Composition d'informatique n°5

Corrigé

⋆ ⋆ ⋆

1 Complexité de Kolmogorov

Question 1 La chaîne v_0 est constituée d'un 1 suivi de 10^{10} zéros. On en déduit que $K(v_0) \leqslant 1 + 10^{10}$.

Question 2 On a :

- $10^n = (10^m)^2$ si n est pair ;
- $10^n = (10^m)^2 \times 10$ sinon.

On en déduit la fonction `exp10` pour calculer 10^n et le code suivant :

```
let rec exp10 = function
    | 0 -> 1
    | n ->
        let x = exp10 (n / 2) in
        if n mod 2 = 0 then x * x
        else x * x * 10
in
string_of_int (exp10 (exp10 10))
```

On en déduit donc que $K(v_0) \leqslant 200$.

Question 3 On peut mentionner les problèmes suivants (certains n'en sont en fait pas) :

- les entiers étant représentés sur un nombre fini fixe de bits, usuellement 63 ou 31 en OCaml, le calcul de $10^{10^{10}}$ mènerait nécessairement à un dépassement d'entiers, donc des résultats incohérents (voire des erreurs si on essaie de créer une chaîne de taille négative) ;
- la fonction étant récursive, on peut se soucier du dépassement de la taille de la pile d'appels. Toutefois, le nombre d'appels récursifs est de l'ordre de $\log_2(10^{10}) = 10 \log_2 10 \leqslant 40$;
- on peut s'inquiéter du temps de calcul, mais celui-ci reste humainement accessible, car il y a de l'ordre de 40 multiplications d'entiers bornés.

En remarque : si on veut travailler avec un type spécial d'entiers non bornés, c'est le temps de calcul lié à la multiplication qui devient problématique.

Question 4 Cette fonction n'est pas bijective. Le problème ici est que la chaîne constituée uniquement du caractère d'indice 0 n'a pas d'antécédent, donc la fonction n'est pas surjective.

Il faut apporter quelques modifications pour obtenir l'unicité. On propose la transformation suivante (non demandée) :

- 0 est encodé en la chaîne vide ε ;
- un entier $n > 0$ a pour premier caractère celui de code ASCII $(n-1) \mod 256$, et pour suffixe l'encodage de $\left\lfloor \frac{n-1}{256} \right\rfloor$.

On obtient alors :

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

Question 5 Si on suppose que K est borné par un entier M, alors toute chaîne de caractère possède une description de taille $\leqslant M$. Mais comme il existe moins de 256^{M+1} descriptions de taille $\leqslant M$, ces descriptions ne peuvent décrire qu'un nombre fini de chaîne, ce qui est absurde (car il existe une infinité de chaînes différentes).

Question 6 On écrit une simple boucle pour faire le calcul.

```
let psi m =
    let n = ref 0 in
    while kolmogorof (phi !n) < m do
        incr n
    done;
    !n
```

Question 7 Par définition, $K(\varphi(\psi(m))) \geqslant m$, car $\psi(m) \in \{n \in \mathbb{N} \mid K(\varphi(n)) \geqslant m\}$ (c'est son minimum). De plus, si on considère le code suivant :

```
let kolmogorov = ...  (* code de la fonction *)
let rec phi n = ...
let psi m = ...

let m = ...  (* écriture de m en base 10 *)
in
phi (psi m)
```

alors ce code est une description de $\varphi(\psi(m))$. Sa taille est donc un majorant de $K(\varphi(\psi(m)))$. Mais l'écriture des trois fonctions nécessite un nombre constant fixe de caractères. De plus, l'écriture de l'entier m en base 10 nécessite un nombre logarithmique en m de chiffres. On en déduit que $K(\varphi(\psi(m))) = \mathcal{O}(\log m)$.

Cela est contradictoire avec le fait que $K(\varphi(\psi(m))) \geqslant m$. On en déduit que la fonction `kolmogorov` ne peut pas exister.

