règle du jeu (et qui est donc vraie).

$$\begin{aligned} F &= (B_1 \wedge B_2) \vee (\overline{B_1} \wedge \overline{B_2}) \\ &\equiv ((P_1 \vee P_2) \wedge \overline{P_1}) \vee (\overline{P_1} \wedge \overline{P_2} \wedge P_1) \\ &\equiv (P_1 \wedge \overline{P_1}) \vee (P_2 \wedge \overline{P_1}) \\ &\equiv P_2 \wedge \overline{P_1} \end{aligned}$$

Q.4 Il faut donc choisir la boîte 2.

I.2 Deuxième épreuve

Q.5 $B_1 = \overline{P_1} \vee P_2$ et $B_2 = P_1$.

Q.6 On a cette fois

$$\begin{aligned} F &\equiv ((\overline{P_1} \vee P_2) \wedge P_1) \vee (P_1 \wedge \overline{P_2} \wedge P_1) \\ &\equiv (\overline{P_1} \wedge P_1) \vee (P_2 \wedge P_1) \vee (P_1 \wedge \overline{P_2}) \\ &\equiv P_1 \wedge (P_2 \vee \overline{P_2}) \\ &\equiv P_1 \end{aligned}$$

Il y a une clé dans la boîte 1 et les deux affirmations sont donc vraies. Avec la première on en déduit qu'il y a aussi une clé verte dans la boïte 2.
On peut ainsi choisir n'importe quelle boîte.

I.3 Troisième épreuve

La façon dont cette partie est traitée est étrange. J'essaye ici de répondre aux question posées dans l'esprit (?) du sujet mais on peut faire plus simple.

Q.7 Pour traduire les inscription, on ne peut a priori se contenter des variables P_i des parties précédentes (ou alors il faudrait travailler avec une logique prenant trois valeurs de vérité).
Je propose de noter
— P_i la variable valant vrai ssi la clé verte est dans la boîte i
— R_i la variable valant vrai ssi la clé rouge est dans la boîte i
On a alors, en notant B_i l'inscription de la boîte i,

$$B_1 = \overline{P_3} \wedge \overline{R_3},\ B_2 = R_1,\ B_3 = \overline{P_3} \wedge \overline{R_3}$$

(on traduit la vacuité par l'absence de clé verte et de clé rouge).

Q.8 On sait déjà qu'il y a exactement une boîte avec clé verte et une boîte avec clé rouge (et donc une boite vide). Ce renseignement peut se traduire par

$$(P_1 \wedge \overline{P_2} \wedge \overline{P_3}) \vee (\overline{P_1} \wedge P_2 \wedge \overline{P_3}) \vee (\overline{P_1} \wedge \overline{P_2} \wedge P_3) \qquad (F_1)$$

en conjonction avec

$$(R_1 \wedge \overline{R_2} \wedge \overline{R_3}) \vee (\overline{R_1} \wedge R_2 \wedge \overline{R_3}) \vee (\overline{R_1} \wedge \overline{R_2} \wedge R_3) \qquad (F_2)$$

On sait aussi qu'il ne peut y avoir deux clés dans la même boîte, ce qui se traduit par

$$\overline{P_1 \wedge R_1} \wedge \overline{P_2 \wedge R_2} \wedge \overline{P_3 \wedge R_3} \qquad (F_3)$$

Par ailleurs, l'inscription de la boîte i est vraie ssi R_i est faux. On a donc aussi les trois formules $(R_i \wedge \overline{B_i}) \vee (\overline{R_i} \wedge B_i)$.
On peut noter que
— $\overline{R_3} \wedge B_3 = \overline{R_3} \wedge \overline{P_3} \wedge \overline{R_3} \equiv \overline{R_3} \wedge \overline{P_3}$ et $R_3 \wedge \overline{B_3} \equiv R_3 \wedge (P_3 \vee R_3) \equiv R_3$ et donc

$$(\overline{R_3} \wedge B_3) \vee (\overline{P_3} \wedge \overline{R_3}) \equiv (\overline{R_3} \wedge \overline{P_3}) \vee R_3 \equiv \overline{P_3} \vee R_3$$

— de manière similaire, on a

$$(\overline{R_2} \wedge B_2) \vee (\overline{P_2} \wedge \overline{R_2}) \equiv (R_1 \vee R_2) \wedge (\overline{R_2} \vee \overline{R_1})$$

On ajoute donc les trois formules

$$(\overline{R_1} \wedge \overline{P_3} \vee \overline{R_3}) \vee (R_1 \wedge (P_3 \vee R_3)) \qquad (F_4)$$

$$(R_1 \vee R_2) \wedge (\overline{R_2} \vee \overline{R_1}) \qquad (F_5)$$

$$\overline{P_3} \vee R_3 \qquad (F_6)$$

Les indications de l'animateur se traduisent par la véracité de $F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6$.

