TD 3 : terminaison et correction des algorithmes

On corrige d'abord les exercices Exercices I à IV

Exercice 1 – Premier exemple

Considérons la fonction `expo : int -> int -> int` répondant à la spécification suivante :

Précondition : $n \geq 0$
Postcondition : `expo(a, n)` $= a^n$

Python

```python
def expo(a, n):
    p = n
    x = 1
    b = a
    while p != 0:
        if p % 2 == 1:
            x = x * b
        b = b * b
        p = p // 2
    return x
```

▶ **Question 1** Montrer que "$xb^p = a^n$ et $p > 0$" est un invariant de boucle.

Supposons que $xb^p = a^n$ et $p \geq 0$ au début d'un tour de la boucle `while`. Alors à la fin de la boucle on a $p' = \left\lfloor \frac{p}{2} \right\rfloor \geq 0$:

- Si p est pair, alors $x' = x$, $b' = b^2$ et $p' = \frac{p}{2}$, donc $x'(b')^{p'} = x(b^2)^{\frac{p}{2}} = xb^p = a^n$
- Si p est impair, alors $x' = xb$, $b' = b^2$ et $p' = \frac{p-1}{2}$, donc $x'(b')^{p'} = xb(b^2)^{\frac{p-1}{2}} = xb^{1+p-1} = xb^p = a^n$.

Donc "$xb^p = a^n$ et $p \geq 0$" est bien un invariant de boucle.

▶ **Question 2** Montrer la correction partielle, puis la correction totale, de la fonction.

Supposons que la fonction termine, c'est-à-dire que la boucle termine. Alors, on a $p = 0$ par la condition de sortie de la boucle et $xb^p = x = a^n$ par l'invariant de boucle, et x est la valeur de retour de `expo(a, n)` donc la fonction est partiellement correcte. Or on a :

- p est un entier
- avant d'arriver dans la boucle, on a $p = n \geq 0$ d'après la précondition de la fonction
- Si la condition de boucle est vérifiée, on a $p' = \left\lfloor \frac{p}{2} \right\rfloor < p$ et $p' \geq 0$

On a donc p qui est strictement décroissant à chaque tour de boucle, minoré par 0, donc c'est un invariant de boucle. Ainsi, la fonction termine pour toute entrée admissible et est partiellement correcte, elle est donc totalement correcte.

Exercice 2 – Algorithme de recherche dichotomique

Considérons maintenant la fonction de recherche dichotomique, répondant à la spécification suivante :

Entrée : un tableau $t = [t_0, \dots, t_{n-1}]$, un élément x
Sortie : s'il existe, un indice $i \in [\![0, n-1]\!]$ tel que $t_i = x$, n sinon

▶ **Question 1** Écrire une première fonction `recherche_naive : 'a array -> 'a -> int` en pseudo-code répondant à cette spécification.

Modifions un peu la spécification attendue :

Entrée : un tableau $t = [t_0, \dots, t_{n-1}]$, un élément x
Précondition : t est trié
Sortie : s'il existe, un indice $i \in [\![0, n-1]\!]$ tel que $t_i = x$, sinon n

Pseudo-code

```
1 : fonction RECHERCHE(x, t) :
2 :     deb, fin ← 0, n
3 :     Tant que fin − deb > 0, faire :
4 :         milieu ← (deb + fin)/2                ▷ division entière
5 :         Si t_milieu = x alors
6 :             Renvoyer milieu
7 :         Sinon, si t_milieu < x
8 :             deb ← milieu + 1
9 :         Sinon
10 :            fin ← milieu
11 :        Fin si
12 :    Fin tant que
13 :    Renvoyer n
14 : Fin fonction
```

▶ **Question 2** Montrer la terminaison de `Recherche`.

La terminaison de la fonction revient à la terminaison de la boucle **Tant que**. Montrons que $fin - deb$ est un variant de boucle.

