# Chapitre 6

# Correction des exercices

**Exercice 1**

– S'il existe une suite $(u_n)_{n\in\mathbb{N}}$ strictement décroissante, posons $A = \{u_n \mid n \in \mathbb{N}\}$. Alors A est non vide et ne peut posséder d'élément minimal $u_n$ car on a pour tout $n \in \mathbb{N}$, $u_{n+1} < u_n$. E n'est donc pas bien fondé.

– Réciproquement, si E n'est pas bien fondé, il existe une partie A sans élément minimal. Construisons à partir de A une suite strictement décroissante :
  – on commence par choisir $u_0 \in A$ ;
  – supposons choisis $u_n < u_{n-1} < \cdots < u_0$ dans A. Puisque $u_n$ n'est pas un élément minimal de A, il existe $x \in A$ tel que $x < u_n$. On pose alors $u_{n+1} = x$.

  On obtient ainsi une suite $(u_n)_{n\in\mathbb{N}}$ strictement décroissante.

**Exercice 2** Il s'agit de montrer que les relations :

$$f(n) = \begin{cases} n-10 & \text{si } n \geq 101 \\ f\big(f(n+11)\big) & \text{sinon} \end{cases}$$

définissent effectivement une fonction sur $\mathbb{N}$.

– Ceci est clair pour $n \geq 101$.

– Si $n \in [\![90,100]\!]$, $f(n+11) = n+1$ car $n+11 \geq 101$, donc $f(n) = f(n+1)$. On en déduit que $f(n) = f(101) = 91$, et $f$ est donc bien définie et constante égale à 91 sur $[\![90,100]\!]$.

– Si $n \in [\![0,89]\!]$, il existe $p \in \mathbb{N}^*$ tel que $90 \leq n+11p \leq 100$. Alors $f(n) = \underbrace{f \circ \cdots \circ f}_{p \text{ fois}}(n+11p) = \underbrace{f \circ \cdots \circ f}_{p-1 \text{ fois}}(91) = 91$.

  $f$ est donc définie sur $[\![0,89]\!]$ et constante égale à 91 sur cet intervalle.

**Remarque**. Cette fonction est connue sous le nom de fonction 91 de McCarthy.

**Exercice 3** Pour justifier la terminaison de cette fonction, on munit $\mathbb{N}^2$ de l'ordre lexicographique, et on prouve par induction que pour tout couple $(n,p) \in \mathbb{N}^2$, le calcul de $A(n,p)$ se termine.
Posons $\mathscr{A} = \{(0,p) \mid p \in \mathbb{N}\}$. Pour tout $(n,p) \in \mathscr{A}$, le calcul de $A(n,p)$ se termine.
Si $(n,p) \notin \mathscr{A}$, supposons le résultat acquis pour tout couple $(n',p') < (n,p)$.

– Si $p = 0$, alors $A(n,0) = A(n-1,1)$ et $(n-1,1) < (n,0)$ donc par hypothèse le calcul de $A(n-1,1)$ se termine.

– Si $p > 0$, alors $(n,p-1) < (n,p)$ donc le calcul de $A(n,p-1)$ se termine, et $\big(n-1,A(n,p-1)\big) < (n,p)$ donc le calcul de $A\big(n-1,A(n,p-1)\big)$ se termine aussi, ce qui achève la démonstration.

Passons maintenant au calcul de $A(n,p)$. Nous savons déjà que pour tout $p \in \mathbb{N}$, $A(0,p) = p+1$.

