Balthazar Patiachvili <balthazar.patiachvili@crans.org>

Problème 2023

Partie I. Préliminaires algorithmiques

Partie 1. Tableaux redimensionnables

Question 1

Voici les différentes étapes d'initialisation d'un tableau auquel on ajoute successivement 3, −2, 4, 1 et −2 :

Étape	Longueur	Capacité	Tableau
0	0	1	
1	1	1	3
2	2	2	3 −2
3	3	4	3 −2 4
4	4	4	3 −2 4 1
5	5	8	3 −2 4 1 −2

Question 2

Parmi les opérations considérées, seule l'opération **ajout à la fin** ne s'effectue pas en temps constant. Plus précisément, si `longueur < capacité`, cette opération s'effectue en temps contant car il ne s'agit que de deux affectations (une à `longueur` et une à `tableau[longueur]`). Cependant, dans le cas `longueur == capacité`, cette opération s'effectue en un temps linéaire : la recopie de l'ancien tableau et les autres affectations prendront un temps $\mathcal{O}(n)$.

Question 3

I_1 correspond au nombre total d'écritures dans `tableau` qui ont été effectuées pour l'écriture d'un élément. Il y a exactement une écriture, à savoir `tableau[0] = x`, donc $I_1 = 1$.

Soient $n \in \mathbb{N}$ et $k \in \mathbb{N}$ l'unique entier tel que $2^{k-1} < n \leqslant 2^k$. I_n est le nombre total d'écritures, on peut les classer dans deux catégories : les nouvelles valeurs ajoutées dans `tableau` et les recopies de celles déjà existantes. Il y exactement n valeurs qui ont été affectées directement, il reste à compter les recopies. Le tableau est recopié à chaque fois qu'une puissance de 2 est dépassée par `longueur` : il y en a exactement k (0, 1, ..., $k-1$). De plus, la recopie d'un tableau de longueur 2^i génère 2^i affectations. Il y a donc $\sum_{i=0}^{k-1} 2^i$ écritures liées à la recopie d'éléments. On a donc : $I_n = n + \sum_{i=0}^{k-1} 2^i$.

Question 4

Soient $n \in \mathbb{N}$ et $k \in \mathbb{N}$ l'unique entier tel que $2^{k-1} < n \leqslant 2^k$.

On a alors d'après la question précédente : $I_n = n + \sum_{i=0}^{k-1} 2^i = n + 1 \times \frac{2^k - 1}{2 - 1} = n + 2^k - 1$.

Avec $2^{k-1} < n$, on a alors : $I_n \leqslant n + 2 \times n - 1 = 3n - 1$.

On a donc : $I_n = \mathcal{O}(n)$, d'où initialiser un tableau puis y insérer n éléments s'effectue en un temps $\mathcal{O}(n)$, les opérations d'affectations étant effectuées en temps constant.

> **Remarque 1**
>
> On peut donc en déduire que l'opération d'**ajout à la fin** pour un tableau redimensionnable s'effectue en temps constant amorti.

Partie 2. Tri lexicographique

Question 5

Le résultat attendu après avoir trié `L` selon l'ordre lexicographique est le suivant :

```python
[
  [1, 0, 3],
  [1, 2, 3],
  [2, 5, 1],
  [4, 2, 1],
  [6, 0, 0]
]
```

Question 6

```python
def plus_petit_k_uple(l1: List[int], l2: List[int]) -> bool:
  n = len(l1)
  for i in range(n):
    if l1[i] < l2[i]:
      return True
    elif l1[i] > l2[i]:
      return False
  return True
```

Question 7

Si on utilise la fonction `plus_petit_k_uple` pour comparer deux listes d'entiers de taille k, on peut programmer une fonction de tri fusion de la manière suivante.

Tout d'abord, on crée une fonction qui fusionne deux listes déjà triées en une nouvelle liste triée en sortie :

```python
def fusion(l1: List[List[int]], l2: List[List[int]]) -> List[List[int]]:
    i1 = 0
    i2 = 0
    l = []

    # On fusionne les listes jusqu'à ce que l'une des deux se finisse
    while i1 < len(l1) and i2 < len(l2):
        if plus_petit_k_uple(l1[i1], l2[i2]):
            l.append(l1[i1])
            i1 += 1
        else:
            l.append(l2[i2])
            i2 += 1

    # On complète la liste créée avec les éléments manquants de l'autre liste
    if i1 != len(l1):
        l += l1[i1:]
    elif i2 != len(l2):
        l += l2[i2:]

    return l
```

La grande boucle `while` est répétée au plus n fois, et chaque tour la seule opération qui ne s'effectue pas en temps constant est `plus_petit_k_uple(l1[i1], l2[i2])` qui prend un temps $\mathcal{O}(k)$. La complétion de la liste `l` par les éléments manquants prend un temps $\mathcal{O}(n)$, donc cette fonction a une complexité temporelle de $\mathcal{O}(n \cdot k)$.

Et voici la fonction de tri fusion complète :

```python
def tri_fusion(l: List[List[int]]) -> List[List[int]]:
    if len(l) <= 1:
        # On copie la liste pour ne pas renvoyer un pointeur vers la même
        return l[:]
    mil = len(l) // 2
    return fusion(tri_fusion(l[:mil]), tri_fusion(l[mil:]))
```

En notant $C(n, k)$ la complexité de cette fonction, où n est la longueur de `l` et k le nombre d'éléments de chaque liste de `l`, on obtient :

$$\forall n \in \{0, 1\}, C(n, k) = \mathcal{O}(1)$$

$$\forall n \in \mathbb{N}, n \geqslant 2 \Rightarrow C(n, k) = \underbrace{2C\left(\left\lfloor \frac{n}{2} \right\rfloor, k\right)}_{\text{appels à } \texttt{tri_fusion}} + \mathcal{O}(n \cdot k)$$

Soient $p \in \mathbb{N}^*$ et $n = 2^p$: on a

$$C(2^p, k) \leqslant 2C(2^{p-1}, k) + A \cdot 2^p \cdot k \qquad \text{avec } A \in \mathbb{R} \text{ une constante}$$
$$C(2^p, k) - 2C(2^{p-1}, k) \leqslant A \cdot 2^p \cdot k$$
$$\frac{C(2^p, k)}{2^p} - \frac{C(2^{p-1}, k)}{2^{p-1}} \leqslant A \cdot k$$
$$\sum_{i=1}^{p} \left(\frac{C(2^i, k)}{2^i} - \frac{C(2^{i-1}, k)}{2^{i-1}} \right) \leqslant \sum_{i=1}^{p} A \cdot k$$
$$\frac{C(2^p, k)}{2^p} - \frac{C(2^0, k)}{2^0} \leqslant A \cdot p \cdot k$$
$$C(n, k) \leqslant n \cdot (A \cdot k \cdot \log(n) + C(1, k))$$

Donc : $C(n, k) = \mathcal{O}(kn \log(n))$.

Donc en utilisant le tri fusion et la fonction `plus_petit_k_uple`, on peut résoudre le problème du tri selon l'ordre lexicographique d'une liste de n listes d'entiers de taille k en un temps $\mathcal{O}(kn \log(n))$.

Question 8

Exécutons pas à pas l'algorithme pour `L = [[2, 5, 1], [6, 0, 0], [1, 2, 3], [1, 0, 3], [4, 2, 1]]` et `j = 1`. On considère que `c = 7` car les éléments vont de 0 à 6.

T	**Reste de L**
`[[], [], [], [], [], [], []]`	`[[2, 5, 1], [6, 0, 0], [1, 2, 3], [1, 0, 3], [4, 2, 1]]`
`[[], [], [], [], [], [[2, 5, 1]], []]`	`[[6, 0, 0], [1, 2, 3], [1, 0, 3], [4, 2, 1]]`
`[[[6, 0, 0]], [], [], [], [], [[2, 5, 1]], []]`	`[[1, 2, 3], [1, 0, 3], [4, 2, 1]]`
`[[[6, 0, 0], [1, 0, 3]], [], [[1, 2, 3], [4, 2, 1]], [], [], [[2, 5, 1]], []]`	`[]`

La sortie de cette exécution est donc : `[[6, 0, 0], [1, 0, 3], [1, 2, 3], [4, 2, 1], [2, 5, 1]]`.

Question 9

```python
def tri_par_cle(L: List[List[int]], j: int, c: int) -> List[List[int]]:
    # Construction de T
    T = [[] for _ in range(c)]
    for l in L:
        T[l[j]].append(l)

    # Concaténation des listes
    res = []
    for k in range(c):
        for l in T[k]:
            res.append(l)

    return res

### Partie II
```

Question 10

Comme vu dans la première partie, toutes les opérations élémentaires sur les tableaux redimensionnables sont effectués en temps constant (amorti pour le cas de l'ajout d'un élément à la fin). Voici le détail des coûts en temps des opérations :

- la création de `T` s'effectue en $\mathcal{O}(c)$
- la première boucle `for` s'effectue en $\mathcal{O}(|\texttt{L}|) = \mathcal{O}(n)$
- la seconde boucle va appeler exactement `L` fois la ligne `res.append(l)` car chaque élément de la liste initiale n'est présent que dans l'une des listes de `T`

La complexité temporelle totale est donc de $\mathcal{O}(c+n)$.

Question 11

Soient $i, i' \in \mathbb{N}$ tels que $0 \leqslant i < i' < n$, $L[i] \neq L[i']$ et $L[i][j] = L[i'][j]$.

Comme $i < i'$, alors $L[i]$ apparaît avant $L[i']$ dans l'énumération des éléments de L par l'algorithme : $L[i]$ va donc être ajouté avant $L[i']$ à la $L[i][j] = L[i'][j]^{\text{ème}}$ case de T, i.e $T[L[i][j]]$. Après concaténation des listes $T[0]$, ..., $T[c-1]$ cette propriété est conservée, on a donc toujours bien que $L[i]$ apparaît avant $L[i']$ dans le résultat de l'exécution K.

Ce tri par clé est donc bien stable.

Question 12

Voici le résultat de l'exécution pas à pas de l'algorithme TriLexicographique sur la liste `[[2, 5, 1], [6, 0, 0], [1, 2, 3], [1, 0, 3], [4, 2, 1]]`.

j	Liste en fin d'itération
	`[[2, 5, 1], [6, 0, 0], [1, 2, 3], [1, 0, 3], [4, 2, 1]]`
2	`[[6, 0, 0], [2, 5, 1], [4, 2, 1], [1, 2, 3], [1, 0, 3]]`
1	`[[6, 0, 0], [1, 0, 3], [4, 2, 1], [1, 2, 3], [2, 5, 1]]`
0	`[[1, 0, 3], [1, 2, 3], [2, 5, 1], [4, 2, 1], [6, 0, 0]]`

Le résultat de cette exécution est donc `[[1, 0, 3], [1, 2, 3], [2, 5, 1], [4, 2, 1], [6, 0, 0]]`.

Question 13

Tout d'abord, comme l'algorithme TriParCle renvoie une permutation de la liste initiale, et que TriLexicographique est une succession d'appels à TriParCle, alors on obtient directement que l'algorithme TriLexicographique renvoie bien une permutation de la liste initiale.

Il reste donc à montrer que les éléments de la liste finale sont bien triés selon l'ordre lexicographique. Soient L une liste de n listes d'entiers de même taille k, et $i, i' \in \mathbb{N}$ tels que $0 \leqslant i, i' < n$ et $L[i] <_{\text{lex}} L[i']$ (avec $<_{\text{lex}}$ l'ordre lexicographique). Montrons que dans la liste finale K, $L[i]$ apparaît avant $L[i']$.

Comme $L[i] <_{\text{lex}} L[i']$, alors $\exists j \in [\![0, k-1]\!]$, tel que $\forall l \in [\![0, j-1]\!], L[i][l] = L[i'][l]$ et $L[i][j] < L[i'][j]$. Durant l'exécution de l'algorithme TriLexicographique, l'appel à TriParCle à l'indice j renverra une liste triée selon l'indice j, c'est-à-dire que $L[i]$ se retrouvera avant $L[i']$ car $L[i][j] < L[i'][j]$. De plus, d'après la question 11, TriParCle est stable donc les prochains appels à TriParCle dans l'exécution de TriLexicographique ne modifiera pas le fait que $L[i]$ apparaît avant $L[i']$ dans la liste résultante, car $\forall l \in [\![0, j-1]\!], L[i][l] = L[i'][l]$ et car TriLexicographique fait des appels à TriParCle par indices décroissants. À la fin de l'algorithme, on aura donc bien $L[i]$ qui apparaît avant

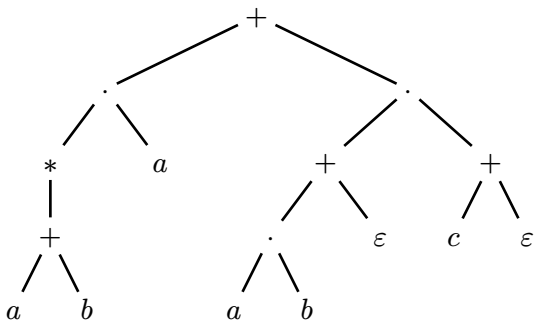
$L[i']$ dans K. Ce résultat est vrai quels que soient les indices de départ i et i', donc cet renvoie bien une permutation de la liste initiale triée par ordre lexicographique.