- $fin - deb$ est bien un entier
- Puisque initialement $fin - deb = n - 0 \geq 0$ et par la condition de boucle, $fin - deb$ est bien minoré par 0 au cours de l'exécution.
- On a donc $deb < fin$ donc $deb \leq fin - 1$. Ainsi, $deb < \frac{deb+fin}{2} < \frac{2fin-1}{2}$, donc $deb \leq \left\lfloor \frac{deb+fin}{2} \right\rfloor \leq fin - 1$, c'est-à-dire $deb \leq milieu < fin$. Ensuite, on a trois cas parmi les conditionnelles :
  - Soit on sort de la boucle
  - Soit $deb' = milieu + 1 > deb$ et $fin' = fin$ donc $fin' - deb' > fin - deb$
  - Soit $deb' = deb$ et $fin' = milieu < fin$ donc $fin' - deb' > fin - deb$

$fin - deb$ est donc bien un variant de boucle, donc la fonction termine.

▶ **Question 3** Montrer que si l'algorithme renvoie un entier $i < n$, alors ce résultat est correct. A-t-on déjà la correction partielle ?

Supposons que l'algorithme renvoie un entier $i < n$. Alors il le renvoie grâce à la ligne 7, ce qui n'est possible que si $t_i = x$. Si l'algorithme renvoie un entier différent de n, alors c'est bien un entier tel que $t_i = x$ conformément à la spécification.
On n'a pas montré la correction partielle, puisqu'on n'a pas encore étudié le cas où l'algorithme renvoie n.

▶ **Question 4** Pour $a \leq b$, notons $t\,[a : b]$ la portion du tableau comprise entre les indices a inclus et b exclu. Montrer que la boucle respecte l'invariant suivant :

$$x \notin t\,[0 : deb] \text{ et } x \notin t\,[fin : n]$$

Initialisation : Avant le premier passage de boucle, on a $deb = 0$ et $fin = n$, donc les deux tranches $t\,[0 : deb]$ et $t\,[fin : n]$ sont vides et l'invariant est trivialement vérifié.

Hérédité : Supposons qu'en début de boucle on a $x \notin t\,[0 : deb] \cup t\,[fin : n]$. Alors :

- Si $t_{milieu} = x$, alors on sort de la fonction ce qui règle le problème *dans une preuve d'invariant, on veut que l'invariant soit toujours vrai à la fin de la boucle. Ainsi, si on a un cas où l'on sort de la boucle comme ici, la question ne se pose plus.*
- Si $t_{milieu} < x$, alors on sait que $x \notin t\,[deb : milieu + 1]$ puisque le tableau est trié. Ainsi, on a $x \notin t\,[deb : milieu + 1] \cdot t\,[0 : deb] = t\,[0 : deb']$ et $x \notin t\,[fin' : n]$.
- Si $t_{milieu} > x$, alors on sait que $x \notin t\,[milieu : n]$ puisque le tableau est trié. Or $x \notin t\,[0 : deb]$ donc $x \notin t\,[0 : deb] \cup t\,[milieu : n] = t\,[0 : deb'] \cup t\,[fin' : n]$.

Conclusion : La propriété est donc bien un invariant de boucle.

Cette opération est appelée *slicing*, et dispose d'une syntaxe dédiée en Python, mais pas en OCaml ni en C.

▶ **Question 5** En déduire la correction (totale) de l'algorithme.

Si l'on sort de la boucle en passant par la ligne 7, alors par la question 3 le résultat est correct. Sinon, on sort de la boucle avec $fin - deb \leq 0$, c'est-à-dire $fin \leq deb$ et donc l'invariant de boucle implique que $x \notin t\,[0 : n]$, et l'on renvoie n qui est le résultat attendu. Or tout autre résultat que n est aussi correct par la question 3, et l'algorithme termine, donc il est totalement correct.

Exercice 3 – Correction du tri par insertion et du tri fusion

Pour cet exercice, vous avez besoin de reprendre le code de votre fonction de tri par insertion et de tri fusion écrite en DM, corrigée si besoin à l'aide de l'indication de corrigé donnée sur `cahier-de-prepa.fr`.

