DS n° 01 (partie A) — corrigé

I Où l'on renverse les listes

Q 1) Pas de difficulté.

C
```c
cell* new_cell(int val) {
  cell* c = (cell*)malloc(sizeof(cell));
  c->val = val;
  c->next = NULL;
  return c;
}
```

Q 2) On itère sur la liste pour ajouter successivement tous les éléments en tête à une liste initialement vide.

C
```c
cell* reverse_copy(cell* list) {
  cell* res = NULL;
  while (list != NULL) {
    cell* head = new_cell(val);
    head->next = res;
    res = head;
    list = list->next;
  }
  return res;
}
```

Q 3) On traite à part le cas de la liste vide. On maintient un pointeur `head` vers le premier maillon de la liste et un pointeur `last` vers le dernier maillon en construction. On itère sur la liste en ajoutant « à la main » les éléments à la fin via `last`.

C
```c
cell* copy(cell* list) {
  if (list == NULL) {return NULL;}
  cell* head = new_cell(list->val);
  cell* last = head;
  while (list->next != NULL) {
    list = list->next;
    cell* next = new_cell(list->val);
    last->next = next;
    last = next;
  }
  return head;
}
```

Q 4) On traite toujours à part le cas de la liste vide. Les pointeurs `list` et `next` pointent sur deux positions consécutives : avant `list` la liste a été renversée et il reste à traiter ce qui se trouve à partir de `next`. On fait pointer `next` sur `list` (ce qui renverse le chaînage) et on avance d'un cran (en sauvegardant le suivant et en faisant les choses dans le bon ordre).

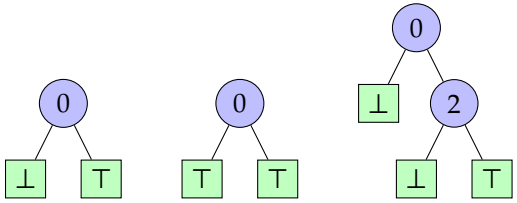
C
```c
cell* reverse(cell* list) {
  if (list == NULL) {return NULL;}
  cell* next = list->next;
  list->next = NULL;
  while (next != NULL) {
    cell* next_next = next->next;
    next->next = list;
    list = next;
    next = next_next;
  }
  return list;
}
```

II Arbres combinatoires

Q 5) On obtient successivement :

- $S(2 \to \bot, \top) = \{\{2\}\}$
- $S(1 \to (2 \to \bot, \top), \top) = \{\{1\}, \{2\}\}$
- $S(1 \to \top, (2 \to \bot, \top)) = \{\varnothing, \{1, 2\}\}$
- $S(0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top))) = \{\{0\}, \{1\}, \{2\}, \{0, 1, 2\}\}$

Q 6) Les contraintes imposent de choisir respectivement les arbres :



Q 7) On peut remarquer que le chemin le plus à droite débouche nécessairement sur une feuille $\top$. Plus formellement, montrons le résultat par récurrence sur $n \in \mathbb{N}$ le nombre de nœuds (internes) d'un arbre.

- **Initialisation** Si $n = 0$, l'arbre est une feuille, différente de $\bot$, et ce ne peut qu'être qu'une feuille $\top$ ce qui démontre la propriété au rang 0.
- **Hérédité** Supposons le résultat acquis au rang $n \in \mathbb{N}$ et considérons un arbre combinatoire à $n+1 \geq 1$ nœuds, donc de la forme $i \to A_1, A_2$ avec A_1 et A_2 deux arbres combinatoires avec moins de n nœuds. Par la propriété (**suppression**), A_2 n'est pas $\bot$ et par hypothèse de récurrence A_2 possède une feuille $\top$ et il en va donc de même de A.

Q 8) On peut suivre la branche la plus à droite qui débouche sur l'ensemble constitué de tous les éléments rencontrés (et qui sont dans le bon ordre). On parcourt une unique branche ce qui garantit une complexité en $\Theta(h(A))$ ou $h(A)$ est la hauteur de l'arbre.

OCaml
```
let rec un_elt a =
  match a with
  | Zero -> failwith "impossible"
  | Un -> []
  | Comb (x, _, fd) -> x :: un_elt fd
```

Q 9) De la même manière, on construit la branche de droite pour aboutir à un arbre peigne où toutes les feuilles gauches sont $\bot$.

OCaml
```
let rec singleton e =
  match e with
  | [] -> Un
  | x :: suite -> Comb (x, Zero, singleton suite)
```

La complexité est linéaire en la taille de la liste `e` qui est majorée par n, d'où une complexité en $O(n)$.

Q 10) On suit les règles. Si $A = \bot$ alors seul $\varnothing$ appartient à $S(A)$. Supposons maintenant que A est de la forme $i \to A_1, A_2$. Si $s = \varnothing$, alors $s \in S(A) \iff s \in S(A_1)$. Sinon, considérons $j = \min s$ le plus petit élément de s, qui est donc la tête de la liste qui représente s, et notons $v = s \setminus \{j\}$ qui est donc représenté par la queue de la liste. Si $j < i$ alors $s \in S(A) \iff s \in S(A_1)$. Si $j = i$ alors $s \in S(A) \iff v \in S(A_2)$. Dans tous les autres cas $s \notin S(A)$.

OCaml
```
let rec appartient e a =
  match e, a with
  | [], Un -> true
  | [], Comb (_, fg, _) -> appartient [] fd
  | j :: suite, Comb (i, _, fd) when x = i ->
      appartient suite fd
  | j :: suite, Comb (i, fg, _) when i < j ->
      appartient e fg
  | _ -> false
```

On ne peut pas directement dire que la complexité est linéaire en la taille de e à cause du cas $\varnothing$ qui impose tout de même un parcours d'une branche. En revanche, on parcourt toujours au plus une branche et la complexité est donc linéaire en la hauteur de l'arbre A qui est majorée par n, d'où une complexité en $O(n)$.

