Université d'Aix-Marseille - Master Informatique 1^{ere} année

UE Complexité - Correction Examen partiel - 2022-2023

Préambule : Pour les questions où vous devez fournir des algorithmes, ceux-ci devront être écrits dans un langage algorithmique avec des instructions de genre "`si alors sinon`", "`tantque`", "`pour`" etc. mais il faudra les définir avec un minimum d'ambiguïté de sorte à ce que le correcteur puisse s'assurer de leur validité. Ainsi, n'utilisez pas les particularités et les spécificités que peuvent avoir certains langages de programmation (par exemple, les boucles "`pour`" sont exprimées de façons différentes en C et en Python ; pour ce type de boucle, il faut préciser exactement les conditions initiales et celles d'arrêt, car sans cela, il est parfois impossible de déterminer la validité d'un algorithme). Quand vous utilisez des boucles, celles-ci doivent impérativement être écrites sur une seule page. Toute structure de données introduite dans vos algorithmes doivent auparavant être explicitées. Il faudra aussi définir avec un maximum de précision vos algorithmes de sorte à ce que l'évaluation de leur complexité soit la plus précise possible (ces évaluations devront toujours être justifiées). Concernant celles-ci, sauf indication contraire, nous utiliserons le modèle de base pour lequel toutes les actions élémentaires, commme les opérations du type tests, affectations ou opérations arithmétiques, sont exécutables en temps constant.

Commentaires généraux : Notons que nous n'utilisons dans ce corrigé que des boucles `tant_que` plutot que `pour` pour éviter toute ambiguïté dans l'interprétation des bornes des énumérations. Vous observerez que des explications sont données à chaque fois qu'un algorithme impose une solution non triviale, comme il faut toujours veiller à le faire dans une copie d'examen. Enfin, ce corrigé atteint un niveau de détail qui n'était pas exigé dans les copies pour obtenir une note maximum.

Exercice 1 (2 points). On considère ici des tableaux de n caractères, qui ne peuvent contenir que les caractères "a", "b" ou "B" (blanc), sachant que les caractères "a" ou "b" ne peuvent apparaître que dans le début du tableau, et dès que le caractère "B" apparaît, toutes les cases suivantes contiennent le caractère "B" jusqu'à la fin du tableau. Il vous faut donner ici un algorithme qui prend en entrée un tel tableau et qui affiche "OUI" pour le cas où le nombre de "a" et le nombre de "b" est impair, et qui affiche "NON" dans les autres cas. Donnez une évaluation de la complexité de l'algorithme.

Solution (Algorithme sur 1,25 points). Il existe bien sûr plusieurs solutions mais nous veillons dans ce corrigé à proposer l'algorithme qui soit le plus simple possible. Le résultat est mémorisé dans la variable `reponse` qui vaut `Oui` en sortie si on a bien un nombre de "a" et un nombre de "b" impair dans le tableau, et `Non` sinon. L'algorithme proposé se limite à parcourir le tableau à partir de sa première case, en comptant le nombre de "a" et le nombre de "b" qui sont présents, et s'arrête dès qu'un "B" est rencontré ou lorsque tout le tableau a été parcouru. Il suffit ensuite de vérifier si le nombre de "a" est impair ainsi que le nombre de "b". Enfin, notons que l'on suppose que l'entrée est conforme à ce qui est prévu (et que notamment, il peut donc n'y avoir aucun "B" dans le tableau).

```
entree t : tableau de caracteres indice de 1 a n
sortie reponse : {Oui,Non}
variables i, nb_a, nb_b : entiers
debut
        i <- 1
        nb_a <- 0 // nb_a compte le nombre de a dans le tableau
        nb_b <- 0 // nb_b compte le nombre de b dans le tableau
        tant_que  ( i <= n ) et ( t[i] != 'B' ) faire
                si ( t[i] = 'a' ) alors nb_a <- nb_a +1
                                  sinon nb_b <- nb_b +1
                i <- i+1
        fin_tant_que
        si (nb_a modulo 2 = 1) et (nb_b modulo 2 = 1) alors reponse <- Oui sinon reponse <- Non
fin
```

On suppose ici que le test d'entrée dans la boucle `tant_que` est réalisé de la gauche vers la droite, et donc que si la première condition `( i <= n )` est falsifiée, alors le test `(t[i] != 'B')` n'est pas exécuté, de sorte que le test `(t[n+1] != 'B')` ne soit jamais réalisé car en ce cas, il y aurait un débordement de tableau. De plus, on considère que l'opérateur `modulo` fournit le reste de la division entière, et donc `(nb_a modulo 2 = 1)` est vrai si `nb_a` est impair (idem pour `(nb_b modulo 2 = 1)`). La question de la validité de l'algorithme ne se pose pas tant celui-ci est trivial car il relève d'une niveau de première année de licence.

