E3A / E4A

Concours d'admission 2014 - filière MP spécialité INFO

Corrigé de l'épreuve d'informatique

Exercice 1

I.1. On obtient facilement :

```
let Valide (i,j)= i>=0 && i<8 && j>=0 && j<8;;
```

I.2. On insère dans une liste L initialement vide les cases valides que l'on peut atteindre depuis la position (i, j) :

```
let CoupSuivant (i,j)=
   let V=[|(1,2);(-1,2);(1,-2);(-1,-2);(2,1);(-2,1);(2,-1);(-2,-1)|] and L = ref [] in
      for k=0 to 7 do
         let a,b=i+fst(V.(k)),j+snd(V.(k)) in
            if Valide (a,b) then
               L := (a,b)::(!L)
      done;
      !L;;
```

I.3. Nous utiliserons l'analyse suivante :

- nous initialisons la matrice M : la case (i_0, j_0) contient la valeur 1 et toutes les autres cases contiennent la valeur $+\infty$ (pratiquement, on représentera $+\infty$ par la valeur 100, puisqu'il n'y a que 64 cases sur l'échiquier et qu'il faut strictement moins de 100 coups pour aller de (i_0, j_0) à une case quelconque en un nombre minimal de coups) ;
- nous initialisons une référence i à la valeur 0 et une liste L à la valeur $[(i_0, j_0)]$: à chaque étape du calcul, L est la liste des cases que l'on peut atteindre depuis (i_0, j_0) en un nombre minimal de coups égal à i ;
- tant que L est non vide, on crée une liste LL contenant tous les successeurs des couples (a, b) stockés dans L et non encore rencontrés : on incrémente i, on affecte aux entrées de M associées à ces nouvelles cases la valeur (nouvelle) de i et on remplace L par LL.

Voici les deux premières étapes du fonctionnement de l'algorithme, quand $i_0 = j_0 = 0$:

- Initialisation : on définit M matrice 8×8 contenant la valeur 100 et on affecte à $M[0, 0]$ la valeur 0 ; on affecte à i la valeur 0 et à L la valeur $[(0, 0)]$.
- Première étape : les successeurs de $(0, 0)$ non encore rencontrés sont les éléments de la liste $LL = [(1, 2); (2, 1)]$: i prend la valeur 1, la matrice M devient (en notant ∞ au lieu de 100, pour faciliter la

lecture) :

$$M = \begin{pmatrix} 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 1 & \infty & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

et L devient $[(1,0);(0,1)]$.

- Deuxième étape : les successeurs de $(2,1)$ ou $(1,2)$ non encore rencontrés sont les éléments de la liste $LL = [(4,0);(0,2);(4,2);(1,3);(3,3);(3,1);(2,0);(0,4);(2,4)]$: i prend la valeur 2, la matrice M devient

$$M = \begin{pmatrix} 0 & \infty & 2 & \infty & 2 & \infty & \infty & \infty \\ \infty & \infty & 1 & 2 & \infty & \infty & \infty & \infty \\ 2 & 1 & \infty & \infty & 2 & \infty & \infty & \infty \\ \infty & 2 & \infty & 2 & \infty & \infty & \infty & \infty \\ 2 & \infty & 2 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

et L devient $[(4,0);(0,2);(4,2);(1,3);(3,3);(3,1);(2,0);(0,4);(2,4)]$.

