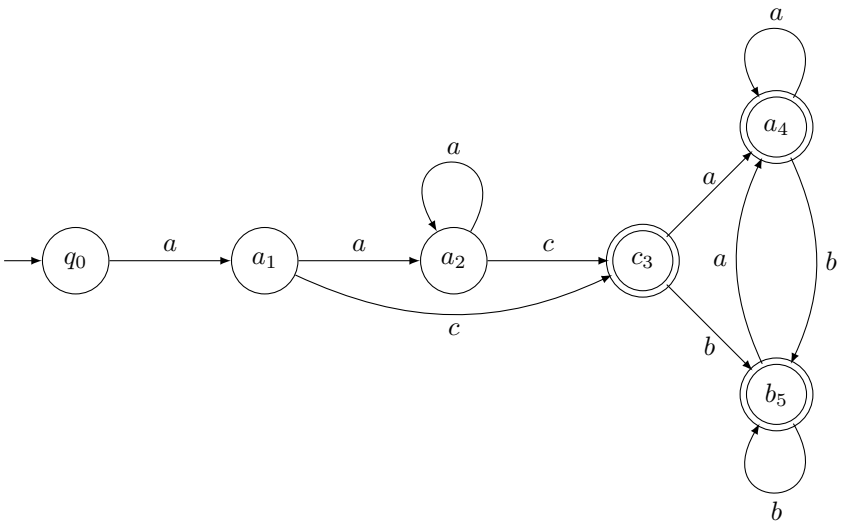
Exercice 1

1. On propose $r = aa^\star c(a|b)^\star$.
2. On commence par linéariser r en $r' = a_1a_2^\star c_3(a_4|b_5)^\star$. On calcule ensuite :
   - $P(r') = \{a_1\}$,
   - $S(r') = \{c_3, a_4, b_5\}$,
   - $F(r') = \{a_1a_2, a_1c_3, a_2a_2, a_2c_3, c_3a_4, c_3b_5, a_4a_4, a_4a_5, b_5a_4, b_5b_5\}$.

   On obtient donc l'automate :

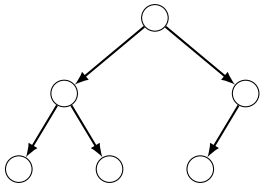


3. L'automate précédent est bien déterministe. Ce n'est pas nécessairement le cas pour tout automate de Glushkov. Par exemple, si l'expression avait été $aa^\star(c|\varepsilon)(a|b)^\star$, on aurait eu un automate similaire, où a_2 aurait été final, et avec des transitions de a_1 et a_2 vers a_4 et b_5, étiquetées par a (vers a_4) et b (vers b_5). L'automate ainsi obtenu n'aurait pas été déterministe.
4. Soit $L \in \mathrm{Rat}(\Sigma)$. Alors il existe un entier $n \in \mathbb{N}^*$ tel que tout mot $u \in L$ tel que $|u| \geqslant n$ admet une décomposition de la forme xyz telle que :
   - $y \neq \varepsilon$;
   - $|xy| \leqslant n$;
   - $\forall k \in \mathbb{N}, xy^kz \in L$ ou, de manière équivalente, $xy^\star z \subseteq L$.
5. Supposons que L' est rationnel. Considérons n la longueur de pompage donnée par le lemme, et posons $u = a^nba^nb$. Par le lemme, on peut écrire $u = xyz$ vérifiant les trois conditions du lemme. Les deux premières impliquent que $xy \in a^*$ et $y = a^k$, $k > 0$. Dès lors, $xz = a^{n-k}ba^nb$ ne peut pas s'écrire comme vv avec $v \in \{a, b\}^*$. En effet, si k est impair, alors xz est de longueur impaire. Si k est pair, alors $k \geqslant 2$ et on aurait $v = a^{n-k}ba^{k/2} = a^{n-k/2}b$ ce qui est absurde. On conclut par l'absurde que L' n'est pas rationnel.
6. On remarque que $L'' = \{a, b\}^*$ est bien un langage rationnel. En effet, tout mot $v \in \{a, b\}^*$ peut s'écrire $v = \varepsilon\varepsilon v$ et $\varepsilon \in \{a, b\}^*$.

   On remarque que $L''' \cap \{a, b\}^* = L'$. Si L''' était rationnel, alors L' le serait aussi, car les langages rationnels sont clos par intersection. On conclut que L''' n'est pas rationnel.

