Licence d'informatique
Algorithmique – LU3IN003
Fascicule de TD (1ère partie)

année 2024–2025

Manuel Amoussou
Mohamed Ouaguenouni
Fanny Pascual
Olivier Spanjaard

Rappels sur les preuves d'algorithmes et l'analyse de complexité

Exercice 1 – Notations de Landau (exercice de base)

Q 1.1 Démontrer que :

1. $n^2 \in O(10^{-4}n^3)$
2. $25n^4 + 13n^2 \in O(n^4)$
3. $2^{n+100} \in O(2^n)$
4. $n^5 \in \Omega(10n^3)$
5. $20n + 5n^2 \in \Theta(n^2)$

Q 1.2 Soient f, g, h, k quatres fonctions de $\mathbb{N}$ dans $\mathbb{N}$. Dire si les implications suivantes sont vraies ou fausses, et dans ce dernier cas donner un contre exemple.

		Vrai	Faux	Eléments de preuve ou contre-exemple
1	$\left.\begin{array}{l} f \in O(h) \\ g \in O(h) \end{array}\right\} \Rightarrow f+g \in O(h)$			
2	$\left.\begin{array}{l} f \in O(h) \\ g \in O(h) \end{array}\right\} \Rightarrow fg \in O(h)$			
3	$\left.\begin{array}{l} f \in O(h) \\ g \in O(k) \end{array}\right\} \Rightarrow fg \in O(hk)$			
4	$\left.\begin{array}{l} f \in \Omega(h) \\ g \in \Omega(h) \end{array}\right\} \Rightarrow f+g \in \Omega(h)$			
5	$\left.\begin{array}{l} f \in O(h) \\ g \in \Omega(h) \end{array}\right\} \Rightarrow f+g \in \Theta(h)$			
6	$\left.\begin{array}{l} f \in O(h) \\ g \in \Theta(h) \end{array}\right\} \Rightarrow f+g \in \Theta(h)$			
7	$f \in O(h) \Rightarrow f \geqslant h$			
8	$f \in O(h) \Rightarrow f \leqslant h$			

Q 1.3 Montrer que, si a est un réel positif, alors $f(n) = 1 + a + a^2 + \ldots + a^n$ est en :

(a) $\Theta(1)$ si $a < 1$.

(b) $\Theta(n)$ si $a = 1$.

(c) $\Theta(a^n)$ si $a > 1$.

Exercice 2 – Complexité au pire cas (exercice de base)

Le but de cet exercice est de tester votre compréhension de la notion de complexité dans le pire des cas.

Q 2.1 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $O(n^2)$, est-il possible qu'il soit en $O(n)$ sur *toutes* les données ?

Q 2.2 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'il soit en $O(n)$ sur *certaines* données ?

Q 2.3 Si je prouve que la complexité dans le pire des cas d'un algorithme est en $\Theta(n^2)$, est-il possible qu'elle soit en $O(n)$ sur *toutes* les données ?

Exercice 3 – Échelle de croissance (exercice de base)

Pour classer des fonctions, on utilisera les notations suivantes :

$$\begin{array}{ll} f(n) < g(n) & \text{si } f(n) \in O(g(n)) \text{ et } f(n) \notin \Theta(g(n)) \\ f(n) \equiv g(n) & \text{si } f(n) \in \Theta(g(n)) \end{array}$$

Sur une échelle croissante, classer les fonctions suivantes selon leur comportement asymptotique :

$$\begin{array}{llll} f_1(n) = 2\,n & f_2(n) = 2^n & f_3(n) = \log(n) & f_4(n) = \frac{n^3}{3} \\ f_5(n) = n! & f_6(n) = [\log(n)]^2 & f_7(n) = n^n & f_8(n) = n^2 \\ f_9(n) = n + \log(n) & f_{10}(n) = \sqrt{n} & f_{11}(n) = \log(n^2) & f_{12}(n) = e^n \\ f_{13}(n) = n & f_{14}(n) = \sqrt{\log(n)} & f_{15}(n) = 2^{\log_2(n)} & f_{16}(n) = n\,\log(n) \end{array}$$

Exercice 4 – Division Euclidienne (exercice de base)

On considère l'algorithme DivisionEuclidienne (page 4), qui prend en arguments deux entiers $a \geq 0$ et $b > 0$ et retourne l'entier $\lfloor \frac{a}{b} \rfloor$.

Algorithme : DivisionEuclidienne
Entrées : a : entier, b : entier
$R \leftarrow a$; $Q \leftarrow 0$;
tant que $R \geq b$ **faire**
  $R \leftarrow R - b$;
  $Q \leftarrow Q + 1$;
retourner Q

Q 4.1 Exécutez cet algorithme avec $a = 14$ et $b = 3$. Vous indiquerez à chaque itération les valeurs de R et de Q.

Q 4.2 Montrez que cet algorithme se termine et est valide.

Exercice 5 – Fonction Q mystère (exercice de base)

On considère l'algorithme récursif suivant, qui prend en entrée un entier $n \geq 1$:

```
fonction Q(n)
    si n = 1 retourner 1
    sinon retourner Q(n − 1) + 2 ∗ n − 1
```

Q 5.1 Calculer les valeurs $Q(n)$ pour $n=1$ à $n=5$. En déduire une expression simple de la valeur retournée par $Q(n)$.

Q 5.2 Prouver ce résultat par récurrence.

Exercice 6 – Fonctions F et G mystères (exercice d'entraînement)

Soit la fonction récursive F suivante, qui prend en argument un entier $n \geq 1$:

```
Algorithme : Fonction F(n)
si n = 1 alors
    retourner 2;
sinon
    retourner F(n − 1) + F(n − 1);
```

Q 6.1 Que calcule cette fonction ? Prouver la terminaison de l'algorithme.

Q 6.2 Déterminer la complexité de la fonction F si l'on suppose que la somme de deux entiers de n bits se fait en $\Theta(n)$ (en posant l'addition de deux nombres binaires comportant chacun n bits). Comment améliorer cette complexité ?

Soit la fonction itérative G suivante, qui prend en argument un entier $n \geq 1$:

```
Algorithme : Fonction G(n)
R ← 2;
pour i variant de 1 à n − 1 faire
    R ← R + R;
retourner R;
```

Q 6.3 Prouver que $G(n)$ retourne la même valeur que $F(n)$.

Soient a et b deux nombres entiers positifs, avec $a \geq b$. On souhaite effectuer la multiplication de a par b. Les deux exercices suivants visent à étudier et comparer deux algorithmes résolvant ce problème.

Exercice 7 – Multiplication intuitive (exercice d'entraînement)

On considère l'algorithme suivant :

Algorithme : Multiplie1(entier a, entier b)
$p \leftarrow 0$;
$m \leftarrow 0$;
tant que $m < a$ **faire**
  $p \leftarrow p + b$;
  $m \leftarrow m + 1$;
retourner p

Q 7.1 Prouvez que l'algorithme se termine et est valide.

Q 7.2 Quelle est sa complexité si l'on suppose que la somme de deux entiers de n bits se fait en $\Theta(n)$?

Exercice 8 – Multiplication russe (exercice d'approfondissement)

On peut améliorer l'algorithme précédent à l'aide de la récurrence :

$$x \times y = \begin{cases} (x/2) \times 2y \text{ si } x \text{ est pair} \\ ((x-1)/2) \times 2y + y \text{ sinon} \end{cases}$$

L'algorithme itératif ci-dessous met en œuvre cette récurrence. Les opérations de multiplication et de division par deux sont simples pour un codage binaire puisqu'elles consistent à décaler d'un bit vers la gauche ou vers la droite. On supposera donc que multiplier ou diviser par deux un entier de n bits se fait en $\Theta(n)$. On supposera également que la somme de deux entiers de n bits se fait en $\Theta(n)$.

Algorithme : Multiplie2(entier a, entier b)
$prod \leftarrow 0$;
$x \leftarrow a$;
$y \leftarrow b$;
tant que $x > 0$ **faire**
  si x *est impair* **alors**
    $prod \leftarrow prod + y$;
    $x \leftarrow x - 1$;
  $x \leftarrow x/2$;
  $y \leftarrow 2 \times y$;
retourner $prod$

Q 8.1 Donner la trace de l'algorithme (valeurs de x, y et $prod$) pour $a = 11$ et $b = 4$.

Q 8.2 Démontrer que la propriété suivante est un invariant de boucle : *À la fin de la i-ème itération,* $prod + x \times y = a \times b$. Autrement dit : pour toute itération i, on a $prod_i + x_i \times y_i = a \times b$, où x_i, y_i et $prod_i$ sont les valeurs de x, y et $prod$ à la fin de l'itération i. En déduire la correction de l'algorithme.

Q 8.3 Quelle est la complexité de cet algorithme ?

Exercice 9 – Les tours de Hanoi (exercice avec corrigé vidéo)

Petit divertissement mathématique mis au point par Édouard Lucas en 1883 : au départ, on dispose de 3 piquets. Le premier porte n disques de tailles toutes différentes et empilés du plus grand (en bas) au plus petit (en haut). Le problème consiste à déplacer tous ces disques jusqu'au troisième piquet sachant que l'on respecte les règles suivantes :

— on ne déplace qu'un disque à la fois ;

— on ne peut déplacer qu'un disque se trouvant en haut d'une pile ;

— aucun disque ne doit être empilé sur un disque de diamètre inférieur.

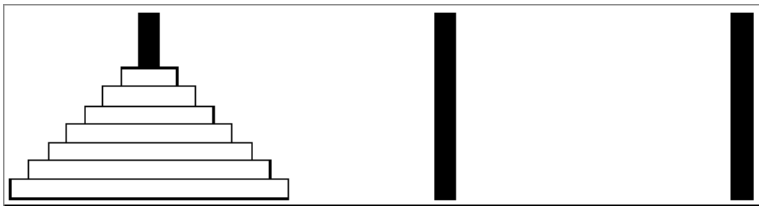


FIGURE 1 – état initial

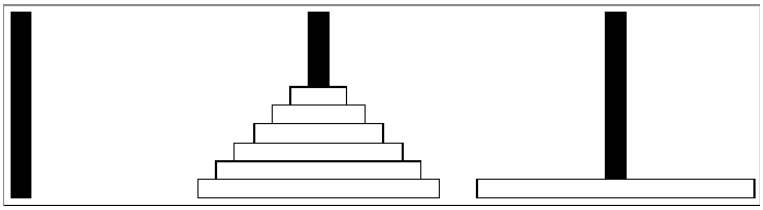


FIGURE 2 – état intermédiaire

L'algorithme récursif *Deplacer* permet de résoudre ce problème, avec :

— *nombre* : nombre de disques utilisés,

— *Bouger_un_disque*(*origine*, *destination*) : déplace le disque se trouvant en haut de la pile du piquet *origine* vers le piquet *destination*.

Algorithme : $Deplacer(nombre, piquet1, piquet3, piquet2)$
si $nombre > 0$ **alors**
  $Deplacer(nombre - 1, piquet1, piquet2, piquet3)$;
  $Bouger\_un\_disque(piquet1, piquet3)$;
  $Deplacer(nombre - 1, piquet2, piquet3, piquet1)$;

Q 9.1 On appellera $D(n)$ le nombre de déplacements de disques effectués par l'algorithme pour n disques. Donner la complexité de cet algorithme en nombre de déplacements de disques et le prouver.

En supposant qu'un être humain met une seconde à déplacer un disque, combien de temps lui faudra-t-il pour résoudre ce casse-tête pour une tour de Hanoi comportant 36 disques ?

Q 9.2 Montrer que tout algorithme résolvant le problème des tours de Hanoï nécessite au minimum $2^n - 1$ déplacements de disques.

Algorithmique – LU3IN003

Programmation récursive

Diviser pour régner

Exercice 1 – Application du théorème maître (exercice de base)

Le but de cet exercice est d'utiliser le théorème maître pour déterminer des complexités à partir d'équations de récurrence. En utilisant le théorème maître, donner un ordre de grandeur asymptotique pour $T(n)$ où $\frac{n}{2}$ peut valoir $\lceil \frac{n}{2} \rceil$ ou $\lfloor \frac{n}{2} \rfloor$:

Q 1.1 $T(n) = 2T\left(\frac{n}{2}\right) + n^2$;

Q 1.2 $T(n) = 2T\left(\frac{n}{2}\right) + n$;

Q 1.3 $T(n) = 2T\left(\frac{n}{2}\right) + \sqrt{n}$;

Q 1.4 $T(n) = T\left(\frac{n}{2}\right) + c$ avec $c > 0$.

