Algorithmique III (I41) - Licence d'Informatique

Contrôle terminal (Session 1 - Mai 2024)

Le sujet précisait que les explications, commentaires et justifications des résultats étaient exigés. Leur absence pour justifier les algorithmes était sanctionnée par la perte de la moitié des points, si les algorithmes étaient justes et compréhensibles, évidemment. Cette correction est *largement* plus détaillée que ce que j'exigeais dans les réponses, mais il fallait *a minima*, que les arguments clefs soient évoqués pour obtenir des points.

Partie 1

Notations : $\mathscr{A}$ est un alphabet q-aire muni d'un ordre total $\leqslant$ (*l'ordre alphabétique*), $\mathscr{A}^*$ est l'ensemble des mots sur $\mathscr{A}$ et S_n le groupe des permutations σ de degré n codées par des listes $[\sigma(1), \ldots, \sigma(n)]$ indexées de 1 à n. On définit une relation binaire $\leftrightsquigarrow$ entre deux mots u et v de $\mathscr{A}^*$ par :

$$u \leftrightsquigarrow v \;\Leftrightarrow\; \underbrace{\exists n \in \mathbb{N}\ (|u| = |v| = n)}_{\text{cond. 1}} \wedge \underbrace{\left(\exists \sigma \in S_n\ \forall i \in [\![1, n]\!]\ v_{\sigma(i)} = u_i\right)}_{\text{cond. 2}} \quad (1)$$

et on dit que v est une *anagramme* de u en notant $v = \sigma(u)$ pour toute permutation σ qui vérifie (1). Par exemple *baliser* est une anagramme de *sablier* et $\sigma(s_5a_2b_1l_3i_4e_6r_7) = b_1a_2l_3i_4s_5e_6r_7$ pour $\sigma = [5, 2, 1, 3, 4, 6, 7]$.

Question 1. Combien d'anagrammes distinctes existe-t-il d'un mot de longueur n : (a) si toutes ses lettres sont différentes ? (b) s'il contient exactement trois lettres identiques ($n \geqslant 3$) ?

Solution. (a) Si u est un mot de longueur n, la condition (1) exprime qu'un mot v est une anagramme de u s'il est obtenu par une permutation les lettres de u. Notons $A(u) \subseteq \mathscr{A}^n$ l'ensemble des anagrammes du mot u et considérons l'application $\alpha_u : S_n \to A(u)$ définie par $\sigma \mapsto \sigma(u)$ et qui associe à une permutation σ de degré n donnée, l'anagramme $v := \sigma(u)$ du mot u. Elle est surjective car toute anagramme de u est par construction une permutation des lettres de u, mais n'est pas nécessairement injective comme le prouve le mot $u = sos$, dont l'anagramme oss est l'image des deux permutations distinctes $\sigma_1 := [2, 1, 3]$ et $\sigma_2 := [3, 1, 2]$ par α_u.

Si toutes les lettres de u sont distinctes, deux permutations différentes fournissent deux anagrammes différentes, i.e. l'application α_u est injective. Par conséquent, α_u est bijective et $A(u) \approx S_n$ ce qui permet de conclure que $|A(u)| = |S_n| = n!$

(b) Si un mot u contient des lettres identiques, l'application α_u n'est plus injective. On définit alors une relation binaire $\bowtie$ sur S_n par $\sigma \bowtie \sigma' \Leftrightarrow \sigma(u) = \sigma'(u)$ et on vérifie aisément qu'il s'agit d'une relation d'équivalence sur S_n. Autrement dit, on regroupe au sein d'une même classe d'équivalence, toutes les permutations qui fournissent la même anagramme du mot u et c'est la bijection $\overline{\alpha}_u : S_n/_{\bowtie} \to A(u)$, qui donnera le résultat en dénombrant le nombre de classes d'équivalence (notons que l'on pouvait procéder de la même manière pour répondre à la première question, chaque classe d'équivalence étant dans ce cas réduite à un unique élément).

Considérons deux permutations σ et σ' d'une même classe d'équivalence, on peut les décomposer en $\sigma = \rho.\tau$ et $\sigma' = \rho.\tau'$ où ρ désigne la permutation commune à σ et σ' des $n-3$ lettres distinctes de u et τ et τ' désignent les deux permutations des 3 lettres identiques du mot u et qui distinguent σ et σ'. Il y a 3! permutations différentes de trois lettres, chaque classe d'équivalence a donc le même cardinal $3! = 6$ et finalement $|S_n/_{\bowtie}| = |S_n|/6 = n!/6$ d'après le principe des Bergers.

