Rapport sur l'épreuve écrite
ENS Informatique MP 2009

Pierre Letouzey, Stéphane Demri, Yves Robert

1er septembre 2009

1 Présentation du sujet

Le sujet se divisait en deux parties indépendantes, portant toutes les deux sur des propriétés de mots.

Dans la première partie, on s'intéressait aux mots de Lyndon (que l'on nommait "mots minimaux" dans le sujet), à savoir des mots strictement inférieurs lexicographiquement à toutes leurs rotations circulaires. Les questions portaient sur différentes manières de caractériser et d'énumérer les mots de Lyndon, ainsi qu'à diverses propriétés de décompositions de ces mots.

La seconde partie étudiait un mot binaire infini servant à engendrer une figure fractale nommée "courbe du dragon". On demandait en particulier de montrer qu'une même suite croissante de préfixes de ce mot infini peut s'obtenir de deux manières récursives différentes, puis de voir ultérieurement la conséquence de cela en terme graphique. On s'intéressait également à différentes manières de déterminer directement une lettre précise du mot infini. Enfin on étudiait une propriété de la courbe du dragon : elle peut rentrer en contact avec elle-même, mais sans jamais se croiser.

2 Remarques générales

Pour une épreuve de quatre heures, ce sujet était d'une longueur importante. La notation a été adaptée en conséquence (voir les données chiffrées de la section suivante). Les questions les plus dures (1.12 puis 2.10 et suivantes) n'ont quasiment pas été traitées. Les questions 1.13 et 1.14, pourtant relativement abordables, ont également été très peu traitées, sans doute leur position en toute fin de première partie a-t-elle joué.

Cette année encore, un net manque de rigueur est à déplorer dans beaucoup de copies. Il est inadmissible de voir aussi souvent la stratégie consistant à en écrire le maximum, puis à laisser le correcteur faire le tri entre ce qui est bon et ce qui est faux. Cela ne peut aboutir qu'à l'agacement du correcteur, au détriment de la note finale. La place manque ici pour énumérer toutes les erreurs de raisonnement rencontrées, ainsi que les propriétés fausses prétendument prouvées. Parmi les erreurs les plus communes, beaucoup sont liées à l'ordre lexicographique, qui a le mauvais goût de contredire fréquemment le "bon sens". Un exemple parmi d'autres : il est faux en général d'affirmer $v < r$ implique $vw < rw$. Idem d'ailleurs pour la réciproque. Autre

erreur fréquente, ce n'est pas parce qu'un mot v est à la fois un début et une fin de u que u peut s'écrire sous la forme $u = vrv$. On peut juste dire (et encore, cela demande à être proprement justifié) que u peut alors s'écrire sous la forme $u = wrw$ avec w un autre début et fin de u (éventuellement plus court que v). Même quand il n'y a pas d'erreur, le style des preuves est souvent perfectible : un bon nombre des raisonnements par l'absurde que l'on rencontre ne sont en fait pas nécessaires, ce qui aboutit à des preuves inutilement complexes.

Concernant les algorithmes, rappelons que tout algorithme non-trivial doit impérativement être justifié, ce qui n'a été que rarement fait. Le pseudo-code attendu en réponse était volontairement laissé assez libre, mais il va de soi que l'usage de "..." ou de "etc" ou encore de phrase descriptives vagues en français ne peuvent compter comme réponses satisfaisantes. Attention également aux cas particuliers : comme précisé dans les préliminaires, un tableau représentant un mot pouvait être de taille nulle. Cela rendait par exemple incorrect tout code commençant par "while u[i]=..." même si i est initialement à 0.

Quelques mots maintenant sur certaines questions précises.

- Les réponses à la question 1.8 se sont avérées très décevantes. Même si cette question peut sembler une simple réciproque de la 1.7, elle était en fait nettement plus difficile : il fallait déjà imaginer où couper le mot en deux, avant de trouver la justification correcte de cette découpe. Et non, séparer première lettre et reste du mot ne convenait pas, il suffit de considérer abb pour s'en convaincre. Idem pour une découpe à la dernière lettre, cf aab. Même s'il était légitime d'essayer initialement ces découpages simples, il est nettement plus gênant de laisser sur sa copie de soit-disantes "preuves" de ces découpages alors même que la question 1.10 forçait à énumérer quelques exemples de base, dont ces abb et aab.
- La question de programmation 1.9 a été globalement négligée, alors qu'il s'agissait d'une des rares questions demandant un codage plus long, non direct, et où de l'originalité était possible, avec un nombre de points qui avait été prévu en conséquence. Si l'idée d'exploiter les questions 1.7 et 1.8 (afin de générer les mots minimaux de taille n à partir de la liste des mots minimaux de tailles inférieures) ne venait pas au candidat, il n'était pas interdit de donner au moins l'algorithme naïf consistant à engendrer tous les mots, puis à vérifier leur minimalité. Sans forcément rapporter le maximum de points, cela constituait au moins une réponse correcte, mieux récompensée qu'une réponse plus ambitieuse mais fausse.
- A propos de la question 1.13, il est dommage de voir que bien peu de candidats aient pensé à l'approche adéquate, dite "gloutonne" (ou "par saturation"). Tant que la décomposition courante est non-minimale, c'est qu'il existe dedans deux mots minimaux successifs u et v avec $u < v$. D'après la question 1.7, uv est alors minimal, ce qui nous donne une nouvelle décomposition. Une fois itéré, ce procédé aboutit forcément à une factorisation adéquate. Quant à l'initialisation, il suffit de partir de la décomposition en lettres. Il est bon de rappeler que la méthode "diviser pour régner" rend certes de nombreux services, mais ne résout pas tous les problèmes du monde. L'utiliser ici était a priori sans espoir, et a donné lieu à un défilé d'algorithmes faux non/mal justifiés.

3 Informations quantitatives

Cette épreuve a été traitée par 356 candidats (sur 488 inscrits). La moyenne globale a été de 10/20 pour un écart-type de 3,9. Les notes se sont réparties entre un 0,70 et plusieurs 20. Vu la longueur du sujet, traiter entièrement une seule des deux parties suffisait largement pour avoir la note maximale. De plus, des réponses totalement satisfaisantes aux questions 1.1 à 1.7 assuraient déjà d'avoir au moins 9,4 : on ne saurait trop redire aux candidats de privilégier la qualité et non de se livrer à une course à la quantité.

Voici quelques statistiques par question (sur environ un tiers des copies) :

Question	% de réponses	Note moyenne de ces réponses
1.1	99%	75%
1.2	97%	80%
1.3	95%	56%
1.4	95%	74%
1.5	92%	79%
1.6	91%	52%
1.7	87%	45%
1.8	47%	8%
1.9	52%	34%
1.10	60%	50%
1.11	35%	76%
1.12	17%	12%
1.13	20%	7%
1.14	10%	14%
2.1	77%	75%
2.2	57%	36%
2.3	37%	74%
2.4	27%	60%
2.5	18%	32%
2.6	7%	0%
2.7	21%	66%
2.8	9%	16%
2.9	1%	50%
2.10	1%	50%
2.11	1%	0%
2.12	1%	0%

4 Éléments de correction

Avertissement : Le corrigé qui suit était initialement destiné à l'équipe de correcteurs. Il est inclus ici à titre indicatif : outre les éventuelles coquilles subsistantes, la rédaction de certaines questions n'est pas forcément à prendre comme modèle de ce qui était attendu de la part des candidats. D'autre part, les portions de code données en solution ont été écrites en Ocaml, langage préférentiel du concepteur. L'adaptation de ces codes à Caml Light (voire à Pascal) est laissé en exercice au

lecteur.

