Composition d'informatique n°5

(Durée : 4 heures)

L'utilisation de la calculatrice **n'est pas autorisée** pour cette épreuve.

⋆ ⋆ ⋆

Complexité de Kolmogorov et plus petite grammaire

La complexité de Kolmogorov est une grandeur permettant de représenter l'entropie d'une donnée. Elle correspond à la taille du plus petit algorithme permettant de créer cette donnée. Dans ce sujet, on étudie la complexité de Kolmogorov, d'abord dans un modèle de calcul complet (le langage OCaml) dans la première partie, puis dans un modèle de calcul plus restreint, à savoir celui des grammaires hors-contexte, dans la deuxième partie. La troisième et dernière partie s'intéresse à la difficulté d'estimer cette complexité pour les grammaires, puis propose un algorithme d'approximation basé sur un algorithme de compression.

Consignes

Les questions de programmation doivent être traitées en langage OCaml. On autorisera toutes les fonctions des modules `Array`, `List`, `String` et `Char`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). Sauf précision de l'énoncé, l'utilisation d'autres modules sera interdite.

Une annexe présente en fin de sujet rappelle l'utilisation de certaines fonctions et opérations sur les caractères et chaînes de caractères et les tables de hachage.

1 Complexité de Kolmogorov

On note Σ l'ensemble ordonné des 256 caractères ASCII étendus usuels et Σ^* l'ensemble des chaînes de caractères.

Dans cette partie, on s'appuie sur une **machine universelle**, c'est-à-dire une fonction OCaml `eval` de type `string -> string` telle que :

- si la chaîne `u`, de type `string`, est le code source d'une expression OCaml `v` de type `string`, et si l'exécution du code `u` se termine sans erreur, alors `eval u` se termine sans erreur et a pour valeur de retour la valeur de `v` ;
- sinon, `eval u` ne termine pas.

Par exemple, l'appel suivant termine et renvoie `"aaaaaa"` :

```
eval "let n = 2 * 3 in String.make n 'a'"
```

On suppose que les exécutions de `eval` ont lieu sur une machine idéale dont la mémoire est infinie et qui est capable de gérer des types natifs de taille quelconque.

Pour toute chaîne de caractères $v \in \Sigma^*$, on dit que la chaîne de caractères $u \in \Sigma^*$ est une **description de** v si le calcul `eval u` termine et renvoie v. On appelle **complexité de Kolmogorov de** v, notée $K(v)$, la longueur de la plus courte description de v.

1.1 Un premier exemple

On considère la chaîne de caractères v_0 :

$$v_0 = \texttt{"10000..."} \in \Sigma^*$$

correspondant à la description en base 10 de l'entier $10^{10^{10}}$.

Question 1 Proposer une première majoration de $K(v_0)$ basée sur la description $u_0 = v_0$ de v_0.

Question 2 Soit $n \in \mathbb{N}$ et $m = \left\lfloor \frac{n}{2} \right\rfloor$. Exprimer la quantité 10^n en fonction de 10^m. Compléter le code OCaml suivant, en utilisant une stratégie « Diviser pour régner », afin d'obtenir une nouvelle description de v_0 et une nouvelle majoration de $K(v_0)$, à 100 près.

```
let (* À compléter *)
in
string_of_int (exp10 (exp10 10))
```

Question 3 Décrire quelle ou quelles difficultés seraient rencontrées si on exécutait le code de la question précédente sur une machine réelle.

1.2 Calculabilité

On se donne une bijection $\varphi : \mathbb{N} \to \Sigma^*$ entre les entiers naturels et l'ensemble des chaînes de caractères. On suppose que la fonction φ est implémentée par une fonction `phi (n : int) : string`.

Question 4 On considère la fonction suivante :

```
let rec phi = function
    | 0 -> ""
    | n ->
        let c = Char.chr (n mod 256) in
        String.make 1 c ^ phi (n / 256)
```

Cette fonction est-elle une implémentation acceptable de la fonction φ ? Justifier.

Question 5 Justifier que la fonction K est non bornée.

