Mines 2017 - Option informatique
Un corrigé

1 Langages et automates

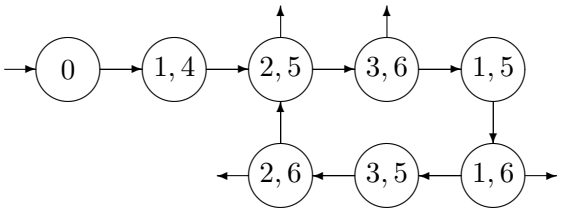
1. $L(\alpha, \beta)$ est fini si et seulement si $\alpha = 0$. De plus, $L(0, \beta) = \{a^\beta\}$ est de cardinal 1.
2. Le langage reconnu par l'automate proposé est $L(4, 2)$.
3. On a cette fois $L_2 = L(3, 2) \cup L(2, 3)$.
4. La table de transition de A_2 est la suivante :

	0	1	2	3	4	5	6
a	1, 4	2	3	1	5	6	5

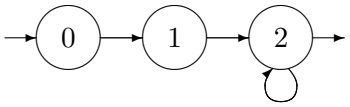
La table du déterminisé est alors (en ne faisant apparaître que les états accessibles)

	0	1, 4	2, 5	3, 6	1, 5	2, 6	3, 5	1, 6
a	1, 4	2, 5	3, 6	1, 5	2, 6	3, 5	1, 6	2, 5

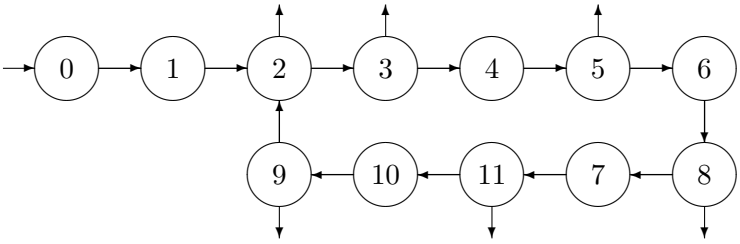
0 est l'état initial et les états terminaux sont ceux contenant 2 ou 6, c'est à dire $(2, 5)$, $(3, 6)$, $(2, 6)$ et $(1, 6)$. On vérifie que tous les états son co-accessibles, c'est à dire que l'automate est émondé. L'automate est le suivant :



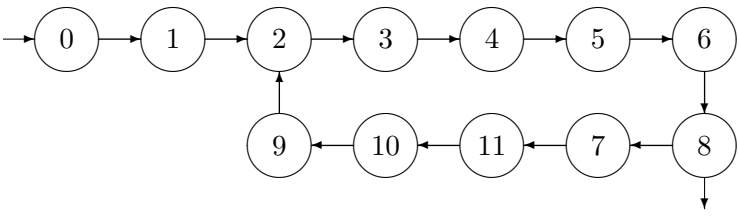
5. Le langage reconnu par A_3 est la réunion des langages des mots menant de l'état initial à chacun des états terminaux :
$$L_3 = L(6, 2) \cup L(6, 3) \cup L(6, 5) \cup L(6, 7)$$
6. L'automate suivant reconnaît $L(1, 2)$ et est de la forme F :



7. On peut envisager une construction comme en questions 3 et 4. Je propose



8. On change les états terminaux



Le langage reconnu par cet automate montre que

$$L(2,3) \cap L(5,2) = L(10,7)$$

Remarque : on pourrait retrouver mathématiquement ce résultat en cherchant les entiers a et b tels que $2a+3 = 5b+2$, ce qui amène à l'équation diophantienne $5b - 2a = 1$.

9. Soit A un automate déterministe émondé dont la fonction de transition est notée δ (c'est une application définie d'une partie de $Q \times \{a\}$ dans Q où Q est l'ensemble des états).
Notons q_0 l'état initial de A. Distinguons deux cas.