Exercice 2 – Arbre des appels récursifs (exercice de base)

Considérons un algorithme dont le nombre d'opérations sur une entrée de taille n est donné par la relation de récurrence suivante :

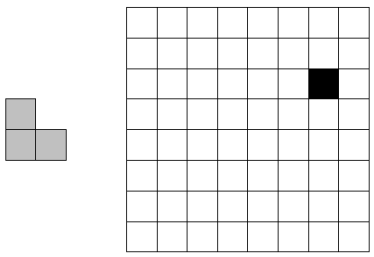
$$T(1) = 0 \quad \text{et} \quad T(n) = T\left(\left\lceil \frac{n}{3} \right\rceil\right) + T\left(\left\lfloor \frac{2\,n}{3} \right\rfloor\right) + n.$$

Q 2.1 Construire l'arbre des appels récursifs de l'algorithme. En déduire la complexité de $T(n)$.

Q 2.2 Peut-on retrouver le résultat de la question précédente à l'aide du théorème maître ?

Exercice 3 – Triominos (exercice d'entraînement)

Un *triomino* est une tuile en forme de L formée de trois carrés 1×1. Le problème est de couvrir par des triominos un échiquier "troué" $2^n \times 2^n$. Les triominos peuvent être orientés de façon arbitraire, mais ils doivent couvrir toutes les cases de l'échiquier (exceptée la case manquante) sans chevauchement.



Q 3.1 Proposer une méthode de type « diviser pour régner » pour résoudre ce problème.

Q 3.2 Prouver la validité de la méthode proposée.

Q 3.3 On considère maintenant un échiquier carré comportant un nombre pair de cases, privé de deux angles diagonalement opposés, qu'on cherche à couvrir de dominos sans chevauchement. Peut-on trouver un algorithme ?

Exercice 4 – Majoritaire (exercice avec corrigé vidéo)

Un élément x est majoritaire dans un multi-ensemble E de n éléments si le nombre d'occurences de x est $> n/2$. La seule opération dont on dispose est la comparaison (égalité ou non) de deux éléments.

Le but de l'exercice est d'étudier le problème suivant : étant donné un multi-ensemble E, représenté par un tableau, existe-t-il un élément majoritaire, et si oui quel est-il ?

Q 4.1 Voilà une idée pour un algorithme naïf.

1. Écrire un algorithme NombreOccurrences qui, étant donné un tableau E, un élément x et deux indices i et j, calcule le nombre d'occurrences de x entre $E[i]$ et $E[j]$. Quelle est sa complexité (en nombre de comparaisons) ?
2. Écrire un algorithme Majoritaire qui vérifie si un tableau E contient un élément majoritaire, et si oui le retourne. Quelle est sa complexité ?

Q 4.2 Décrire le principe d'un algorithme de type « diviser pour régner » pour résoudre ce problème.

Q 4.3 Écrire un algorithme reposant sur le principe décrit à la question précédente. Analyser sa complexité.

Q 4.4 Donner le principe d'un algorithme permettant de résoudre le problème en $O(n)$ si on suppose que les éléments de E sont à valeurs dans $\{1, \cdots, k\}$, k étant une constante.

Exercice 5 – Identification d'espions (exercice d'entraînement)

Dans un contexte de guerre froide, des espions se sont glissés parmi les civils travaillant dans un centre de recherche et ont volé des données secrètes de la plus haute importance. Arrive alors sur les lieux un inspecteur qui cherche à démasquer les espions. Il peut organiser des confrontations entre deux personnes, celles-ci pouvant alors reconnaître l'autre comme un civil, ou au contraire l'accuser d'être un espion.

Il sait que les espions peuvent mentir (ou dire la vérité), mais que les civils disent toujours la vérité. Chaque civil sait qui est civil et qui est espion. Ainsi, lors d'une confrontation entre deux personnes, les déclarations des civils sont toujours exactes. Pour chaque couple de déclarations lors d'une confrontation entre deux personnes A et B, le tableau ci-dessous indique ce que l'on peut déduire sur le *statut* (espion ou civil) des deux personnes confrontées.

A déclare :	B déclare :	conclusion sur le statut de A et B
"B est un civil"	"A est un civil"	A et B sont tous les deux civils, ou tous les deux espions
"B est un civil"	"A est un espion"	il y a au moins un espion parmi A et B
"B est un espion"	"A est un civil"	il y a au moins un espion parmi A et B
"B est un espion"	"A est un espion"	il y a au moins un espion parmi A et B

On suppose qu'il y a **strictement plus de civils que d'espions**. On note n le nombre total de personnes (l'ensemble des civils et des espions).

Q 5.1 En supposant que l'inspecteur a réussi à identifier un civil, expliquer comment il peut identifier tous les espions. Donner la complexité en nombre de confrontations (en fonction de n).

On se concentre donc maintenant sur le problème d'identifier **un** civil parmi les n personnes (pour pouvoir identifier ensuite tous les espions). Nous allons étudier deux méthodes pour ce faire.

I) Première méthode : approche naïve.

Q 5.2 Expliquer comment déterminer en $O(n)$ confrontations si une personne est civil ou espion.

Q 5.3 En répétant simplement cette procédure jusqu'à ce qu'on détermine qu'une personne est un civil, quelle est la complexité au pire cas (en nombre de confrontations) pour trouver un civil ?

II) Deuxième méthode : approche diviser pour régner.

On considère maintenant l'algorithme suivant :

- **Si** $n=1$ **ou** $n=2$, toutes les personnes sont des civils (puisqu'il y a strictement plus de civils que d'espions).

- **Si** n **est impair**, on choisit au hasard une personne, et on utilise la méthode de la question 2 pour déterminer si cette personne est un civil ou un espion :
  - Si c'est un civil, on a résolu le problème.
  - Sinon, on élimine cette personne. On résout alors récursivement le problème pour les $n-1$ personnes restantes, sachant que $n-1$ est pair.

- **Si** n **est pair**, on sépare les personnes en deux files de $m=\frac{n}{2}$ personnes, et on fait se confronter les premiers de chaque file, puis les deux seconds et ainsi de suite. Pour chaque confrontation :
  - Si les deux personnes se déclarent l'une l'autre comme civil, on en élimine une des deux.
  - Sinon, on élimine les deux.

  Une fois les m confrontations réalisées, on résout récursivement le problème pour les personnes restantes (remarquons qu'il y en a au plus m).

Lors d'un appel récursif de la méthode diviser pour régner, notons P l'ensemble initial de personnes et P' l'ensemble de personnes qui n'ont pas été éliminées. **On suppose qu'il y a strictement plus de civils que d'espions dans P.** Pour montrer que l'algorithme est correct, on va montrer qu'il y a toujours strictement plus de civils que d'espions dans P'.

Q 5.4 Soit n le nombre de personnes dans P. On suppose que n est impair, et que la personne choisie au hasard n'est pas un civil. Expliquer pourquoi il y a plus de civils que d'espions dans P'.

On suppose dans la question qui suit que le nombre n de personnes dans P est pair, et on considère l'ensemble des $m=\frac{n}{2}$ confrontations induites par les deux files (les confrontations entre deux personnes au même rang dans les files). Soit :

— m_{ee} le nombre de confrontations dans lesquelles deux espions s'affrontent (on parle ici du vrai *statut* des personnes, et non des déclarations),

— m_{cc} le nombre de confrontations dans lesquelles deux civils s'affrontent,

— m_{ce} le nombre de confrontations dans lesquelles un civil et un espion s'affrontent.

On a bien sûr $m_{ee}+m_{cc}+m_{ce}=m$. Etant donné un ensemble X de personnes, on notera $n_c(X)$ le nombre de civils dans X, et $n_e(X)$ le nombre d'espions dans X.

Q 5.5 On suppose que n est pair. L'objet des questions a), b) et c) est de montrer qu'il y a nécessairement plus de civils que d'espions dans P'.

a) Donner l'expression de $n_e(P)$ en fonction de m_{ce} et m_{ee}, et de $n_c(P)$ en fonction de m_{ce} et m_{cc}.

b) Montrer que $m_{cc} > m_{ee}$.

c) Montrer que $n_c(P') > n_e(P')$.

Les questions 4 et 5 prouvent que si il y a plus de civils que d'espions dans P, alors cela restera vrai dans P'. Ainsi, lorsqu'il ne restera plus qu'une ou deux personnes dans P', ces personnes seront bien des civils. Cet algorithme est donc bien valide.

Q 5.6 Déterminer la complexité de cet algorithme en nombre de confrontations en fonction du nombre initial n de personnes. Justifiez votre réponse.

Exercice 6 – Nombre d'intersections de segments (exercice d'entraînement)

Dans cet exercice, on considère deux droites parallèles (d_1) et (d_2), ainsi qu'un ensemble de n segments de la forme $s_i = [p_i, q_i]$, où p_i est un point de (d_1) et q_i est un point de (d_2). Pour simplifier, on suppose que les segments n'ont aucune extrémité commune, et que les segments sont numérotés de sorte que les indices des points p_i sont croissants de la gauche vers la droite, comme sur l'exemple de la figure 3. L'objectif est de concevoir un algorithme pour déterminer le nombre d'intersections de segments.

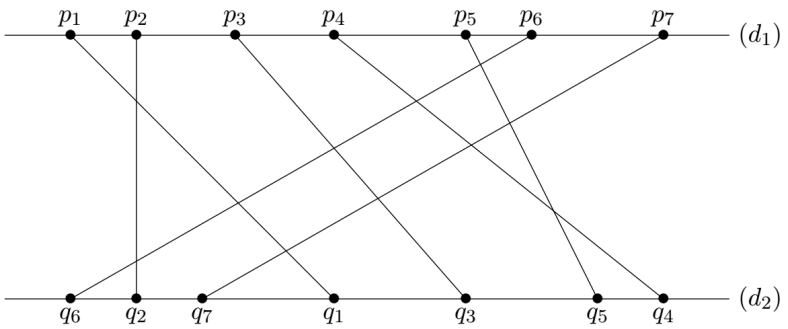


FIGURE 3 – Une instance du problème de comptage d'intersections.

On considère le tableau de n entiers, noté A, dont les cellules 1 à n comportent les indices j des extrémités q_j successivement rencontrées en parcourant la droite (d_2) de la gauche vers la droite. Autrement dit, $A[i] = j$ si le point q_j correspond à la $i^{\text{ème}}$ extrémité la plus à gauche sur (d_2). Ainsi, le tableau A associé à l'exemple de la figure 3 est le suivant : $A = [6, 2, 7, 1, 3, 5, 4]$.

Q 6.1 Soient $i \in \{1, \ldots, n\}$ et $j \in \{1, \ldots, n\}$. Une paire (i, j) est une *inversion* pour le tableau A si et seulement si $i < j$ et $A[i] > A[j]$. Donnez toutes les inversions pour le tableau de la question précédente.

Q 6.2 Comparer le nombre d'inversions que vous avez trouvées à la question précédente avec le nombre d'intersections de segments dans l'exemple. Que constatez-vous ? Est-ce vrai pour toutes les instances de ce problème ? Justifier brièvement.

Q 6.3 En vous aidant de la question précédente, donner en quelques mots le principe d'un algorithme naïf permettant de compter le nombre d'intersections de segments. Quelle est la complexité de cet algorithme ?

On s'intéresse à présent à un algorithme de type diviser pour régner, appelé Trier-et-Compter, dont le pseudo-code est donné ci-après, afin de compter le nombre d'inversions dans un tableau A. Dans la suite de l'exercice, on notera $|A|$ la taille d'un tableau A.

Entrées : A : un tableau d'entiers
si $|A| = 1$ **alors retourner** $(0, A)$
sinon
  Diviser le tableau A en deux moitiés A_1 et A_2
  $(m_1, A'_1) \leftarrow$ Trier-et-Compter(A_1)
  $(m_2, A'_2) \leftarrow$ Trier-et-Compter(A_2)
  $(m_3, A') \leftarrow$ Fusionner-et-Compter(A'_1, A'_2)
retourner $(m_1 + m_2 + m_3, A')$

Algorithme 1 : Trier-et-Compter

L'algorithme Trier-et-Compter est fondé sur l'observation suivante. L'ensemble des inversions (i, j) pour le tableau A peut être partitionné en trois catégories :

1. L'ensemble des inversions (i, j) avec $i \leq n/2$ et $j \leq n/2$.
2. L'ensemble des inversions (i, j) avec $i > n/2$ et $j > n/2$.
3. L'ensemble des inversions (i, j) avec $i \leq n/2$ et $j > n/2$.

Le nombre d'inversions du premier type, noté m_1, est obtenu par un appel récursif sur la première moitié du tableau A (notée A_1). De même, le nombre d'inversions du second type, noté m_2, est obtenu par un appel récursif sur la deuxième moitié de A (notée A_2). Pour le nombre d'inversions du troisième type (noté m_3), on fait appel à un algorithme, nommé Fusionner-et-Trier (voir pseudo-code, où $|A|$ correspond à la taille d'un tableau A), qui prend en entrée les sous-tableaux A_1 et A_2 préalablement triés (notés A'_1 et A'_2), et qui réalise les deux opérations suivantes simultanément :

— Le calcul de la valeur de m_3.