On suppose, pour les deux questions suivantes, qu'il existe une fonction `kolmogorov (v : string) : int` qui calcule la complexité de Kolmogorov $K(v)$.

Question 6 Écrire une fonction `psi (m : int) : int` qui calcule l'entier

$$\psi(m) = \min\{n \in \mathbb{N} \mid K(\varphi(n)) \geqslant m\}$$

où φ est la bijection définie en début de partie.

Question 7 Montrer que $K(\varphi(\psi(m))) \geqslant m$. En établissant une borne supérieure bien choisie de $K(\varphi(\psi(m)))$, discuter de l'existence de la fonction `kolmogorov`.

2 Complexité grammaticale

2.1 Grammaires hors-contextes

On appelle **grammaire hors-contexte** un quadruplet $G = (T, V, P, S)$ où :

- T est un alphabet dont les éléments sont appelés **terminaux** ;
- V est un alphabet disjoint de T dont les éléments sont appelés **variables** ou **non terminaux** ;
- $P \subseteq V \times (T \cup V)^*$ est un ensemble **fini** de **règles de productions**. Une règle de production (X, α) sera notée $X \to \alpha$. S'il existe deux règles $X \to \alpha$ et $X \to \beta$, on pourra utiliser la notation plus concise $X \to \alpha \mid \beta$;
- $S \in V$ est le **symbole de départ**.

Si $\alpha, \gamma \in (T \cup V)^*$ et $X \to \beta \in P$, on dit que $\alpha X \gamma$ se **dérive immédiatement** en $\alpha\beta\gamma$, noté $\alpha X \gamma \Rightarrow \alpha\beta\gamma$.

Pour $n \in \mathbb{N}$, on écrit $\alpha \Rightarrow^n \beta$ s'il existe une suite de n dérivations immédiates partant de α et terminant en β. On écrit $\alpha \Rightarrow^* \beta$ s'il existe $n \in \mathbb{N}$ tel que $\alpha \Rightarrow^n \beta$. On dit alors que α se **dérive** en β.

Enfin, on appelle **langage engendré** par G, noté $L(G)$, l'ensemble

$$L(G) = \{u \in T^* \mid S \Rightarrow^* u\}$$

Par exemple, si la grammaire G_0 est définie par $T = \{a, b\}$, $V = \{S, X\}$ et P contient les règles :

$$S \to XXS \mid \varepsilon$$
$$X \to a \mid b$$

Alors $L(G_0) = \{u \in \{a, b\}^* \mid |u| \equiv 0 \pmod 2\}$.

Question 8 Déterminer, en justifiant, le langage engendré par la grammaire G_1 définie par $T = \{a, b\}$, $V = \{S\}$ et P défini par :

$$S \to aSa \mid bSb \mid \varepsilon$$

Question 9 Le langage engendré par la grammaire G_1 de la question précédente est-il rationnel ? Justifier.

On dit qu'une grammaire $G = (T, V, P, S)$ est **acyclique** si et seulement si, pour tout $X \in V$, si $X \Rightarrow^k \alpha$ avec $k > 0$, alors $|\alpha|_X = 0$. Autrement dit, une dérivation non vide de X ne peut pas faire apparaître la variable X.

Pour une grammaire $G = (T, V, P, S)$, on appelle **ordre topologique** de V une énumération de V en $(X_0, X_1, \ldots, X_{n-1})$ telle que pour toute règle $X_i \to \alpha \in P$, si $|\alpha|_{X_j} > 0$, alors $i < j$. Autrement dit, une règle depuis X_i ne peut pas faire apparaître de X_j avec $j \leqslant i$.

Question 10 Montrer qu'une grammaire est acyclique si et seulement si elle possède un ordre topologique.

Question 11 En déduire que si G est une grammaire acyclique, alors $L(G)$ est fini.