   - Si on peut définir la suite $(q_i)_{i\in\mathbb{N}}$ par $q_{i+1} = \delta(q_i, a)$ (pour chaque état atteint, il existe un successeur), ce qui revient à dire (puisque l'automate est émondé et que tous les états sont donc accessibles) que l'automate est complet.
   Comme Q est fini, il existe deux q_i égaux. Notons s le premier s tel que $q_{s+1} \in \{q_0, \dots, q_s\}$ (s existe avec l'hypothèse faite).
   Il existe $r \in \{0, \dots, s\}$ tel que $q_{s+1} = q_r$. L'automate est alors de la forme F avec les mêmes notations que celles qui suivent la question 5.
   - Sinon, on note s le premier entier tel que q_s n'a pas de successeur. L'automate est alors le suivant (sans les états terminaux)

   Notons que puisque l'automate est émondé, q_s doit être terminal. Dans ce cas, le langage est fini et son cardinal égal au nombre des états terminanux.

Si le langage reconnu est infini, on est forcément dans le premier cas et l'automate est de la forme F.

On se donne maintenant un automate de la forme F. Adoptons les notations de l'énoncé dans le dessin qui suit la question 5. Si aucun des états $q_r, \dots, q_s$ n'est terminal alors un mot reconnu est de longueur $\leq r-1$ et le langage reconnu est fini. Sinon, en notant q_i avec $r \leq i \leq s$ un état terminal, tout mot du type $a^{i+k(s-r+1)}$ est reconnu et le langage est infini. Une CNS pour que le langage soit infini est donc qu'il existe un état terminal au moins parmi $q_r, \dots, q_s$.

10. On vient de voir que si L est rationnel unaire infini, il est le langage reconnu par un automate de la forme F et, avec les notations précédentes, qu'il contient $L(s-r+1, i)$.

11. Supposons, par l'absurde, que L soit rationnel. L est infini car $\forall n \geq 1,\ u_{n+1} - u_n > u_1 - u_0 \geq 0$ est donc $u_{n+1} > u_n$ (les a^{u_n} pour $n \geq 1$ sont donc deux à deux distincts). La question précédente donne l'existence de $\alpha \geq 1$ et $\beta > 0$ tels que $L(\alpha, \beta) \subset L$.
Les mots de L (sauf a^{u_0}) classés par longueur strictement croissante sont $a^{u_1}, a^{u_2}, \dots$. Comme $(u_{n+1} - u_n)$ est strictement croissante, on a par récurrence

$$\forall n \in \mathbb{N}^*,\ u_{n+1} - u_n \geq n$$

A partir d'un certain rang n, on aura $u_{n+1} - u_n \geq \alpha + 1$. Il n'existe donc qu'un nombre fini de couples (m, m') de mots de l tels que $|m| - |m'| = \alpha$. ceci contredit à l'évidence $L(\alpha, \beta) \subset L$.

12. $(n+1)^2 - n^2 = 2n+1$ est le terme général d'une suite strictement croissante positive. De plus, (n^2) est une suite d'entiers positifs ou nuls. La question précédente indique que $\{a^{n^2} / \ n \in \mathbb{N}\}$ n'est pas rationnel.

2 Algorithmique et programmation

13. Notons $(n_0, \dots, n_r)$ la suite associée à un calcul de a^n. On montre par récurrence que

$$\forall k \in [|0, r|],\ n_k \leq 2^k$$

- Initialisation : le résultat est vrai initialement car $n_0 = 1 = 2^0$.
- Hérédité : supposons le résultat vrai jusqu'à un rang $k \in [|0, r-1|]$. Il existe alors $i, j \leq k$ tels que $n_{k+1} = n_i + n_j$ et par hypothèse de récurrence,

$$n_{k+1} \leq 2^i + 2^j \leq 2^k + 2^k = 2^{k+1}$$

ce qui prouve le résultat au rang $k+1$.

On en déduit que $n = n_r \leq 2^r$, c'est à dire que $\log_2(n) \leq r$. r étant entier, on a $\lceil \log_2(n) \rceil \leq r$. Un minorant du nombre de multiplications effectuées est donc $\lceil \log_2(n) \rceil$.
Si $n = 2^k$ alors, on peut considérer la suite $1, 2, 4, \dots, 2^{k-1}, 2^k$ qui montre que a^n peut être calculé en $k = \log_2(n) = \lceil \log_2(n) \rceil$ multiplications. Ceci donne une infinité de valeurs pour lesquelles le minorant trouvé est optimal.