Le corrigé reprend les conventions et notations introduites dans le sujet, en particulier à la page 2 “Préliminaires”.

Pour information, une large partie des preuves qui suivent ont été formalisées sur ordinateur à l'aide du logiciel Coq, ce qui a permis de vérifier mécaniquement leur validité. Le lecteur intéressé pourra trouver plus d'informations sur ce logiciel d'aide à la démonstration à l'adresse `http://coq.inria.fr`. Quant à la formalisation proprement dite de ce sujet, son code source est disponible à l'adresse suivante :

`http://www.pps.jussieu.fr/~letouzey/ens2009`

Corrigé Partie 1 : Rotation de mots et minimalité

Ordre lexicographique sur les mots. À partir de l'ordre total $<$ sur les lettres de l'alphabet, on obtient un ordre sur les mots comme suit :

- Pour deux mots u et v, on note $u \lhd v$ lorsqu'il existe un entier naturel i tel que $i < min(|u|, |v|)$ et $u[i] < v[i]$, ainsi que $u[j] = v[j]$ pour tout $j < i$.
- On note ensuite $u < v$ lorsque $u \in Pre(v) \setminus \{v\}$ ou $u \lhd v$.
- Enfin $u \leq v$ lorsque $u < v$ ou $u = v$.

On parlera d'*ordre lexicographique* (strict ou large) pour ces relations $<$ et $\leq$ sur les mots.

Question 1.1 *Montrer que la relation $\leq$ est une relation d'ordre totale sur les mots. Admet-elle de plus petit et de plus grand élément ? Écrire en pseudo-code un algorithme* EstPlusPetit *prenant en argument deux mots u et v et déterminant si $u < v$. Donner sa complexité.*

Correction : une question simple quoique assez laborieuse à rédiger.

- Réflexivité de $\leq$: d'après la définition.
- La transitivité de $\leq$ découle aisément de celle de $<$. Pour cette dernière, il y a 4 cas :
  - Considérons tout d'abord le cas $u \lhd v$ et $v \lhd w$. Soit i l'indice où u et v diffèrent, et j celui où v et w diffèrent. Avec $k = min(i, j)$, on a $u[k] < v[k] = w[k]$ ou bien $u[k] = v[k] < w[k]$ ou bien $u[k] < v[k] < w[k]$, et u et w coïncident avant cela, d'où $u \lhd w$.
  - Si $u \in Pre(v) \setminus \{v\}$ et $v \in Pre(w) \setminus \{w\}$ alors $u \in Pre(w) \setminus \{w\}$.
  - Si $u \lhd v$ et $v \in Pre(w) \setminus \{w\}$ alors $u \lhd w$ (la dissemblance entre u et w se produit au même endroit que pour u et v).
  - Enfin, supposons $u \in Pre(v) \setminus \{v\}$ et $v \lhd w$. Soit i l'indice où v et w diffèrent. Si $|u| < i$ alors $u \in Pre(w) \setminus \{w\}$. Si $i \leq |u|$ alors $u \lhd w$.
- Antisymétrie. Si $u < v$ et $v < u$, alors d'après la transitivité que l'on vient d'établir, $u < u$. Mais cela est exclu, vu qu'on ne peut avoir ni $u \lhd u$ ni $u \in Pre(u) \setminus \{u\}$. Au final, $u \leq v \leq u$ implique donc forcément $u = v$. Remarque : il est également possible de faire une preuve directe d'antisymétrie

sans exploiter la transitivité, mais la preuve par cas est longue.

- Totalité. Soit il existe un (premier) indice i où u et v diffèrent, et alors $u \lhd v$ ou bien $v \lhd u$ selon la comparaison de $u[i]$ et $v[i]$. Sinon u et v coïncident jusqu'à la fin du plus petit des deux. On a alors $u < v$ ou $u = v$ ou $v < u$ selon que $|u| < |v|$ ou $|u| = |v|$ ou $|v| < |u|$.
- Un codage de EstPlusPetit en Ocaml :

```
exception Stop of bool

let estpluspetit u v =
  try
    for i = 0 to min (Array.length u) (Array.length v) - 1 do
      if u.(i) < v.(i) then raise (Stop true)
      else if u.(i) > v.(i) then raise (Stop false)
    done;
    (* si on arrive ici, u est un prefixe de v ou vice-versa *)
    (Array.length u < Array.length v)
  with Stop b -> b
```

Cet algorithme a une complexité $O(min(|u|, |v|))$.

Rotation et minimalité. Un mot v est une rotation d'un autre mot u lorsqu'il existe deux autres mots non-vides w et r tels que $u = wr$ et $v = rw$. Un mot u sera dit *minimal* s'il est non-vide et que $u < v$ pour toute rotation v de u. On remarquera en particulier qu'un mot de longueur 1 est minimal.

1.1 Tests de minimalité

Question 1.2 *Écrire en pseudo-code un algorithme* EstMinimal1 *testant si un mot est minimal. Justifier sa correction et donner sa complexité.*

Correction :

```
(* rotation: pour un mot u non-vide de la forme u=wr avec |w|=k, fabrique v=rw *)

let rotation u k =
  let n = Array.length u in
  let v = Array.make n u.(0) in
  for i = 0 to n - 1 do
    v.(i) <- u.((i+k) mod n)
  done;
  v

let estminimal1 u =
  let n = Array.length u in
  if n = 0 then false
  else
    try
      for k = 1 to n - 1 do
        if not (estpluspetit u (rotation u k)) then raise (Stop false)
      done; true
    with Stop b -> b
```

On implémente simplement la définition. La complexité est quadratique en la taille de u. Remarque : il est possible de travailler "en place", en comparant des indices

grâce à des modulo.

Question 1.3 *Soit u un mot tel que $Deb(u) \cap Fin(u) \neq \varnothing$. Montrer que u ne peut être minimal. Cette propriété suffit-elle à caractériser les mots non-minimaux ?*

Correction : On écrit $u = vw = rv$ avec $0 < |v| < |u|$. Supposons par l'absurde u minimal. Comme vr est une rotation de u, on a $u < vr$, ce qui signifie $vw < vr$, et donc $w < r$. Comme w et r ont la même taille, c'est donc que $w \lhd r$. L'existence d'un point de dissemblance ne change pas si l'on rallonge ces mots, donc $wv \lhd rv$ et du coup $wv < rv = u$, ce qui contredit la minimalité de u.

Remarque : il est tentant, mais faux, d'affirmer des choses du genre $w < r \Rightarrow wv < rv$ sans plus d'hypothèses sur w et v.

Enfin, si a et b sont deux lettres telles que $a < b$, alors le mot ba n'est pas minimal vu que sa rotation ab est plus petite, mais pourtant $Deb(ba) \cap Fin(ba) = \{b\} \cap \{a\} = \varepsilon$.

Question 1.4 *Montrer qu'un mot non-vide u est minimal si et seulement si toute fin v de u est telle que $u < v$.*

Correction : Soit $u \neq \varepsilon$. Supposons u minimal, et considérons une fin v de u : $u = rv$. Alors par minimalité $u < vr$. Comme u ne peut être un préfixe d'un mot de même longueur sans lui être égal, on a donc $u \lhd vr$. Si le premier indice où u et vr diffèrent était supérieur à $|v|$, on aurait $v \in Deb(u) \cap Fin(u)$ et donc u ne pourrait être minimal. On obtient donc $u \lhd v$ et donc $u < v$.