(Complexité sur 0,75 points). On dénombre 3 affectations pour les initialisations. Concernant la boucle, le pire des cas se présente quand tout le tableau est visité. Dans ce cas, le test `( i <= n )` est réalisé $n+1$ fois et le test `(t[i] != 'B')` est réalisé n fois. Comme il y a exactement n passages dans la boucle, le test `(t[i] = 'a')` est réalisé n fois. On obtient alors pour cette boucle un total de $3n+1$ tests auquels on peut rajouter les $n+1$ tests de la conjonction testée en entrée de boucle, soit un total de $4n+2$ tests (notons que l'on compte ou non cette conjonction ne changera rien à la complexité de l'algorithme). Dans le corps de la boucle, on réalise exactement 2 incrémentations, soit `nb_a <- nb_a +1`, soit `nb_b <- nb_b +1`, et dans tous les cas `i <- i+1`, ce qui donne 2 affectations et 2 additions, soit globalement, $2n$ affectations et $2n$ additions. En sortie, le test final `(nb_a modulo 2 = 1) et (nb_b modulo 2 = 1)` exécute 2 opérations (cf. `modulo`) et 2 tests d'égalité, auquels on peut rajouter le test de la conjonction `et`, et on aura ensuite une affectation de la variable `reponse`. On obtient au final un nombre total de $2n+4$ affectations, de $2n$ additions, 2 opérations `modulo`, et $4n+5$ tests. Globalement, on arrive donc à $8n+11$ opérations fondamentales exécutables en temps constant et la complexité de l'algorithme est donc $\Theta(n)$.

Exercice 2 (4 points). Définissez un programme pour Machine de Turing Déterministe qui reconnaît le langage L défini sur l'alphabet $\{a,b\}$ et constitué des mots qui contiennent un nombre impair de "a" et un nombre impair de "b". On rappelle que pour définir un programme pour Machine de Turing Déterministe, il faut notamment préciser ses états, son alphabet d'entrée ainsi que son alphabet de ruban, et la fonction de transition (vous pouvez fournir celle-ci par un dessin). Donnez une évaluation de la complexité de votre programme. À quelle classe de complexité appartient ce langage L ? Justifiez votre réponse.

Solution (Programme (sur 2 points). Ce programme pour machine de Turing déterministe est défini sur l'alphabet d'entrée $\Sigma = \{a,b\}$ et l'alphabet de ruban $\Gamma = \{a,b,\beta\}$ où β correspond au (symbole) blanc d'une case vide. Elle fonctionne en déplaçant systématiquement la tête de lecture/écriture de la gauche vers la droite d'une case à chaque transition, jusqu'à arriver sur la première case contenant le symbole β qui est située à droite du mot en entrée. La machine est définie avec les états suivants:

- q_{oui} et q_{non} ;
- l'état initial q_0 ;
- l'état q_{p_p} qui signifie qu'après le dernier symbole lu, le nombre de a est pair ainsi que le nombre de b (notons que cet état pourrait être remplacé par q_0) ;
- l'état q_{p_i} qui signifie qu'après le dernier symbole lu, le nombre de a est pair et le nombre de b est impair ;
- l'état q_{i_p} qui signifie qu'après le dernier symbole lu, le nombre de a est impair et le nombre de b est pair ;
- l'état q_{i_i} qui signifie qu'après le dernier symbole lu, le nombre de a est impair comme le nombre de b.

Dans l'état q_0, le nombre de a est pair ainsi que le nombre de b, et selon le symbole courant, qui peut être un blanc, on bascule dans l'état q_{i_p} si le symbole courant est un a, dans l'état q_{p_i} si le symbole courant est un b, voire dans q_{non} si le symbole courant est β pour le cas où le mot vide est en entrée. Les autres changements d'états sont évidents en fonction de leur définition. Par exemple, dans l'état q_{p_p} si le symbole courant est un a, on passe dans l'état q_{i_p} car le nombre de a lus qui était pair est maintenant impair, et si le symbole courant est un b, on passe dans l'état q_{p_i} car le nombre de b lus qui était pair est maintenant impair. Notons néanmoins que l'accès à l'état q_{oui} n'est possible qu'à partir de l'état q_{i_i} et lorsque le symbole courant est β, et que pour tous les autres états, si le symbole courant est β, la machine s'arrête sur q_{non}. On en déduit ainsi que dans tous les cas, la machine s'arrête sur le premier symbole β rencontré. La fonction de transition δ est donc :