Partie II. Des expressions rationnelles aux automates

Partie 3. Représenter des expressions rationnelles non vides

Question 14

Voici l'arbre syntaxique correspondant à l'expression rationnelle $(a+b)^*a+(ab+\varepsilon)(c+\varepsilon)$:



Cette même expression peut être représentée en Python de la manière suivante :

```python
["+",
  [".",
    ["*", ["+", ["a"], ["b"]]],
    ["a"]
  ],
  [".",
    ["+", [".", ["a"], ["b"]], [""]],
    ["+", ["c"], [""]]
  ]
]
```

Question 15

```python
def expression(e) -> str:
  if e[0] == "*":
    return f"(%s)*" % (expression(e[1]))
  elif e[0] == "+" or e[0] == ".":
    return f"(%s)%s(%s)" % (expression(e[1]), e[0], expression(e[2]))
  elif e[0] == "":
    return "_"
  else:
    return e[0]
```

Question 16

```python
def contient_mot_vide(e) -> bool:
  if e[0] == "*":
```

```
    return True
  elif e[0] == "+":
    return contient_mot_vide(e[1]) or contient_mot_vide(e[2])
  elif e[0] == ".":
    return contient_mot_vide(e[1]) and contient_mot_vide(e[2])
  elif e[0] == "":
    return True
  else:
    return False
```

Partie 4. Automate de Glushkov

Question 17

```python
def linearisation(e):
  # Construit l'expression linéarisée de exp en indiçant à partir de nb
  # et renvoie l'expression linéarisée et le prochain indice non utilisé
  def aux(exp, nb):
    if exp[0] == "*":
      el, prochain = aux(exp[1], nb)
      return (["*", el], prochain)
    elif exp[0] == "+" or exp[0] == ".":
      e1l, prochain = aux(exp[1], nb)
      e2l, prochain = aux(exp[2], prochain)
      return ([exp[0], e1l, e2l], prochain)
    elif exp[0] == "":
      return ([""], nb)
    else:
      return ([exp[0] + str(nb)], nb + 1)

  el, _ = aux(e, 1)
  return el

### Partie III
```

Question 18

Pour $E = (a+b)^*a + (ab+\varepsilon)(c+\varepsilon)$, on a $E_l = (a_1+b_2)^*a_3 + (a_4b_5+\varepsilon)(c_6+\varepsilon)$ et :

- $\text{First}(E) = \{a_1, b_2, a_3, a_4, c_6\}$
- $\text{Last}(E) = \{a_3, b_5, c_6\}$
- $\text{Null}(E) = \text{vrai}$
- $\text{Follow}(E) = \{a_1a_1, a_1a_3, a_1b_2, b_2a_1, b_2a_3, b_2b_2, a_4b_5, b_5c_6\}$

Question 19

Montrons tout d'abord par induction structurelle sur une expression rationnelle E que :

$$\mathcal{L}(E_l) \setminus \{\varepsilon\} = \underbrace{(\mathrm{First}(E)\Sigma_l^* \cap \Sigma_l^* \,\mathrm{Last}(E)) \setminus \Sigma_l^* \overbrace{(\Sigma_l^2 \setminus \mathrm{Follow}(E))}^{\mathrm{NFollow}(E)} \Sigma_l^*}_{\mathrm{LL}(E)}$$

On observe que pour toute expression rationnelle E, tout mot non vide du langage de son expression linéarisée E_l débute par une lettre de $\mathrm{First}(E)$, finit par une lettre de $\mathrm{Last}(E)$, et ne comporte que des facteurs de taille 2 de $\mathrm{Follow}(E)$. On a donc immédiatement $\mathcal{L}(E_l) \setminus \{\varepsilon\} \subseteq \mathrm{LL}(E)$. Il reste à montrer l'autre inclusion par induction.

- Si $E = \varepsilon$, alors $E_l = \varepsilon$ et les deux quantités sont directement égales à $\emptyset$.
- Si $E = a \in \Sigma$, alors $E_l = a_i \in \Sigma_l$, et on a : $\mathrm{First}(E) = \mathrm{Last}(E) = a_i$, et $\mathrm{Follow}(E) = \emptyset$ donc $\mathcal{L}(E_l) = \{a_i\}$ et $\mathrm{LL}(E) = \{a_i\}$.
- Si $E = E' + E''$ avec E' et E'' des expressions rationnelles que l'on linéarise sur des ensembles d'indices distincts. On a alors : $E_l = E'_l + E''_l$. Par hypothèses d'induction sur E' et E'', on a : $\mathcal{L}(E'_l) \setminus \{\varepsilon\} = \mathrm{LL}(E')$ et $\mathcal{L}(E''_l) \setminus \{\varepsilon\} = \mathrm{LL}(E'')$. De plus, on a :
  - $\mathrm{First}(E) = \mathrm{First}(E') \sqcup \mathrm{First}(E'')$ (avec $\sqcup$ l'union disjointe) car E_l est linéaire donc toutes ses lettres apparaissent au plus une fois, et donc les variables de E' et de E'' sont disjointes
  - $\mathrm{Last}(E) = \mathrm{Last}(E') \sqcup \mathrm{Last}(E'')$ pour les mêmes raisons
  - $\mathrm{Follow}(E) = \mathrm{Follow}(E') \sqcup \mathrm{Follow}(E'')$ pour les mêmes raisons, donc
    $$\begin{aligned}\mathrm{NFollow}(E) &= \Sigma_l^2 \setminus \mathrm{Follow}(E) \\ &= \Sigma_l^2 \setminus (\mathrm{Follow}(E') \sqcup \mathrm{Follow}(E'')) \\ &= (\Sigma_l^2 \setminus \mathrm{Follow}(E')) \cap (\Sigma_l^2 \setminus \mathrm{Follow}(E'')) \\ &= \mathrm{NFollow}(E') \cap \mathrm{NFollow}(E'')\end{aligned}$$
    .
  - $\mathrm{Null}(E) = \mathrm{Null}(E') \vee \mathrm{Null}(E'')$
  - $\mathcal{L}(E_l) \setminus \{\varepsilon\} = (\mathcal{L}(E'_l) \cup \mathcal{L}(E''_l)) \setminus \{\varepsilon\} = (\mathcal{L}(E'_l) \setminus \{\varepsilon\}) \cup (\mathcal{L}(E''_l) \setminus \{\varepsilon\})$

    Donc : $\mathcal{L}(E_l) \setminus \{\varepsilon\} = \mathrm{LL}(E') \cup \mathrm{LL}(E'')$ par hypothèses d'induction.

  Montrons que $\mathrm{LL}(E) \subseteq \mathcal{L}(E_l) \setminus \{\varepsilon\}$, i.e $\mathrm{LL}(E) \subseteq \mathrm{LL}(E') \cup \mathrm{LL}(E'')$.

  Soit $w = w_1 ... w_n \in \mathrm{LL}(E)$, alors $w_1 \in \mathrm{First}(E) = \mathrm{First}(E') \cup \mathrm{First}(E'')$, $w_n \in \mathrm{Last}(E) = \mathrm{Last}(E') \cup \mathrm{Last}(E'')$ et $\forall i \in [\![1, n-1]\!], w_i w_{i+1} \in \mathrm{Follow}(E) = \mathrm{Follow}(E') \cup \mathrm{Follow}(E'')$.

  On suppose sans perdre de généralité que $w_1 \in \mathrm{First}(E')$. On a de plus $w_1 w_2 \in \mathrm{Follow}(E') \cup \mathrm{Follow}(E'')$. Or, comme E_l est linéarisée, alors chaque lettre apparaît au plus une fois dans l'expression, donc w_1 n'apparaît que dans E'_l, et donc w_2 ne peut qu'apparaître dans E'_l également car si $w_2 \in E''_l$, alors $w_1 w_2 \notin \mathrm{Follow}(E)$. On répète alors ce raisonnement pour $i \in [\![0, n-1]\!]$, et on obtient directement que $\forall i \in [\![1, n]\!], w_i$ n'apparaît que dans E'_l. Donc : $\forall i \in [\![0, n-1]\!], w_i w_{i+1} \in \mathrm{Follow}(E')$ et $w_n \in \mathrm{Last}(E')$. On a donc directement : $w \in \mathrm{LL}(E')$.

  Par le même raisonnement, on obtient que si $w_1 \in \mathrm{First}(E'')$, alors $w \in \mathrm{LL}(E'')$.

  Dans tous les cas, on obtient : $w \in \mathrm{LL}(E') \cup \mathrm{LL}(E'')$, d'où $\mathrm{LL}(E) \subseteq \mathrm{LL}(E') \cup \mathrm{LL}(E'')$.
- Si $E = E' \cdot E''$ avec E' et E'' des expressions rationnelles que l'on linéarise sur des ensembles d'indices distincts. On a alors : $E_l = E'_l \cdot E''_l$. Par hypothèses d'induction sur E' et E'', on a : $\mathcal{L}(E'_l) \setminus \{\varepsilon\} = \mathrm{LL}(E')$ et $\mathcal{L}(E''_l) \setminus \{\varepsilon\} = \mathrm{LL}(E'')$. De plus, on a :

- $\mathrm{First}(E) = \begin{cases} \mathrm{First}(E') \sqcup \mathrm{First}(E'') \text{ si } \mathrm{Null}(E') = \text{vrai} \\ \mathrm{First}(E') \text{ si } \mathrm{Null}(E') = \text{faux} \end{cases}$
- $\mathrm{Last}(E) = \begin{cases} \mathrm{Last}(E'') \sqcup \mathrm{First}(E') \text{ si } \mathrm{Null}(E'') = \text{vrai} \\ \mathrm{Last}(E'') \text{ si } \mathrm{Null}(E'') = \text{faux} \end{cases}$
- $\mathrm{Follow}(E) = \mathrm{Follow}(E') \sqcup \mathrm{Follow}(E'') \sqcup \mathrm{Last}(E') \cdot \mathrm{First}(E'')$, donc
  $\begin{aligned} \mathrm{NFollow}(E) &= \Sigma_l^2 \setminus \mathrm{Follow}(E) \\ &= \Sigma_l^2 \setminus (\mathrm{Follow}(E') \cup \mathrm{Follow}(E'') \cup \mathrm{Last}(E') \cdot \mathrm{First}(E'')) \\ &= (\Sigma_l^2 \setminus \mathrm{Follow}(E')) \cap (\Sigma_l^2 \setminus \mathrm{Follow}(E'')) \cap (\Sigma_l^2 \setminus (\mathrm{Last}(E') \cdot \mathrm{First}(E''))) \\ &= \mathrm{NFollow}(E') \cap \mathrm{NFollow}(E'') \cap (\Sigma_l^2 \setminus (\mathrm{Last}(E') \cdot \mathrm{First}(E''))) \end{aligned}$.

  Donc : $\mathrm{NFollow}(E) \subseteq \mathrm{NFollow}(E') \cap \mathrm{NFollow}(E'')$.
- $\mathrm{Null}(E) = \mathrm{Null}(E') \wedge \mathrm{Null}(E'')$
- On fait une disjonction de cas sur $\mathrm{Null}(E')$ et $\mathrm{Null}(E'')$:

$\mathcal{L}(E_l) \setminus \{\varepsilon\} = (\mathcal{L}(E_l') \cdot \mathcal{L}(E_l'')) \setminus \{\varepsilon\}$

$$= \begin{cases} \mathrm{LL}(E') \cdot \mathrm{LL}(E'') \cup \mathrm{LL}(E') \cup \mathrm{LL}(E'') & \text{si } \mathrm{Null}(E') = \text{vrai et } \mathrm{Null}(E'') = \text{vrai} \\ \mathrm{LL}(E') \cdot \mathrm{LL}(E'') \cup \mathrm{LL}(E') & \text{si } \mathrm{Null}(E') = \text{faux et } \mathrm{Null}(E'') = \text{vrai} \\ \mathrm{LL}(E') \cdot \mathrm{LL}(E'') \cup \mathrm{LL}(E'') & \text{si } \mathrm{Null}(E') = \text{vrai et } \mathrm{Null}(E'') = \text{faux} \\ \mathrm{LL}(E') \cdot \mathrm{LL}(E'') & \text{si } \mathrm{Null}(E') = \text{faux et } \mathrm{Null}(E'') = \text{faux} \end{cases}$$

avec $\mathcal{L}(E') \setminus \{\varepsilon\} = \mathrm{LL}(E')$ et $\mathcal{L}(E'') \setminus \{\varepsilon\} = \mathrm{LL}(E'')$.

Montrons que $\mathrm{LL}(E) \subseteq \mathcal{L}(E_l) \setminus \{\varepsilon\}$.

Soit $w = w_1 ... w_n \in \mathrm{LL}(E)$.

On a alors : $w_1 \in \mathrm{First}(E)$, $w_n \in \mathrm{Last}(E)$ et $\forall i \in [\![1, n-1]\!], w_i w_{i+1} \in \mathrm{Follow}(E) = \mathrm{Follow}(E') \cup \mathrm{Follow}(E'') \cup \mathrm{Last}(E') \cdot \mathrm{First}(E'')$.

- Si $\mathrm{Null}(E') = \text{faux}$ et $\mathrm{Null}(E'') = \text{faux}$, alors $\mathcal{L}(E_l) \setminus \{\varepsilon\} = \mathrm{LL}(E') \cdot \mathrm{LL}(E'')$, $w_1 \in \mathrm{First}(E) = \mathrm{First}(E')$ et $w_n \in \mathrm{Last}(E) = \mathrm{Last}(E'')$.

