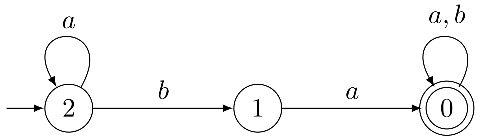
Devoir surveillé n°2

Corrigé

⋆⋆⋆

Question 1 Il n'est pas forcément évident de comprendre ce qui est attendu par « décrire ». On propose ici de décrire le langage par une expression régulière : le langage L_1 est l'interprétation de $(a|b)^*aba^*$. Son langage miroir $\tilde{L}_1$ est l'interprétation de $a^*ba(a|b)^*$.

Question 2 On propose l'automate suivant (on représente les états finaux par des doubles cercles plutôt que par une flèche sortante).



Question 3 On pose $\tilde{A} = (Q, I', F', T')$ où :

- $I' = F$;
- $F' = I$;
- $T' = \{(q, a, p) \mid (p, a, q) \in T\}$, autrement dit on retourne chaque transition.

Montrons que cet automate reconnaît bien $\tilde{L}$:

- si $u = a_0 \ldots a_{n-1} \in L$, alors il existe un chemin étiqueté par u dans A de la forme $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-2}} q_{n-1} \xrightarrow{a_{n-1}} q_n$, avec $q_0 \in I$ et $q_n \in F$. Ainsi, il existe un chemin étiqueté par $\tilde{u}$ dans $\tilde{A}$ de la forme $q_n \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_{n-2}} \ldots \xrightarrow{a_1} q_1 \xrightarrow{a_0} q_0$, avec $q_n \in F = I'$ et $q_0 \in I = F'$. On en déduit que $\tilde{u} \in L_{\tilde{A}}$, soit $\tilde{L} \subseteq L_{\tilde{A}}$;
- on raisonne de même pour le sens réciproque pour montrer que $L_{\tilde{A}} \subseteq \tilde{L}$.

Question 4 On applique ce qui est décrit à la question précédente.

```
let transpose a =
    let retourne (p, a, q) = (q, a, p) in
    {nb = a.nb;
     init = a.final;
     final = a.init;
     trans = List.map retourne a.trans}
```

Question 5 La seule opération qui n'est pas en temps constant est l'appel à `List.map`. La complexité totale est donc linéaire en le nombre de transitions.

Question 6 Il suffit de parcourir le mot jusqu'à la moitié, et de vérifier si chaque lettre est bien égale à sa symétrique.

```
let palindrome w =
    let n = String.length w in
    let i = ref 0 in
    while !i < n / 2 && w.[!i] = w.[n - 1 - !i] do
        incr i
    done;
    !i = n / 2
```

Question 7 Sur un alphabet à une lettre, tout mot est un palindrome. On en déduit que $\mathrm{Pal}(\Sigma) = \Sigma^*$ est bien rationnel.

Question 8 Supposons que Σ contient au moins deux lettres a et b. Supposons par l'absurde que $\mathrm{Pal}(\Sigma)$ est rationnel et soit n sa longueur de pompage. On pose $w = a^n b a^n$. Par le lemme de pompage, il existe une décomposition $w = xyz$ avec :

- $|xy| \leqslant n$;
- $y \neq \varepsilon$;
- pour $k \in \mathbb{N}$, $xy^k z \in \mathrm{Pal}(\Sigma)$.

Cependant, d'après les deux premiers points, $xy \in \mathcal{L}(a^*)$. On en déduit que pour $k = 0$, $xy^k z = xz = a^m b a^n$, avec $m < n$. Ce mot n'est pas un palindrome, ce qui est absurde. On en déduit que $\mathrm{Pal}(\Sigma)$ n'est pas rationnel.