δ	a	b	β
q_0	$(q_{i\_p},a,+1)$	$(q_{p\_i},b,+1)$	$(q_{non},?,?)$
q_{p_p}	$(q_{i\_p},a,+1)$	$(q_{p\_i},b,+1)$	$(q_{non},?,?)$
q_{p_i}	$(q_{i\_i},a,+1)$	$(q_{p\_p},b,+1)$	$(q_{non},?,?)$
q_{i_p}	$(q_{p\_p},a,+1)$	$(q_{i\_i},b,+1)$	$(q_{non},?,?)$
q_{i_i}	$(q_{p\_i},a,+1)$	$(q_{i\_p},b,+1)$	$(q_{oui},?,?)$

(Complexité sur 1 point). Dans tous les cas, donc aussi le pire des cas, un mot m en entrée doit être totalement parcouru, sachant qu'à chaque transition, la tête de lecture/écriture se déplace d'une case vers la droite, il y aura exactement n transitions, avec $n = |m|$, pour arriver au premier symbole β situé a droite du mot en entrée. Une fois arrivé sur cette case, il faut encore réaliser une transition pour la tester, mais cela conduira, dans tous les cas, à l'arrêt de la machine. On a donc, pour un mot m en entrée avec $n = |m|$, exactement $n + 1$ transitions. La complexité est donc $\Theta(|m|) = \Theta(n)$.

(Classe de complexité du langage L sur 1 point). Du fait de l'existence du programme pour machine de Turing déterministe précédent, et de sa complexié en $\Theta(n)$, donc polynomiale sur modèle de calcul déterministe, on peut affirmer que ce langage appartient à la classe de complexité **P**.

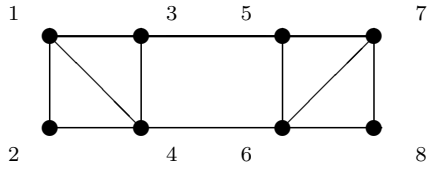
Exercice 3 (14 points). On considère ici seulement des graphes non-orientés et sans boucle (donc sans arête reliant un sommet à lui-même). On rappelle qu'une *clique* dans un graphe non-orienté $G = (S,A)$ est un sous-ensemble C de S, i.e. $C \subseteq S$ tel que le sous-graphe de G induit par ces sommets est un graphe complet, i.e. le graphe $G[C]$ est complet, c'est-à-dire que tous les sommets de $G[C]$ sont reliés deux à deux par une arête (notez que le terme *clique* désigne soit le sous-ensemble C, soit le sous-graphe de G induit par C). À partir de cette notion de clique, il est possible de définir le problème de décision suivant :

> **PARTITION EN CLIQUES**
>
> **Donnée** : Un graphe non-orienté $G = (S,A)$ et un entier $k \leq |S|$.
>
> **Question** : L'ensemble de sommets S peut-il être partitionné en k sous-ensembles $S_1,S_2,\dots S_k$ tels que $\forall i,1 \leq i \leq k$ $G[S_i]$ est un graphe complet, i.e. S_i est une clique ?

On rappelle que si l'ensemble S est partitionné en k sous-ensembles $S_1,S_2,\dots S_k$, alors on a $\forall i,1 \leq i \neq j \leq k$, $S_i \cap S_j = \emptyset$ et $\cup_{1\leq i\leq k} S_i = S$ (avec bien sûr $\forall i,1 \leq i \leq k$, $S_i \neq \emptyset$).

Question 1 (1 point). Dessinez un graphe non-orienté G de 8 sommets et 12 arêtes tel que le couple $(G,3)$ (i.e. $k = 3$) est une instance positive de *PARTITION EN CLIQUES* et le couple $(G,2)$ (i.e. $k = 2$) est une instance négative de *PARTITION EN CLIQUES*. Dans chaque cas, il vous faut justifier votre réponse.

Solution (sur 1 point avec 0,5 point par instance). Comme précisé dans la question, une instance du problème de décision *PARTITION EN CLIQUES* est un couple constitué d'un graphe non-orienté G et d'un entier k. Nous proposons donc un unique graphe G dessiné ci-dessous, mais en considérant comme instances, un couple (G,3) (i.e. $k = 3$) et un couple (G,2) (i.e. $k = 2$) :



Pour la première instance avec $k = 3$, donc l'instance (G,3), on constate que l'ensemble de sommets $S = \{1,2,3,4,5,6,7,8\}$ peut être partitionné en 3 sous-ensembles $S_1 = \{1,2,4\}$, $S_2 = \{3,5\}$ et $S_3 = \{6,7,8\}$ tels que chacun constitue une clique de G. Cette instance (G,3) est donc bien une instance positive du problème de décision *PARTITION EN CLIQUES*. Pour l'instance avec $k = 2$, donc l'instance (G,2), il faut arriver à partionner S en 2 sous-ensembles. Or, pour partitionner $S = \{1,2,3,4,5,6,7,8\}$ en 2 sous-ensembles, il en faut au moins un de taille 4 au minimum car S possède 8 éléments. Et comme on constate que dans G, il n'y a aucune clique de taille 4, et *a fortiori* de taille supérieure, il n'existe donc aucune partition en 2 cliques de ce graphe et cette instance (G,2) est bien une instance négative du problème de décision *PARTITION EN CLIQUES*.