  Par l'absurde, si $\forall i \in [\![1, n-1]\!], w_i w_{i+1} \in \mathrm{Follow}(E') \cup \mathrm{Follow}(E'')$, alors en reprenant le raisonnement du cas $E = E' + E''$, on en déduit que $\forall i \in [\![1, n]\!], w_i \in \Sigma_l'$, en particulier $w_n \in \Sigma_l' \cap \Sigma_l''$ qui sont disjoints : absurde.

  Donc : il existe $k \in [\![1, n-1]\!]$ tel que $w_k w_{k+1} \in \mathrm{Last}(E') \cdot \mathrm{First}(E'')$, c'est-à-dire $w_k \in \mathrm{Last}(E')$ et $w_{k+1} \in \mathrm{First}(E'')$.

  On a alors : $w = \underbrace{w_1}_{\in \mathrm{First}(E')} \cdot w_2 \ldots \underbrace{w_k}_{\in \mathrm{Last}(E')} \cdot \underbrace{w_{k+1}}_{\in \mathrm{First}(E'')} \ldots \underbrace{w_n}_{\in \mathrm{Last}(E'')}$: en reprenant le raisonnement du cas $E = E' + E''$, on a $w_1 ... w_k \in \mathrm{LL}(E')$ en remarquant que $w_1, ..., w_k \in \Sigma_l'$, et $w_{k+1} ... w_n \in \mathrm{LL}(E'')$, en remarquant que $w_{k+1}, ..., w_n \in \Sigma_l''$.

  On a donc : $w \in \mathrm{LL}(E') \cdot \mathrm{LL}(E'') = \mathcal{L}(E_l) \setminus \{\varepsilon\}$.
- Si $\mathrm{Null}(E') = \text{faux}$ et $\mathrm{Null}(E'') = \text{vrai}$, alors $\mathcal{L}(E_l) \setminus \{\varepsilon\} = \mathrm{LL}(E') \cdot \mathrm{LL}(E'') \cup \mathcal{L}(E')$, $w_1 \in \mathrm{First}(E) = \mathrm{First}(E')$ et $w_n \in \mathrm{Last}(E) = \mathrm{Last}(E') \cup \mathrm{Last}(E'')$.

  Si $w_n \in \mathrm{Last}(E'')$, alors on raisonne de la même manière que le 1er cas, et on obtient directement que $w \in \mathrm{LL}(E') \cdot \mathrm{LL}(E'')$.

Si $w_n \in \text{Last}(E')$, alors en reprenant le raisonnement du cas $E = E' + E''$, on conclut que $w_1, ..., w_n \in \Sigma'_l$, donc $w \in \text{LL}(E')$.

Dans tous les cas, on a bien : $w \in \text{LL}(E') \cdot \text{LL}(E'') \cup \text{LL}(E') = \mathcal{L}(E_l) \setminus \{\varepsilon\}$.

- Si $\text{Null}(E') =$ vrai et $\text{Null}(E'') =$ faux, alors on reprend le même raisonnement que le cas précédent car E' et E'' jouent des rôles symétriques.
- Si $\text{Null}(E') =$ vrai et $\text{Null}(E'') =$ vrai, alors en combinant les raisonnement des deux cas précédents, on fait un disjonction de cas selon les ensembles où sont w_1 et w_n.

Dans tous les cas, on a bien : $\text{LL}(E) \subseteq \mathcal{L}(E_l) \setminus \{\varepsilon\}$.

- Si $E = (E')^*$ avec E' une expression rationnelle que l'on linéarise. On a alors : $E_l = (E'_l)^*$. Par hypothèse d'induction sur E', on a : $\mathcal{L}(E'_l) \setminus \{\varepsilon\} = \text{LL}(E')$. De plus, on a :
  - $\text{First}(E) = \text{First}(E')$
  - $\text{Last}(E) = \text{Last}(E')$
  - $\text{Follow}(E) = \text{Follow}(E') \cup \text{Last}(E') \cdot \text{First}(E')$
  - $\text{Null}(E) =$ vrai
  - $\mathcal{L}(E) \setminus \{\varepsilon\} = \mathcal{L}(E')^* \setminus \{\varepsilon\} = (\mathcal{L}(E') \setminus \{\varepsilon\})^* \setminus \{\varepsilon\} = \text{LL}(E')^* \setminus \{\varepsilon\}$

  Montrons que $\text{LL}(E) \subseteq \mathcal{L}(E) \setminus \{\varepsilon\}$, i.e $\text{LL}(E) \subseteq \text{LL}(E')^* \setminus \{\varepsilon\}$.

  Soit $w = w_1...w_n \in \text{LL}(E)$. Comme $w \in \text{LL}(E)$, alors $w_1 \in \text{First}(E) = \text{First}(E')$, et $w_n \in \text{Last}(E) = \text{Last}(E')$. De plus, pour tout $i \in [\![0, n-1]\!]$, on a $w_i w_{i+1} \in \text{Follow}(E) = \text{Follow}(E') \cup \text{Last}(E') \cdot \text{First}(E')$: on découpe le mot w en sous-mots contigus $w_{i_k} w_{i_k+1} ... w_{i_{k+1}-1}$ tels que $\forall i_k \in [\![1, p]\!], w_{i_k} \in \text{First}(E')$ et $w_{i_{k+1}-1} \in \text{Last}(E')$. Ce découpage vérifiant $i_1 = 1$ et $i_p - 1 = n$ est toujours possible car chaque sous-mot $w_{i_k} ... w_{i_{k+1}-1}$ correspond à un mot de $\mathcal{L}(E'_l) \setminus \{\varepsilon\}$, c'est-à-dire $\text{LL}(E')$ par la propriété précédente.

  On a donc : $w = w_1...w_n = \underbrace{\left(w_{i_1} ... w_{i_2-1}\right)}_{\in \text{LL}(E')} ... \underbrace{\left(w_{i_{p-1}} ... w_{i_p-1}\right)}_{\in \text{LL}(E')} \in \text{LL}(E')^*$.

  Donc : $\text{LL}(E) \subseteq \mathcal{L}(E) \setminus \{\varepsilon\}$.

□

Soient E une expression rationnelle et E_l sa linéarisée.

Montrons que $\mathcal{L}(\mathcal{A}_l) = \mathcal{L}(E_l)$ par double inclusion.

- Soit $w = w_1...w_n \in \mathcal{L}(\mathcal{A}_l) \setminus \varepsilon$. De plus, l'état initial de l'automate $\mathcal{A}_l$ est i : la première transition prise sera donc une de la forme $\delta_l(i, w_1)$ car w est accepté par l'automate (et donc il n'y a pas de blocage) et car $i \notin \Sigma_l$. Par définition de δ_l, on a donc $w_1 \in \text{First}(E)$.

  De plus, on a également : $\forall k \in [\![1, n-1]\!], \delta_l(w_i, w_{i+1}) = w_{i+1}$ car il n'y a pas de blocage (car le mot est accepté par l'automate).

  Par définition de $\mathcal{A}_l$, on a alors que $\forall i \in [\![1, n-1]\!], w_i w_{i+1} \in \text{Follow}(E)$.

  Finalement, comme w est accepté par l'automate, alors $w_n \in F$, et $w \neq \varepsilon$ donc $w \in \text{Last}(E)$.

Donc w débute par une lettre dans $\text{First}(E)$, finit par une lettre de $\text{Last}(E)$, et tous ses facteurs de taille 2 sont dans $\text{Follow}(E)$. On a donc $w \in \text{LL}(E)$, d'où $w \in \mathcal{L}(E_l) \setminus \{\varepsilon\}$ par la propriété ci-dessus.

- Soit $w \in \mathcal{L}(E_l) \setminus \{\varepsilon\}$. Alors, par la propriété précédente, on a $w \in \text{LL}(E)$, c'est-à-dire que w débute par une lettre de $\text{First}(E)$, termine par une lettre de $\text{Last}(E)$ et ne contient que des facteurs de taille 2 de $\text{Follow}(E)$. On remarque qu'un tel mot est directement accepté par $\mathcal{A}_l$, d'où $\mathcal{L}(E_l) \setminus \{\varepsilon\} \subseteq \mathcal{L}(\mathcal{A}_l)$.

De plus, si $\text{Null}(E) = \text{vrai}$, alors $\mathcal{A}_l$ reconnaît le mot vide car $i \in F$, et inversement : si $\mathcal{A}_l$ reconnaît le mot vide, alors on a forcément $i \in F$, ce qui correspond au cas où $\text{Null}(E) = \text{vrai}$.

Dans tous les cas, on a donc : $\mathcal{L}(E_l) = \mathcal{L}(\mathcal{A}_l)$.

Question 20

Par définition de $\mathcal{A}_l$, toutes les transitions vers un état $x \in \Sigma_l$ sont étiquetées par x, que ce soient les transitions de l'état initial i ou celles induites par $\text{Follow}(E)$. Donc dans $\mathcal{A}_l$ toutes les transitions vers un état portent la même étiquette. Supprimer les indices des étiquettes ne change pas ce fait : toutes les transitions de $\mathcal{A}$ arrivant à un état $x \in \Sigma_l$ portent la même étiquette x à laquelle on a retiré son indice.

Question 21

Montrons tout d'abord par induction sur E l'équivalence suivante :

$$\forall w \in \Sigma^*, \quad w = a^{(1)}...a^{(n)} \in \mathcal{L}(E) \iff \exists i_1, ..., i_n \in \mathbb{N}, a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l)$$

avec $n \in \mathbb{N}$, $a^{(1)}, ..., a^{(n)} \in \Sigma$ et $a_{i_1}^{(1)}, ..., a_{i_n}^{(n)} \in \Sigma_l$.

- Si $E = \varepsilon$, alors $E_l = \varepsilon$ donc leurs langages sont égaux, et l'équivalence est immédiate.
- Si $E = a \in \Sigma$, alors $E_l = a_1 \in \Sigma_l$, $\mathcal{L}(E) = \{a\}$ et $\mathcal{L}(E_l) = \{a_1\}$ et l'équivalence est immédiate.
- Si $E = E' + E''$, alors on raisonne par double implication. On fait attention à linéariser E' et E'' en E_l' et E_l'' sur des ensembles d'indices distincts pour conserver le caractère linéaire de E. On a alors : $E_l = E_l' + E_l''$.
  - Soit $w = a^{(1)}...a^{(n)} \in \mathcal{L}(E) = \mathcal{L}(E') \cup \mathcal{L}(E'')$. Sans perdre de généralité, on suppose $w \in \mathcal{L}(E')$: par hypothèse d'induction sur E', $\exists i_1, ..., i_n \in \mathbb{N}, a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l') \subseteq \mathcal{L}(E_l) = \mathcal{L}(E_l') \cup \mathcal{L}(E_l'')$. Donc $a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l)$. On raisonne de la même manière pour E''.
  - Soit $\overline{w} = a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l) = \mathcal{L}(E_l') \cup \mathcal{L}(E_l'')$. Sans perte de généralité, on suppose que $\overline{w} \in \mathcal{L}(E_l')$, par hypothèse d'induction sur E', on a alors $w = a^{(1)}...a^{(n)} \in \mathcal{L}(E') \subseteq \mathcal{L}(E)$. On raisonne de la même manière pour E''.

  On a donc l'équivalence.
- Si $E = E' \cdot E''$, alors on raisonne par double implication. On fait attention à linéariser E' et E'' en E_l' et E_l'' sur des ensembles d'indices distincts pour conserver le caractère linéaire de E. On a alors : $E_l = E_l' \cdot E_l''$.
  - Soit $w = a^{(1)}...a^{(n)} \in \mathcal{L}(E) = \mathcal{L}(E') \cdot \mathcal{L}(E'')$. Donc, $\exists k \in [\![0, n]\!], a^{(1)}...a^{(k)} \in \mathcal{L}(E')$ et $a^{(k+1)}...a^{(n)} \in \mathcal{L}(E'')$. Par hypothèses d'induction sur E' et E'', on a directement $\exists i_1, ..., i_n \in \mathbb{N}, a_{i_1}^{(1)}...a_{i_k}^{(k)} \in \mathcal{L}(E_l')$ et $a_{i_{k+1}}^{(k+1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l'')$. On a donc : $a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l') \cdot \mathcal{L}(E_l'') = \mathcal{L}(E_l)$.

- Soit $\overline{w} = a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l) = \mathcal{L}(E_l') \cdot \mathcal{L}(E_l'')$. Donc, $\exists k \in [\![0, n]\!]$, $a_{i_1}^{(1)}...a_{i_k}^{(k)} \in \mathcal{L}(E_l')$ et $a_{i_{k+1}}^{(k+1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l'')$. Par hypothèses d'induction sur E' et E'', on a directement $a^{(1)}...a^{(k)} \in \mathcal{L}(E')$ et $a^{(k+1)}...a^{(n)} \in \mathcal{L}(E'')$. On a donc : $a^{(1)}...a^{(n)} \in \mathcal{L}(E') \cdot \mathcal{L}(E'') = \mathcal{L}(E)$.

  On a donc l'équivalence.