Question 2. Démontrez que la relation binaire $\leftrightsquigarrow$ définie sur $\mathscr{A}^*$ est une *relation d'équivalence*.

Solution. Il faut donc prouver qu'elle est *réflexive*, *symétrique* et *transitive*. Dans les trois cas, on peut se dispenser de vérifier la première condition sur la longueur des mots dans la conjonction (1) puisque l'égalité est une relation d'équivalence.

- Réflexivité : montrons que $\forall u \in \mathscr{A}^*$, $u \leftrightsquigarrow u$. La permutation σ qui satisfait (1) est l'identité.

- Symétrie : montrons que $\forall (u, v) \in (\mathscr{A}^*)^2$ $u \leftrightsquigarrow v \Rightarrow v \leftrightsquigarrow u$. Comme $u \leftrightsquigarrow v$, on dispose d'une permutation σ satisfaisant (1), et la permutation inverse σ^{-1} est celle qui satisfait cette même condition pour $v \leftrightsquigarrow u$.

- Transitivité : montrons que $\forall (u, v, w) \in (\mathscr{A}^*)^3$ $(u \leftrightsquigarrow v) \wedge (v \leftrightsquigarrow w) \Rightarrow (u \leftrightsquigarrow w)$. La prémisse nous fournit deux permutations σ_1 et σ_2 telles que $v = \sigma_1(u)$ et $w = \sigma_2(v)$. On en déduit que $w = \sigma_2(\sigma_1(u))$ autrement

dit que $w = \sigma_2 \circ \sigma_1(u)$ et la permutation composée $\sigma := \sigma_2 \circ \sigma_1$ satisfait (1) prouvant que $u \leftrightsquigarrow w$.

Question 3. Quelles sont les classes d'équivalence du langage $\mathscr{L}$ ci-dessous pour la relation $\leftrightsquigarrow$?

$$\mathscr{L} := \{set,car,test,à,lu,tes,misa,mais,arc,est,amis\}$$

Solution. Il s'agit simplement des ensembles :

$$\{à\},\{amis,mais,misa\},\{car,arc\},\{lu\},\{set,est,tes\},\{test\}.$$

Partie II

Vous *pouvez* utiliser, *sans les écrire*, les algorithmes qui suivent. La fonction de complexité en temps est fournie pour chacun d'eux :

(1) `Adr(lettre)` : renvoie le rang de `lettre` dans l'alphabet (1ère lettre de $\mathscr{A}$ en 1). ($\Theta(1)$).

(2) `Enfiler(liste, pos)` : rajoute une position en fin de liste. ($\Theta(n)$ où $n = |\texttt{liste}|$).

(3) `Defiler(liste)` : libère la dernière cellule de la liste et renvoie sa valeur. ($\Theta(n)$ où $n = |\texttt{liste}|$).

(4) `Empiler(liste, pos)` : rajoute une position en tête de liste. ($\Theta(1)$).

(5) `Depiler(liste)` : libère la première cellule de la liste et renvoie sa valeur. ($\Theta(1)$).

Question 1. Si pour chaque lettre ℓ de l'alphabet $\mathscr{A}$, on dispose d'une liste contenant toutes les positions de ℓ dans un mot $u \in \mathscr{A}^*$ donné (la liste est vide si le mot ne contient pas cette lettre), qu'obtient-on en concaténant les $q := |\mathscr{A}|$ listes dans l'ordre alphabétique ? Justifiez et illustrez avec le mot *attribuer*.