Question 2 (3 points). À partir du problème de décision *PARTITION EN CLIQUES*, définissez le problème de **recherche** associé, puis celui d'**optimisation** (ici on considère bien sûr comme critère d'optimisation le nombre de cliques), celui de **comptage** et celui d'**énumération**. Enfin, définissez le problème **complémentaire** du problème de décision *PARTITION EN CLIQUES*.

Solution (Sur 3 points, soit 0,5 point par problème sauf pour *optimisation* qui est moins simple).

PARTITION EN CLIQUES RECHERCHE
Donnée : Un graphe non-orienté $G = (S,A)$ et un entier $k \leq |S|$.
Question : Si l'ensemble de sommets S peut être partitionné en k sous-ensembles $S_1,S_2,\dots S_k$ tels que $\forall i,1 \leq i \leq k$, $G[S_i]$ est un graphe complet, i.e. S_i est une clique, alors donner une telle partirion.

PARTITION EN CLIQUES OPTIMISATION
Donnée : Un graphe non-orienté $G = (S,A)$.
Question : Donner le plus petit entier k tel que l'ensemble de sommets S peut être partitionné en k sous-ensembles $S_1,S_2,\dots S_k$ tels que $\forall i,1 \leq i \leq k$, $G[S_i]$ est un graphe complet, i.e. S_i est une clique.

Notons qu'ici, l'optimisation consiste à minimiser le nombre de parties de la partition en cliques. En effet, trouver une solution avec un maximum de parties est trivial puisqu'il suffit, pour un graphe quelconque de n sommets, de proposer une partition en n parties, une par sommet, sachant que tout sommet seul constitue une clique triviale. Par contre, minimiser le nombre de parties est bien plus difficile.

PARTITION EN CLIQUES COMPTAGE
Donnée : Un graphe non-orienté $G = (S,A)$ et un entier k.
Question : Donner le nombre de possiblités qu'il y a de partitionner l'ensemble de sommets S en k sous-ensembles $S_1,S_2,\dots S_k$ tels que $\forall i,1 \leq i \leq k$, $G[S_i]$ est un graphe complet, i.e. S_i est une clique.

PARTITION EN CLIQUES ÉNUMÉRATION
Donnée : Un graphe non-orienté $G = (S,A)$ et un entier k.
Question : Donner toutes les possiblités qu'il y a de partitionner l'ensemble de sommets S en k sous-ensembles $S_1,S_2,\dots S_k$ tels que $\forall i,1 \leq i \leq k$, $G[S_i]$ est un graphe complet, i.e. S_i est une clique.

PROBLÈME COMPLÉMENTAIRE DE PARTITION EN CLIQUES
Donnée : Un graphe non-orienté $G = (S,A)$ et un entier $k \leq |S|$.
Question : Est-il impossible de partitionner l'ensemble de sommets S en k sous-ensembles $S_1,S_2,\dots S_k$ tels que $\forall i,1 \leq i \leq k$, $G[S_i]$ est un graphe complet, i.e. S_i est une clique ?

Dans la suite de cet exercice, on supposera que pour un graphe $G = (S,A)$, une partition de $S = \{1,2,\dots n\}$ (les sommets dans S sont représentés par les n premiers entiers non nuls) en k sous-ensembles est représentée par un tableau `t` d'entiers appartenant à l'ensemble $\{1,2,\dots,k\}$, indicé par les sommets de G, tel que $\forall i,1 \leq i \leq n$, `t[i]=p` si $i \in S_p$ avec donc $p \in \{1,2,\dots,k\}$. Par exemple si $S = \{1,2,3,4,5,6,7,8\}$, une partition de S, en 3 sous-ensembles $S_1 = \{1,3,5\}$, $S_2 = \{2,7\}$ et $S_3 = \{4,6,8\}$ sera représentée par `t[1]=1, t[2]=2, t[3]=1, t[4]=3, t[5]=1, t[6]=3, t[7]=2` et `t[8]=3`.