Exercice 2

1. Il n'existe qu'un seul arbre parfait de taille 6. On peut le représenter par :



2. La structure de tas binaire peut s'implémenter avec des arbres presque-parfaits. Elle permet de réaliser une file de priorité efficace.
3. On propose la définition suivante :
   - l'arbre constitué d'un seul nœud est un arbre parfait de hauteur 0 ;
   - pour $h > 0$, l'arbre constitué d'un nœud et ayant pour enfants gauche et droit l'arbre parfait de hauteur $h-1$ est un arbre parfait de hauteur h.

   Montrons qu'elle coïncide avec celle de l'énoncé. Appelons temporairement « parfait (bis) » un arbre défini de cette manière.
   - Montrons par récurrence sur h qu'un arbre parfait A de hauteur h est un arbre parfait (bis) :
     * si $h = 0$, alors A est constitué d'un seul nœud dont est parfait (bis) ;
     * supposons la propriété établie jusqu'à $h \in \mathbb{N}$ fixé. Soit A un arbre parfait de hauteur $h+1$. Comme A est binaire strict, il possède deux enfants. De plus, chacun de ses enfants est de hauteur h (car toutes les feuilles sont à même profondeur), est binaire strict et possède chacun toutes ses feuilles à même profondeur. Les enfants sont donc des arbres parfaits de hauteur h, donc des arbres parfaits (bis) de hauteur h par hypothèse de récurrence. On en conclut que A est bien un arbre parfait (bis) de hauteur $h+1$.

     On conclut par récurrence.
   - Montrons par induction qu'un arbre parfait (bis) est un arbre parfait :
     * comme précédemment, le cas de base est assuré ;
     * supposons que A est un arbre parfait de hauteur h. Alors l'arbre $B = N(A, A)$ reste binaire strict et possède toutes ses feuilles à même profondeur. Il est donc parfait.
4. On propose la fonction suivante (on rappelle qu'il n'existe pas de fonction `max` en C) :

```c
int hauteur(arbre* a){
    if (a == NULL){
        return -1;
    } else {
        int hg = hauteur(a->gauche);
        int hd = hauteur(a->droite);
        return 1 + ((hg<hd)?hd:hg);
    }
}
```

5. On utilise la définition inductive. Les cas de base sont l'arbre vide et la feuille.

```c
bool est_parfait(arbre* a){
    if (a == NULL || (a->gauche == NULL && a->droite == NULL)) {
        return true;
    } else {
        int hg = hauteur(a->gauche);
        int hd = hauteur(a->droite);
        hg == hd && est_parfait(a->gauche) && est_parfait(a->droite);
    }
}
```

On remarque que la fonction `hauteur` a une complexité qui est linéaire en la taille de l'arbre. On obtient donc pour `est_parfait` une complexité vérifiant :

$$C(a) = \begin{cases} \mathcal{O}(1) & \text{si } |a| \leqslant 1 \\ C(g) + C(d) + \mathcal{O}(|a|) & \text{si } a = N(g,d) \text{ et } |a| > 1 \end{cases}$$

Pour un arbre parfait a de taille n, on a donc comme formule de récurrence : $C(n) = 2C\left(\frac{n-1}{2}\right) + \mathcal{O}(n)$. On reconnait une formule qui se résout en $C(n) = \mathcal{O}(n \log n)$.

6. Pour réussir à écrire une fonction efficace, il faut remarquer qu'un arbre presque-parfait de hauteur $h > 0$ a pour enfants gauche et droit :
   - soit un arbre parfait de hauteur $h-1$ et un arbre presque-parfait de hauteur $h-1$;
   - soit un arbre presque-parfait de hauteur $h-1$ et un arbre parfait de hauteur $h-2$.

   en remarquant qu'un arbre parfait est un arbre presque-parfait particulier. Ainsi, on peut écrire la fonction en utilisant une fonction intermédiaire qui calcule pour chaque nœud un quadruplet contenant :
   - la taille et la hauteur du sous-arbre ;
   - un booléen qui détermine si le sous-arbre est parfait ;
   - un booléen qui détermine si le sous-arbre est presque-parfait.

   Si on connait ces informations pour les deux enfants d'un nœud, alors on peut les calculer pour ce nœud en temps constant. Cela donnera bien une complexité linéaire en la taille de l'arbre.
   On implémente dans un premier temps une structure de quadruplet :

```
struct Quad{
    int t;
    int h;
    bool parf;
    bool pparf;
};

typedef struct Quad quad;
```

On commence alors par écrire une fonction auxiliaire qui fait ces calculs, tout en mettant à jour un pointeur de pointeur d'arbre et un pointeur d'entier, pointant vers le plus grand sous-arbre presque-parfait et sa taille respectivement.

