**École des PONTS ParisTech,
ISAE-SUPAERO, ENSTA ParisTech,
TÉLÉCOM ParisTech, MINES ParisTech,
MINES Saint-Étienne, MINES Nancy,
TÉLÉCOM Bretagne, ENSAE ParisTech (Filière MP).**

CONCOURS 2016

ÉPREUVE D'INFORMATIQUE

**(Durée de l'épreuve : 3 heures)
L'usage d'une calculatrice est autorisé.**

**Sujet mis à la disposition des concours :
Concours Commun TPE/EIVP, Concours Mines-Télécom,
Concours Centrale-Supélec (Cycle international).**

Les candidats sont priés de mentionner de façon apparente sur la première page de la copie :

INFORMATIQUE - MP

L'énoncé de cette épreuve comporte 10 pages de texte. L'épreuve est composée de deux exercices indépendants, l'ensemble du sujet comportant 24 questions.

— Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

— Tout résultat fourni dans l'énoncé peut être utilisé pour les questions ultérieures même s'il n'a pas été démontré.

— Il ne faut pas hésiter à formuler les commentaires qui semblent pertinents même lorsque l'énoncé ne le demande pas explicitement.

Préliminaire concernant la programmation

Il faudra coder des fonctions à l'aide du langage de programmation Caml, tout autre langage étant exclu. Lorsque le candidat écrira une fonction, il pourra faire appel à d'autres fonctions définies dans les questions précédentes ; il pourra aussi définir des fonctions auxiliaires. Quand l'énoncé demande de coder une fonction, il n'est pas nécessaire de justifier que celle-ci est correcte, sauf si l'énoncé le demande explicitement. Enfin, si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien vérifiées.

Dans les énoncés du premier exercice, un même identificateur écrit dans deux polices de caractères différentes désignera la même entité, mais du point de vue mathématique pour la police en italique (par exemple n) et du point de vue informatique pour celle en romain avec espacement fixe (par exemple `n`).

1 Graphe du Web

Le *World Wide Web*, ou Web, est un ensemble de pages Web (identifiées de manière unique par leurs *adresses Web*, ou *URL* pour *Uniform Resource Locators*, de la forme `http://mines-ponts.fr/index.php`) reliées les unes aux autres par des hyperliens. Le Web est souvent modélisé comme un graphe orienté dont les sommets sont les pages Web et les arcs les hyperliens entre pages. Le Web étant potentiellement infini, on s'intéresse à des sous-graphes du Web obtenus en *naviguant* sur le Web, c'est-à-dire en le parcourant page par page, en suivant les hyperliens d'une manière bien déterminée. Ce parcours du Web pour en collecter des sous-graphes est réalisé de manière automatique par des logiciels autonomes appelés *Web crawlers* ou *crawlers* en anglais, ou *collecteurs* en français.

Fonctions utilitaires

Nous allons tout d'abord coder certaines fonctions de manipulation de structures de données de base, qui seront utiles dans le reste de l'exercice.

☐ 1 – Coder une fonction `aplatir : ('a * 'a list) list -> 'a list`, telle que, si *liste* est une liste de couples $[(x_1, l_{x_1}); \ldots; (x_n, l_{x_n})]$, où chaque x_i est un élément de type `'a`, et l_{x_i} une liste d'éléments de type `'a` de la forme $[y_{i1}; \ldots; y_{ik_i}]$, `aplatir liste` est une liste d'éléments de type `'a` :

$$[x_1; y_{11}; \ldots; y_{1k_1}; x_2; y_{21}; \ldots; y_{2k_2}; \ldots; x_n; y_{n1}; \ldots; y_{nk_n}].$$

☐ 2 – Coder une fonction `tri_fusion : ('a * 'b) list -> ('a * 'b) list` triant une liste de couples (x, y) par ordre *décroissant* de la valeur de la seconde composante y de chaque couple. On devra utiliser l'algorithme de tri par partition–fusion (aussi appelé « tri fusion »). Quelle est la complexité de cet algorithme ?