Question 3 (finalement sur 2 points avec 1 point "bonus"). Donnez un algorithme qui prend en entrées un graphe non-orienté G de n sommets, un entier k, et un tableau `t` d'entiers indicé de 1 à n, et qui vérifie si `t` représente une partition des n sommets de G en k sous-ensembles. Donnez une évaluation de la complexité de votre algorithme.

Solution. (Algorithme sur 1,5 points) On note ici que la donnée du graphe G n'est pas essentielle car la notion de partition nécessite uniquement de disposer d'un ensemble d'éléments (ici les n sommets du graphe) et que l'algorithme sera valable pour toute représentation de partition telle qu'elle est prévue en préambule de cette question. Pour traiter cette question, il faut vérifier deux choses :

1. Que la réunion de l'ensemble des parties couvre bien l'ensemble des n sommets, c'est-à-dire, que $\forall i$, $1 \leq i \leq n$, on doit avoir `t[i]` $\in \{1,2,\dots k\}$ car ainsi, on sait que chaque sommet est dans l'une des k parties. Nous utilisons un booléen appelé `reunion` pour représenter cette propriété.
2. Qu'aucune partie n'est vide, ce qui conduit à vérifier que $\forall j$, $1 \leq j \leq k$, $\exists i$, $1 \leq i \leq n$, tel que `t[i]`$= j$ (cela veut dire que S_j contient au moins 1 sommet, précisément le sommet i). Pour réaliser cela, on utilise un tableau de booleens appelé `vide` indicé de 1 à k, tel que, pour $1 \leq j \leq k$, `vide[j]=vrai` signifie que S_j est vide, et `vide[j]=faux` signifie que S_j n'est pas vide. Le tableau `vide[]` est donc initialisé en affectant à `vrai` toutes ses cases, et une case `vide[j]` de ce tableau conservera cette valeur tant que l'on n'aura pas vérifié qu'il existe au moins un élément `i` dans la partie correspondant à l'indice du tableau, c'est-à-dire que `t[i]=j`. En d'autres termes, il suffit d'affecter `vide[j]` à `faux` dès que l'on trouve un sommet $i \in S_j$, c'est-à-dire quand on a `t[i]=j`.

Enfin, notons que par définition du tableau t, un sommet ne peut figurer au plus que dans une seule partie, et donc qu'il n'est pas nécessaire de vérifer que l'intersection entre deux parties différentes est vide.

```
Algorithme Test_de_partition
entree G : graphe
entree k : entier non nul
entree t : tableau d'entiers indice de 1 a n
sortie partition : booleen
variables i,j : entiers
variables reunion, partie_vide_existe : booleens
variable vide : tableau de booleens indice de 1 a k
debut
        // on verifie le point (1)
        reunion <- vrai
        i <- 1
        tant_que reunion et ( i <= G.n ) faire
                si ( 1 <= t[i] <= k ) alors i <- i+1 sinon reunion <- faux
        fin_tant_que

        si non reunion alors partition <- faux
        sinon // on verifie maintenant le point (2)

                // initialisation a vrai du tableau vide[]
                j <- 1
                tant_que ( j <= k ) faire
                        vide[j] <- vrai
                        j <- j+1
                fin_tant_que

                // on modifie le tableau vide[] en passant sur tous les sommets i de 1 a n :
                // t[i]=j signifie que la partie Sj n'est pas vide et vide[t[i]] est donc affecte a faux
                i <- 1
                tant_que ( i <= G.n ) faire
                        vide[t[i]] <- faux
                        i <- i+1
                fin_tant_que

                // on verifie qu'aucune partie n'est vide (on utilise pour cela le booleen "partie_vide_existe"
                partie_vide_existe <- faux
                j <- 1
                tant_que ( non partie_vide_existe ) et ( j <= k ) faire
                        si ( vide[j] ) alors partie_vide_existe <- vrai sinon j <- j+1
                fin_tant_que

                si partie_vide_existe alors partition <- faux sinon partition <- vrai
        fin_si
fin
```

Notons que pour la partie (2) de l'algorithme, il serait possible d'avoir une version qui compte le nombre de parties non vides, et s'arrête dès que l'on en a dénombré k. Mais en fait, cela ne modifierait en rien la complexité du traitement.

(Complexité sur 0,5 points). La première boucle `tant_que` qui affecte le booléen `reunion` a une complexité $\Theta(n)$ car il y a n passages réalisés en temps constant dans la boucle. Pour le cas où en sortie de cette boucle, le booléen `reunion` est `vrai`, l'affectation du booléen `partie_vide_existe` exécute 3 boucles respectivement de complexité $\Theta(k)$, $\Theta(n)$ et $\Theta(k)$, sachant que $k \leq n$, ce qui cumulé a un coût de $\Theta(n)$. Ainsi, la complexité globale de cet algorithme est donc $\Theta(n)$.