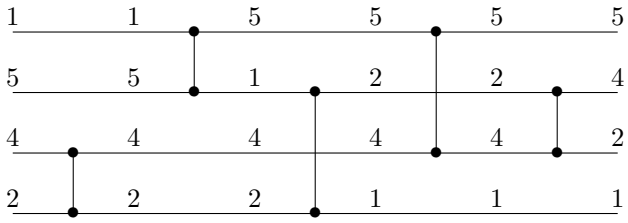
```
quad pgpp_aux(arbre* a, arbre** best, int* tbest){
    if (a == NULL){
        quad q = {.t = 0, .h = -1, .parf = true, .pparf = true};
        return q;
    } else {
        quad qg = pgpp_aux(a->gauche, best, tbest);
        quad qd = pgpp_aux(a->droite, best, tbest);
        quad q = {.t = 1 + qg.t + qd.t,
                  .h = 1 + ((qg.h<qd.h)?qd.h:qg.h),
                  .parf = qg.h == qd.h && qg.parf && qd.parf,
                  .pparf = (qg.h == qd.h && qg.parf && qd.pparf) ||
                           (qg.h == qd.h + 1 && qg.pparf && qd.parf)};
        if (q.pparf && q.t > *tbest){
            *best = a;
            *tbest = q.t;
        }
        return q;
    }
}
```

enfin, on peut écrire la fonction finale :

```
arbre* plus_grand_presque_parfait(arbre* a){
    arbre* best = NULL;
    int tbest = 0;
    pgpp_aux(a, &best, &tbest);
    return best;
}
```

Exercice 3

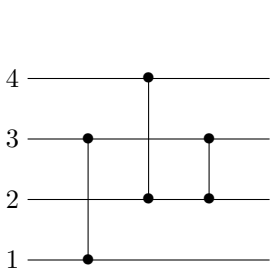
1. On obtient le déroulement suivant :

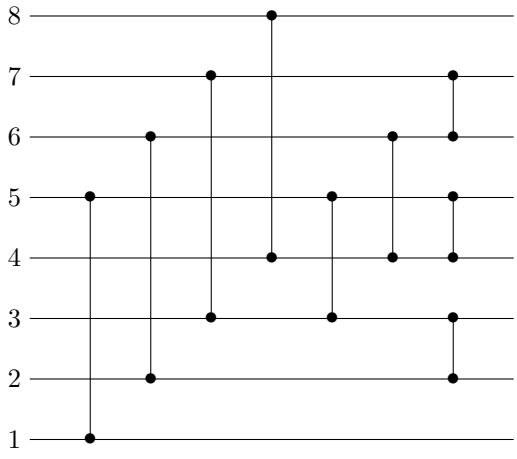


   Après les actions des deux premiers modules, les indices 1 et 3 contiennent les minimums de $\{T[1], T[2]\}$ et $\{T[3], T[4]\}$ respectivement (et de même pour les indices 2 et 4 qui contiennent les maximums). Après l'action du module $(1, 3)$, l'indice 1 contient le minimum du tableau. Après l'action de $(2, 4)$, l'indice 4 contient le maximum du tableau. Enfin, après l'action du module $(2, 4)$, les deux derniers éléments sont placés correctement. Ce réseau de tri trie effectivement tout tableau de taille 4.

2. On peut implémenter un tri par insertion par exemple : en notant R_k le réseau $(k-1, k), (k-2, k-1), \ldots, (1, 2)$, il suffit de concaténer R_2, R_3, …, R_n. Il est clair que R_k contient $k-1$ modules, donc ce réseau contient $\frac{n(n-1)}{2}$ modules.

3. On s'inspire de la première question :





   Le deuxième réseau illustre l'idée des questions suivantes.

4. Il est clair que c_1 est le plus petit élément du tableau, car $c_1 = \min(a_1, b_1)$. De même, d_{2n} est clairement la plus grande valeur du tableau.

   Supposons que c_p soit la k-ème valeur dans le tableau $[a_1, a_3, \ldots, a_{2n-1}]$. On a donc $c_p = a_{2k-1}$. Il y a donc $2k-2$ éléments de a plus petits que c_p. Par ailleurs, comme c_p est en p-ème position dans le tableau d, il y a $p-1-(k-1) = p-k$ valeurs b_{2j-1} plus petites que c_p, soit $2(p-k)$

ou $2(p-k)-1$ valeurs plus petites que c_p dans b. Par conséquent, il a $2p-2$ ou $2p-3$ valeurs plus petites que c_p dans le tableau constitué de a et b trié. c_p est donc à la $2p-1$ ou $2p-2$-ème position dans le tableau.