2 Complexité grammaticale

Question 8 On a $L(G_1) = L_1 = \{v\tilde{v} \mid v \in \{a,b\}^*\}$, où $\tilde{v}$ désigne le mot-miroir de v.

Montrons par récurrence sur n que si $S \Rightarrow^n \alpha$, alors α est de la forme $v\tilde{v}$ ou $vS\tilde{v}$ pour un certain $v \in \{a,b\}^*$.

- si $n = 0$, alors $S \Rightarrow^0 S$ est la seule dérivation vide possible et $S = \varepsilon S \varepsilon$;
- supposons le résultat établi pour $n \in \mathbb{N}$ fixé. Si $S \Rightarrow^{n+1} \alpha$, alors $S \Rightarrow vS\tilde{v}$ (car si S n'apparait pas dans la dérivation de taille n, il n'existe pas de dérivation de taille $n+1$). Dès lors, les seules dérivations de taille $n+1$ possibles sont :
  - $S \Rightarrow^{n+1} vaSa\tilde{v}$, et $a\tilde{v} = \widetilde{va}$;
  - $S \Rightarrow^{n+1} vbSb\tilde{v}$, et $b\tilde{v} = \widetilde{vb}$;
  - $S \Rightarrow^{n+1} v\tilde{v}$.

  Le résultat est bien établi pour $n+1$.

On en déduit que si $S \Rightarrow^* u \in \{a,b\}^*$, alors $u \in L_1$.

Réciproquement, si $u = a_1 \dots a_n a_n \dots a_1 \in L_1$, alors la dérivation suivante permet de montrer que $u \in L(G_1)$:

$$S \Rightarrow a_1 S a_1 \Rightarrow a_1 a_2 S a_2 a_1 \Rightarrow \dots \Rightarrow a_1 \dots a_n S a_n \dots a_1 \Rightarrow u$$

Question 9 Le langage $L(G_1)$ n'est pas rationnel. En effet, supposons qu'il le soit et soit n sa longueur de pompage. En posant $u = a^n bba^n \in L(G_1)$, par le lemme de pompage, $u = xyz$ avec :

- $|xy| \leqslant n$;
- $y \neq \varepsilon$;
- $\forall k \in \mathbb{N}, xy^k z \in L(G_1)$.

Mais d'après les deux premières conditions, $y = a^\ell$ avec $1 \leqslant \ell \leqslant n$, et alors $xz = xy^0 z = a^{n-\ell} bba^n \notin L(G_1)$. Par l'absurde, on en déduit que le langage n'est pas rationnel.

Question 10 Supposons que G n'est pas acyclique. Alors il existe $X \in V$ et $k > 0$ tel que $X \Rightarrow^k \alpha$, avec $|\alpha|_X > 0$. Cela signifie qu'il existe des variables $X = Y_0, Y_1, \dots, Y_m = X$ et des règles $Y_i \to \alpha_i Y_{i+1} \beta_i$, pour $i \in [\![0, m-1]\!]$. S'il existait un ordre topologique $(X_0, \dots, X_{n-1})$ et un indice i tel que $X = X_i$, alors par transitivité de $<$ et par définition de l'ordre topologique, on aurait $i < i$, ce qui est absurde.

Réciproquement, supposons que G est acyclique. Il existe donc une variable X qui n'apparaît dans aucun membre droit des règles de production. En effet, si ce n'est pas le cas, on peut construire une dérivation cyclique :

- on pose $Y_0 \in V$ quelconque ;
- pour $i \in \mathbb{N}$, on pose Y_{i+1} une variable telle qu'il existe une règle $Y_{i+1} \to \alpha_i$, avec Y_i qui apparaît dans α_i.

Le nombre de variables étant fini, on obtient bien un cycle, donc une contradiction.

Dès lors, on pose $X_0 = X$. En supprimant les règles de production ayant X_0 comme membre gauche, la grammaire résultante est toujours acyclique, et on peut réitérer le processus pour construire l'ordre topologique par récurrence.