Question 9 Le langage $L_{q,q'}$ est bien reconnaissable, reconnu par l'automate $(Q, \{q\}, \{q'\}, T)$. On peut exprimer L_A comme :

$$L_A = \bigcup_{q \in I} \bigcup_{q' \in F} L_{q,q'}$$

Question 10 On procède par double inclusion :

- $(\subseteq)$: soit $w \in \mathrm{Pal}(\Sigma) \cap (\Sigma^2)^*$. w est donc de taille paire, donc peut s'écrire $w = a_0 a_1 \ldots a_{2n-1}$. Par hypothèse, pour $i \in [\![0, n-1]\!]$, $a_i = a_{2n-1-i}$. En posant $u = a_0 \ldots a_{n-1}$, on en déduit que $w = u\tilde{u}$.
- $(\supseteq)$: soit $u \in \Sigma^*$. Alors sachant que $|\tilde{u}| = |u|$, le mot $w = u\tilde{u} \in (\Sigma^2)^*$. De plus, $\tilde{w} = \widetilde{u\tilde{u}} = \tilde{\tilde{u}}\tilde{u} = u\tilde{u} = w$. On en déduit que $w \in \mathrm{Pal}(\Sigma)$. Remarque : on a utilisé ici le fait que $\widetilde{xy} = \tilde{y}\tilde{x}$.

Question 11 On a $D(a^*b) = \{a^n bba^n \mid n \in \mathbb{N}\}$ et $R(a^*b^*a^*) = a^*b^*$.

En effet, si $u \in a^*b$, $u = a^n b$ pour un certain $n \in \mathbb{N}$ et $u\tilde{u} = a^n bba^n$.

Si $u \in R(a^*b^*a^*)$, alors $u\tilde{u} \in a^*b^*a^*$. Or, tous les palindromes de $a^*b^*a^*$ sont de la forme $a^n b^m a^n$. Ainsi, $u\tilde{u} = a^n b^m a^n$, donc $u = a^n b^{m/2}$.

Question 12 Dans un premier temps, $D(L)$ n'est pas rationnel. On peut faire la même preuve qu'en question 8, en utilisant $a^n bba^n$ comme mot permettant d'arriver à une contradiction.

Par ailleurs, $R(L)$ est rationnel. En effet, on peut exprimer $R(L)$ comme :

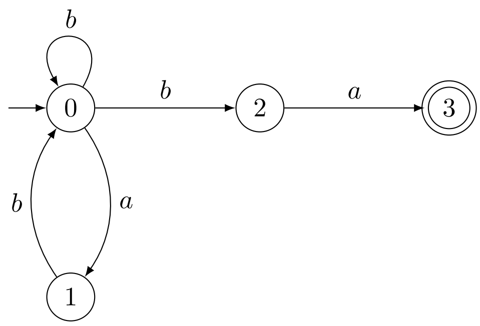
$$R(L) = \bigcup_{p \in I} \bigcup_{q \in F} \bigcup_{r \in Q} L_{p,r} \cap \widetilde{L_{r,q}}$$

En effet, si $w = a_0 \ldots a_{n-1} \in R(L)$, alors $w\tilde{w} \in L$. Il existe donc un chemin $q_0 \xrightarrow{a_0} \ldots \xrightarrow{a_{n-1}} q_n \xrightarrow{a_{n-1}} q_{n+1} \xrightarrow{a_{n-2}} \ldots \xrightarrow{a_0} q_{2n}$ dans A, avec $q_0 \in I$ et $q_{2n} \in F$. En posant $p = q_0$, $q = q_{2n}$ et $r = q_n$, on a bien $w \in L_{p,r}$ et $\tilde{w} \in L_{r,q}$, soit $w \in \widetilde{L_{r,q}}$.

L'inclusion réciproque se fait de manière similaire.

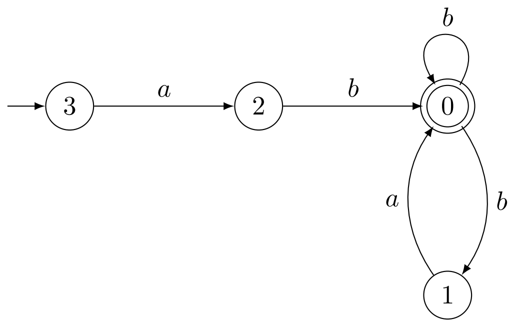
Finalement, on a montré que les $L_{q,q'}$ étaient rationnels, et l'inclusion, l'intersection et le passage au miroir conservent la rationalité, donc $R(L)$ est bien rationnel.

Question 13 On propose :



Pour les deux questions qui suivent, on a choisi de ne pas représenter l'état ensemble vide, même s'il peut techniquement faire partie du déterminisé accessible d'un automate.