On va utiliser dans la suite de l'exercice un type de données `dictionnaire` qui permet de stocker des couples formés d'une chaîne de caractères (une *clef*) et d'un entier (une *valeur*). On dit que le dictionnaire *associe* la valeur à la clef. À chaque clef présente dans le dictionnaire est associée une seule valeur. Les fonctions suivantes sont supposées être prédéfinies :

— `dictionnaire_vide : unit -> dictionnaire`.
L'appel `dictionnaire_vide ()` crée un nouveau dictionnaire vide.

— `ajoute : string -> int -> dictionnaire -> dictionnaire`.
L'appel `ajoute clef valeur dict` renvoie un nouveau dictionnaire identique au dictionnaire *dict*, sauf qu'un couple (*clef*, *valeur*) y a été ajouté. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.

— `contient : string -> dictionnaire -> bool`.
L'appel `contient clef dict` renvoie un booléen indiquant s'il y a un couple dont la clef est *clef* dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.

— `valeur : string -> dictionnaire -> int`.
L'appel `valeur clef dict` renvoie la valeur associée à la clef *clef* dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire. Cette fonction ne peut être appelée que si la clef *clef* est présente dans le dictionnaire.

On suppose pour la suite de l'exercice que le type de données `dictionnaire` est prédéfini ; on ne demande pas de l'implémenter.

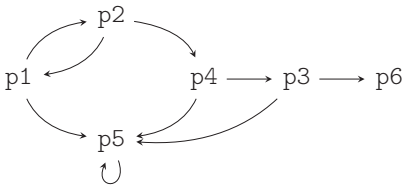
☐ 3 – Coder `unique : string list -> string list * dictionnaire`, qui est telle que `unique liste` renvoie un couple ($liste'$, *dict*) où $liste'$ est la liste des chaînes de caractères de *liste* distinctes (dans l'ordre de leur première occurrence dans *liste*) et où *dict* associe à chaque chaîne de caractères dans $liste'$ sa position dans $liste'$ (en numérotant à partir de 0). Ainsi l'appel `unique ["x";"zz";"x";"x";"zz";"yt"]` renvoie un couple formé de la liste `["x";"zz";"yt"]` et d'un dictionnaire associant à `"x"` la valeur `0`, à `"zz"` la valeur `1` et à `"yt"` la valeur `2`.

☐ 4 – Quelle est la complexité de la fonction `unique` en terme de la longueur n de la liste *liste* en argument et du nombre m d'éléments distincts dans la liste *liste* ? Justifier la réponse.

Crawler simple

Nous allons maintenant implémenter un crawler simple en Caml. On suppose fournie une fonction `recupere_liens : string -> string list` prenant en argument l'URL d'une page Web p et renvoyant la liste des URL des pages q pour lesquelles il existe un hyperlien de p à q, dans l'ordre lexicographique.

Pour illustrer le comportement de cette fonction, nous considérons un exemple de mini-graphe du Web à six pages et neuf hyperliens comme suit :



Dans cette représentation, p1, p2, etc., sont les URL de pages Web (simplifiées pour l'exemple), et les arcs représentent les hyperliens entre pages Web.

Dans ce mini-graphe, un appel à `recupere_liens "p1"` retourne la liste `["p2"; "p5"]`.

Un *crawler* est un programme qui, à partir d'une URL, parcourt le graphe du Web en visitant progressivement les pages dont les liens sont présents dans chaque page rencontrée, en suivant une stratégie de parcours de graphe (par exemple, largeur d'abord, ou profondeur d'abord). À chaque nouvelle page, si celle-ci n'a pas déjà été visitée, tous ses hyperliens sont récupérés et ajoutés à une liste de liens à traiter. Le processus s'arrête quand une condition est atteinte (par exemple, un nombre fixé de pages ont été visitées). Le résultat renvoyé par le crawler, que l'on définira plus précisément plus loin, est appelé un *crawl*.