Question 4 (4 points). Donnez un algorithme qui prend en entrées un graphe non-orienté G de n sommets, un entier k, et un tableau `t` d'entiers indicé de 1 à n, et qui vérifie si `t` représente une partition des n sommets de G en k sous-ensembles dont chacun est bien une clique de G. Donnez une évaluation de la complexité de votre algorithme. Nous vous demandons d'utiliser à partir de cette question, une représentation des graphes par matrices d'adjacence même si cela engendre une (petite) dégradation de l'efficacité de vos algorithmes.

SolutionS. Nous allons donner ici 2 solutions. Dans chaque cas, il s'agit d'abord de vérifier que le tableau `t` représente bien une partition des n sommets de G en k sous-ensembles. Pour s'en assurer, il suffit d'utiliser l'algorithme proposé dans la réponse à la question précédente. Sinon, la première solution est assez naturelle car elle consiste à tester chaque partie de la partition pour vérifier si ses sommets sont tous voisins et qu'ils constituent donc bien une clique du graphe. Cette solution est cependant plus difficile à analyser en termes de complexité. La seconde solution est plus astucieuse mais aussi, plus simple à mettre en œuvre et à analyser en termes de complexité. Elle repose sur le constat suivant : la partition n'est pas une partition en clique s'il existe un couple de sommets du graphe qui figurent dans une même partie et qui ne partagent pas une arête. Et pour tester cela, il suffit de tester tous les couples de sommets, et si un couple est dans une même partie, vérifier s'ils partagent ou non une même arête. Nous présentons ces 2 solutions ci-dessous.

Solution 1 (Algorithme sur 3 points). La première chose à vérifier ici est donc que le tableau `t` représente bien une partition des n sommets de G en k sous-ensembles. Pour s'en assurer, il suffit d'utiliser l'algorithme proposé dans la réponse à la question précédente. Une fois que cela est vérifié, il faut s'assurer que chaque partie correspond à une clique du graphe G. Pour cela, nous allons définir une fonction qui prend en entrées le graphe G, le tableau t, et un entier p tel que $1 \leq p \leq k$, et qui vérifie si la partie S_p est telle que ses sommets constituent une clique de G. Cette fonction sera utilisée pour vérifier que chacune des k parties est une clique de G. Nous décrivons d'abord cette fonction :

```
Fonction Clique
entree G : graphe
entree p : entier
entree t : tableau d'entiers indice de 1 a n
sortie est_clique : booleen
variables i,j : entiers

debut
        est_clique <- vrai
        i <- 1
        tant_que est_clique et ( i < G.n ) faire
                si ( t[i] = p ) alors // on verifie que tous les sommets j de Sp, avec i < j sont voisins de i
                        j <- i+1
                        tant_que est_clique et ( j <= G.n ) faire
                                si ( t[j] = p ) et ( G.A[i][j] = 0 ) alors est_clique <- faux sinon j <- j+1
                        fin_tant_que
                fin_si
                i <- i+1
        fin_tant_que
fin
```

On remarque que du fait de la représentation matricielle du graphe, on a nécessairement `G.A[i][j] = G.A[j][i]` et donc qu'il suffit de tester l'existence d'une arête $\{i,j\}$ avec $i < j$ sans qu'il soit nécessaire de la tester à nouveau quand on traite le sommet j. Ainsi, le traitement n'est pas nécessaire pour le sommet n et la boucle sur i peut s'arrêter quand on a traité $i = n-1$. On peut donc maintenant plus facilement définir l'algorithme vérifiant si le tableau `t` représente une partition des n sommets de G en k sous-ensembles dont chacun est bien une clique de G :

```
Algorithme Partition_en_Cliques_1
entree G : graphe
entree k : entier
entree t : tableau d'entiers indice de 1 a n
sortie partition_en_cliques : booleen
variable p : entier
```

```
debut
        si non Test_de_partition(G,k,t) alors partition_en_cliques <- faux
        sinon
                partition_en_cliques <- vrai
                p <- 1
                tant_que partition_en_cliques et ( p <= k ) faire
                        // on verifie si Sp est une clique de G
                        si Clique(G,p,t) alors p <- p+1 sinon partition_en_cliques <- faux
                fin_tant_que
        fin_si
fin
```

