Partie I - Automates

1. $L_2\left(\mathcal{A}_{\{q_0\}}\right)$ est le langage décrit par l'expression régulière b^*aa.
2. Les mots reconnus par l'automate $\mathcal{A}$ sont les étiquettes des calculs de la forme $q_0^k q_1 q_2^{k'}$, avec k et k' deux nombres entiers naturels non-nuls.
   Les seuls calculs réussis au seuil 2 pour l'automate $\mathcal{A}_{q_1}$ sont les séquences :
   $$q_0^k q_1 q_2^{k'}, \quad (k,k') \in [\![1,2]\!]^2$$
   On a donc :
   $$L_2\left(\mathcal{A}_{q_1}\right) = \{aa,aab,baa,baab\}$$
3. La condition *iii)* de la définition 3 est vérifiée pour $\mathcal{A}_\emptyset$ pour un calcul dont l'étiquette est de longueur n, si et seulement sipour tout $i \in \mathbb{N}$, on a $i+s>n$, ce qui revient à $s>n$. $L_s\left(\mathcal{A}_\emptyset\right)$ est donc l'ensemble des mots acceptés par l'automate $\mathcal{A}$ de longueur strictement inférieure à s.
   On a donc :
   $$L_s\left(\mathcal{A}_\emptyset\right) = \left\{b^p aab^q, \quad (p,q) \in \mathbb{N}^2 \quad p+q+2<s\right\}$$
   En particulier pour $s \in [\![0,2]\!]$, $\left|L_s\left(\mathcal{A}_\emptyset\right)\right|$ et pour tout nombre entier s tel que $s>2$:
   $$\left|L_s\left(\mathcal{A}_\emptyset\right)\right| = \left|\left\{(p,q) \in \mathbb{N}^2, \quad p+q<s-2\right\}\right| = \sum_{p=0}^{s-3} s-2-p = \sum_{p=1}^{s-2} p = \frac{(s-2)(s-1)}{2}$$
4. *On revient au cas général*
   On ne peut répondre à la question qu'en l'interprétant. Une première interprétation consiste à penser que l'on majore le cardinal de l'ensemble des calculs réussis au seuil s ne passant par aucun état de O. La seconde consiste à penser que l'on majore l'ensemble des longueurs des mots reconnus par un calcul réussi au seuil s ne passant par aucun état de O. On propose une solution pour les deux interprétations, même si la seconde était sans doute celle à laquelle pensait le concepteur du sujet.

   - **Interprétation 1**
     Un majorant est le cardinal de l'ensemble des mots acceptés par $\mathcal{A}$ de longueur strictement inférieure à s, soit :
     $$\sum_{k=0}^{s-1} |\Sigma|^k$$
     c'est-à-dire s si $|\Sigma|=1$, $\dfrac{|\Sigma|^s-1}{|\Sigma|-1}$ sinon.
   - **Interprétation 2**
     s est un majorant.

5. On a
   i) $i_0 \leqslant s$;
   ii) $\ell - s \leqslant i_k \leqslant \ell$;
   iii) Pour tout $j \in [\![0,k-1]\!]$:
   $$i_{j+1} - i_j \leqslant s+1$$
6. Pour $\mathcal{A} = (Q,\Sigma,I,\delta,F)$, $O \subset Q$ et $s \in \mathbb{N}$, on considère l'automate $\mathcal{B} = (Q_B,\Sigma,\delta_B,F_B)$ défini par :
   $$Q_B = Q \times [\![0,s]\!], \quad F_B = F \times [\![0,s]\!]$$
   et pour tout $(q,k) \in Q_B$ et pour tout $a \in \Sigma$:
   - Si $k=s$ alors $\delta_B\left((q,s),a\right) = \emptyset$.
   - Si $k \in [\![0,s-1]\!]$:
   $$\delta_B\left((q,s),a\right) = \left(\delta\left(q,a\right) \cap O\right) \times \{0\} \cup \left(\delta\left(q,a\right) \setminus O\right) \times \{k+1\}$$

   Le principe de construction de l'automate $\mathcal{B}$ consiste, dans un calcul, à incrémenter le compteur défini dans la deuxième composante des états de $\mathcal{B}$ tant que l'on n'a pas rencontré un état de l'ensemble O et transformer en état puits, tous les états atteints par un calcul de longueur s qui ne rencontre pas un état de O.
   Par construction $\mathcal{B}$ reconnaît le langage $L_2\left(\mathcal{A}_O\right)$.

