Correction - Mines-Ponts 2013

Merci de signaler les erreurs à l'adresse suivante : mickaelpechaud (arobase) gmail (point) com.

Première partie : algorithme simple

1.
```
let est_present m t s =
  let lm = string_length m in
  let b = ref true and i = ref 0 in
  while !b && !i < lm do
    b :=m.[!i]=t.[s+(!i)] ;
    i := !i+1 ;
  done ;
  !b
;;
```

2. Dans le pire des cas, le motif est présent dans le texte. `est_present` est alors de complexité linéaire en la taille du motif.

   Dans le meilleur des cas, le premier test renvoie faux. `est_present` est alors de complexité constante.

3.
```
let positions m t =
  let rec aux m t s =
    if s=(-1) then []
    else
      if est_present m t s
      then s : :(aux m t (s-1))
      else (aux m t (s-1))
  in
  aux m t (string_length t - string_length m)
;;
```

4. Dans le pire des cas, tous les tests sont effectués. C'est par exemple le cas si $m = a^{l(m)}$ et $t = a^{l(n)}$.

   Le nombre de positions testées étant $(l(t) - l(m) + 1)$, le nombre de comparaisons est alors $(l(t) - l(m) + 1)l(m)$.

Deuxième partie : une amélioration de la méthode précédente

5.
```
let calcul_D m nbSigma =
  let d = Array.make nbSigma (-1) in
  for i=0 to string_length m - 1 do
```

```
    d.(numero m.[i])<-i
  done ;
  d
; ;
```

La complexité est linéaire en $l(m)$ – une seule boucle `for` étant utilisée.

6. La lettre $'c'$ n'apparait pas dans le motif. La prochaine position est 9 – qui ne correspond pas à une occurrence du motif dans le mot.
7. À l'indice 13 se trouve la lettre $'a'$ – qui apparait pour la dernière fois en dernière position du motif. La prochaine position sera donc 10 qui correspond à une occurrence du motif.

   À l'indice 14 se trouve la lettre $'b'$ – qui apparait pour la dernière fois en position 1 du motif. Comme positionner la fenêtre en 13 violerait la contrainte $\Phi \leqslant l(t) - l(m)$, l'algorithme termine.
8. La condition donnée par l'énoncé garantit que la prochaine fenêtre ne dépassera pas la fin du texte.

   Si la fenêtre est en position Φ, la lettre immédiatement à droite du motif dans le texte est en position $\Phi + l(m)$ – il s'agit de la lettre $a = t(\Phi + l(m))$.

   La nouvelle position Φ' de la fenêtre doit alors être telle que $\Phi' + D(numero(a)) = \Phi + l(m)$ (cela fonctionne également si la lettre n'apparaît pas dans le motif avec la convention adoptée dans ce cas).

   On en déduit que $\Phi' = \Phi + l(m) - D(numero(t(\Phi + l(m))))$.
9. ```
   let positions2 m t nbSigma =
     let lm = string_length m and lt = string_length t in
     let d = calcul_D m nbSigma in
     let phi = ref 0 in
     let r = ref [] in
     while  !phi+lm <= lt do
       if est_present m t  !phi then
         r := !phi : : !r ;
       let p =  !phi+lm in
       if p < lt then
         phi :=p-d.(numero(t.[p]))
       else
         phi  := lt ;
     done ;
      !r
   ; ;
   ```
10. Dans le meilleur des cas, la lettre suivant la fenêtre n'apparait jamais dans le motif – et de plus `est_present` échoue dès la première comparaison de lettre.

    C'est par exemple le cas si $m = a^{l(m)}$ et $t = b^{l(t)}$.

    Dans ce cas, les positions successives des fenêtres sont $0, l(m) + 1, 2l(m) + 2$ et plus généralement $kl(m) + k$ tant que $kl(m) + k + l(m) \leqslant l(t)$. À chaque position, un nombre constant d'opérations sont effectuées. La complexité dans le meilleur des cas est donc de l'ordre de $O(l(t)/(l(m) + 1))$.
   ```

Troisième partie : implémentation d'un automate

11.
```
let rec estdansliste e l = match l with
    [] -> false
  | t : :q -> t=e || estdansliste e q
; ;

let est_final a p =
  estdansliste p a.f
; ;
```
Cette fonction est de complexité linéaire en le nombre d'état (finaux) de l'automate.