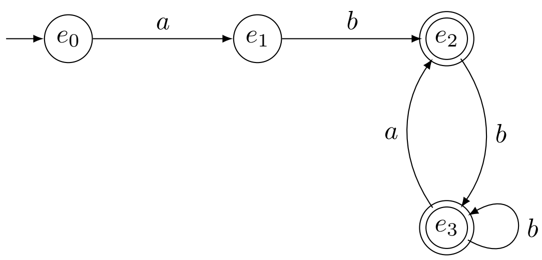
Question 14 L'automate $\widetilde{\mathcal{A}_2}$ est :



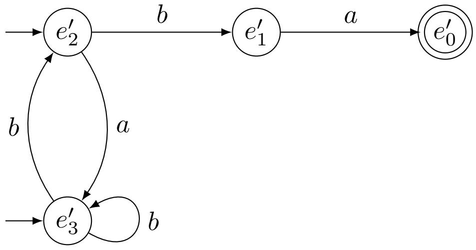
La table de transition de l'automate des parties s'écrit :

Partie	a	b
$e_0 = \{3\}$	2	$\emptyset$
$e_1 = \{2\}$	$\emptyset$	0
$e_2 = \{0\}$	$\emptyset$	$0,1$
$e_3 = \{0,1\}$	0	$0,1$

Les parties finales sont $\{0\}$ et $\{0,1\}$. L'automate est donc :



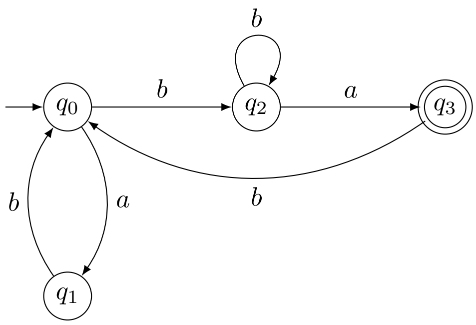
Question 15 L'automate $\widetilde{\mathcal{A}_3}$ est :



La table de transition de l'automate des parties est donc :

Partie	a	b
$q_0 = \{e'_2, e'_3\}$	e'_3	e'_1, e'_2, e'_3
$q_1 = \{e'_3\}$	$\emptyset$	e'_2, e'_3
$q_2 = \{e'_1, e'_2, e'_3\}$	e'_0, e'_3	e'_1, e'_2, e'_3
$q_3 = \{e'_0, e'_3\}$	$\emptyset$	e'_2, e'_3

L'unique partie finale est $\{e'_0, e'_3\}$. L'automate est donc :



Question 16 L'automate $\mathcal{A}_4$ reconnaît le même langage que $\widetilde{\mathcal{A}_3}$, qui reconnaît le miroir du langage de $\mathcal{A}_3$, donc le miroir du miroir du langage de $\mathcal{A}_2$, c'est-à-dire le langage L_2.

Question 17 Comme on ne fait aucune hypothèse sur le type `'a`, on s'attend ici à une solution naïve quadratique (c'est cohérent avec le paragraphe qui précède).

```
let rec supprimer = function
    | [] -> []
    | x :: q ->
        if List.mem x q then
            supprimer q
        else
            x :: supprimer q
```

À noter, on garde ici les dernières occurrences dans la liste.

Question 18 Pour une liste de taille n, l'appel à `List.mem` se fait en temps linéaire en la taille de la queue. La complexité temporelle totale est donc linéaire en $\mathcal{O}\left(\sum_{k=0}^{n-1} k\right) = \mathcal{O}(n^2)$.

Question 19 Il s'agit de tester si le q-ème bit de k est égal à 1 ou non. On fait des divisions et des modulos bien choisis pour obtenir ce résultat. Étant données les hypothèses, ces calculs se font en $\mathcal{O}(1)$. L'ordre des arguments n'étant pas précisé de manière réellement explicite, on choisit de mettre `q` en deuxième pour alléger l'écriture de la fonction `intersecte`.

```
let est_dans k q =
    (k / pow.(q)) mod 2 = 1
```

À noter, on aurait pu utiliser `lsl` pour faire les calculs de puissances de 2 sans le tableau `pow`. Autre remarque : le tableau aurait pu être de taille 20 plutôt que 21 (puisque $n \leqslant 20$).