Le calcul de LL se fait facilement : on initialise une liste LL à la valeur [], on lit l'un après l'autre chaque couple (a,b) stocké dans L, et pour chaque successeur (c,d) de (a,b) non encore rencontré, on modifie l'entrée (c,d) de M et on insère (c,d) dans LL. Le parcours des listes se fera bien évidemment au moyen de références :

```
let Cavalier (i0,j0)=
   let M=make_matrix 8 8 100 and i=ref 0 and L=ref [i0,j0] in
      M.(i0).(j0) <- 0;
      while !L<>[] do         (* tant qu'il y a des cases \`a \'etudier *)
         incr i;
         let LL= ref [] in                  (* on initialise LL *)
            while !L<>[] do                 (* on parcourt L *)
               let (a,b)=hd(!L) in          (* pour chaque \'el\'ement (a,b) de L *)
                  L := tl(!L);
                  let pile=ref (CoupSuivant (a,b)) in
                  while !pile<>[] do            (* on \'etudie chaque successeur de (a,b) *)
                     let (c,d)=hd(!pile) in
                     pile := tl(!pile);
                     if M.(c).(d)=100 then   (* si le successeur (c,d) est nouveau *)
                        begin
                        M.(c).(d) <- !i;           (* il est \`a la distance i de (i0,j0) *)
                        LL := (c,d)::(!LL)         (* et on l'ajoute \`a LL *)
                        end
                  done;
            done;                                  (* L a \'et\'e vid\'e *)
            L := !LL;                              (* L prend sa nouvelle valeur *)
      done;
      M;;
```

Exercice 2

II.1a L'automate possède un unique état initial et pour chaque état q, il existe exactement une transition partant de q et d'étiquette a (resp. b) : l'automate est donc déterministe complet.

II.1b Le mot w_0 est l'étiquette du chemin réussi :

$$0 \xrightarrow{a} 2 \xrightarrow{a} 4 \xrightarrow{b} 3 \xrightarrow{b} 2 \xrightarrow{b} 4 \xrightarrow{a} 1 \xrightarrow{b} 1 \xrightarrow{b} 1 \xrightarrow{a} 2 \xrightarrow{a} 4$$

II.1c Les mots $x = a$, $y = abb$ et $z = babbaa$ conviennent : on a bien $w_0 = xyz$, $|xy| = 4 \leq 5$, $y \neq \varepsilon$ et $xy^n z \in L_0$ pour tout n, puisque ce mot est l'étiquette du chemin réussi :

$$0 \xrightarrow{x} 2 \underbrace{\xrightarrow{y} 2 \cdots 2 \xrightarrow{y}}_{n \text{ fois}} 2 \xrightarrow{z} 4$$

II.2 $w = a_1 \dots a_n$ est dans L, donc il est associé au chemin $q_0 = e_0 \xrightarrow{a_1} e_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} e_n$, avec e_n état final. Comme $n \geq N$, les $N+1$ états $e_0, e_1, \dots, e_N$ ne peuvent être tous distincts (A ne possède que N états distincts). Il existe donc deux entiers i et j tels que $0 \leq i < j \leq N$ et $e_i = e_j$. Posons alors $q_1 = e_i = e_j$, $x = a_1 \dots a_i$, $y = a_{i+1} \dots a_j$ et $z = a_{j+1} \dots a_n$: nous avons $w = xyz$, $|xy| = j \leq N$, $|y| = j - i \neq 0$, $q_0.x = q_1$ et $q_1.y = e_i.y = e_j = q_1$.

II.3 Si L est un langage rationnel, il est aussi reconnaissable par automate (théorème de Kleene) : soit donc A un automate déterministe complet reconnaissant L et N le nombre d'état de A. Soit $w \in L$ de longueur supérieure ou égale à N et q_1, x, y et z les éléments obtenus à la question précédente. Nous avons alors $w = xyz$, $|xy| \leq N$, $y \neq \varepsilon$ et $xy^n zz \in L$ pour tout $n \in \mathbb{N}^*$, puisque $q_0.(xy^n z) = (q_0.x).y^n z = (q_1.y^n).z = q_1.z = e_n$ est un état final : L vérifie donc la propriété de l'étoile.

II.4 Montrons par l'absurde que X ne vérifie pas la propriété de l'étoile : il ne sera donc pas rationnel, par contraposée du résultat précédemment.

Supposons donc qu'il existe $N \in \mathbb{N}$ vérifiant la condition caractéristique de la propriété de l'étoile et soit $w = a^N b^N$. Comme $|w| \geq N$ et $w \in X$, il existe x, y, z tels que $w = xyz$, $|xy| \leq N$, $y \neq \varepsilon$ et $xy^n z \in X$ pour tout $n \in \mathbb{N}$. La condition $|xy| \leq N$ impose que $xy = a^{i+j}$, avec $i = |x|$ et $j = |y|$. On a alors $xz \in X$, ce qui est absurde car $xz = a^{N-j} b^N$ avec $N - j < N$ (puisque y n'est pas le mot vide).