— Le tri du tableau A en fusionnant les tableaux triés A'_1 et A'_2 (le tableau trié est noté A').

Enfin, Trier-et-Compter termine en retournant le nombre d'inversions du tableau A (c'est-à-dire $m_1 + m_2 + m_3$) ainsi que le tableau trié A'.

Q 6.4 Compléter la ligne 5 de Fusionner-et-Compter avec un calcul en $O(1)$ afin que m_3 corresponde bien au nombre d'inversions du troisième type au terme de la boucle tant que.

```
Entrées : A'_1 et A'_2 : deux tableaux triés
i ← 1 ; j ← 1 ; A' ← [ ] ; m_3 ← 0
tant que i ≤ |A'_1| et j ≤ |A'_2| faire
    si A'_1[i] > A'_2[j] alors
        Ajouter A'_2[j] à la fin du tableau A'        // instruction en O(1)
        m_3 ← .....
        j ← j + 1
    sinon
        Ajouter A'_1[i] à la fin du tableau A'        // instruction en O(1)
        i ← i + 1
tant que i ≤ |A'_1| faire
    Ajouter A'_1[i] à la fin du tableau A'            // instruction en O(1)
    i ← i + 1
tant que j ≤ |A'_2| faire
    Ajouter A'_2[j] à la fin du tableau A'            // instruction en O(1)
    j ← j + 1
retourner (m_3, A')
```

Algorithme 2 : Fusionner-et-Compter

Indication : Supposons que $A'_1 = [2, 6, 7]$ et $A'_2 = [1, 3, 4, 5]$ en entrée de Fusionner-et-Compter.
Pour $i = 1$ et $j = 1$, on détecte 3 inversions impliquant l'entier 1 car $A'_1[i] = 2 > 1 = A'_2[j]$.
Pour $i = 1$ et $j = 2$, pas d'inversion supplémentaire impliquant 2 car $A'_1[i] = 2 < 3 = A'_2[j]$.
Pour $i = 2$ et $j = 2$, on détecte 2 inversions impliquant l'entier 3 car $A'_1[i] = 6 > 3 = A'_2[j]$. Etc.

Q 6.5 Quelle est la complexité de l'algorithme Fusionner-et-Compter ? Justifier brièvement votre réponse.

Q 6.6 Soit $T(n)$ le nombre d'opérations effectuées par l'algorithme Trier-et-Compter. Donner la formule de récurrence vérifiée par T. A l'aide d'un théorème vu en cours, en déduire la complexité de l'algorithme.

Exercice 7 – Produit matriciel (exercice d'approfondissement)

Le but de cet exercice est de réaliser un produit de deux matrices carrées X et Y de dimension $n \times n$ en utilisant l'approche *diviser pour régner*. Un produit de matrices est particulièrement facile à diviser en sous-problèmes car il peut être réalisé *par blocs*. Pour voir ce que cela signifie, partitionnons X en quatre blocs $n/2 \times n/2$, ainsi que Y :

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Le produit XY peut être réalisé par blocs de la façon suivante :

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Q 7.1 On suppose que l'addition de deux scalaires se fait en $\Theta(1)$. Quelle est la complexité (en fonction de n) de l'opération de fusion des réponses aux sous-problèmes ?

Q 7.2 Soit $T(n)$ le nombre d'additions de scalaires à effectuer pour réaliser un produit de matrices carrées $n \times n$. Ecrire l'équation de récurrence vérifiée par $T(n)$. En déduire la complexité de la méthode en fonction de n.

On suppose désormais qu'on divise le problème en réalisant les calculs suivants :

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

où

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

Q 7.3 On vérifie facilement que la complexité de l'opération de fusion reste inchangée. Ecrire l'équation de récurrence vérifiée par $T(n)$ pour cette nouvelle méthode de produit matriciel. En déduire la complexité.

Exercice 8 – Sous-tableau maximum (exercice d'approfondissement)

Dans cet exercice, on s'intéresse au *problème du sous-tableau maximum*. Etant donné un tableau $A[1, \ldots, n]$, le problème du sous-tableau maximum de A est de déterminer :

$$\text{stm}(A) = \max\{\sum_{k=i}^{j} A[k] : 1 \leq i \leq j \leq n\}$$

Par exemple, on a $\text{stm}([-3, 4, 5, -1, 2, 3, -6, 4]) = 13$, valeur que l'on obtient pour le sous-tableau $[4, 5, -1, 2, 3]$. Dans un premier temps, on considère l'algorithme de résolution de la page 17.

Q 8.1 Quelle est la complexité de cet algorithme ?

Q 8.2 Proposer une variante en $\Theta(n^2)$ de cet algorithme.

```
max = max{A[1], ..., A[n]};
pour i = 1 ··· n faire
    pour j = i ··· n faire
        temp ← 0;
        pour k = i ··· j faire
            temp ← temp + A[k];
        si temp > max alors
            max ← temp;
retourner max
```

Dans la suite de l'exercice, on s'intéresse à des algorithmes de type "diviser pour régner" pour ce problème. Soit $m = \lfloor (n+1)/2 \rfloor$, $A_1 = A[1, \ldots, m]$ et $A_2 = A[m+1, \ldots, n]$. Le cas de base est le cas d'un tableau de taille 1, qu'on sait bien sûr résoudre en temps constant. Soit $\text{stm}_1 = \text{stm}(A_1)$, $\text{stm}_2 = \text{stm}(A_2)$ et stm_3 la valeur maximum d'un sous-tableau qui contient à la fois $A[m]$ et $A[m+1]$.

Q 8.3 Comment calculer stm_3 efficacement ? Quelle est la complexité en fonction de n ?

Q 8.4 Dans le cadre d'une méthode « diviser pour régner », supposons que l'on a calculé stm_1 et stm_2 (par des appels récursifs). En vous aidant de la réponse à la question précédente, donner le principe d'une opération de fusion des résultats des deux appels, et indiquer la complexité de cette opération.

Q 8.5 Soit $T(n)$ la complexité de la méthode « diviser pour régner » pour un tableau de taille n. Donner l'équation de récurrence satisfaite par $T(n)$. A l'aide du théorème maître, en déduire la complexité de la méthode.

On s'intéresse désormais à une autre approche, toujours de type "diviser pour régner", afin de réduire encore la complexité de l'algorithme. Pour ce faire, on définit :

$$\begin{aligned} \text{pref}(A) &= \max\{\textstyle\sum_{k=1}^{i} A[k] : 1 \le i \le n\}, \\ \text{suff}(A) &= \max\{\textstyle\sum_{k=i}^{n} A[k] : 1 \le i \le n\}, \\ \text{tota}(A) &= \textstyle\sum_{k=1}^{n} A[k], \end{aligned}$$

Autrement dit, $\text{pref}(A)$ (resp. $\text{suff}(A)$) est la valeur maximum d'un sous-tableau préfixe (resp. suffixe) de A, et $\text{tota}(A)$ est la somme des cases de A.

Q 8.6 Supposons qu'on vise à déterminer non plus simplement $\text{stm}(A)$, mais le triplet $\text{stm}(A)$, $\text{pref}(A)$, $\text{suff}(A)$. Soit $(\text{stm}_1, \text{pref}_1, \text{suff}_1)$ et $(\text{stm}_2, \text{pref}_2, \text{suff}_2)$ les triplets retournés par les appels récursifs sur A_1 et A_2. Expliquer pourquoi ce serait faux de faire la fusion en retournant $(\max\{\text{suff}_1 + \text{pref}_2, \text{stm}_1, \text{stm}_2\}, \text{pref}_1, \text{suff}_2)$ pour A.

Q 8.7 Supposons maintenant qu'on vise à déterminer le quadruplet $(\text{stm}(A), \text{pref}(A), \text{suff}(A), \text{tota}(A))$. Soit $(\text{stm}_1, \text{pref}_1, \text{suff}_1, \text{tota}_1)$ et $(\text{stm}_2, \text{pref}_2, \text{suff}_2, \text{tota}_2)$ les quadruplets retournés par les appels récursifs sur A_1 et A_2. Donner la formule permettant de déterminer le quadruplet pour A à partir de stm_1, pref_1, suff_1, tota_1, stm_2, pref_2, suff_2, tota_2. Quelle est la complexité de cette opération de fusion ?

Q 8.8 Soit $T(n)$ la complexité de la méthode « diviser pour régner » correspondante pour un tableau de taille n. Donner l'équation de récurrence satisfaite par $T(n)$. A l'aide du théorème maître, en déduire la complexité de cette nouvelle méthode.

Exercice 9 – Algorithme de Stooge (exercice avec corrigé vidéo)

On considère l'algorithme récursif ci-dessous, qui trie le sous-tableau $A[i \dots j]$.

```
Algorithme : Stooge(tableau A, entier i, entier j)
si A[i] > A[j] alors
    échanger le contenu de A[i] et A[j];
si j − i > 1 alors
    k ← (j − i + 1)/3 ; // quotient de la division entière
    Stooge(A, i, j − k) ; // premier 2/3 du tableau
    Stooge(A, i + k, j) ; // dernier 2/3 du tableau
    Stooge(A, i, j − k) ; // premier 2/3 à nouveau
retourner A;
```

L'appel initial est *Stooge(A,1,n)*, où A est un tableau indicé de 1 à n. Pour simplifier, on suppose dans tout l'exercice que n est une puissance de 3 (autrement dit, $n = 3^p$ pour un entier $p \geq 0$), ce qui garantit que le reste de la division entière est nul dans tous les appels récursifs.

Q 9.1 Montrer par récurrence forte que *Stooge(A,1,n)* trie le tableau A en ordre croissant.

Q 9.2 Soit $T(n)$ le nombre d'échanges au pire réalisé par *Stooge* sur un tableau de taille n. Ecrire la relation de récurrence permettant de calculer $T(n)$ pour $n > 2$. En déduire la complexité de l'algorithme à l'aide du théorème maître.

Q 9.3 Est-ce que *Stooge* est un bon algorithme de tri ?

Algorithmes d'exploration d'un arbre d'énumération

Exercice 1 – Boîte à coucou (exercice de base)

La boîte à coucou est une petite boîte de forme cubique à commande tactile. Toquer n fois la boîte déclenche un mécanisme qui provoque, à de multiples reprises, l'ouverture du couvercle, l'apparition d'un œuf et l'émission d'un "coucou" sonore. Le nombre d'ouvertures dépend de n.

Q 1.1 Déterminer la complexité des fonctions `F`, `G`, `H` et `J` ci-dessous, où n est un entier positif ou nul, qui correspondent à différentes variantes de l'algorithme contrôlant le nombre d'ouvertures de la boîte si l'on toque n fois. On suppose qu'une ouverture de la boîte se fait en $O(1)$.

```
Algorithme : F(n)
Ouvrir(boîte à coucou);
si n ≥ 2 alors
    F(n − 1);
    F(n − 2);
    F(n − 2);
    F(n − 2);
    F(n − 2);
    F(n − 2);
    F(n − 2)
```

```
Algorithme : G(n)
pour i allant de 1 à n faire
    Ouvrir(boîte à coucou);
si n ≥ 3 alors
    G(n − 1);
    G(n − 2);
    G(n − 3);
    G(n − 3)
```

```
Algorithme : H(n)
Ouvrir(boîte à coucou);
si n ≥ 2 alors
    H(n − 2);
    H(n − 2)
```

```
Algorithme : J(n)
pour i allant de 1 à n faire
    Ouvrir(boîte à coucou);
si n ≥ 1 alors
    J(n − 1)
```

Exercice 2 – Ascension d'un escalier (exercice de base)

On considère un escalier de n marches. Les marches n'étant pas très hautes, on peut franchir une ou deux marches à chaque pas lorsque l'on monte les escaliers. Par exemple, il y a 3 façons de monter un escalier de 3 marches : soit une marche à la fois, soit deux marches puis une marche, soit une marche puis deux marches.

Q 2.1 Proposer un algorithme récursif qui énumère toutes les façons possibles de monter un escalier de n marches.

Q 2.2 Quelle est la complexité de l'algorithme ?

Exercice 3 – Pavage d'une grille $3 \times n$ (exercice d'entraînement)

Considérons le problème qui consiste à identifier tous les pavages d'une grille $3 \times n$ à l'aide de tuile des trois formes ci-dessous, *qu'il n'est pas possible de pivoter.*

Q 3.1 Proposer un algorithme récursif retournant la liste de tous les pavages possibles.