– Pour tout $p \geq 1$, $A(1,p) = A\big(0,A(1,p-1)\big) = A(1,p-1)+1$. La croissance est arithmétique, donc :

$$A(1,p) = A(1,0) + p = p+2.$$

– Pour tout $p \geq 1$, $A(2,p) = A\big(1,A(2,p-1)\big) = A(2,p-1)+2$. La croissance est arithmétique, donc :

$$A(2,p) = A(2,0) + 2p = 2p+3.$$

– Pour tout $p \geq 1$, $A(3,p) = A\big(2,A(3,p-1)\big) = 2A(3,p-1)+3$. La croissance est arithmético-géométrique donc :

$$A(3,p) = 2^p\big(A(3,0)+3\big) - 3 = 2^{p+3} - 3.$$

– Pour tout $p \geq 1$, $A(4,p) = A\big(3,A(4,p-1)\big) = 2^{A(4,p-1)+3} - 3$. La suite $u_p = A(4,p)+3$ vérifie la relation : $u_p = 2^{u_{p-1}}$ donc $u_p = 2^{2^{\cdot^{\cdot^{2^{u_0}}}}}$ (le chiffre 2 apparaît $p$ fois) avec $u_0 = 16$ et : $A(4,p) = 2^{2^{\cdot^{\cdot^{2}}}} - 3$ (où 2 apparaît $p+3$ fois).

**Remarque**. On peut observer une certaine régularité dans ces formules :

$$A(1,p) = 2+(p+3)-3, \quad A(2,p) = 2\times(p+3)-3, \quad A(3,p) = 2\uparrow(p+3)-3$$

(en notant $\uparrow$ l'opérateur d'élévation à la puissance). Pour poursuivre, on utilise la notation des puissances itérées de Knuth, qui se définit ainsi : $a\uparrow\uparrow b = \underbrace{a\uparrow a\uparrow\cdots\uparrow a}_{b \text{ exemplaires de } a}$. On a alors $A(4,p) = 2\uparrow\uparrow(p+3)-3$.

Poursuivons en notant $a\uparrow\uparrow\uparrow b = \underbrace{a\uparrow\uparrow a\uparrow\uparrow\cdots\uparrow\uparrow a}_{b \text{ exemplaires de } a}$ et plus généralement : $a\uparrow^n b = \begin{cases} 1 & \text{si } b=0 \\ a^b & \text{si } n=1 \\ a\uparrow^{n-1}(a\uparrow^n(b-1)) & \text{sinon} \end{cases}$

Il n'est alors pas difficile de prouver que $A(n,p) = 2\uparrow^{n-2}(p+3)-3$ pour $n\geq 3$.

**Exercice 4** Notons $L = (c_1,\ldots,c_p)$ la liste des pièces que l'on peut utiliser, et $Q = L\setminus\{c_1\}$. Le nombre de décompositions de $n$ utilisant la pièce $c_1$ est égal :

- au nombre de décompositions de $(n-c_1)$ à l'aide des pièces de L si $n > c_1$ ;
- à 1 si $n = c_1$ ;
- à 0 sinon.

Les autres décompositions n'utilisent que les pièces de Q. On en déduit la fonction suivante :

```
let rec compte n c = match c with
  | []              -> 0
  | t::q when n > t -> compte (n-t) c + compte n q
  | t::q when n = t -> 1 + compte n q
  | _::q            -> compte n q ;;
```

Pour afficher les différentes décompositions, il suffit d'adapter la fonction précédente : chaque pièce utilisée est stockée dans une liste, et cette liste est affichée lorsque la somme est atteinte.

```
let rec affiche = function
  | []   -> print_newline ()
  | t::q -> print_int t ; print_char ' ' ; affiche q ;;

let décompose =
  let rec aux l n c = match c with
    | []              -> ()
    | t::q when n > t -> aux (t::l) (n-t) c ; aux l n q
    | t::q when n = t -> affiche (t::l) ; aux l n q
    | _::q            -> aux l n q
  in aux [] ;;
```