II.5 Soit $X_2 = \{a^{n!},\ n \in \mathbb{N}\}$. Une nouvelle fois, montrons par l'absurde que X_2 n'a pas la propriété de l'étoile : sinon, en choisissant encore un entier N caractéristique de cette propriété, on pourrait écrire le mot $w = a^{N!}$ de X_2 sous la forme $w = xyz$ avec $|xy| \leq N$, $y \neq \varepsilon$ et $xy^n z \in X_2$ pour tout $n \in \mathbb{N}$. En notant i, j et k les longueurs respectives de x, y et z, pour tout entier naturel n, il existe un unique entier naturel m_n tel que $i + jn + k = (m_n)!$. Comme j est non nulle, la suite (m_n) est strictement croissante, ce qui donne :

$$\forall\, n \in \mathbb{N},\ i + jn + k = (m_n)! \geq (m_0 + n)!$$

qui est absurde car $i + jn + k$ est négligeable devant $(m_0 + n)!$ quand n tend vers l'infini.

Ainsi X_2 n'est pas rationnel, mais $a^* X_2 = \{a^{n+m!},\ n, m \geq 0\} = \{a^k,\ k \geq 1\}$ est rationnel.

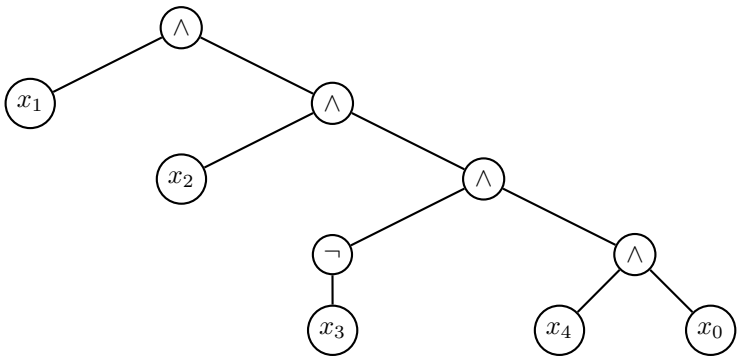
II.6 Il y a une erreur grossière d'énoncé. En effet, pour tout entier $N \geq 1$, le mot $w = a^N b^N$ est élément de Y et si on l'écrit sous la forme $w = xyz$ avec $y \neq \varepsilon$ et $|xy| \leq N$, x et y sont nécessairement des puissance de a. Il

existe ainsi $j \geq 1$ tel que $y = a^j$ et $xy^0z = a^{N-j}b^N \notin Y$, donc $xy^*z \not\subset Y$. Comme $N = 0$ ne peut pas non plus vérifier la propriété demandée (car Y est non vide), le langage Y ne vérifie pas le lemme de l'étoile.

La question b) n'a alors plus d'objet, puisqu'un langage ne vérifiant pas le lemme de l'étoile ne peut être rationnel. L'auteur du sujet voulait évidemment donner un exemple de langage non rationnel mais vérifiant le lemme de l'étoile.

Exercice 3

III.1 La formule $x_1 \wedge (x_2 \wedge ((\neg x_3) \wedge (x_4 \wedge x_0)))$ est représentée par l'arbre :



III.2 L'évaluation se fait récursivement de façon évidente :

```
let rec eval n phi t = match phi with
     Var(i) -> t.(i)
   | OR(phi1,phi2) -> (eval n phi1 t) || (eval n phi2 t)
   | AND(phi1,phi2) -> (eval n phi1 t) && (eval n phi2 t)
   | NOT(phi1) -> not(eval n phi1 t);;
```