Q 3.2 Quelle est la complexité de cet algorithme ?

Exercice 4 – Plus longue sous-chaîne commune (exercice d'entraînement)

Q 4.1 Donner un algorithme d'exploration d'un arbre d'énumération pour identifier une plus longue sous-chaîne commune entre deux chaînes de caractères u et v. Cette sous-chaîne n'a pas besoin d'être contigüe. Par exemple, si $u = abcaba$ et $v = abaccab$, l'algorithme retourne $abcab$.

Q 4.2 Analyser la complexité de l'algorithme proposé en fonction des longueurs n et m des chaînes u et v.

Exercice 5 – Procédures mutuellement récursives (exercice d'entraînement)

Dans cet exercice, on considère une chaîne s de bits (0 ou 1), indicée de 1 à n. Initialement, la chaîne s est constituée uniquement de 1. On vise à transformer s en une chaîne constituée uniquement de 0 en ne s'autorisant que les deux types suivants de transformations de s (que l'on peut réaliser autant de fois que l'on souhaite), appelées *permutations autorisées* dans la suite :

1. on peut toujours permuter la valeur du bit d'indice 1 (changer un 0 en 1, ou un 1 en 0) ;
2. si s débute par une séquence d'exactement i bits à 0 (le bit d'indice $i+1$ est 1), alors on peut permuter la valeur du bit d'indice $i+2$ (changer un 0 en 1, ou un 1 en 0).

Par exemple, pour $n = 5$, la séquence de permutations autorisées suivante permet de transformer s=11111 en la chaîne 00000 (le chiffre au dessus de chaque flèche est l'indice du bit permuté) :

$$11111 \xrightarrow{1} 01111 \xrightarrow{3} 01011 \xrightarrow{1} 11011 \xrightarrow{2} 10011 \xrightarrow{1} 00011 \xrightarrow{5} 00010$$
$$\xrightarrow{1} 10010 \xrightarrow{2} 11010 \xrightarrow{1} 01010 \xrightarrow{3} 01110 \xrightarrow{1} 11110 \xrightarrow{2} 10110 \xrightarrow{1} 00110 \xrightarrow{4} 00100$$
$$\xrightarrow{1} 10100 \xrightarrow{2} 11100 \xrightarrow{1} 01100 \xrightarrow{3} 01000 \xrightarrow{1} 11000 \xrightarrow{2} 10000 \xrightarrow{1} 00000$$

Les deux procédures *mutuellement récursives* ci-dessous (c'est-à-dire qui s'appellent l'une l'autre) permettent d'effectuer une mise à zéro de s qui fait uniquement appel à des permutations autorisées :

— MiseAZero(k), pour $k \geq 0$, produit une séquence de permutations autorisées qui change les k premiers bits de s, qui doivent être tous de valeur 1 en entrée, en une séquence de k bits 0.

— MiseAUn(k), pour $k \geq 0$, produit une séquence de permutations autorisées qui change les k premiers bits de s, qui doivent être tous de valeur 0 en entrée, en une séquence de k bits 1.

Ces séquences de permutations autorisées sont appelées séquences *valides* dans la suite. L'appel initial avec une chaîne s constituée de n bits tous de valeur 1 est MiseAZero(n). L'objectif de l'exercice est de prouver la validité de l'algorithme et d'analyser sa complexité.

```
MiseAZero(k)
  si k=1
    s[1] ← 0
  sinon si k>1
    MiseAZero(k−2)
    s[k] ← 0
    MiseAUn(k−2)
    MiseAZero(k−1)
```

```
MiseAUn(k)
  si k=1
    s[1] ← 1
  sinon si k>1
    MiseAUn(k−1)
    MiseAZero(k−2)
    s[k] ← 1
    MiseAUn(k−2)
```

Q 5.1 Donner l'arbre des appels récursifs généré par l'appel MiseAZero(3). En déduire la séquence de permutations autorisées pour convertir la chaîne 111 en 000.

Q 5.2 Prouver par récurrence que les appels MiseAZero(k) et MiseAUn(k) produisent chacun une séquence valide. En déduire la validité de MiseAZero(n) pour transformer la chaîne s complète.
Indication : Il y a deux cas de base $k = 0$ et $k = 1$. L'étape inductive consistera à montrer que si MiseAZero($k-2$), MiseAZero($k-1$), MiseAUn($k-2$) et MiseAUn($k-1$) produisent chacun une séquence valide, alors MiseAZero(k) et MiseAUn(k) également. On prouvera l'étape inductive uniquement pour MiseAZero(k) (la preuve pour MiseAUn(k) étant similaire).

Q 5.3 Soit $A(n)$ le nombre de nœuds dans l'arbre des appels récursifs de MiseAZero(n) ou MiseAUn(n) (par symétrie, le nombre est le même dans les deux cas). Donner l'équation de récurrence que vérifie $A(n)$. En déduire la complexité de MiseAZero(n).

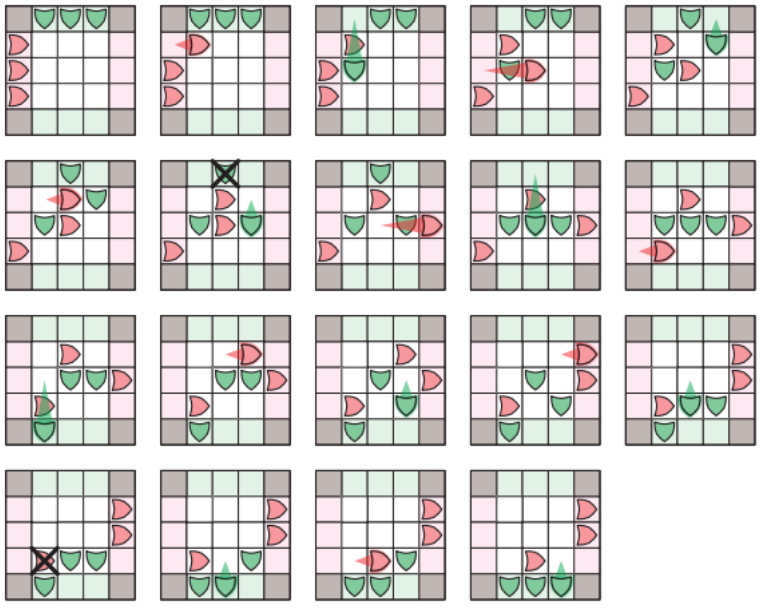
Q 5.4 Une séquence de permutations autorisées est dite *élémentaire* si elle ne comporte pas deux permutations successives dont l'une est l'inverse de la précédente (par exemple, $111 \to 011 \to 111$ n'est pas élémentaire). L'objet de cette question est de montrer que la séquence produite par MiseAZero(n) est en fait la *seule* séquence élémentaire possible pour transformer une chaîne de n bits 1 en une chaîne de n bits 0 en utilisant uniquement des permutations autorisées. Pour cela, on va s'aider d'une modélisation par un graphe et de la propriété suivante (admise) :

> **Propriété.**
> *Soit G un graphe non-orienté. Les composantes connexes de G sont uniquement des chaînes et/ou des cycles si et seulement si les degrés des sommets de G sont tous dans $\{1, 2\}$.*

Indiquer quel graphe non-orienté considérer pour prouver l'unicité de la séquence élémentaire de permutations autorisées, c'est-à-dire à quoi correspond l'ensemble des sommets et sous quelle condition une arête existe entre deux sommets. Justifier que le graphe est non-orienté. Expliquer pourquoi les degrés des sommets sont tous dans $\{1, 2\}$. Identifier les deux seuls sommets de degré 1 dans ce graphe, et en déduire le résultat recherché en vous appuyant sur le résultat de la question 2.

Exercice 6 – Jeu à deux joueurs (exercice d'approfondissement)

On considère un jeu à deux joueurs qui se joue sur une grille carrée $n \times n$. Appelons Anaïs la première joueuse et Bruno le second joueur. Chaque joueurs doit déplacer $n - 2$ jetons d'un bord à l'autre de la grille (Anaïs joue horizontalement de la gauche vers la droite, et Bruno verticalement du haut vers le bas). Le premier à avoir déplacé tous ses jetons sur le bord opposé de la grille remporte la partie. Les deux joueurs jouent à tour de rôle. A son tour, on déplace n'importe quel jeton d'une case dans la direction souhaitée (vers la droite pour Anaïs, vers le bas pour Bruno). Si un *unique* jeton adverse se trouve devant son jeton, alors on peut sauter par dessus. Si l'on ne peut pas déplacer de jeton, alors on passe son tour. Dans le cas contraire, on est obligé de jouer. Il est facile de montrer qu'il y a toujours au moins un joueur qui peut jouer, et qu'il n'y a donc pas de situation de blocage. Ci-dessous un exemple de partie remportée par Bruno.



L'*arbre de jeu* se définit comme suit : chaque nœud correspond à une position du jeu, et les enfants d'un nœud correspondent aux positions atteignables en un coup depuis la position courante (connaissant également le joueur dont c'est le tour à la position courante).

Q 6.1 Tracer une portion de l'arbre de jeu pour $n=4$.

On appelle position *gagnante* une position où le joueur dont c'est le tour a déjà gagné, ou à partir de laquelle on peut réaliser un déplacement d'un jeton qui conduit à une position perdante pour l'adversaire. Inversement, on appelle position *perdante* une position où le joueur dont c'est le tour a déjà perdu, ou à partir de laquelle tout déplacement d'un jeton conduit à une position gagnante pour l'adversaire.

Q 6.2 Dans la portion d'arbre de jeu donnée à la question précédente, indiquer pour chaque nœud s'il correspond à une position gagnante ou perdante. En déduire qu'Anaïs a une stratégie gagnante.

Q 6.3 Montrer par récurrence sur l'arbre de jeu que toute position du jeu est gagnante ou perdante pour le joueur dont c'est le tour.

Q 6.4 Donner un algorithme qui prend en entrée la position courante P du jeu et le joueur j dont c'est le tour, et retourne "Gagné" si j a gagné dans la position P, un coup à jouer pour j si la partie n'est pas terminée mais que la position est gagnante, ou "Position perdante" si P est une position perdante pour j.

Exercice 7 – Ensemble stable de cardinal maximum (exercice d'approfondissement)

Etant donné un graphe non-orienté G (représenté en machine sous forme d'un tableau de listes d'adjacence), on cherche à déterminer le cardinal maximum d'un sous-ensemble *stable* de sommets de G, c'est-à-dire sans arête entre eux. L'algorithme récursif ci-dessous, où $V(s)$ désigne l'ensemble des voisins d'un sommet s, retourne cette valeur, appelée *nombre de stabilité* de G, et notée $\alpha(G)$. L'algorithme exploite le fait que, si un ensemble stable contient un sommet s, alors il ne peut comporter aucun de ses voisins.

```
Algorithme : Alpha1(G)
Entrées : G=(S, A) : graphe non-orienté
si G est vide alors
    retourner 0
sinon
    s ← un sommet de G;
    avec_s ← 1+Alpha1(G[S − V(s) − {s}]);
    sans_s ← Alpha1(G[S − {s}]);
    retourner max{avec_s, sans_s}
```

Q 7.1 Quelle est la complexité de Alpha1 en fonction du nombre n de sommets et m d'arêtes ?

On remarque que si un sommet s n'a pas de voisin, alors tout ensemble stable de G de cardinal maximum comporte s. On propose donc la variante ci-dessous de l'algorithme récursif.

```
Algorithme : Alpha2(G)
Entrées : G=(S, A) : graphe non-orienté
si G est vide alors
    retourner 0
sinon
    si G comporte un sommet s de degré 0 alors
        retourner 1+Alpha2(G[S − {s}])
sinon
    s ← un sommet de G (de degré au moins 1);
    avec_s ← 1+Alpha2(G[S − V(s) − {s}]);
    sans_s ← Alpha2(G[S − {s}]);
    retourner max{avec_s, sans_s}
```

Q 7.2 Quelle est la complexité de Alpha2 en fonction du nombre n de sommets et m d'arêtes ?

On remarque maintenant que si un sommet s a un unique voisin s', alors à tout ensemble stable I comportant s' correspond un ensemble stable comportant s de même cardinal, à savoir $I \cup \{s\} \setminus \{s'\}$. De plus, tout ensemble stable de cardinal maximum comporte s ou s', car à un ensemble stable qui ne comporterait ni s ni s' on pourrait adjoindre s pour obtenir un ensemble stable de plus grand cardinal. On peut donc sélectionner s par défaut. On en déduit la variante Alpha3 de l'algorithme Alpha2.