Q.9 Supposons, par l'absurde, que la clé verte soit dans la boîte 2. Ainsi, $P_2 = 1$ et $R_2 = 0$.
(F_5) indique que $R_1 = 1$ et la clé rouge est dans la boîte 1.
La boîte 3 est alors vide (car une boîte de chaque nature) et l'inscription de la boîte 1 est vraie ce qui contredit la règle (la boîte contenant la clé rouge a une inscription fausse).

Q.10 La clé verte ne peut être dans la boîte 3 (sinon l'inscription est juste et la boîte est vide). On vient de voir qu'elle n'est pas dans la boîte 2 et donc

La clé verte est dans la boîte 1

L'inscription de la boîte 1 est donc vraie et

La boîte 3 est vide

On en déduit que

La clé rouge est dans la boîte 2

II. Automates

II.1 Définitions

Q.11 On a

$$a^{-1}L = \{b, ab\}$$

Q.12 On a trois propriétés à prouver.
- Réflexivité : si $u \in \Sigma^*$, on a $u^{-1}L = u^{-1}L$ et donc $u \sim_L u$.
- Symétrie : si $u, v \in \Sigma^*$ vérifient $u \sim_L v$, on a $u^{-1}L = v^{-1}L$ et donc $v^{-1}L = u^{-1}L$ i.e. $v \sim_L u$.
- Transitivité : soient $u, v, w \in \Sigma^*$ tels que $u \sim_L v$ et $v \sim_L w$. Alors $u^{-1}L = v^{-1}L = w^{-1}L$ et donc $u \sim_L w$.

$\sim_L$ est une relation d'équivalence sur Σ^*

Supposons $u \sim_L v$, c'est à dire $u^{-1}L = v^{-1}L$.
- Soit $x \in (uw)^{-1}L$. On a $uwx \in L$ et donc $wx \in u^{-1}L = v^{-1}L$ et donc $vwx \in L$ ce qui donne $x \in (vw)^{-1}L$.
- De manière symétrique, si $x \in (vw)^{-1}L$ alors $x \in (uw)^{-1}L$.

Ainsi, $(uw)^{-1}L = (vw)^{-1}L$ et $uw \sim_L vw$.

$\sim_L$ est compatible avec la concaténation à droite

Q.13 (i) $\Lambda \in b^{-1}L$ (car $b \in L$) mais $\Lambda \notin (ab)^{-1}L$ (car $ab \notin L$). Ainsi

$$b \not\sim_L ab$$

(ii) $a \in (aba)^{-1}L$ (car $abaa \in L$) mais $a \notin (bab)^{-1}L$ (car $baba \notin L$). Ainsi

$$aba \not\sim_L bab$$

(iii) Si $u \in (abbaba)^{-1}L$ alors $|abbaba \cdot u|_a = 0[3]$ et donc $|u|_a = 0[3]$ et donc $|aaa \cdot u|_a = 0[3]$ et donc $u \in (aaa)^{-1}L$. La réciproque est identique et

$$abbaba \sim_L aaa$$

Q.14 L étant régulier, il est reconnu par un automate $A = (Q, \Sigma, q_0, F, \delta)$.
Considérons l'application φ définie de Q dans Q_L par

$$\varphi(q) = q^{-1}L$$

Soit $u \in \Sigma^*$ et $q = \delta^*(q_0, u)$. D'après la propriété admise par l'énoncé, $\varphi(q) = q^{-1}L = u^{-1}L$. L'application φ est donc surjective. Comme l'ensemble de départ est fini, il en est de même de l'ensemble d'arrivée (qui a même un cardinal inférieur ou égal). Ainsi, Q_L est fini.