☐ 5 – Coder `crawler_bfs : int -> string -> (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_bfs` parcoure le graphe du Web en suivant une stratégie *en largeur d'abord* (*breadth-first search*), c'est-à-dire en visitant en priorité les pages rencontrées le plus tôt dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type `dictionnaire` pour se souvenir des pages déjà visitées.

Par exemple, sur le mini-graphe, `crawler_bfs 4 "p1"` pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p5", ["p5"];
 "p4", ["p3"; "p5"]]
```

☐ 6 – Coder `crawler_dfs : int -> string -> (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page

visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_dfs` parcoure le graphe du Web en suivant une stratégie *en profondeur d'abord* (*depth-first search*), c'est-à-dire en visitant en priorité les pages rencontrées le plus récemment dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type `dictionnaire` pour se souvenir des pages déjà visitées.

Par exemple, sur le mini-graphe, `crawler_dfs 4 "p1"` pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p4", ["p3"; "p5"];
 "p3", ["p5"; "p6"]]
```

☐ 7 – Coder une fonction Caml `construit_graphe` :
`(string * string list) list -> string list * int vect vect`
telle que si *crawl* est le résultat renvoyé par un crawler (une liste de couples formés d'une URL v et de la liste des liens récupérés sur la page v), alors `construit_graphe crawl` est un couple (l, G) où l est une liste de toutes les URL de pages contenues dans la liste *crawl* et G est la matrice d'adjacence du sous-graphe partiel du Web restreint aux pages de la liste l : G_{ij} est le nombre de liens découverts dans le crawl de la page d'indice i dans l vers la page d'indice j dans l. On fera commencer les indices à 0. Pour coder la fonction `construit_graphe`, on pourra utiliser les fonctions `aplatir` et `unique`.

Par exemple, sur le mini-graphe, si `crawl` est une variable contenant le résultat de l'appel `crawler_bfs 4 "p1"` (voir question 5), alors `construit_graphe crawl` doit renvoyer :

```
["p1"; "p2"; "p5"; "p4"; "p3"],
[|[|0; 1; 1; 0; 0|];
  [|1; 0; 0; 1; 0|];
  [|0; 0; 1; 0; 0|];
  [|0; 0; 1; 0; 1|];
  [|0; 0; 0; 0; 0|]|]
```

En particulier :

— p3 apparaît même s'il n'a pas été visité dans le crawl ;

— p6 n'apparaît pas car il n'a pas été découvert dans le crawl ;

— l'hyperlien de p3 à p5 n'apparaît pas car p3 n'a pas été visité.

Calcul de PageRank

PageRank est une manière d'affecter un score à l'ensemble des pages du Web, imaginée par Sergey Brin et Larry Page, les fondateurs du moteur de recherche Google. L'introduction de PageRank a révolutionné la technologie des moteurs de recherche sur le Web. Nous allons maintenant implémenter le calcul de PageRank.

Étant donnée une partie du Web (où l'ensemble des pages est indexé entre 0 et $n-1$), la *matrice de surf aléatoire* dans cette partie du Web est la matrice M de taille $n \times n$ définie comme suit :

— S'il n'y a aucun lien depuis une page Web d'indice i, alors pour tout j, $M_{ij} \coloneqq 1/n$.

— Sinon, s'il y a k_i liens depuis la page Web d'indice i, alors pour tout j, on a $M_{ij} \coloneqq (1-d) \times G_{ij}/k_i + d/n$, où G_{ij} est le nombre de liens depuis la page d'indice i vers la page d'indice j et d est un nombre réel fixé appartenant à $[0,1]$ (on prend souvent $d = 0{,}15$).

Cette matrice peut être vue comme décrivant la *marche aléatoire* d'un surfeur sur le Web. À chaque fois que celui-ci visite une page Web :

— Si cette page ne comporte aucun lien, il visite une page Web arbitraire, choisie aléatoirement de façon uniforme.

— Si cette page comporte au moins un lien, il visite avec une probabilité égale à $1-d$ un des liens sortants de cette page, et avec une probabilité égale à d une page Web arbitraire, choisie aléatoirement de façon uniforme.