Pour implémenter une grammaire $G = (T, V, P, S)$ en OCaml, on suppose que $T = \Sigma$ et que $V = [\![0, n-1]\!]$. On représente un élément de $T \cup V$ par le type `symbole` suivant, tel que `T c` correspond au caractère c et `V i` correspond à la variable d'indice i. Un mot de $(T \cup V)^*$ est alors une suite de symboles.

```
type symbole = T of char | V of int

type mot = symbole list
```

On représente une grammaire comme un tableau de listes de mots

```
type grammaire = mot list array
```

tel que si `g` est un objet de type `grammaire` représentant G, alors `g` est un tableau de taille n tel que pour $X \in V$, `g.(x)` est la liste des mots α tels que $X \to \alpha \in P$. La variable S correspondra toujours à la variable d'indice 0.

Par exemple, la grammaire G_0 donnée en exemple précédemment pourrait être définie par :

```
let g0 = [|[ [V 1; V 1; V 0]; [] ];
           [ [T 'a']; [T 'b'] ]|]
```

Question 12 Écrire une fonction `acyclique (g : grammaire) : bool` qui détermine si une grammaire est acyclique.

Question 13 Déterminer la complexité temporelle de la fonction `acyclique` en fonction de $|V|$, $|P|$ et de la taille maximale d'un membre droit dans une règle de production.

On dit qu'une grammaire $G = (T, V, P, S)$ est **élémentaire** si et seulement si, pour tout $X \in V$, l'ensemble $(X \times (T \cup V)^*) \cap P$ est de cardinal 1. Autrement dit, il n'existe qu'une seule règle de production ayant X comme membre gauche.

Question 14 Une grammaire élémentaire peut-elle être ambiguë ? Justifier. Même question pour une grammaire acyclique.

On dit qu'une grammaire est **accessible** si pour tout $X \in V$, il existe une dérivation $S \Rightarrow^* \alpha$ avec $|\alpha|_X > 0$.

Question 15 Soit G une grammaire élémentaire et accessible. Montrer que $|L(G)| = 1$ si et seulement si G est acyclique.

Pour la suite, les grammaires manipulées dans les fonctions seront élémentaires. Pour simplifier l'écriture des fonctions, on utilisera le nouveau type `grammaire` défini par :

```
type grammaire = mot array
```

tel que si `g` est de type `grammaire` et représente $G = (T, V, P, S)$, alors pour $X \in V$, `g.(x)` est égal au mot α tel que $X \to \alpha$ est l'unique règle de production ayant X comme membre gauche.

Question 16 Écrire une fonction `ordre_topologique (g : grammaire) : int list` qui prend en argument une grammaire $G = (T, V, P, S)$ supposée acyclique et élémentaire et renvoie un ordre topologique $(X_0, \ldots, X_{n-1})$ de V sous la forme d'une liste.

Question 17 Écrire une fonction `engendre (g : grammaire) : string` qui prend en argument une grammaire supposée acyclique et élémentaire et renvoie le mot engendré par cette grammaire.

Question 18 En notant v le mot engendré par la grammaire, déterminer la complexité temporelle de la fonction `engendre` en fonction de $|v|$, $|V|$ et de la taille maximale m d'un membre droit d'une règle de production.

2.2 Bornes supérieures et inférieures

Pour $G = (T, V, P, S)$ une grammaire hors-contexte, on appelle **taille de** G, notée $|G|$, la somme des tailles des membres droits de toutes les règles de production :

$$|G| = \sum_{X \to \alpha \in P} |\alpha|$$

Pour $v \in \Sigma^*$ une chaîne de caractères, on appelle **complexité grammaticale de Kolmogorov** de v, notée $K_G(v)$, la plus petite taille d'une grammaire engendrant le langage $\{v\}$.

Question 19 Soit v une chaîne de caractères. Montrer qu'il existe une grammaire acyclique, élémentaire et accessible G telle que $L(G) = \{v\}$ et $|G| = K_G(v)$.

Question 20 Montrer que pour $v, w \in \Sigma^*$, $K_G(vw) \leqslant K_G(v) + K_G(w) + 2$.

Question 21 Montrer que pour $v \in \Sigma^*$ et $k \in \mathbb{N}$, $K_G(v^k) \leqslant K_G(v) + \mathcal{O}(\log k)$.

