DS n° 01 (partie A) — 2h30

LA CALCULATRICE N'EST PAS AUTORISÉE.

On veillera à présenter *très clairement* sa copie, on attachera un soin particulier à la rédaction et on encadrera les réponses. Des points pourront être accordés à la présentation, à la lisibilité et au soin accordé à la rédaction. Il est impératif d'utiliser un brouillon.

Les programmes seront rédigés clairement et proprement, en mettant en valeur l'indentation et en choisissant des noms de variables explicites. Il est conseillé d'utiliser de la **couleur** et les commentaires des programmes doivent dans tous les cas être dans une autre couleur que le programme lui-même. Il est impératif de respecter l'indentation usuelle des fonctions OCAML et C.

L'introduction et l'utilisation de fonctions auxiliaires est autorisée, et même encouragée, d'autant plus si cela améliore la lisibilité et la compréhension. Les fonctions auxiliaires **doivent être précédées de leur type (en OCAML) et commentées**.

Lorsque l'on pose des hypothèses sur les arguments, il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée.

Le sujet est composé de trois parties indépendantes que vous pouvez traiter dans l'ordre de votre choix.

I Où l'on renverse les listes

On propose le modèle de structure suivant pour représenter des listes chaînées dans le langage C. Une cellule contient une donnée `val` et pointe sur une cellule, appelée cellule suivante. Une liste est représentée par un pointeur sur une cellule, de type `cell*`. La liste vide est représentée par le pointeur `NULL` et une liste non vide par un pointeur sur sa première cellule.

C

```c
struct cell {
  int val;
  struct cell* next;
};

typedef struct cell cell;
```

Pour les questions suivantes, il est demandé d'adopter un style purement impératif et **de ne parcourir les listes qu'une seule fois**. **Les fonctions récursives ne sont donc pas autorisées**. Certaines des fonctions demandées peuvent être relativement délicates à concevoir : il est impératif de bien choisir le nom des variables intermédiaires et d'**expliquer très clairement** le fonctionnement de la méthode proposée. Les schémas seront les bienvenus.

Q 1) Écrire une fonction `cell* new_cell(int val)` qui créé une nouvelle cellule, de donnée `val` et pointant sur `NULL`, et qui renvoie un pointeur sur cette cellule.

Q 2) Écrire une fonction `cell* reverse_copy(cell* list)` qui prend en argument une liste et qui renvoie une copie de cette liste, à l'envers. La liste et sa copie seront totalement indépendantes en mémoire.

Q 3) Écrire une fonction `cell* copy(cell* list)` qui effectue la copie d'une liste à l'endroit. *Rappel : on parcourra la liste une seule fois ; il ne s'agit pas d'utiliser la fonction précédente.*

Q 4) Écrire une fonction `cell* reverse(cell* list)` qui renverse une liste « en place », c'est-à-dire sans créer aucune nouvelle cellule, par simple modification des pointeurs. On renverra un pointeur vers la cellule en tête de la nouvelle liste (qui est donc la dernière cellule de l'ancienne liste, si celle-ci n'était pas vide). *Rappel : expliquez clairement votre démarche si vous voulez être corrigé.*

II Arbres combinatoires

Dans l'ensemble de cette partie, on se fixe une constante entière $n \in \mathbb{N}$. On note E l'ensemble $\{0, 1, \dots, n-1\}$. Si $n = 0$ alors $E = \varnothing$.

Dans cette partie, on étudie des arbres combinatoires, une structure de données pour représenter un élément de $\mathcal{P}(\mathcal{P}(E))$, c'est à dire un ensemble de parties de E, donc un ensemble d'ensemble d'entiers de $\{0, 1, \dots, n-1\}$. Un arbre combinatoire est un arbre binaire dont les noeuds sont étiquetés par des éléments de E et les feuilles par $\bot$ et $\top$.