Réciproquement, supposons que toute fin v de u est telle que $u < v$. Soit vr une rotation de $u = rv$. On a à la fois $u < v$ (car v est une fin de u) et $v < vr$ (préfixe plus court), d'où $u < vr$ par transitivité. Donc u est bien minimal.

Question 1.5 *En utilisant cette caractérisation, écrire en pseudo-code un algorithme* EstMinimal2 *qui teste si un mot u est minimal. Par rapport à* EstMinimal1*, déterminer le gain en nombre de comparaisons de lettres nécessaires dans le cas le pire.*

Correction :

```
let estminimal2 u =
  let n = Array.length u in
  if n = 0 then false
  else
    try
      for k = 1 to n - 1 do
        if not (estpluspetit u (Array.sub u k (n-k))) then raise (Stop false)
      done; true
    with Stop b -> b
```

On a encore une complexité quadratique, mais on économise la moitié des comparaisons au pire : $n(n-1)/2$ contre $n(n-1)$.

1.2 Énumération des mots minimaux

Question 1.6 *Montrer que l'ordre $\leq$ restreint aux mots minimaux admet un plus petit élément et un plus grand.*

Correction : Appelons a la première lettre de l'alphabet A, et z la dernière, selon l'ordre $<$ sur les lettres. Alors pour tout mot u non-vide (et pas seulement ceux minimaux), alors $a \leq u$. En effet, soit la première lettre de u n'est pas a, et alors $a \lhd u$, soit u commence par un a, et donc $a \in Pre(u)$. D'autre part, pour tout mot u minimal, alors $u \leq z$. En effet, si on avait $z < u$, alors u ne peut être de longueur 1, et donc sa dernière lettre l forme une fin de u, et donc (par la question précédente) $z < u < l$. On obtiendrait donc une lettre strictement supérieure à z, ce qui est impossible.

Pour $n > 0$, on appelle n-minimal tout mot minimal de longueur inférieure ou égale à n, et on note $M(n)$ le nombre de mots n-minimaux.

Question 1.7 *Soient v et w deux mots minimaux tels que $v < w$. Montrer alors que vw est également minimal.*

Correction : On utilise la caractérisation de la question 1.4. Soit f une fin de vw. Trois cas selon la longueur de f :

- Soit f est une fin de w. Alors $w < f$, et comme on sait $v < w$, on obtient $v < f$ par transitivité. Si cela provient de $v \lhd f$, alors on peut rallonger le premier mot sans changer l'ordre, et donc $vw < f$. L'autre éventualité est que v soit un préfixe de f différent de f, notons alors $f = vr$. Ce r est une fin de w, donc $w < r$, ce qui implique $vw < vr = f$.
- Soit $f = w$, et alors $v < w = f$. On conclut par le même raisonnement qu'au premier cas.
- Soit $f = gw$ avec g une fin de v. Alors $v < g$. Comme $|g| < |v|$, c'est donc que $v \lhd g$, on peut donc rallonger ceci en $vw < gw = f$.

Question 1.8 *Réciproquement, pour un mot u minimal tel que $|u| > 1$, prouver l'existence de deux mots minimaux v et w tels que $u = vw$ et $v < w$.*

Correction : Déjà, si u minimal se découpe en deux mots non-vides $u = vw$, alors $v < w$. En effet, d'après la question 1.4, on a $u < w$ et d'autre part v est un début de u, donc $v < u$, d'où la conclusion par transitivité.

Cherchons maintenant à découper u en deux mots minimaux. Comme $|u|$ est de longueur au moins 2, il existe forcément des fins de u qui sont minimales : par exemple, la dernière lettre de u forme un mot minimal. Soit w la plus longue fin minimale de u, notons v le début correspondant : $u = vw$. Supposons par l'absurde que v n'est pas minimal. Il existe alors une fin f de v qui ne vérifie pas $v < f$, ce qui signifie que $f < v$ (vu que $f = v$ est exclu). On choisit la plus courte de ces fins f. On a $f < v < w$. f ne peut être minimal, sinon fw le serait aussi d'après la question précédente, ce qui contredirait le choix de taille maximale de w. La non-minimalité de f donne une fin g de f (et donc de v) telle que $g < f < v$. Cela contredit le choix de taille minimale de f.

Remarque : ce découpage de u n'est pas forcément unique. Le plus long début minimal de u a l'air de donner également un découpage adéquat (mais la preuve a l'air plus dure).

Question 1.9 *À l'aide des questions précédentes, écrire en pseudo-code un algorithme* EnumMinimaux *prenant en entrée la liste des lettres de l'alphabet A dans l'ordre, ainsi que l'entier n, et retournant en sortie la liste de tous les mots n-minimaux dans l'ordre lexicographique. Justifier la correction de* EnumMinimaux *et donner sa complexité.*

Correction :

```
(** Pour un n-minimal u et la liste l des n-minimaux > u,
    fabrique la liste des minimaux de taille n de la forme uv
    avec v dans l *)

let rec prod n u = function
  | [] -> []
  | v::l ->
      if Array.length u + Array.length v = n then
        Array.append u v :: prod n u l
      else prod n u l

(** Pour une liste de n-minimaux triés selon <, fabrique
    la liste des minimaux de taille n de la forme uv avec
    u et v dans l. *)

let rec prod_cart n = function
  | [] -> []
  | u::l -> prod n u l @ prod_cart n l

(** On suppose savoir trier une liste de mots selon < et
    y enlever les redondances *)

let rec uniq = function
  | a :: (b :: _ as l) when a = b -> uniq l
  | a :: l -> a :: uniq l
  | [] -> []

let compare u v = if u = v then 0 else if lt u v then -1 else 1

let sort_uniq l = uniq (List.sort compare l)

(** L'algorithme enumminimaux en lui-même: a chaque tour, on obtient
    les mots minimaux de taille n grace a [prod_cart] appliqué à la liste
    des (n-1)-minimaux obtenu au tour précédents. Puis on combine et
    trie tout cela pour obtenir les n-minimaux.
*)

let rec enumminimaux lettres n =
 if n = 0 then []
 else if n = 1 then List.map (fun lettre -> [|lettre|]) lettres
 else
   let l = enumminimaux lettres (n-1) in
   sort_uniq (l @ prod_cart n l)
```

La correction repose sur la question précédente : on est sûr qu'un mot minimal de taille $n > 1$ sera le collage de deux mots $(n-1)$-minimaux v et w tels que $v < w$. La complexité est exponentielle : à chaque tour, on travaille sur une liste l de taille $M(n-1)$ (qui est bornée par $|A|^{n-1}$), puis on crée un sous-ensemble de $l \times l$, avant

de trier cela. Cela donne une complexité en $\Sigma_n \phi(M(n))$ avec $\phi(k) = k^2 ln(k)$, que l'on peut par exemple majorer grossièrement par un $O(|A|^{3n})$.

Pour le reste de cette section, on utilise l'alphabet A_2.