En remarque : on aurait pu définir un graphe associé à la grammaire et vérifier que l'acyclicité du graphe est équivalente avec celle de la grammaire, et de même pour l'ordre topologique.

Question 11 G étant acyclique, elle possède un ordre topologique $(X_0, \dots, X_{n-1})$. Montrons par récurrence décroissante que $L(X_i)$ est fini, où $L(X_i) = L(T, V, P, X_i)$ (la grammaire avec les mêmes règles, où le symbole de départ est X_i) :

- $L(X_{n-1})$ est fini, car les règles ayant X_{n-1} comme membre gauche n'ont pas de variable dans les membres droits. $|L(X_{n-1})|$ est donc égal au nombre de règles ayant X_{n-1} comme membre gauche ;
- supposons le résultat vrai pour $0 < i \leqslant n-1$ fixé. Alors il y a un nombre fini de règles ayant X_{i-1} comme membre gauche, et que les variables apparaissant dans les membres droits ne peuvent engendrer qu'un nombre fini de mots, on en déduit que $L(X_{i-1})$ est fini.

On conclut par récurrence et on répond à la question en remarquant que $L(G) = L(S)$.

Question 12 Il s'agit plus ou moins d'un parcours de graphe, avec les variables comme états et les règles de productions donnant les arêtes. On garde en mémoire trois états possibles pour une variable : pas encore vue (0), en cours d'exploration des variables accessibles (1) et dont l'exploration a été terminée (2).

```
let acyclique g =
    let n = Array.length g in
    let etats = Array.make n 0 in
    let rec existe_cycle i =
        if etats.(i) = 1 then true
        else if etats.(i) = 2 then false
        else begin
            etats.(i) <- 1;
            let traiter_symb = function
                | T _ -> false
                | V j -> existe_cycle j
            in
            let traiter_mot alpha = List.exists traiter_symb alpha in
            let b = List.exists traiter_mot g.(i) in
            etats.(i) <- 2;
            b
        end
    in
    let cycle = ref false in
    for i = 0 to n - 1 do
        cycle := !cycle || existe_cycle i
    done;
    not !cycle
```

La fonction `existe_cycle` détermine s'il existe une dérivation cyclique qui peut être obtenue en commençant par une variable donnée. On l'exécute sur chaque variable pour trouver le résultat.

Question 13 La fonction ne parcourt qu'une seule fois chaque membre droit d'une règle de production. Comme la création du tableau `etats` nécessite un temps $\mathcal{O}(|V|)$, on en déduit une complexité en $\mathcal{O}(|V|+|P|\times(m+1))$ où m est la taille maximale d'un membre droit.

Question 14 Comme chaque variable ne peut être dérivée que d'une seule manière, on en déduit par récurrence que pour tout $k \in \mathbb{N}$, il n'existe qu'au plus une dérivation gauche de taille k de S. Par ailleurs, si une dérivation gauche de taille k donne un mot $v \in T^*$, il n'existe aucune dérivation de taille $> k$. On en déduit que pour tout mot $v \in L(G)$, il n'existe qu'au plus une dérivation gauche de v (on a même montré que $|L(G)| \leqslant 1$). Une grammaire élémentaire ne peut donc pas être ambiguë.

Une grammaire acyclique peut être ambiguë, par exemple $S \to X \mid \varepsilon$, $X \to \varepsilon$.

Question 15 On a montré à la question précédente qu'il n'existe qu'au plus une dérivation gauche de S de taille $k \in \mathbb{N}$. On peut montrer que le résultat reste vrai pour toute variable. Distinguons :

- s'il existe une variable X telle que $X \Rightarrow^k \alpha$, avec $k > 0$ et $|\alpha|_X > 0$, alors comme X est accessible, une dérivation de X (donc de S) contiendra toujours au moins une variable, donc $L(G) = \emptyset$;
- sinon, il n'existe pas de dérivation de taille arbitrairement grande (sinon il y aurait un cycle), donc il existe une dérivation gauche de S qui termine en un mot $v \in T^*$. On en déduit que $|L(G)| \geqslant 1$. Comme on l'a dit précédemment, on a également $|L(G)| \leqslant 1$.