Voici un exemple d'arbre combinatoire :

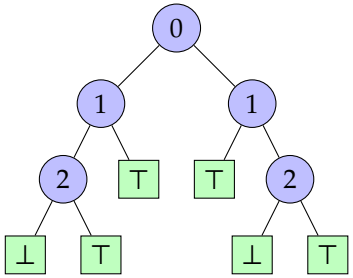


FIGURE 1 – Un arbre combinatoire.

Un noeud étiqueté par i, de sous-arbre gauche A_1 et de sous-arbre droit A_2 sera noté $i \to A_1, A_2$. L'arbre ci-dessus peut donc également s'écrire sous la forme

$$0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top)) \tag{1}$$

On impose de plus les deux propriété suivantes sur tout (sous-)arbre combinatoire de la forme $i \to A_1, A_2$:

- A_1 et A_2 ne contiennent pas d'élément j avec $j \leq i$ (**ordre**)
- $A_2 \neq \bot$ (**suppression**)

Ainsi, les deux arbres ci-dessous ne correspondent pas à des arbres combinatoires, car celui de gauche ne vérifie pas la condition (**ordre**) et celui de droite ne vérifie pas la condition (**suppression**).

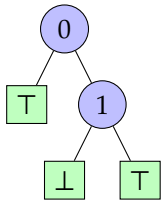
FIGURE 2 – Deux arbres binaires qui ne sont pas des arbres combinatoires.

À tout arbre combinatoire A on associe un ensemble de parties de E, noté $S(A)$, défini par récurrence sur le nombre de nœuds de A par :

$$\begin{cases} S(\bot) = \varnothing \\ S(\top) = \{\varnothing\} \\ S(i \to A_1, A_2) = S(A_1) \cup \{\{i\} \cup s \mid s \in S(A_2)\} \end{cases}$$

L'interprétation d'un arbre A de la forme $i \to A_1, A_2$ est donc la suivante : i est le plus petit élément appartenant à au moins un ensemble de $S(A)$, A_1 est le sous-ensemble de $S(A)$ des ensembles qui ne contiennent pas i et A_2 est le sous-ensemble de $S(A)$ des ensembles qui contiennent i auxquels on a enlevé i.

Ainsi, l'arbre suivant est interprété comme l'ensemble $\{\varnothing, \{0, 1\}\}$.



Q 5) Donner l'ensemble défini par l'arbre combinatoire de la figure 1.

Q 6) Donner, sous forme de dessins, les trois arbres combinatoires correspondant aux trois ensembles $\{\{0\}\}$, $\{\varnothing, \{0\}\}$ et $\{\{0, 2\}\}$.

Q 7) Soit A un arbre combinatoire différent de $\bot$. Montrer que A contient au moins une feuille $\top$.

On se donne le type `ac` suivant pour représenter les arbres combinatoires.

OCaml
```
type ac =
  | Zero
  | Un
  | Comb of int * ac * ac
```

Le constructeur `Zero` represente $\bot$ et le constructeur `Un` représente $\top$.

Dans toute la suite, une partie de E est représentée par la liste de ses éléments **triés par ordre croissant**. On note `ensemble` le type OCaml correspondant, c'est à dire

OCaml
```
type ensemble = int list
```

Q 8) Écrire une fonction `un_elt : ac -> ensemble` qui prend en argument un arbre combinatoire A, supposé différent de $\bot$, et qui renvoie *un* ensemble $s \in S(A)$ arbitraire. On garantira une complexité au plus égale à la hauteur de A ce que l'on justifiera rapidement.

Q 9) Écrire une fonction `singleton : ensemble -> ac` qui prend en argument un ensemble $s \in \mathcal{P}(E)$ et qui renvoie l'arbre combinatoire représentant le singleton $\{s\}$. On garantira une complexité $O(n)$ ce que l'on justifiera rapidement.

Q 10) Écrire une fonction `appartient : ensemble -> ac -> bool` qui prend en argument un ensemble $s \in \mathcal{P}(E)$ et un arbre combinatoire A et qui teste si s appartient à $S(A)$. On garantira une complexité $O(n)$ ce que l'on justifiera rapidement.

