Devoir surveillé n° 1 — 3 h

La calculatrice n'est pas autorisée.

Les programmes seront rédigés clairement et proprement, en couleur, et en mettant en valeur l'indentation. On veillera à choisir des noms de variables formant des mots complets et explicites. Il est indispensable de commencer par un brouillon avant de recopier la version finale au propre. *Dans toute question qui demande d'implémenter une fonction, vous veillerez à introduire votre programme par quelques lignes explicatives et à préciser le type de vos fonctions.*

I Un peu de typage

I.1) Déterminer le type des fonctions suivantes, et préciser quelles sont les définitions équivalentes :

OCaml
```
let f01 x y = x
let f02 x = fun y -> y
let f03 x y = x y
let f04 x y = y x
let f05 x y = x y y
let f06 x y = x (x y)
let f07 = fun x -> fun y -> y (x y)
let f08 x y = (x y) y
let f09 x y = x (y x)
let f10 x y = y (x y)
```

II La fonction d'Ackermann

Dans cet exercice *on ne demande pas* de justifier que la fonction termine, ni de calculer sa complexité.

La fonction d'Ackermann A est définie sur $\mathbb{N} \times \mathbb{N}$ par :

$$\begin{cases} A(0,p) = p+1 & \text{pour } p \geq 0 \\ A(n,0) = A(n-1,1) & \text{pour } n \geq 1 \\ A(n,p) = A(n-1,A(n,p-1)) & \text{si } n \geq 1, p \geq 1 \end{cases}$$

II.1) Calculer $A(0,0)$ puis $A(2,2)$.

II.2) Implémenter la fonction d'Ackermann en CAML en utilisant des expressions conditionnelles, sans utiliser de filtrage.

II.3) Implémenter la fonction d'Ackermann en CAML en utilisant un filtrage, sans utiliser d'expressions conditionnelles.

☞ **Remarque 1**

La fonction d'Ackermann croît très rapidement, en particulier $n \mapsto A(n,n)$ croît plus rapidement que n'importe quelle fonction polynôme, exponentielle ou même très certainement n'importe quelle autre fonction que vous êtes capables d'écrire (ou même d'imaginer). Par exemple, $A(4,2)$ a déjà 19729 chiffres et représente bien plus *que le nombre estimé d'atomes dans l'univers.*

III Représentation d'ensembles avec des listes

En informatique, un *ensemble* est une structure de données qui permet de stocker une collection d'objets, sans ordre particulier et sans doublons. Il s'agit d'une mise en œuvre informatique de la notion mathématique d'ensemble fini. Dans cet exercice on propose de représenter un ensemble d'entiers à l'aide d'une liste *sans doublons*.

En CAML, on peut définir des alias de type :

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

Dans toute la suite le type `ensemble` est maintenant *synonyme* de `int list`.

III.1) Par quoi peut-on représenter un ensemble vide ?

III.2) Proposer une fonction `est_vide : ensemble -> bool` qui teste si un ensemble est vide ou non. Quelle est sa complexité ?

III.3) Écrire une fonction `cardinal : ensemble -> int` qui renvoie le cardinal d'un ensemble (on rappelle que les listes sont supposées être sans doublons).

III.4) Écrire une fonction `appartient : int -> ensemble -> bool` qui vérifie si un élément est dans un ensemble. Quelle est sa complexité ?

III.5) Écrire une fonction `ajoute : int -> ensemble -> ensemble` qui ajoute un élément dans un ensemble. Insérer un élément déjà présent doit être sans effet. Quelle est sa complexité ?

III.6) Écrire une fonction `supprime : int -> ensemble -> ensemble` qui élimine un élément s'il est présent et est sans effet sinon.

III.7) Proposer une fonction réalisant l'intersection de deux ensembles, en indiquant son type et sa complexité.

III.8) Proposer une fonction réalisant l'union de deux ensembles, en indiquant son type et sa complexité.

III.9) Proposer une fonction permettant de tester l'égalité de deux ensembles, en indiquant son type et sa complexité.

IV Deux algorithmes de tri

IV.1 Tri par insertion

IV.1) Écrire une fonction `insere : 'a -> 'a list -> 'a list` prenant en argument un objet et une liste triée par ordre croissant et renvoyant la liste obtenue en insérant l'objet dans la liste en conservant son caractère trié.

Par exemple, `insere 5 [2; 4; 7; 8; 9]` renvoie `[2; 4; 5; 7; 8; 9]`.

IV.2) Écrire une fonction `tri_insertion : 'a list -> 'a list` triant la liste : lorsque c'est encore possible, il suffit d'insérer la tête dans la queue qui aura été triée récursivement.

IV.3) Quelle est la complexité temporelle dans le meilleur et le pire des cas ? Préciser la forme de listes pour lesquelles ces cas sont atteints.

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) Implémenter ce tri en CAML, en expliquant votre démarche, en donnant le type des fonctions utilisées et en analysant sa complexité dans le meilleur et dans le pire cas.

V Problème

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`. Soit x 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. L'ensemble U peut aussi être l'ensemble des matrices carrées booléennes (ou d'entiers ou de réels) d'une même dimension d avec le produit usuel comme multiplication, l'élément neutre étant la matrice identité booléenne (ou entière ou réelle) de dimension d.

Définition V.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.

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 V.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 2**

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.

V.1) Notons $(n_0, n_1, \dots, n_r)$ une suite pour l'obtention de la puissance n de a^n. Montrer que $\forall k \in [\![0, r]\!]$, $n_k \leq 2^k$.

V.2) En déduire que tout calcul de a^n qui n'utilise que des multiplications nécessite un nombre de multiplications au moins égal à $\lceil \log_2(n) \rceil$.

V.3) 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 q, de $n/2$;
  - On calcule par l'algorithme *par_division* la valeur de $b = a^q$;
  - 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.

V.4) 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.

V.5) Écrire en CAML 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]
```

V.6) Montrer que l'algorithme `par_division` effectue au plus $2 \times \lfloor \log_2(n) \rfloor$ multiplications sur une entrée n.

V.7) 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.

V.8) 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 $(1, 0, 1, 0, 0, 1, 1)$.

V.9) 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.

V.10) 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*.

V.11) 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 de l'énoncé —