Partie II - Autour des tas

II.1 - Arbre binomial

7. On propose :

```
def Vide(a):
  return a == None

def Racine(a):
  assert not Vide(a), "arbre vide"
  return a[0]

def Fils(a):
  assert not Vide(a), "arbre vide"
  return a[1]
```

8. On choisit la plus petite des deux racines comme racine de l'arbre résultat. On définit l'arbre ayant la plus grande racine est défini comme fils de l'arbre résultat.

```
def ArbreBinomial(a1,a2):
  assert not Vide(a1) and not Vide(a2), "arbres vides"
  rac1,rac2,f1 = Racine(a1),Racine(a2),Fils(a1)
  if rac1 < rac2 :
    f1.append(a2)
    return (rac1,f1)
  else :
    return ArbreBinomial(a2,a1)
```

9. On démontre la propriété par récurrence sur $k \in \mathbb{N}$.

   - Pour $k = 0$ la propriété est vraie puisque l'ensemble des fils d'un arbre binomial d'ordre 0 est vide.
   - On considère $k \in \mathbb{N}$ et on suppose que les arbres binomiaux d'ordre k ont exactement k fils.
     On considère $a_{k+1} = (r, [t_0, \cdots, t_{n-1}])$ un arbre binomial d'ordre $k+1$, alors $(r, [t_0, \cdots, t_{n-2}])$ est un arbre binomial d'ordre k et par hypothèse de récurrence a exactement k fils.
     a_{k+1} a donc exactement $k+1$ fils, ce qui permet de conclure.

10. Le résultat de la question précédente assure que pour obtenir l'ordre d'un arbre binomial, il suffit de déterminer la longueur de la liste de ses fils.

```
def Ordre(a):
  assert not Vide(a), "arbre vide"
  return len(a[1])
```

11. On démontre le résultat par récurrence sur $k \in \mathbb{N}$.
    - Pour $k = 0$, le résultat est immédiat.
    - On considère $k \in \mathbb{N}$ et on suppose le résultat vrai pour les arbres d'ordre k.
      On considère $a_{k+1} = (r, [t_0, \cdots, t_k])$ un arbre binomial d'ordre $k+1$.
      le nombre de nœuds de a_{k+1} est égal à la somme du nombre de nœuds de t_k et $(r, [t_0, \cdots, t_{k-1}])$, deux arbres binomiaux d'ordre k.
      L'hypothèse de récurrence assure alors que le nombres de nœuds de a_{k+1} est égal $2^k + 2^k$, ce qui permet de conclure.

12. Le code de la fonction s'inspire directement de la définition 6.

```
def EstUnArbreBinomial(a):
  assert not Vide(a), "arbre vide"
  r,f = Racine(a), Fils(a)
  return f==[] or (EstUnArbreBinomial(f[-1]) and EstUnArbreBinomial((r,f[:-1])) and
      Racine(f[-1])>= r)
```

II.2 - Tas binomial

13. Un tas binomial a besoin d'être mutable (en lisant les questions suivantes), la structure de tuple ne convient donc pas pour coder un tas. La structure de liste est adaptée.
14. Le résultat de la question 11, la définition de la signature et la définition de la signature, assurent que :
$$|T| = \sum_{i=0}^{k-1} s_i 2^i + 2^k$$
on en déduit que :
$$2^k \leqslant |T| \leqslant \underbrace{\sum_{i=0}^{k} 2^i}_{=2^{k+1}-1} < 2^{k+1}$$
15. Le code proposé s'appuie sur le fait que l'on sait que le dernier arbre de la liste n'est pas vide et que sa racine est le plus petit élément :

```
def MinimumTas(T):
  m = Racine(T[-1:])
  for a in reversed(T):
    if not Vide(a):
      m = min(m,Racine(a))
  return m
```

La complexité est clairement linéaire sur $|T|$.
16. Le code demandé suit l'algorithme proposé par l'énoncé.

```
def Insertion(T,p):
  n , a , i = len(T), (p,[]), 0
  while i<n and not Vide(a) :
    if Vide(T[i]) :
      T[i] = a
      a = None
    else :
      a = ArbreBinomial(a,T[i])
      T[i] = None
    i += 1
  if not Vide(a):
    T.append(a)
  return T
```

17. Dans le pire des cas, lorsque les arbres binomiaux a_0 à a_{k-1} sont vides, la boucle de l'algorithme est parcourue pour i de 0 à k.
Le résultat de la question 14 assure donc que la complexité de l'algorithme d'insertion est en $O(\log(|T|))$.
18. On note $s_0 \cdots s_k$ la signature de T.