Q 11) Écrire une fonction `cardinal : ac -> int` qui prend en argument un arbre combinatoire A et qui renvoie $\text{card}(S(A))$, le cardinal de $S(A)$.

Pour $n \in \mathbb{N}$, on note u_n le nombre d'arbres combinatoires distincts pour un ensemble E de cardinal n.

Q 12) Montrer que $u_0 = 2$ et que pour $n \geq 1$, $u_n = u_{n-1}^2$. En déduire une expression de u_n en fonction de n.

Q 13) Montrer que l'application S réalise une bijection entre l'ensemble des arbres combinatoires et l'ensemble $\mathcal{P}(\mathcal{P}(E))$.

III Chaînes d'addition

Dans cette partie on ne se préoccupera pas d'un éventuel dépassement du plus grand entier représentable. On pourra utiliser librement les fonctions `List.lenght` et `List.rev` en OCaml. Si x est un nombre réel positif, on note $\lfloor x \rfloor$ la partie entière par défaut de x et $\lceil x \rceil$ sa partie entière par excès.

On considère un ensemble U muni d'une loi de composition interne associative appelée *multiplication* et possédant un neutre pour cette loi noté e. Cette multiplication est notée avec le signe $\times$. Par exemple, U peut être l'ensemble des entiers ou des réels munis de la multiplication usuelle, l'élément neutre étant 1.

Définition III.1

Soit $a \in U$ et soit $n \in \mathbb{N}$. On définit a^n de la façon suivante :
- $a^0 = e$
- si $n \geq 1$, $a^n = a^{n-1} \times a$.

La multiplication étant associative, si i et j sont deux entiers positifs ou nuls de somme n alors $a^n = a^i \times a^j$. Un élément $a \in U$ et un entier $n \in \mathbb{N}^*$ étant donnés, on cherche à calculer a^n en s'intéressant au nombre de multiplications effectuées.

Exemple 1

Si $n = 14$, on peut calculer a^{14} en multipliant 13 fois l'élément a par lui même. On effectue alors 13 multiplications.

Q 14) Écrire en C une fonction de prototype `double expo(double a, int n)` qui calcule a^n en effectuant $n - 1$ multiplications.

Q 15) Justifier la terminaison de votre fonction en exhibant un variant de boucle.

Q 16) Proposer un invariant (utile et complet) pour la boucle de votre programme et le justifier rigoureusement.

Q 17) Démontrer soigneusement la correction de votre fonction.

Exemple 2

Si $n = 14$, on peut calculer a^{14} en calculant a^2 par $a^2 = a \times a$, puis a^3 par $a^3 = a^2 \times a$, puis a^6 par $a^6 = a^3 \times a^3$, puis a^7 par $a^7 = a^6 \times a$, puis enfin a^{14} par $a^{14} = a^7 \times a^7$. On aura ainsi obtenu le résultat en effectuant 5 multiplications.

Exemple 3

Si $n = 14$, on peut aussi calculer a^{14} en calculant a^2 par $a^2 = a \times a$, puis a^4 par $a^4 = a^2 \times a^2$, puis a^6 par $a^6 = a^4 \times a^2$ puis a^8 par $a^8 = a^4 \times a^4$, puis a^{14} par $a^{14} = a^8 \times a^6$. On aura ainsi obtenu le résultat en effectuant 5 multiplications.

✔

Dans toute la suite, a et n désignent respectivement un élément quelconque de U et un élément de $\mathbb{N}^*$.

L'objectif est de déterminer des algorithmes qui effectuent peu de multiplications.

Définition III.2

On appelle *suite pour l'obtention de la puissance n* toute suite non vide croissante d'entiers distincts $(n_0, \dots, n_r)$ telle que

- $n_0 = 1$
- $n_r = n$
- pour tout indice k vérifiant $1 \leq k \leq r$, il existe deux entiers i et j distincts ou non vérifiant $0 \leq i \leq k-1$, $0 \leq j \leq k-1$ et $n_k = n_i + n_j$ (la paire $\{i, j\}$ n'est pas forcément unique).