▶ **Question 1** Formaliser les spécifications des fonctions `insere` et `tri_insertion`, avec la présentation et le vocabulaire vu en cours.

Pour rappel, les fonctions de la correction :

OCaml
```
let rec insere l x = match l with
  | [] -> [x]
  | h :: t ->
    if h < x then h :: insere t x else x :: l

let rec tri_insertion l = match l with
  | [] -> []
  | h :: t -> insere (tri_insertion t) h

let rec divise l = match l with
  | [] -> ([], [])
  | [x] -> ([x], [])
  | h1 :: h2 :: t ->
    let (tg, td) = divise t in
    (h1 :: tg, h2 :: td)

let rec fusionne l1 l2 = match l1, l2 with
  | [], l | l, [] -> l
  | h1 :: t1, h2 :: t2 ->
    if h1 <= h2 then
      h1 :: fusionne t1 l2
    else
      h2 :: fusionne l1 t2

let rec tri_fusion l = match l with
  | [] | [_] -> l
  | h :: t ->
    let lg, ld = divise l in
    fusionne (tri_fusion lg) (tri_fusion ld)
```

Pour `insere` :
Entrée : une liste `l` et un élément `x`.
Précondition : `l` est trié
Sortie : une liste triée contenant tous les éléments de `x :: l`.
Pour `tri_insertion` :
Entrée : une liste `l`
Sortie : une liste triée des éléments de `l`.

▶ **Question 2** Montrer la correction de ces fonctions.

On va montrer la terminaison et la correction de `insere` par récurrence sur la longueur de `l`, notée $n \in \mathbb{N}$.

Initialisation : Pour $n = 0$, l'appel `insere l x [x]` est bien trié et composé des éléments de `x :: []`.

Hérédité : Soit `l` une suite triée de taille $n \in \mathbb{N}^*$ et `x` un élément de la liste. Alors, on a deux cas :

— Si $h < x$ avec h la tête de la liste, alors par hypothèse de récurrence l'appel `insere x t` renvoie une liste triée contenant les éléments de `x :: t`, et h est strictement inférieur à tous ces éléments (car $h < x$ et `t` est trié), donc `h :: insere t x` est trié et contient tous les éléments de `h :: t = l` et `x`.

— Sinon, x est inférieur ou égal à tous les éléments de `l` car `l` est trié, donc `x :: l` est trié.

Donc `insere` est correct. Par une récurrence très similaire et par correction de `insere`, `tri_insertion` est correct.

▶ **Question 3** Faire de même pour le `tri_fusion`.

Pour `divise` :

Entrée : une liste `l`

Sortie : un couple de listes `lg,ld` tel que leur concaténation contient exactement les mêmes éléments que `l` et $\big||lg| - |ld|\big| \leq 1$.

Pour `fusionne` :

Entrée : un couple de listes `(l1,l2)`

Précondition : `l1` et `l2` sont triées

Sortie : une permutation de la liste `l1 @ l2` triée

Pour `tri_fusion`, c'est la même spécification que `tri_insertion`.

Montrons la terminaison / correction de `divise` par récurrence double sur la taille de `l`, notée n :

Initialisation : Pour $n = 0$, une partition de `[]` donne forcément `[],[]` qui respecte bien la contrainte sur la différence de taille. Pour $n = 1$, `[x],[]` est bien une partition de `[x]` respectant la contrainte sur la différence des tailles. On a donc la terminaison et la correction pour $n \geq 1$.

Hérédité : Pour $n \geq 2$, soit `l = h1 :: h2 :: t` de taille n et supposons que l'appel `divise t = (lg, ld)` termine et soit correct, donc l'appel `divise l` termine. Ainsi, on a $\big||lg| - |ld|\big| \leq 1$, donc $\big||h1 :: lg| - |h2 :: ld|\big| = \big|1 + |lg| - (1 + |ld|)\big| \leq 1$. Ainsi, l'appel `divise l` termine et est correct.

