OPTION INFORMATIQUE
Devoir Surveillé 1

1 Stratégies d'exponentiation (Mines 2017)

On suppose dans la suite disposer de deux fonctions

```
longueur : 'a list -> int
inverse : 'a list -> 'a list
```

donnant pour la première le nombre d'éléments d'une liste et pour la seconde une liste "miroir" de la première. Ainsi, `inverse [1;2;3]` est égal à la liste `[3;2;1]`.

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

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.

L'objectif est de déterminer des algorithmes qui effectuent peu de multiplications. 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 appelle *suite pour l'obtention de la puissance n* toute suite non vide croissante d'entiers 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 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).

A une suite pour l'obtention de la puissance n correspond une suite de multiplications conduisant au calcul de a^n. 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.
A l'exemple 1 est associé la suite $(1, 2, 3, 4, 5, \ldots, 14)$ de longueur 14.
A l'exemple 2 est associé la suite $(1, 2, 3, 6, 7, 14)$ de longueur 6.
A l'exemple 3 est associé la suite $(1, 2, 4, 6, 8, 14)$ de longueur 6.
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. Montrer que tout calcul de a^n qui n'utilise que des multiplications nécessite un nombre de multiplications au moins égal à $\lceil \log_2(n) \rceil$. 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" 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.

2. Calculer (sans justification) la suite correspondant à l'algorithme *par_division* successivement :
   - pour l'obtention de la puissance 15 ;
   - pour l'obtention de la puissance 16 ;
   - pour l'obtention de la puissance 27 ;
   - pour l'obtention de la puissance 125.

   Dans chaque cas, indiquer la longueur de la suite obtenue.
3. écrire en Caml la fonction `par_division : int → int list` qui calcule la suite de la puissance n correspondant à l'algorithme *par_division*.
4. Montrer que l'algorithme *par_division* appliqué à n effectue au plus $2 \times \lfloor \log_2(n) \rfloor$. 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.

- 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 $(1, 2, 4, 6, 8, 14)$.
- 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 $(1, 2, 4, 8, 16, 18)$.
- 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.

5. Calculer (sans justification) la suite correspondant à l'algorithme *par_décomposition_binaire* successivement :
   - pour l'obtention de la puissance 15 ;
   - pour l'obtention de la puissance 16 ;
   - pour l'obtention de la puissance 27 ;
   - 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 $(c_0, c_1, \ldots, c_k)$.
Par exemple, 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 $(1, 0, 1, 0, 0, 1, 1)$.

6. 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.
7. 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*.
8. On suppose que $n = 3^k$ où $k \in \mathbb{N}$. En utilisant la formule $3^k = 3^{k-1} + 2 \times 3^{k-1}$, montrer qu'il existe une suite pour l'obtention de la puissance n de longueur $2k + 1$. Indiquer la longueur de cette suite pour $n = 27$ et comparer avec le résultat obtenu par l'algorithme *par_division*.
9. Soit $k \in \mathbb{N}$. Ecrire en Caml une fonction `suite_3` qui à un entier n supposé être une puissance de 3 associe la liste de longueur $2k + 1$ évoquée en question précédente.
10. On suppose que $n = 5^k$ avec $k \in \mathbb{N}$. Montrer qu'il existe une suite pour l'obtention de la puissance n de longueur $3k + 1$. Indiquer la suite dans le cas $n = 125$ et comparer avec le résultat dans le cas de l'algorithme *par_division*.
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 ?
12. On considère un tableau T, indicé à partir de 0, contenant une suite pour l'obtention d'une certaine puissance positive n. Soit k un entier compris entre 1 et la longueur de T diminuée de 1. Soit *val* la valeur contenue dans T à l'indice k. On sait que *val* est la somme de deux valeurs du tableau T situées à des indices (éventuellement confondus, éventuellement non uniques) strictement inférieurs à k. Programmer une fonction `chercher_indice : int array → int → int*int` qui étant donnés T et k calcule un couple (i, j) convenable (n'importe lequel). On supposera que k vérifie les contraintes exposées ci-dessus.
    Par exemple, si $T = [|1; 2; 3; 4; 7; 14; 17; 31|]$, la longueur du tableau est 8. Si $k = 6$, *val* vaut 17 et la fonction renvoie $(2, 5)$ (correspondant aux valeurs 3 et 14 du tableau). Si $k = 1$, la fonction renvoie $(0, 0)$.
13. Soit x un réel représenté par un élément de type `float`. Soit T un tableau contenant une suite pour l'obtention d'une certaine puissance positive n. écrire en Caml une fonction `puissance : float → int array → float` qui prend en argument x et T et renvoie x^n en utilisant la stratégie associée à la suite associée à T. Indiquer (sans justification) la complexité $C(k)$ de cette fonction.
14. Décrire le principe d'un algorithme *suite_optimale* permettant d'exhiber une suite de longueur minimale pour l'obtention de la puissance n en effectuant une énumération exhaustive des suites possibles. Cette fonction fera appel à un algorithme récursif *suite_optimale_rec* dont on donnera aussi le principe.
15. Ecrire les fonctions `suite_optimale_rec` et `suite_optimale` correspondant aux algorithmes décrits.