- Si $E = (E')^*$, alors on raisonne par double implication. On linéarise E' en E_l', et on obtient alors $E_l = (E_l')^*$.
  - Soit $w = a^{(1)}...a^{(n)} \in \mathcal{L}(E) = \mathcal{L}(E')^*$. Donc, $\exists\ 0 < k_1 < k_2 < ... < k_p < n$ tels que $a^{(1)}...a^{(k_1)}, a^{(k_1+1)}...a^{(k_2)}, ..., a^{(k_{p-1}+1)}...a^{(k_p)} \in \mathcal{L}(E')$. Par hypothèses d'induction sur tous les sous-mots entre les indices $k_1, k_2, ..., k_p$, on a alors : $\exists i_1, ..., i_n \in \mathbb{N}$ tels que $a_{i_1}^{(1)}...a_{i_{k_1}}^{(k_1)}, a_{i_{k_1+1}}^{(k_1+1)}...a_{k_2}^{(k_2)}, ..., a_{i_{k_{p-1}+1}}^{(k_{p-1}+1)}...a_{i_{k_p}}^{(k_p)} \in \mathcal{L}(E_l')$, donc $\left(a_{i_1}^{(1)}...a_{i_{k_1}}^{(k_1)}\right) \cdot \left(a_{i_{k_1+1}}^{(k_1+1)}...a_{k_2}^{(k_2)}\right) \cdot ... \cdot \left(a_{i_{k_{p-1}+1}}^{(k_{p-1}+1)}...a_{i_{k_p}}^{(k_p)}\right) \in \mathcal{L}(E_l')^* = \mathcal{L}(E_l)$.
  - Soit $\overline{w} = a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l) = \mathcal{L}(E_l')^*$. Donc, $\exists\ 0 < k_1 < k_2 < ... < k_p < n$ tels que $a_{i_1}^{(1)}...a_{i_{k_1}}^{(k_1)}, a_{i_{k_1+1}}^{(k_1+1)}...a_{i_{k_2}}^{(k_2)}, ..., a_{i_{k_{p-1}+1}}^{(k_{p-1}+1)}...a_{i_{k_p}}^{(k_p)} \in \mathcal{L}(E')$. Par hypothèses d'induction sur tous les sous-mots entre les indices $k_1, k_2, ..., k_p$, on a alors : $a^{(1)}...a^{(k_1)}, a^{(k_1+1)}...a^{(k_2)}, ..., a^{(k_{p-1}+1)}...a^{(k_p)} \in \mathcal{L}(E')$, donc $\left(a^{(1)}...a^{(k_1)}\right) \cdot \left(a^{(k_1+1)}...a^{(k_2)}\right) \cdot ... \cdot \left(a^{(k_{p-1}+1)}...a^{(k_p)}\right) \in \mathcal{L}(E')^* = \mathcal{L}(E)$.

  On a donc l'équivalence.

□

Montrons que $\mathcal{L}(E) = \mathcal{L}(\mathcal{A})$ par double inclusion.

- Soit $w = a^{(1)}...a^{(n)} \in \mathcal{L}(E)$. D'après la propriété ci-dessus, il existe $i_1, ..., i_n \in \mathbb{N}$ tels que $\overline{w} := a_{i_1}^{(1)}...a_{i_n}^{(n)} \in \mathcal{L}(E_l)$. D'après la question 19, on a $\mathcal{L}(E_l) = \mathcal{L}(\mathcal{A}_l)$, d'où $\overline{w} \in \mathcal{L}(\mathcal{A}_l)$. On considère l'exécution de l'automate $\mathcal{A}_l$ sur le mot $\overline{w}$: en supprimant tous les indices des transitions de $\mathcal{A}_l$ et des lettres de $\overline{w}$, on obtient alors une exécution valide de l'automate non déterministe $\mathcal{A}$ sur le mot w. Comme il existe une exécution de l'automate $\mathcal{A}$ acceptant w (car $\mathcal{A}_l$ accepte $\overline{w}$) alors $w \in \mathcal{L}(\mathcal{A})$.
- Soit $w = a^{(1)}...a^{(n)} \in \mathcal{L}(\mathcal{A})$.

  Si $n = 0$, alors $w = \varepsilon$ et $i \in F$, donc $\text{Null}(E) = \text{vrai}$, d'où $w = \varepsilon \in \mathcal{L}(E)$.

  Sinon, comme w est accepté par $\mathcal{A}$ (qui est à priori non déterministe), alors il existe une exécution de l'automate $\mathcal{A}$ passant par les états $q_1 = i, ..., q_{n+1}$ telle que $\forall i \in [\![1, n]\!]$, $q_{i+1} \in \delta\left(q_i, a^{(i)}\right)$. En considérant la même exécution sur $\mathcal{A}_l$ passant par les états $q_1, ..., q_{n+1}$, on obtient en concaténant les étiquettes des transitions prises (qui sont uniques entre deux états donc sans ambiguïté) un mot $\overline{w}$ tel que $\exists i_1, ..., i_n \in \mathbb{N}$, $\overline{w} = a_{i_1}^{(1)}...a_{i_n}^{(n)}$. On a $\overline{w} \in \mathcal{L}(A)_l$, donc $\overline{w} \in \mathcal{L}(E_l)$ d'après la question 19, donc $w \in \mathcal{L}(E)$ d'après la propriété ci-dessus.

On a donc bien : $\mathcal{L}(\mathcal{A}) = \mathcal{L}(E)$.

Question 22

Il y a autant d'états dans $\mathcal{A}$ que dans $\mathcal{A}_l$ (car ce sont les mêmes), et il y a exactement 1 état de plus dans $\mathcal{A}_l$ qu'il n'y a de lettres dans l'alphabet Σ_l. Or, comme E_l est l'expression linéarisée, alors chaque lettre de E_l est utilisée exactement une fois, donc $\mathcal{A}$ possède autant d'états qu'il n'y a de lettres dans

E_l, et donc dans E, plus 1. On peut alors directement en conclure, étant donné la définition de $|\cdot|$, que $|\mathcal{A}| \leqslant |E| + 1$, avec $|\mathcal{A}|$ le nombre d'états de l'automate $\mathcal{A}$.

Question 23

Voici le déroulé de l'exécution de la fonction `glushkov` sur l'entrée `["+", ["*", ["·", ["a1"], ["b2"]]], ["·", ["a3"], ["a4"]]]` :

- `glushkov(["+", ["*", ["·", ["a1"], ["b2"]]], ["·", ["a3"], ["a4"]]])`

  La liste passée en paramètre a une longueur de 3 et débute par « + », deux appels récursifs vont donc être effectués.

  - `glushkov(["*", ["·", ["a1"], ["b2"]]])`

    La liste en paramètre a une longueur de 2 donc débute par « ∗ », un appel récursif va donc être effectué.

    - `glushkov(["·", ["a1"], ["b2"]])`

      La liste passée en paramètre a une longueur de 3 et débute par « · », deux appels récursifs vont donc être effectués.

      - `glushkov(["a1"])`

        La liste passée en paramètre contient uniquement une lettre : les ensembles First, Last et Follow, et le booléen Null sont directement renvoyés car ils sont respectivement égaux à $\{a_1\}$, $\{a_1\}$, $\emptyset$ et faux.

      - `glushkov(["b2"])`

        Même raisonnement que pour `glushkov(["a1"])`.

      On peut maintenant calculer les ensembles First, Last et Follow, ainsi que le booléen Null pour l'expression passée en paramètre, ceux-ci étant induits par les valeurs renvoyées par les appels récursifs selon les formules données dans le programme ou dans la question 19. First, Last, Follow et Null valent respectivement $\{a_1\}$, $\{b_2\}$, $\{a_1b_2\}$ et faux.

    On peut maintenant déterminer les valeurs de First, Last, Follow et Null pour l'expression entière, correspondant à $(a_1 \cdot b_2)^*$. Ceux-ci valent respectivement $\{a_1\}$, $\{b_2\}$, $\{a_1b_2, b_2a_1\}$ et vrai.

  - `glushkov(["·", ["a3"], ["a4"])`

    De la même façon que pour `["·", ["a1"], ["b2"]]`, on fait des appels récursifs sur `["a3"]` et `["a4"]`, et on renvoie First, Last, Follow et Null qui valent respectivement $\{a_3\}$, $\{a_4\}$, $\{a_3a_4\}$ et faux.

  On peut maintenant déterminer les valeurs de First, Last, Follow et Null pour l'expression entière, correspondant à $(a_1 \cdot b_2)^* + a_3 \cdot a_4$. Ceux-ci valent respectivement : $\{a_1, a_3\}$, $\{b_2, a_4\}$, $\{a_1b_2, b_2a_1, a_3a_4\}$ et vrai.

Question 24

On appelle ce parcours un parcours en profondeur, car en observant l'arbre syntaxique de l'expression, on parcours tous les enfants d'un nœud en augmentant le plus possible la profondeur tant que c'est possible, puis on remonte en traitant les nœuds dont tous les enfants sont traités.

Question 25

Les utilisations de `disjoint_extend` sont adéquates car elles permettent de faire l'union des deux ensembles en un temps linéaire, alors que la fonction `extend` fait la même chose en un temps proportionnel au produit des cardinaux des deux ensembles. Pour pouvoir utiliser la première fonction, il faut être certain que les deux ensembles considérés soient bien disjoints, ce qui est le cas dans toutes les occurrences faites dans la fonction `glushkov`.

Par exemple, on a bien $\mathrm{First}(E'_l + E''_l) = \mathrm{First}(E'_l) \sqcup \mathrm{First}(E''_l)$ car E'_l et E''_l sont linéaires, i.e chaque variable n'est présente qu'une seule fois dans l'expression (ce qui est une précondition de la fonction `glushkov`). On peut appliquer un raisonnement similaire dans tous les autres cas.

Question 26

La dernière extension du code à la ligne 29 n'est pas disjointe : par exemple, avec l'expression $E' = (a_1 + \varepsilon) \cdot (b_2 + \varepsilon)$, et $E = (E')^*$, on a : $\mathrm{Follow}(E) = \mathrm{Follow}(E') \cup \mathrm{Last}(E') \cdot \mathrm{First}(E')$ avec $\mathrm{Follow}(E') = \{a_1 b_2\}$, $\mathrm{First}(E') = \{a_1, b_2\}$ et $\mathrm{Last}(E') = \{a_1, b_2\}$ donc $\mathrm{Follow}(E') \cap \mathrm{Last}(E') \cdot \mathrm{Last}(E') = \{a_1 b_2\} \neq \emptyset$. L'union n'est donc pas disjointe.

Question 27

Pour montrer la correction totale de la fonction `glushkov`, il faut montrer sa correction partielle et sa terminaison.

En supposant que l'entrée est une expression linéaire bien formée, les appels récursifs à la fonction `glushkov` se font sur des sous-expressions de tailles strictement inférieures à celle de l'expression d'entrée. Les cas de base, à savoir $E = \varepsilon$ et $E = a \in \Sigma$ ne font pas d'appels récursifs. Comme il n'y a aucune boucle `while` et que les appels récursifs font décroître strictement la taille de l'expression d'entrée (qui est donc un variant d'appels récursifs) : la fonction `glushkov` termine bien.

Pour montrer la correction partielle, on suppose que la fonction termine bien, et on doit montrer qu'elle renvoie bien ce qui est attendu. Ici, comme la fonction a comme précondition que la formule d'entrée doit être locale et non vide, alors en utilisant la preuve de la question 19 on observe que les formules inductives pour First, Last, Follow et Null sont bien celles utilisées dans la fonction `glushkov`. La fonction renvoie donc bien les bonnes valeurs des ensembles First, Last et Follow, et la bonne valeurs du booléen Null.

Question 28

Pour montrer que la fonction `glushkov` restreinte au calcul de Null s'effectue en un temps $\Theta(n)$, il faut montrer que le temps d'exécution est de $\mathcal{O}(n)$ et $\Omega(n)$.

En restreignant tous les calculs à simplement ceux nécessaires pour l'obtention de la valeur de Null, il ne reste plus que les appels récursifs sur les sous-expressions et des opérations sur les booléens en temps $\mathcal{O}(1)$. Comme les appels récursifs décrivent un parcours en profondeur comme expliqué en question 24, et qu'un même sommet n'est visité qu'une seule fois, alors la fonction s'exécute bien en un temps $\mathcal{O}(n)$ avec n la taille de l'expression d'entrée.

Pour montrer que la fonction s'exécute bien en un temps $\Omega(n)$, exhibons une famille d'instances de taille n sur laquelle le temps d'exécution est au moins $C \times n$ avec C une constante. On définit récursivement la famille d'expressions (E_n) de la manière suivante : $E_0 = \varepsilon$ et $\forall n \in \mathbb{N}, E_{n+1} = (a_n + E_n)$, avec $a_n \in \Sigma$ pour tout n. Ainsi défini, il faut parcourir l'ensemble de l'arbre pour traiter ε (en remarquant que l'on traite les enfants d'un nœud de gauche à droite), ce qui permettra de faire remonter

l'information de $\text{Null}(E_n) = \text{vrai}$. Cette opération se fait en temps linéaire de la taille de l'expression, d'où le temps d'exécution de cette fonction est $\Omega(n)$.

Donc cette fonction restreinte au calcul de $\text{Null}(E)$ s'effectue en temps $\Theta(n)$.

Question 29

On considère la famille d'expressions (F_n) définie récursivement par : $F_1 = a_1^*$ et $F_{n+1} = (F_n + a_{n+1})^*$.

Montrons que $\forall n \in \mathbb{N}^*, \text{First}(F_n) = \text{Last}(F_n) = \{a_1, ..., a_n\}, \text{Follow}(F_n) = \{a_1, ..., a_n\}^2$.