- Si pour tout $i \in [\![0,k]\!]$, $s_i = 1$, la signature du tas résultant après insertion de p dans T est $s'_0 \cdots s'_{k+1}$ avec pour tout $i \in [\![0,k]\!]$, $s'_i = 0$ et $s'_{k+1} = 1$.
- Sinon, on peut considérer $q \overset{\text{def}}{=} \min\{i \in [\![0,k-1]\!], \quad s_i = 0\}$, la signature $s'_0 \cdots s'_k$ avec pour tout $i \in [\![0,q-1]\!]$, $s'_i = 0$, $s'_q = 1$ et pour tout $i \in [\![q+1,k]\!]$, $s'_i = s_i$.

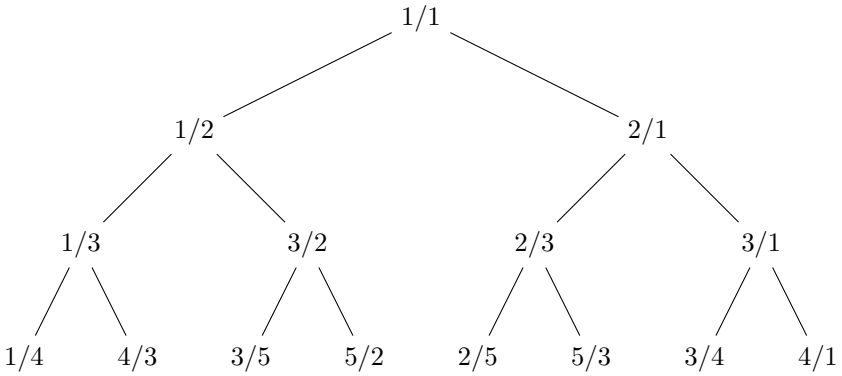
Plus simplement, si on considère la signature de T comme le code binaire d'un nombre entier pour lequel s_0 correspond au poids faible, la signature du tas résultant après insertion de p dans T est le code binaire du nombre entier obtenu en ajoutant 1 au nombre entier codé par la signature de T.

19. Invariant de boucle :

$$\begin{gathered} i \leqslant k+1 \\ \wedge \\ \text{si } j < i \text{ alors } a_j \text{ est vide.} \\ \wedge \\ \text{si } j \geqslant i \ a_j \text{ est vide ou d'ordre } j. \\ \wedge \\ p \text{ est une valeur de } a \text{ ou de } a_i. \end{gathered}$$

Partie III - Autour de l'énumération des fractions positives

20. La représentation de l'arbre de Calkin-Wilf jusqu'à la profondeur 3 est :



21. Pour pouvoir représenter des arbres de profondeur finie, on distingue les nœuds des feuilles et on propose le type suivant :

```
type acw = F of fraction | N of acw * fraction * acw ;;
```

22. Montrons par récurrence sur le niveau d'exploration que si n/d est un nœud de l'arbre alors $n \wedge d = 1$.
    - Au niveau 0, la seule fraction représentée est $1/1$ et le résultat est immédiat.
    - On considère $k \in \mathbb{N}$ et on suppose que tous les nœuds du niveau k sont de la forme n/d avec $n \wedge d = 1$.
      On peut donc considérer un couple de Bézout $(u,v) \in \mathbb{Z}^2$ associé au couple (n,d) :

      $$un + vd = 1$$

      On a alors :

      $$(u-v)n + v(n+d) = 1, \qquad u(n+d) + (v-u)d = 1$$

      Ces deux égalités assurent que $d \wedge n+d = n \wedge (n+d) = 1$. Les couples $(n+d,d)$, $(n,n+d)$ sont donc des couples de nombres entiers premiers entre eux. Ce qui permet de conclure.