Question 1.10 *Calculer alors $M(n)$ pour $0 < n \leq 5$. Montrer que tous les mots minimaux de longueur au moins 2 partagent la même première lettre, ainsi que la même dernière lettre. Pour $n > 0$, prouver l'encadrement suivant : $2 + n(n-1)/2 \leq M(n) \leq 2^{(n-1)} + 1$.*

Correction :
$M(1) = 2$ (mots a et b)
$M(2) = 3$ (on ajoute ab)
$M(3) = 5$ (on ajoute aab et abb)
$M(4) = 8$ (on ajoute $aaab$, $aabb$ et $abbb$)
$M(5) = 14$ (on ajoute $aaaab$, $aaabb$, $aabab$, $aabbb$, $ababb$ et $abbbb$)

Pour un mot minimal u de taille au moins 2, sa dernière lettre l est une fin de u, donc $u < l$ par la question 1.4. On en déduit que la première lettre de u est strictement inférieure à sa dernière, ce qui n'est possible que s'il s'agit respectivement de a et b.

Soit $L(n)$ le nombre de mots minimaux de taille exactement n. Pour $n > 1$, on a donc $L(n) \leq 2^{n-2}$ une fois fixé la première et dernière lettre. Par ailleurs, pour $0 < k < n$, les mots $a^k b^{n-k}$ sont minimaux de taille n, donc $L(n) \geq n - 1$. En sommant ces encadrements, on obtient les inégalités voulues.

Remarque : on peut aussi minorer $M(n)$ par une exponentielle, mais c'est plus ardu. Expérimentalement $Fib(n+1)$ convient, mais pas moyen de l'établir directement. On peut établir une formule précise pour $L(n)$, vu qu'un mot minimal de taille n est le représentant de la classe de ses rotations, classe qui est de taille n, et que les mots ne tombant pas dans une de ces classes sont les mots périodiques, ce qui fait intervenir un $L(d)$ pour $d|n$. De là, on peut montrer que $L(n) \sim 2^n/n$ par un encadrement donnant aussi $L(n) \geq F(n-1)$, d'où on déduit la minoration $M(n) \geq Fib(n+1)$.

Il existe en fait un algorithme direct permettant de passer d'une étape dans l'énumération des n-minimaux à l'étape suivante :

```
MinimalSuivant(entiers n et m, tableau T de taille n)
 1  ▷ Les m premières cases de T doivent former un mot n-minimal u ≠ b
 2  Pour i ← m à n − 1 Faire
 3      T[i] ← T[i modulo m]
 4  Fin Pour
 5  p ← n − 1
 6  Tant Que T[p] = b Faire
 7      p ← p − 1
 8  Fin Tant Que
 9  T[p] ← b
10  Retourner p + 1
11  ▷ Les p + 1 premières cases de T forment maintenant le mot
    n-minimal qui suit u
```

Question 1.11 *Montrer que toute utilisation de* MinimalSuivant *conforme aux conditions de la ligne 1 termine en retournant un entier q tel que $1 \leq q \leq n$. Déterminer la complexité de* MinimalSuivant.

Correction : Tout mot n-minimal distinct de b admet a comme première lettre. Le "Tant Que" des lignes $6-8$ va donc bien s'arrêter sur un $p \geq 0$, d'où $1 \leq p+1 \leq n$ vu la valeur initiale de p. Enfin, la complexité est $O(n)$.

Question 1.12 *Pour $n > 0$ et u un mot n-minimal différent de b, montrer que* MinimalSuivant *permet bien d'obtenir un mot n-minimal v, puis montrer que v est le mot n-minimal le plus petit (au sens de $<$) tel que $u < v$.*

Correction : Pour un mot w contenant au moins un a, on appelle $succ(w)$ le mot dans lequel le dernier a de w devient un b, et les éventuels b ultérieurs sont enlevés. On remarque que $w \lhd succ(w)$, que $|succ(w)| \leq |w|$, et que $succ(w)$ est le plus petit mot selon $<$ vérifiant ces deux propriétés.

De même, pour un mot w terminant par b, on appelle $pred_m(w)$ le mot dans lequel ce b final devient un a suivi de m lettres b. On a donc $pred_m(w) < w$ et $|pred_m(w) \leq |w| + m$, et $pred_m(w)$ est le plus grand mot vérifiant ces deux propriétés. Enfin, on a $succ(pred_m(w)) = w$.

Si u est le mot n-minimal reçu par MinimalSuivant, alors cet algorithme fabrique en sortie le mot $u^k succ(v)$ avec $k = \lfloor (n-1)/m \rfloor$ et v le préfixe de u de taille $n - km$. En fait, k vaut $\lfloor n/m \rfloor$ sauf si la division tombe juste, auquel cas on a $k = n/m - 1$, et du coup $|u'| = n \bmod m$ sauf si ce dernier est nul, et alors il s'agit de m. On remarque en particulier $k = 0$ ssi $n = m$, et alors $v = u$.

On note $w = succ(v)$ et r le suffixe de u tel que $u = vr$. Soit $m = |w| - |v|$. Du coup $v = pred_m(w)$. On va montrer que $u^k w$ est minimal en considérant une de ses fins f.

- Si $|f| \geq |w|$, on peut déjà remarquer que k ne peut être nul (sinon f ne

pourrait être une fin de w). Il y a alors deux cas :

- Si $|f| - |w|$ est un multiple de $|u|$, alors f est de la forme $u^l w$ avec $0 \leq l < k$. Comme $v \lhd w$ et que u prolonge v, on a $u \lhd w$ puis $u^l u \lhd u^l w$ puis $u^k w \lhd u^l w = f$.
- Si $|f| - |w|$ ne divise pas $|u|$, alors il existe un début g de f qui est également une fin de u. Comme u est minimal, $u < g$, et plus précisément $u \lhd g$, ce qui ne change pas en prolongeant à droite et à gauche : $u^k w \lhd f$.

- Si $|f| < |w|$: On considère alors $g = pred_m(f)$, qui est une fin de v. Du coup gr est une fin de $u = vr$. Par minimalité de u, on a donc $u < gr$, et plus précisément $u \lhd gr$. Il y a alors deux possibilités :
  - Soit la dissemblance se fait avant la fin de g, et donc $u \lhd g$. Si $k \neq 0$ on peut conclure : $u^k w \lhd g \lhd f$. Si $k = 0$, alors $v = u$, et donc $v \lhd g$. Comme $w = succ(v)$ est le plus petit des mots supérieurs à v ayant une taille inférieure à v, on a donc $w \leq g < f$.
  - Soit la dissemblance se fait non pas dans g mais dans r. et donc g est un début de u. Ce cas implique $k \neq 0$, car sinon $r = \varepsilon$. Or $g \lhd f$, donc en prolongeant à droite $u^k w \lhd f$.

On a $u \leq u^k v$ car u est un préfixe de $u^k v$ (sachant que si $k = 0$, alors $u = v$). De plus $v < w$ donc $u^k v < u^k w$, donc au final $u < u^k w$. Montrons enfin que pour tout mot s de taille inférieure à n et tel que $u < s < u^k w$, alors s ne peut être minimal. Déjà, comme $w = succ(v)$ et donc que $u^k w = succ(u^k v)$, et vu les propriétés de *succ* vis-à-vis de l'ordre sur les mots de taille inférieure à $n = |u^k v|$, on a alors $u < s \leq u^k v$. Du coup, $k = 0$ interdit l'existence d'un tel mot s, sinon on aurait $u < s < u$. Considérons maintenant le cas $k \neq 0$. De l'encadrement $u < s \leq u^k v$, on peut déjà affirmer que u est un début de s. Il y a alors deux cas :

- s est un préfixe de $u^k v$. Alors s est de la forme $u^l t$ avec t préfixe de u. Si t est vide, $l > 1$, et u est un début et une fin de s. Si t n'est pas vide, c'est un préfixe de u qui est lui-même un début de s, donc t est un début et une fin de s.
- $s \lhd u^k v$. Alors s est de la forme $u^l t$ avec $t \lhd u$ (ou éventuellement $t \lhd v$, ce qui revient au même). On obtient alors $t \lhd s$ en prolongeant à droite : il existe donc une fin de s qui n'est pas plus grande que s.