Un raisonnement similaire permet de trouver que d_p se trouve à la $2p$ ou $2p+1$-ème position, ce qui permet de conclure.

5. Pour fusionner deux tableaux de taille 2^k, on fusionne les sous-tableaux d'indices pairs et les sous-tableaux d'indices impairs, puis on utilise 2^k-1 modules pour les dernières comparaisons. En notant $M(k)$ le nombre de modules pour fusionner deux tableaux de taille 2^k, on a :
   - $M(0)=1$;
   - pour $k>0$, $M(k)=2M(k-1)+2^k-1$.

   On en déduit que $M(k)=k2^k+1$ (par une récurrence rapide), soit $n\log_2 n+1$ modules nécessaires pour fusionner deux tableaux de taille n qui est une puissance de 2.

6. Le principe ici est d'utiliser les n fils pour faire les fusions récursives, en ajoutant une barrière de synchronisation à chaque étape. On peut mettre en œuvre une barrière de synchronisation pour k fils en utilisant un tourniquet. Le principe est le suivant :

```
Fonction Créer(k)
    Renvoyer {compteur = k, m = Mutex(), s = Semaphore(0)}
```

```
Fonction Rendez_vous(B)
    Verrouiller B.m.
    B.compteur ← B.compteur − 1.
    Si B.compteur = 0 Alors
        Libérer B.s.
    Déverouiller B.m.
    Attendre B.s.
    Libérer B.s.
```

   Pour effectuer la fusion, il faut alors prendre en argument l'indice de début d et le pas p (initialement 1 et 1), fusionner récursivement un début d et un pas $2p$ d'une part, et un début $d+p$ et un pas $2p$ d'autre part, passer la barrière de synchronisation pour n/p fils, puis appliquer les modules $(d+p, d+2p), (d+3p, d+4p), \ldots$

7. C'est le principe du tri fusion :
   - on trie un tableau de taille 1 sans module ;
   - on trie un tableau de taille 2^{k+1} en triant chaque moitié de taille 2^k, puis en fusionnant ces deux moitiés.

   En notant $M(k)$ le nombre de modules utilisés pour trier un tableau de taille 2^k, on a :
   - $M(0)=0$;
   - pour $k>0$, $M(k)=2M(k-1)+k2^k+1$.

   À nouveau, on montre par récurrence que $M(k)=k(k+1)2^{k-1}+2^k-1$, soit $\mathcal{O}(n(\log n)^2)$ pour trier un tableau de taille n.

Exercice 4

1. $2,2$ est une séquence d'élimination : soit c un chemin de longueur 1, $c=u_0u_1$. On distingue les cas :
   - Si $u_0=2$, alors les séquences se rencontrent.
   - Sinon, $u_0=1$ ou 3, mais alors $u_1=2$.

2. Supposons qu'il existe une séquence d'élimination $\sigma=v_1\ldots v_n$. Construisons alors un chemin qui ne rencontre jamais σ. Soit C un cycle de longueur $k\geqslant 3$ dans le graphe. On pose u_1 un sommet de C différent de v_1 (il en existe un). Dès lors, on construit u_{i+1} en choisissant l'un des voisins

de u_i dans C, qui est différent de v_{i+1}. Comme u_i a toujours deux voisins dans le cycle, cette construction est possible. On conclut par l'absurde.

3. Si $c = v_1 \dots v_n$ est un chemin dans le graphe, remarquons que v_i et v_j ont la même parité si et seulement si i et j ont la même parité.
   De plus, si $n = k$ et v_1 est impair, alors la séquence $\sigma = 1, 2, \dots, k$ rencontre c. En effet, si on suppose que que les deux séquences ne se rencontrent pas, il existe $i \in [\![1; k]\!]$ tel que $i < v_i$ et $i_1 > v_{i+1}$. C'est impossible, car si $i < v_i$, alors $v_i \geqslant i+1$ (car v_i a la même parité que i). Dès lors, on propose la séquence d'élimination suivante (l'idée étant de faire deux parcours, en s'assurant que l'un a la même parité que le chemin à rencontrer) :
   $$\sigma = 1, 2, \dots, k-1, k, k, k-1, k-2, \dots, 2, 1$$

4. Informellement, $\phi_E(X)$ est l'ensemble des sommets qui n'ont que des voisins dans X. Dès lors, avec les hypothèses de l'énoncé, soit $c = v_1 \dots v_n v_{n+1}$ un chemin dont le dernier élément est dans $\phi_E(X) \cup \{v\}$.
   - Si $v_{n+1} = v$, alors c et σ' se rencontrent bien (en $i = n+1$).
   - Sinon, $v_{n+1} \in \phi_E(X)$ et donc, par définition, tout voisin de v_{n+1} est dans X. On en déduit que $v_n \in X$, et par définition de σ que σ rencontre $v_1 \dots v_n$ et donc que σ' rencontre c.