Solution. On dispose donc de q listes puisque l'alphabet est q-aire. Pour le mot *attribuer* défini sur l'alphabet latin, en rangeant les positions des lettres dans l'ordre d'apparition, on obtient :

$a = [1]$	$g = [\,]$	$m = [\,]$	$s = [\,]$	$y = [\,]$
$b = [6]$	$h = [\,]$	$n = [\,]$	$t = [2,3]$	$z = [\,]$
$c = [\,]$	$i = [5]$	$o = [\,]$	$u = [7]$	
$d = [\,]$	$j = [\,]$	$p = [\,]$	$v = [\,]$	
$e = [8]$	$k = [\,]$	$q = [\,]$	$w = [\,]$	
$f = [\,]$	$l = [\,]$	$r = [4,9]$	$x = [\,]$	

La concaténation de ces listes dans l'ordre alphabétique donne la liste

$$[1,6,8,5,4,9,2,3,7]$$

codant la permutation *inverse* de la permutation σ telle que

$$\sigma(a_1t_2t_3r_4i_5b_6u_7e_8r_9) = a\,b\,e\,i\,r\,r\,t\,t\,u$$

puisque les listes non-vides contiennent les positions des lettres de l'alphabet dans le mot considéré.

Question 2. Si v désigne le mot obtenu en triant les lettres du mot u dans l'ordre alphabétique, par exemple $v = aerst$ pour $u = rates$, écrivez un algorithme `TriAlpha(u)` qui renvoie σ^{-1}, la permutation inverse de la permutation σ telle que $\sigma(u) = v$ en adaptant le tri par répartition ($\sigma(r_3a_1t_5e_2s_4) = a_1e_2r_3s_4t_5$ avec $\sigma = [3,1,5,2,4]$)

Solution. On formalise ce qui a été décrit dans la première question, en créant une table de q listes initialement vides, dans lesquelles on insère les positions des lettres du mot u au fur et à mesure, puis on concatène ces listes (cf. Algo 1).
À défaut d'utiliser la concaténation des listes, algorithme qui n'était pas explicitement cité dans le sujet, il était tout à fait possible de vider les éléments des casiers les uns après les autres pour reconstituer la permutation inverse, en défilant ou en dépilant, du moment que le calcul de complexité était cohérent.

Question 3. Calculez sa complexité.

Solution. La complexité dépend du nombre $n := |u|$ de symboles dans le mot u mais également de l'instance u choisie pour une même longueur. On distingue donc les instructions dont le coût ne dépend que de la longueur de u des autres. L'allocation des casiers `#01` ainsi que leur concaténation via la boucle `#09:#12` a un coût de $\Theta(q)$, les deux initialisations `#02` et `#08`

```
     ALGORITHME TriAlpha(u):liste
     DONNEES
      · u: mot
     VARIABLES
      · C: liste de listes          # les casiers
      · invsigma: liste d'entiers   # la permutation inverse
      · i: entier
     DEBUT
 01   · Allouer(C,q,[])             # allocation des q "casiers"
 02   · i ← 1
>03   · TQ (i <= |u|) FAIRE         # on parcourt le mot u et on
 04   ·  · Enfiler(C[Adr(u[i])], i) # insère la position i du symbole
 05   ·  · i ← i + 1                # u[i] à la fin du casier Adr(u[i])
<06   · FTQ
 07   · invsigma ← []               # initialisation de la permutation
 08   · i ← 1
>09   · TQ (i <= q) FAIRE           # on concatène les casiers
 10   ·  ·  invsigma ← Concat(invsigma, C[i])
 11   ·  ·  i ← i + 1
<12   · FTQ
 13   · RENVOYER invsigma
     FIN
```

Algo. 1. Tri alphabétique des lettres d'un mot u.

de la variable `i` et celle de la permutation en `#07` a un coût de $\Theta(1)$, soit un total de $\Theta(q)$. Notons que $\Theta(q) = \Theta(1)$ puisque la taille de l'alphabet est une constante, conserver la valeur q a pour simple objectif de mettre en évidence la constante cachée.

Seule la complexité de la boucle `#03:#06` dépend de l'instance u. Dans le meilleur des cas, les n symboles de u sont uniformément répartis dans l'alphabet $\mathscr{A}$ (la complexité est une notion *asymptotique*, il faut considérer que $n \to \infty$) dans les casiers différents. Dans ce cas, notons $n = qk + r$ où k est le quotient et $r < q$ le reste de la division euclidienne de n par q. On a donc r casiers contenant $k+1$ positions et $q - r$ en contenant k. L'insertion se faisant en fin de liste, pour insérer le i-ème terme dans une liste, il faut $i - 1$ opérations en $\Theta(1)$ pour parcourir les termes déjà présents dans la liste et une dernière pour l'insérer, donc i opérations en $\Theta(1)$. Au total on a donc pour nombre d'opérations

$$\left(r \sum_{i=1}^{k+1} i + (q-r) \sum_{i=1}^{k} i \right) \Theta(1) = \frac{1}{2}(k+1)(qk+2r)\Theta(1)$$