À une suite pour l'obtention de la puissance n correspond une suite de multiplications conduisant au calcul de a^n.

Exemple 4

Par exemple, la suite $(1, 2, 4, 6, 7, 12, 19)$ correspond au calcul de a^{19} en faisant les 6 multiplications suivantes : $a^2 = a \times a$, $a^4 = a^2 \times a^2$, $a^6 = a^4 \times a^2$, $a^7 = a^6 \times a$, $a^{12} = a^6 \times a^6$, $a^{19} = a^{12} \times a^7$.

Réciproquement, considérons un calcul de a^n dans lequel on fait en sorte d'ordonner les multiplications pour que les puissances calculées soient d'exposants croissants ; on peut associer à ce calcul une suite pour l'obtention de la puissance n.

Exemple 5

À l'exemple 1 est associé la suite $(1, 2, 3, 4, 5, \dots, 14)$ de longueur 14.

Exemple 6

À l'exemple 2 est associé la suite $(1, 2, 3, 6, 7, 14)$ de longueur 6.

Exemple 7

À l'exemple 3 est associé la suite $(1, 2, 4, 6, 8, 14)$ de longueur 6.

☞ **Remarque 1**

Le nombre de multiplications correspondant à une suite pour l'obtention de la puissance n est égal à la longueur de la suite diminuée de 1.

Q 18) Notons $(n_0, n_1, \dots, n_r)$ une suite pour l'obtention de la puissance n de a^n. Montrer que $\forall k \in [\![0, r]\!]$, $n_k \leq 2^k$.

Q 19) En déduire que tout calcul de a^n qui n'utilise que des multiplications nécessite un nombre de multiplications au moins égal à $\lceil \log_2(n) \rceil$.

Q 20) Donner une famille infinie de valeurs de n qui peuvent être calculées en effectuant exactement ce nombre de multiplications. Justifier la réponse.

On considère un algorithme appelé *par_division* ayant pour objectif le calcul de a^n. Cet algorithme s'appuie sur le principe récursif suivant :

- Si n vaut 1 alors a^n vaut a ;
- Sinon :
  - On calcule la partie entière par défaut, notée $\lfloor n/2 \rfloor$, de $n/2$;
  - On calcule par l'algorithme *par_division* la valeur de $b = a^{\lfloor n/2 \rfloor}$;
  - Si n est pair, alors $a^n = b \times b$ et sinon $a^n = (b \times b) \times a$.

Ainsi, pour obtenir a^{14} l'algorithme *par_division* fait appel au calcul de a^7 qui fait appel au calcul de a^3 (pour obtenir a^6 en multipliant a^3 par a^3 puis a^7 en multipliant a^6 par a) qui fait appel au calcul de a^1 (pour obtenir a^2 puis a^3). Les différentes puissances calculées sont les puissances 1, 2, 3, 6, 7 et 14. On constate que la suite pour l'obtention de la puissance 14 correspondant à l'algorithme *par_division* est la suite $(1, 2, 3, 6, 7, 14)$, de longueur 6. De même, la suite pour obtention de la puissance 19 correspondant à l'algorithme *par_division* est $(1, 2, 4, 8, 9, 18, 19)$ de longueur 7.

Q 21) Calculer (sans justification) la suite correspondant à l'algorithme *par_division* successivement :

(a) pour l'obtention de la puissance 15

(b) pour l'obtention de la puissance 16

(c) pour l'obtention de la puissance 27

(d) pour l'obtention de la puissance 125

Dans chaque cas, indiquer la longueur de la suite obtenue.

Q 22) Écrire en OCaml une fonction `par_division : int -> int list` qui calcule la suite de la puissance n correspondant à l'algorithme `par_division`. Par exemple :

```
OCaml
par_division 19;;
- : int list = [1; 2; 4; 8; 9; 18; 19]
```

Q 23) Montrer que l'algorithme `par_division` effectue au plus $2 \times \lfloor \log_2(n) \rfloor$ multiplications sur une entrée n.

