DS1.

Important : Les programmes doivent être écrits en langage Ocaml. Si l'énoncé spécifie que des arguments de fonctions vérifient certaines propriétés, il n'est pas nécessaire de les vérifier dans votre code.

Vous ferez précéder ou suivre tous les codes de plus de 7 lignes d'une phrase expliquant l'idée que vous souhaitez implémenter.

1 Exercices

Exercice 1 Donner le type des fonctions suivantes ; dans le cas où le typage serait incohérent, justifier.

```
let f0 x y = 2*x -.y;;
let rec f1 h = match h with
|0 -> h
|_ -> f1 (h-1);;

let f2 x y = x y;;
let max3 x y z = max (max x y) z;;

let f4 x y =
x := !x + !y ;
y := !x - !y ;
x := !x - !y ;;
```

- $f0$ a un typage incohérent car le résultat de $2 * x$ est entier et on applique l'opérateur flottant $-.$ au résultat.
- $f1$ est de type `int->int`.
- $f2$ est de type `('a->'b) -> 'a -> 'b`.
- $max3$ est de type `('a -> 'a -> 'a -> 'a)` (on rappelle que `max` est polymorphe.

Exercice 2 On considère la fonction suivante, appelée la fonction `zip` :

```
let rec zip l1 l2 = match l1, l2 with
|[], [] -> []
|[], _  -> failwith "erreur"
|_, []  -> failwith "erreur"
|t::q, t1::q1 -> (t, t1)::(zip q q1);;
```

1. `'a list -> 'b list -> ('a*'b) list`
2. Cette fonction prend deux listes de même longueur et renvoie une liste de couples en correspondance dans chacune des listes.
3.
```
let rec unzip l = match l with
|[]          -> ([], [])
|(a,b)::q -> let q1, q2 = unzip q in
(a::q1, b::q2);;
```

Exercice 3 On considère la fonction suivante :

```
let rec f n = match n with
| n when n>100 -> n - 10
| _ -> f (f (n + 11));;
```

1. Le cas de base est celui des entiers supérieurs ou égaux à 101 puisqu'on n'effectue alors aucun appel récursif.

2. $f - 37$ renvoie une erreur à cause de la curryfication, f ne prenant qu'un argument, `Caml` interprète en f(-) 37. Il suffit alors de taper $f(-37)$.
3. Il semblerait que $f(x) = 91$ quel que soit $x < 101$.
4. On calcule $f(100) = f(f(111)) = f(101) = 91$, puis $f(99) = f(f(110)) = f(100) = 91$... et ainsi de suite jusqu'à 91 qui est point fixe de la fonction. En particulier pour tout entier $p > 0$, la composée $p-$ème de f appliquée à un entier compris entre 91 et 101 inclus vaut toujours 91.
5. Soit $x < 101$. Ecrivons la division Euclidienne de $101 - x$ par 11. il existe $p, r > 0$ tels que $101 - x = 11p + r$ et $r < 11$. On calcule $f(x) = f(-11p + 101 - r) = f(f(-11(p-1) + 101 - r)) = f(f(f(11(p-2) + 101 - r)) = \ldots = f^{(p)}$ où $f^p(101 - r)$ est la p ème itérée de f. Or comme r est le reste de la division Euclidienne par 11, $91 \geq 101 - r \leq 101$. Donc par la question qui précède $f(x) = f^p(101 - r) = 91$.

2 Problème

Dans ce 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` qui respectivement calculent la longueur d'une liste et la renversent. Pour tout nombre réel positif x, on notera $\lfloor x \rfloor$ la partie entière par défaut de x et $\lceil x \rceil$ sa partie entière par excès.

On considère un ensemble U disposant d'une loi de composition interne associative notée $\times$ que nous appellerons «multiplication». Cette loi possède un élément neutre que nous noterons e. Par exemple, si U est l'ensemble des entier munis de la multiplication usuelle, l'élément neutre est 1. L'ensemble U peut aussi être celui des matrices carrées d'une même dimension d avec le produit matriciel comme multiplication. L'élément neutre est alors la matrice identité.

On définit l'exponentiation de la manière inductive suivante. Soit $a \in U, n \in \mathbb{N}$, on pose :

— $a^0 = e$,

— pour $n \geq 1$, $a^n = a^{n-1} \times a$.

La multiplication étant associative, pour tous entiers positifs i, j, si on pose $n = i + j$, on a 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 minimisant le nombre de multiplications effectuées.

En effet, pour calculer a^{14} par exemple, on peut

1. multiplier 13 fois a par lui-même,
2. calculer $a^2 = a \times a$ (une seule multiplication) puis $a^3 = a^2 \times a$ puis $a^6 = a^3 \times a^3$ puis $a^7 = a^6 \times a$ puis enfin $a^{14} = a^7 \times a^7$. Au total on aura fait 5 multiplications.