2 Étude des classes sylvestres (CCINP 2021)

Étant donné un arbre binaire de recherche T, sa classe sylvestre est l'ensemble des mots qui donnent l'arbre T après insertion dans l'arbre vide. L'objectif de cette partie est de donner une caractérisation et une description de cet ensemble. On commence par rappeler la structure des arbres binaires de recherche ainsi que leurs propriétés usuelles. Puis, on introduit la notion de S-équivalence sur les mots qui permet de caractériser la classe sylvestre d'un arbre T. Enfin, à l'aide du produit de mélange, on présente un algorithme calculant la classe sylvestre d'un arbre donné.
Dans toute la suite, Σ désigne un alphabet fini totalement ordonné. Le symbole ε désigne le mot vide.

2.1 Algorithme d'insertion dans un arbre binaire

2.1.1 Définition (Arbre binaire)

Un **arbre binaire** T (étiqueté par les éléments de Σ) est récursivement soit :

- l'arbre vide que l'on note $\circ$;
- un triplet (T_g, r, T_d) où r est un élément de Σ, T_g et T_d des arbres binaires. Les éléments r, T_g et T_d sont respectivement appelés racine, sous-arbre gauche et sous-arbre droit de T.

2.1.2 Définition (Arbre binaire de recherche)

Un **Arbre Binaire de Recherche** (abrégé en ABR) T est récursivement soit :

- l'arbre vide ;
- un triplet (T_g, r, T_d) où r est un élément de Σ, T_g et T_d des ABR. De plus, toute valeur apparaissant dans T_g est strictement inférieure à r et toute valeur apparaissant dans T_d est supérieure ou égale à r.

2.1.3 Définition (Insertion dans un arbre binaire)

L'insertion d'un élément a de Σ dans un arbre binaire T est une opération que l'on note $T \leftarrow a$ définie récursivement comme suit :

- si $T = \circ$, alors $T \leftarrow a = (\circ, a, \circ)$;
- si $T = (T_g, r, T_d)$ et $r \leq a$, alors $T \leftarrow a = (T_g, r, T_d \leftarrow a)$;
- si $T = (T_g, r, T_d)$ et $r > a$, alors $T \leftarrow a = (T_g \leftarrow a, r, T_d)$.

On définit récursivement alors l'insertion d'un mot w dans un arbre binaire T noté également $T \leftarrow w$ comme suit :

- si $w = \varepsilon$, alors $T \leftarrow w = T$;
- si $w = av$ avec $a \in \Sigma$, alors $T \leftarrow w = (T \leftarrow a) \leftarrow v$.

Dans le cas où T est un ABR, on admet que $T \leftarrow a$ et $T \leftarrow w$ sont également des ABR.
Étant donné un mot w sur Σ, l'arbre binaire de recherche associé à w est l'arbre $\circ \leftarrow w$.

2.1.4 Définition (Classe sylvestre)

Soit T un arbre binaire de recherche. La classe sylvestre de T que l'on note $\mathrm{Syl}(T)$ est l'ensemble des mots w vérifiant : $(\circ \leftarrow w) = T$.
Par exemple :
si $T = ((\circ, a, \circ), b, (\circ, c, \circ))$, on a $\mathrm{Syl}(T) = \{bac, bca\}$;
si $T = \circ$, alors $\mathrm{Syl}(T) = \{\varepsilon\}$.
Pour simplifier la représentation des arbres binaires, on peut utiliser des graphes.
Par exemple, l'arbre $T = ((\circ, a, ((\circ, b, \circ), a, \circ)), f, (\circ, c, \circ))$ est représenté par :

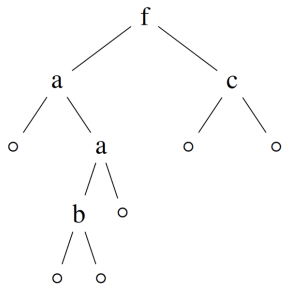

FIGURE 1 – Représentation de T

Dans la suite, les lettres de Σ sont représentées en Ocaml par des `char` et les mots sur l'alphabet Σ par des `char list`. Ainsi, le mot $w =$ "arbre" est représenté par la liste $w = [\prime a\prime; \prime r\prime; \prime b\prime; \prime r\prime; \prime e\prime]$. On représente les arbres binaires en Ocaml à l'aide de la structure arbre suivante :

```
type 'a arbre = Vide | Noeud of ('a arbre) * 'a * ('a arbre).
```