14. On a les résultats suivants.
- Pour a^{15} : $(1, 2, 3, 6, 7, 14, 15)$ de longueur 7.
- Pour a^{16} : $(1, 2, 4, 8, 16)$ de longueur 5.
- Pour a^{27} : $(1, 2, 3, 6, 12, 13, 26, 27)$ de longueur 8
- Pour a^{125} : $(1, 2, 3, 6, 7, 14, 15, 30, 31, 62, 124, 125)$ de longueur 12.

15. Si on veut exploiter l'algorithme donné, il me semble que l'on construit la liste dans le mauvais sens (même si l'énoncé n'est pas très clair quand au résultat désiré). En effet, on déduit la liste pour n de celle pour $n/2$ et l'élement n sera, par exemple, le dernier élément ajouté à la liste et se trouvera naturellement en début de liste.
C'est pourquoi, j'écri une fonction auxiliaire récursive construisant la liste dans le mauvais sens en suivant l'algorithme. Il me reste alors à en prendre l'inverse.

```
let rec par_division_aux n =
   if n=1 then [1]
   else begin
      let l=par_division_aux (n/2) in
      let t=hd l in
      if n mod 2 = 0 then (2*t)::l
      else (2*t+1)::(2*t)::l
      end;;

let par_division n = inverse (par_division_aux n);;
```

On pourrait aussi écrire une fonction auxiliaire utilisant un accumulateur pour ne pas avoir recours à `inverse`.

16. Prouvons le résultat par récurrence sur n (ce qui est naturel puisque l'énoncé propose un algorithme récursif). L'hypothèse au rang n est que pour tout a, le calcul de a^n utilise au plus $2 \times \lfloor \log_2(n) \rfloor$ multiplication.
- Si $n = 1$, l'algorithme n'effectue aucune multiplication et on a bien $2 \times \lfloor \log_2(1) \rfloor = 0$.
- Supposons le résultat vrai jusqu'à un rang $n - 1 \geq 1$. Soit $n \geq 2$ et $n = 2q + r$ la division euclidienne de n par 2. Le calcul de a^n se fait de la façon suivante :

on calcule $b = a^q$ ce qui coûte au plus $2 \times \lfloor \log_2(q) \rfloor$ multiplications

on calcule $b \times b$ ce qui coûte une multiplcation

on multiplie éventuellement par a

En notant M_n, le nombre de multiplications, on a alors

$$\begin{aligned} M_n &\leq 2 \times \lfloor \log_2(q) \rfloor + 2 \\ &\leq 2 \times \lfloor \log_2(q) + 1 \rfloor \\ &\leq 2 \times \lfloor \log_2(2q) \rfloor \\ &\leq 2 \times \lfloor \log_2(n) \rfloor \end{aligned}$$

puisque $x \mapsto \lfloor \log_2(x) \rfloor$ est croissante. Ceci prouve le résultat au rang n.

Le cas le pire est celui où l'on tombe toujours sur des exposants impairs. Prenons le cas où $n = 2^k - 1$, qui s'écrit en binaire avec k bits égaux à 1. Notons C_k le nombre de multiplications pour $n = 2^k - 1$. On a $C_1 = 0$ et $C_k = 2 + C_{k-1}$. Ainsi, $C_k = 2(k-1)$ qui est bien égal à $\lfloor \log_2(2^k - 1) \rfloor$.

17. On a les résultats suivants.
    - Pour a^{15} : $(1, 2, 3, 4, 7, 8, 15)$ de longueur 7.
    - Pour a^{16} : $(1, 2, 4, 8, 16)$ de longueur 5.
    - Pour a^{27} : $(1, 2, 3, 4, 8, 11, 16, 27)$ de longueur 8
    - Pour a^{125} : $(1, 2, 4, 5, 8, 13, 16, 29, 32, 61, 64, 125)$ de longueur 12.

18. Si $n = 2q + r$ (division euclidienne), on obtient la décomposition de n en ajoutant le bit r à la décomposition de q.

```
let rec binaire_inverse n=
   if n=0 then []
   else (n mod 2)::(binaire_inverse (n/2));;
```

19. L'algorithme donné par l'énoncé n'est pas très formalisé. Au vu des exemples donnés, la suite voulue contient toutes les puissances de 2 inférieures à n ainsi que les "sommes partielles" dans la décomposition de n. Par exemple,
    - dans le cas où $n = 15 = 2^0 + 2^1 + 2^2 + 2^3$, les sommes partielles sont $1, 3, 7, 15$;
    - dans le cas où $n = 27 = 2^0 + 2^1 + 2^3 + 2^5$, les sommes partielles sont $1, 3, 11, 27$.