12.
```
let rec arrivee e l = match l with
    [] -> -1
  | (t,a) : :q -> if t=e then a else arrivee e q
; ;


let etat_suivant a p x =
  arrivee x (a.t.(p))
; ;
```
Cette fonction est de complexité linéaire en le nombre de transitions sortantes de l'état p – et donc de complexité linéaire en la taille de l'alphabet, étant donné que l'automate est déterministe.

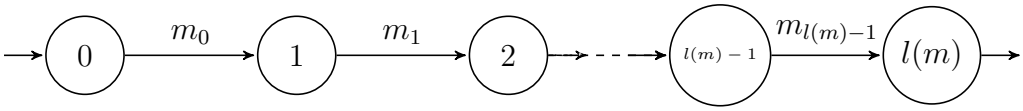
13.
```
let execution a u =
  let lu = string_length u in
  let i = ref 0 and r = ref 0 in
  while !i < lu && !r != (-1) do
    r  := etat_suivant a !r u.[!i] ;
    i := !i+1 ;
  done ;
   !r
; ;
```
Cette fonction est de complexité $O(|\Sigma|l(u))$ d'après la question précédente.

14.
```
let reconnait a u =
  let r = execution a u in
  if r=(-1) then false else est_final a r
; ;
```
Cette fonction est de complexité $O(|\Sigma|l(u) + |F|)$ d'après les questions précédentes.

15.
```
let rec remplace x trans q = match trans with
    [] -> []
  | (c,i) : :t -> if x=c then (x,q) : :t else (c,i) : :(remplace x t q)
; ;
```
La complexités moyenne et au pire de cette fonction sont – ô surprise – linéaires en la longueur de la liste `trans`.

Quatrième partie : utilisation d'automates

16. L'automate suivant convient.



17.
```
let automate_de_mot m =
  let lm = string_length m in
  let t = Array.make (lm+1) [] in
  for i = 0 to lm-1 do
    t.(i) <- [(m.[i],i+1)]
  done ;
  {nbQ = lm+1 ; f = [lm] ; t=t}
 ; ;
```

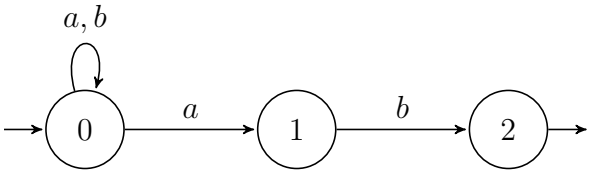
18. Il suffit de vérifier qu'il existe un calcul dans l'automate A_m partant de l'état initial et étiquetté par u.

```
let est_prefixe m u =
  let a = automate_de_mot m in
  execution a u  != (-1)
 ; ;
```

À noter que l'énoncé ne demande pas la complexité de l'algorithme.

19. On a $LS_m = \sigma^*.m$ avec les conventions d'écriture habituelles sur les langages rationnels. LS_m est donc rationnel.

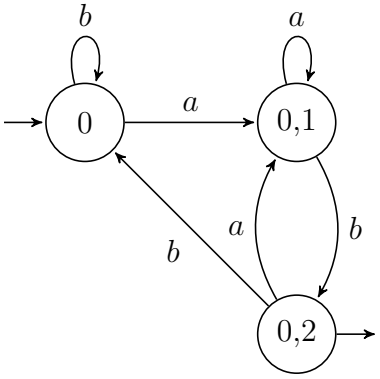
20. Il suffit des transitions allant de l'état initial vers lui-même pour toute lettre de l'alphabet.



21. On utilise l'algorithme classique de déterminisation vu en cours :

	a	b
$\to \{0\}$	$\{0,1\}$	$\{0\}$
$\{0,1\}$	$\{0,1\}$	$\{0,2\}$
$\{0,2\} \to$	$\{0,1\}$	$\{0\}$

On obtient donc l'automate suivant.