III.3 On obtient directement :

```
let rec maxVar phi = match phi with
     Var(i) -> i
   | OR(phi1,phi2) -> max (maxVar phi1) (maxVar phi2)
   | AND(phi1,phi2) -> max (maxVar phi1) (maxVar phi2)
   | NOT(phi1) -> maxVar phi1;;
```

III.4 L'énoncé demande d'écrire un algorithme naïf : pour une formule φ, on calcule l'entier $n = \texttt{maxVar}(\varphi)$ puis on parcourt l'ensemble des interprétations des variables de X_n à la recherche d'une interprétation t telle que $\texttt{eval}(n, \varphi, t) = \texttt{vrai}$. Si on trouve une telle interprétation, on arrête le calcul : la formule est satisfiable. Si on a éclusé toutes les interprétations, la formule n'est pas satisfiable. Nous aurons donc besoin de parcourir toutes les interprétations, par exemple dans l'orde lexicographique :

$$\begin{aligned} t_0 &= [|\texttt{faux}; \ldots; \texttt{faux}; \texttt{faux}; \texttt{faux}|] \\ t_1 &= [|\texttt{faux}; \ldots; \texttt{faux}; \texttt{faux}; \texttt{vrai}|] \end{aligned}$$

$$
\begin{aligned}
t_2 &= [|\texttt{faux}; \ldots; \texttt{faux}; \texttt{vrai}; \texttt{faux}|] \\
&\vdots \\
t_i &= [|\ldots, \texttt{faux}, \underbrace{\texttt{vrai}, \texttt{vrai}, \ldots, \texttt{vrai}}_{k \text{ termes}}|] \\
t_{i+1} &= [|\ldots, \texttt{vrai}, \underbrace{\texttt{faux}, \texttt{faux}, \ldots, \texttt{faux}}_{k \text{ termes}}|] \\
&\vdots \\
t_{2^n-1} &= [|\texttt{vrai}; \ldots; \texttt{vrai}; \texttt{vrai}; \texttt{vrai}|]
\end{aligned}
$$

Nous utiliserons un vecteur t de longueur $n+1$ initialisé à la valeur t_0 et une fonction `suivant` permettant de transformer t en l'interprétation suivante. Quand t représente t_i, avec $i < 2^n - 1$, `suivant` t transforme t en t_{i+1} et renvoie le booléen `vrai` et quand t représente t_{2^n-1}, `suivant` t transforme t en $[|\texttt{faux}; \texttt{faux}; \ldots; \texttt{faux}|]$ et renvoie le booléen `faux` :

```
let suivant t =
   let n=vect_length t in
   let i=ref (n-1) in          (* on part de la fin du tableau *)
   while !i>=0 && t.(!i) do    (* tant qu'on rencontre true *)
      t.(!i) <- false;               (* on le remplace par false *)
      i := (!i)-1                    (* et on d\'ecr\'emente i  *)
   done;
   if !i<0 then              (* si i=-1 *)
      false                       (* on est arriv\'e \`a la derni\`ere interpr\'etation *)
   else                      (* sinon *)
      begin
      t.(!i) <- true;             (* on remplace le premier false rencontr\'e par true *)
      true
      end;;
```

Il reste à écrire la fonction `satisfiable` : le booléen `résultat` prend la valeur `vrai` dès qu'une interprétation satisfait la formule, permettant ainsi d'arrêter le calcul :

```
let satisfiable phi=
   let n = maxVar phi in
   let t = make_vect (n+1) false in           (* on initialise t \`a la valeur t0 *)
   let resultat = ref (eval n phi t) in       (* resultat = eval(phi,t0) *)
   while not(!resultat) && (suivant t) do
      resultat:= eval n phi t
   done;
   !resultat;;
```

III.5 L'analyse est identique, à ceci près que l'on sort de la boucle dès que l'on a trouvé une interprétation qui donne à la formule la valeur de vérité `faux` :

```
let tautologie phi=
   let n = maxVar phi in
   let t = make_vect (n+1) false in           (* on initialise t \`a la valeur t0 *)
   let resultat = ref (eval n phi t) in       (* resultat = eval(phi,t0) *)
```

```
while (!resultat) && (suivant t) do
   resultat:= eval n phi t
done;
!resultat;;
```