1.3 Factorisation en mots minimaux

Soit u un mot non-vide. Une *factorisation minimale* de u est une suite finie de mots minimaux $u_0, \ldots, u_{p-1}$ tels que $u = u_0 \ldots u_{p-1}$ et $u_{i+1} \leq u_i$ pour tout $i < p-1$.

Question 1.13 *Écrire en pseudo-code un algorithme* FACTORISE *permettant d'obtenir une factorisation minimale de u. On pourra partir d'une décomposition de u en mots minimaux et faire progressivement évoluer cette décomposition vers une factorisation minimale. Déterminer la complexité de* FACTORISE

Correction : On part de la factorisation en mots (minimaux) de longueur 1. Puis on exploite la question 1.8 pour concaténer les couples de mots minimaux consécutifs croissants, tant qu'il en reste. Comme le nombre de mots dans la décomposition

décroît à chaque tour, on est sûr de converger, et de répondre finalement une factorisation minimale.

```
let rec une_passe = function
  | u::v::l when estpluspetit u v -> Array.append u v :: une_passe l
  | u :: l -> u :: une_passe l
  | [] -> []

let factorise u =
  let init = List.map (fun lettre -> [|lettre|]) (Array.to_list u) in
  let rec loop l =
    let l' = une_passe l in
    if List.length l' = List.length l then l else loop l'
  in
  loop init
```

On fait au plus $|u|$ passes. Lors d'une passe, le coût des comparaisons est au plus de $|u|$ (vu que chaque portion de $|u|$ peut se retrouver à gauche d'une comparaison), et le coût des concaténations est également au plus $|u|$. D'où un coût total en $O(|u|^2)$.

Parmi les variantes, il est possible de relancer `une_passe` sur `Array.append u v :: l` au lieu de le faire sur `l`. Le nombre de passes nécessaires diminue, et la convergence est sans doute plus rapide. Par contre l'analyse de complexité se corse.

On peut aussi chercher à éviter de comparer de nouveaux des couples de mots déjà rencontrés. Cela peut se faire avec une notion d'âge (jeune/vieux) associé à chaque sous-mot de la décomposition. Là encore, le gain en complexité n'est pas clair.

Question 1.14 *Démontrer que tout mot non-vide possède une unique factorisation minimale.*

Correction : L'existence d'au moins une factorisation minimale est l'objet de la question précédente. Pour un mot non-vide u, montrons maintenant l'unicité d'une telle factorisation minimale par récurrence sur la taille de u. C'est évident pour un mot de taille 1. Sinon, prenons deux factorisations minimales $u = u_1u_2...u_m = v_1v_2...v_p$. Si $|u_1| = |v_1|$, alors $u_1 = v_1$ (ce sont deux préfixes de même taille du même mot). Du coup, $u_2...u_m$ et $v_2...v_p$ sont soit simultanément vides, soit forment deux factorisations minimales d'un même mot de taille strictement inférieure à celle de u : l'hypothèse de récurrence nous permet d'affirmer que ces factorisations coïncident. Au final, les deux factorisations de u coïncident également. Montrons maintenant que nécessairement $|u_1| = |v_1|$, en supposant par l'absurde le contraire, par exemple que $|u_1| > |v_1|$. On a alors $u_1 = v_1...v_{k-1}v$ pour un certain k tel que $1 < k \leq p$ et un certain v préfixe non-vide (mais pas forcément début) de v_k. Alors $v_1 < u_1$ car v_1 est un début de u_1. De plus $u_1 < v$ car v est une fin du mot minimal u_1. Et enfin $v \leq v_k$ car v est un préfixe de v_k. En combinant ces trois inégalités, on obtient $v_1 < v_k$, ce qui contredit la définition d'une factorisation minimale.

Corrigé Partie 2 : Repliage de mots

Pour une lettre α de l'alphabet A_2, on appelle *r*enversement $\overline{\alpha}$ de α la lettre telle que $\overline{a} = b$ et $\overline{b} = a$. On étend cette opération de renversement à tout mot u sur l'alphabet A_2 de la manière suivante :

- $|\overline{u}| = |u|$.
- $\overline{u}[i] = \overline{u[j]}$ pour toute paire d'entiers naturels i et j tels que $i + j = |u| - 1$.

On s'intéresse désormais à une suite particulière de mots sur l'alphabet A_2, que l'on appellera *suite des repliages*. Cette suite $(u_i)_{i\in\mathbb{N}}$ est définie de la manière suivante :

$$\begin{cases} u_0 &= \varepsilon \\ u_{n+1} &= u_n a \overline{u_n} \quad \text{pour tout } n \geq 0 \end{cases}$$

Question 2.1 *Écrire en pseudo-code une procédure* NiemeRepliage *prenant en entrée un entier n et retournant le mot u_n. Donner la complexité de cette procédure.*

Correction :

```
(** Dans cette version, la lettre a est codé par true, et b par false,
    et on représente les mots par des listes de lettres. *)

let rec niemerepliage n =
  if n = 0 then []
  else
    let l = niemerepliage (n-1) in
    l@[true]@List.rev_map not l  (** NB: rev_map = map composé avec rev *)
```

La dernière ligne a un coût proportionnel à $|u_n| = 2^n - 1$. Le coût total de NiemeRepliage est donc une somme de puissance de deux successives, ce qui donne un $O(2^n)$.

2.1 Une caractérisation alternative

Question 2.2 *Montrer que pour tout entier n, le mot u_n est exactement constitué des lettres d'indices impaires du mot u_{n+1} (prises de gauche à droite). Montrer également que les lettres d'indices pairs de u_{n+1} sont une alternance de a et de b. En déduire une procédure* NiemeRepliageBis *basée sur ces constatations et produisant le même résultat que* NiemeRepliage.

Correction : Pour tout n, soit v_n (resp. w_n) le mot formé des lettres d'indices pairs (resp. impaires) de u_n. Comme u_n est de taille $2^n - 1$, on peut en déduire $|v_n| = 2^{n-1}$ et $|w_n| = 2^{n-1} - 1$ (dès que $n > 0$, et $|v_0| = |w_0| = 0$ sinon). Autrement dit,

$$u_n = v_n[0]w_n[0]...w_n[2^{n-1} - 2]v_n[2^{n-1} - 1]$$

Maintenant, dans l'équation de récurrence $u_{n+1} = u_n a \overline{u_n}$, le a central est sur un indice impair dès que $n > 0$. De plus, comme $|u_n|$ est impair pour $n > 0$, les lettres d'indices pairs de $\overline{u_n}$ sont le renversement des lettres d'indices pairs de u_n, et idem pour les indices impairs. Au final, on a les équations de récurrences suivantes lorsque $n > 0$:

$$v_{n+1} = v_n \overline{v_n}$$

$$w_{n+1} = w_n a \overline{w_n}$$

Comme $w_1 = \varepsilon = u_0$, et vu que les équations de récurrences pour (u_n) et (w_n) sont les mêmes, on a donc $\forall n > 0, w_{n+1} = u_n$. Quant à (v_n), les premiers cas sont $v_0 = \varepsilon$ et $v_1 = a$. Ensuite, on peut montrer par récurrence que $\forall n \geq 2, v_n = (ab)^{2^{n-2}}$. En

effet $v_2 = ab$, et si pour un certain $n \geq 2$ on sait que v_n a la forme voulue,

$$v_{n+1} = v_n\overline{v_n} = (ab)^{2^{n-2}}\overline{(ab)^{2^{n-2}}} = (ab)^{2^{n-2}}(\overline{ab})^{2^{n-2}} = (ab)^{2^{n-2}}(ab)^{2^{n-2}} = (ab)^{2^{n-1}}$$