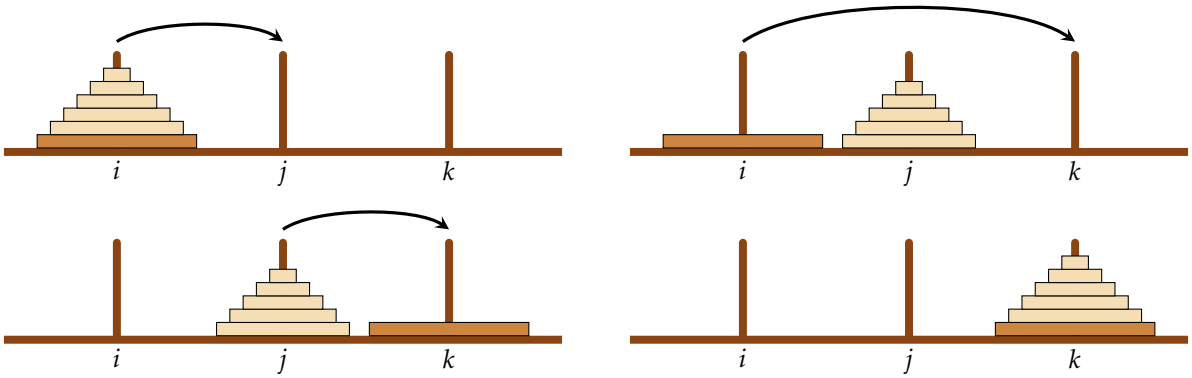
Par exemple :

```
# décompose 11 [1;2;5;10;20] ;;
1 1 1 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1 1
2 2 1 1 1 1 1 1 1
5 1 1 1 1 1 1
2 2 2 1 1 1 1 1
5 2 1 1 1 1
2 2 2 2 1 1 1
5 2 2 1 1
2 2 2 2 2 1
5 5 1
10 1
5 2 2 2
- : unit = ()
```

**Exercice 5**

*a)* Pour déplacer $n$ disques de la tige $i$ vers la tige $k$ on procède ainsi :

- on déplace $n-1$ disques de la tige $i$ vers la tige $j$ ;
- on déplace le dernier disque de la tige $i$ vers la tige $k$ ;
- on déplace les $n-1$ disques de la tige $j$ vers la tige $k$.




Ceci conduit naturellement à la définition suivante :

```
let déplace i j = print_string "déplacer un disque de " ;
                  print_int i ; print_string " vers " ; print_int j ;
                  print_newline () ;;

let hanoi =
  let rec aux i k = function
    | 0 -> ()
    | n -> let j = 6-i-k in
             aux i j (n-1) ;
             déplace i k ;
             aux j k (n-1)
  in aux 1 3 ;;
```

Si on note $d_n$ le nombre de disques déplacés, nous avons $d_0 = 0$ et $d_n = 2d_{n-1} + 1$, donc $d_n = 2^n - 1$.

*b)* Notons $\widetilde{d}_n$ le nombre minimal de déplacements nécessaires pour résoudre le problème à $n$ disques.

Pour pouvoir déplacer le plus gros disque, il est nécessaire que les $n-1$ autres disques soient tous enfilés sur un même autre tige, ce qui par définition nécessite au moins $\widetilde{d}_{n-1}$ déplacements.

Ce gros disque est au moins déplacé une fois lors de la résolution.

Enfin, lors du dernier déplacement du plus gros des disques, tous les autres doivent être enfilés sur une même tige. Il faudra donc encore au minimum $\widetilde{d}_{n-1}$ déplacements pour aboutir à la position finale.

Tout ceci montre que $\widetilde{d}_n \geq 2\widetilde{d}_{n-1} + 1$. Sachant que $\widetilde{d}_0 = 0$, on obtient : $\widetilde{d}_n \geq 2^n - 1$. Ceci montre que la solution proposée plus haut est optimale, et ainsi $\widetilde{d}_n = 2^n - 1$.