(Complexité sur 1 point). Le test réalisé avec l'appel `Test_de_partition(G,k,t)` a une complexité $\Theta(n)$. S'il a pour résultat `vrai`, le `sinon` est exécuté. Cela peut engendrer k appels de la fonction `Clique`. Nous évaluons donc le coût d'un appel de la forme `Clique(G,p,t)`. On constate que pour un p donné (avec $1 \leq p \leq k$), il y a exactement n passages dans la boucle `tant_que` itérant sur `i`, et donc exactement n tests de la forme `( t[i] = p )`. À chaque passage, seul les éléments de S_p seront traités car le test `( t[i] = p )` est vrai uniquement pour les éléments de S_p. Le coût pour un sommet $i \in S_p$ est de l'ordre de $n - i + 1$ car il y a une itération sur j de $i + 1$ à n, ce qui donne, pour un appel de `Clique(G,p,t)`, un coût de l'ordre de $\Sigma_{i \in S_p}(n - i + 1)$. On peut alors majorer la complexité d'un de ces traitements en considérant qu'il y aura moins de n passages dans la boucle `tant_que` itérant sur `j`. Cela permet déjà de majorer le coût d'un appel de la fonction `Clique` par $O(n^2)$. En considérant que $k \leq n$, cela nous permet de majorer la complexité de `Partition_en_Cliques` par $O(n^3)$.

Une telle réponse à l'examen était suffisante pour le correcteur. Cela étant, on peut procéder à une analyse plus fine, mais celle-ci impose de regrouper tous les k appels de la fonction `Clique`. En effet, le test `( t[i] = p )` exécuté dans la fonction `Clique` est au total vrai exactement n fois, pour tous les k appels de cette fonction. Ainsi, la boucle interne `tant_que` itérant sur `j` ne sera globalement exécutée au plus que n fois, une fois pour chaque élément de chaque partie, car `t` représente une partition de l'ensemble des sommets du graphe. De façon plus précise, la globalisation de k appels de `Clique(G,p,t)` conduit à un coût global de l'ordre de $\Sigma_{1 \leq p \leq k}(n + \Sigma_{i \in S_p}(n - i + 1))$. Or, $\Sigma_{1 \leq p \leq k}(n + \Sigma_{i \in S_p}(n - i + 1)) = k \times n + \Sigma_{1 \leq p \leq k}(\Sigma_{i \in S_p}(n - i + 1))$ et comme les ensembles $S_1, S_2, \ldots S_k$ constituent une partition des n sommets de S, on peut affirmer que $\Sigma_{1 \leq p \leq k}(\Sigma_{i \in S_p}(n - i + 1)) = \Sigma_{1 \leq i \leq n}(n - i + 1)$. Sachant que $\Sigma_{1 \leq i \leq n}(n - i + 1) = \Sigma_{1 \leq i \leq n} n - \Sigma_{1 \leq i \leq n} i + \Sigma_{1 \leq i \leq n} 1 = n^2 - \frac{n(n+1)}{2} + n = n^2 - \frac{n^2}{2} - \frac{n}{2} + n = \frac{n^2}{2} + \frac{n}{2}$. On arrive ainsi à un coût global de l'ordre de $k \times n + \frac{n^2}{2} + \frac{n}{2}$. Comme $k \leq n$, la complexité de `Partition_en_Cliques` est $\Theta(n^2)$, et on notera que cette complexité est atteinte dans tous les cas.

Solution 2 (Algorithme sur 3 points). Comme pour la solution précédente, la première chose à vérifier ici est donc que le tableau `t` représente bien une partition des n sommets de G en k sous-ensembles. Pour s'en assurer, il suffit d'utiliser l'algorithme proposé dans la réponse à la question précédente. Une fois que cela est vérifié, nous allons tester tous les couples de sommets $\{i,j\}$ du graphe, et si on trouve un couple qui appartient à une même partie mais qui n'est pas relié par une arête, on vient de constater que l'on ne dispose pas en entrée d'un tableau t représentant une partition en cliques du graphe G. Notons que du fait de la représentation matricielle du graphe, on a nécessairement `G.A[i][j] = G.A[j][i]` et donc qu'il suffit de tester l'existence d'une arête $\{i,j\}$ avec $i < j$ sans qu'il soit nécessaire de la tester à nouveau quand on traite le sommet j.

```
Algorithme Partition_en_Cliques_2
entree G : graphe
entree k : entier
entree t : tableau d'entiers indice de 1 a n
sortie partition_en_cliques : booleen
variables i,j : entiers

debut
        si non Test_de_partition(G,k,t) alors partition_en_cliques <- faux
        sinon
                 partition_en_cliques <- vrai
                 i <- 1
                 tant_que partition_en_cliques et ( i < G.n ) faire
                         j <- i+1
                         tant_que partition_en_cliques et ( j <= G.n ) faire
                                 si ( t[i] = t[j] ) et ( G.A[i][j] = 0 ) alors partition_en_cliques <- faux
                                                                         sinon j <- j+1
                                 fin_si
                         fin_tant_que
                         i <- i+1
                 fin_tant_que
        fin_si
fin
```