Question 16 Cette fonction correspond à un calcul de parcours en profondeur postfixe d'un graphe orienté. Elle ressemble à la fonction de détection de cycle écrite précédemment.

```
let ordre_topologique g =
    let n = Array.length g in
    let vu = Array.make n false and
        topo = ref [] in
    let rec dfs i =
        if not vu.(i) then begin
            vu.(i) <- true;
            let traiter_symb = function
                | T _ -> ()
                | V j -> dfs j
            in
            List.iter traiter_symb g.(i);
            topo := i :: !topo
        end
    in
    for i = 0 to n - 1 do
        dfs i
    done;
    !topo
```

Question 17 L'idée est de calculer, pour chaque variable $X \in V$, le mot (unique) engendré par la variable X. Pour ce faire, on utilise un tableau d'options de chaînes de caractères, qu'on remplit au fur et à mesure. La fonction récursive `genere` est une fonction mémoïsée qui renvoie la chaîne engendré par une variable.

```
let engendre g =
    let n = Array.length g in
    let gen = Array.make n None in
    let rec genere i =
        if gen.(i) = None then begin
            let traiter_symb = function
                | T c -> String.make 1 c
                | V j -> genere j
            in
            let s = List.fold_left (fun str symb -> str ^ traiter_symb symb) "" g.(i) in
            gen.(i) <- Some s
        end;
        match gen.(i) with
            | None -> failwith "Erreur"
            | Some s -> s
    in
    genere 0
```

Question 18 Sans compter les appels récursifs, le premier appel à `genere` pour chaque variable va faire au plus m concaténations, avec des mots de taille au plus $|v|$. Comme dans le pire cas on lance un appel à `genere` par variable, la complexité totale est en $\mathcal{O}(m|v||V|)$.

Question 19 Soit G une grammaire telle que $K_G(v) = |G|$. Si G n'est pas accessible, supprimer les variables non accessibles et leurs règles de productions ne change pas le langage engendré et réduit la taille de la grammaire. On peut donc supposer G accessible.

Si G n'est pas élémentaire et possède deux règles $X \to \alpha \mid \beta$, alors on distingue :

- soit $\alpha \Rightarrow^* u$ et $\beta \Rightarrow^* v$ avec $u, v \in T^*$, $u \neq v$, ce qui est absurde car alors $|L(G)| > 1$;
- soit toute dérivation de α et β donnent nécessairement le même mot de T^*, auquel cas on peut supprimer une des deux règles.

On peut donc supposer G élémentaire.

Dès lors, si G est accessible et élémentaire, sachant que $|L(G)| = 1$, on en déduit que G est acyclique, d'après la question 15.

Question 20 Soit $G_1 = (T, V_1, P_1, S_1)$ et $G_2 = (T, V_2, P_2, S_2)$ des grammaires telles que $L(G_1) = \{v\}$, $L(G_2) = \{w\}$ et $|G_1| = K_G(v)$, $|G_2| = K_G(w)$.

On pose $G = (T, V, P, S)$, avec :

- $V = V_1 \cup V_2 \cup \{S\}$;
- $P = P_1 \cup P_2 \cup \{S \to S_1 S_2$.

Alors il est clair que $L(G) = \{vw\}$, et $|G| = |G_1| + |G_2| + 2$, ce qui donne la majoration voulue.