☐ 8 – Coder `surf_aleatoire : float -> int vect vect -> float vect vect` telle que si d est un nombre entre 0 et 1, et si G est la matrice d'adjacence d'un sous-graphe partiel du Web, alors `surf_aleatoire d G` renvoie la matrice M de surf aléatoire dans ce sous-graphe.

☐ 9 – Coder `multiplie : float vect -> float vect vect -> float vect`, une fonction prenant en argument un vecteur ligne v de taille n et une matrice M de taille $n \times n$ et renvoyant le vecteur ligne w de taille n résultant du produit de v par la matrice M : $w = vM$. En d'autres termes, pour tout j, $w_j = \sum_i v_i M_{ij}$.

Le PageRank des pages d'un sous-graphe du Web à n pages se calcule par des multiplications successives d'un vecteur ligne par la matrice de surf aléatoire M de ce sous-graphe. Plus précisément, soit θ un nombre réel strictement positif (par exemple, $\theta = 10^{-4}$) et soit $v^{(0)}$ le vecteur ligne de taille n dont toutes les composantes valent $1/n$. On pose pour un entier naturel p arbitraire $v^{(p)} \coloneqq v^{(0)} M^p$. L'algorithme de PageRank calcule la suite des $v^{(p)}$ pour $p = 0, 1, \ldots$ jusqu'à ce que $\|v^{(p+1)} - v^{(p)}\|_1 \leqslant \theta$ et renvoie alors le vecteur $v^{(p+1)}$, considéré comme le vecteur des scores de PageRank. On peut montrer (à l'aide du théorème de Perron–Frobenius) que l'algorithme termine dès lors que d est strictement positif.

PageRank est utilisé pour affecter un score d'*importance* aux pages du Web. Le vecteur de scores v retourné par l'algorithme de PageRank donne dans v_i le score d'importance

de la page d'indice i. Les pages de plus haut score de PageRank sont considérées comme les plus importantes.

☐ 10 – Coder `pagerank : float -> float vect vect -> float vect`, une fonction prenant en argument un nombre $\theta > 0$ et une matrice M de surf aléatoire d'un sous-graphe du Web et renvoyant le vecteur des scores de PageRank pour θ et M. La fonction `pagerank` devra faire appel à la fonction `multiplie` précédemment codée.

☐ 11 – Coder `calcule_pagerank :`
`float -> float -> (string * string list) list -> (string * float) list` telle que `calcule_pagerank d theta crawl` renvoie une liste de couples (u, s), un couple pour chaque URL découverte dans le crawl *crawl*, triée par valeur décroissante de s, où u est l'URL de cette page et s son score de PageRank. Ici, d et θ sont les deux paramètres nécessaires au calcul de la matrice de surf aléatoire et du PageRank respectivement. On pourra faire appel à la fonction `tri_fusion` et à l'ensemble des fonctions développées dans les questions précédentes.

2 Automates probabilistes

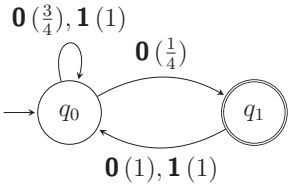
On fixe dans cet exercice un alphabet $\Sigma = \{\mathbf{0}, \mathbf{1}\}$.

Un *automate probabiliste* sur l'alphabet Σ est un quadruplet $\mathcal{A} = (Q, q_0, F, \Pr)$ où :

(i) Q est un ensemble fini non vide dont les éléments sont appelés *états* ;
(ii) $q_0 \in Q$ est appelé *état initial* ;
(iii) $F \subseteq Q$ est un ensemble dont les éléments sont appelés *états finals* ;
(iv) $\Pr : Q \times \Sigma \times Q \to [0, 1]$ est une application appelée *fonction probabiliste de transition* ; on suppose que pour tout $q \in Q$, pour tout $\alpha \in \Sigma$, $\sum_{q' \in Q} \Pr(q, \alpha, q') = 1$. On note $\Pr(q \xrightarrow{\alpha} q')$ pour $\Pr(q, \alpha, q')$.