On aurait évidemment pu écrire :

```
let tautologie phi = not(satisfiable (NOT(phi)));;
```

Exercice 4

IV On peut construire les tables de vérité des fonctions f et g (on note $\dot\vee$ le ou exclusif) :

x	y	$x \wedge y$	$x \vee y$	$x \dot\vee y$	$x \Longrightarrow y$	$y \Longrightarrow x$	$f(x,y)$	$g(x,y)$
V	V	V	V	F	V	V	F	V
V	F	F	V	V	F	V	F	F
F	V	F	V	V	V	F	V	F
F	F	F	F	F	V	V	F	V

On en déduit que $f(x,y) \equiv (\neg x) \wedge y$ et $g(x,y) \equiv \neg(x \dot\vee y)$, ce qui donne les circuits contenant un nombre minimal de portes (on ne peut évidemment pas construire f ou g avec une seule porte) :

Exercice 5

V.1 On initialise T à la valeur $[0,0,0,0]$ puis on parcourt A :

- $i = 0$: j vaut 1 et on incréments la case 1 de T, qui devient $[0,1,0,0]$;
- $i = 1$: j vaut 2 et on incréments la case 2 de T, qui devient $[0,1,1,0]$;
- $i = 2$: j vaut 3 et on incréments la case 3 de T, qui devient $[0,1,1,1]$;
- $i = 3$: j vaut 1 et on incréments la case 1 de T, qui devient $[0,2,1,1]$;
- $i = 4$: j vaut 2 et on incréments la case 2 de T, qui devient $[0,2,2,1]$;
- $i = 5$: j vaut 3 et on incréments la case 3 de T, qui devient $[0,2,2,2]$.

V.2 L'invariant de boucle est le suivant : au moment du passage à la ligne 3, chaque case m du tableau T contient le nombre d'élément du tableau $[A[0], A[1], \dots A[i-1]]$ dont la clef est égale à m, valeur que nous noterons $occ(m, i-1)$.

- la propriété est bien vérifiée au premier passage dans 3, i.e. quand $i = 0$, puisque T ne contient que des 0 ;

- supposons que la propriété soit vérifiée quand le compteur i prend une certaine valeur i_0 strictement inférieure à la longueur de A. A la ligne 4, j reçoit la valeur $m_0 = \text{clef}(A[i_0])$ et on ajoute à la ligne 5 une unité à la case m_0 de T et nous avons après l'instruction 5 :

$$\forall m \in \{0, \ldots, k\},\ T[m] = \begin{cases} occ(m_0, i_0 - 1) + 1 = occ(m_0, i0) & \text{si } m = m_0 \\ occ(m, i_0 - 1) = occ(m, i_0) & \text{sinon} \end{cases}$$

  donc la propriété est vérifiée lors du passage suivant à la ligne 3, puisque i pend maintenant la valeur $i_0 + 1$

Nous obtenons donc par récurrence que la propriété est vérifiée au dernier passage par la ligne 3, i.e. quand $i = \texttt{longueur}(A)$, ce qui prouve que le tableau T renvoyé contient, dans chaque case m, le nombre d'éléments du tableau A possédant m pour clef.

V.3 Soient $[a_0, a_1, \ldots, a_k]$ la valeur d'entrée du tableau T. T devient successivement $[a_0, a_0 + a_1, a_2, \ldots, a_k]$, $[a_0, a_0 + a_1, a_0 + a_1 + a_2, a_3, \ldots, a_k]$, jusqu'à $[a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + a_1 + \cdots + a_k]$. Après l'appel $\texttt{Algo2}(T)$, chaque case p de T contient $a_0 + \cdots + a_p$.