Algorithme : Alpha3(G)
Entrées : $G=(S, A)$: graphe non-orienté
si *G est vide* **alors**
  retourner 0
sinon
  si *G comporte un sommet s de degré 0 ou 1* **alors**
    retourner 1+Alpha3($G[S - V(s) - \{s\}]$)
sinon
  $s \leftarrow$ un sommet de G (de degré au moins 2);
  avec_s $\leftarrow$ 1+Alpha3($G[S - V(s) - \{s\}]$);
  sans_s $\leftarrow$ Alpha3($G[S - \{s\}]$);
  retourner max{avec_s, sans_s}

Q 7.3 Quelle est la complexité de Alpha3 en fonction du nombre n de sommets et m d'arêtes ?

Le pire cas est maintenant celui d'un graphe où tous les sommets sont de degré au moins 2. Ce pire cas peut être divisé en deux sous-cas. Si $G = (S, A)$ a un sommet s de degré 3 ou plus, alors $G[S - V(s) - \{s\}]$ comporte au plus $n-4$ sommets. Sinon tous les sommets de G sont de degré 2 (car on a déjà considéré les sommets de degré 0 ou 1). Soit (s, s', s'') une chaîne de trois sommets dans G (avec possiblement s adjacent à s''). Dans tout ensemble stable maximum, soit s' est présent et s et s'' sont absents, soit s est présent et ses deux voisins sont absents, soit s'' est présent et ses deux voisins sont absents. On en déduit la variante Alpha4 de l'algorithme Alpha3.

Algorithme : Alpha4(G)
Entrées : $G=(S, A)$: graphe non-orienté
si *G est vide* **alors**
  retourner 0
sinon
  si *G comporte un sommet s de degré 0 ou 1* **alors**
    retourner 1+Alpha4($G[S - V(s) - \{s\}]$)
sinon
  si *G comporte un sommet s de degré au moins 3* **alors**
    avec_s $\leftarrow$ 1+Alpha4($G[S - V(s) - \{s\}]$);
    sans_s $\leftarrow$ Alpha4($G[S - \{s\}]$);
    retourner max{avec_s, sans_s}
  sinon
    $(s, s', s'') \leftarrow$ une chaîne de trois sommets dans G (chacun de degré 2);
    avec_s' $\leftarrow$ 1+Alpha4($G[S - \{s, s', s''\}]$);
    avec_s $\leftarrow$ 1+Alpha4($G[S - V(s) - \{s\}]$);
    avec_s" $\leftarrow$ 1+Alpha4($G[S - V(s'') - \{s''\}]$);
    retourner maxs{avec_s, avec_s', avec_s"}

Q 7.4 Quelle est la complexité de Alpha4 en fonction du nombre n de sommets et m d'arêtes ?

Pour la dernière variante de l'algorithme, il reste à remarquer que la topologie de G si tous les sommets sont de degré 2 est très particulière : G est en effet alors constitué d'un ensemble de cycles. On peut tirer parti de cette topologie particulière pour déterminer le cardinal d'un ensemble stable maximum de G sans appel récursif : pour chaque cycle de longueur p dans G, $\lfloor p/2 \rfloor$ sommets figureront dans un ensemble stable maximum (un sommet sur deux le long du cycle). On en déduit la variante Alpha5.

```
Algorithme : Alpha5(G)
Entrées : G=(S, A) : graphe non-orienté
si G est vide alors
    retourner 0
sinon
    si G comporte un sommet s de degré 0 ou 1 alors
        retourner 1+Alpha5(G[S − V(s) − {s}])
sinon
    si G comporte un sommet s de degré au moins 3 alors
        avec_s ← 1+Alpha5(G[S − V(s) − {s}]);
        sans_s ← Alpha5(G[S − {s}]);
        retourner max{avec_s, sans_s}
sinon
    // tout sommet de G est de degré 2
    total ← 0;
    pour chaque cycle de G faire
        p ← nombre d'arêtes dans le cycle;
        total ← total +⌊p/2⌋
    retourner total
```

Q 7.5 Quelle est la complexité de Alpha5 en fonction du nombre n de sommets et m d'arêtes ?

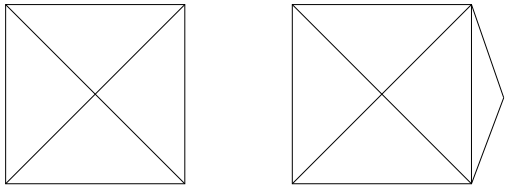
Algorithmique – LU3IN003

Rappels sur les graphes et applications des parcours en profondeur

Modélisation par les graphes et rappels sur leurs représentations

Exercice 1 – Casse-têtes (exercice de base)

Q 1.1 Est-il possible de tracer ces figures sans lever le crayon ni passer deux fois sur le même trait ?



Q 1.2 On considère un jeu constitué des 9 dominos ci-dessous. Est-il possible de les arranger en une ligne comportant les 9 dominos, en respectant la règle habituelle de contact entre dominos imposant que deux numéros adjacents doivent coïncider ?

Q 1.3 Un groupe de personnes se retrouvent pour une randonnée. Certaines personnes se connaissent, d'autres non. Montrer que, dans cette situation, il y a toujours au moins deux personnes qui connaissent exactement le même nombre de personnes dans le groupe.

Exercice 2 – Réseaux de capteurs (exercice d'entraînement)

On dispose de n capteurs sans fil représentés par n points dans un plan. Deux capteurs x et y peuvent communiquer si la distance qui les sépare est d'au plus 500 mètres ou s'il existe une chaîne de capteurs entre x et y tel que la distance entre deux capteurs adjacents de cette chaîne est d'au plus 500 mètres.

On impose la contrainte suivante : chaque capteur doit être à une distance d'au plus 500 mètres d'au moins $\frac{n}{2}$ autres capteurs. On suppose que n est pair.

Cette contrainte est-elle suffisante pour permettre à tout couple de capteurs de communiquer ?

Q 2.1 Formuler la question posée par une propriété dans un graphe non orienté à n sommets (n pair).

Q 2.2 Si la proposition est vraie, la prouver. Sinon, fournir un contre-exemple.

Exercice 3 – Monsieur et Madame Mongraf (exercice d'entraînement)

Q 3.1 Monsieur et Madame Mongraf ont organisé une soirée avec quatre couples d'amis. Certaines personnes, issues de couples différents, se sont serrées la main. À la fin du repas, Madame Mongraf demande à chaque convive combien de mains il a serrées. Elle reçoit neuf réponses différentes. Combien de mains Monsieur Mongraf a-t-il serrées ?

Exercice 4 – Deux prisonniers (exercice d'entraînement)

Deux prisonniers sont soumis à une épreuve. Dans une pièce se trouve une grande table et sur la table sont alignées 50 boîtes fermées contenant des cartes numérotées de 1 à 50. Il y a une carte pour chaque numéro de 1 à 50 et chaque boîte contient une carte. La distribution des cartes dans les boîtes a été faite aléatoirement et elle est inconnue des prisonniers. Les deux prisonniers peuvent convenir d'une stratégie avant, mais ne peuvent plus se parler le moment venu de l'épreuve. Le premier prisonnier rentre dans la pièce, regarde le contenu de toutes les boîtes et, s'il le désire, il inverse le contenu de deux boîtes (mais seulement deux). Le deuxième prisonnier entre alors dans la pièce (le premier est sorti), les gardiens de la prison lui assignent un numéro (aléatoirement) entre 1 et 50, et il doit trouver ce numéro en ouvrant au maximum 25 boîtes. S'il trouve le numéro qui lui a été donné, les deux prisonniers sont libérés, sinon, ils sont exécutés.

Q 4.1 Le problème semble défier l'entendement, car on a du mal à comprendre la stratégie que pourrait adopter le premier prisonnier sans avoir connaissance du numéro qui va être attribué au second. Pourtant, s'ils sont malins, les deux prisonniers sont certains d'être libérés. Comment font-ils ?

Exercice 5 – L'énigme du duc de Densmore (exercice avec corrigé vidéo)

Le duc de Densmore connut une fin tragique puisqu'il périt dans l'explosion qui détruisit son château. Son testament fut détruit ; or celui-ci avait tout pour déplaire à l'une de ses sept ex-femmes.

Peu avant le crime, elles étaient toutes venues au château. Chacune affirme s'être rendue une seule fois au château. Elles peuvent donc toutes être coupables, mais le dispositif ayant provoqué l'explosion avait été dissimulé dans une armure dans la chambre du duc, et sa pose avait nécessité plus d'une visite. Donc la coupable a menti : elle est venue plusieurs fois. On dispose des déclarations suivantes :

1. Ann dit avoir rencontré Betty, Charlotte, Félicia et Georgia ;
2. Betty dit avoir rencontré Ann, Charlotte, Edith, Félicia et Helen ;
3. Charlotte dit avoir rencontré Ann, Betty et Edith ;
4. Edith dit avoir rencontré Betty, Charlotte et Félicia ;
5. Félicia dit avoir rencontré Ann, Betty, Edith et Helen ;
6. Georgia dit avoir rencontré Ann et Helen ;
7. Helen dit avoir rencontré Betty, Félicia et Georgia.

Q 5.1 Tracer le graphe des rencontres, dont les sommets représentent les sept femmes et où l'on trace une arête entre deux sommets lorsque deux femmes se sont rencontrées.

Q 5.2 Montrer que si personne ne ment alors il ne peut pas exister de cycle de longueur 4 sans corde (arête reliant deux sommets non consécutifs du cycle) dans le graphe de rencontres.

Q 5.3 En déduire qui est la coupable (en considérant que les six innocentes n'ont pas menti !).

Exercice 6 – Représentation des graphes et primitives élémentaires (exercice de base)

Il existe différentes façons de représenter les graphes, dont les deux représentations suivantes :

— la représentation par *matrice d'adjacence* consiste en une matrice carrée M d'ordre n telle que $M_{ij} = 1$ si $\{i, j\}$ est une arête (respectivement si (i, j) est un arc) de G, $M_{ij} = 0$ sinon ;

type *graphe*	composé des champs suivants :
	n : *entier*
	M : *matrice*$[1..n, 1..n]$ d'*entiers* 0 ou 1

— la représentation par *liste d'adjacence* consiste en un tableau Adj de n listes. Pour tout sommet i de S, la liste $Adj[i]$ contient la liste des sommets j de S tels que $\{i, j\}$ est une arête (respectivement (i, j) est un arc) de G.

type *acellule*	qui est l'adresse du type *cellule*
type *cellule*	constitué des champs suivants :
	sommet : *entier*
	suivant : *acellule*
type *graphe*	constitué des champs suivants :
	n : *entier*
	Adj : *tableau*$[1..n]$ de *acellule*

Q 6.1 Quelle est la taille en mémoire requise pour représenter un graphe de n sommets et m arêtes (respectivement arcs) selon les deux représentations ?

Q 6.2 Si G est non orienté, déterminer pour chacune des représentations, la complexité des primitives suivantes :

1. fonction *Adjacent*$(i, j$: *sommet*$)$: *booléen*, qui rend "vrai" si i et j sont adjacents ;
2. fonction *Degré*$(i$: *sommet*$)$: *entier*, qui calcule le degré du sommet i.

Q 6.3 Si G est orienté, déterminer pour chacune des représentations, la complexité des primitives suivantes :

1. fonction *Successeur*$(i, j$: *sommet*$)$: *booléen*, qui rend "vrai" si j est un successeur de i ;
2. fonction *Prédécesseur*$(i, j$: *sommet*$)$: *booléen*, qui rend "vrai" si j est un prédécesseur de i ;
3. fonction *Degré_Extérieur*$(i$: *sommet*$)$: *entier*, qui calcule le degré extérieur du sommet i ;
4. fonction *Degré_Intérieur*$(i$: *sommet*$)$: *entier*, qui calcule le degré intérieur du sommet i.

Q 6.4 Évaluer, selon la représentation choisie, la complexité d'un algorithme qui calcule le nombre de sommets de degré impair d'un graphe non orienté.

Exercice 7 – Décompte des triangles dans un graphe (exercice d'entraînement)

Considérons un graphe qui représente un réseau social : les sommets sont des personnes et les arêtes entre deux sommets représentent les liens d'amitié (ou de connaissance) entre les personnes correspondant à ces sommets. Suivant l'adage "les amis des mes amis sont mes amis", il est probable que si la personne a connaît deux personnes b et c alors b et c se connaissent aussi. Ce lien est en tout cas plus probable que le fait que deux personnes au hasard dans le graphe se connaissent. Les graphes des réseaux sociaux comportent ainsi un grand nombre de triangles, un triangle étant un ensemble de

3 personnes se connaissant mutuellement. Le but de cet exercice est de concevoir un algorithme qui compte le nombre de triangles dans un graphe, et ce de manière efficace.