Q 11) On remarque que l'union dans la définition de $S(A)$ est disjointe et on traduit directement les relations qui définissent $S(A)$ pour calculer le cardinal.

OCaml
```
let rec cardinal a =
  match a with
  | Zero -> 0
  | Un -> 1
  | Comb (_, fg, fd) -> cardinal fg + cardinal fd
```

Q 12) Pour $n = 0$, les deux seuls arbres combinatoires sont les feuilles $\bot$ et $\top$ corespondant aux ensembles de parties $\varnothing$ et $\{\varnothing\}$ d'où $u_0 = 2$. Pour $n \geq 1$, il y a u_{n-1} arbres combinatoires sur E qui n'utilisent pas l'entier 0, qui sont les arbres combintatoires sur $\{1, 2, \ldots, n-1\}$. Si un arbre combinatoire comporte un nœud d'étiquette 0, il s'agit obligatoirement de la racine vu la règle (**ordre**) et l'abre est de la forme $0 \to A_1, A_2$ avec A_1 et A_2 des arbres combinatoires sur $\{1, 2, \ldots, n-1\}$. Il y a donc u_{n-1} possibilités pour A_1 et $u_{n-1} - 1$ possibilités pour A_2 car il faut exclure le cas $\bot$ vu la règle (**suppression**). Il y a donc $u_{n-1}(u_{n-1} - 1)$ arbres combinatoires avec une étiquette 0. On a donc $u_n = u_{n-1} + u_{n-1}(u_{n-1} - 1) = u_{n-1}^2$ arbres combinatoires sur E. On en déduit par récurrence immédiate que $\forall n \in \mathbb{N}, u_n = 2^{2^n}$.

Q 13) On vient de montrer que l'ensemble des arbres combinatoires et l'ensemble $\mathcal{P}(\mathcal{P}(E))$ ont le même cardinal. Il suffit donc de montrer, par exemple, que S est surjective. Montons par récurrence forte sur $k \in \mathbb{N}$ que si $X \subseteq \mathcal{P}(E)$ est un ensemble de parties dont le nombre d'éléments de E apparaissant dans au moins ensemble de X est k, c'est-à-dire $|\bigcup_{x \in X} x| = k$, alors il existe un arbre combinatoire A tel que $X = S(A)$. Pour $k = 0$, $X = \varnothing$ ou $X = \{\varnothing\}$ et l'arbre combinatoire $\bot$ ou $\top$ convient. Supposons le résultat acquis pour $k \in \mathbb{N}$ et soit X un ensemble d'ensembles de E utilisant $k+1$ éléments de E. Soit $i = \min \bigcup_{x \in X} x$ le plus petit de ces éléments, soit $X_1 = \{x \in X | i \notin x\}$ et $X_2 = \{x \setminus \{i\} \mid x \in X \wedge i \in x\}$. Comme i n'est pas un élément d'un ensemble de X_1 ni de X_2, le nombre total d'élements de X_1 et de X_2 est inférieur à k et par hypothèse de récurrence on peut trouver A_1 et A_2 deux arbres combinatoires tels que $X_1 = S(A_1)$ et $X_2 = S(A_2)$. On vérifie enfin, vu la construction effectuée, que $X = S(A)$ avec $A = i \to A1, A2$, la propriété (**ordre**) est bien vérifiée et la propriété (**suppression**) également car $X_2 \neq \varnothing$ donc $A_2 \neq \bot$.

III Chaînes d'addition

Q 14) Une simple boucle et le tour est joué. Il faut simplement faire attention à initialiser `res` à `a` pour faire $n-1$ multiplications et non pas n. Le programme est donné à la page suivante.

Q 15) On montre sans difficulté que $n - i$ (sous l'hypothèse $n \in \mathbb{N}^*$) est un variant de boucle qui montre la terminaison de cette fonction.

Q 16) Un invariant utile et complet pour montrer la correction de cette fonction est « $i \leq n$ et res $= a^i$ ».

Q 17) La négation de la condition de boucle et l'invariant précédent montrent qu'à la fin de la fonction $i = n$ et res $= a^i = a^n$ et que le résultat renvoyé est bien celui escompté.

C
```
double expo(double a, int n) {
  double res = a;
  for (int i = 1; i < n; i++) {
    res *= a;
  }
  return res;
}
```

Q 18) 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 $k+1$.

Q 19) 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.

Q 20) 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.

Q 21) 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

Q 22) La traduction directe en OCaml de l'algorithme récursif construit la liste à l'envers. On propose une version avec un accumulateur, qui est plus simple ici. On introduit donc une fonction auxiliaire `par_division_aux : int -> int list -> int list` telle que `par_division_aux n accu` ajoute à l'accumulateur la suite de la puissance n correspondant à l'algorithme proposé. 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 $\lfloor \frac{n}{2} \rfloor$. On utilise ici une petite astuce si n est impaire qui consiste à ajouter n puis relancer la procédure avec $n-1$ qui est pair, ce qui va donc ajouter $n-1$ et calculer récursivement pour $\lfloor \frac{n}{2} \rfloor = \lfloor \frac{n-1}{2} \rfloor$.

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 []
```

Q 23) 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.

Q 24) 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 $2\lfloor \log_2(2^k - 1) \rfloor = 2\lfloor \log_2(n) \rfloor$.

Q 25) 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.

Q 26) 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))
```

Q 27) 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.