V.4 Soit n la longueur du tableau A. Pour simplifier l'écriture, si T est un tableau de taille n et si $0 \leq i \leq j \leq n-1$, nous noterons $T[i; j] = [T[i], \ldots, T[j]]$. Notons, comme ci-dessus, a_p le nombre de valeurs stockées dans A possédant p pour clef. Nous voulons donc construire un tableau B tel que pour tout p compris entre 0 et k, chaque "paquet" d'éléments

$$B[a_0 + \cdots + a_{p-1}; a_0 + \cdots + a_{p-1} + a_p - 1]$$

contienne les a_p éléments stockés dans A de clef p. L'invariant de boucle qui va assurer que `Algo3` fait bien ce travail peut s'exprimer ainsi : à chaque passage dans la ligne 16, les éléments de $A[n - i + 1; n - 1]$ ont été placés dans B et pour chaque clef p, les éléments de $A[n - i + 1; n - 1]$ de clef p sont rangés dans le "bon paquet" dans le sous-tableau $B[T[p]; a_0 + \cdots + a_p - 1]$.

Cette propriété est bien vérifiée au premier passage à la ligne 16, puisque $i = 1$, aucune valeur n'a été rangée dans B et chaque $T[p]$ vaut $a_0 + \cdots + a_p$: le paquet $B[T[p]; a_0 + \cdots + a_p - 1]$ est bien vide.

Soit i compris entre 1 et $n - 1$ et supposons que la propriété est vérifiée au i-ème passage dans la ligne 16. Pour chaque clef p, soit t_p la valeur contenue dans $T[p]$ à cet instant du calcul. On donne alors à j la valeur $n - i$ et à p la valeur $\texttt{clé}(A[n - i])$. $T[p]$ prend la valeur $t_p - 1$ et on place $A[n - i]$ dans la case $B[t_p - 1]$. Pour chaque clef q distinctes de p, les éléments de $A[n - i; n - 1]$ de clef q sont donc rangés dans le sous-tableau $B[T[q]; a_0 + \cdots + a_q - 1]$, puisque $T[q] = t_q$. Les éléments de $A[n - i; n - 1]$ de clef p sont ceux de $A[n - i + 1 \ldots n - 1]$ augmentés de l'élément $A[n - i]$. Comme ceux de $A[n - i + 1; n - 1]$ sont rangés dans le sous-tableau $B[t_p; a_0 + \cdots + a_p - 1]$ et que $B[t_q - 1] = A[n - i]$, ils sont donc rangés dans le sous-tableau $B[t_p - 1; a_0 + \cdots + a_p - 1] = B[T[p]; a_0 + \cdots + a_p - 1]$ et l'invariant est vérifié au rang $i + 1$.

On en déduit que la propriété est vérifiée au dernier passage dans la ligne 16, quand $i = n + 1$: le tableau B contient alors les éléments de A rangés par ordre croissant de clefs.

V.5 On pourrait compléter la preuve précédente en précisant que les éléments de $A[n - i + 1; n - 1]$ de clef p sont rangés dans le sous-tableau $B[T[p]; a_0 + \cdots + a_p - 1]$ dans le même ordre que celui dans lequel ils apparaissaient dans A. L'auteur attend peut-être une preuve plus globale : notons p la clef commune de a et de b, j_a et j_b les indices [1] tels que $A[j_a] = a$ et $A[j_b] = b$ et posons $i_a = n - j_a$ et $i_b = n - j_b$. Sans

1. L'énoncé est peu précis : il devrait raisonner en terme de place dans le tableau A, et pas en terme de valeurs (sauf si l'on suppose qu'un valeur ne peut apparaître deux fois dans A). Plus précisément, `algol` construit une permutation σ de $0, \ldots, n - 1$

perte de généralité, supposons que a est avant b dans le tableau A, i.e. que $i_b < i_a$. L'élément b est copié le premier dans le tableau B quand l'indice i dans la boucle (ligne 16) prend la valeur i_b : le contenu de $T[p]$ est décrémenté (ligne 19) et b est copié dans la case $k_b = T[p]$ de B (ligne 20). Ensuite, quand i prendra la valeur i_a, le contenu de $T[p]$ sera une nouvelle fois décrémenté (ligne 19) et a sera copié dans la case $k_a = T[p]$ de B (ligne 20). Comme les contenus des cases du tableau B décroissent, on a $k_a < k_b$ (plus précisément k_a est strictement inférieur à la valeur de $T[p]$ avant la dernière décrémentation, qui est inférieure ou égale à la valeur k_b). On en déduit donc que a et b sont stockés dans le même ordre dans B que dans A.