Ainsi, l'arbre $T = (\circ, a, (\circ, b, \circ))$ est représenté en Ocaml par :

```
Noeud(Vide, 'a', Noeud(Vide, 'b', Vide)).
```

16. Représenter le graphe de l'ABR associé au mot "fantastique", l'ordre sur les lettres étant l'ordre alphabétique.
17. Écrire une fonction récursive en Ocaml de signature :

    ```
    insertion_lettre : char -> char arbre -> char arbre
    ```

    et telle que `insertion_lettre a t` est l'arbre binaire obtenu en insérant la lettre `a` dans l'arbre binaire `t`.
18. Écrire une fonction récursive en Ocaml de signature :

    ```
    insertion_mot : char list -> char arbre -> char arbre
    ```

    et telle que `insertion_mot w t` est l'arbre binaire obtenu en insérant le mot `w` dans l'arbre binaire `t`.

2.1.5 Définition (Lecture préfixe)

Soit T un arbre binaire. La lecture préfixe de T que l'on note w_T est le mot défini récursivement par :

- si $T = \circ$, alors $w_T = \varepsilon$;
- si $T = (T_g, r, T_d)$, alors $w_T = r w_g w_d$ où w_g et w_d sont les lectures préfixes respectives de T_g et de T_d.

19. Expliciter w_T pour l'arbre binaire T représenté ci-dessous :

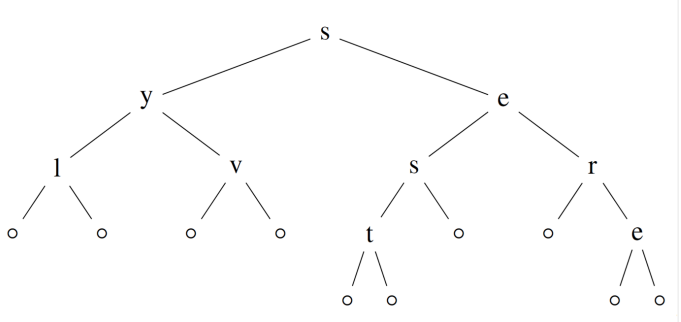

FIGURE 2 – Représentation de T de Q19

20. Écrire une fonction récursive en Ocaml de signature :

    ```
    prefixe : char arbre -> char list
    ```

    et telle que `prefixe t` est le mot qui correspond à la lecture préfixe de l'arbre `t`.
21. Soit T un ABR, soit w_T la lecture préfixe de T. Montrer que T est l'ABR associé à w_T.

2.2 Une relation d'équivalence sur les mots

2.2.1 Définition (S-adjacence)

Soient u et v deux mots sur Σ. On dit que u est S-adjacent à v (ou que u et v sont S-adjacents) s'il existe trois lettres $a < b \leq c$ de Σ et trois mots f_1, f_2, f_3 sur Σ vérifiant :

$$(u = f_1bf_2acf_3 \text{ et } v = f_1bf_2caf_3) \text{ ou } (u = f_1bf_2caf_3 \text{ et } v = f_1bf_2acf_3).$$

2.2.2 Définition (S-équivalence)

Soient u et v deux mots sur Σ. On dit que u est S-équivalent à v (ou que u et v sont S-équivalents) s'il existe $n \in \mathbb{N}, (w_0, w_1, \ldots, w_n) \in (\Sigma^*)^{n+1}$ vérifiant :

$$u = w_0, v = w_n, \forall i \in \{0, 1, \ldots, n-1\}, w_i \text{ est S-adjacent à } w_{i+1}.$$

22. Montrer que la relation "être S-équivalent à" est bien une relation d'équivalence sur Σ^*.
23. Soient a, b, c trois lettres de Σ vérifiant $a < b \leq c$. Montrer que pour tout arbre binaire de recherche T contenant au moins la lettre b, on a : $T \leftarrow ac = T \leftarrow ca$. On pourra raisonner par récurrence sur le nombre de noeuds de T.
24. Montrer que si deux mots u et v sont S-équivalents, alors les ABR $(\circ \leftarrow u)$ et $(\circ \leftarrow v)$ sont égaux.
25. Soit w un mot de longueur $n \geq 1$ sur Σ, soit r la première lettre de w. On veut montrer qu'il existe un mot $w' = ruv$ où u (respectivement v) est un mot dont toutes les lettres sont plus petites strictement (respectivement plus grandes au sens large) que r et tels que w et w' sont S-équivalents. On note A l'ensemble des mots S-équivalents à w.
    Pour tout $a = a_0a_1 \ldots a_{n-1} \in A$, on pose :

    $$N(a) = \{(i, j) \in [\![1, n-1]\!]^2 \mid i < j, a_j < r \leq a_i\}.$$

    (a) Justifier que l'ensemble A est fini.