Pour toute la suite du sujet, on ne considère que des grammaires acycliques, élémentaires et accessibles.

Pour $X \in V$, on note $g(X)$ le mot engendré par la grammaire (T, V, P, X) (c'est-à-dire en supposant que X est le nouveau symbole de départ). On suppose pour les deux questions suivantes que $V = \{X_0, \ldots, X_{n-1}\}$ et que $P = \{X_i \to \alpha_i \mid i \in [\![0, n-1]\!]\}$.

Question 22 Montrer que si $\alpha_i \notin T^*$, alors il existe j tel que $|\alpha_i|_{X_j} > 0$ et $|g(X_i)| \leqslant |\alpha_i| \times \max(|g(X_j)|, 1)$.

On admet que si la somme de n entiers est majorée par M, alors le produit de ces n entiers est majoré par $3^{M/3}$.

Question 23 Montrer que $|g(S)| \leqslant \prod_{i=0}^{n-1} \max(1, |\alpha_i|)$. En déduire que pour toute chaîne v, $K_G(v) = \Omega(\log |v|)$.

On rappelle que $f = \Omega(g)$ si et seulement si $g = \mathcal{O}(f)$.

3 Plus petite grammaire

3.1 Un problème difficile

On s'intéresse au problème de décision `Plus Petite Grammaire (PPG)` :

* **Instance** : un mot $v \in \Sigma^*$, où Σ est un alphabet quelconque, et un entier $k \in \mathbb{N}$.
* **Question** : est-ce que $K_G(v) \leqslant k$?

On souhaite montrer que ce problème est NP-complet, par une réduction depuis le problème `COUV` :

* **Instance** : un graphe [1] $\Gamma = (S, A)$ et un entier $k \in \mathbb{N}$.
* **Question** : existe-t-il une couverture des arêtes par les sommets, c'est-à-dire un ensemble $C \subseteq S$ tel que pour $a \in A$, $a \cap C \neq \emptyset$, de cardinal $|C| \leqslant k$?

On admet que le problème `COUV` est NP-complet.

Question 24 Montrer que `PPG` $\in$ NP.

Étant donné un graphe $\Gamma = (S, A)$, avec $S = \{s_1, \ldots, s_p\}$ et $A = \{a_1, \ldots, a_q\}$, et un entier $k \in \mathbb{N}$, on construit une instance du problème `PPG` :

– l'alphabet Σ est défini par :

$$\Sigma = S \cup A \cup \{\#\} \cup \bigcup_{i=1}^{p} \{a_i, b_i, c_i, d_i, e_i\}$$

autrement dit, chaque sommet et chaque arête est une lettre, et on rajoute une lettre spéciale # et cinq nouvelles lettres par sommet ;

1. On désigne un graphe par la lettre Γ au lieu de l'usuelle notation G pour éviter de confondre avec la notation pour les grammaires.

– pour $s = s_i \in S$, on définit $v_i = \#sa_is\#b_i\#sc_is\#d_i\#s\#e_i$;
– pour $a = a_j = \{s, t\} \in A$, on définit $w_j = \#s\#t\#a$;
– enfin, on pose $v = v_1v_2 \ldots v_pw_1w_2 \ldots w_q$.

Question 25 Montrer que s'il existe une couverture des arêtes par les sommets C de Γ de cardinal k, alors $K_G(v) \leqslant 15p + 3q + k$. On décrira la grammaire qui permet d'obtenir cette borne.

Pour la réciproque, on se donne une grammaire acyclique, élémentaire et accessible $G = (T, V, P, X_0)$, où $V = \{X_0, \ldots, X_{n-1}\}$, telle que $|G| = K_G(v)$. On suppose que $P = \{X_i \to \alpha_i \mid i \in [\![0, n-1]\!]\}$.

Question 26 Montrer qu'on peut supposer sans perte de généralité que, pour tout $i > 0$, $g(X_i)$ est de la forme $\#s$, $s\#$ ou $\#s\#$ pour un certain $s \in S$.