- Si $n = 1$, alors $\text{First}(F_1) = \text{Last}(F_1) = \{a_1\}$ et $\text{Follow}(F_1) = \text{Follow}(a_1^*) = \{a_1 a_1\} = \{a_1\}^2$.
- Soit $n \in \mathbb{N}^*$ tel que $\text{Follow}(F_n) = \{a_1, ..., a_n\}^2$.

  $\text{First}(F_{n+1}) = \text{First}(F_n + a_{n+1}) = \text{First}(F_n) \cup \text{First}(a_{n+1}) = \{a_1, ..., a_{n+1}\}$

  $\text{Last}(F_{n+1}) = \text{Last}(F_n + a_{n+1}) = \text{Last}(F_n) \cup \text{Last}(a_{n+1}) = \{a_1, ..., a_{n+1}\}$

  $$\begin{aligned}\text{Follow}(F_{n+1}) &= \text{Follow}\big((F_n + a_{n+1})^*\big) \\ &= \text{Follow}(F_n + a_{n+1}) \cup \text{Last}(F_n + a_{n+1}) \cdot \text{First}(F_n + a_{n+1}) \\ &= \text{Follow}(F_n) \cup \underbrace{\text{Follow}(a_{n+1})}_{\emptyset} \cup \{a_1, ..., a_{n+1}\} \cdot \{a_1, ..., a_{n+1}\} \\ &= \{a_1, ..., a_{n+1}\}^2\end{aligned}$$

Donc $\forall n \in \mathbb{N}^*, \text{Follow}(F_n) = \{a_1, ..., a_n\}^2$, donc $|\text{Follow}(F_n)| = n^2$, et $\text{First}(F_n) = \text{Last}(F_n) = \{a_1, ..., a_n\}$ donc $|\text{First}(F_n)| = |\text{Last}(F_n)| = n$.

Montrons maintenant par récurrence sur $n \in \mathbb{N}^*$ que la complexité du calcul de Follow dans la fonction `glushkov` pour F_n, que l'on note C_n, vérifie $C_n \geqslant \sum_{k=1}^{n-1} k^4$.

- Si $n = 1$, alors C_1 est le temps de calcul de Follow dans `glushkov` pour $F_1 = a_1^*$. Celui-ci est constant et contient plus d'une quinzaine d'opérations élémentaires, on obtient alors directement $C_1 \geqslant \sum_{k=1}^{0} k^4 = 0$.
- Soit $n \in \mathbb{N}^*$ tel que $C_n \geqslant \sum_{k=1}^{n-1} k^4$. Alors, considérons un appel à la fonction `glushkov` sur l'expression $F_{n+1} = (F_n + a_{n+1})^*$. On dénombre alors environ une dizaine d'opérations élémentaires un appel récursif à l'expression $F_n + a_{n+1}$ et une extension d'ensemble.

  L'opération d'extension d'ensemble (celle de la ligne 29) s'effectue en un temps proportionnel au produit des cardinaux des deux ensembles. Les ensembles sont `follow1`, représentant $\text{Follow}(F_n + a_{n+1})$, contenant n^2 éléments, et `last1.product(first1)`, représentant $\text{Last}(F_n + a_{n+1}) \cdot \text{First}(F_n + a_{n+1})$. L'opération `product` s'effectue en un temps proportionnel au produit des cardinaux des deux ensembles, ici n^2. Cette opération s'effectue en un temps au moins proportionnel à n^4.

  L'appel récursif s'effectue en environ une quinzaine d'opérations élémentaires, un appel récursif sur F_n, un sur a_{n+1} et une extension disjointe. Les appels récursifs prennent respectivement un temps C_n et environ une dizaine d'opérations élémentaires.

Le temps total pris par cet appel est donc : $C_{n+1} \geqslant C_n + n^4 + \mathcal{O}(1) \geqslant \sum_{k=1}^{n-1} k^4 + n^4 = \sum_{k=1}^{n} k^4$.

Or, $\sum_{k=1}^{n} k^4$ est proportionnel à n^5, donc $\exists C \in \mathbb{R}_+^*, C_n \geqslant Cn^5$.

Donc le temps de calcul de Follow pour l'expression F_n dans la fonction `glushkov` est au moins proportionnel à n^5.

Donc le calcul de Follow dans la fonction `glushkov` s'effectue en un temps $\Omega(n^5)$.

Question 30

On a :

- $\mathrm{RFoll}(\varepsilon) = (\mathrm{Last}(\varepsilon) \times \mathrm{First}(\varepsilon)) \setminus \mathrm{Follow}(\varepsilon) = \emptyset$
- $\forall a \in \Sigma, \mathrm{RFoll}(a) = (\mathrm{Last}(a) \times \mathrm{First}(a)) \setminus \mathrm{Follow}(a) = (\{a\} \times \{a\}) \setminus \emptyset = \{aa\}$

De plus, soit E une expression rationnelle locale.

Alors, on a $\mathrm{First}(E^*) = \mathrm{First}(E)$, $\mathrm{Last}(E^*) = \mathrm{Last}(E)$ et $\mathrm{Follow}(E^*) = \mathrm{Follow}(E) \cup \mathrm{Last}(E) \cdot \mathrm{First}(E)$.

On a alors :

$$\begin{aligned}\mathrm{RFoll}(E^*) &= (\mathrm{Last}(E^*) \times \mathrm{First}(E^*)) \setminus \mathrm{Follow}(E^*) \\ &= (\mathrm{Last}(E) \times \mathrm{First}(E)) \setminus (\mathrm{Follow}(E) \cup \mathrm{Last}(E) \cdot \mathrm{First}(E))\end{aligned}$$

$\mathrm{RFoll}(E^*) = \emptyset$

Question 31

Soient E et F deux expressions rationnelles locales telles que les lettres utilisées dans E et F sont toutes distinctes. On a donc $E + F$ et $E \cdot F$ locales.

On a alors :

- $\mathrm{First}(E + F) = \mathrm{First}(E) \sqcup \mathrm{First}(F)$ car les lettres de E et F sont distinctes, donc $\mathrm{First}(E) \cap \mathrm{First}(F) = \emptyset$
- $\mathrm{Last}(E + F) = \mathrm{Last}(E) \sqcup \mathrm{Last}(F)$ pour les mêmes raisons
- $\mathrm{Follow}(E + F) = \mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)$

On a donc :

$$\begin{aligned}
\mathrm{RFoll}(E+F) &= (\mathrm{Last}(E+F) \times \mathrm{First}(E+F)) \setminus \mathrm{Follow}(E+F) \\
&= ((\mathrm{Last}(E) \sqcup \mathrm{Last}(F)) \times (\mathrm{First}(E) \sqcup \mathrm{First}(F))) \setminus (\mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)) \\
&= (\mathrm{Last}(E) \times \mathrm{First}(E) \sqcup \mathrm{Last}(F) \times \mathrm{First}(F) \sqcup \mathrm{Last}(E) \times \mathrm{First}(F) \sqcup \mathrm{Last}(F) \times \mathrm{First}(E)) \\
&\quad \setminus (\mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)) \\
&= (\mathrm{Last}(E) \times \mathrm{First}(E)) \setminus (\mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)) \\
&\quad \sqcup (\mathrm{Last}(F) \times \mathrm{First}(F)) \setminus (\mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)) \\
&\quad \sqcup (\mathrm{Last}(E) \times \mathrm{First}(F)) \setminus (\mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)) \\
&\quad \sqcup (\mathrm{Last}(F) \times \mathrm{First}(E)) \setminus (\mathrm{Follow}(E) \sqcup \mathrm{Follow}(F)) \\
&= (((\mathrm{Last}(E) \times \mathrm{First}(E)) \setminus \mathrm{Follow}(E)) \cap ((\mathrm{Last}(E) \times \mathrm{First}(E)) \setminus \mathrm{Follow}(F))) \\
&\quad \sqcup (((\mathrm{Last}(F) \times \mathrm{First}(F)) \setminus \mathrm{Follow}(E)) \cap ((\mathrm{Last}(F) \times \mathrm{First}(F)) \setminus \mathrm{Follow}(F))) \\
&\quad \sqcup (((\mathrm{Last}(E) \times \mathrm{First}(F)) \setminus \mathrm{Follow}(E)) \cap ((\mathrm{Last}(E) \times \mathrm{First}(F)) \setminus \mathrm{Follow}(F))) \\
&\quad \sqcup (((\mathrm{Last}(F) \times \mathrm{First}(E)) \setminus \mathrm{Follow}(E)) \cap ((\mathrm{Last}(F) \times \mathrm{First}(E)) \setminus \mathrm{Follow}(F)))
\end{aligned}$$

Or, comme $\mathrm{Last}(E) \times \mathrm{First}(E)$ ne contient que des lettres présentes dans l'expression E, alors $(\mathrm{Last}(E) \times \mathrm{First}(E)) \cap \mathrm{Follow}(F) = \emptyset$, et de même pour F. De plus, comme $\mathrm{Last}(E) \times \mathrm{First}(F)$ contient une lettre de E puis une lettre de F, et que $\mathrm{Follow}(E)$ contient deux lettres de E, alors $(\mathrm{Last}(E) \times \mathrm{First}(F)) \cap \mathrm{Follow}(E) = \emptyset$. On raisonne de la même façon pour les autres cas, et on peut alors réécrire cette égalité de la manière suivante :

$$\begin{aligned}
\mathrm{RFoll}(E+F) &= ((\mathrm{Last}(E) \times \mathrm{First}(E)) \setminus \mathrm{Follow}(E)) \sqcup ((\mathrm{Last}(F) \times \mathrm{First}(F)) \setminus \mathrm{Follow}(F)) \\
&\quad \sqcup (\mathrm{Last}(E) \times \mathrm{First}(F)) \sqcup (\mathrm{Last}(F) \times \mathrm{First}(E)) \\
\mathrm{RFoll}(E+F) &= \mathrm{RFoll}(E) \sqcup \mathrm{RFoll}(F) \sqcup \mathrm{Last}(E) \times \mathrm{First}(F) \sqcup \mathrm{Last}(F) \times \mathrm{First}(E)
\end{aligned}$$

De même, on a :

- $\mathrm{First}(E \cdot F) = \mathrm{First}(E) \sqcup \mathrm{Null}(E) \cdot \mathrm{First}(F)$ car les lettres de E et de F sont distinctes
- $\mathrm{Last}(E \cdot F) = \mathrm{Last}(F) \sqcup \mathrm{Null}(F) \cdot \mathrm{Last}(E)$ pour les mêmes raisons
- $\mathrm{Follow}(E \cdot F) = \mathrm{Follow}(E) \sqcup \mathrm{Follow}(F) \sqcup \mathrm{Last}(E) \cdot \mathrm{First}(F)$

On a donc :

$$\begin{aligned}
\mathrm{RFoll}(E \cdot F) &= (\mathrm{Last}(E \cdot F) \times \mathrm{First}(E \cdot F)) \setminus \mathrm{Follow}(E \cdot F) \\
&= ((\mathrm{Last}(F) \sqcup \mathrm{Null}(F) \cdot \mathrm{Last}(E)) \times (\mathrm{First}(E) \sqcup \mathrm{Null}(E) \cdot \mathrm{First}(F))) \\
&\quad \setminus (\mathrm{Follow}(E) \cup \mathrm{Follow}(F) \cup \mathrm{Last}(E) \times \mathrm{First}(F)) \\
&= (\mathrm{Last}(F) \times \mathrm{First}(E) \sqcup \mathrm{Last}(F) \times \mathrm{Null}(E) \cdot \mathrm{First}(F) \sqcup \mathrm{Null}(F) \cdot \mathrm{Last}(E) \times \mathrm{First}(E) \\
&\quad \sqcup \mathrm{Null}(F) \cdot \mathrm{Last}(E) \times \mathrm{Null}(E) \cdot \mathrm{First}(F)) \setminus (\mathrm{Follow}(E) \cup \mathrm{Follow}(F) \cup \mathrm{Last}(E) \times \mathrm{First}(F)) \\
&= ((\mathrm{Last}(F) \times \mathrm{First}(E)) \setminus (\mathrm{Follow}(E) \cup \mathrm{Follow}(F) \cup \mathrm{Last}(E) \times \mathrm{First}(F))) \\
&\quad \sqcup (\mathrm{Null}(E) \cdot (\mathrm{Last}(F) \times \mathrm{First}(F)) \setminus (\mathrm{Follow}(E) \cup \mathrm{Follow}(F) \cup \mathrm{Last}(E) \times \mathrm{First}(F))) \\
&\quad \sqcup (\mathrm{Null}(F) \cdot (\mathrm{Last}(E) \times \mathrm{First}(E)) \setminus (\mathrm{Follow}(E) \cup \mathrm{Follow}(F) \cup \mathrm{Last}(E) \times \mathrm{First}(F))) \\
&\quad \sqcup ((\mathrm{Null}(E) \wedge \mathrm{Null}(F)) \cdot (\mathrm{Last}(E) \times (\mathrm{First}(F))) \\
&\qquad \setminus (\mathrm{Follow}(E) \cup \mathrm{Follow}(F) \cup \mathrm{Last}(E) \times \mathrm{First}(F)))
\end{aligned}$$

En utilisant les mêmes types de raisonnement que pour le cas $E + F$, on peut simplifier cette expression comme ceci :

$$\begin{aligned}\mathrm{RFoll}(E \cdot F) &= \mathrm{Last}(F) \times \mathrm{First}(E) \\ &\quad \sqcup (\mathrm{Null}(E) \cdot (\mathrm{Last}(F) \times \mathrm{First}(F))) \setminus \mathrm{Follow}(F) \\ &\quad \sqcup (\mathrm{Null}(F) \cdot (\mathrm{Last}(E) \times \mathrm{First}(E))) \setminus \mathrm{Follow}(E) \\ &\quad \sqcup \emptyset \\ &= \mathrm{Last}(F) \times \mathrm{First}(E) \\ &\quad \sqcup \mathrm{Null}(E) \cdot (\mathrm{Last}(F) \times \mathrm{First}(F) \setminus \mathrm{Follow}(F)) \\ &\quad \sqcup \mathrm{Null}(F) \cdot (\mathrm{Last}(E) \times \mathrm{First}(E) \setminus \mathrm{Follow}(E))\end{aligned}$$

$$\mathrm{RFoll}(E \cdot F) = \mathrm{Last}(F) \times \mathrm{First}(E) \sqcup \mathrm{Null}(E) \cdot \mathrm{RFoll}(F) \sqcup \mathrm{Null}(F) \cdot \mathrm{RFoll}(E)$$

Question 32

On peut utiliser RFoll de la manière suivante : on peut le calculer récursivement dans la fonction `glushkov` de la même façon que First, Last, Follow et Null, ce qui permettra d'accélérer le cas problématique dans le calcul de Follow, à savoir le cas $E = (E')^*$: on peut alors remplacer la fonction `extend` à la ligne 29 par la ligne `follow = follow1.disjoint_extend(rfoll1)`, avec `rfoll1` renvoyé par l'appel récursif de la ligne 27. Cela permet de passer d'une complexité temporelle pour ce calcul de $\Omega(n^4)$ (comme on peut le voir en question 29) à une complexité $\Omega(n^2)$. La fonction `glushkov` restreinte au calcul de Follow prendrait alors un temps de calcul en $\Omega(n^3)$.