Question 20 On calcule récursivement le numéro de la queue de la liste, puis on ajoute 2^q si le bit correspondant ne vaut pas 1. L'énoncé suggérait que l'utilisation de `supprimer` n'était pas efficace ici (la fonction est de complexité linéaire plutôt que quadratique).

```
let rec numero = function
    | [] -> 0
    | q :: lst ->
        let k = numero lst in
        if not (est_dans k q) then
            k + pow.(q)
        else
            k
```

Question 21 On se contente de réutiliser la fonction `est_dans`.

```
let intersecte lst k =
    List.exists (est_dans k) lst
```

Question 22 On commence par calculer les images sous forme de listes, puis on calcule leurs numéros.

```
let etat_suivant k trans =
    let lsta, lstb = ref [], ref [] in
    let traiter_transi (p, a, q) =
        if est_dans k p then
            if a = 'a' then lsta := q :: !lsta
                       else lstb := q :: !lstb
    in
    List.iter traiter_transi trans;
    numero !lsta, numero !lstb
```

Question 23 On aurait préféré utiliser des tables de hachage au lieu de listes de couples, mais bon...

```
let rec cherche k = function
    | [] -> -1
    | (l, v) :: q ->
        if k = l then v
        else cherche k q
```

À noter, la fonction demandée est à peu de choses près `List.assoc`.

```
let cherche k lst =
    try List.assoc k lst
    with _ -> -1
```

Question 24 La fonction est assez longue à écrire, il faut bien structurer le code. On écrit ici une sorte de parcours de graphe pour ne traiter que les états accessibles de l'automate des parties.

```
let determinise aut =
    let dict = ref [] and
        trans = ref [] and
        final = ref [] and
        nb = ref 0 in
    let rec parcours k =
        if cherche k !dict = -1 then begin
            dict := (k, !nb) :: !dict; incr nb;
            if intersecte aut.final k then
                final := (!nb - 1) :: !final;
            let ka, kb = etat_suivant k aut.trans in
            trans := (k, 'a', ka) :: (k, 'b', kb) :: !trans;
            parcours ka; parcours kb
        end
    in
    parcours (numero aut.init);
    let renumeroter (k, c, kc) = (cherche k !dict, c, cherche kc !dict) in
    {nb = !nb; init = [0]; final = !final; trans = List.map renumeroter !trans}
```

Dans ce code :

- `dict` garde en mémoire les associations (numéro encodé, numéro final de l'état) ;
- la fonction `parcours` prend en argument un ensemble d'état encodé, ajoute son numéro associé à l'ensemble des états finaux s'il est final, ajoute les transitions de cet état à ses deux voisins, puis relance éventuellement un parcours depuis les voisins s'ils n'ont pas déjà été vus. Les transitions sont stockées en utilisant les numéros d'ensembles d'états
- la fonction `renumeroter` permet de renommer les états dans les transitions. On l'applique une fois le parcours terminé.

Question 25 On détaille les complexités des différentes fonctions :

- `numero` : en $\mathcal{O}(n)$ (ici avec $n \leqslant 20$... cela n'a pas trop de sens d'un point de vue asymptotique) ;
- `intersecte` : en $\mathcal{O}(n^2)$ (ici, la liste d'états sera de taille au plus $|T| \leqslant 2n^2$) ;
- `etat_suivant` : en $\mathcal{O}(|T| + n)$, c'est-à-dire $\mathcal{O}(n^2)$;
- `cherche` : en $\mathcal{O}(N)$;
- `parcours` : pour chaque état de A_{det}, on lance un appel à `intersecte`, un appel à `etat_suivant` et 5 appels à `cherche`, soit une complexité totale en $\mathcal{O}(N(N + n^2))$. Cela correspond à la complexité de `determinise`.

Question 26 Notons $\tilde{\delta}$ la fonction de transition de $\tilde{A}$ (qui est supposé déterministe accessible).

$q \in \delta^*(\{I\}, u)$ si et seulement s'il existe $q_0 \in I$ tel que $q_0 \xrightarrow{u}^* q$ dans l'automate A. On en déduit que $\tilde{\delta}(q, \tilde{u}) \in I$ dans l'automate $\tilde{A}$.