23. On s'appuie sur l'algorithme d'Euclide :

```
let rec pgcd m n =
  if m mod n = 0 then n else pgcd n (m mod n) ;;
```

24. On propose un code qui tient compte de la phrase "Au besoin, elle simplifie la fraction par $n \wedge d$". On pourrait faire la simplification dans tous les cas.

```
let fraction n d =
  if n <= 0 || d <= 0 then failwith "fracirreduc" else
   begin
    let delta = pgcd n d in match delta with
     | 1 -> {n=n;d=d}
     | _ -> {n=n/delta;d=d/delta}
   end;;
```

25. Avec les notations de l'énoncé, on pose :

$$N_1 \overset{\text{def}}{=} \frac{k}{l}, \quad N_2 \overset{\text{def}}{=} \frac{n}{d}$$

On suppose que les fils gauches sont égaux, alors :

$$\frac{N_1}{N_1+1} = \frac{N_2}{N_2+1}$$

Le résultat de la question 22 assure que les fractions $\frac{N_1}{N_1+1}$ et $\frac{N_2}{N_2+1}$ sont irréductibles.
Par unicité du représentant irréductible d'un nombre rationnel positif, on en déduit que $N_1 = N_2$.
Puisque les fractions $\frac{k}{l}$ et $\frac{n}{d}$ sont irréductibles et égales on en déduit que :

$$k = n, \quad l = d$$

On démontre de la même façon le résultat si les fils droits sont égaux.

26. Le résultat se démontre par récurrence sur $k \in \mathbb{N}$ en s'appuyant sur le fait que le fils gauche de N est $\frac{N}{1+N}$ (*Les détails ne sont pas donnés dans cette correction.*)
On montre de même que le nœud provenant de d'une suite de k fils droits de N est $N + k$.

27. En s'appuyant sur l'arbre représenté dans la question 20, on obtient :

i	0	1	2	3	4	5	6	7
v_i	0	1	1/2	2	1/3	3/2	2/3	3/1

28.
- Pour $n + d = 2$, on a $n = d = 1$, puisque $v_1 = 1$ le résultat est vrai pour $n + d = 2$.
- On considère $k \in \mathbb{N}\setminus\{0,1\}$ et on suppose que pour tout $(n,d) \in (\mathbb{N}^*)^2$, tel que $n \wedge d = 1$ et $n + d \leqslant k$, la fraction $\frac{n}{d}$ apparaît dans l'arbre.
  On considère alors deux nombres entiers naturels non-nuls n et d tels que $n \wedge d = 1$ et $n + d = k + 1$.

  * Si $\frac{n}{d} < 1$, alors n et $d - n$ sont deux nombres entiers naturels non-nuls premiers entre eux et :
  $$n + (d - n) \leqslant k$$
  l'hypothèse de récurrence assure alors que la fraction rationnelle $\frac{n}{d-n}$ apparaît dans l'arbre. Or $\frac{n}{d}$ est le fils gauche de ce nœud, la fraction $\frac{n}{d}$ apparaît donc dans l'arbre.
  * Si $\frac{n}{d} > 1$, alors $n - d$ et d sont deux nombres entiers naturels non-nuls premiers entre eux et :
  $$(n - d) + d \leqslant k$$
  l'hypothèse de récurrence assure alors que la fraction rationnelle $\frac{n-d}{n}$ apparaît dans l'arbre. Or $\frac{n}{d}$ est le fils droit de ce nœud, la fraction $\frac{n}{d}$ apparaît donc dans l'arbre.

  On peut donc conclure que pour tout couple de nombres entiers naturels non-nuls (n,d) premiers entre eux, la fraction n/d apparaît dans l'arbre.

29. On va montrer par récurrence que pour tout $k \in \mathbb{N}$, l'arbre de Calkin-Wilf de profondeur k, noté CW_k ne répète aucun nœud.

- Pour $k = 0$, le résultat est immédiat.
- On considère $k \in \mathbb{N}$ et on suppose le résultat vrai pour CW_k.
  Supposons que l'arbre CW_{k+1} admet deux nœuds distincts de même valeur N.