Question 33

En implémentant les ensembles par des listes doublement chaînées, on peut représenter un ensemble comme une liste doublement chaînée et deux pointeurs (un sur le nœud de début et un sur le nœud de fin) où chaque nœud correspond à un élément de l'ensemble.

Ainsi définie, l'extension disjointe s'effectue en temps constant en concaténant les deux listes doublement chaînées, ce qui se fait en temps constant en ajoutant le nœud de début de la seconde liste comme nœud suivant du dernier nœud de la première liste, et inversement.

Avec cette représentation, renvoyer un ensemble vide ou un singleton s'effectue en temps constant, et le produit et l'extension classique s'effectuent en un temps proportionnel au produit des tailles des deux listes : pour le produit, on crée tous les couples d'éléments en faisant deux boucles imbriquées sur chaque liste, et pour l'extension classique on s'assure pour chaque élément de la seconde liste qu'il n'est pas déjà présent dans la première liste en parcourant tous ses éléments.

Partie III. Test d'égalité de langages reconnus par automates

Partie 5. Partition des états

Question 34

Soient $\mathcal{A}$ l'automate décrit et $\mathcal{P} = \{\{0,1\},\{2,3\}\}$.

On a alors : $0 \sim_{\mathcal{P}} 1$ et $2 \sim_{\mathcal{P}} 3$.

De plus, on a :

- $\delta(0,a) = 2$ et $\delta(0,b) = 1$
- $\delta(1,a) = 3$ et $\delta(1,b) = 1$
- $\delta(2,a) = 3$ et $\delta(2,b) = 1$
- $\delta(3,a) = 3$ et $\delta(3,b) = 2$

En notant $\mathcal{P}'$ le raffinement de $\mathcal{P}$, on regarde tous les couples $(p, q) \in Q^2$ tels que $p \sim_{\mathcal{P}'} q$: ainsi définis, p et q doivent vérifier 2 propriétés, à savoir $p \sim_{\mathcal{P}} q$ et $\forall a \in \Sigma, \delta(p, a) \sim_{\mathcal{P}} \delta(q, a)$.

La première propriété n'est vérifiée que pour 0 et 1 d'un côté, et 2 et 3 de l'autre. De plus, on a $\delta(0, a) = 2 \sim_{\mathcal{P}} 3 = \delta(1, a)$, et de même pour b, d'où $0 \sim_{\mathcal{P}'} 1$. Cependant, $\delta(2, b) = 1 \nsim_{\mathcal{P}} 2 = \delta(3, b)$ donc $2 \nsim_{\mathcal{P}'} 3$.

On a donc : $\mathcal{P}' = \{\{0, 1\}, \{2\}, \{3\}\}$.

Question 35

```python
def nombre_parts(part: List[int]) -> int:
    maximum = part[0]
    for el in part:
        if maximum < el:
            maximum = el
    return maximum + 1
```

Comme les parts d'une partition composée de c sont numérotées entre 0 et $c - 1$, alors pour retrouver c il suffit de prendre le maximum atteint dans la liste et ajouter 1.

Question 36

```python
def raffine(A, part: List[int]) -> List[int]:
    (m, n, delta, q0, F) = A

    # On construit s pas à pas
    s = []
    for p in range(n):
        sp = [part[p]]
        for k in range(m):
            sp.append(part[delta[p][k]])
        sp.append(p)
        s.append(sp)
    # s est maintenant construite pour tout état p

    # On tri selon l'ordre lexicographique la liste s
    # sur des éléments compris entre 0 et n-1
    s = tri_par_cle(s, n)

    # On construit la partition en parcourant la liste s dans l'ordre
    raff = [0] * n
    p = 0
    s_actuel = s[0][:-1]
    for i in range(1, n):
        if s[i][-1] != s_actuel:
            p += 1
            s_actuel = s[i][:-1]
```

```
26      raff[s[i][-1]] = p
27
28    return raff
```

Partie 6. Partition de Nerode

Question 37

```python
def calcule_partition0(A) -> List[int]:
  (m, n, delta, q0, F) = A
  p0 = [0] * n
  for p in range(n):
    if F[p]:
      p0[p] = 1
  return p0
```

Question 38

Pour l'automate de la question 34, on a $F = \{0, 3\}$. On a donc directement $\mathcal{P}_0 = \{\{0, 3\}, \{1, 2\}\}$.

On peut donc calculer le Σ-raffinement de $\mathcal{P}_0$ nommé $\mathcal{P}_1$ de la même manière que dans la question 34, et on obtient : $\mathcal{P}_1 = \{\{0\}, \{1, 2\}, \{3\}\}$. On peut continuer de la même manière, et on obtient $\mathcal{P}_2 = \{\{0\}, \{1, 2\}, \{3\}\}$.

Question 39

Soit $\mathcal{A} = (\Sigma, Q, \delta, i_0, F)$ un automate déterministe complet.

Montrons que $\forall i \in \mathbb{N}, \forall (p, q) \in Q^2, p \sim_i q \iff \mathcal{L}_p \cap \Sigma^{\leqslant i} = \mathcal{L}_q \cap \Sigma^{\leqslant i}$ par récurrence.

- Si $i = 0$, alors soit $(p, q) \in Q^2$.
  - Si $p \sim_0 q$, alors soit $p, q \in F$ soit $p, q \in \overline{F}$.

    Si $p, q \in F$, alors $\mathcal{L}_p \cap \Sigma^{\leqslant 0} = \{\varepsilon\}$, et de même pour q, donc $\mathcal{L}_p \cap \Sigma^{\leqslant 0} = \mathcal{L}_q \cap \Sigma^{\leqslant 0}$. Sinon, $p, q \in F$ alors $\mathcal{L}_p \cap \Sigma^{\leqslant 0} = \emptyset$, et de même pour q.

    On a donc bien dans tous les cas : $\mathcal{L}_p \cap \Sigma^{\leqslant 0} = \mathcal{L}_q \cap \Sigma^{\leqslant 0}$.
  - Si $\mathcal{L}_p \cap \Sigma^{\leqslant 0} = \mathcal{L}_q \cap \Sigma^{\leqslant 0}$, alors comme $\Sigma^{\leqslant 0} = \{\varepsilon\}$, on a soit $\varepsilon \in \mathcal{L}_p$ et $\varepsilon \in \mathcal{L}_q$, soit $\varepsilon \notin \mathcal{L}_p$ et $\varepsilon \notin \mathcal{L}_q$.

    Si $\varepsilon \in \mathcal{L}_p$ et $\varepsilon \in \mathcal{L}_q$, alors comme $\mathcal{A}_p$ a pour état initial p, alors $p \in F$, et de même pour q, donc $p \sim_0 q$. Sinon, $\varepsilon \notin \mathcal{L}_p$ et $\varepsilon \notin \mathcal{L}_q$ donc $p \notin F$ d'où $p \in \overline{F}$, et de même pour q, donc $p \sim_0 q$.

    On a donc bien dans tous les cas : $p \sim_0 q$.
- Soit $i \in \mathbb{N}$ tel que $\forall (p, q) \in Q^2, p \sim_i q \iff \mathcal{L}_p \cap \Sigma^{\leqslant i} = \mathcal{L}_q \cap \Sigma^{\leqslant i}$. Montrons que cette propriété est vraie au rang $i + 1$.
  - Soient $(p, q) \in Q^2$ tel que $p \sim_{i+1} q$ et $w \in \mathcal{L}_p \cap \Sigma^{\leqslant i+1}$. Si $w \in \Sigma^{\leqslant i}$, alors on a directement $w \in \mathcal{L}_q \cap \Sigma^{\leqslant i}$ par hypothèse de récurrence. Il reste donc le cas $w = w_0 ... w_i \in \mathcal{L}_p \cap \Sigma^{i+1}$.

    Comme $w \in \mathcal{L}_p$, alors le mot w est accepté par l'automate déterministe complet $\mathcal{A}_p$. On note $p_0 = p, p_1, ..., p_{i+1}$ les $p + 2$ états de l'exécution de w sur $\mathcal{A}_p$, on a alors : $\forall k \in [\![0, i +$

$1]\!], p_{k+1} = \delta(p_k, w_k)$. Comme $p \sim_{i+1} q$, alors $\delta(p, w_0) \sim_i \delta(q, w_0)$: par hypothèse de récurrence, on a donc $\mathcal{L}_{\delta(p,w_0)} \cap \Sigma^{\leqslant i} = \mathcal{L}_{\delta(q,w_0)} \cap \Sigma^{\leqslant i}$. Le mot $w_1 ... w_i$ est un mot de longueur i, donc dans $\Sigma^{\leqslant i}$, et est reconnu par l'automate $\mathcal{A}$ en commençant de l'état $\delta(p, w_0)$ (car w l'est en partant de l'état p), donc $w_1 ... w_i \in \mathcal{L}_{\delta(p,w_0)} \cap \Sigma^{\leqslant i} = \mathcal{L}_{\delta(q,w_0)} \cap \Sigma^{\leqslant i}$, donc $w_1 ... w_i$ est reconnu également par l'automate $\mathcal{A}$ en partant de l'état $\delta(q, w_0)$. De fait, w est reconnu par l'automate $\mathcal{A}$ en partant de l'état q, d'où $w \in \mathcal{L}_q \cap \Sigma^{\leqslant i+1}$. On a donc : $\mathcal{L}_p \cap \Sigma^{\leqslant i+1} \subseteq \mathcal{L}_q \cap \Sigma^{\leqslant i+1}$.

En raisonnant de la même manière, on montre que $\mathcal{L}_q \cap \Sigma^{\leqslant i+1} \subseteq \mathcal{L}_p \cap \Sigma^{\leqslant i+1}$ car p et q jouent des rôles symétriques.

On en conclut donc que $\mathcal{L}_p \cap \Sigma^{\leqslant i+1} = \mathcal{L}_q \cap \Sigma^{\leqslant i+1}$.

- Soit $(p, q) \in Q^2$ tel que $\mathcal{L}_p \cap \Sigma^{\leqslant i+1} = \mathcal{L}_q \cap \Sigma^{\leqslant i+1}$. Montrons que $p \sim_{i+1} q$.

  Soit $w \in \mathcal{L}_p \cap \Sigma^{\leqslant i}$: comme $\Sigma^{\leqslant i} \subseteq \Sigma^{\leqslant i+1}$, alors $w \in \mathcal{L}_q \cap \Sigma^{\leqslant i+1} \cap \Sigma^{\leqslant i} = \mathcal{L}_q \cap \Sigma^{\leqslant i}$. Par hypothèse de récurrence, on a alors immédiatement : $p \sim_i q$.

  Soit $a \in \Sigma$: montrons que $\delta(p, a) \sim_i \delta(q, a)$. Soit $w \in \mathcal{L}_{\delta(p,a)} \cap \Sigma^{\leqslant i}$. Par définition, w est un mot d'au plus i lettres accepté par l'automate $\mathcal{A}_{\delta(p,a)}$, c'est-à-dire par l'automate $\mathcal{A}$ avec $\delta(p, a)$ pour état de départ. Ainsi, le mot $a \cdot w$ est accepté par l'automate $\mathcal{A}$ avec p pour sommet de départ, donc $aw \in \mathcal{L}_p \cap \Sigma^{\leqslant i+1} = \mathcal{L}_q \cap \Sigma^{\leqslant i+1}$. On a donc : $w \in \mathcal{L}_{\delta(q,a)} \cap \Sigma^{\leqslant i}$, d'où $\mathcal{L}_{\delta(p,a)} \cap \Sigma^{\leqslant i} \subseteq \mathcal{L}_{\delta(q,a)} \cap \Sigma^{\leqslant i}$. En raisonnement de la même façon sur q, on montre que $\mathcal{L}_{\delta(q,a)} \cap \Sigma^{\leqslant i} \subseteq \mathcal{L}_{\delta(p,a)} \cap \Sigma^{\leqslant i}$ par symétrie entre p et q. On a donc : $\mathcal{L}_{\delta(p,a)} \cap \Sigma^{\leqslant i} = \mathcal{L}_{\delta(q,a)} \cap \Sigma^{\leqslant i}$, donc par hypothèse de récurrence, $\delta(p, a) \sim_i \delta(q, a)$.