Mais dans l'automate $\tilde{A}$, tous les états sont accessibles. Il existe donc $v \in \Sigma^*$ tel que $\tilde{\delta}(f, v) = q$. Dès lors, $\tilde{\delta}(f, v\tilde{u}) \in I$.

Cela signifie que $v\tilde{u} \in \tilde{L}$, donc $u\tilde{v} \in L$. En posant $w = \tilde{v}$, on obtient le résultat attendu.

Question 27 On a $u^{-1}L = v^{-1}L$ si et seulement si pour $w \in \Sigma^*$, $uw \in L \Leftrightarrow vw \in L$.

Soit alors $q \in \delta^*(\{I\}, u)$ et w le mot de la question précédente tel que $uw \in L$. On a alors $\tilde{\delta}(f, \tilde{w}) = q$. Dès lors :

$$uw \in L \Rightarrow vw \in L \Rightarrow \tilde{w}\tilde{v} \in \tilde{L} \Rightarrow \tilde{\delta}(q, \tilde{v}) \in I \Rightarrow q \in \delta(\{I\}, v)$$

On en déduit que $\delta^*(\{I\}, u) \subseteq \delta^*(\{I\}, v)$. On montre de même que $\delta^*(\{I\}, v) \subseteq \delta^*(\{I\}, u)$.

Question 28 Notons $C = (\tilde{B})_{\text{det}} = (Q_C, \delta_C, q_C, F_C)$. Le fait que C reconnaisse L découle du même argument qu'à la question 16. Le fait que la propriété $(*)$ soit vérifiée découle de la question 27, appliqué à l'automate $\tilde{B}$, dont le miroir (B) est bien déterministe accessible.

Il n'est pas clair que cette question garantisse le caractère minimal de l'automate C. J'apporte donc ici des précisions sur cette propriété (qui n'étaient bien évidemment pas demandées par l'énoncé). Soit $(Q_D, \delta_D, q_D, F_D)$ un automate déterministe complet reconnaissant L.

Supposons par l'absurde que $|Q_D| < |Q_C|$. Pour $q \in Q_C$, notons $u_q \in \Sigma^*$ un mot tel que $\delta_C^*(q_C, u_q) = q$ (existe car les états de C sont accessibles). Par le principe des tiroirs, il existe $p \neq q$ deux états de Q_C tels que $\delta_D^*(q_D, u_p) = \delta_D^*(q_D, u_q)$. Cela implique que $u_p^{-1}L = u_q^{-1}L$, donc d'après la propriété $(*)$ que $p = \delta_C^*(q_C, u_p) = \delta_C^*(q_C, u_q) = q$, ce qui est absurde.

Question 29 Aucune difficulté ici.

```
let minimal aut =
    determinise (transpose (determinise (transpose aut)))
```

On fait abstraction du fait de devoir supprimer l'éventuel état puits (correspondant à l'ensemble vide), puisque l'énoncé demande uniquement d'appliquer la construction de Brzozowski.

Question 30 Un simple parcours de l'expression régulière (ici avec des cas de filtrage regroupés).

```
let rec lettre = function
    | Vide | Epsilon -> 0
    | Lettre _ -> 1
    | Union (e, f) | Concat (e, f) -> lettre e + lettre f
    | Etoile e -> lettre e
```

Question 31 Ici, il y a 4 cas de base (car l'étoile d'un langage n'est jamais vide). Les cas d'induction se traitent facilement.

```
let rec est_vide = function
    | Union (e, f) -> est_vide e && est_vide f
    | Concat (e, f) -> est_vide e || est_vide f
    | e -> e = Vide
```

Question 32 On se contente de coder les règles qui sont rappelées par l'énoncé.

```
let se = function
    | Etoile Vide | Etoile Epsilon -> Epsilon
    | Etoile (Etoile e) -> Etoile e
    | e -> e
```

Question 33 Il y a une application de la règle $E \cdot \varnothing \equiv \varnothing$ par concaténation, et une application de $E + \varnothing \equiv E$, soit $n + 1$ règles au total.