Question 21 On considère $G = (T, V, P, S)$ une grammaire telle que $L(G) = \{v\}$ et $|G| = K_G(v)$. On construit par récurrence une grammaire G_k qui engendre v^k :

- $G_1 = G$;
- pour $k > 1$, on pose $\ell = \lfloor \frac{k}{2} \rfloor$. Si $G_\ell = (T, V_\ell, P_\ell, S_\ell)$, on distingue :
  - si $k = 2\ell$, on pose $G_k = (T, V_\ell \cup \{S_k\}, P_\ell \cup \{S_k \to S_\ell S_\ell\}, S_k)$;
  - si $k = 2\ell + 1$, on pose $G_k = (T, V_\ell \cup \{S_k\}, P_\ell \cup \{S_k \to S S_\ell S_\ell\}, S_k)$

  Il est clair que $L(G_k) = \{v^k\}$ (c'est le même principe que l'exponentiation rapide dans la question 2). Dans tous les cas, $|G_k| \leqslant |G_\ell| + 3$.

On en déduit par récurrence que $|G_k| \leqslant |G| + 3 \log_2 k$, ce qui donne le résultat attendu.

Question 22 Si $\alpha_i \notin T^*$, on considère la variable X_j apparaissant dans α_i qui maximise $|g(X_j)|$. Chaque lettre de α_i peut être dérivée en un mot de taille au plus $\max(1, |g(X_j)|)$ (le max est nécessaire si $g(X_j) = \varepsilon$, car il peut y avoir des terminaux dans α_i). On en déduit la majoration attendue.

Question 23 Si on suppose que $(X_0, \ldots, X_{n-1})$ est un ordre topologique, alors dans la question précédente, on a nécessairement $j > i$. On en déduit par récurrence que $|g(S)| \leqslant |g(X_0)|$ est de taille inférieure au produit des $|\alpha_i|$ non nuls.

En utilisant la propriété mathématique énoncée, on obtient donc que $|v| = |g(S)| \leqslant 3^{\frac{M}{3}}$, avec $M = \sum_{i=0}^{n-1} |\alpha_i| = |G|$. Cela donne $3 \log_3 |v| \leqslant |G|$, et ce pour toute grammaire G qui engendre v, donc $K_G(v) = \Omega(\log |v|)$.

3 Plus petite grammaire

Question 24 Un certificat est une grammaire (élémentaire, acyclique et accessible) $G = (T, V, P, S)$. Pour que la vérification se fasse en temps polynomial, il faut prendre quelques précautions :

- on peut supposer que $|V| \leqslant |v|$, car dans tous les cas, $K_G(v) \leqslant |v|$ (car une règle $S \to v$ suffirait) ;
- on peut supposer que pour chaque règle $X \to \alpha$, on a $|\alpha| \leqslant |v|$ (car sinon on aurait des règles $Y \to \varepsilon$ inutiles).

Avec le type OCaml décrit précédemment, une telle grammaire serait bien de taille polynomiale en l'entrée. D'après la question 18, la vérification que $L(G) = \{v\}$ se fait bien en temps polynomial. On peut également vérifier en temps polynomial que $|G| \leqslant k$. On en déduit que $\mathsf{PPG} \in \mathsf{NP}$.

Question 25 On définit les règles suivantes (qui donnent les variables apparaissant dans la grammaire) :

- $X_i \to \#s_i$;
- $Y_i \to s_i\#$;
- pour $s_i \in C$, $Z_i \to X_i\#$.

Pour la règle initiale, on définit des mots intermédiaires :

– pour $s = s_i \in C$, on pose $\alpha_i = X_i a_i Y_i b_i X_i c_i Y_i d_i Z_i e_i$;
– pour $s = s_i \notin C$, on pose $\alpha_i = X_i a_i Y_i b_i X_i c_i Y_i d_i X_i \# e_i$;
– pour $a = a_j = \{s_i, s_k\} \in A$, si $s \in C$, on pose $\beta_j = Z_i Y_k a$, sinon on pose $\beta_j = X_i Z_k a$.

Finalement, on définit la règle $S \to \alpha_1 \dots \alpha_p \beta_1 \dots \beta_q$.