5. On ramène à un problème d'accessibilité dans un graphe. On construit le graphe orienté $G' = (V', E')$ tel que :
   - $V' = \mathcal{P}(V)$
   - Pour $A \in V'$ et $v \in V$, $(A, \phi_E(A) \cup \{v\}) \in E'$. On peut considérer que l'arête est étiquetée par v.

   S'il existe un chemin de $\emptyset$ à V dans G', alors, d'après la question précédente et par une récurrence, il existe une séquence d'élimination pour G. De plus, le chemin permet de construire la séquence d'élimination.
   Réciproquement, s'il existe une séquence d'élimination pour G, $\sigma = v_1 \dots v_n$. Par construction du graphe, il existe un unique chemin étiqueté par σ passant par les sommets $V_1, \dots, V_n$. Montrons que ce chemin termine en V en montrant que si $u \notin V_i$, alors il existe un chemin c se terminant par u tel que c ne rencontre pas $\sigma_i = v_1 \dots v_i$.
   - Le résultat est trivialement vrai pour V_1 (qui ne contient qu'un seul élément).
   - Supposons le résultat vrai pour $i < n$. On sait que $V_{i+1} = \phi_E(V_i) \cup \{v_i\}$. Supposons $u \notin V_{i+1}$. On sait donc que $u \neq v_i$ et $u \notin \phi_E(V_i)$. On en déduit qu'il existe $u' \notin V_i$ tel que u et u' sont voisins. Dès lors, il existe un chemin c terminant par u' qui ne rencontre pas σ_i et donc cu ne rencontre pas $\sigma_{i+1} = \sigma_i v_i$.

   Dès lors, comme $\sigma_n = \sigma$ est une séquence d'élimination, on en déduit que $V_n = V$.
   La complexité d'un tel algorithme est exponentielle :
   - La construction du graphe G' (dont le nombre de sommets est $2^{|V|}$) est exponentielle et nécessite le calcul des arêtes (on calcule $\phi_E(A)$ pour chaque ensemble).
   - La recherche du chemin se fait en temps $O(|V'|+|E'|)$ ou $O(|V'|^2)$ selon la structure (parcours en largeur par exemple). À nouveau, la complexité est exponentielle.
   - La longueur du chemin est potentiellement exponentielle également.

6. Soit $\sigma' = (1, 0, 2, 0, \dots k)$ et $\sigma = \sigma'\sigma'$. Alors $\sigma = (s_1, \dots, s_{4k-2})$ est une séquence d'élimination pour S_k.
   Appelons sommet impair un sommet $i \in [\![1, k]\!]$ et pair un des autres. Comme à la question 3, un chemin change toujours de parité à chaque étape. Si le premier sommet est un sommet impair, alors il y aura une rencontre pendant le premier passage de σ'. De même pour le deuxième passage si le premier sommet est pair.

7. Soit $\sigma = v_1 \dots v_n$ une séquence d'éléments de V, ne commençant pas par 0. On définit un chemin c de la façon suivante :
   - $u_1 = 0$

- Pour $i \in [\![1; n-1]\!]$:
  - Si $u_i = j''$, alors $u_{i+1} = j'$ (seul choix possible)
  - Si $u_i = j'$, alors $u_{i+1} = j$ sauf si $v_{i+1} = j$, auquel cas $u_{i+1} = j''$
  - Si $u_i = j$, alors $u_{i+1} = 0$ sauf si $v_{i+1} = 0$, auquel cas $u_{i+1} = j'$
  - Si $u_i = 0$, alors $u_{i+1} = j$ où j est choisi de la façon suivante : $j \neq v_{i+1}$ et on pose, pour $k \in \{1, 2, 3\} \setminus \{v_{i+1}\}$: $i < \ell_k < n$ la plus petite position après i dont la parité est $\neq i$ et qui satisfait $v_{\ell_k} = k$ et $v_{\ell_k+1} = k'$. Si l'un des deux ℓ_k n'est pas défini, on pose $j = k$. Sinon, on pose $j = k$ tel que ℓ_k est maximal.

On définit ici bien un chemin, et il ne rencontre pas σ. Le seul cas de rencontre possible serait lorsque $u_i = 0$, mais on choisit le bon sommet pour éviter une éventuelle collision par la suite.