*c)* Pour résoudre le problème en s'interdisant tout mouvement entre les tiges 1 et 3, on procède ainsi :

- on déplace $n-1$ disques de la tige 1 vers la tige 3 en respectant la contrainte ;
- on déplace le dernier disque de la tige 1 vers la tige 2 ;
- on déplace $n-1$ disques de la tige 3 vers la tige 1 en respectant la contrainte ;
- on déplace le dernier disque de la tige 2 vers la tige 1 ;
- on déplace $n-1$ disques de la tige 1 vers la tige 3 en respectant la contrainte.

Ceci conduit à la fonction suivante :

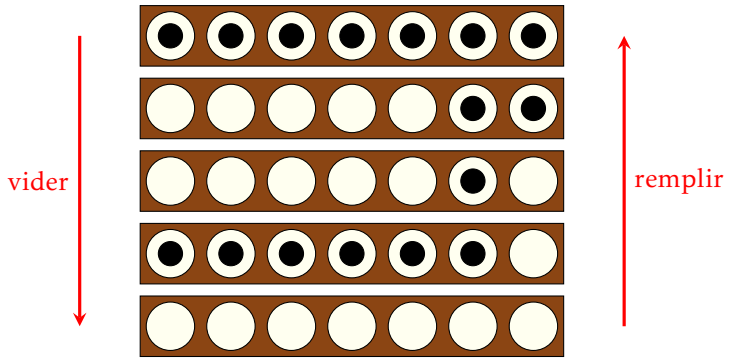
```
let hanoi_bis =
  let rec aux i j = function
    | 0 -> ()
    | n -> aux i j (n-1) ;
           déplace i 2 ;
           aux j i (n-1) ;
           déplace 2 j ;
           aux i j (n-1)
  in aux 1 3 ;;
```

Le nombre $d_n$ de déplacements effectués vérifie cette fois les relations $d_0 = 0$ et $d_n = 3d_{n-1} + 2$, ce qui conduit à : $d_n = 3^n - 1$.

**Exercice 6**

*a)* Si $n \geq 2$, on vide les $n$ premières cases initialement pleines de la manière suivante :

- vider les $n-2$ premières cases ;
- enlever le pion de la case $n$ ;
- remplir les $n-2$ premières cases ;
- vider les $n-1$ premières cases.




De même, on remplit les $n$ premières cases initialement vides de la manière suivante :

- remplir les $n-1$ premières cases ;
- vider les $n-2$ premières cases ;
- poser un pion dans la case $n$ ;
- remplir les $n-2$ premières cases.

Ceci nous amène à écrire deux procédures mutuellement récursives :

```
let enlever n = print_string "enlever le pion " ; print_int n ;
                print_newline () ;;

let poser n = print_string "poser le pion " ; print_int n ;
              print_newline () ;;

let rec vider = function
  | 0 -> ()
  | 1 -> enlever 1
  | n -> vider (n-2) ; enlever n ; remplir (n-2) ; vider (n-1)
and remplir = function
  | 0 -> ()
  | 1 -> poser 1
  | n -> remplir (n-1) ; vider (n-2) ; poser n ; remplir (n-2) ;;
```

Illustrons ceci par un exemple :

```
# vider 4 ;;
enlever le pion 2
enlever le pion 1
enlever le pion 4
poser le pion 1
poser le pion 2
enlever le pion 1
enlever le pion 3
poser le pion 1
enlever le pion 2
enlever le pion 1
- : unit = ()
```

*b)* Notons $t_n$ le nombre de mouvements nécessaires pour vider les $n$ premières cases, et $u_n$ celui nécessaire pour les remplir. On dispose des relations :

$$\begin{cases} t_0 = 0,\ t_1 = 1 \\ \forall n \geq 2,\ t_n = t_{n-2} + 1 + u_{n-2} + t_{n-1} \end{cases} \quad \text{et} \quad \begin{cases} u_0 = 0,\ u_1 = 1 \\ \forall n \geq 2,\ u_n = u_{n-1} + t_{n-2} + 1 + u_{n-2} \end{cases}$$