22. On part de l'état initial. On lit successivement les lettres du texte (on tient à jour un compteur de la position dans le texte), en suivant la transition correspondante (c'est

toujours possible : l'automate est complet). À chaque fois qu'une transition mène à l'état acceptant, on ajoute la position courante (à laquelle on soustrait $l(m) - 1$ pour renvoyer la position de début du motif, et non pas la position de fin) à la liste des positions où le motif est rencontré.

D'après les questions précédentes, la complexité est effectivement de l'ordre de la longueur du texte.

Prouvons la correction.

Par définition de DS_m, lorsque l'on a lu le texte jusqu'à la lettre i, on est dans l'état final de DS_m Ssi le mot $t_0 \dots t_i$ admet m pour suffixe, i.e. Ssi m apparait dans le texte en position $i - l(m) + 1$.

L'algorithme proposé est donc correct.

23.
```
let positions3 m t =
  let n = string_length m in
  let a = ds m in
  let l = ref [] in
  let pos = ref 0 in
  for i=0 to string_length t - 1 do
    pos  := etat_suivant a  !pos t.[i] ;
    if  !pos = n then l :=(i-n+1) : : !l ;
  done ;
  !l
; ;
```

Cinquième partie : automate des suffixes

24. $h_m(u) = x$ lorsque u termine par x, et $h_m(u) = \epsilon$ sinon.

25. Si $u \in LS_m$, m est un suffixe de u. Comme m est le plus long suffixe de lui-même, on a $m = h_m(u)$.

    Réciproquement, si $m = h_m(u)$, par définition, m est un suffixe de u d'où $u \in LS_m$.

26. $h_m(\epsilon) = \epsilon$, et lorsque u est un préfixe de m, on a $h_m(u) = u$ par définition.

27. Soit u et m deux mots, et x une lettre.

    $h_m(u)$ est le plus long mot à la fois suffixe de u et préfixe de m. $h_m(u)x$ est donc un suffixe de ux, et donc $p = h_m(h_m(u)x)$, qui est un suffixe de $h_m(u)x$, est également un suffixe de ux.

    $p = h_m(h_m(u)x)$ est également un préfixe de m par définition de h_m.

    p est donc préfixe de m et suffixe de ux. Reste à montrer que c'est le plus long.

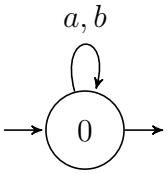
    Soit $p' = vx$ un autre mot préfixe de m et suffixe de ux. v est alors suffixe de u.

    - Si v n'est pas un préfixe de m, nécessairement, $m = vx$, et m est donc suffixe de u. On a alors $h_m(u) = m$, puis $h_m(ux) = h_m(mx)$. L'égalité demandée est alors vraie.
    - Sinon, est préfixe de m. On en déduit que $|v| \leqslant |h_m(u)|$, et que v est un suffixe de $h_m(u)$. vx est donc un suffixe de $h_m(u)x$ – et par définition de v, c'est également un préfixe de m.

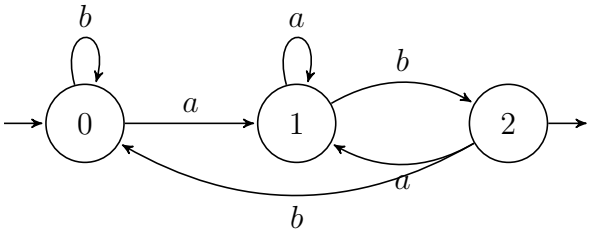
      On en déduit que $|p'| = |vx| \leqslant |h_m(h_m(u)x)| = |p|$.

      Le résultat est donc démontré.

28. S_ϵ est l'automate suivant :



29. S_{ab} est l'automate suivant :



Il s'agit de l'automate de la question 22.

30. On procède par récurrence sur la longueur de u.

Initialisation si $u = \epsilon$, on a $h_m(u) = \epsilon$, $\delta^*(\epsilon, u) = \epsilon$. Le résultat est montré dans ce cas.