  * Si $N < 1$, il s'agit nécessairement de deux fils gauches. Or l'application :
  $$\begin{array}{ccc} \mathbb{Q}^{+*} & \to & \mathbb{Q} \\ x & \mapsto & \dfrac{x}{x+1} \end{array}$$
  est injective. Les deux nœuds parents sont donc égaux et de profondeur strictement inférieure à k.
  Cela contredit l'hypothèse de récurrence.

∗ Si $N > 1$, on conclut de la même façon en raisonnant sur les fils droit et en utilisant le caractère injectif de l'application :

$$\begin{array}{ccc} \mathbb{Q}^{+*} & \to & \mathbb{Q} \\ x & \mapsto & x+1 \end{array}$$

Pour tout $k \in \mathbb{N}$, l'arbre CW_k est donc sans répétition, ce qui permet de conclure.

30. Le résultat de la question 28 assure que l'application :

$$\begin{array}{ccc} \mathbb{N}^* & \to & \mathbb{Q}^+ \\ i & \mapsto & v_i \end{array}$$

est surjective. Le résultat de la question 29 assure qu'elle est injective.
$v_0 = 0$ permet de conclure que la suite $(v_i)_{i\in\mathbb{N}}$ définit une bijection de $\mathbb{N}$ dans $\mathbb{Q}^+$.

31. On montre classiquement qu'à la profondeur k, il y a 2^k nœuds d'indices variant de 2^k à $2^{k+1}-1$. Pour $i \in [\![2^k, 2^{k+1}-1]\!]$, les 2^k fils gauches sont les nœuds d'indices pairs obtenus en multipliant l'indice du nœud père par 2.
Puisque chaque fils droit est d'indice consécutif de l'indice du fils gauche, on obtient le résultat souhaité.

32. On a :

i	0	1	2	3	4	5	6	7	8	9
s_i	0	1	1	2	1	3	2	3	1	4

33. Le code suivant s'appuie sur la définition récursive de la fonction.

```
let rec stern i = match i with
 | 0                  -> 0
 | 1                  -> 1
 | n  when n mod 2 = 0 -> stern (n/2)
 | n                  -> let i = (n-1)/2 in (stern i) + (stern (i+1));;
```

34. Montrons par récurrence sur la profondeur k des nœuds, que pour tout $i \in \mathbb{N}$:

$$v_i = \frac{s_i}{s_{i+1}}$$

- Pour $k = 0$, le résultat découle immédiatement de la définition de s_0 et s_1.
- On considère $k \in \mathbb{N}$ et on suppose le résultat vrai pour k.
  Les nœuds de profondeurs $k+1$ sont des nœuds fils des nœuds de profondeur k. En notant i l'indice d'un nœud de profondeur k, il s'agit donc de montrer que :

$$v_{2i} = \frac{s_{2i}}{s_{2i+1}}, \quad v_{2i+1} = \frac{s_{2i+1}}{s_{2i+2}}$$

  Or :

$$v_{2i} = \frac{v_i}{v_i+1}$$

  l'hypothèse de récurrence assure alors que :

$$v_{2i} = \frac{\dfrac{s_i}{s_{i+1}}}{\dfrac{s_i}{s_{i+1}}+1} = \frac{s_i}{s_i+s_{i+1}} = \frac{s_{2i}}{s_{2i+1}}$$

  De même :

$$v_{2i+1} = v_i + 1 = \frac{s_i}{s_{i+1}} + 1 = \frac{s_i+s_{i+1}}{s_i} = \frac{s_{2i+1}}{s_{2i}}$$

  ce qui permet de conclure.

35. Dans le premier cas, on peut considérer $j \in \mathbb{N}^*$ tel que $i = 2j$ et $i+1 = 2j+1$.
Le résultat de la question 34 assure alors que :

$$v_i = \frac{s_{2j}}{s_{2j+1}}, \quad v_{i+1} = \frac{s_{2j+1}}{s_{2j+2}}$$

Soit encore :

$$v_i = \frac{s_j}{s_j + s_{j+1}}, \quad v_{i+1} = \frac{s_j + s_{j+1}}{s_{j+1}}$$

Or :

$$\frac{s_j}{s_j + s_{j+1}} = 1 - \underbrace{\frac{s_{j+1}}{s_j + s_{j+1}}}_{\in]0,1[}$$

Donc $\left\lfloor \frac{s_j}{s_j+s_{j+1}} \right\rfloor = 0$ et :

$$f\left(\frac{s_j}{s_j + s_{j+1}}\right) = \frac{1}{1 + 2\left\lfloor \frac{s_j}{s_j+s_{j+1}} \right\rfloor - \frac{s_j}{s_j+s_{j+1}}} = \frac{1}{1 - \frac{s_j}{s_j+s_{j+1}}} = \frac{s_j + s_{j+1}}{s_{j+1}} = v_{i+1}$$

ce qui permet de conclure.