Plus formellement : étant donné un graphe $G = (S, A)$ non-orienté, on dira que le triplet de sommets $\{i, j, k\}$ forme un triangle de G si et seulement si les arêtes $\{i, j\}$, $\{j, k\}$ et $\{k, i\}$ appartiennent à A. On notera par la suite $n = |S|$, $m = |A|$ et d_{max} le degré maximum d'un sommet de G.

Q 7.1 Quelle est la complexité temporelle d'un algorithme vérifiant que le triplet $\{i, j, k\}$ est un triangle de G

a) si G est représenté par une matrice d'adjacence ?

b) si G est représenté par un tableau de listes d'adjacences ?

Q 7.2 On suppose que le graphe G est représenté par une matrice d'adjacence M. Déduisez de la question précédente un algorithme en $\Theta(n^3)$ qui compte le nombre de triangles de G.

Dans de nombreux graphes représentant des réseaux sociaux, on observe en pratique que le degré maximum d_{max} est très petit devant n. Par la suite, nous supposerons que $d_{max} \leq \sqrt{n}$.

Q 7.3 Dans ce cas, quelle est la complexité de l'algorithme proposé à la question précédente ?

Afin d'améliorer cette complexité, nous allons maintenant considérer que le graphe G est représenté par un tableau de listes d'adjacences. On considère l'algorithme suivant :

```
tr ← 0 ; it ← 1
Tab ← tableau de n entiers, initialisé à 0
pour toute arête {i, j} de A faire
    pour tout sommet x voisin de i faire
        Tab[x] ← it
    pour tout sommet y voisin de j faire
        si Tab[y] = it alors tr ← tr + 1
    it ← it + 1
retourner tr/3
```

Algorithme 2 : Comptage_par_arête(G)

Q 7.4 Prouver que cet algorithme est valide (c'est-à-dire qu'il retourne bien le nombre de triangles dans le graphe). Vous pourrez pour cela indiquer de quelle valeur est incrémentée la variable tr lors de la k^{ieme} itération de la la première boucle "pour" (ligne 4).

Q 7.5 On se propose dans cette question d'analyser la complexité de l'algorithme.

a) Expliquez en une phrase comment parcourir l'ensemble des arêtes du graphe en $O(n + m)$ (et justifiez cette complexité).

b) En déduire la complexité de cet algorithme en fonction de n, m et d_{max}.

c) Si l'on suppose que $d_{max} \leq \sqrt{n}$, quelle est la complexité de l'algorithme en fonction de n ?

Exercice 8 – Tri topologique (exercice d'approfondissement)

On rappelle que dans une liste topologique L, pour tout arc (x, y) du graphe, x est placé avant y dans la liste. Un graphe (orienté) a une liste topologique si et seulement s'il est sans circuit.

Q 8.1 Proposer un algorithme de construction d'une liste topologique basé sur la propriété suivante : *tout graphe sans circuit possède un sommet sans prédécesseur.* Calculer sa complexité.

Q 8.2 Appliquer l'algorithme proposé sur le graphe représenté ci-dessous :

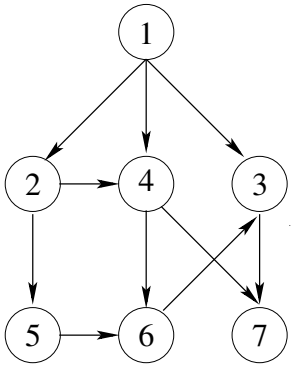


FIG. 5– Un graphe sans circuit.

Parcours génériques et applications des parcours en profondeur

Exercice 1 – Parcours générique (exercice de base)

On considère le graphe $G_1 = (S, A)$ représenté par la figure 6.

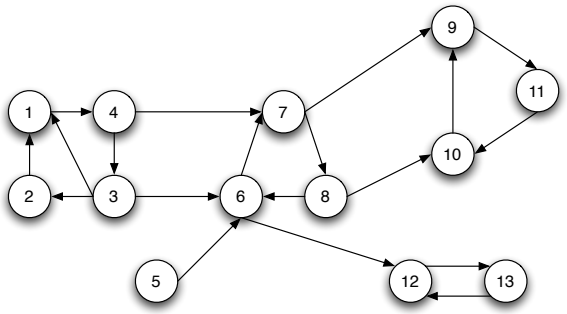


FIG. 6– Graphe G_1

Q 1.1 Déterminer les composantes fortement connexes de G_1 et le graphe réduit $\hat{G}_1$ de G_1.

Q 1.2 Montrer que la liste $L = (6, 7, 9, 11, 10, 8, 12, 13, 5, 4, 3, 2, 1)$ est un parcours de G_1. Quels sont les points de régénération de L ? Donner une forêt couvrante associée à L.

Q 1.3 Expliquer pourquoi tout parcours de G_1 possède au moins 2 points de régénération.
Donner un parcours de G_1 possédant exactement 2 points de régénération.
Expliquer pourquoi tout parcours de G_1 possède au plus 5 points de régénération.
Donner un parcours de G_1 possédant exactement 5 points de régénération.

Q 1.4 Dans le cas général, donner le nombre de points de régénération minimum et maximum du parcours d'un graphe orienté.

Exercice 2 – Reconnaissance de graphe biparti (exercice avec corrigé vidéo)

Soit $G = (S, A)$ un graphe non orienté *connexe*. L'objet de cet exercice est de concevoir un algorithme qui détermine si G est *biparti*. Un graphe est biparti si l'ensemble S peut être partitionné en deux sous-ensembles S_1 et S_2 (avec $S_1 \cup S_2 = S$ et $S_1 \cap S_2 = \emptyset$) tel qu'il n'existe pas d'arêtes entre sommets d'un même sous-ensemble (par exemple, si $(x, y) \in S_1^2$ alors il n'existe pas d'arête entre x et y).

Soit $L = (s_1, \ldots, s_n)$ un parcours de G à partir d'un sommet quelconque s_1 de S et $F(L)$ la forêt couvrante associée à L. On colore (en rouge ou bleu) tous les sommets de S selon la règle de coloration suivante :

- le sommet s_1 est bleu ;
- pour $i = 2, \ldots, n$, le sommet s_i est bleu (resp. rouge) si son père dans $F(L)$ est rouge (resp. bleu).

Le but de cet exercice est de montrer que G est biparti si et seulement si il n'existe pas d'arête de A entre deux sommets de même couleur.

Q 2.1 Faire un parcours (quelconque) du graphe de la figure 9 à partir du sommet 1, en indiquant la liste L obtenue et une forêt couvrante associée $F(L)$. En utilisant la règle de coloration présentée plus haut, préciser si le graphe est biparti.

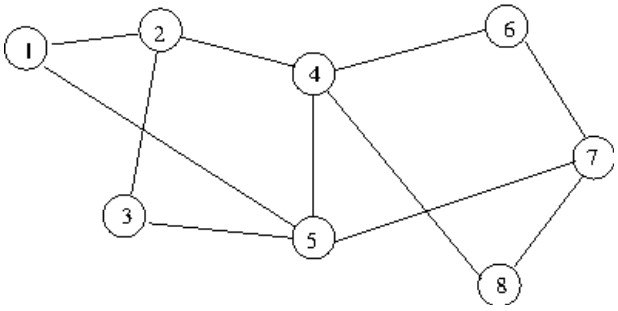


FIGURE 9 – Graphe non orienté.

On considère dorénavant un graphe $G = (S, A)$ non-orienté connexe quelconque. L'objet des questions 2 à 4 est de prouver que G est biparti si et seulement si il n'existe pas d'arête de A entre deux sommets de même couleur.

Q 2.2 Montrer que s'il n'existe pas d'arête entre deux sommets de même couleur dans $A \setminus F(L)$, alors le graphe G est biparti.

Q 2.3 Soit G un graphe biparti. Montrer que :

a) s'il existe une chaîne de longueur paire (en nombre d'arêtes) dans G entre deux sommets x et y alors ces deux sommets x et y appartiennent nécessairement au même sous-ensemble (S_1 ou S_2) dans la partition des sommets ;

b) s'il existe une chaîne de longueur impaire dans G entre deux sommets x et y alors l'un appartient à S_1 et l'autre à S_2.

Q 2.4 Soit L un parcours de G. Après coloration, on considère une arête $\{x, y\}$ entre deux sommets de même couleur. En considérant la chaîne qui relie x et y dans $F(L)$, montrer que le graphe n'est pas biparti.

Q 2.5 Compléter la procédure ci-dessous pour qu'elle permette de détecter si un graphe est biparti en $O(n + m)$. La variable globale *est_biparti* est initialisée à vrai. La valeur de cette variable à l'issue de la procédure doit être vrai si le graphe est biparti, et faux sinon. Le tableau C indexé sur les sommets indique la couleur des sommets. Il est initialisé de la façon suivante : pour tout x, $C[x]$ = incolore. Etant donné un sommet s_1 de départ, le premier appel de la procédure est $Parcours\_Rec(G, s_1, bleu)$.

Algorithme : Parcours_Rec(G : Graphe, s : Sommet, c : Couleur)
variable locale x : Sommet;
$C[s] \leftarrow c$;
pour tous les *voisins x de s* **faire**
  si **alors**
    est_biparti $\leftarrow$ faux;
  si $C[x] =$ *incolore* **alors**

Exercice 3 – Piratage informatique (exercice d'entraînement)

Ca y est ! Vous avez mis au point un nouveau virus informatique. Votre but est d'infecter l'ensemble des ordinateurs d'un réseau. Un exemple de réseau d'ordinateurs est représenté sur la figure 10 : les sommets sont les ordinateurs, et un arc (i, j) signifie que si l'ordinateur i est infecté, alors il va transmettre le virus à l'ordinateur j qui sera donc aussi infecté. Si par exemple vous introduisez le virus dans l'ordinateur 2, 2 est infecté, donc 1, 5 et 3 le seront aussi, donc 6 et 7 aussi, donc 8 aussi, et ainsi de suite ...

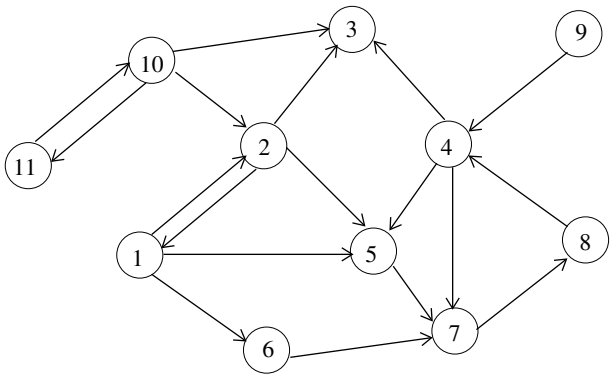


FIGURE 10 – graphe G'

Votre stratégie consiste à pirater certains ordinateurs du réseau pour y introduire le virus. Ensuite, vous laissez le virus se propager de lui-même. Vous devez choisir l'ensemble des ordinateurs piratés de manière à ce que tous les ordinateurs du réseau soient infectés (à la fin du processus de propagation). Sachant qu'il n'est pas évident de pirater un ordinateur, vous souhaitez parvenir à votre but en piratant un nombre minimum d'ordinateurs. Le but de cet exercice est de résoudre ce problème.

Nous notons $G = (S, A)$ le graphe orienté représentant le réseau d'ordinateurs. Nous dirons indifféremment qu'un sommet est infecté ou qu'un ordinateur est infecté.

Q 3.1 Soit $R \subset S$ le sous-ensemble de sommets (d'ordinateurs) que vous allez pirater. A quelle condition un sommet j du graphe sera-t-il infecté par le virus ? Sur l'exemple du graphe G' (figure 10), si l'on pirate $R = \{1, 7, 9\}$, est-ce que tous les sommets seront infectés ? De manière générale, que doit vérifier R (en termes de chemins dans le graphe) pour que la contrainte selon laquelle tous les sommets doivent être infectés soit vérifiée ?

Nous cherchons donc maintenant à trouver un ensemble R de taille minimale qui satisfait cette contrainte.

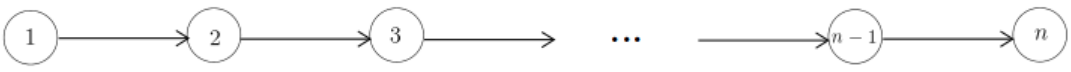


FIGURE 11 – graphe G''

Q 3.2 Votre complice vous propose la stratégie suivante : faire un parcours de G, et pirater l'ensemble des racines des arborescences composant la forêt couvrante associée au parcours.

1. Cette solution permet-elle à tous les ordinateurs d'être infectés à la fin du processus de propagation ? Justifiez votre réponse.
2. Considérons le graphe G'' qui est simplement un chemin $(s_1, s_2, \ldots, s_{n-1}, s_n)$ (voir figure 11). Quel est l'ensemble R^* optimal ? Dans le pire des cas, quel ensemble R la stratégie proposée par votre complice va-t-elle donner ? Est-ce une bonne stratégie ?