Pour la fusion, on va le montrer par récurrence sur la somme des tailles des deux listes en entrée.

Initialisation : Pour $n = 0$, les deux listes sont forcément `[]`,`[]`. Ainsi, l'appel termine par le premier cas et renvoie la sortie correcte `[]`.

Hérédité : Supposons que $n \in \mathbb{N}$. Alors, on a trois cas :

— Si l'une des deux listes est vide, l'appel termine par le premier cas du filtrage et puisque l'on suppose que les deux listes sont triées, l'autre liste est triée : ainsi, la renvoyer en sortie correspond à la spécification.

— Sinon, reprenons les notations du deuxième filtrage. Puisque $|t1| + |l2| = |l1| + |t2| = n - 1$, et que `t1`, `t2`, `l1`, `l2` sont triés, les appels `fusionne t1 l2` et `fusionne l1 t2` terminent par hypothèse d'induction. De plus, on trouve un plus petit élément de `l1 @ l2` avec `h1` ou `h2` : en effet, `l1` et `l2` sont triées, donc `h1`, resp. `h2`, est le plus petit élément de `l1`, resp. `l2`. Ainsi, si $h1 \leq h2$, alors $h1$ est un plus petit élément de `l1 @ l2` : puisque l'appel `fusionne t1 l2` renvoie une permutation triée de `t1 @ l2`, la valeur retournée est bien une permutation triée de `l1 @ l2`. Similairement, on a aussi la correction dans le cas $h1 > h2$.

D'où la correction de `fusionne`. Et on tire la correction du `tri_fusion` par correction de `divise`, `fusionne`, avec une récurrence forte immédiate.

Exercice 4 – Problème de l'arrêt (spoiler de la MPI)

On définit le problème de l'arrêt comme suis :

Problème de l'arrêt

Entrée : un programme P exprimé comme un mot sur un alphabet Σ, qui prend en entrée un mot sur Σ, et qui renvoie `vrai` ou `faux`,
un mot u sur l'alphabet Σ

Sortie : `vrai` si P termine sur l'entrée u, `faux` sinon

▶ **Question 1** Montrer qu'il existe un algorithme partiellement correct pour cette spécification (on suppose que l'on peut exécuter P à partir de sa représentation comme un mot sur Σ).

Il suffit d'utiliser le programme suivant, exprimé en pseudo-code :

Pseudo-code

```
1 : fonction SimuleProgramme(P, u) :
2 :     Lancer l'exécution de P sur l'entrée u
3 :     retourne vrai
4 : Fin fonction
```

Si ce programme termine, alors la ligne 2 termine et la ligne 3 renvoie `vrai`, c'est-à-dire le programme P termine sur l'entrée u. Ainsi, pour toute entrée admissible sur laquelle le programme termine, la sortie de SimuleProgramme est correcte, donc la fonction est partiellement correcte.

▶ **Question 2** Montrer qu'il n'existe pas d'algorithme correct décidant le problème de l'arrêt. *On supposera qu'un algorithme en pseudo-code est exprimable sur l'alphabet Σ.*

Supposons qu'un tel algorithme, nommé Arrêt, existe. Il prend en entrée un programme P et un mot u et, en temps fini, dit si le programme P termine sur l'entrée u. Alors, soit l'algorithme suivant :

Pseudo-code

```
1 : fonction PARADOXE(x) :
2 :     Si ARRÊT(P, P) renvoie vrai alors        ▷ le deuxième argument est P, ce qui est possible puisque P est un mot sur Σ
3 :         Tant que vrai, faire :
4 :             rien                              ▷ boucle infinie
5 :         Fin tant que
6 :     Sinon
7 :         retourne vrai
8 :     Fin si
9 : Fin fonction
```

Cet algorithme est exprimable sur l'alphabet Σ, et regardons l'appel à Paradoxe(Paradoxe), c'est-à-dire l'exécution de l'algorithme sur son propre code exprimé dans Σ.