Par définition, C étant une couverture des arêtes par les sommets, on a bien $S \Rightarrow^* v$. De plus, on a :

$$|G| = 2|S| + 2|S| + 2|C| + 10|S| + (|S| - |C|) + 3|A| = 15|S| + 3|A| + k$$

Question 26 On remarque que les symboles qui ne sont ni un $s \in S$, ni un # n'apparaissent qu'une seule fois dans le mot v. On en déduit qu'on ne fait pas diminuer la taille de la grammaire en supposant que ces lettres n'apparaissent que dans la règle de production ayant S comme membre gauche (qu'on appelle la « règle initiale »). Sachant que les facteurs qui apparaissent plusieurs fois ne sont que les $\#s$, $s\#$ ou $\#s\#$, on en déduit qu'on peut supposer que les autres variables que S ne peuvent se dériver qu'en un mot d'une de ces trois formes.

Question 27 Supposons qu'il n'existe aucun X_i tel que $g(X_i) = \#s$. Alors $\#s$ est un facteur qui apparait au moins deux fois dans la règle initiale. En remplaçant ces occurrences par une nouvelle variable X et une règle $X \to \#s$, la taille de la grammaire n'augmente pas. On raisonne de même pour $s\#$.

Question 28 Supposons qu'il existe une arête $a = a_j = \{s, t\}$ telle que $s \notin C$ et $t \notin C$. Alors obtenir le facteur w_j nécessite au moins 4 symboles dans les membres gauches. Si $g(X_i) = \#s$ et $g(X_k) = t\#$, alors ajouter une règle $X \to X_i\#$ et remplacer les symboles qui génère w_j par XX_ka fait augmenter de 1 la taille de la grammaire, mais on peut alors la faire diminuer de 1 en remplaçant par X la partie qui génère $\#s\#$ dans le mot $v_{i'}$ correspondant. Sans perte de généralité, on peut donc supposer que C est une couverture des arêtes par les sommets.

La taille de la grammaire est alors $15|S| + 3|A| + |C|$, et on obtient bien une couverture des arêtes par les sommets de taille $|C|$.

Question 29 Les questions 25 et 28, et le fait que la construction du mot v et de l'entier $15|S| + 3|A| + k$ se fassent en temps polynomial donne le fait que $\texttt{COUV} \leqslant^{\mathsf{p}}_{\mathsf{m}} \texttt{PPG}$. On en déduit que PPG est NP-difficile, donc NP-complet (car il est dans NP).

Question 30 On obtient l'encodage $(0, 1, 4, 2, 6, 3, 8, 10)$ et la table :

a	b	c	d	ab	ba	abc	ca	$abcd$	da	$abcda$
0	1	2	3	4	5	6	7	8	9	10

Question 31 On obtient le mot *aaabaabbacdbac* et la table :

a	b	c	d	aa	aab	ba	$aabb$	bac	cd	db
0	1	2	3	4	5	6	7	8	9	10

Question 32 On applique l'algorithme tel qu'il est décrit. La fonction `plus_long_prefixe` prend en argument un mot u et, si le plus long préfixe de u est de la forme wa avec $w \in D$ et $a \in \Sigma$, alors la fonction renvoie le triplet w, a, u' tel que $u = wau'$.

```
let compresse_lz78 v =
    let d = Hashtbl.create 1 in
    Hashtbl.add d "" 0;
    let x = ref [] and k = ref 1 and vcur = ref v in
    let plus_long_prefixe u =
        let n = String.length u in
        let i = ref 0 in
        while !i < n &&
            Hashtbl.mem d (String.sub u 0 !i) do
            incr i;
        done;
        String.sub u 0 (!i - 1), u.[!i-1], String.sub u !i (n - !i)
    in
    while !vcur <> "" do
        let w, a, reste = plus_long_prefixe !vcur in
        x := (Hashtbl.find d w, a) :: !x;
        Hashtbl.add d (w ^ String.make 1 a) !k;
        incr k;
        vcur := reste;
    done;
    List.rev !x
```

Question 33 On remarque que lorsqu'on retire un préfixe wa à v, on ajoute le couple $(D[w], a)$. On en déduit réciproquement qu'à chaque fois qu'on lit un couple (k_i, a_i), il faut rajouter un facteur $D^{-1}(k_i)a_i$. Il suffit alors de montrer que $g(X_i) = D^{-1}(k_i)a_i$.