Mais $n = qk + r$ et $k = (n-r)/q$, donc finalement

$$\check{T}(n) = \Theta(q) + \frac{\Theta(1)}{2q}\left(n^2 + qn + r(q-r)\right) \quad \text{avec} \quad 0 \leqslant r < q.$$

Une simple étude de signe de la fonction dérivée $r \mapsto q - 2r$ de la fonction définie par $f : r \mapsto r(q-r)$ montre que f est croissante dans l'intervalle $[0, \frac{q}{2}]$ puis décroissante dans l'intervalle $[\frac{q}{2}, q[$. Son minimum 0 est atteint en $r = 0$ et son maximum $\frac{q^2}{4}$, en $r = \frac{q}{2}$. Dans les deux cas, $\check{T}(n) = \Theta(n^2)$ avec une constante cachée de $\frac{1}{2q}$.

Dans le pire des cas, les n symboles de u sont identiques et sont insérés dans le même casier, ce qui a un coût de $\frac{1}{2}n(n+1)\Theta(1) = \Theta(n^2)$ opérations. Finalement

$$\hat{T}(n) = \Theta(q) + \Theta(n^2) = \Theta(n^2)$$

mais cette fois avec une constante cachée égale à $\frac{1}{2}$.

Question 4. Montrez que dans ce contexte, on peut se contenter d'insérer les positions en tête des listes au lieu de le faire en fin de liste.

Solution. L'ordre dans lequel apparaissent des lettres identiques dans un mot n'a pas d'importance puisqu'on ne les distingue pas dans ce mot. Par conséquent, le produit de la permutation σ avec n'importe quelle permutation qui laisse chaque groupe de lettres identiques globalement invariant fournit la même anagramme. En inversant l'ordre des positions, on a simplement choisi une de ces variantes.

NB. L'étude de la complexité de l'algorithme `TriAlpha` avec insertion en tête de liste est plus simple puisque l'insertion d'une position est toujours en $\Theta(1)$ ce qui donne une complexité dans tous les cas de $\Theta(n)$.

Question 5. Soit u et v deux mots de $\mathscr{A}^*$. Démontrez que $u \leftrightsquigarrow v$ si et seulement si $\tau(u) = \tau(v)$ où τ désigne la fonction qui trie un mot dans l'ordre alphabétique.

Solution. Supposons que $u \leftrightsquigarrow v$, on dispose donc d'un entier n tel que $|u| = |v| = n$ et d'une permutation σ telle que $v = \sigma(u)$. Notons t le mot u trié dans l'ordre l'aphabétique qui est une anagramme de u par

construction. On dispose donc d'un permutation σ_u telle que $\sigma_u^{-1}(t) = u$ et comme $v = \sigma(u)$, on en déduit :

$$\begin{aligned} v &= \sigma(\sigma_u^{-1}(t)) \\ \sigma^{-1}(v) &= \sigma_u^{-1}(t) \\ \sigma_u(\sigma^{-1}(v)) &= t \\ (\sigma_u \circ \sigma^{-1})(v) &= t. \end{aligned}$$

Par conséquent, v et t ont même longueur n et la permutation $\sigma_u \circ \sigma^{-1}$ trie le mot v en le même mot t que le tri alphabétique de u.

Réciproquement, si u et v ont le même mot t une fois triés dans l'ordre alphabétique, ils ont bien sûr même longueur n par transitivité de l'égalité et on dispose de deux permutations σ_u et σ_v telles que

$$(\sigma_u(u) = t) \wedge (\sigma_v(v) = t). \tag{2}$$

Par transitivité de l'égalité à nouveau, on en déduit que

$$\begin{aligned} \sigma_v(v) &= \sigma_u(u) \\ v &= \sigma_v^{-1}(\sigma_u(u)) \end{aligned} \tag{3}$$

Donc $v = \sigma(u)$ pour la permutation $\sigma := \sigma_v^{-1} \circ \sigma_u$, ce qui achève de prouver que $u \leftrightsquigarrow v$.

Question 6. Écrivez un algorithme `Inverser(s)` qui renvoie la permutation inverse $\mathtt{s}^{-1}$ de la permutation `s`. Quelle est sa complexité ?