    (b) Justifier que tous les mots de A commencent par r.

    (c) Soit $a \in A$. Montrer que si $N(a) \neq \emptyset$, alors il existe $k \in [1, n-2]$ tel que $(k, k+1) \in N(a)$.

    (d) Soit $a \in A$. Montrer que si $N(a) = \emptyset$, alors il existe u (respectivement v) un mot dont toutes les lettres sont plus petites strictes (respectivement plus grandes ou égales) que r et tels que $a = ruv$.

    (e) Soit a un élément de A tel que le nombre d'éléments de $N(a)$ soit le plus petit possible. Montrer par l'absurde que $N(a) = \emptyset$ et en déduire qu'il existe u, v vérifiant les conditions de la question tels que ruv et w sont S-équivalents.
26. Montrer que tout mot w est S-équivalent à w_T, où w_T est la lecture préfixe de $T = (\circ \leftarrow w)$. On pourra raisonner par récurrence sur la longueur du mot w et exploiter les résultats de la question précédente.
27. En déduire que deux mots sont S-équivalents si et seulement si ils sont des éléments d'une même classe sylvestre.

2.3 III.3 - Construction de classes sylvestres

Pour construire des classes sylvestres, il est utile d'utiliser le produit de mélange.

2.3.1 Définition (Mélange)

Soient u et v deux mots sur Σ. Le **mélange** de u et v, noté $u \bowtie v$, est l'ensemble récursivement défini comme suit :

- si $v = \varepsilon$, alors $u \bowtie v = \{u\}$;
- si $u = \varepsilon$, alors $u \bowtie v = \{v\}$;
- si $u = au'$ et $v = bv'$ où a et b sont des lettres, u' et v' des mots, alors : $u \bowtie v = \{aw \mid \exists w \in u' \bowtie v\} \cup \{bw \mid \exists w \in u \bowtie v'\}$.

2.3.2 Définition (Mélange de langages)

Soient L et M deux langages. Le **mélange des langages** L et M, noté $L \bowtie M$, est défini par :

$$L \bowtie M = \bigcup_{u \in L, v \in M} u \bowtie v.$$

Si au moins un des deux langages est égal à l'ensemble $\emptyset$ ($\neq \{\varepsilon\}$), alors on a $L \bowtie M = \emptyset$.
Étant donné un ABR $T = (T_s, r, T_d)$, on peut montrer que :

$$\mathrm{Syl}(T) = \{rw \mid \exists w \in \mathrm{Syl}(T_s) \bowtie \mathrm{Syl}(T_d)\}.$$

On admet ce résultat pour la suite de cette sous-partie. On note que $\mathrm{Syl}(\circ) = \{\varepsilon\}$.

28. Expliciter sans justification tous les éléments de l'ensemble `abba` $\bowtie$ `ba`.
29. Écrire une fonction récursive en Ocaml de signature :

    ```
    ajout_lettre : char -> char list list -> char list list
    ```

    et telle que `ajout_lettre a liste` est la liste de mots de la forme `a::w` où `w` est un élément de `liste`.
30. Écrire une fonction récursive en Ocaml de signature :

    ```
    shuffle : char list -> char list -> char list list
    ```

    et telle que `shuffle u v` est une liste contenant exactement tous les mots de $u \bowtie v$ avec répétitions possibles d'un même mot. On devra utiliser la fonction auxiliaire `ajout_lettre` et l'opérateur de concaténation `@`.
31. Écrire une fonction récursive en Ocaml de signature :

    ```
    shuffle_l : char list list -> char list list -> char list list
    ```

    et telle que `shuffle_l l m` est une liste contenant exactement tous les mots de $L \bowtie M$ où la liste `l` (respectivement `m`) représente le langage L (respectivement M), avec répétitions possibles d'un même mot. Seules les fonctions définies précédemment peuvent être utilisées en fonctions auxiliaires.
32. Écrire une fonction récursive en Ocaml de signature :

    ```
    sylvestre_T : char arbre -> char list list
    ```

    et telle que `sylvestre_T t` est une liste contenant exactement tous les éléments de la classe sylvestre de `t`. Seules les fonctions définies précédemment peuvent être utilisées en fonctions auxiliaires.