    Il me semble donc que pour être efficace, il convient de garder trace de la "puissance courante" de 2 et de la somme partielle. J'écris ainsi une fonction `parcours : int list → int → int → int list`. Dans l'appel `parcours l p s`, `l` est la liste des bits, `p` la puissance de 2 correspondant au premier bit et `s` est l'entier correspondant aux bits déjà consommés. Cette fonction renvoie la suite voulue.
    Il faut bien sûr ne pas ajouter une somme partielle à la liste, par exemple ne pas ajouter deux fois 1 dans le cas de 15 ou de 27.

```
let par_decomposition_binaire n =
    let rec parcours l p s= match l with
      |[] -> []
      |0::q -> p::(parcours q (2*p) s)
      |1::q -> if p+s<>p then p::(p+s)::(parcours q (2*p) (p+s))
               else p::(parcours q (2*p) (p+s))
    in parcours (binaire_inverse n) 1 0;;
```

20. On peut utiliser la suite constituée de $1 = 3^0$ et des $2 \times 3^p, 3^{p+1}$ pour $p = 0, \ldots, k-1$. C'est une suite convenable car 2×3^p s'obtient en sommant deux fois le terme qui le précède et 3^{p+1} en sommant les deux termes qui le précèdent. On obtient une suite de longueur $2k+1$. Dans le cas de 27, c'est la suite

$$(1, 2, 3, 6, 9, 18, 27)$$

On obtient une suite de longueur 7, meilleure que celle de l'algorithme *par_division*.

21. Comme on veut construire une liste, on utilise naturellement une stratégie récursive. On veut ainsi contruit la suite associée à n à partir de celle associée à $n/3$ (division entière bien sûr). C'est facile en ajoutant n et $2 * (n/3)$. Cependant, on construit ainsi la suite "à l'envers" et on doit alors prendre l'image miroir (d'où l'utilisation d'une fonction auxiliaire récursive).

```
let suite_3 n =
  let rec suite n =
   if n=1 then [1]
   else n::(2*(n/3))::(suite (n/3))
 in inverse (suite n);;
```

Comme indiqué plus haut, on pourrait éviter l'utilisation de `inverse` par utilisation d'un accumulateur

```
let suite_3 n =
  let rec suite n accu=
   if n=1 then 1::accu
   else suite (n/3) ((2*(n/3))::n::accu)
 in suite n [];;
```

On pourrait aussi utiliser la construction itérative suggérée plus haut mais cela suppose d'utiliser une référence de liste. Là encore, la construction se fait à l'envers.

```
let suite_3 n =
   let x=ref 1 and l=ref [1] in
   while !x<n do
      l:=(3*(!x))::(2*(!x))::(!l);
      x:=3*(!x)
      done;
   inverse !l;;
```

22. On remarque cette fois que $5^k = 2 \times 2 \times 5^{k-1} + 5^{k-1}$. On peut alors utiliser la suite constituée de 1 et des $2 \times 5^p, 2 \times 2 \times 5^p, 5^{p+1}$ pour $p = 0, \ldots, k-1$. Cette suite est de longueur $3k+1$. Pour $n = 125$, on obtient la suite

$$(1, 2, 4, 5, 10, 20, 25, 50, 100, 125)$$

On obtient une suite de longueur 10 meilleure que celle obtenue avec *par_division*.

23. La suite $(1, 2, 3, 5, 10, 15)$ convient. Les algorithmes précédents ne sont donc pas optimaux (ce que montre aussi la question 20 ou la question 22).

24. Pour $i = k-1$, on cherche un $j \leq i$ tel que $t.(j) + t.(i) = t.(k)$. Si on n'en trouve pas, on cherche avec $k-2$ etc. En faisant les choses dans cet ordre, on obtiendra le plus grand des couples (s'il en existe au moins un).
A i fixé, si on ne trouve pas de j convenable, on finit par obtenir $j = -1$ et il faut alors poursuivre la recherche. Ceci explique la valeur initiale donnée à j.

```
let chercher_indice t k =
   let j=ref (-1) and i=ref (k-1) in
   while !i>=0 && !j=(-1) do
      j:= !i;
      while !j>=0 && t.(!j)+t.(!i)<>t.(k) do decr j done;
      if !j=(-1) then decr i
      done;
   (!j,!i);;
```

La complexité de cette fonction est $O(k^2)$.

