Devoir surveillé n° 1 — corrigé

I Un peu de typage

I.1) Les fonctions `f05` et `f08` d'une part et `f07` et `f10` sont identiques, seule la syntaxe utilisée pour les définir diffère.

OCaml
```
f01 : 'a -> 'b -> 'a
f02 : 'a -> 'b -> 'b
f03 : ('a -> 'b) -> 'a -> 'b
f04 : 'a -> ('a -> 'b) -> 'b
f05 : ('a -> 'a -> 'b) -> 'a -> 'b
f06 : ('a -> 'a) -> 'a -> 'a
f07 : (('a -> 'b) -> 'a) -> ('a -> 'b) -> 'b
f08 : ('a -> 'a -> 'b) -> 'a -> 'b
f09 : ('a -> 'b) -> (('a -> 'b) -> 'a) -> 'b
f10 : (('a -> 'b) -> 'a) -> ('a -> 'b) -> 'b
```

II La fonction d'Ackermann

II.1) Le premier cas donne directement $A(0,0) = 1$. Pour $p \geq 1$, on remarque que $A(1,p) = A(0,A(1,p-1)) = A(1,p-1)+1$ et donc de proche en proche $A(1,p) = p + A(1,0) = p+2$. On a ensuite :

$$\begin{aligned} A(2,2) &= A(1,A(2,1)) \\ &= A(2,1)+2 \\ &= A(1,A(2,0))+2 \\ &= A(2,0)+4 \\ &= A(1,1)+4 \\ &= 3+4 \\ &= 7 \end{aligned}$$

II.2) On traduit directement la définition en CAML :

OCaml
```
let rec ackermann n p =
  if n = 0 then
    p + 1
  else if p = 0 then
    ackermann (n - 1) 1
  else
    ackermann (n - 1) (ackermann n (p - 1))
```

II.3) De la même manière :

OCaml
```
let rec ackermann n p =
  match n, p with
  | 0, p -> p + 1
  | n, 0 -> ackermann (n - 1) 1
  | n, p -> ackermann (n - 1) (ackermann n (p - 1))
```

III Représentation d'ensembles avec des listes

III.1) Par la liste vide bien sûr.

III.2) Il suffit de tester si l'ensemble est la liste vide. Cette fonction s'exécute en temps constant.

OCaml
```
let est_vide ens = ens = []
```

III.3) C'est la fonction `List.length` de complexité linéaire en la taille de la liste.

OCaml
```
let rec cardinal ens =
  match ens with
  | [] -> 0
  | _ :: suite -> 1 + cardinal suite
```

III.4) C'est la fonction `List.mem` que l'on écrit élégamment en utilisant le caractère paresseux de l'opérateur booléen. Sa complexité dans le pire cas est linéaire en la taille de la liste.

OCaml
```
let rec appartient elt ens =
  match ens with
  | [] -> false
  | obj :: suite -> (obj = elt) || appartient elt suite
```

III.5) Il faut bien veiller à ne pas ajouter de doublons. La complexité est linéaire en la taille de la liste : c'est celle de la fonction précédente plus un temps constant.

OCaml
```
let ajoute elt ens =
  if appartient elt ens then
    ens
  else
    elt :: ens
```

III.6) On pourrait encore utiliser la fonction `appartient`, on aurait alors deux parcours de la liste si l'élément est présent, mais on éviterait de reconstruire inutilement la liste si ce n'est pas le cas. Dans tous les cas la complexité dans le pire cas est linéaire en la taille de la liste. Dans le deuxième cas on sait que l'élément n'est plus présent dans la suite puisque la liste est sans doublons.

OCaml
```
let rec supprime elt ens =
  match ens with
  | [] -> []
  | obj :: suite when obj = elt -> suite
  | obj :: suite -> obj :: (supprime elt suite)
```

III.7) La fonction `intersection : ensemble -> ensemble -> ensemble`, ajoute successivement tous les éléments du premier ensemble qui appartiennent aussi au deuxième. Il y a exactement $|ens_1|$ appels récursifs, chacun de complexité $\Theta(|ens_2|)$. La complexité est donc en $\Theta(|ens_1||ens_2|)$.

OCaml
```
let rec intersection ens1 ens2 =
  match ens1 with
  | [] -> []
  | elt1 :: suite1 when appartient elt1 ens2 ->
      elt1 :: (intersection suite1 ens2)
  | _ :: suite1 ->
      intersection suite1 ens2
```

III.8) La fonction `union : ensemble -> ensemble -> ensemble` ajoute au deuxième ensemble tous les éléments du premier qui ne sont pas dans le deuxième. Sa complexité est aussi en $\Theta(|ens_1||ens_2|)$.