Question 34 On simplifie récursivement les sous-expressions, puis on applique les simplifications à la racine.

```
let rec simplifie = function
    | Union (e, f) -> su (Union (simplifie e, simplifie f))
    | Concat (e, f) -> sc (Concat (simplifie e, simplifie f))
    | Etoile e -> se (Etoile (simplifie e))
    | e -> e
```

Question 35 On applique l'union coefficient par coefficient. On fait attention à ne pas modifier les matrices données en argument.

```
let somme m1 m2 =
    let n = Array.length m1 and p = Array.length m1.(0) in
    let m = Array.make_matrix n p Vide in
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
            m.(i).(j) <- Union (m1.(i).(j), m2.(i).(j))
        done
    done;
    m
```

De par la création de la matrice et la taille des boucles `for`, la complexité est en $\mathcal{O}(n \times p)$ (l'opération dans la boucle étant en temps constant).

Question 36 À nouveau, pas de difficulté, il faut bien appliquer la formule de calcul des coefficients du produit matriciel.

```
let produit m1 m2 =
    let n = Array.length m1 and
        p = Array.length m2 and
        q = Array.length m2.(0) in
    let m = Array.make_matrix n q Vide in
    for i = 0 to n - 1 do
        for j = 0 to q - 1 do
            m.(i).(j) <- Concat (m1.(i).(0), m2.(0).(j));
            for k = 0 to p - 1 do
                m.(i).(j) <- Union (m.(i).(j), Concat (m1.(i).(k), m2.(k).(j)))
            done
        done
    done;
    m
```

La complexité est en $\mathcal{O}(n \times p \times q)$.

Question 37 On remarque que $L_{i,j}$ est l'ensemble des mots permettant d'aller de i à j dans l'automate. On obtient :

- $L_{0,0}$: $(a \mid bd^*c)^*$;
- $L_{0,1}$: $a^*b(d \mid ca^*b)^*$;
- $L_{1,0}$: $d^*c(a \mid bd^*c)^*$;
- $L_{1,1}$: $(d \mid ca^*b)^*$.

Question 38 Étant données les hypothèses de l'énoncé, rien ne garantit que lors des calculs récursifs, le premier coefficient de la matrice sera une lettre seule.

On détaille chacun des calculs :

- `decouper` : en $\mathcal{O}(n^2)$;
- calculs d'étoiles :
  * D^* : en $C(n-1)$;
  * $(a + BD^*C)$: en $\mathcal{O}(1)$ (car matrice de taille 1) ;
  * a^* : en $\mathcal{O}(1)$;
  * $(D + Ca^*B)^*$: en $C(n-1)$;
- produits de matrices : en $\mathcal{O}(n^2)$, car les matrices B et C ont une dimension égale à 1 ;
- sommes de matrices : en $\mathcal{O}(n^2)$, car les matrices sont de taille au plus $(n-1) \times (n-1)$;

– `recoller` : en $\mathcal{O}(n^2)$.

On obtient bien un total de $C(n) = 2C(n-1) + \mathcal{O}(n^2)$.

Pour déterminer la complexité de l'algorithme, on peut diviser par 2^n. On pose $T(n) = \frac{C(n)}{2^n}$. On obtient alors :

$$T(n) - T(n-1) = \mathcal{O}\left(\frac{n^2}{2^n}\right)$$

Comme le terme dans le $\mathcal{O}$ est le terme général d'une série convergente, on en déduit que $T(n)-T(1) = \mathcal{O}(1)$, soit $T(n) = \mathcal{O}(1)$, puis $C(n) = \mathcal{O}(2^n)$.

Question 39 De même :

- Découpage et recollement : en $\mathcal{O}(n^2)$;
- calculs d'étoiles : 4 calculs d'étoiles, avec des matrices de taille $\frac{n}{2} \times \frac{n}{2}$: en $4C\left(\frac{n}{2}\right)$;
- produits de matrices : en $\mathcal{O}(n^3)$ car les matrices sont de dimensions $\mathcal{O}(n)$;
- sommes de matrices : en $\mathcal{O}(n^2)$.

Soit $C(n) = 4C(n/2) + \mathcal{O}(n^3)$.

Sans outil adapté (comme le théorème maître, qui est hors-programme), l'analyse est un peu plus complexe. On peut poser :

- $T(k) = C(2^k)$, ce qui donne $T(k) = 4T(k-1) + \mathcal{O}(8^k)$;
- $U(k) = \frac{T(k)}{4^k}$, ce qui donne $U(k) - U(k-1) = \mathcal{O}(2^k)$.