Q 24) Montrer que ce nombre est atteint pour un nombre infini de valeurs de n.

On considère un algorithme *par_décomposition_binaire* dont l'objectif est aussi le calcul de a^n. Cet algorithme utilise la décomposition d'un entier suivant les puissances de 2. L'algorithme est expliqué ci-dessous à l'aide d'exemples.

Exemple 8

Soit $n = 14$. *On décompose* 14 *selon les puissances de* 2 : $14 = 2 + 4 + 8$. *On a donc :* $a^{14} = (a^2 \times a^4) \times a^8$, *ce qui conduit à calculer les puissances de* a *d'exposants* $2, 4, 8$ *mais aussi* 6 *et* 14 ; *la suite pour l'obtention de la puissance* 14 *correspondant à cet algorithme est la suite* $(1, 2, 4, 6, 8, 14)$.

Exemple 9

Soit $n = 18$. *On a* $18 = 2 + 16$ *et donc* $a^{18} = a^2 a^{16}$. *L'algorithme calcule les puissances d'exposant* $2, 4, 8, 16$ *puis* 18 ; *la suite pour l'obtention de la puissance* 18 *correspondant à cet algorithme est la suite* $(1, 2, 4, 8, 16, 18)$.

Exemple 10

Soit $n = 101$. *On a* $101 = 1 + 4 + 32 + 64$. *L'algorithme calcule* a^{101} *en utilisant les multiplications impliquées par la formule* $a^{101} = ((a \times a^4) \times a^{32}) \times a^{64}$; *on calcule les puissances* $2, 4, 5$ *(pour* $a \times a^4 = a^5$*),* $8, 16, 32, 37$ *(pour* $a^5 \times a^{32} = a^{37}$*),* 64 *et* 101 *(pour* $a^{37} \times a^{64} = a^{101}$*) ; la suite pour l'obtention de la puissance* 101 *correspondant à cet algorithme est la suite* $(1, 2, 4, 5, 8, 16, 32, 37, 64, 101)$.

De manière générale, l'algorithme procède en écrivant la décomposition unique de n comme une somme de puissances croissantes du nombre 2, et calcule la valeur cible de a^n en effectuant les produits correspondant aux sommes partielles de cette somme.

Q 25) Calculer la suite correspondant à l'algorithme *par_décomposition_binaire* (sans justification) :

(a) pour l'obtention de la puissance 15 ;

(b) pour l'obtention de la puissance 16 ;

(c) pour l'obtention de la puissance 27 ;

(d) pour l'obtention de la puissance 125.

Dans chaque cas, indiquer la longueur de la suite obtenue.

On considère la décomposition de n suivant les puissances croissantes du nombre 2 :

$$n = c_0 + c_1 \times 2 + \cdots + c_i \times 2^i + \cdots + c_k \times 2^k$$

où pour i vérifiant $0 \leq i < k$, le coefficient c_i vaut 0 ou 1 et c_k vaut 1.

On appelle *écriture binaire inverse* de n la suite $(c_0, c_1, \ldots, c_k)$.

Exemple 11

L'écriture binaire inverse de l'entier 14 *est* $(0, 1, 1, 1)$, *celle de l'entier* 18 *est* $(0, 1, 0, 0, 1)$ *et celle de l'entier* 101 *est* $(1, 0, 1, 0, 0, 1, 1)$.

Q 26) Écrire en Caml une fonction `binaire_inverse : int -> int list` qui à un entier $n \geq 1$ donné associe une liste correspondant à son écriture binaire inverse.

Q 27) Écrire en Caml la fonction `par_decomposition_binaire : int -> int list` qui à un entier $n \geq 1$ donné associe une liste correspondant à l'algorithme *par_decomposition_binaire*.

Q 28) Donner une suite de longueur 6 pour l'obtention de la puissance 15. Qu'en déduire quant aux algorithmes *par_division* et *par_decomposition_binaire* étudiés précédemment ?

— Fin du sujet —