Q 3.3 On considère dans cette question le graphe G' *de la figure 10* (et non le graphe G'' de la figure 11).

1. Déterminez les composantes fortement connexes de G'.
2. Dessinez le graphe réduit G'_r de G'.
3. Déterminez un ensemble R' de G' qui vous semble optimal. Vous expliquerez brièvement votre choix (on ne demande pas de prouver ici l'optimalité de R').

Q 3.4 On considère dans cette question un graphe G orienté quelconque. Expliquez comment déterminer un ensemble R optimal (à partir du graphe G et de son graphe réduit G_r). Prouver l'optimalité de l'ensemble R construit (ainsi que le fait qu'à la fin du processus de propagation tous les ordinateurs soient infectés).

Exercice 4 – Ensemble stable maximum dans une forêt (exercice d'entraînement)

Soit un graphe non-orienté $G = (V, E)$ avec $V = \{1, \ldots, n\}$. Un sous-ensemble de sommets est dit *stable* s'il n'existe pas d'arête entre eux. Le problème du *stable maximum* vise à déterminer un sous-ensemble stable $S \subseteq \{1, \ldots, n\}$ comportant un nombre maximum de sommets.

Trouver un stable maximum dans un graphe est un problème difficile. Par contre, si le graphe est une *forêt non orientée* (c'est-à-dire un ensemble d'arbres), c'est un cas particulier plus facile.

Nous nous intéressons précisément dans cet exercice au cas où G est une forêt non orientée. On considère l'algorithme $Stable(G)$ ci-dessous, qui retourne un sous-ensemble stable S sous la forme d'un tableau $S[1 \ldots n]$ tel que $S[s] = vrai$ si et seulement si le sommet s appartient à S. Pour chaque sommet s voisin d'un sommet appartenant à S, on indique que s ne peut pas être inclus dans le sous-ensemble stable en l'incluant dans un sous-ensemble I de sommets interdits, représenté en machine sous la forme d'un tableau $I[1 \ldots n]$ tel que $I[s] = vrai$ si et seulement si s est interdit.

```
variables locales
    V, S, I : Tableaux[1 ... n] de booléens ;
    s : Sommet ;
pour s variant de 1 à n faire
    V[s] ← faux ; S[s] ← faux ; I[s] ← faux ;
pour s variant de 1 à n faire
    si V[s] = faux alors
        Descente_Rec(G, s, V, S, I) ;
retourner S
```

Algorithme 3 : $Stable(G$: Graphe)

```
   variable locale
         x : Sommet ;
   V[s] ← vrai ;
1  pour tout voisin x de s faire
2  |  si V[x] = faux alors
3  |  |  Descente_Rec(G, x, V, S, I);
4  si I[s] = faux alors
5  |  S[s] ← vrai;
6  |  pour tout voisin x de s faire
7  |  |  I[x] ← vrai
```

Algorithme 4 : *Descente_Rec*(G : Graphe, s : Sommet, V, S, I : Tableaux$[1 \dots, n]$)

Q 4.1 A quel type de parcours de graphe correspond cet algorithme ?

◯ parcours générique ◯ parcours en largeur ◯ parcours en profondeur

Q 4.2 Appliquer l'algorithme *Stable*(G) au graphe G de la figure 12. On indiquera l'ordre de parcours des sommets dans le cadre ci-dessous (en supposant que les boucles sur les voisins d'un sommet se font selon l'ordre croissant des indices), le sous-ensemble stable S obtenu directement sur la figure en entourant les sommets de S, et la forêt couvrante associée au parcours directement sur la figure également (forêt orientée constituée d'arborescences).

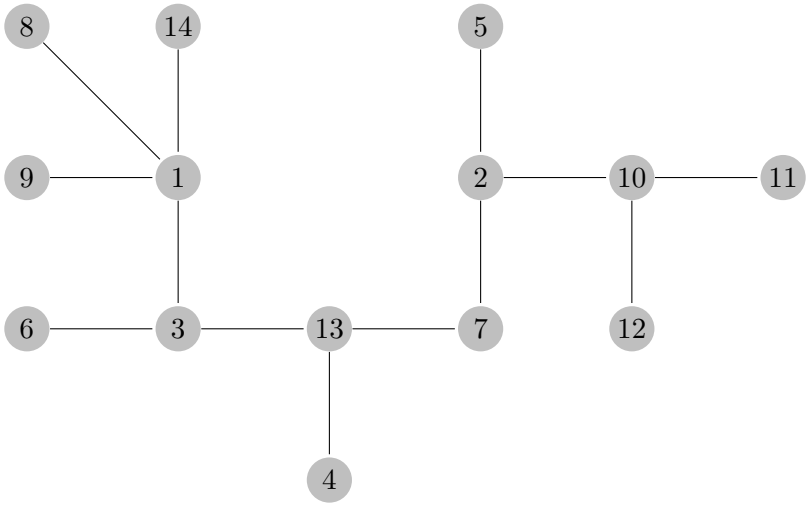


FIGURE 12 – Graphe G.

Q 4.3 Si G est représenté par des listes d'adjacence, quel nombre *total* d'itérations de la boucle des lignes 1–3 de *Descente_Rec* est réalisé par *Stable*(G) ? Donner de même un majorant du nombre total d'itérations de la boucle des lignes 6–7. En déduire la complexité de *Stable*(G). On donnera une complexité la plus précise possible, en tenant compte que G est une forêt.

L'objet des questions 4 à 8 est de prouver la validité de l'algorithme *Stable*(S), c'est-à-dire qu'il retourne bien un stable maximum.

Dans la suite, on suppose que les sommets s du graphe G sont numérotés dans l'ordre dans lequel les appels *Descente_Rec*(G, s, V, S, I) se terminent (numéro 1 pour le sommet s dont l'appel est le premier à se terminer, 2 pour le sommet dont l'appel est le deuxième à se terminer, etc.).

Q 4.4 Indiquer ci-dessous la numérotation obtenue pour l'exemple de la question 2. Reproduire à

nouveau sur la figure la forêt couvrante associée (la même qu'à la question 2).

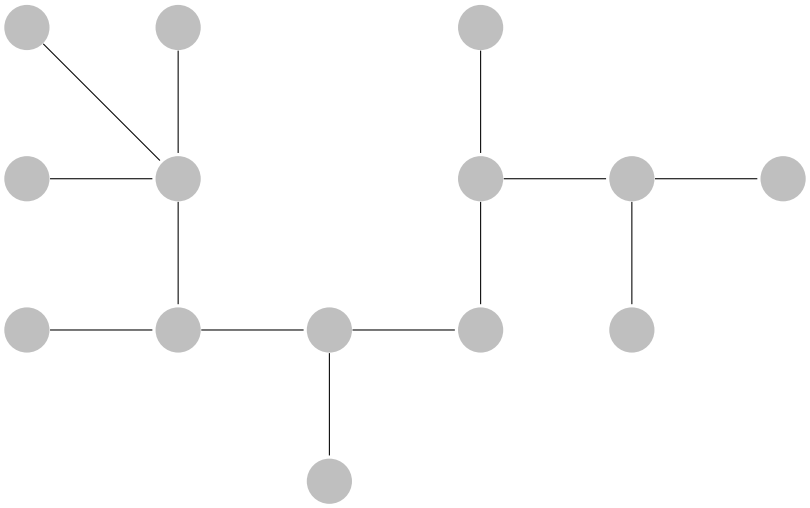


FIGURE 13 – Graphe G numéroté dans l'ordre de fin des appels récursifs.

Pour les questions qui suivent, on se place à nouveau dans le cas d'une forêt non orientée *quelconque* G (et non spécifiquement le graphe de l'exemple), *avec les sommets numérotés dans l'ordre de fin des appels récursifs de* $Stable(G)$.

Q 4.5 Montrer que si un sommet s est une feuille (sommet de degré 0 ou 1) d'une forêt non orientée G, alors il existe un stable maximum qui contient s.
Indication : On distinguera les cas $d(s) = 0$ et $d(s) = 1$. Pour $d(s) = 1$, on considérera un stable maximum qui ne contient pas s et on montrera comment obtenir un stable max qui contient s.

En s'appuyant sur cette propriété, on voit donc qu'il possible d'obtenir un stable maximum S dans une forêt non orientée G en commençant par insérer dans S une feuille, puis en supprimant de G cette feuille et son voisin. On répète ensuite l'opération sur le sous-graphe ainsi obtenu (qui est toujours une forêt). On répète l'opération jusqu'à ce que le sous-graphe obtenu soit vide.

Soit S_i (resp. I_i) l'état de l'ensemble S (resp. I) avant le test ligne 4 de $Descente\_Rec(G, i, V, S, I)$, autrement dit quand on examine le sommet i pour l'insérer dans S s'il n'est pas interdit. Les questions qui suivent visent à montrer que tout sommet i inséré dans S par $Stable(G)$ correspond à une feuille du sous-graphe $G[V \setminus (S_i \cup I_i)]$, ce qui assure la validité de $Stable(G)$.

Q 4.6 Soit (i, j) un arc de la forêt couvrante associée à un parcours en profondeur d'une forêt G, où i et j sont les numéros de fin des appels récursifs réalisés par le parcours de G (tels que ceux obtenus dans la question précédente). Quelle relation d'ordre existe-t-il entre i et j ?

Soit G la forêt non orientée en entrée de $Stable(G)$ et F sa forêt couvrante (orientée) associée. On définit $G_i = G[\{i, \ldots, n\} \setminus I_i]$ et $F_i = F[\{i, \ldots, n\} \setminus I_i]$.

Q 4.7 On se convaincra facilement que $S_i \subseteq \{1, \ldots, i-1\} \subseteq S_i \cup I_i$ (admis). En déduire que $G_i = G[V \setminus (S_i \cup I_i)]$ et $F_i = F[V \setminus (S_i \cup I_i)]$.

On note $d_G(s)$ le degré d'un sommet s dans un graphe G (orienté ou non), $d_G^-(s)$ (resp. $d_G^+(s)$) le demi-degré intérieur (resp extérieur) d'un sommet s dans un graphe orienté G.

Q 4.8 On cherche ici à montrer que si $i \notin I_i$, alors i est une feuille de $G[V \setminus (S_i \cup I_i)]$.

a) Que vaut $d_{F_i}^+(i)$? (justifier la réponse)

b) En déduire les deux valeurs possibles de $d_{F_i}(i)$ (en justifiant).

c) Quelle est la relation entre $d_{G_i}(s)$ et $d_{F_i}(s)$ (pour tout sommet s) ?

d) Conclure.

Exercice 5 – Détection de circuits dans un graphe (exercice d'entraînement)

Partie 1 : Application

La mise en œuvre d'une série de programmes $P_1, \cdots, P_n$ sur un réseau d'ordinateurs doit être planifiée. On donne les relations "P_j utilise les données calculées par le programme P_i" par les couples (P_i, P_j) suivants : (P_1, P_6), (P_2, P_1), (P_3, P_8), (P_4, P_1), (P_4, P_2), (P_5, P_{10}), (P_6, P_2), (P_6, P_{12}), (P_6, P_{13}), (P_8, P_{10}), (P_8, P_{12}), (P_9, P_1), (P_9, P_4), (P_{10}, P_7), (P_{11}, P_4), (P_{12}, P_3), (P_{12}, P_5), (P_{13}, P_3).

Q 5.1 Comment tester si cette mise en œuvre est possible ?

Partie 2 : Algorithme de parcours en profondeur

Q 5.2 Est-ce que les listes $L_1 = (1, 3, 4, 7, 2, 6, 5, 8, 9)$ et $L_2 = (8, 4, 3, 7, 9, 1, 2, 6, 5)$ sont des parcours en profondeur du graphe de la figure 14 ? Si oui, fournir une forêt couvrante associée et préciser si elle est unique.

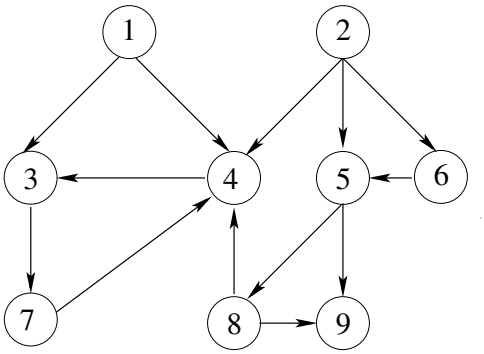


FIG. 14– Un graphe orienté.

Q 5.3 Ecrire un algorithme récursif de parcours en profondeur du graphe G.