Remarque : attention aux raisonnements faux si on oublie de dissocier le cas $n = 0$. Par exemple on peut être tenté de "prouver" $\forall n, v_n = \varepsilon$ vu que $v_0 = \varepsilon$ et $v_{n+1} = v_n\overline{v_n}$.

```
(** version avec les mots représentés par des tableaux de booleens *)

let rec niemerepliagebis n =
  if n = 0 then [||]
  else
    let a = niemerepliagebis (n-1) in
    let len = Array.length a in
    let a' = Array.make (2*len+1) true in
    for i = 0 to len - 1 do
      a'.(2*i+1) <- a.(i);
      a'.(2*i) <- (i mod 2 = 0);
    done;
    a'.(2*len) <- (len mod 2 = 0);
    a'

(** version avec les mots représentés par des listes de booleens *)

let rec niemerepliagebis' n =
  if n = 0 then []
  else
    let l = niemerepliagebis' (n-1) in
    let rec mix c = function
      | [] -> [c]
      | x::l -> c::x::(mix (not c) l)
    in mix true l
```

2.2 Repliages infinis

Chaque u_n étant un début du mot suivant u_{n+1}, il existe alors un mot infini u_∞ dont tous les u_n sont des préfixes : on peut par exemple le définir via la relation $u_\infty[i] = u_{i+1}[i]$ pour tout entier i.

Question 2.3 *À l'aide de la question précédente, écrire en pseudo-code un algorithme* RepliageInfini *calculant directement $u_\infty[i]$ en fonction de i sans passer par le calcul complet des u_n. Quelle est la complexité de* RepliageInfini *?*

Correction :

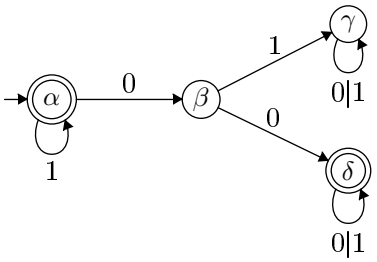
```
let rec repliageinfini i =
 if i mod 4 = 0 then true
 else if i mod 2 = 0 then false
 else repliageinfini (i/2)
```

Si l'algorithme ne s'arrête pas avant, on va effectuer des divisions successives par 2 jusqu'à tomber sur 0. Ceci ne se produit d'ailleurs que si i est de la forme $2^k - 1$. La complexité de RepliageInfini est donc $O(lg(i))$ (nombre de shift droit possible dans un nombre binaire).

Pour les trois prochaines questions, on considère l'alphabet $B = \{0, 1\}$. On identifie tout entier naturel i avec le mot sur l'alphabet B correspondant à l'écriture binaire de i. Pour la question qui vient, les bits de poids faibles sont placés à gauche. Par exemple, le nombre 6 correspond au mot 011, tandis que le nombre 0 correspond au mot vide.

Question 2.4 *Dessiner un automate fini déterministe (de transitions étiquetées par* 0 *et* 1*) acceptant précisément les mots correspondant aux entiers i tels que $u_\infty[i] = a$. Justifier brièvement la correction de cet automate.*

Correction : Vu la question précédente, on cherche en fait à reconnaître les nombres qui, une fois débarrassés de leurs 1 de poids faibles par la partie récursive de l'algorithme précédent, commence alors par au moins deux zéros. Les mots correspondants à ces nombres forment alors le langage rationnel $1^*00(0|1)^*|1^*$. Il ne faut pas oublier en effet l'alternative 1^* : le nombre peut être nul après toutes les divisions par 2. Ceci nous donne l'automate suivant :



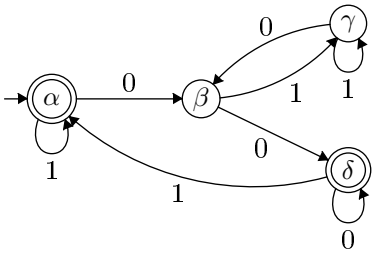
Être dans l'état δ signifie précisément avoir lu un certain nombre de 1, puis exactement deux 0, puis n'importe quoi : cela correspond à la partie $1^*00(0|1)^*$ de la définition du langage rationnel. Quant à la portion 1^*, cela signifie rester dans l'état α. Enfin, l'état γ est un état-puit qu'on peut éventuellement effacer de l'automate.

Remarque : pour cette question et la suivante, la justification attendue est soit un tel descriptif du rôle de chaque état, soit une explication de la construction à partir du langage rationnel (p.ex. méthode des résidus).

On suppose désormais que la représentation des nombres se fait avec les bits de poids faibles à droite (6 correspond maintenant à 110).

Question 2.5 *Dessiner et justifier brièvement l'automate fini déterministe correspond à cette nouvelle situation.*

Correction : Il faut maintenant renverser le sens de lecture : on cherche maintenant à reconnaître des mots du langage $(0|1)^*001^*|1^*$.



L'état α correspond aux mots terminant par 1 dont le dernier groupe de 0 (s'il existe) est au moins de taille 2. C'est donc un état acceptant. L'autre état acceptant δ correspond lui aux mots terminant par au moins deux zéros. Du coté des états

non-acceptant, β correspond aux mots terminant par un 0, mais pas deux, tandis que γ correspond aux mots terminant par 1, et dont le dernier 0 est isolé.

Remarque : même si ce n'est pas vraiment utile ici, on peut noter qu'il existe une méthode générale pour obtenir l'automate (non-déterministe) reconnaissant le langage miroir d'un langage donné.

Pour la question suivante, on fixe une certaine longueur k, et l'on considère des entiers *k-bornés* : n est k-borné si et seulement si $0 \leq n < 2^k$. Une fois choisi un ordre sur les bits, un nombre k-borné peut être identifié avec un mot sur B de longueur k. Par exemple, pour $k = 8$ et les bits de poids faibles à droite, le nombre 6 correspond au mot 00000110. On suppose disposer de plus des opérations primitives suivantes sur ces entiers k-bornés :

- *incr* : ajout de 1 modulo 2^k à un nombre k-borné
- *double* : doublement modulo 2^k d'un nombre k-borné
- *non* : inversion de tous les bits d'un nombre k-borné
- *et* : calcul du "et" bit-à-bit de deux nombres k-bornés
- *estnul* : vérification de la nullité d'un nombre k-borné

On notera que ces définitions sont indépendantes du choix de l'ordre des bits.