Une *transition* est un triplet $(q, \alpha, q') \in Q \times \Sigma \times Q$, noté $q \xrightarrow{\alpha} q'$, avec $\Pr(q \xrightarrow{\alpha} q') > 0$. On représente un automate probabiliste de manière graphique, de façon similaire à la représentation des automates non-déterministes classiques : les états sont représentés par des cercles, l'état initial par une flèche arrivant sur le cercle correspondant, les états finals par des cercles doubles. La fonction probabiliste de transition est représentée par une flèche entre états : si $\Pr(q \xrightarrow{\alpha} q')$ est un nombre $p > 0$, on met une flèche de l'état q à l'état q', annotée par « $\alpha\,(p)$ ».

Ainsi, l'automate $\mathcal{A}_0 = (\{q_0, q_1\}, q_0, \{q_1\}, \Pr)$ représenté ci-dessous :



a pour fonction probabiliste de transition Pr la fonction suivante (seules les valeurs non nulles sont mentionnées) :

q	α	q'	$\Pr(q \xrightarrow{\alpha} q')$
q_0	**0**	q_0	$3/4$
q_0	**0**	q_1	$1/4$
q_0	**1**	q_0	1
q_1	**0**	q_0	1
q_1	**1**	q_0	1

Étant donné un automate probabiliste $\mathcal{A} = (Q, q_0, F, \Pr)$ sur Σ, un *chemin* ρ est une suite finie de transitions $q_{i_1} \xrightarrow{\alpha_1} q_{i_2}, \ldots, q_{i_n} \xrightarrow{\alpha_n} q_{i_{n+1}}$ aussi notée $q_{i_1} \xrightarrow{\alpha_1} q_{i_2} \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} q_{i_{n+1}}$; on dit que ρ a pour *étiquette* le mot $\alpha_1 \ldots \alpha_n \in \Sigma^*$, pour *état de départ* l'état $q_{i_1} \in Q$ et pour *état d'arrivée* l'état $q_{i_{n+1}} \in Q$. La *probabilité* de ρ, notée $\Pr(\rho)$, est définie par $\Pr(\rho) \coloneqq \prod_{k=1}^{n} \Pr(q_{i_k} \xrightarrow{\alpha_k} q_{i_{k+1}})$. Un état peut être vu comme un chemin de longueur nulle ; la probabilité d'un chemin de longueur nulle est égale à 1. Un *chemin pour le mot* $u \in \Sigma^*$ est un chemin dont l'étiquette est u et l'état de départ est q_0. Ce chemin est *acceptant* si l'état d'arrivée est un état de F, *non-acceptant* sinon. La *probabilité d'un mot* $u \in \Sigma^*$, notée $\Pr(u)$, est par définition la somme des probabilités de tous les chemins acceptants pour le mot $u \in \Sigma^*$:

$$\Pr(u) \coloneqq \sum_{\rho \text{ chemin acceptant pour } u} \Pr(\rho).$$

☐ 12 – Calculer les probabilités $\Pr(\varepsilon)$, $\Pr(\mathbf{0})$, $\Pr(\mathbf{010})$ pour l'automate $\mathcal{A}_0$ (ici, ε est le mot vide).

☐ 13 – Montrer, pour tout automate probabiliste $\mathcal{A} = (Q, q_0, F, \Pr)$ et tout mot $u \in \Sigma^*$, l'égalité suivante en utilisant une récurrence sur la longueur du mot u :

$$\Pr(u) = 1 - \sum_{\rho \text{ chemin non-acceptant pour } u} \Pr(\rho).$$

☐ 14 – On revient à l'automate probabiliste $\mathcal{A}_0$. Quels sont les mots u dont la probabilité $\Pr(u)$ pour $\mathcal{A}_0$ est égale à 0 ? Quels sont ceux dont la probabilité est égale à 1 ?

☐ 15 – Proposer (sans justification) une expression rationnelle pour le langage des mots u dont la probabilité $\Pr(u)$ pour $\mathcal{A}_0$ est non nulle.