Hérédité soit $n \in \mathbb{N}$. Supposons le résultat acquis pour tout mot de longueur n, et soit $u = vx$ un mot de longueur $n + 1$.

Par hypothèse de récurrence, on a $\delta^*(\epsilon, v) = h_m(v)$.

On a

$h_m(u) = h_m(vx) = h_m(h_m(v)x)$ d'après la question 27.

$= h_m(\delta^*(\epsilon, v)x)$ par hypothèse de récurrence

$= \delta(\delta^*(\epsilon, v), x)$ par définition de δ

$= \delta^*(\epsilon, vx) = \delta^*(\epsilon, u)$ par définition e δ^*.

L'Hérédité est donc prouvée.

31. Pour tout mot u,

u est accepté par S_m

Ssi $\delta^*(\epsilon, u) = m$

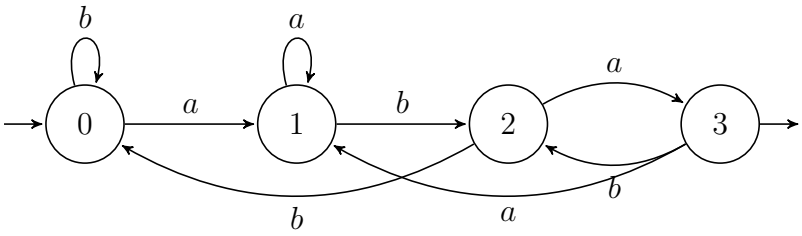
Ssi $h_m(u) = m$ (d'après la question précédente)

Ssi $u \in LS_m$.

32. On note $0 \dots n-1$ les états de S_m. Pour passer de S_m à $S_{m'}$:

- on rajoute un nouvel état $n + 1$ qui va devenir le nouvel état final de l'automate ;
- la transition sortante de $n - 1$ étiquetée par x est redirigés vers n. Toutes les autres transitions de S_m sont préservées ;
- on note j l'état vérifiant $j = \delta(n - 1, x)$. Pour toute transition (j, a, q) de $S_{m'}$, on ajoute la transition (n, s, w) dans $S_{m'}$.

33. L'état noté j dans la question précédente est l'état 1. On obtient donc l'automate suivant :



34. On écrit une fonction qui donne l'automate de la question 28 :

```
let automate_mot_vide () =
  {nbQ=1 ; f=[0] ; t=[|[('a',0) ;('b',0)]|]}
 ; ;
```

La fonction suivante effectue la construction décrite à la question 32 :

```
let ajout_lettre a x =
  let n = Array.length a.t in
  let t = Array.make (n+1) [] in
  for i=0 to n-1 do
    t.(i)<-a.t.(i)
  done ;
  let j = etat_suivant a (n-1) x in
  t.(n-1)<-remplace x a.t.(n-1) n ;
  t.(n)<-t.(j) ;
  {
    nbQ=a.nbQ+1 ; f=[a.nbQ] ; t=t ;
  }
 ; ;
```

La fonction demandée peut alors s'écrire de la façon suivante :

```
let ds m =
  let a = ref (automate_mot_vide ()) in
  for i=0 to string_length m - 1 do
    a :=ajout_lettre ( !a) (m.[i])
  done ;
   !a
 ; ;
```

35. L'énoncé ne me parait pas très précis. Faut-il tenir compte de la complexité de `ds` dans le cas où Σ est quelconque ?

`ajout_lettre` est de complexité $O(n+|\Sigma|)$, où n est le nombre d'état de l'automate. Celui-ci étant (à un près) la longueur du mot considéré.

On en déduit que `ds` est de complexité $O(l(m)^2 + l(m)^2|\Sigma|)$

Une fois l'automate calculé, pour tout texte t, la complexité de `positions3` est $O(|\Sigma|l(t))$ d'après la question 12.

Ne considérant que des automates complets, il serait meilleur de coder les transitions par un tableau de tableaux, et non pas par un tableau de listes. `etat_suivant` serait alors de complexité constante, et `position3` de complexité $\sim l(t)$.