Q 5.4 Effectuer un parcours en profondeur du graphe de la figure 14, à partir du sommet 1, en supposant que les successeurs d'un sommet sont visités dans l'ordre des indices croissants. On précisera quels sont les arcs de la forêt couvrante $\mathcal{F}(L, G)$, ainsi que les arcs avants, arrières, transverses et de liaison.

Partie 3 : Algorithme de détection de circuits

On souhaite adapter l'algorithme de parcours en profondeur pour qu'il puisse détecter la présence de circuits dans le graphe. À cette fin, on démontre tout d'abord qu'il y a équivalence entre l'existence d'un arc arrière et l'existence d'un circuit. Dans la suite, L est un parcours en profondeur de G.

Q 5.5 Montrer que s'il y a un arc arrière dans le parcours L, alors G possède un circuit (condition nécessaire).

Q 5.6 Rappeler le théorème du chemin blanc vu en cours.

Q 5.7 Montrer que tout circuit de G possède un arc arrière (condition suffisante).

Q 5.8 Comment peut-on adapter l'algorithme de parcours en profondeur pour qu'il détecte la présence de circuits dans le graphe ?

Exercice 6 – Résolution du problème 2-SAT (exercice d'entraînement)

Un *littéral* est une variable booléenne, ou la négation d'une variable booléenne (par exemple x ou $\overline{x}$). Une *clause* est composée de plusieurs littéraux liés par des $\vee$ (par exemple $x_1 \vee \overline{x_2} \vee x_3$), le symbole $\vee$ représentant l'opérateur logique "ou". Une formule booléenne est en forme normale conjonctive (FNC) si elle est constituée de plusieurs clauses liées par des $\wedge$ (par exemple $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3})$), le symbole $\wedge$ représentant l'opérateur logique "et".

Une formule booléenne 2FNC est une formule booléenne en forme normale conjonctive telle que chaque clause a 2 littéraux. Par exemple $(x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (x_2 \vee \overline{x_4})$ est une formule booléenne 2FNC.

Une formule booléenne ϕ en FNC est satisfiable s'il est possible d'affecter une valeur de vérité aux variables x_i (VRAI ou FAUX) de façon à ce que toutes les clauses de ϕ soient vérifiées (aient la valeur VRAI). Par exemple $\phi_A = (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ est safisfiable (on peut le voir en fixant x_1 à VRAI et x_2 à FAUX). Au contraire, $\phi_B = (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ n'est pas satisfiable.

Le problème 2-SAT est le suivant : étant donnée une formule booléenne 2FNC ϕ, ϕ est-elle satisfiable ?

Remarque importante : dans le problème considéré dans cet exercice, on ne s'intéresse donc pas à fournir l'affectation de valeurs de vérité correspondante si la formule ϕ est satisfiable.

Partie 1 : Transformation du problème 2-SAT en un problème de graphe.

Q 6.1 Afin résoudre le problème 2-SAT, on considère le graphe orienté $G(\phi)$ défini de la façon suivante :

- G a $2n$ sommets : un sommet pour chaque variable (x_i) et un sommet pour la négation de chaque variable ($\overline{x_i}$).
- Pour chaque clause $(a \vee b)$ de ϕ, où a et b sont des littéraux, on crée l'arc $(\overline{a}, b)$ et l'arc $(\overline{b}, a)$. Un arc (x, y) signifiera que $x \Rightarrow y$ (si x est vrai dans ϕ alors y doit être vrai pour que ϕ soit satisfiable). Ainsi, les arcs $(\overline{a}, b)$ et $(\overline{b}, a)$ créés signifient que si a est faux alors b doit être vrai pour que ϕ soit satisfiable, et si b est faux alors a doit être vrai.

Soit $\phi_1 = (\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee x_3)$. Représenter le graphe $G(\phi_1)$.

Q 6.2 Montrer qu'une formule ϕ n'est pas satisfiable si il existe une variable x_i telle que :

1. il existe un chemin entre x_i et $\overline{x_i}$ dans $G(\phi)$, et
2. il existe un chemin entre $\overline{x_i}$ et x_i dans $G(\phi)$.

En déduire que la formule ϕ_1 n'est pas satisfiable.

Inversement, on peut montrer que s'il n'existe pas de variable x_i vérifiant à la fois les conditions 1 et 2 alors la formule ϕ est satisfiable. Autrement dit, une formule ϕ est satisfiable *si et seulement si* il n'existe pas de variable x_i vérifiant à la fois les conditions 1 et 2.

Q 6.3 En déduire un algorithme en $O(n(n+m))$ qui résout le problème 2-SAT. On ne demande pas le pseudo-code mais une description informelle de l'algorithme.

Partie 2 : Algorithme de Kosaraju-Sharir.

On cherche maintenant à avoir un algorithme de meilleure complexité. Remarquons que la condition nécessaire et suffisante de satisfiabilité d'une formule ϕ peut se reformuler en : une formule ϕ est satisfiable si et seulement si il n'existe pas de variable x_i telle que x_i et $\overline{x_i}$ appartiennent à une même composante fortement connexe de $G(\phi)$. Nous allons donc étudier l'algorithme de Kosaraju-Sharir, qui détermine les composantes fortement connexes d'un graphe $G = (S, A)$, dont on suppose que les sommets sont numérotés de 1 à n. On note $G^I = (S, A^I)$ le graphe *inverse* de G, c'est-à-dire le graphe G dont on a inversé l'orientation de tous les arcs (ainsi $(a, b) \in A \Leftrightarrow (b, a) \in A^I$). L'algorithme de Kosaraju-Sharir peut se formuler comme suit :

1. Déterminer G^I
2. Appeler $DéterminerPost(G^I)$
3. Appeler $DéterminerCFC(G)$

où les procédures $DéterminerPost(G)$ et $DéterminerCFC(G)$ sont définies comme indiqué ci-dessous, avec i et *comp* des variables globales de type entier, V une variable globale de type tableau (indicé de 1 à n) de booléens, et $Post$ et CFC des variables globales de type tableau (indicé de 1 à n) d'entiers. A l'issue de l'algorithme, chaque cellule $CFC[s]$ comporte un entier identifiant la composante fortement connexe à laquelle appartient le sommet s.

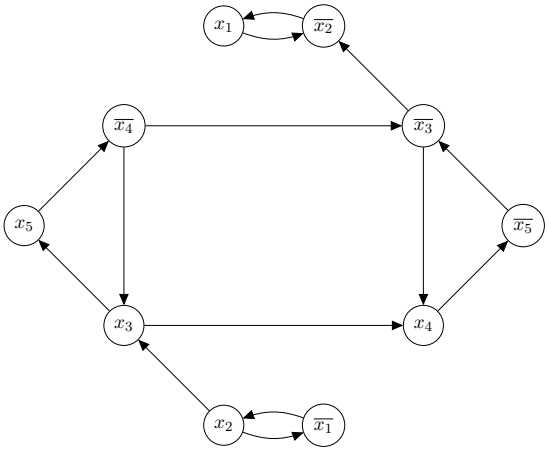
```
Procédure DéterminerPost(G : Graphe)
    pour s croissant de 1 à n faire
        V[s] ← faux ;
    i ← 1 ;
    pour s croissant de 1 à n faire
        si V[s] = faux alors
            DescentePost(G, s)
```

```
Procédure DescentePost(G : Graphe, s : Sommet)
    V[s] ← vrai ;
    pour chaque successeur x de s faire
        si V[x] = faux alors
            DescentePost(G, x)
    Post[i] ← s ;
    i ← i + 1 ;
```

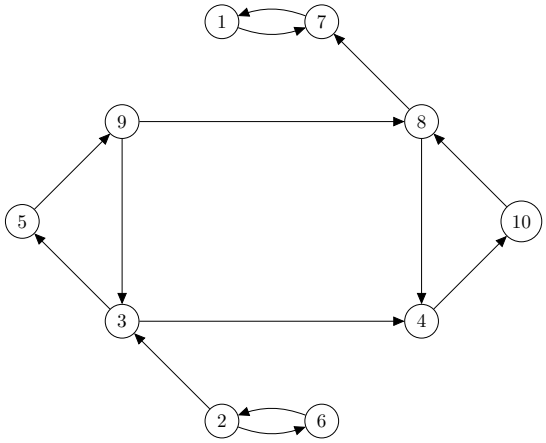
```
Procédure DéterminerCFC(G : Graphe)
    pour s croissant de 1 à n faire
        V[s] ← faux ;
    comp ← 1 ;
    pour i décroissant de n à 1 faire
        s ← Post[i] ;
        si V[s] = faux alors
            DescenteCFC(G, s) ;
            comp ← comp + 1 ;
```

```
Procédure DescenteCFC(G : Graphe, s : Sommet)
    V[s] ← vrai ;
    CFC[s] ← comp ;
    pour chaque successeur x de s faire
        si V[x] = faux alors
            DescenteCFC(G, x)
```

Afin d'illustrer le fonctionnement de cet algorithme, on cherche à résoudre le problème 2-SAT pour la formule $\phi_2 = (x_1 \vee x_2) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_3 \vee x_4) \wedge (\overline{x_3} \vee x_5) \wedge (\overline{x_4} \vee \overline{x_5}) \wedge (\overline{x_3} \vee x_4)$. Le graphe $G(\phi_2)$ correspondant est indiqué sur la gauche ci-dessous. Le graphe G de droite est obtenu en renumérotant les sommets de 1 à 10 pour pouvoir appliquer l'algorithme de Kosaraju-Sharir.



Graphe $G(\phi_2)$

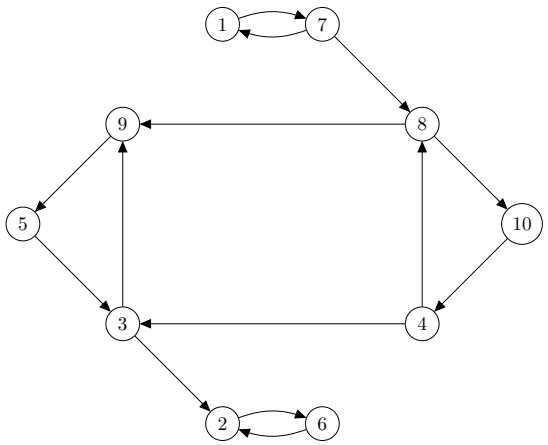


Graphe G

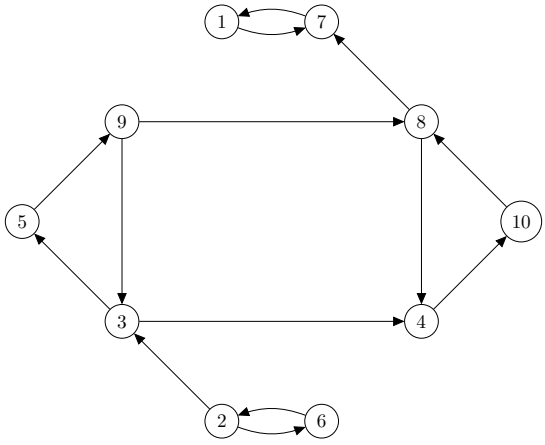
Q 6.4 L'objet de cette question est d'appliquer l'algorithme de Kosaraju-Sharir afin de déterminer les composantes fortement connexes de G.

1. Le graphe G^I est représenté à gauche ci-dessous. Compléter le tableau *Post* obtenu à l'issue de l'appel à $\mathit{DéterminerPost}(G^I)$, en supposant que les successeurs d'un sommet sont examinés par ordre croissant d'indices dans $\mathit{DescentePost}(G^I, s)$.
2. Le graphe G est rappelé à droite ci-dessous. Compléter le tableau CFC obtenu à l'issue de l'appel à $\mathit{DéterminerCFC}(G)$, en supposant que les successeurs d'un sommet sont examinés par ordre croissant d'indices dans $\mathit{DescenteCFC}(G, s)$.

1. Graphe G^I



2. Graphe G



i	1	2	3	4	5	6	7	8	9	10
***Post*[*i*]**										

s	1	2	3	4	5	6	7	8	9	10
***CFC*[*s*]**										

3. En déduire les composantes fortement connexes de G.
4. La formule ϕ_2 est-elle satisfiable ? Justifier la réponse.

L'objet des questions 5 à 7 est d'analyser la complexité de l'algorithme de Kosaraju-Sharir.

Q 6.5 Donner une description informelle de l'algorithme permettant de générer la représentation sous forme d'un tableau de listes de successeurs de G^I à partir de G, si on suppose que G est lui-même représenté sous forme d'un tableau de liste de successeurs. Indiquer la complexité de cet algorithme.

Q 6.6 A quel type de parcours correspondent $DéterminerPost$ et $DéterminerCFC$? En déduire la complexité de $DéterminerPost$ et $DéterminerCFC$.

Q 6.7 Déduire des questions 5 et 6 la complexité de l'algorithme de Kosaraju-Sharir.