II.2 Construction de l'automate minimal

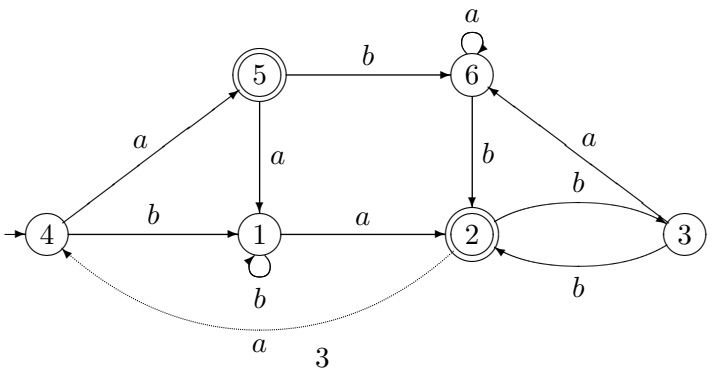
Q.15 Si $p \in F$ et $q \in Q \setminus F$ alors Λ distingue p et q. Il faut donc distinguer (p, q). Notons qu'u vu de l'avant dernière ligne de l'algorithme, il semblerait cohérent d'ajouter aussi (q, p) à l'ensemble N_0.

Q.16 Soit $j \geq 1$. Supposons $N_j \neq \emptyset$. Il existe alors $p, q \in Q$ et $u \in \Sigma^*$ de longueur j tels que $\delta^*(p, u) \in F$ et $\delta^*(q, u) \notin F$. Comme $j \geq 1$, u s'écrit $a \cdot v$ avec $v \in \Sigma^*$ de longueur $j - 1$ et $a \in \Sigma$.
Posons $p' = \delta(p, a)$ et $q' = \delta(q, a)$. On a alors $\delta^*(p', v) = \delta^*(q', v)$ et v distingue p' et q'.
Par ailleurs, si $w \in \Sigma^*$ est de longueur $\leq j-2$ alors $a \cdot w$ ne distingue pas p et q (car $(p, q) \in N_j$).
Or, $\delta^*(p', w) = \delta^*(p, a \cdot w)$ et $\delta^*(q', w) = \delta^*(q, a \cdot w)$ et donc w ne distigue pas p' et q'.
Ainsi, $(p', q') \in N_{j-1}$ et $N_{j-1} \neq \emptyset$. En contraposant, on a

$$N_{j-1} = \emptyset \Rightarrow N_j = \emptyset$$

ce qui donne le résultat demandé (par récurrence immédiate, si $N_i = \emptyset$ alors $\forall k \geq 0,\ N_{i+k} = \emptyset$).

Q.17



Q.18 - Fin de l'initialisation

	1	2	3	4	5	6
1		0			0	
2	0		0	0		0
3		0			0	
4		0			0	
5	0		0	0		0
6		0			0	

- Fin de l'étape 1

	1	2	3	4	5	6
1		0	1		0	1
2	0		0	0		0
3	1	0		1	0	
4		0	1		0	1
5	0		0	0		0
6	1	0		1	0	

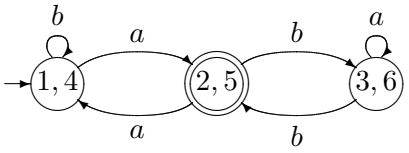
- Fin de l'étape 2

	1	2	3	4	5	6
1		0	1		0	1
2	0		0	0		0
3	1	0		1	0	
4		0	1		0	1
5	0		0	0		0
6	1	0		1	0	

Il n'y a pas eu d'évolution à l'étape 2 et il n'y en aura plus.
Les classes d'équivalences regroupent les sommets qui ne pourront être distingués. Il y en a trois qui sont

$$\{1,4\},\ \{2,5\},\ \{3,6\}$$

Q.19 On obtient un automate a trois états (l'état initial et les états finaux ne sont pas demandés).



III. Algorithmique et programmation

III.1 Transformation de Burrows-Wheeler (BWT)

Q.20

t	u	r	l	u	t	u	t	u	$\vert$
$\vert$	t	u	r	l	u	t	u	t	u
u	$\vert$	t	u	r	l	u	t	u	t
t	u	$\vert$	t	u	r	l	u	t	u
u	t	u	$\vert$	t	u	r	l	u	t
t	u	t	u	$\vert$	t	u	r	l	u
u	t	u	t	u	$\vert$	t	u	r	l
l	u	t	u	t	u	$\vert$	t	u	r
r	l	u	t	u	t	u	$\vert$	t	u
u	r	l	u	t	u	t	u	$\vert$	t