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

De manière plus formelle, on appelle *suite pour l'obtention de la puissance n* toute liste croissante non vide d'entiers naturels distincts $(n_0, \ldots, n_r)$ 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 (éventuellement égaux) vérifiant $0 \leq i \leq k-1, 0 \leq j \leq k-1$ et $n_k = n_i + n_j$. Notez qu'il n'y a pas nécessairement unicité de la paire $\{i, j\}$.

Les deux exemples précédemment évoqués correspondaient aux suites pour l'obtention de la puissance 14 suivante :

1. $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)$ de longueur 14,
2. $(1, 2, 3, 6, 7, 14)$ de longueur 6.

On remarque que 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.

1. Notons $(n_0, \ldots, n_r)$ une suite pour l'obtention de la puissance n de a. Montrer que $\forall k \in [\![0, r]\!], n_k \leq 2^k$. Le résultat se démontre bien par récurrence forte.
   — **Initialisation :** $n_0 = 1 \leq 2^0 = 1$
   — **Hérédité :** Supposons que la propriété soit vraie pour tout $p < k$, par définition de n_k il existe deux indices $i \leq k-1, j \leq k-1$ tels que $n_k = n_i + n_j$. Par hypothèse de récurrence $n_k \leq 2^i + 2^j \leq 2^{k-1} + 2^{k-1=2^k}$.

   L'hérédité est établie et la propriété est vraie quel que soit k.

2. En déduire que tout calcul de a^n n'utilisant que des multiplications nécessite un nombre de multiplications au moins égal à $\lceil \log_2(n) \rceil$. Pour qu'une suite de longueur p puisse calculer n, par le résultat précédent, il faut que $n \leq 2^p$ Donc il faut que $p \geq \log_2(n)$, donc comme p est un entier, $p \geq \lceil \log_2(n) \rceil$.
3. Donner une famille de valeurs de n qui peuvent être calculées en effectuant exactement ce nombre de multiplications.
   Si $n = 2^p$ est une puissance de 2, la suite associée est $(1, 2, 4, \ldots, 2^p)$, il faut donc p multiplications, ce qui est bien égal à $\lceil \log_2(n) \rceil$.

2.1 Algorithme par division

On considère un algorithme appelé *parDivision* [1] ayant pour objectif le calcul de a^n par une approche du type «diviser-pour-régner». Sa description récursive succinte est donnée ci-dessous :

— Si $n = 1$ alors $a^n = a$,
— — On calcule la partie entière par défaut, $q = \lfloor \frac{n}{2} \rfloor$,
— On calcule récursivement $b = a^q$,
— Si n est pair alors $a^n = b \times b$, sinon $a^n = (b \times b) \times a$.

4. Proposer une fonction `calculParDivision : int -> int -> int` qui implémente le calcul de a^n par l'algorithme décrit ci-dessus. Votre fonction devra ne faire apparaître *qu'un seul* appel récursif.

```
let rec calculParDivision a n = match n with
  |1 -> a
  |_ -> let b =  calculParDivision a (n/2) in
        if n mod 2 = 0 then b*b
     else a*b*b;;
```

5. Pour un entier n, combien d'appels récursifs sont ils nécessaires pour atteindre le cas de base ?
   Décomposons n en base 2, $n = (1a_1 \ldots a_p)$; on a $p = \lfloor \log_2(n) \rfloor$. A chaque appel récursif, n est divisé par deux en partie entière, ce qui revient à enlever le dernier chiffre binaire. Donc Au bout de p appels récursifs, on est arrivé au cas de base.
6. Montrer que l'algorithme `calculParDivision` effectue au plus $2\lfloor \log_2(n) \rfloor$ multiplications pour le calcul de a^n.
   A chaque appel récursif, on effectue au plus deux multiplications. Il y a $\lfloor \log_2(n) \rfloor$ appels récursifs. D'où le résultat
7. Donner sans justifier une famille (infinie) de nombres n nécessitant ce nombre de multiplications pour le calcul de a^n.
   Les nombres de Mersenne de la forme $n = 2^p - 1$ atteignent la borne précédente.
8. Donner sans justifier la suite des puissances calculées pour obtenir :
   (a) a^{14} : (1,2,3,6,7,14)
   (b) a^{19}, : (1,2,4,8,9,18,19).
   (c) a^{125}. : (1,2,3,6,7,14,15,30,31,62,124,125).
9. Ecrire en Ocaml une fonction `parDivision : int -> int list` qui calcule la suite de la puissance n correspondant à l'algorithme `parDivision`. Par exemple : `parDivision 3` renverra la liste `[1,2,3]`.

```

let parDivision a n =
let rec parDivisionRec a n acc = match n with
  |v when v = 1 -> acc
  |v when v mod 2 = 0 -> parDivisionRec a (n/2) ((n/2)::acc)
  |_ -> parDivisionRec a (n/2) ((n/2)::(n-1)::acc)
     in parDivisionRec a n [n];;
```

10. Montrer que l'algorithme `calculParDivision` effectue au plus $2\lfloor \frac{n}{2} \rfloor$ multiplications pour une entrée n.
11. Montrer que ce nombre est atteint pour un nombre infini de valeurs de n.