Solution. Pour toute permutation $\sigma \in S_n$, puisque $\sigma(i)$ est l'image de i et que σ est une bijection, l'image inverse de $\sigma(i)$ est donc i. Ainsi, il suffit de parcourir les n valeurs $\sigma(i)$ qui sont les indices de la liste inverse et d'y ranger la valeur i (cf. Algo 2).
La complexité de l'algorithme dépend uniquement du degré n de la permutation et c'est une simple boucle `#3:#6` dont les deux instructions sont en $\Theta(1)$, sa complexité est donc $T(n) = \Theta(n)$.

Question 7. Écrivez un algorithme `Composer(s1, s2)` qui renvoie la permutation composée `s1` $\circ$ `s2`. Quelle est sa complexité ?

Solution. Par définition, $s_1 \circ s_2(i) = s_1(s_2(i))$, il suffit donc de composer les deux permutations en paramètre, l'étude de la complexité de l'algorithme étant identique à celle de la question précédente (cf. Algo. 3). NB. On suppose que les permutations sont de même longueur.

```
     ALGORITHME Inverser(s):liste
     DONNEES
      · s: liste d'entiers # permutation
     VARIABLES
      · is: liste d'entiers # permutation inverse de s
      · i: entier
     DEBUT
 01   · Allouer(is,|s|,0)    # création de la liste is
 02   · i ← 1
>03   · TQ (i <= |s|) FAIRE  # on parcourt la permutation s:
 04   ·  · is[s[i]] ← i
 05   ·  · i ← i + 1
<06   · FTQ
 07   · RENVOYER is
     FIN
```

Algo. 2. Calcul de la permutation inverse de s.

```
     ALGORITHME Composer(s1,s2):liste
     DONNEES
      · s1,s2: listes de n valeurs # permutations
     VARIABLES
      · s: liste d'entiers # permutation produit s1°s2
      · i: entier
     DEBUT
 01   · Allouer(s,|s1|,0)      # création de la liste t
 02   · i ← 1
>03   · TQ (i <= |s|) FAIRE   # on parcourt les n valeurs [1,n]
 04   ·  ·  s[i] ← s1[s2[i]]
 05   ·  ·  i ← i + 1
<06   · FTQ
 07   · RENVOYER s
     FIN
```

Algo. 3. Calcul de la permutation composée de s_1 et s_2.

Question 8. En déduire un algorithme `EstAnagramme(u,v)` qui renvoie la permutation σ telle que $v = \sigma(u)$ si $u \leftrightsquigarrow v$ et la liste vide sinon.

Solution. D'après les questions précédentes, on sait que $u \leftrightsquigarrow v$ si et seulement si en triant u et v dans l'ordre alphabétique, on obtient le même mot t. Si σ_u^{-1} et σ_v^{-1} désignent les permutations renvoyées par l'algorithme `TriAlpha` pour les mots u et v respectivement, alors on a vu en (3) que $\sigma := \sigma_v^{-1} \circ \sigma_u$. L'algorithme en découle, on teste au préalable l'égalité des longueurs entre les deux mots pour éviter des calculs inutiles, puis on appelle l'algorithme `TriAlpha` pour chacun des mots u et v pour calculer la permutation composée $\sigma := \sigma_v^{-1} \circ \sigma_u$ que l'on renvoie si $v = \sigma(u)$ (cf. Algo. 4). L'algorithme auxiliaire `Permuter(u,sigma)` (Algo. 5) renvoie

```
    ALGORITHME EstAnagramme(u,v):liste
    DONNEES
     · u,v: mots
    VARIABLES
     · s: liste d'entiers # permutation
    DEBUT
01   · SI (|u| = |v|) ALORS
02   ·  · s ← Composer(TriAlpha(v), Inverser(TriAlpha(u)))
03   ·  · SI (v = Permuter(u,s))
04   ·  ·  · RENVOYER s
05   ·  · FSI
06   · FSI
07   · RENVOYER []
    FIN
```

Algo. 4. Calcul de la permutation σ si $v = \sigma(u)$.

le mot u dont les lettres ont été permutées suivant la permutation σ, sa complexité est en $\Theta(n)$.

Question 9. Calculez sa complexité.