Q.21 On nous demande une fonction récursive. Il faut donc se demander comment permuter à droite un mot $x :: q$ sachant permuter à droite q. Supposons que $q = [q_1; \ldots; q_n]$. La permutation à droite de q est $[q_n; q_1; \ldots, q_{n-1}]$. Celle de $x :: q$ est alors $[q_n; x; q_1; \ldots; q_{n-1}]$. On cherche donc le permuté à droite de q à partir duquel on construit celui de $x :: q$ en intercalant x juste après la tête du permuté de q. Ceci n'est possible que si q est au moins de taille 2 et donc si le mot initial est de taille au moins 3. Notons que le cas d'un mot argument de taille ≤ 1 est un peu douteux (on ne peut rien décaler).

```
let rec circulaire mu = match mu with
   |[] -> []
   |[a] -> [a]
   |[a;b] -> [b;a]
   |x::q -> let mot = circulaire q in
            (List.hd mot)::x::(List.tl mot);;
```

Ceci étant, il m'aurait semblé plus naturel d'écrire une fonction auxilaire récursive `dernier : 'a list → 'a*'a list` telle que dans l'appel `dernier l`, on suppose `l` non vide et on renvoie le couple formé de la dernière lettre du mot et du mot amputé de cette dernière lettre.

```
let rec dernier l = match l with
   |[x] -> (x,[])
   |x::q -> let (y,u)=dernier q in (y,x::u);;
```

La fonction demandé s'en déduit (en supposant le mot argument non vide).

```
let circulaire mu=
    let (y,u)=dernier mu in y::u;;
```

Q.22 Ici, on ne demande plus une fonction récursive et on pourrait itérer en utilisant des références de listes. Par ailleurs, on n'impose a priori pas d'ordre pour la liste résultat mais il semble plus naturel d'obtenir celle qui était demandée plus haut (décalages successifs à droite).
Je choisit d'écrire une fonction récursive auxilaire `construit : int → 'a list → 'a list list`. Dans l'appel `construit k mot`, on renvoie la liste de taille k contenant `mot` et les `k-1` décalages suivants de ce ce mot.

```
let matrice_mot mu =
   let rec construit k mot =
      if k=0 then []
      else mot::(construit (k-1) (circulaire mot))
   in construit (List.length mu) mu;;
```

Q.23 La matrice M' est la suivante

|	t	u	r	l	u	t	u	t	u
l	u	t	u	t	u	|	t	u	r
r	l	u	t	u	t	u	|	t	u
t	u	|	t	u	r	l	u	t	u
t	u	r	l	u	t	u	t	u	|
t	u	t	u	|	t	u	r	l	u
u	|	t	u	r	l	u	t	u	t
u	r	l	u	t	u	t	u	|	t
u	t	u	|	t	u	r	l	u	t
u	t	u	t	u	|	t	u	r	l

et on a

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Q.24 Une fonction `insere : 'a → 'a list` prend en argument un élément, une liste triée par ordre croissant et renvoie la liste triée obtenue en ajoutant l'élément à la liste.
La fonction principale s'en déduit.

```
let rec insere x l = match l with
   |[] -> [x]
   |y::q -> if x<=y then x::l
            else y::(insere x q);;

let rec tri l = match l with
   |[] -> []
   |x::q -> insere x (tri q);;
```

Q.25 Il suffit de trier la matrice M des mots permutés.

```
let matrice_mot_triee mu =
    tri (matrice_mot mu) ;;
```

Notons qu'il peut y avoir un doute : Ocaml sait-il comparer deux listes de caractères et si oui, est-ce pour l'ordre lexicographique ? La réponse est OUI. On peut cependant écrire une fonction `compare : 'a list → 'a list → bool` tel que `compare mot1 mot2` indique si le premier argument est inférieur au second. Il suffit alors, dans `insere`, de remplacer `x<=y` par `compare x y`.

```
let rec compare mot1 mot2 = match (mot1,mot2) with
    |[],_ -> true
    |_,[] -> false
    |x1::q1,x2::q2 -> (x1<x2) || (x1=x2 && (compare q1 q2));;
```

Q.26 On peut penser qu'il s'agit ici d'estimer la complexité de la comparaison de deux mots de taille k, c'est à dire le nombre d'opérations dans l'appel `compare m1 m2` quand `m1` et `m2` sont deux listes de même taille k.
En notant C_k ce nombre d'opérations, on a $C_k = O(1)+C_{k-1}$ et ainsi $C_k = O(k)$. La complexité est linéaire en fonction de la taille.