1. Aussi appelé «algorithme d'exponentiation rapide».

2.2 Algorithme par décomposition binaire

Nous allons nous intéresser maintenant à un autre algorithme, dit *par décomposition binaire*. On rappelle tout d'abord que tout entier naturel non nul peut se décomposer de manière unique en binaire (base 2), c'est à dire que

$$\forall n \geq 1, \exists k > 0, a_0, a_1, \ldots, a_k \in \{0,1\}, a_k \neq 0, n = \sum_{p=0}^{k} a_p 2^p.$$

Nous allons commencer par donner un pseudo-code vague pour décrire la méthode de décomposition binaire puis nous allons la préciser grâce à des exemples. L'idée de l'algorithme est la suivante pour calculer a^n :

— on décompose n en base 2, $n = \sum_{p=0}^{k} a_p 2^p$.

— on calcule la valeur cible de a^n en effectuant les produits correspondant aux sommes partielles de la somme ci-dessus.

Donnons deux exemples :

— Soit $n = 14$, on écrit $14 = 2 + 2^2 + 2^3 = 2 + 4 + 8$. On a donc $a^14 = (a^2 \times a^4) \times a^8$, ce qui conduit donc à calculer les puissances de a d'exposants $2, 4, 8$ mais également 6 et 14 (à cause des produits intermédiaires). La suite pour l'obtention de la puissance 14 est donc $(1, 2, 4, 6, 8, 14)$.

— Soit $n = 18$. On a $18 = 2^1 + 2^4 = 2 + 16$. L'algorithme consiste à calculer les puissances d'exposants $2, 4, 8, 16$ puis 18. La suite correspondant est donc $(1, 2, 4, 8, 16, 18)$.

— Soit $n = 101 = 1 + 2^2 + 2^5 + 2^6 = 1 + 4 + 32 + 64$. Donc $a^{101} = \left((a \times a^4) \times a^{32}\right) \times a^{64}$. On calcule les puissances $2, 4, 5$ (pour $(a \times a^4)$), $8, 16, 32, 37$ (pour $a^5 \times a^{32}$), $64, 101$ ($a^{37} \times a^{64}$).

1. Donner sans justifier la suite correspondant à l'algorithme *par décomposition binaire* :
   (a) pour l'obtention de la puissance 15, $15 = 1 + 2 + 2^2 + 2^3$ donc la suite correspondante est (1,2,3,4,7,8,15)
   (b) pour l'obtention de la puissance 16, $16 = 2^4$ donc la suite correspondante est (2,4,8,16)
   (c) pour l'obtention de la puissance 125. $125 = 64 + 32 + 16 + 8 + 4 + 1 = 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$ d'où la suite $(1, 2, 4, 5, 8, 13, 16, 29, 32, 61, 64, 125)$

On appelle *écriture binaire inverse* d'un entier n la suite $(a_0, \ldots, a_k)$ des termes de la décompositions binaires (on met le bit de poids faible en premier). Par exemple $14 = 2 + 2^2 + 2^3$ a pour écriture binaire inverse $(0, 1, 1, 1)$.

1. Ecrire en CAML une fonction `binaireInverse : int -> int list` qui à un entier $n \geq 1$ donné associe une liste correspondant à son écriture binaire inverse.

```
let binaireInverse n =
let rec binaireInverseAux n acc = match n with
  | x when x = 0 -> acc
  | x when x mod 2 = 0 -> binaireInverseAux (n/2) (0::acc)
  | _ -> binaireInverseAux (n/2) (1::acc)
in
List.rev (binaireInverseAux n []);;
```

2. Ecrire en CAML la fonction `parDecompositionBinaire : int -> int list` qui à un entier $n \geq 1$ associe une liste correspondant à l'algorithme par décomposition binaire.

```
let puissance a b = match b with
   |0-> 1
   |_ -> a*(puissance (b-1));;

let parDecompositionBinaire n =
  let rec parDecompositionBinaireAux n acc acc2 acc3 =
  match (binaireInverse n) with
    (*acc2 contient la somme partielle, on ne l'augmente que si on lit un 1,
    si on lit un 0, on calcule la puissance 2**acc3*)
    |1::[] -> let u = puissance 2 acc3 in  u::acc
    |0::q -> let u = puissance 2 acc3 in parDecompositionBinaireAux (n/2) (u::acc)
    |1::q when acc2 <> 0->  let u = puissance 2 acc3 in parDecompositionBinaireAux
    |1::q when acc2 = 0->  let u = puissance 2 acc3 in parDecompositionBinaireAux (
    |_ -> acc
in List.rev (parDecompositionBinaireAux (n) [] 0 0);;
```

3. Donner une suite de longueur 6 pour l'obtention de la puissance 15. Qu'en déduire quant aux algorithmes des deux parties précédentes ?

   La suite $(1, 2, 3, 5, 10, 15)$ semble convenir. Les algorithmes précédents ne sont donc pas optimaux tout le temps.