On obtient alors $U(k) - U(1) = \mathcal{O}\left(\sum_{i=2}^{k} 2^i\right) = \mathcal{O}(2^k)$ (on peut sommer les $\mathcal{O}$, car c'est la même constante de majoration), puis $T(k) = \mathcal{O}(8^k)$ et enfin $C(n) = \mathcal{O}(n^3)$.

Question 40 Une possibilité est d'agrandir la matrice jusqu'à la puissance de 2 supérieure, en rajoutant des coefficients $\varnothing$. Après avoir calculé l'étoile de cette matrice, on peut supprimer les lignes et les colonnes rajoutées (qui vaudront toutes $\varnothing$). Comme la puissance de 2 supérieure vaut moins de $2n$, la complexité temporelle est en $\mathcal{O}(n^3)$ avec le deuxième algorithme (qui est le plus efficace).

Une autre manière de faire est de faire un découpage « à peu près au milieu », même si n n'est pas pair. Par monotonie de la complexité on obtiendra à nouveau du $\mathcal{O}(n^3)$.

Question 41 On peut appliquer l'algorithme précédent (la deuxième version). Ici, le piège est d'appliquer bêtement les formules de l'énoncé, en calculant plusieurs fois A^* et D^* d'une part et A' et D' d'autre part. Il faut faire attention à l'ordre dans lequel on fait les calculs pour ne pas avoir une mauvaise complexité. La fonction n'a pas de grand intérêt à écrire à part ce détail. Notons qu'on aurait pu faire moins de produits de matrice, mais cela ne change epas la complexité.

```
let rec etoile m = match Array.length m with
    | 1 -> [| [|se (Etoile m.(0).(0))|]|]
    | n ->
        let n1 = n / 2 and n2 = n - n / 2 in
        let a, b, c, d = decouper m n1 n2 in
        let a_et = etoile a and d_et = etoile d in
        let a' = etoile (somme a (produit b (produit d_et c))) and
            d' = etoile (somme d (produit c (produit a_et b))) in
        let b' = produit a_et (produit b d') and
            c' = produit d_et (produit c a') in
        recoller a' b' c' d'
```

Question 42 On pose $x_i = \left\{\begin{array}{ll} \varepsilon & \text{si } i \in I \\ \varnothing & \text{sinon} \end{array}\right.$ et $y_j = \left\{\begin{array}{ll} \varepsilon & \text{si } j \in F \\ \varnothing & \text{sinon} \end{array}\right.$.

Alors pour $j \in [\![0, n-1]\!]$, $[XM_A^*]_{0,j} = \sum_{i=0}^{n-1} x_i [M_A^*]_{ij} = \sum_{i \in I} L_{ij}$.

Dès lors, $[XM_A^*Y]_{0,0} = \sum_{j=0}^{n-1} \sum_{i \in I} L_{ij} y_j = \sum_{i \in I} \sum_{j \in F} L_{ij} = L_A$.

Notons qu'on a omis ici l'opérateur d'interprétation.

Question 43 On se contente d'appliquer la question précédente.

```
let langage aut =
    let m = Array.make_matrix aut.nb aut.nb Vide in
    List.iter (fun (i, c, j) -> m.(i).(j) <- su (Union (m.(i).(j), Lettre c))) aut.trans;
    let x = Array.make_matrix 1 aut.nb Vide and
        y = Array.make_matrix aut.nb 1 Vide in
    List.iter (fun i -> x.(0).(i) <- Epsilon) aut.init;
    List.iter (fun j -> y.(j).(0) <- Epsilon) aut.final;
    (produit x (produit (etoile m) y)).(0).(0)
```

La construction de M_A, se fait en $\mathcal{O}(n^2 + |T|)$. La constuction de X et Y se font en $\mathcal{O}(n)$. Le calcul de XM_A^*Y se fait en $\mathcal{O}(n^3)$. La complexité totale est donc en $\mathcal{O}(n^3 + |T|)$. Notons qu'ici, même si la complexité est polynomiale, l'expression régulière obtenue peut être de taille exponentielle (car certaines sous-expressions peuvent apparaître plusieurs fois sans être recalculées).

⋆⋆⋆