- Si cet appel termine, alors il le fait forcément grâce à la ligne 7, c'est-à-dire Paradoxe termine sur l'entrée Paradoxe. Ainsi, Arrêt(Paradoxe,Paradoxe) renvoie `faux`.
- S'il ne termine pas, alors puisque Arrêt termine sur toute entrée, le programme boucle forcément infiniment dans la boucle ligne 3. Ainsi, Arrêt(Paradoxe,Paradoxe) renvoie `vrai`.

C'est-à-dire, l'appel Paradoxe(Paradoxe) termine ssi Arrêt(Paradoxe,Paradoxe) renvoie `faux` ssi l'appel Paradoxe(Paradoxe) ne termine pas, ce qui est une contradiction.

Exercice 5 – Le principe d'induction bien fondée

La relation binaire $\leq$ sur un ensemble X est une relation d'ordre si elle est réflexive, antisymétrique et transitive.
On dit que la relation d'ordre $\leq$ sur X est *bien fondée* si tout sous-ensemble non vide Y de X possède un élément minimal, c.-à-d. un élément de Y tel qu'il n'existe pas d'élément de Y strictement plus petit pour $\leq$.
Un ordre $\leq$ total et bien fondé est appelé un *bon ordre*, et X un ensemble *bien ordonné* par $\leq$.

▶ **Question 1** Montrer le principe d'induction bien fondée :

Théorème 1 – Principe d'induction bien fondée

Soit $\mathcal{P}$ un prédicat sur X, et $\leq$ une relation d'ordre bien fondée sur X. Supposons que pour tout $x \in X$, si tout élément $y < x$ vérifie $\mathcal{P}$, alors x vérifie $\mathcal{P}$. Alors, $\mathcal{P}(x)$ est `vrai` pour tout $x \in X$.

On considère l'ensemble $Y = \mathcal{P}^{-1}(\texttt{faux})$. Supposons que $Y \neq \emptyset$. Alors il possède un plus petit élément x, par bonne fondation de $\leq$. Ainsi, pour tout élément $y < x$, on a $y \notin Y$ donc $\mathcal{P}(y) = \texttt{vrai}$. Ainsi, par l'hypothèse de l'induction bien fondée, on a $\mathcal{P}(x) = \texttt{vrai}$, ce qui contredit $x \in Y$. Par l'absurde, on a $Y = \emptyset$, et $\mathcal{P}$ vérifiée sur tout l'ensemble X.

▶ **Question 2** Comment s'appelle le principe d'induction bien fondée pour $X = \mathbb{N}$ et $\leq$ l'ordre canonique sur les entiers naturels ?

C'est le principe de *récurrence forte*.

▶ **Question 3** Montrer que $\leq$ sur X est bien fondé si et seulement s'il n'existe pas de suite infinie strictement décroissante d'éléments de X, si et seulement si le principe de récurrence forte est vrai pour $\leq$ sur X pour tout prédicat $\mathcal{P}$ sur X.

On va le montrer en montrant que (1) implique (2), (2) implique (3) et (3) implique (1).
(1) ⇒ (2) On raisonne par l'absurde. Soit $(u_n)_{n\in\mathbb{N}}$ et soit $A = \{u_k \mid k \in \mathbb{N}\}$. A admet un élément minimal par bonne fondation de $\leq$, donc il existe $n \in \mathbb{N}$ tel que u_n est l'élément minimal de A. Notamment, on a $u_n \leq u_{n+1} \in A$, donc la suite $(u_n)_{n\in\mathbb{N}}$ ne peut pas être strictement décroissante.
(2) ⇒ (3) Soit $\mathcal{P}$ un prédicat sur X et supposons que les hypothèses du principe de récurrence forte y est vrai. Soit $Y = \{x \in X \mid \mathcal{P}(x) = \texttt{faux}\}$, et soit $x_0 \in A$. Comme $\mathcal{P}(x_0) = \texttt{faux}$, par contraposée des hypothèses du principe de récurrence forte, il existe $x_1 \in Y$ tel que $x_1 < x_0$ et $\mathcal{P}(x_1) = \texttt{faux}$. En itérant ce raisonnement, il existe une suite $(x_k)_{k\in\mathbb{N}}$ strictement décroissante d'élément de Y, ce qui est absurde.
(3) ⇒ (1) Soit $Y \subseteq X$ non vide, et :

$$\mathcal{P} : x \in X \mapsto \begin{cases} \texttt{faux} & \text{si } x \in Y \\ \texttt{vrai} & \text{sinon} \end{cases}$$