OCaml
```
let rec union ens1 ens2 =
  match ens1 with
  | [] -> ens2
  | elt1 :: suite1 when appartient elt1 ens2 ->
      union suite1 ens2
  | elt1 :: suite1 ->
      elt1 :: (union suite1 ens2)
```

III.9) On peut implémenter une fonction `inclusion : ensemble -> ensemble -> bool` et vérifier la double inclusion. On utilise ici directement les fonctions précédentes pour implémenter la fonction `egal : ensemble -> ensemble -> bool`. Dans tous les cas, la complexité est en $\Theta(|ens_1||ens_2|)$.

OCaml
```
let egal ens1 ens2 =
  let c = cardinal (intersection ens1 ens2) in
  c = (cardinal ens1) && c = (cardinal ens2)
```

IV Deux algorithmes de tri

IV.1 Tri par insertion

Cf. TD n° 2.

IV.2 Tri par sélection

L'idée du tri par sélection (ou tri par extraction) est d'extraire le minimum d'une liste, de trier récursivement cette liste puis de placer ce minimum en tête.

IV.4) On commence par implémenter une fonction `minimum : 'a list -> 'a` qui cherche le minimum d'une liste, de complexité linéaire dans tous les cas.

OCaml
```
let rec minimum liste =
  match liste with
  | [] -> failwith "minimum : liste vide"
  | elt :: [] -> elt
  | elt :: queue ->
      let m = minimum queue in
      min elt m
```

Pour supprimer la première occurrence de ce minimum, on peut utiliser la fonction `supprime : 'a -> 'a list -> 'a list` de la question III.6), de complexité linéaire dans le pire cas et constante dans le meilleur cas. La fonction `tri_selection : 'a list -> 'a list` implémente alors l'idée du tri par sélection. La complexité dans le meilleur comme dans le pire cas suit une équation de récurrence de type $C(n) = C(n-1) + \Theta(n)$ vu l'appel à la fonction `minimum` et la complexité dans le meilleur comme dans le pire cas est donc en $\Theta(n^2)$.

OCaml
```
let rec tri_selection liste =
  if liste = [] then
    []
  else
    let m = minimum liste in
    let reste = supprime m liste in
    m :: (tri_selection reste)
```

V Problème

V.1) Montrons par récurrence sur $k \in [\![0, r]\!]$ que $n_k \leq 2^k$.

- Le résultat est vrai initialement car $n_0 = 1 = 2^0$.
- Supposons le résultat vrai jusqu'à un rang $k \in [|0, r-1|]$. Il existe alors $0 \leq i, j \leq k$ tels que $n_{k+1} = n_i + n_j$ et par hypothèse de récurrence, $n_{k+1} \leq 2^i + 2^j \leq 2^k + 2^k = 2^{k+1}$ ce qui prouve le résultat au rang $k+1$.

V.2) À un calcul de a^n qui n'utilise que des multiplications on peut associer une suite pour l'obtention de cette puissance $(n_0, n_1, \ldots, n_r)$ avec $n_r = n \leq 2^r$ et donc $\log_2 n \leq r$ puis $\lceil \log_2 n \rceil \leq r$ car r est un entier. Comme r désigne le nombre de multiplications pour cette suite, on vient donc de montrer que $\lceil \log_2 n \rceil$ est un minorant du nombre de multiplications nécessaires.

V.3) Si $n = 2^k$ pour un entier k, on peut considérer la suite $(1, 2, 4, 8, \ldots, 2^k)$ qui montre que a^n peut être calculé en $k = \log_2(n) = \lceil \log_2(n) \rceil$ multiplications. La suite $(2^k)_{k \in \mathbb{N}}$ convient donc.

V.4) On a les résultats suivants.

(a) Pour a^{15} : $(1, 2, 3, 6, 7, 14, 15)$ de longueur 7

(b) Pour a^{16} : $(1, 2, 4, 8, 16)$ de longueur 5

(c) Pour a^{27} : $(1, 2, 3, 6, 12, 13, 26, 27)$ de longueur 8

(d) Pour a^{125} : $(1, 2, 3, 6, 7, 14, 15, 30, 31, 62, 124, 125)$ de longueur 12

V.5) La traduction en CAML de l'algorithme construit la liste à l'envers. On introduit donc une fonction auxiliaire `par_division_aux` de même type qui implémente directement l'algorithme donné. Dans tous les cas la puissance n à calculer fait partie de la liste, et selon les cas $n-1$ également, puis on calcule récursivement la suite pour $\lfloor \frac{n}{2} \rfloor$. Il reste enfin à inverser l'ordre des éléments.