Solution. Dans le meilleur des cas, les mots ont des longueurs différentes et la complexité est $\check{T}(n) = \Theta(1)$. Le pire des cas est obtenu dès que v est une anagramme de u et en supposant que l'on a utilisé la version linéaire de `TriAlpha`, tous les algorithmes appelés sont linéaires en $\Theta(n)$, il suffit de les compter. Il y en a 6 : deux tris, une inversion, une composition,

```
    ALGORITHME Permuter(u,sigma):mot
    DONNEES
     · u: mot
     · sigma: liste # Permutation
    VARIABLES
     · i: entier
     · pu: mot # Le mot u permuté
    DEBUT
 01  · Allouer(pu,|u|)
 02  · i ← 1
>03  · TQ (i <= |u|) FAIRE
 04  ·  · pu[sigma[i]] ← u[i] # Le i-ème symbole de u
 05  ·  · i ← i + 1           # va en position sigma[i]
<06  · FTQ
 07  · RENVOYER pu
    FIN
```

Algo. 5. Calcul de la permutation des lettres du mot u.

une permutationet une comparaison dur n symboles. On en conclut que $\hat{T}(n) = \Theta(n)$.

Question 10. Écrivez un algorithme `Partitionner(L)` qui partitionne une liste L de mots définis sur $\mathscr{A}$ pour la relation $\leftrightsquigarrow$.

Solution. On cherche, comme toujours, à écrire l'algorithme le plus efficace possible. On sait d'après ce qui précède que deux mots u et v sont dans la même classe d'équivalence si et seulement $\tau(u) = \tau(v)$. On commence donc par trier chaque mot u de la liste L dans l'ordre alphabétique et on range le couple $(u, \tau(u))$ dans une nouvelle liste A. On trie ensuite la liste A dans l'ordre lexicographique suivant les mots $\tau(u)$, ainsi tous les mots u d'une même classe d'équivalence se succèdent. Il ne restera plus qu'à parcourir cette liste et à ranger le mot u dans la classe en cours de construction ou à créer une nouvelle classe si $\tau(u)$ a changé de valeur. (cf. Algo. 6)

Question 11. Calculez sa complexité.

```
    ALGORITHME Partitionner(L):liste de listes
    DONNEES
     · L: liste de mots
    VARIABLES
     · A: liste de couples de mots # la liste triée des mots triés
     · P: liste de listes de mots  # la partition
     · C: liste de mots            # une classe d'anagrammes
     · r: mot                      # représentant d'une classe
    DEBUT
 01  · i ← 1
 02  · Allouer(A,|L|,(,))
>03  · TQ (i <= |L|) FAIRE  # rangement du couple (u,utrié) dans A
 04  ·  · A[i] ← (L[i], Permuter(L[i], Inverser(TriAlpha(L[i]))))
 05  ·  · i ← i + 1
<06  · FTQ
 07  · TriLexico(A)          # tri lexico. de A suivant mots triés
 08  · P ← []                # initialisation de la partition
 09  · C ← []                # initialisation de la 1ère classe
 10  · r ← ""                # initialisation du représentant
 11  · i ← 1
>12  · TQ (i <= |A|) FAIRE   # on parcourt A
 13  ·  · (u, utrie) ← A[i]
 14  ·  · SI (utrie != r) ALORS
 15  ·  ·  · SI (r != "") ALORS
 16  ·  ·  ·  · Empiler(P,C) # on empile la classe dans la partition
 17  ·  ·  · FSI
 18  ·  ·  · C ← [u]         # on initialise une nouvelle classe
 19  ·  ·  · r ← utrie       # on met à jour son représentant
 20  ·  · SINON
 21  ·  ·  · Empiler(C,u)    # on empile le mot dans sa classe
 22  ·  · FSI
 23  ·  · i ← i + 1
<24  · FTQ
 25  · Empiler(P,C)          # on empile la classe dans la partition
 26  · RENVOYER P
 27  FIN
```

Algo. 6. Calcul de la partition des anagrammes.

Solution. Nous savons que le tri lexicographique est linéaire en le nombre m de mots à trier si l'on suppose que la longueur des mots est asymptotiquement négligeable devant m. Il reste à intégrer le coût de la segmentation de la liste triée en classes qui se fait également en temps linéaire en m puisque l'on empile les mots dans les différentes classes et les classes dans la partition. On a donc une complexité en $\Theta(m)$.