Par la contraposée du principe d'induction, il existe $x \in X$ tel que $\mathcal{P}(x) = \texttt{faux}$ et pour tout $y \in X$, si $y < x$ alors $\mathcal{P}(y)$. Ainsi, $x \in Y$ par définition de $\mathcal{P}$. Donc pour tout élément $y \in Y$, on a $y \geq x : x$ est un élément minimal de Y.

▶ **Question 4** Montrer que la relation d'ordre naturelle sur $\mathbb{N}$ est un ordre bien fondé, mais pas sur $\mathbb{Z}$.

Pour $\mathbb{Z}$, le sous-ensemble $\mathbb{Z} \subseteq \mathbb{Z}$ n'a pas de plus petit élément, donc $\mathbb{Z}$ n'est pas bien ordonné. Pour $\mathbb{N}$, soit (u_n) une suite strictement décroissante sur $\mathbb{N}$. Montrons par récurrence sur n que $u_n \leq u_0 - n$.
Initialisation : C'est évident pour $n = 0 : u_0 \leq u_0$
Hérédité : Supposons que $u_n \leq u_0 - n$. Alors $u_{n+1} < u_n$ par décroissance stricte de la suite, donc $u_{n+1} < u_0 - n$, donc $u_{n+1} \leq u_0 - n - 1 = u_0 - (n+1)$.
Notamment, si la suite était infinie, on aurait $u_{u_0+1} \leq -1$ ce qui contredit sa définition comme suite d'entiers naturels. Ainsi, la suite est finie.

▶ **Question 5** Montrer que si $\leq_X$, resp. $\leq_Y$ sont deux ordres bien fondés sur l'ensemble X, resp. Y, alors l'ordre lexicographique $\leq_{X\times Y}$ sur l'ensemble $X \times Y$ est bien fondé. *Cet exemple est généralisable à tout produit d'ensembles bien ordonnés.*

Soit $Z \subseteq X \times Y$, et notons $X_Z = \{x \in X \mid \text{il existe } y \in Y, (x,y) \in Z\}$ (la projection de Z sur la première coordonnée). Alors X_Z possède un élément minimal par bonne fondation de $\leq_X$, noté x_0. Définissons de la même manière $Y_Z = \{y \in Y \mid (x_0, y) \in Z\}$. Cet ensemble est non vide vu la définition de x_0, il possède donc un plus petit élément y_0 par bonne fondation de Y. Alors, (x_0, y_0) est un élément minimal de $X \times Y$ pour l'ordre $\leq_{X\times Y}$.

▶ **Question 6** Montrer que l'ordre lexicographique sur les suites (infinies) d'entiers naturels n'est pas bien fondée.

On prend les suites de la forme $(0, \dots 1, 0 \dots)$: la suite des suites où le $n$ième élément est 1 est bien strictement décroissante pour l'ordre lexicographique et infinie.

Exercice 6 – La fonction d'Ackermann est bien définie

▶ **Question 1** Rappeler la définition de l'ordre lexicographique sur l'ensemble $\mathbb{N}^2$. *Dans la suite, on le note $\leq_{lex}$.*

$(n,m) \leq_{lex} (n',m')$ ssi $n < n'$, ou $n = n'$ et $m \leq m'$

▶ **Question 2** Montrer par induction bien fondée que l'algorithme suivant *termine*, c'est-à-dire que la séquence d'instructions exécutées dans l'algorithme sur toute entrée est finie (pas de "boucles infinies") :

Pseudo-code

```
1 : fonction ACKERMANN(m,n) :
2 :     Si m = 0 alors
3 :         retourne n + 1
4 :     Sinon, si n = 0
5 :         retourne ACKERMANN(m-1,1)
6 :     Sinon
7 :         k ← ACKERMANN(m,n-1)
8 :         retourne ACKERMANN(m-1, k)
9 :     Fin si
10 : Fin fonction
```

Indication : on peut utiliser dans cette question que $\leq_{lex}$ est bien fondé.