V.6 Notons n la longueur de A. L'algorithme `Algo1` a une complexité en $O(n+k)$ (la création de T demande un temps de l'ordre de k, puis la boucle sur i demande un temps de l'ordre de n puisque chaque passage dans la boucle 4-5 se fait en temps constant). L'algorithme `Algo2` a une complexité en $O(\text{longueur}(T))$, i.e. en $O(k)$. Enfin, chaque passage dans la boucle 17-20 se fait en temps constant, donc la complexité de l'appel `Algo2`(A) est $O(n) + O(n+k) + O(k) + O(n)$, ce qui donne un $O(n+k)$.

V.7 `Algo4` renvoie un tableau contenant les même valeur que A, mais rangées dans l'ordre croissant. L'invariant de boucle est qu'au i-ème passage dans la ligne 25, B contient les mêmes valeurs que A ordonnées "par rapport à leurs $i-1$ premiers chiffres" (il y a une maladresse dans l'énoncé : il faut comprendre que le premier chiffre est celui des unités ... par exemple, en base 10, le premier chiffre de 137 est 7, le deuxième est 3 et le troisième est 1). Si on veut être plus précis, en notant N_i le reste modulo k^{i-1} d'un entier N, nous avons :

$$\forall j \in \{0, \ldots, n-1\},\ B[1]_i \leq B[2]_i \leq \ldots \leq B[n]_i$$

Nous traduirons cette propriété en disant que B est " croissant modulo k^{i-1}".

La propriété est vérifiée au premier passage dans la boucle : i vaut 1, B est égal à A et les éléments sont triés modulo 1.

Soit $i < d$ et supposons que la propriété est vérifiée au i-ème passage dans la ligne 25. Les valeurs prises par la fonction $\texttt{chiffre}_{k,i}$ sont comprises entre 0 et $k-1$, donc `Algo3` est appelée avec des paramètres cohérents : les valeurs stockées dans B après l'instruction 26 sont les valeurs de A classées par ordre croissant de leur i-ème chiffre et en cas d'égalité, l'ordre modulo k^{i-1} est préservé (question 5) : B est donc maintenant classé modulo k^i et la propriété est vérifiée au passage suivant dans la boucle, après que i ait été incrémenté.

Ainsi, au dernier passage dans la boucle, i vaut $d+1$ et B contient les valeurs de A classées modulo k^d, i.e. triées par ordre croissant puisque les nombres stockés dans A ont au plus d chiffres en base k.

V.8 Chaque passage dans la boucle 25-27 est de complexité $O(n+k)$, donc la complexité du tri `Algo4` est $O(n) + d\,O(n+k)$, soit $O(d(n+k))$.

V.9a On choisit $k = 2^r$. Les éléments de A sont strictement majorés par $2^b = (2^r)^{b/r}$. En posant $d = \lceil b/r \rceil$, nous aurons besoin d'au plus d chiffres pour écrire les éléments de A, donnant le temps demandé.

V.9b En prenant $r = \lceil \ln_2 n \rceil$, $k = 2^r$ et $d = \lceil b/r \rceil$, d est majoré par λ et 2^r est équivalent à n : on obtient bien un temps linéaire en n.

V.10 Les algorithmes de tri efficaces (tri fusion, tri par tas) ont une complexité en $O(n \ln n)$: le "tri baquet" présenté ici est plus efficace, mais nécessite une hypothèse forte sur la taille des données à trier.

---

telle qu'à la fin du calcul, on ait $A[j] = B[\sigma(j)]$ pour tout j. La question posée s'écrit alors :

$$\forall j_1, j_2 \in \{0, \ldots, n-1\},\ (\texttt{clef}(A[j_1]) = \texttt{clef}(A[j_2]) \text{ et } j_1 < j_2) \Longrightarrow \sigma(j_1) < \sigma(j_2).$$