Q.27 Une insertion dans une liste de taille p coûte au pire p comparaisons, c'est à dire $O(pk)$ opérations (en supposant que l'on travaille avec une liste composée de listes de taille k).
En notant C_p la complexité du tri d'une matrice de p permutations circulaires d'un mot de taille k, on a donc $C_p = O(pk) + C_{p-1}$. On en déduit que le tri de la matrice M a un côut $O\left(k\sum_{p=1}^{k} p\right) = O(k^3)$.

Q.28 Comme proposé, on écrit une fonction `last : 'a list → 'a` renvoyant la dernière lettre d'un mot supposé non vide.

```
let rec last l = match l with
```

```
    |[] -> failwith "erreur"
    |[x] -> x
    |x::q -> last q ;;
```

Il suffit alors de créer la matrice M' et de récupérer les dernières lettres. On écrit pour cela une fonction `parcours : 'a list list` $\to$ `'a list` renvoyant le mot correspondant à la dernière colonne de l'argument. On suppose ici que l'argument donné à `codageBWT` est le mot $\hat{\mu}$ (c'est à dire que le symbole | a déjà été ajouté).

```
let codageBWT mu =

   let rec parcours mat = match mat with
       |[] -> []
       |mot::q ->(last mot)::(parcours q)

   in parcours (matrice_mot_triee mu);;
```

D'après la question 23, dans le cadre de l'exemple, le codage est

$$uruu|utttl$$

Q.29 $\ell = BWT(\mu)$ contient exactement les mêmes lettres que $\hat{\mu}$ (par construction de M' où chaque lettre de $\hat{\mu}$ se retrouve une fois en dernière position d'une ligne).
Notons m la version triée de ℓ. Comme M' est triée par ordre lexicographique, m correspond alors exactement à la première colonne de M' (puisque dans cet ordre, on prend d'abord en compte la première lettre).
Dans le cadre de l'exemple, la première colonne sera

$$|adeegnnv$$

Q.30 Comme indiqué par l'énoncé, les sous-mots de taille 2 de $\hat{\mu}$ sont alors

$$e|, da, nd, ge, ve, ng, en, an, |v$$

Pour obtenir la colonne suivante, on ordonne cette liste de mots de taille 2

$$|v, an, da, e|, en, ge, nd, ng, ve$$

A nouveau, par définition de M', on obtient là les deux premières colonnes de M'. En particulier, la seconde colonne est

$$vna|nedge$$

Q.31 De façon générale, on suppose connues les $n-1$ première colonnes.
En ajoutant DEVANT la dernière colonne, on obtient tous les facteurs de taille n.
On ordonne ces facteurs de taille n et on obtient alors les n premières colonnes de M' (et donc la n-ième).

Q.32 L'algorithme est donc le suivant. On prend en argument le mot $\ell = BWT(\mu)$ que l'on suppose de taille k. On initialise une liste m de taille k composée de mots tous vides.

1. Ajouter devant chaque mot de m la lettre correspondante de ℓ (ceci modifie m en ajoutant une lettre à chaque mot).
2. Trier m (i.e. remplacer m par sa version triée)
3. retourner au point 1 si les éléments de m sont de taille $\leq k$
4. Renvoyer la ligne de m se terminant par |.

Q.33 En appliquant l'algorithme, on obtient la liste de listes suivante

```
[['|'; 'v'; 'e'; 'n'; 'd'; 'a'; 'n'; 'g'; 'e'];
 ['a'; 'n'; 'g'; 'e'; '|'; 'v'; 'e'; 'n'; 'd'];
 ['d'; 'a'; 'n'; 'g'; 'e'; '|'; 'v'; 'e'; 'n'];
 ['e'; '|'; 'v'; 'e'; 'n'; 'd'; 'a'; 'n'; 'g'];
 ['e'; 'n'; 'd'; 'a'; 'n'; 'g'; 'e'; '|'; 'v'];
 ['g'; 'e'; '|'; 'v'; 'e'; 'n'; 'd'; 'a'; 'n'];
 ['n'; 'd'; 'a'; 'n'; 'g'; 'e'; '|'; 'v'; 'e'];
 ['n'; 'g'; 'e'; '|'; 'v'; 'e'; 'n'; 'd'; 'a'];
 ['v'; 'e'; 'n'; 'd'; 'a'; 'n'; 'g'; 'e'; '|']]
```

Le mot décodé est donc

vendange

III.2 Codage par plages RLE (Informatique pour tous)

Q.34 Le résultat d'un codage RLE est naturellement une liste de tuples de taille 2 (des couples).