On démontre aisément par récurrence que : $\forall n \in \mathbb{N},\ u_n = t_n$, et donc :

$$\begin{cases} t_0 = 0,\ t_1 = 1 \\ \forall n \geq 2,\ t_n = t_{n-1} + 2t_{n-2} + 1 \end{cases}$$

La résolution de l'équation $\ell = \ell + 2\ell + 1$ donne $\ell = -\frac{1}{2}$, ce qui suggère de poser $t'_n = t_n + \frac{1}{2}$. Alors :

$$t'_0 = \frac{1}{2},\ t'_1 = \frac{3}{2},\ \text{et } \forall n \geq 2,\ t'_n = t'_{n-1} + 2t'_{n-2}.$$

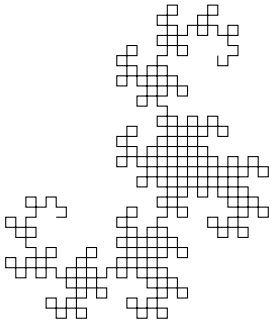
La résolution de l'équation caractéristique $\lambda^2 = \lambda + 2$ donne deux solutions $\lambda = -1$ et $\lambda = 2$. Il existe donc $(a,b) \in \mathbb{R}^2$ tel que : $\forall n \in \mathbb{N}\ t'_n = a(-1)^n + b2^n$. Les valeurs obtenues pour $n = 0$ et $n = 1$ donnent $a = -\frac{1}{6}$ et $b = \frac{2}{3}$, puis : $\forall n \in \mathbb{N}$, $t_n = u_n = \dfrac{2^{n+2} + (-1)^{n+1} - 3}{6}$.

**Exercice 7** Notons que si $(x,y)$ et $(z,t)$ sont les coordonnées respectives de P et Q, les coordonnées de R sont : $\left(\frac{x+z}{2} + \frac{t-y}{2}, \frac{y+t}{2} + \frac{x-z}{2}\right)$. Nos points seront représentés par des couples de type *int * int* ; pour éviter les erreurs de calculs liées aux divisions entières par 2, on prendra soin de faire en sorte que $z - x$ et $y - t$ soient des puissances de 2.
La première version de la fonction s'écrit alors :

```
let rec dragon p q = function
  | 0 -> moveto (fst p) (snd p) ; lineto (fst q) (snd q)
  | n -> let r = ((fst p + fst q - snd p + snd q)/2,
                  (fst p - fst q + snd p + snd q)/2)
         in dragon p r (n-1) ; dragon q r (n-1) ;;
```

Par exemple :

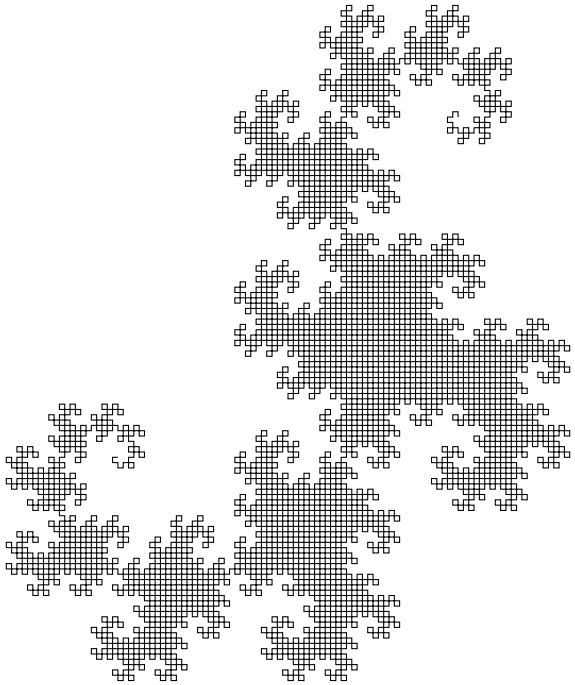
```
# dragon (100,100) (228,228) 9 ;;
- : unit = ()
```