Montrons que l'hypothèse du principe d'induction est bien vérifiée par la propriété $\mathcal{P}_{m,n}$: "l'algorithme termine sur l'entrée (m, n)".
Supposons que pour tout couple $(m', n') <_{lex} (n, m)$, on a $\mathcal{P}_{m,n}$.

- Si $m = 0$, alors l'algorithme termine trivialement en ligne 4 en passant le premier test.
- Si $m \neq 0$ et $n = 0$, alors le premier test échoue, mais le second est passé, ce qui nous fait arriver en ligne 6. Or, puisque $(m-1, 1) <_{lex} (m, 0) = (m, n)$, on a $\mathcal{P}_{m-1,1}$ qui est vraie par hypothèse d'induction : ainsi, l'appel récursif à l'algorithme sur l'entrée $(m-1, 1)$ termine, donc l'algorithme termine sur l'entrée (m, n) aussi.
- Sinon, l'algorithme ne passe pas les deux tests et arrive en ligne 8. On fait donc deux appels récursifs à l'algorithme. D'abord, le premier appel se fait sur l'entrée $(m, n-1) <_{lex} (m, n)$ donc il termine par hypothèse d'induction, et le second appel se fait sur l'entrée $(m-1, k) <_{lex} (m, n)$ qui termine aussi (même si k peut être très gros !).

Par le principe d'induction bien fondée, l'algorithme termine sur toute entrée (admissible).

▶ **Question 3** Écrire explicitement la sortie de ACKERMANN$(1, k)$, ACKERMANN$(2, k)$ et ACKERMANN$(3, k)$ pour tout $k \in \mathbb{N}$.

D'abord, on remarque que pour tout $k \in \mathbb{N}$, ACKERMANN$(0, k) = k + 1$.
Ainsi, on a :

$$\begin{aligned}
\text{ACKERMANN}(1, 0) &= \text{ACKERMANN}(0, 1) = 2 \\
\text{ACKERMANN}(1, n) &= \text{ACKERMANN}(0, \text{ACKERMANN}(1, n-1)) \quad \text{pour } n > 0 \\
&= 1 + \text{ACKERMANN}(1, n-1)
\end{aligned}$$

Donc ACKERMANN$(1, n) = 2 + n$. De même, on a :

$$\begin{aligned}
\text{ACKERMANN}(2, 0) &= \text{ACKERMANN}(1, 1) = 3 \\
\text{ACKERMANN}(2, n) &= \text{ACKERMANN}(1, \text{ACKERMANN}(2, n-1)) \quad \text{pour } n > 0 \\
&= 2 + \text{ACKERMANN}(2, n-1)
\end{aligned}$$

Donc ACKERMANN$(1, n) = 3 + 2n$. De même, on a :

$$\begin{aligned}
\text{ACKERMANN}(3, 0) &= \text{ACKERMANN}(2, 1) = 5 \\
\text{ACKERMANN}(3, n) &= \text{ACKERMANN}(2, \text{ACKERMANN}(3, n-1)) \quad \text{pour } n > 0 \\
&= 3 + 2\text{ACKERMANN}(3, n-1)
\end{aligned}$$

Donc ACKERMANN$(3, n) = -3 + 8 \times 2^n$.

La fonction d'Ackermann calculée par cet algorithme est l'exemple classique de fonction *non primitive récursive* : il n'existe pas d'algorithme (n'utilisant pas d'appels récursifs) calculant cette fonction sans utiliser de boucles non bornées, c'est-à-dire sans utiliser de **while**. Le montrer est cependant un exercice difficile.