☐ 16 – Montrer que pour tout automate probabiliste $\mathcal{A} = (Q, q_0, F, \Pr)$, il existe un automate non nécessairement déterministe $\mathcal{A}'$ qui accepte exactement les mots u dont la probabilité $\Pr(u)$ pour $\mathcal{A}_0$ est non nulle.

☐ 17 – Appliquer la construction de la question précédente à l'automate $\mathcal{A}_0$ pour obtenir un automate non-déterministe qui accepte exactement les mots u dont la probabilité $\Pr(u)$ pour $\mathcal{A}_0$ est non nulle. Déterminiser cet automate.

Soit $\mathcal{A} = (Q, q_0, F, \Pr)$ un automate probabiliste sur Σ. Pour un réel $\eta \in [0, 1[$, le *η-langage reconnu par* $\mathcal{A}$, noté $\mathcal{L}_\eta(\mathcal{A})$, est défini par :

$$\mathcal{L}_\eta(\mathcal{A}) \coloneqq \{\, u \in \Sigma^* \mid \Pr(u) > \eta \,\}.$$

On dit qu'un langage $L \subseteq \Sigma^*$ est *stochastique* s'il existe un automate probabiliste $\mathcal{A}$ et un réel $\eta \in [0, 1[$ tel que $L = \mathcal{L}_\eta(A)$.

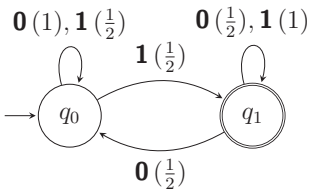
☐ 18 – Démontrer que tout langage rationnel est stochastique.

Étant donné un mot $\alpha_1 \dots \alpha_n$ sur l'alphabet $\{\mathbf{0}, \mathbf{1}\}$, on dit que l'expression « $\underline{0, \alpha_1 \dots \alpha_n}_2$ » est une *écriture (finie) en base 2* du nombre réel $\sum_{i=1}^{n} 2^{-i}\alpha_i$. On note alors :

$$\sum_{i=1}^{n} 2^{-i}\alpha_i = \underline{0, \alpha_1\alpha_2 \dots \alpha_n}_2.$$

Ainsi, $\frac{1}{4} = 2^{-2} = \underline{0, \mathbf{01}}_2 = \underline{0, \mathbf{0100}}_2$.

On considère maintenant l'automate $\mathcal{A}_1 = (\{q_0, q_1\}, q_0, \{q_1\}, \Pr)$ ci-dessous :



☐ 19 – Dans l'automate $\mathcal{A}_1$, calculer $\Pr(q_0 \xrightarrow{\mathbf{1}} q_0 \xrightarrow{\mathbf{1}} q_1 \xrightarrow{\mathbf{1}} q_1 \xrightarrow{\mathbf{0}} q_0 \xrightarrow{\mathbf{1}} q_1)$ et en donner une écriture finie en base 2.

☐ 20 – Dans l'automate $\mathcal{A}_1$, calculer $\Pr(\mathbf{10})$ et en donner une écriture finie en base 2.

☐ 21 – Dans l'automate $\mathcal{A}_1$, calculer $\Pr(\mathbf{1101})$ et en donner une écriture finie en base 2.

☐ 22 – Soit $u \in \Sigma^*$ un mot arbitraire sur Σ. Montrer que $\Pr(u)$ pour $\mathcal{A}_1$ admet une écriture finie en base 2, et en donner une expression. Prouver que cette écriture est correcte.

☐ 23 – Soit $\eta \in [0, 1[$. Prouver l'égalité suivante :

$$\mathcal{L}_\eta(\mathcal{A}_1) = \{\, \alpha_1 \dots \alpha_n \in \Sigma^* \mid \underline{0, \alpha_n \dots \alpha_1}_2 > \eta \,\}.$$

☐ 24 – En déduire qu'il existe des langages stochastiques qui ne sont pas rationnels.

Fin de l'épreuve