  On a donc : $p \sim_{i+1} q$.

On a donc : $\forall i \in \mathbb{N}, \forall (p, q) \in Q^2, p \sim_i q \iff \mathcal{L}_p \cap \Sigma^{\leqslant i} = \mathcal{L}_q \cap \Sigma^{\leqslant i}$.

Question 40

On s'intéresse ici à la variable `part` dans la fonction `moore` : elle représente les raffinements successifs en débutant de $\mathcal{P}_0$. Par récurrence immédiate, on montre que $\forall i \in \mathbb{N}, |\mathcal{P}_i| \leqslant |\mathcal{P}_{i+1}|$ par le fait que $\forall p, q \in Q, p \sim_{i+1} q \implies p \sim_i q$. La quantité `nombre_parts(part)` est donc croissante le long des itérations de la boucle `while`, et est bornée entre 1 et n avec $n \geqslant 2$ le nombre d'états de l'automate $\mathcal{A}$ d'entrée. Comme la comparaison dans la condition de la boucle `while` porte sur l'évolution de la quantité `nombre_parts(part)`, que celle-ci est croissante et qu'elle est bornée entre 1 et n, alors il peut y avoir au plus n itérations de boucle. De plus, si $F = \emptyset$ ou $F = Q$, alors il n'y aura qu'un unique tour de boucle effectué car dans ces cas, on a directement $\mathcal{P}_1 = \mathcal{P}_0$ car tous les éléments sont dans la même classe d'équivalence. Il reste donc le cas $F \neq \emptyset$ et $F \neq Q$, auquel cas `nombre_parts(part)` est borné entre 2 et n, il peut donc y avoir au plus $n - 1$ itérations de la boucle. Donc d'une manière générale, si $n \geqslant 2$, alors il peut y avoir au plus $n - 1$ itérations de la boucle `while` .

Question 41

Pour montrer la correction totale de l'algorithme de Moore, il faut démontrer sa terminaison et sa correction. La terminaison est directement établie par la question 40. Il reste à montrer la correction partielle.

Montrons que l'algorithme de Moore renvoie la partition de Nerode de l'automate $\mathcal{A}$ d'entrée.

On considère la suite $(|\mathcal{P}_n|)_{n \in \mathbb{N}}$: elle est à valeurs dans $\mathbb{N}$ et est bornée, elle est donc stationnaire à partir d'un rang n_0, c'est-à-dire que $\forall i \in \mathbb{N}, |\mathcal{P}_{n_0}| = |\mathcal{P}_{n_0+i}|$.

Montrons que si $\mathcal{P}$ et $\mathcal{P}'$ sont deux partitions d'un automate $\mathcal{A}$ telles que $|\mathcal{P}| = |\mathcal{P}'|$ et que $\mathcal{P}'$ est le raffinement de $\mathcal{P}$, alors $\mathcal{P} = \mathcal{P}'$. Soit $E \in \mathcal{P}$: si $|E| = 1$, alors $E = \{x\} \in \mathcal{P}$, donc x est seul dans sa classe d'équivalence dans $\mathcal{P}$, et $p \sim_{\mathcal{P}'} q \implies p \sim_{\mathcal{P}} q$ donc x est seul dans sa classe d'équivalence dans $\mathcal{P}'$, donc $E \in \mathcal{P}'$. Sinon, $|E| \geqslant 2$ donc E selon la seconde condition de la seconde propriété (1) pour les éléments de E, tous les éléments de E seront dans la même classe d'équivalence ou non. Dans tous les cas, un élément de E ne peut se retrouver que seul dans sa classe d'équivalence dans $\mathcal{P}'$ ou avec d'autres éléments de E du fait de la première propriété de (1). On peut alors considérer chaque classe d'équivalence de $\mathcal{P}$ indépendamment des autres, et montrer que ses éléments seront soit tous dans la même classe d'équivalence de $\mathcal{P}'$, ce qui ne fait pas changer le cardinal de $\mathcal{P}'$ par rapport à celui de $\mathcal{P}$, soit seront séparés en plusieurs classes d'équivalences de $\mathcal{P}'$, auquel cas le cardinal de $\mathcal{P}'$ est strictement supérieur à celui de $\mathcal{P}$. Comme $|\mathcal{P}| = |\mathcal{P}'|$ par hypothèse, alors on est dans le premier cas : pour toute classe d'équivalence $E \in \mathcal{P}$, tous les éléments de E sont dans la même classe d'équivalence de $\mathcal{P}'$, d'où $E \in \mathcal{P}'$. On a alors $\mathcal{P} \subseteq \mathcal{P}'$, et $|\mathcal{P}| = |\mathcal{P}'|$ donc $\mathcal{P} = \mathcal{P}'$.

On en déduit donc immédiatement que $\forall i \in \mathbb{N}, \mathcal{P}_{n_0} = \mathcal{P}_{n_0+i}$. À partir de la fonction `moore` on conclut directement qu'elle calcule $\mathcal{P}_{n_0}$. Il suffit donc de montrer que $\mathcal{P}_{n_0} = \mathcal{N}_{\mathcal{A}}$.

Soient $p, q \in Q$ tels que $p \sim q$. Alors, on a $\mathcal{L}_p = \mathcal{L}_q$, donc $\mathcal{L}_p \cap \Sigma^{\leqslant n_0} = \mathcal{L}_q \cap \Sigma^{\leqslant n_0}$, d'où d'après la question 39, on a : $p \sim_{n_0} q$.

Soient $p, q \in Q$ tels que $p \sim_{n_0} q$. Montrons que $\mathcal{L}_p = \mathcal{L}_q$: soit $x \in \mathcal{L}_p$, on a $x \in \mathcal{L}_p \cap \Sigma^{\leqslant |x|}$. Par ailleurs, si $|x| \leqslant n_0$, alors $p \sim_{|x|} q$ car $p \sim_{n_0} q \implies p \sim_{n_0-1} q \implies \ldots \implies p \sim_{|x|} q$, et sinon $|x| > n_0$ donc $\mathcal{P}_{|x|} = \mathcal{P}_{n_0}$ par définition, d'où $\sim_{n_0} = \sim_{|x|}$ et $p \sim_{|x|} q$. Dans tous les cas, on a donc $p \sim_{|x|} q$, donc d'après la question 39, on a $x \in \mathcal{L}_q \cap \Sigma^{\leqslant |x|}$, d'où $x \in \mathcal{L}_q$ et $\mathcal{L}_p \subseteq \mathcal{L}_q$, et réciproquement en raisonnant de la même manière. On a donc : $\mathcal{L}_p = \mathcal{L}_q$, d'où $p \sim q$.

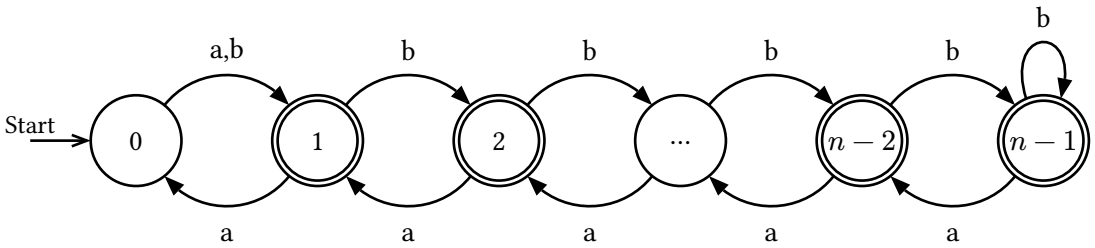
On a donc : $\forall p, q \in Q, p \sim q \iff p \sim_{n_0} q$ donc $\sim = \sim_{n_0}$ et $\mathcal{P}_{n_0} = \mathcal{N}_{\mathcal{A}}$. Donc la fonction `moore` renvoie bien la partition de Nerode de l'automate $\mathcal{A}$.

Question 42

L'appel à la fonction `calcule_partition0` s'effectue en $\mathcal{O}(n)$ d'après la question 37, et la boucle `while` itère au plus $n-1$ fois, d'après la question 40, avec à tour de boucle un appel à la fonction `raffine` effectué en $\mathcal{O}(nm)$ d'après la question 36. L'algorithme de Moore a donc une complexité temporelle de $\mathcal{O}(mn^2)$ où n est le nombre d'états de l'automate et m le nombre de lettres dans l'alphabet.

Question 43

On considère la suite d'automates $(\mathcal{A}_n)_{n\geqslant 2}$ où $\mathcal{A}_n$ est définie sur l'alphabet $\Sigma = \{a, b\}$ par :



Montrons que $\forall n \geqslant 2$, le plus petit indice n_0 tel que $\mathcal{P}_{n_0} = \mathcal{N}_{\mathcal{A}_n}$ pour l'automate $\mathcal{A}_n$ est $n-1$.

Soit $n \geqslant 2$. Toutes les partitions, fonction de transition, ... considérées ici sont pour l'ensemble des états de l'automate $\mathcal{A}_n$.

On a alors : $\mathcal{P}_0 = \{\{0\}, \{1, ..., n-1\}\}$. Pour calculer $\mathcal{P}_1$, on remarque que $\forall i \in [\![2, n-1]\!], \forall c \in \Sigma, \delta(i, c) \in [\![1, n-1]\!] \in \mathcal{P}_0$. Donc les éléments $2, 3, ..., n-1$ sont dans la même classe d'équivalence de $\mathcal{P}_1$. Pour 1, on a $\delta(1, a) = 0 \notin \{1, ..., n-1\}$, donc 1 est dans une autre classe d'équivalence. On a donc : $\mathcal{P}_1 = \{\{0\}, \{1\}, \{2, ..., n-1\}\}$. Par récurrence immédiate, on a alors : $\forall k \in [\![1, n-1]\!], \mathcal{P}_i = \{\{0\}, \{1\}, ..., \{k\}, \{k+1, ..., n-1\}\}$, et donc $|\mathcal{P}_k| = k+1$. Donc, le premier indice où la suite $(|\mathcal{P}_n|)$ est stationnaire est $n-1$, car avant la suite est strictement croissante, et $\mathcal{P}_{n-1} = \{\{0\}, \{1\}, ..., \{n-1\}\}$, et tout raffinement successif est égale à cette partition.

Donc sur l'entrée $\mathcal{A}_n$, l'algorithme de Moore fera $n-1$ tours de boucle `while`, et chaque tour fera appel à la fonction `raffine` qui fait exactement $\Theta(nm)$ opérations, d'où la fonction `moore` s'exécute en $\Omega(mn^2)$.

Par ailleurs, l'algorithme de Moore s'exécute en $\mathcal{O}(mn^2)$ d'après la question précédente, donc la complexité temporelle de l'algorithme de Moore est $\Theta(mn^2)$.

Partie 7. Test d'égalité utilisant la partition de Nerode

Question 44

Soient $p, q \in Q$ tels que $p \sim q$.Soient $p, q \in Q$ tels que $p \sim q$. D'après la question 41, il existe $n_0 \in \mathbb{N}$ tel que $\mathcal{P}_{n_0} = \mathcal{N}_{\mathcal{A}}$, donc $p \sim_{n_0} q$. On a donc : $p \sim_{n_0} q \implies p \sim_{n_0-1} q$, $p \sim_{n_0-1} q \implies p \sim_{n_0-2} q$, ..., $p \sim_1 q \Rightarrow p \sim_0 q$ donc par récurrence immédiate en utilisant la transitivité de $\implies$, $p \sim_{n_0} q \implies p \sim_0 q$, d'où $p \sim_0 q$. Comme $\mathcal{P}_0 = \left\{F, \overline{F}\right\}$, alors on a $(p, q) \in F^2$ ou $(p, q) \in \overline{F}^2$.

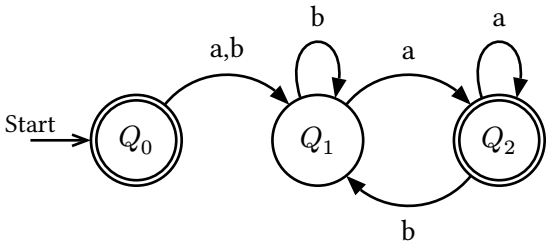
Question 45

Soient $p, q \in Q$ et $a \in \Sigma$ tels que $p \sim q$.Soient $p, q \in Q$ tels que $p \sim q$. D'après la question 41, il existe $n_0 \in \mathbb{N}$ tel que $\mathcal{P}_{n_0} = \mathcal{P}_{n_0+1} = \mathcal{N}_{\mathcal{A}}$, donc $p \sim_{n_0+1} q$. Par définition de $\sim_{n_0+1}$, on a $\delta(p, a) \sim_{n_0} \delta(q, a)$, donc $\delta(p, a) \sim \delta(q, a)$.

Question 46

En utilisant les résultats de la question 38, on a $\mathcal{P}_1 = \mathcal{P}_2 = \{\{0\}, \{1, 2\}, \{3\}\}$ donc $\mathcal{N}_{\mathcal{A}} = \{\{0\}, \{1, 2\}, \{3\}\}$ avec $Q_0 = \{0\}$, $Q_1 = \{1, 2\}$ et $Q_2 = \{3\}$.