C'est vrai par définition, car lorsque la i-ème association (wa, k) est ajoutée au dictionnaire, on a la règle $X_i \to X_k a$, donc si on suppose que $g(X_k) = w$, on obtient bien que $g(X_i) = wa$.

Question 34 Il suffit de construire la grammaire, puis de renvoyé le mot engendré par cette grammaire. On met X_0 en indice $n+1$ pour laisser l'indice 0 pour le symbole S.

L'appel à `List.iteri` rajoute chaque règle $X_i \to X_{k_i} a_i$, en prenant garde à renuméroter X_0 à l'indice $n+1$.

```
let decompresse_lz78 x =
    let n = List.length x in
    let g = Array.make (n + 2) [] in
    g.(0) <- List.init n (fun i -> V (i + 1));
    g.(n + 1) <- [];
    List.iteri (fun i (k, a) -> g.(i + 1) <- [V (if k = 0 then n + 1 else k); T a]) x;
    engendre g
```

Question 35 Soit $X \to \alpha$ une règle. Alors un facteur propre de taille k de $g(X)$ (c'est-à-dire un facteur de $g(X)$ qui n'est pas facteur d'un $g(Y)$ avec Y apparaissant dans α) peut commencer soit par un terminal de α (il y en a au plus $|\alpha|$), soit par entre 1 et $k-1$ terminaux d'un $g(Y)$ avec Y apparaissant dans α (il y a donc $(k-1)|\alpha|$ tels choix). On en déduit que $g(X)$ contient au plus $k|\alpha|$ facteurs propres.

En sommant sur toutes les règles, il y a au plus $k|G|$ facteurs propres.

Il reste à montrer que tout facteur de taille k de v est un facteur propre d'un des $g(X)$ pour $X \in V$. En effet, soit w un tel facteur et X tel que w est un facteur de $g(X)$, avec $|g(X)|$ le plus petit possible. Par définition de X, w est un facteur propre de $g(X)$.

Question 36 Grâce à la question précédente, on en déduit que :

- il existe au plus κ facteurs de taille 1 de v, donc au plus κ variables X telles que $|g(X)| = 1$, donc la somme des tailles des $g(X)$ pour les variables X du premier paquets est minorée par $1 \times \kappa$;

– il existe au plus 2κ facteurs de taille 2 de v, donc au plus 2κ variables X telles que $|g(X)| = 2$, donc la somme des tailles des $g(X)$ pour les variables X du deuxième paquets est minorée par $2 \times 2\kappa$;
– ...

En appliquant ce raisonnement sur chaque paquet, et en ignorant le dernier paquet dans la minoration, on obtient bien $|v| = |g(X_1)| + |g(X_2)| + \ldots + |g(X_n)| \geqslant 1^2\kappa + 2^2\kappa + 3^2\kappa + \ldots + p^2\kappa$.

Question 37 Par les inégalités précédentes et la question 23, on a :

– $|G| = 3n = \mathcal{O}(p^2\kappa)$;
– $|v| = \Omega(p^3\kappa)$;
– $\kappa = \Omega(\log |v|)$.

On cherche à obtenir une majoration de $\frac{|G|}{\kappa}$ pour obtenir le facteur d'approximation :

$$\frac{|G|}{\kappa} = \mathcal{O}(p^2) = \mathcal{O}\left(\left(\frac{|v|}{\kappa}\right)^{2/3}\right) = \mathcal{O}\left(\left(\frac{|v|}{\log |v|}\right)^{2/3}\right)$$

Ce qui est bien le résultat attendu.

⋆ ⋆ ⋆