25. On utilise un tableau `tab` de flottants de même taille que l'argument `t`. L'idée est de remplir ce tableau afin que `tab.(i)` contienne finalement `x` à la puissance `t.(i)`. Pour cela, on peut utiliser un remplissage de gauche à droite d'après la propriété des suites.

```
let puissance x t =
   let n=vect_length t in
   let tab=make_vect n x in
   for k=1 to (n - 1) do
      let (i,j)=chercher_indice t k in
      tab.(k) <- tab.(i) *. tab.(j)
      done;
   tab.(n-1);;
```

26. Je propose de considérer un algorithme récursif auxiliaire qui prend en argument :
    - l'entier n pour lequel on cherche une suite optimale ;
    - une liste d'éléments *pris* avec les éléments déjà mis dans la suite ;
    - une liste d'élément *aprendre* qui sont les éléments à tester pour compléter la suite.

    On suppose toujours que la liste *pris* est une suite (construite dans le mauvais sens) convenable pour un élément $< n$ (on n'a pas encore gagné).
    On suppose aussi que la liste *aprendre* est non vide et constituée d'éléments tous plus petits que n et strictement plus grands que ceux de *pris* (ce sont des éléments effectivement possibles).
    Avec ces hypothèses, il est TOUJOURS possible de compléter *pris* en commençant par un élément de *aprendre* de façon à obtenir une suite associée à n. En effet, *aprendre* est non vide et contient un élément et on pourra compléter en prenant cet élément x puis $x+1$, $x+2$, etc. (1 est le premier élément de *pris*).
    Le but de la fonction est de renvoyer une complétion optimale avec les contraintes fixées. Il y a deux façons de compléter :
    - en utilisant le premier élément x de *aprendre* et il suffit alors de faire un appel récursif avec $x :: pris$ et les possibles éléments que l'on pourra prendre ensuite (ce sont ceux somme de deux éléments de $x :: pris$, qui sont $> x$ et $\leq n$). ATTENTION, cet appel récursif ne doit être fait que si $x \neq n$ pour respecter les préconditions. Dans le cas où $x = n$, la meilleure complétion est $x :: pris$.
    - sans utiliser cet élément et il s'agit alors de faire un appel récursif avec *pris* et la suite des éléments de *aprendre*. ATTENTION cet appel ne doit être fait que si *aprendre* a au moins deux éléments pour respecter la précondition.

    Il suffit alors d'appeler cette fonction avec $pris = []$ et $aprendre = [1]$.

27. On a besoin d'une première fonction qui donne les suivants potentiels d'une liste (voir ci-dessus). On commence par une fonction `somme` qui à partir de y, x, l et n renvoie la liste de tous les éléments $y + l_i$ qui sont $> x$ et $\leq n$.

```
let rec somme y x l n = match l with
   |[] -> []
   |z::q -> if (y+z)<=n && (y+z)>x then (y+z)::(somme y x q n) else (somme y x q n);;
```

La fonction `suivants` prend en argument l, x et n et renvoie la liste de tous les $l_i + l_j$ qui sont $> x$ et $\leq n$.

```
let rec suivants l x n =
    match l with
   |[] -> []
   |y::q -> union (somme y x l n) (suivants q x n);;
```

Notons l'utilisation de `union` qui donne une "concaténation sans doublon".
La fonction récursive évoquée s'en déduit ainsi que la fonction principale.

```
let rec suite_optimale_rec n pris possible=
     match possible with
     |[x] -> if x=n then n::pris
             else suite_optimale_rec n (x::pris) (suivants (x::pris) x n)
     |x::q -> if x=n then n::pris
              else begin
                 let l1=suite_optimale_rec n (x::pris) (suivants (x::pris) x n)
                 and l2=suite_optimale_rec n pris q
                 in if (longueur l1)<=(longueur l2) then l1 else l2
                 end;;

let suite_optimale n = suite_optimale_rec n [] [1];;
```