Question 2.6 *En composant les seules primitives précédentes, comment peut-on obtenir une fonction qui pour tout nombre k-borné i détermine si $u_\infty[i] = a$ ou non ? On pourra tout d'abord chercher à construire le nombre ayant un bit 1 à l'emplacement du 0 de plus faible poids dans i (s'il existe), et des bits 0 partout ailleurs.*

Correction : On va noter les mots avec les bits de poids faibles à droite, même si cela n'a pas d'influence sur la réponse. Considérons tout d'abord le cas où le mot correspondant à i contient au moins un zéro qui n'est pas tout à gauche : si l'on s'intéresse au zéro le plus à droite, i s'écrit alors $???x01...1$. Dans ce cas, $u_\infty[i]$ vaudra a ssi le bit marqué x est en fait un zéro. On va donc chercher à isoler ce bit :

$$\begin{aligned} i &= ???x01\ldots1 \\ incr(i) &= ???x10\ldots0 \\ non(i) &= \overline{???x}10\ldots0 \\ j := incr(i)\&non(i) &= 0\,..\,010\ldots0 \\ double(j) &= 0\,.\,010\ldots.0 \\ i\&double(j) &= 0\,.\,0x0\ldots.0 \end{aligned}$$

Il ne reste plus qu'à demander si ce dernier nombre est nul ou non. La formule complète est donc :

$$estnul(i\&double(incr(i)\&non(i)))$$

On peut enfin vérifier que lorsqu'il n'y a pas de zéro dans l'écriture du nombre, la formule répond positivement. Enfin, s'il y a un unique zéro tout à la droite du nombre, la réponse est aussi positive.

2.3 Repliages et géométrie

Soit φ la fonction qui à la lettre a associe le nombre complexe i et à la lettre b associe $-i$. À partir de u_∞, on définit une suite de points $(z_n)_{n\in\mathbb{N}}$ du plan complexe comme suit :

$$\begin{cases} z_0 &= 0 \\ z_1 &= 1 \\ z_{n+2} &= z_{n+1} + \varphi(u_\infty[n]) * (z_{n+1} - z_n) \quad \text{pour tout } n \geq 0 \end{cases}$$

On s'intéresse désormais aux propriétés de la courbe C obtenue en reliant les points successifs de cette suite.

Question 2.7 *On suppose donnée une primitive* TraceLigne *prenant les coordonnées cartésiennes (x_1, y_1) et (x_2, y_2) des extrémités d'un segment à afficher. Écrire en pseudo-code un algorithme* TraceCourbe *qui pour un entier n trace la courbe C jusqu'au point z_n.*

Correction :

```
(** on represente les points par des couples d'entiers *)

let rot_trig = function (x,y) -> (-y,x)  (* rotation de +90 deg *)
let rot_antitrig = function (x,y) -> (y,-x)  (* rotation de -90 deg *)
let rot b = if b then rot_trig else rot_antitrig
let add (x,y) (x',y') = (x+x',y+y')
let diff (x,y) (x',y') = (x-x',y-y')

let tracecourbe n =
  let avant = ref (0,0) and actuel = ref (1,0) in
  if n>0 then traceligne !avant !actuel;
  for i=2 to n do
    let direction = repliageinfini (i-2) in
    let apres = add !actuel (rot direction (diff !actuel !avant)) in
    traceligne !actuel apres;
    avant := !actuel;
    actuel := apres;
  done
```

Question 2.8 *Soit p une puissance de deux. Déterminer quelle transformation géométrique permet d'obtenir la portion de C comprise entre z_p et z_{2p} à partir de celle comprise entre z_0 et z_p.*

Correction : Soit n tel que $p = 2^n$. La portion de courbe comprise entre z_0 et z_p fait intervenir p segments z_0z_1 ... $z_{p-1}z_p$ dont les orientations relatives sont déterminés par $p-1$ lettres $u_\infty[0]$ ... $u_\infty[p-2]$. Or on sait que u_n a pour taille $2^n - 1 = p - 1$, et que l'on a $u_{n+1} = u_n a \overline{u_n}$. Ceci indique que l'angle entre $z_{p-1}z_p$ et z_pz_{p+1} est $\varphi(a)$, soit $+90$ degrés. Hormis cela, un parcours de z_p vers z_{2p} va donner lieu précisément aux mêmes virages à droite et à gauche que le parcours de z_p vers z_0. En effet, en "redescendant" de z_p à z_0, il faut non seulement inverser l'ordre des virages, mais aussi leur direction : un virage à droite à l'aller donne un virage à gauche au retour. Les courbes z_pz_{2p} et z_pz_0 sont donc semblables, et la première s'obtient de la seconde par rotation de -90 degrés autour de z_p. Plus précisément, pour $k \leq p$, $z_{p+k} - z_p = -i(z_{p-k} - z_p)$. Si l'on souhaite être tout-à-fait rigoureux,

cette dernière formule peut s'établir par récurrence (sur k, pour un p fixé). On peut remarquer que cela nous donne une première manière d'établir $z_{2p} = (1+i) * z_p$ pour p puissance de 2, et donc $z_p = (1+i)^k$ si $p = 2^k$.

Question 2.9 *Montrer qu'il existe une similitude qui pour tout entier n envoie z_{2n} sur z_n. En déduire en particulier que pour tout entier k, si $p = 2^k$ alors $z_p = (1+i)^k$.*

Correction : Si l'on relie les points d'indice pair z_{2n}, on obtient une courbe ayant la même succession de virages à droite ou à gauche que C. Informellement, cela provient des propriétés vues à la question 2.2. La similitude f recherchée est donc celle qui envoie z_2 sur z_1 tout en fixant $z_0 = 0$, il s'agit donc d'une homothétie de rapport $1/\sqrt{2}$ suivi d'une rotation d'angle -45 degrés. L'expression précise de f est $f(z) = z * (1-i)/2$, ce qui pour des coordonnées cartésiennes $z = x + iy$ donne $f(z) = (x+y)/2 + i * (y-x)/2$.

Montrons que la sous-suite (z_{2n}) suit la même équation de récurrence que la suite principale. Soit $n \leq 0$. On sait d'après les questions précédentes que $\varphi(u_\infty[2n+1]) = \varphi(u_\infty[n])$ et $\varphi(u_\infty[2n+2]) = -\varphi(u_\infty[2n])$. Notons α et β respectivement ces deux quantités.

On a alors :

$$z_{2n+4} - z_{2n+3} = \beta * (z_{2n+3} - z_{2n+2}) \quad (1)$$
$$z_{2n+3} - z_{2n+2} = \alpha * (z_{2n+2} - z_{2n+1}) \quad (2)$$
$$z_{2n+2} - z_{2n+1} = -\beta * (z_{2n+1} - z_{2n}) \quad (3)$$

En utilisant la combinaison linéaire $(1) + (2) * (\beta + 1) + (3) * \alpha * \beta$ des trois équations précédentes, puis en simplifiant (p.ex. par $\beta^2 = -1$), on obtient l'équation de récurrence voulue :

$$z_{2n+4} - z_{2n+2} = \varphi(u_\infty[n]) * (z_{2n+2} - z_{2n})$$

Remarque : on peut aussi traiter séparément les deux cas $\alpha = \beta$ et $\alpha = -\beta$, vu que ces deux cas donnent des systèmes plus simples.

En tout cas, comme $f(z_0) = z_0$ et $f(z_2) = z_1$ et f commute avec l'addition de nombres complexes et la multiplication par un scalaire, il est aisé de montrer que pour tout n, $f(z_{2n}) = z_n$.

Cela permet de retrouver la valeur de z_p pour $p = 2^k$: on a $f^k(z_p) = z_1 = 1$, or $f^{-1}(z) = z * (1+i)$, donc $z_p = (1+i)^k$.

On appelle *point de contact* de la courbe C tout point z pour lequel il existe deux indices $n \neq m$ tels que $z = z_n = z_m$. De tels points de contact existent, par exemple $z_7 = z_{11} = i - 2$.