Ainsi, on peut calculer l'automate quotient de l'automate de la question 34 :



Question 47

On peut tester si deux automates déterministes et complets $\mathcal{A}$ et $\mathcal{B}$ dont tous les états sont accessibles depuis l'état initial par l'algorithme suivant :

- On vérifie tout d'abord que $|Q_{\mathcal{A}}| = |Q_{\mathcal{B}}|$, et on renvoie faux si ce n'est pas le cas (car ils ne peuvent pas alors être en bijection).
- On construit ensuite petit à petit une bijection potentielle ϕ que l'on stocke dans un tableau de taille $n = |Q_{\mathcal{A}}| = |Q_{\mathcal{B}}|$, et on pose initialement $\phi[i_{\mathcal{A}}] = i_{\mathcal{B}}$.

- Ensuite, on effectue un parcours en profondeur (en largeur est également possible) partant de $i_{\mathcal{A}}$ tel que pour découvrir un sommet $q \in Q_{\mathcal{A}}$:
  - si q a déjà été découvert on ne fait rien ;
  - sinon, pour chaque lettre $a \in \Sigma$, si $\phi[\delta_{\mathcal{A}}(q,a)]$ n'existe pas, alors on lui assigne la valeur $\delta_{\mathcal{B}}(\phi[q],a)$, sinon on vérifie que $\phi[\delta_{\mathcal{A}}(q,a)] = \delta_{\mathcal{B}}(\phi[q],a)$, et on renvoie faux si ce n'est pas le cas. Dans tous les cas, on découvre par la suite tous les successeurs de q.
- Si le parcours se termine sans avoir renvoyé faux, on renvoie vrai.

Cet algorithme détermine bien si les deux automates sont isomorphes en construisant une bijection entre $Q_{\mathcal{A}}$ et $Q_{\mathcal{B}}$ si elle existe grâce au fait que tous les états de $\mathcal{A}$ seront découverts par le parcours en profondeur.

On effectue un parcours en profondeur, et toutes les opérations élémentaires s'effectuent en $\mathcal{O}(1)$, donc la complexité temporelle de cet algorithme est $\mathcal{O}\left(\underbrace{\text{nombre de sommets}}_{|Q_{\mathcal{A}}|} + \underbrace{\text{nombre d'arêtes}}_{|Q_{\mathcal{A}}| \times |\Sigma|}\right)$ donc $\mathcal{O}(\|\mathcal{A}\| \times |\Sigma|)$.

Partie 8. Test d'égalité avec la structure « unir & trouver »

Question 48

Montrons l'invariant de boucle `while` suivant :
« Au début de chaque tour de boucle, $\forall (p,q) \in$ `a_traiter`, $\exists u \in \Sigma^*, \delta_{\mathcal{A}}(i_{\mathcal{A}},u) = p \wedge \delta_{\mathcal{B}}(i_{\mathcal{B}},u) = q$ ».

- Au premier tour de boucle, on a `a_faire = [(iA, iB)]` et $\delta_{\mathcal{A}}(i_{\mathcal{A}},\varepsilon) = i_{\mathcal{A}}$ et $\delta_{\mathcal{B}}(i_{\mathcal{B}},\varepsilon) = i_{\mathcal{B}}$ donc la propriété est vérifiée.
- Supposons que la propriété soit vérifiée au début du k^{e} tour de boucle. Montrons qu'elle le reste en fin du tour de boucle. On redéfinit ligne 9 `p` et `q` comme un couple de `a_traiter`, on peut donc en déduire qu'il existe $u \in \Sigma^*$ tel que $\delta_{\mathcal{A}}(i_{\mathcal{A}},u) = p$ et $\delta_{\mathcal{B}}(i_{\mathcal{B}},u) = q$.
  - Si p et q, c'est-à-dire `p` et `q + nA`, sont dans la même partition, alors on a retiré un couple de `a_traiter` donc l'invariant de boucle reste vrai en fin de boucle car le seul changement qui a eu lieu n'impacte pas le $\forall$ de l'invariant.
  - Sinon, p et q ne sont pas dans la même partition : si `FA[p] != FB[q]` alors l'algorithme termine et la variable `a_traiter` n'a pas été modifiée.

    Sinon, on a `FA[p] == FB[q]` : on unit les partitions de p et de q et on ajoute à `a_traiter` tous les couples $(\delta_{\mathcal{A}}(p,a), \delta_{\mathcal{B}}(q,a))$ pour $a \in \Sigma$. On note les nouvelles variables de fin de boucle par un exposant $+$, et celles de début de boucle par un exposant $-$.

    Soit $(p',q') \in$ `a_traiter`$^+$: si $(p',q') \in$ `a_traiter`$^-$, alors on obtient directement le résultat par l'invariant de début boucle. Sinon, $\exists a \in \Sigma, p' = \delta_{\mathcal{A}}(p,a)$ et $q' = \delta_{\mathcal{B}}(q,a)$ car ce sont les seuls couples ajoutés à la liste `a_traiter`. Or, on a $\delta_{\mathcal{A}}(i_{\mathcal{A}},u) = p$ et $\delta_{\mathcal{B}}(i_{\mathcal{B}},u) = q$ donc $p' = \delta_{\mathcal{A}}(i_{\mathcal{A}}, u \cdot a)$ et $q' = \delta_{\mathcal{B}}(i_{\mathcal{B}}, u \cdot a)$. La propriété est donc vérifiée à tout moment de la boucle.

On en conclut donc qu'à tout moment de l'exécution, $\forall (p,q) \in$ `a_traiter`, $\exists u \in \Sigma^*, \delta_{\mathcal{A}}(i_{\mathcal{A}},u) = p$.

Si le programme renvoie faux, alors on considère les variables `p` et `q` telles que `FA[p] != FB[q]`. On suppose sans perte de généralité que `FA[p] == True` et `FB[q] == False`. D'après la propriété que

nous venons de montrer, comme `(p, q)` était un couple de `a_traiter`, alors $\exists u \in \Sigma^*, \delta_{\mathcal{A}}(i_{\mathcal{A}}, u) = p$ et $\delta_{\mathcal{B}}(i_{\mathcal{B}}, u) = q$. Comme `FA[p] == True`, alors $p \in F_{\mathcal{A}}$, donc $u \in \mathcal{L}(\mathcal{A})$. De même, comme `FB[q] == False`, alors $q \notin F_{\mathcal{B}}$. De plus, $\mathcal{B}$ est déterministe donc il n'y a qu'une seule exécution pour le mot u, d'où $u \notin \mathcal{L}(\mathcal{B})$.

Donc si l'algorithme renvoie faux, alors $\mathcal{L}(A) \neq \mathcal{L}(B)$.

Question 49

Remarque 2

Je ne vois pas comment faire cette question par récurrence, je présente donc ici une version par invariant de boucle.

On montre l'invariant de boucle suivant : « Si l'algorithme renvoie vrai, alors au début de chaque tour de boucle, $\forall P \in$ `partition`, $\forall p, q \in P, \forall a \in \Sigma$, $\delta(p, a)$ et $\delta(q, a)$ appartiennent à la même partie de $\mathcal{P}_{\text{fin}}$ ».

- Initialement, chaque élément est seul dans sa partie, donc $\forall P \in$ `partition`, $\forall p, q \in P, p = q$, donc l'invariant est immédiatement vérifié.
- On suppose que la propriété est vérifiée en début de tour de boucle, montrons qu'elle est toujours vérifiée en fin de boucle. Le seul cas pouvant modifier l'état de la partition est celui où $(p, q) \in$ `a_traiter` est tel que p et q sont dans deux partitions séparées et $p \in F_{\mathcal{A}} \iff q \in F_{\mathcal{B}}$. Dans ce cas, les partitions de p et q, respectivement P et Q, sont alors fusionnées dans une nouvelle partition R.

  Toutes les partitions autres que R n'ont pas été modifiées dans ce tour de boucle, la propriété reste donc valide pour elles. Soient $r, s \in R$ et $a \in \Sigma$. Montrons que $\delta(r, a)$ et $\delta(s, a)$ appartiennent à la même partie de $\mathcal{P}_{\text{fin}}$. Or, on a $R = P \cup Q$, donc si $r, s \in P$ ou $r, s \in Q$ alors on a directement le résultat avec l'invariant de boucle supposé au début du tour de boucle. On suppose donc sans perte de généralité que $r \in P$ et $s \in Q$. Par hypothèse d'invariant de boucle, on obtient donc que $\delta(p, a)$ et $\delta(r, a)$ appartiennent à la même partie de $\mathcal{P}_{\text{fin}}$, et de même pour $\delta(q, a)$ et $\delta(s, a)$. Il suffit alors de montrer que $\delta(p, a)$ et $\delta(q, a)$ appartiennent à la même partie de $\mathcal{P}_{\text{fin}}$. Comme le couple $(\delta(p, a), \delta(q, a))$ sera ajouté à la liste `a_traiter`, et que l'algorithme renvoie vrai, alors il y aura dans la suite de l'exécution une boucle où `p` $= \delta(p, a)$ et `q` $= \delta(q, a)$. Lors de cette boucle, comme l'algorithme ne renvoie pas faux, alors `p` et `q`, donc $\delta(p, a)$ et $\delta(q, a)$, finiront forcément par être dans la même partie de la partition (soit ils l'étaient déjà initialement, soit ils le deviendront par la ligne 14). Donc $\delta(p, a)$ et $\delta(q, a)$ font bien partie de la même partie de $\mathcal{P}_{\text{fin}}$, et donc $\delta(r, a)$ et $\delta(s, a)$ aussi.

  L'invariant de boucle est donc bien vérifié en fin de boucle également.

Donc tout au long de l'exécution de l'algorithme, si celui-ci renvoie vrai, alors si deux états $p, q \in Q$ sont dans la même partie P de la partition courante `partition`, alors pour toute lettre $a \in \Sigma$, $\delta(p, a)$ et $\delta(q, a)$ seront dans la même partie de $\mathcal{P}_{\text{fin}}$ à la fin de l'exécution.

Question 50

Si le programme renvoie vrai, alors soit $u = u_1 ... u_n \in \mathcal{L}(A)$ et montrons que $u \in \mathcal{L}(B)$.

Comme $i_{\mathcal{A}}$ et $i_{\mathcal{B}}$ sont dans la même partie de la partition courante après le premier tour de boucle, alors en utilisant la question précédente, on a $\delta(i_{\mathcal{A}}, u_1)$ et $\delta(i_{\mathcal{B}}, u_1)$ appartiennent à la même partie de $\mathcal{P}_{\text{fin}}$.

On peut donc en conclure que $\delta(i_{\mathcal{A}}, u_1)$ et $\delta(i_{\mathcal{B}}, u_1)$ ont fait partie de la même partie dans la partition courante et donc que $\delta(\delta(i_{\mathcal{A}}, u_1), u_2) = \delta(i_{\mathcal{A}}, u_1 \cdot u_2)$ et $\delta(\delta(i_{\mathcal{B}}, u_1), u_2) = \delta(i_{\mathcal{B}}, u_1 \cdot u_2)$ font partie de la même partie de $\mathcal{P}_{\text{fin}}$. En continuant de la même façon, on obtient par récurrence immédiate le fait que $\delta(i_{\mathcal{A}}, u)$ et $\delta(i_{\mathcal{B}}, u)$ font partie de la même partie de $\mathcal{P}_{\text{fin}}$.

Comme l'algorithme ne renvoie pas faux, alors on peut par invariant de boucle que si p et q deux états accessibles, sont dans la même partie de $\mathcal{P}_{\text{fin}}$, alors p est final si et seulement si q est final. Cela provient du fait que dans une même partition d'au moins deux éléments, il y a au moins deux éléments distincts p et q qui ont été ajoutés à `a_traiter`, et ceux-ci vérifient `FA[p] == FB[q]`.

On peut donc en déduire que $\delta(i_{\mathcal{A}}, u)$ est final si et seulement si $\delta(i_{\mathcal{B}}, u)$. Or, $u \in \mathcal{L}(A)$ donc $\delta(i_{\mathcal{A}}, u)$ est un état final, donc $\delta(i_{\mathcal{B}}, u)$ aussi, d'où $u \in \mathcal{L}(B)$.

La réciproque se fait exactement de la même façon en inversant les rôles de $\mathcal{A}$ et $\mathcal{B}$, et on en déduit que $\mathcal{L}(A) = \mathcal{L}(B)$.

Question 51

Toutes les opérations entre les lignes 3 et 7 s'effectuent en $\mathcal{O}(1)$. Il reste à compter le coût temporel de la boucle `while`.

À part pour l'initialisation, chaque ajout d'éléments dans `a_traiter` se fait par paquet de $m = |\Sigma|$ éléments, et cela est forcément précédé d'une opération d'union qui diminue de 1 le nombre total de parties dans la partition courante. Comme le nombre de parties de la partition courante est initialement n et est minorée par 1, alors il peut y avoir au plus $n - 1$ unions, chacune ajoutant exactement m éléments. De plus, il n'y a aucune autre manière d'ajouter des éléments à `a_traiter` et chaque tour de boucle a au plus deux opération `trouver` et une opération `union` sur la partition de taille n.

Cet algorithme a donc une complexité temporelle de $\mathcal{O}(m \times n \times 3\alpha(n))$, donc $\mathcal{O}(mn \cdot \alpha(n))$.