Pour éviter de lever le crayon, il est nécessaire de définir une deuxième courbe, qu'on appellera la courbe du nogard, symétrique de la courbe du dragon par rapport à la droite (PQ). Ces deux courbes sont alors mutuellement récursives :

```
let rec dragon p q = function
  | 0 -> lineto (fst q) (snd q)
  | n -> let r = ((fst p + fst q - snd p + snd q)/2,
                  (fst p - fst q + snd p + snd q)/2)
         in dragon p r (n-1) ; nogard r q (n-1)
and nogard p q = function
  | 0 -> lineto (fst q) (snd q)
  | n -> let r = ((fst p + fst q + snd p - snd q)/2,
                  (fst q - fst p + snd p + snd q)/2)
         in dragon p r (n-1) ; nogard r q (n-1) ;;
```

```
# clear_graph () ; moveto 100 100 ; dragon (100,100) (228,228) 13 ;;
- : unit = ()
```



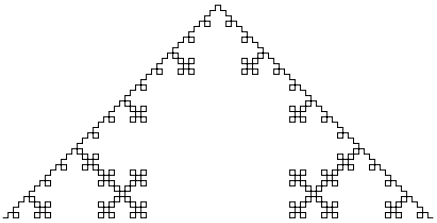
**Exercice 8** Pour décrire les constructions récursives de cet exercice, nous allons employer le vocabulaire des L-system (inventé par le biologiste Lindenmayer pour modéliser le développement de certaines plantes), qui utilise la notion de grammaire formelle. Nous allons utiliser trois caractères : + qui désigne la rotation à gauche, – la rotation à droite, et F qui désigne le déplacement d'un pas unitaire.

Ainsi, la courbe quadratique de von Koch se définit par la donnée d'un axiome F et d'une règle F → F+F–F–F+F. Ceci conduit à la définition suivante :

```
let rec koch p = function
  | 0 -> avance p
  | n -> koch p (n-1) ; gauche () ; koch p (n-1) ;
         droite () ; koch p (n-1) ; droite () ;
         koch p (n-1) ; gauche () ; koch p (n-1) ;;
```

En voici un exemple d'utilisation :

```
# let trace (x,y) p n =
    clear_graph () ;
    tortue.X <- x ; tortue.Y <- y ; moveto x y ;
    tortue.Dir <- E ;
    koch p n ;;
trace : int * int -> int -> int -> unit = <fun>
# trace (50,50) 2.5 4 ;;
- : unit = ()
```



La courbe de Hilbert est un peu plus complexe car il faut définir de manière conjointe la courbe de Hilbert et la courbe inverse. Dans le vocabulaire des L-system, ceci se traduit par la donnée de deux variables A (représentant la courbe de Hilbert) et B (son inverse), les deux axiomes A = ε, B = ε et les deux règles : A → −BF + AFA + FB− et B → +AF − BFB − FA+.
Ceci se traduit par la définition :

```
let rec hilbertA p = function
  | 0 -> ()
  | n -> gauche () ; hilbertB p (n-1) ; avance p ;
         droite () ; hilbertA p (n-1) ; avance p ;
         hilbertA p (n-1) ; droite () ; avance p ;
         hilbertB p (n-1) ; gauche ()
and hilbertB p = function
  | 0 -> ()
  | n -> droite () ; hilbertA p (n-1) ; avance p ;
         gauche () ; hilbertB p (n-1) ; avance p ;
         hilbertB p (n-1) ; gauche () ; avance p ;
         hilbertA p (n-1) ; droite () ;;
```

Illustration :

```
# let trace (x,y) p n =
    clear_graph () ;
    tortue.X <- x ; tortue.Y <- y ; moveto x y ;
    tortue.Dir <- E ;
    hilbertA p n ;;
trace : int * int -> int -> int -> unit = <fun>
# trace (10,10) 5 6 ;;
- : unit = ()
```