Question 2.10 *Soit un point de contact $z = z_n = z_m$ avec $n \neq m$. Montrer que n et m sont de même parité. En déduire que si n est non nul, alors $z_{n-1} \neq z_{m+1}$.*

Correction : On montre facilement que $\forall n, z_{n+1} - z_n = \pm i^n$: la progression se fait alternativement à l'horizontale ou verticalement, et toujours de 1 en valeur absolue. On en déduit donc que $Re(z_n) + Im(z_n) = n \ mod\ 2$. Un point de contact ne peut donc se produire que pour des indices de même parité. Ensuite, si n et m sont par exemple pairs, alors $z_{m+1} = z_m \pm 1 = z \pm 1$ tandis que $z_n - z_{n-1} = \pm i$

donc $z_{n-1} = z \pm i$. En fait, quel que soit la parité de n et m, z_{n-1} et z_{m+1} s'écartent de z selon des axes orthogonaux, ils ne peuvent donc être égaux.

Question 2.11 *Soit un point de contact $z = z_n = z_m$ avec $n \neq m$. Montrer que n et m sont non nuls et que les segments $z_{n-1}z_{m-1}$ et $z_{n+1}z_{m+1}$ sont les diagonales d'un carré de centre z.*

Correction : Supposons par l'absurde que $0 = z_0$ soit un point de contact, c'est-à-dire qu'il existe $m > 0$ tel que $z_0 = z_m$. On prend alors m_0 le plus petit de ces m. D'après la question précédente, m_0 est de même parité que 0, il peut donc s'écrire $m_0 = 2q$. On a donc $z_{2q} = 0$. Mais alors, en reprenant la fonction f de la question 2.9, on a $z_q = f(z_{2q}) = f(0) = 0$, ce qui nous donne un nouveau point de contact $z_0 = z_q$ avec $0 < q < m_0$, ce qui contredit la minimalité de m_0, et permet de conclure le raisonnement par l'absurde : 0 ne peut être un point de contact.

On dira qu'un point de contact $z = z_n = z_m$ avec $0 < n < m$ vérifie la propriété du carré lorsque $z_{n+1}z_{m+1}$ et $z_{n-1}z_{m-1}$ sont les diagonales d'un carré de centre z.

Pour un point de contact $z = z_n = z_m$ avec $0 < n < m$, on remarque que $z_{n+1} = z_n \pm i^n$ et $z_{m+1} = z_n \pm i^m$. Vu l'égalité des parités de n et m, on en déduit que soit $z_{n+1} = z_{m+1}$, soit $z_{n+1}z_{m+1}$ est de taille 2 et a pour centre z. On obtient également la même alternative pour $z_{n-1}z_{m-1}$. De plus, lorsque ces deux segments sont non-triviaux, ils sont alors nécessairement orthogonaux. Bref, pour un point de contact $z = z_n = z_m$ avec $0 < n < m$, dire qu'il vérifie la propriété du carré revient à dire que $z_{n-1} \neq z_{m-1}$ et $z_{n+1} \neq z_{m+1}$.

Supposons maintenant par l'absurde qu'il existe des points de contact $z = z_n = z_m$ avec $0 < n < m$ qui ne vérifient pas la propriété du carré, et prenons parmi eux le point de contact tel que (n, m) est minimal selon l'ordre lexicographique. On a alors $z_{n-1} = z_{m-1}$ ou $z_{n+1} = z_{m+1}$ (ou les deux). Mais si la première égalité est vraie, alors $z_{n-1} = z_{m-1}$ est un point de contact avec $n - 1 < m - 1$, et donc $0 < n-1$ d'après la première partie de la question. Et ce point de contact ne vérifie pas non plus la propriété du carré vu que $z_n = z_m$. Ceci contredit le choix minimal de (n, m). On a donc $z_{n-1} \neq z_{m-1}$ mais par contre $z_{n+1} = z_{m+1}$. On distingue alors deux cas selon la parité de n et m.

- Soit n et m sont pairs : $n = 2n'$ et $m = 2m'$. Alors $z_{n'} = f(z_{2n'}) = f(z_{2m'}) = z_{m'}$, et on a donc un nouveau point de contact en $0 < n' < m'$. Comme $(n', m') < (n, m)$, ce point de contact vérifie la propriété du carré, et en particulier $z_{n'+1}z_{m'+1}$ est de taille 2. Après passage par la similitude f^{-1} qui est de rapport $\sqrt{2}$, on obtient que $z_{n+2}z_{m+2}$ doit être de taille $2\sqrt{2}$. Cela est incompatible avec $z_{n+1} = z_{m+1}$, car alors la distance entre z_{n+2} et z_{m+2} devrait être au plus de 2.
- Soit n et m sont impairs : $n + 1 = 2n'$ et $m + 1 = 2m'$. Alors $z_{n'} = f(z_{2n'}) = f(z_{2m'}) = z_{m'}$, et on a donc un nouveau point de contact en $0 < n' < m'$. Maintenant, $n' \leq n$ avec égalité possible pour $n = 1$, tandis que $m' < m$ vu que $0 < n < m$ et donc $1 < m$. Comme on a alors $(n', m') < (n, m)$, ce nouveau point de contact vérifie la propriété du carré, et en particulier $z_{n'-1}z_{m'-1}$ est de taille 2. Après passage par la similitude f^{-1}, on obtient que $z_{n-1}z_{m-1}$ doit être de taille $2\sqrt{2}$. Or on sait déjà que $z_{n-1}z_{m-1}$ est de taille 2 car de part et d'autre de $z_n = z_m$.

Comme les deux cas mènent à des contradictions, on peut donc conclure le raisonnement par l'absurde : tous les points de contact vérifient la propriété du carré. Remarque : cette question mériterait des schémas.

Question 2.12 *En déduire qu'un segment de la courbe C ne peut y apparaître qu'une unique fois, et que C ne passe qu'au plus deux fois par un point donné.*

Correction : Si un segment apparaît deux fois sous la forme $z_n z_{n+1} = z_m z_{m\pm1}$ avec $n \neq m$, alors $z_n = z_m$ est un point de contact, et la question précédente empêche alors d'avoir $z_{n+1} = z_{m\pm1}$.

D'autre part, pour un point donné, il existe quatre segments possibles pour arriver ou partir de ce point (nord/sud/est/ouest). Comme C ne peut passer qu'au plus une fois par un segment donné, la courbe C ne peut donc arriver et repartir qu'au plus deux fois d'un point donné.

* *
*

Note historique. Les mots minimaux de la partie 1 sont également appelés *mots de Lyndon*. L'algorithme MinimalSuivant a été conçu par Duval en 1988. En 1992, Berstel et Pocchiola ont établi que le coût moyen de MinimalSuivant est constant : visiter successivement tous les mots n-minimaux via $M(n)$ utilisations de cet algorithme a un coût proportionnel à $M(n)$ (on ne compte pas ici d'actions de sauvegarde ou d'affichage des mots n-minimaux rencontrés lors de ces itérations). Duval a par ailleurs proposé également un algorithme permettant d'effectuer la factorisation minimale d'un mot en temps linéaire, ce qui permet également de résoudre la question du test de minimalité en temps linéaire.

La courbe C de la partie 2 est nommée *courbe du Dragon* ou *courbe de Harter-Heighway*. Un article de Martin Gardner dans le Scientific American l'a fait plus largement connaître en 1967. Cette courbe fractale possède de nombreuses propriétés d'autosimilarité et de pavage du plan. À noter qu'il est possible de fabriquer son début simplement : il suffit de replier en deux une bande de papier plusieurs fois, toujours dans le même sens, puis d'ouvrir chaque pli à angle droit.