Q.35 On gère trois variables.
`res` est la liste code en construction à laquelle on ajoute des couples au fur et à mesure.
`car` est le caractère composant le bloc en cours de lecture.
`taille` est la taille du bloc en cours de lecture.
Initialement, on détecte un début de bloc de taille 1 composé du caractère numéro 0 du mot.
On parcourt le mot en incrémentant `taille` si le caractère rencontré complète le bloc ou en ajoutant un bloc à `res` sinon (en redémarrant un nouveau bloc). En fin de parcours, il reste à ajouter le couple correspondant au dernier bloc détecté.

```
def RLE(mot):
    res=[]
    car=mot[0]
    taille=1
    for i in range(1,len(mot)):
        if mot[i]==car:
            taille=taille+1
        else:
            res.append((taille,car))
            car=mot[i]
            taille=1
    res.append((taille,car))
    return res
```

Q.36 On gère une variable `mot` correspondant au mot décodé que l'on construit. Pour chaque couple `(taille,car)` du mot codé, on ajoute à la variable `mot` un nombre `taille` de carcatères `car`.

```
def decodeRLE(codeRLE):
    mot=[]
    for i in range(len(codeRLE)):
        (taille,car)=codeRLE[i]
        for j in range(taille):
            mot.append(car)
    return mot
```

III.3 Codage de Huffman (Informatique pour tous)

Q.37 Initialement, on a (on identifie une feuille et son étiquette qui est un couple)

$$\mathcal{L} = [(l,1),(r,1),(t,3),(u,4)] \text{ et } \mathcal{A} = []$$

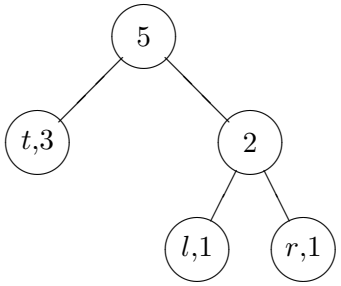
A la première étape, on utilise les deux premiers arbres dans $\mathcal{L}$ que l'on fusionne pour obtenir un arbre placé dans $\mathcal{A}$. Ainsi, en notant $\mathtt{N}(n,g,d)$ un noeud interne de poids n et de fils g et d,

$$\mathcal{L} = [(t,3),(u,4)] \text{ et } \mathcal{A} = [\mathtt{N}(2,(l,1),(r,1))]$$

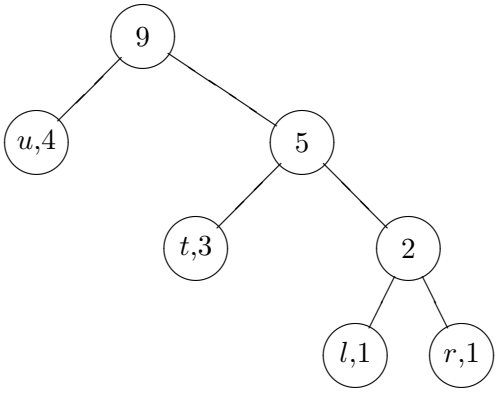
A l'étape suivante, on sélectionne les premiers éléments des deux listes. On les fusionne et on ajoute le résultat dans $\mathcal{A}$. On obtient

$$\mathcal{L} = [(u,4)] \text{ et } \mathcal{A} = [t]$$

où t est l'arbre binaire suivant



A l'étape suivante, on sélectionne les premiers éléments des deux listes. On les fusionne et on ajoute le résultat dans $\mathcal{A}$. Ce sera le résultat final



Q.38 Si μ est un mot où tous les symboles ont même nombre d'occurrences p, l'algorithme commence par "vider" $\mathcal{L}$ en ajoutant à $\mathcal{A}$ des arbres de hauteur 1 et de poids $2p$. $\mathcal{A}$ est alors une file dans laquelle on enlève à chaque étape 2 éléments pour en ajouter un. Le processus va donner un arbre équilibré en hauteur et donc touffu.