Question 27 Montrer qu'on peut supposer sans perte de généralité que, pour tout $s \in S$, il existe i et j tels que $g(X_i) = \#s$ et $g(X_j) = s\#$.

Question 28 En considérant $C = \{s \in S \mid \exists i \in [\![1, n-1]\!], g(X_i) = \#s\#\}$, montrer que si $K_G(v) \leqslant 15p + 3q + k$, alors Γ possède une couverture des arêtes par les sommets de cardinal k.

Question 29 En déduire que `PPG` est NP-complet.

3.2 Algorithme d'approximation

On considère dans cette partie le problème d'optimisation correspondant, auquel on donne le même nom : `Plus Petite Grammaire (PPG)` :

* **Instance** : un mot $v \in \Sigma^*$.
* **Solution** : une grammaire $G = (T, V, P, S)$ telle que $L(G) = \{v\}$.
* **Optimisation** : minimiser $|G|$.

Question 30 Appliquer l'algorithme de compression de Lempel-Ziv-Welch sur la chaîne $v = ababcabcdabcdabcda$, en supposant que la table initiale est :

a	b	c	d
0	1	2	3

On donnera la suite numérique obtenue pour l'encodage, ainsi que l'état final de la table.

Question 31 Appliquer l'algorithme de décompression de Lempel-Ziv-Welch sur la suite $x = (0, 4, 1, 5, 6, 2, 3, 8)$, toujours en supposant que la table initiale est :

a	b	c	d
0	1	2	3

On donnera le mot décompressé, ainsi que l'état final de la table.

L'algorithme LZ78 (Lempel-Ziv 1978) est un prédécesseur de l'algorithme LZW (Lempel-Ziv-Welch). Comme son successeur, il permet de la compression et décompression en utilisant une table d'encodage, mais la particularité est que cette table est initialement vide. L'idée est la suivante :

Entrée : un mot $v \in \Sigma^*$.
Début algorithme
  $D \leftarrow$ dictionnaire contenant uniquement l'association $(\varepsilon, 0)$
  $x \leftarrow$ liste vide
  $k \leftarrow 1$
  Tant que $v \neq \varepsilon$ **Faire**
    $wa \leftarrow$ plus long préfixe de v tel que $w \in D$ et $a \in \Sigma$
    Ajouter le couple $(D[w], a)$ à x
    Ajouter l'association (wa, k) à D
    $k \leftarrow k + 1$
    $v \leftarrow$ le mot v privé de son préfixe wa
  Renvoyer x

Par exemple, appliquer l'algorithme à la chaîne $v = ababcabcdabcda$ permet d'obtenir le code de compression $x = ((0,a),(0,b),(1,b),(0,c),(3,c),(0,d),(5,d),(0,a))$, qui correspondrait au découpage de v de la forme $v = (a)(b)(ab)(c)(abc)(d)(abcd)(a)$, et l'état final de la table serait :

ε	a	b	ab	c	abc	d	$abcd$	$abcda$
0	1	2	3	4	5	6	7	8

On rappelle que l'annexe contient des indications sur l'utilisation des tables de hachage.

Question 32 Écrire une fonction `compresse_lz78 : string -> (int * char) list` qui prend en argument une chaîne v et renvoie la liste correspondant à l'encodage de v selon l'algorithme LZ78.

On applique l'algorithme LZ78 sur une chaîne $v \in \Sigma^*$ et on obtient un code $x = ((k_1, a_1), (k_2, a_2), \ldots, (k_n, a_n))$. À partir de ce code, on définit la grammaire $G_x = (T, V, P, S)$ par :

- $T = \Sigma$;
- $V = \{S, X_0, X_1, X_2, \ldots, X_n\}$;
- $P = \{X_0 \to \varepsilon, S \to X_1 X_2 \ldots X_n\} \cup \{X_i \to X_{k_i} a_i \mid i \in [\![1, n]\!]\}$.

Question 33 Montrer que $L(G_x) = \{v\}$.