OCaml
```
let par_division n =
  let rec par_division_aux n =
    if n = 1 then
      [1]
    else if n mod 2 = 0 then
      n :: (par_division_aux (n / 2))
    else
      n :: (n - 1) :: (par_division_aux (n / 2))
  in
  List.rev (par_division_aux n)
```

On peut aussi donner une version avec un accumulateur, qui me semble plus indiquée ici :

OCaml
```
let par_division n =
  let rec par_division_aux n accu =
    if n = 1 then
      1 :: accu
    else if n mod 2 = 1 then
      par_division_aux (n - 1) (n :: accu)
    else
      par_division_aux (n / 2) (n :: accu)
  in
  par_division_aux n []
```

V.6) On peut remarquer qu'il y a exactement $k = \lfloor \log_2 n \rfloor$ appels récursifs et que chaque appel récursif ajoute une ou deux multiplications. Dans le pire des cas il y a donc besoin de $2\lfloor \log_2 n \rfloor$ multiplications.

Montrons ce résultat par récurrence sur $n \in \mathbb{N}^*$ (ce qui est naturel puisque l'énoncé propose un algorithme récursif). L'hypothèse au rang n est que pour tout a, le calcul de a^n utilise au plus $M_n(a) = 2 \times \lfloor \log_2(n) \rfloor$ multiplication.

- Si $n = 1$, l'algorithme n'effectue aucune multiplication et on a bien $2 \times \lfloor \log_2(1) \rfloor = 0$.
- Supposons le résultat vrai jusqu'à un rang $n - 1 \geq 1$. Soit $n \geq 2$ et $n = 2q + r$ la division euclidienne de n par 2. Le calcul de a^n se fait de la façon suivante :
  - on calcule $b = a^q$ ce qui coûte au plus $M_q(a) = 2 \times \lfloor \log_2(q) \rfloor$ multiplications par hypothèse de récurrence ;
  - on calcule $b \times b$ ce qui coûte une multiplication ;
  - on multiplie éventuellement par a.

  On a alors

$$M_n(a) \leq 2\lfloor \log_2(q) \rfloor + 2 \leq 2\lfloor \log_2(q) + 1 \rfloor \leq 2\lfloor \log_2(2q) \rfloor \leq 2\lfloor \log_2(n) \rfloor$$

  puisque $x \mapsto \lfloor \log_2(x) \rfloor$ est croissante. Ceci prouve le résultat au rang n.

V.7) Le cas le pire est celui où l'on tombe toujours sur des exposants impairs. Pour $k \in \mathbb{N}^*$, prenons le cas où $n = 2^k - 1$, qui s'écrit en binaire avec k bits égaux à 1. Notons $C_k(a) = M_{2^k-1}(a)$ le nombre de multiplications pour $n = 2^k - 1$. On a $C_1(a) = 0$ et $C_k(a) = 2 + C_{k-1}(a)$. Ainsi, $C_k(a) = 2(k-1)$ qui est bien égal à $2\lfloor \log_2(2^k - 1) \rfloor = 2\lfloor \log_2(n) \rfloor$.

V.8) On a les résultats suivants :

(a) Pour a^{15} : $(1, 2, 3, 4, 7, 8, 15)$ de longueur 7 ;
(b) Pour a^{16} : $(1, 2, 4, 8, 16)$ de longueur 5 ;
(c) Pour a^{27} : $(1, 2, 3, 4, 8, 11, 16, 27)$ de longueur 8 ;
(d) Pour a^{125} : $(1, 2, 4, 5, 8, 13, 16, 29, 32, 61, 64, 125)$ de longueur 12.

V.9) Si $n = 2q + r$ (division euclidienne), on obtient la décomposition de n en ajoutant le bit r à la décomposition de q.

OCaml
```
let rec binaire_inverse n =
  if n = 0 then []
  else (n mod 2) :: (binaire_inverse (n / 2))
```

V.10) La suite voulue contient toutes les puissances de 2 inférieures à n ainsi que les « sommes partielles » dans la décomposition de n. Pour être efficace, il convient de garder trace de la « puissance courante » de 2 et de la somme partielle calculée jusque-là. On écrit ainsi une fonction auxiliaire `parcours : int list -> int -> int -> int list`. Dans l'appel `parcours bits puiss somme_p`, `bits` est la liste des bits restants, `puiss` la puissance de 2 correspondant au premier bit et `somme_p` est la somme partielle correspondant aux bits déjà consommés. Il faut aussi faire attention ne pas ajouter une somme partielle nulle.

OCaml
```
let par_decomposition_binaire n =
  let rec parcours bits puiss somme_p =
    match bits with
    | [] -> []
    | 0 :: reste ->
      puiss :: (parcours reste (2 * puiss) somme_p)
    | 1 :: reste when somme_p = 0 ->
      puiss :: (parcours reste (2 * puiss) puiss)
    | 1 :: reste ->
      let somme_p = puiss + somme_p in
      let suite = parcours reste (2 * puiss) somme_p in
      puiss :: somme_p :: suite
  in
  parcours (binaire_inverse n) 1 0
```

V.11) La suite $(1, 2, 3, 5, 10, 15)$ convient. Les algorithmes précédents ne sont donc pas optimaux.