36. Le résultat de la question 26 permet d'établir par une récurrence rapide que la dernier nœud à droite à la profondeur k, correspond à

$$v_{2^{k+1}-1} = k + 1$$

On montre de même que le nœud suivant, qui est le nœud le plus à gauche de la profondeur $k+1$ vérifie :

$$v_{2^{k+1}} = \frac{1}{k+2}$$

Par ailleurs :

$$f(k+1) = \frac{1}{1 + 2\lfloor k+1 \rfloor - k - 1} = \frac{1}{k+2}$$

ce qui permet de conclure.

37. Pour construire le chemin, on part de $\frac{n}{d}$ et on remonte jusqu'à la racine.

- Si $\frac{n}{d} < 1$ le parent du nœud est $\frac{n}{d-n}$.
- Si $\frac{n}{d} > 1$ le parent du nœud est $\frac{n-d}{d}$.

On réitère le procédé jusqu'à obtenir la racine.
On peut noter que l'on n'a pas besoin de la fonction `rev` pour obtenir le résultat souhaité.

```
let chemin frac =
 let rec cons res n d = match n,d with
  | 1,1            -> res
  | n,d when n/d<1 -> cons (G::res) n (d-n)
  | n,d            -> cons (D::res) (n-d) d
 in cons [] frac.n frac.d;;
```

38. La fonction `direction` est construite sur le même principe :

```
let direction liste =
 let rec cons n d liste = match liste with
  | []     -> {n=n;d=d}
  | D::fin -> cons (n+d) d fin
  | G::fin -> cons n (n+d) fin
 in cons 1 1 liste;;
```

39. On obtient un ancêtre commun en atteignant le nœud obtenu en parcourant le plus long chemin commun depuis la racine vers les deux nœuds.
Le code de la fonction `ancetre` en découle.
On utilise la fonction auxiliaire `commun` qui détermine le plus long préfixe commun à deux listes.

```
let ancetre f1 f2 =
 let rec commun res l1 l2 = match l1,l2 with
  | [],l2                          -> res
  | l1,[]                          -> res
  | d1::fin1,d2::fin2 when d1 = d2 -> commun (d1::res) fin1 fin2
```

```
  | _,_                              -> res
in let ch1,ch2 = chemin f1, chemin f2 in
direction (commun [] ch1 ch2);;
```

40. Puisque v_i et v_{i+1} ont des indices consécutifs, si v_p est le premier ancêtre commun k' niveaux au-dessus des deux nœuds, les chemins de v_p à v_i et de v_p à v_{i+1} sont respectivement de la forme :

$$[\texttt{G},\texttt{D},\cdots,\texttt{D}],\quad [\texttt{D},\texttt{G},\cdots,\texttt{G}]$$

Où les deux fins de listes sont de longueur $k'-1$ et sont composées respectivement de `D` uniquement, de `G` uniquement.

41. Les résultats des questions 26 et 40 assurent que si $v_p = \dfrac{n}{d}$, alors :

$$v_i = \frac{n}{n+d} + k' - 1,\quad v_{i+1} = \frac{N}{1+(k'-1)N} \text{ où } N = \frac{n+d}{d}$$

Puisque $\dfrac{n}{n+d} \in]0,1[$, on a :

$$\lfloor v_i \rfloor = \left\lfloor \frac{n}{n+d} + k' - 1 \right\rfloor = k' - 1$$

Donc :

$$f(v_i) = \frac{1}{k' - \frac{n}{n+d}} = \frac{n+d}{k'(n+d)-n}$$

Par ailleurs :

$$\frac{N}{1+(k'-1)N} = \frac{\frac{n+d}{d}}{1+(k'-1)\frac{n+d}{d}} = \frac{n+d}{d+(k'-1)(n+d)} = \frac{n+d}{k'(n+d)-n}$$

ce qui permet de conclure.