Question 34 En déduire une fonction `decompresse_lz78 : (int * char) list -> string` qui prend en argument un code x obtenu par l'algorithme de compression LZ78 et renvoie la chaîne initiale.

On cherche à montrer que la construction de cette grammaire fournit un algorithme d'approximation pour le problème d'optimisation `PPG`.

Question 35 Soit $G = (T, V, P, S)$ une grammaire et v le mot engendré par G. Montrer que pour $k \in \mathbb{N}^*$, v contient au plus $|G| \times k$ facteurs de taille k distincts.

On pourra, pour chaque règle $X \to \alpha$, majorer le nombre de facteurs de $g(X)$ qui ne sont pas facteur d'un $g(Y)$ avec Y apparaissant dans α.

On considère la grammaire G_x définie précédemment à l'aide de l'algorithme LZ78, engendrant le mot v. On pose $\kappa = K_G(v)$, et on renomme les variables $X_1, \ldots, X_n$ en $Y_1, \ldots, Y_n$ de telle sorte que $|g(Y_1)| \leqslant |g(Y_2)| \leqslant \ldots \leqslant |g(Y_n)|$.

On regroupe les (Y_i) par paquets de plus en plus grands (sauf éventuellement le dernier) :

- le premier paquet, de taille κ, contient $Y_1, \ldots, Y_\kappa$;
- le deuxième paquet, de taille 2κ, contient $Y_{\kappa+1}, \ldots, Y_{3\kappa}$;
- le troisième paquet, de taille 3κ, ...
- ...

– l'avant-dernier paquet est de taille $p \times \kappa$;
– le dernier paquet, éventuellement vide, de taille strictement inférieure à $(p+1)\kappa$, contient les variables restantes.

Par définition de p et des groupements par paquets, on a :

$$\kappa + 2\kappa + \ldots + p\kappa + (p+1)\kappa > n$$

Question 36 Montrer que $|v| \geqslant 1^2\kappa + 2^2\kappa + 3^2\kappa + \ldots + p^2\kappa$.

Question 37 En déduire que l'algorithme Lempel-Ziv 78 donne une approximation de `PPG` à un facteur $\mathcal{O}\left(\left(\frac{|v|}{\log|v|}\right)^{2/3}\right)$.

Le paradoxe de Berry demande de trouver « le plus petit entier qui ne peut pas être décrit en strictement moins de dix-sept mots. » On peut faire le lien avec la calculabilité de la complexité de Kolmogorov.

Annexe : rappels en OCaml

On rappelle les fonctions et opérations suivantes :

– sur les chaînes de caractères :
  * `string_of_int : int -> string` convertit un entier en une chaîne de caractères ;
  * `String.make : int -> char -> string` prend en argument un entier n et un caractère c et renvoie une chaîne de caractères de taille n ne contenant que le caractère c. Cette fonction a une complexité linéaire en n ;
  * `String.sub : string -> int -> int -> string` prend en argument une chaîne v et deux entiers i et n et renvoie la sous-chaîne de v de taille n, commençant à l'indice i ;
  * `Char.chr : int -> char` prend en argument un entier entre 0 et 255 et renvoie le caractère ASCII portant ce numéro ;
  * si `u` et `v` sont deux chaînes de caractères, alors `u ^ v` est la chaîne de caractères correspondant à la concaténation uv. Cette fonction a une complexité linéaire en $|uv|$.
– sur les tables de hachage : un objet de type `('a, 'b) Hashtbl.t` correspond à une table de hachage contenant des associations (clé, valeur) où les clés sont de type `'a` et les valeurs de type `'b`.
  * `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` prend en argument un entier m et crée une table vide occupant un espace mémoire de taille proportionnelle à m (on pourra choisir $m = 1$ en pratique) ;
  * `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` prend en argument une table, une clé et une valeur et rajoute une association (clé, valeur) dans la table de hachage ;
  * `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` teste si une table contient une clé donnée ;
  * `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` prend en argument une table et une clé et renvoie la valeur associée.

⋆ ⋆ ⋆