(Complexité sur 1 point). Le test réalisé avec l'appel `Test_de_partition(G,k,t)` a une complexité $\Theta(n)$. S'il a pour résultat `vrai`, le `sinon` est exécuté. Nous ne détaillons pas l'analyse de la complexité de ces boucles imbriquées car on se limite ici à constater que le test `( t[i] = t[j] ) et ( G.A[i][j] = 0 )` est exécuté au pire $\Sigma_{1 \leq i < n}(n-i)$ fois, et on en déduit que la complexité globale de `Partition_en_Cliques_2` est donc ici aussi $\Theta(n^2)$.

Question 5 (4 points). Donnez un schéma d'algorithme (il n'est pas nécessaire de donner votre algorithme dans le détail) qui prend en entrées un graphe non-orienté $G = (S,A)$ et un entier k, et teste si S peut être partitionné en $k \leq n$ sous-ensembles $S_1, S_2, \ldots S_k$ tels que $\forall i, 1 \leq i \leq k$, $G[S_i]$ est un graphe complet, i.e. S_i est une clique? Donnez une évaluation de la complexité de votre algorithme. Avant de présenter le code de votre algorithme, vous expliquerez clairement la méthode sur laquelle il se base.

Solution (Schéma sur 3 points + 1 point pour la complexité). En fait, une solution simple consiste à tester toutes les partitions possibles en k sous-ensembles, sachant que pour chacune, pour vérifier s'il s'agit d'une partition en cliques de G, on peut utiliser l'algorithme proposé dans la réponse à la question 4 et dont la complexité est $\Theta(n^2)$. Il suffit pour cela de développer une arborescence de recherche (déterministe) qui va construire toutes les partitions possibles des sommets de G en k sous-ensembles, une à une si nécessaire (si on en trouve 1, le traitement peut cependant s'arrêter). Pour cela, pour chaque sommet s du graphe, on a exactement k possibilités, soit $s \in S_1$, soit $s \in S_2$,... soit $s \in S_k$. On arrive ainsi à une arborescence de hauteur n dont chaque nœud interne possède exactement k fils (1 pour chaque possibilité d'appartenance à une partie). Or, on sait que le nombre de nœuds d'une telle arborescence (pour $k > 1$) est exactement égal à $\frac{k^{n+1}-1}{k-1}$, et qu'il possède exactement k^n feuilles. Sachant qu'à chaque feuille, il y aura un test de coût n^2 à réaliser (cf. appel de `Partition_en_Cliques`). On obtient alors une complexité de $O(n^2 \times k^n)$, donc exponentielle. Notons que la note maximale était possible, sans qu'un code ne soit donné, sachant qu'avant tout, c'est le schéma de l'algorithme qui était demandé. Cependant, nous donnons ici le code simplifié d'une telle approche. S'agissant du développement d'une arborescence, nous proposerons naturellement un code récursif. L'appel initial sera de la forme `Resolution_Partition_en_Cliques(G,0,k,t)`. Au préalable, notons que pour le cas où on aurait $k = 1$, cette situation pourrait être traitée comme exception, sachant qu'il suffirait alors de tester le cas pour lequel le graphe G serait complet.

```
Algorithme Resolution_Partition_en_Cliques
entree G : graphe
entree s : entier
entree k : entier
entree t : tableau d'entiers indice de 1 a n
sortie solution : booleen
variable p : entier
variable sol_part_cliq : booleen
debut
        si s = G.n alors
                si Partition_en_Cliques(G,k,t) alors renvoyer vrai (et stopper l'execution)
                                                sinon renvoyer faux
        sinon
                s <- s+1
                p <- 1
                sol_part_cliq <- faux
                tant_que (non sol_part_cliq ) et ( p <= k ) faire
                        t[s] <- p
                        sol_part_cliq <- Resolution_Partition_en_Cliques(G,s,k,t)
                fin_tant_que
        fin_si
fin
```

Question 6 (1 point). Pouvez-vous affirmer que le problème de décision *PARTITION EN CLIQUES* appartient à la classe de complexité **P**? Justifiez votre réponse.

Solution. (Sur 1 point) En fait, le schéma d'algorithme proposé dans la question précédente correspond à un algorithme qui résout ce problème et dont la complexité est exponentielle. Cela étant, nous ne pouvons prouver que c'est le meilleur algorithme pour résoudre le problème de décision *PARTITION EN CLIQUES*. Aussi, nous ne pouvons affirmer avec ce qui précède si le problème de décision *PARTITION EN CLIQUES* appartient à la classe de complexité **P**, ni même le contraire.