Ensembles et Séquences vs. Tableaux — Exercices corrigés

22 septembre 2017

Ce document montre des exemples de problèmes utilisant des ensembles et des séquences avec la représentation par tableau avec longueur explicite. Il est conseillé de d'abord essayer de résoudre les exercices seul·e, puis dans un deuxième temps seulement de comparer votre solution avec la (ou les) solution(s) proposée(s).

Nous avons essayé pour chaque exercice de proposer une solution qui garantirait la note maximale à un·e étudiant·e. La solution est donc la plus concise et rapide possible.

Informations complémentaires

Vous trouverez dans ces encadrés des informations complémentaires qui ne seraient pas attendues dans une réponse d'étudiant·e, mais que nous trouvons important de mentionner pour vous aider dans votre apprentissage.

Exercice 1 (Somme des éléments)

Écrire un algorithme haut-niveau puis un bas-niveau qui somme les éléments d'un ensemble d'entiers. En faire une analyse de complexité.

Exercice 2 (Recherche)

Écrire un algorithme haut-niveau puis un bas-niveau qui recherche si un élément appartient à un ensemble. L'algorithme haut-niveau renvoie un booléen, tandis que le bas-niveau pourra renvoyer l'indice de l'élément s'il a été trouvé, ou une valeur particulière sinon. En faire une analyse de complexité.

Solution de l'exercice 1: Nous allons parcourir tous les éléments de l'ensemble et à rajouter les valeurs respectives à une variable `somme`. Cette variable est initialisée à 0. Notez que le résultat reste correct même si l'ensemble est vide.

Cas limites

Il est important de montrer que vous avez bien pensé aux cas « limites » (ou cas *particuliers*). Ici, quand l'ensemble est vide.

Algorithme haut niveau :

On fait pour l'instant abstraction de la représentation bas-niveau des ensembles.

```
somme_ensemble (E : Ensemble) : entier
    somme ← 0
    pour chaque e ∈ E faire
        somme ← somme + e
    retourner somme
```

Algorithme bas niveau : Avec une représentation par tableau avec longueur explicite cela donne :

```
somme_ensemble (E : Ensemble) : entier
    somme ← 0
    pour i de 0 à E.longueur−1 faire
        somme ← somme + E.tab[i]
    retourner somme
```

Analyse de complexité : L'initialisation (somme à 0) et la finalisation (retour de la valeur) sont en $O(1)$. Le corps de boucle est en $O(1)$ et la boucle est exécutée autant de fois que la longueur de l'ensemble. La complexité est donc de $O(1) + n \times O(1) + O(1) = O(n)$.

Pour juger des algorithmes, deux critères majeurs : la correction et la performance. La correction indique si l'algorithme renvoie une réponse correcte. La performance est liée aux ressources nécessaires à l'exécution de l'algorithme. On s'intéresse ici au temps d'exécution (on pourrait s'occuper aussi de la mémoire).

Pour donner une indication de performance de l'algorithme sans se soucier de l'environnement d'exécution exact, les algorithmes sont caractérisés en nombres d'instructions : que plus il y a d'instructions à exécuter, plus cela sera long (et donc moins efficace).

Le nombre d'instructions exécutées par un algorithme peut dépendre ou non de la taille des données qui sont à traiter. On parle de *complexité* d'algorithmes. On a ici deux parties dans notre algorithme :

— $O(1)$: exécution d'un nombre constant d'instructions, quelque soient les données traitées.
— $O(n)$: nombre d'instructions est linéairement proportionnel à la taille des données traitées (parcours avec une boucle).

Ce qui nous intéresse ici est ce qui « compte le plus », i.e., ce qui est le plus sensible à une augmentation de la taille de l'entrée. On garde donc la partie $O(n)$ dans notre algorithme et pas les $O(1)$ qui deviennent négligeable quand n augmente.

Solution de l'exercice 2: On initialise une variable booléenne `trouvé` avec la valeur `Faux` puis on parcourt un à un tous les éléments de l'ensemble. Si la valeur correspond à celle recherchée, on l'a trouvée et on met la variable `trouvé` à `Vrai`. Si la valeur recherchée n'est pas dans l'ensemble, la variable `trouvé` restera à `Faux`. À la fin du parcours on retourne la variable `trouvé`.

Cas particulier

On pourrait également préciser ici que si l'ensemble est vide, la valeur n'appartient pas à l'ensemble et donc la variable `trouvé` qui est à `Faux` est bien la valeur correcte à retourner.

Algorithme haut niveau

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← Faux
    pour chaque e ∈ E faire
        si x = e alors
            trouvé ← Vrai
    retourner trouvé
```

Algorithme bas niveau

Dans le cas où un ensemble est représenté par un tableau avec longueur explicite, l'algorithme bas niveau peut retourner un indice plutôt qu'un booléen. On utilise la valeur -1 si l'élément n'a pas été trouvé.

```
Rechercher(x : élément, E : Ensemble) : entier
    trouvé ← -1
    pour i de 0 à E.longueur−1 faire
        si x = E.tab[i] alors
            trouvé ← i
    retourner trouvé
```

Si l'ensemble contient plusieurs fois l'élément recherché, la valeur renvoyée sera l'indice de la *dernière* occurrence.

Erreur fréquente

Si l'on cherche toujours à renvoyer un booléen, une erreur fréquente consiste à écrire l'algorithme comme ci-dessous.

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← Faux
    pour i de 0 à E.longueur−1 faire
        trouvé ← (x = E.tab[i])
    retourner trouvé
```

Cet algorithme est faux ! En effet, la valeur de la variable `trouvé` reflète le *dernier* test effectué. Ainsi, si la valeur recherchée a été précédemment trouvée mais que la comparaison courante échoue, `trouvé` sera remis à `Faux`, alors qu'il devrait garder une valeur `Vrai`. Si on tient absolument à mettre à jour la valeur de `trouvé` à chaque comparaison, il faudrait écrire l'algorithme comme ceci :

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← Faux
    pour i de 0 à E.longueur−1 faire
        trouvé ← (trouvé ou (x = E.tab[i]))
    retourner trouvé
```

Ainsi, dès que `trouvé` passe à `Vrai`, il le reste car `Vrai ou XXX` est toujours `Vrai`. Néanmoins, mettre à jour la valeur de `trouvé` à chaque itération rajoute des instructions inutiles ce qui rend l'algorithme moins efficace.

Algorithme plus efficace

Une solution plus efficace est d'arrêter le parcours dès que la valeur recherchée a été trouvée. L'algorithme haut niveau devient :

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← Faux
    pour chaque e ∈ E faire
        si x = e alors
            trouvé ← Vrai
            sortir de la boucle
        /* sinon, continuer                                    */
    retourner trouvé
```

Une implémentation bas-niveau possible, en utilisant des tableaux à longueur explicite,

une boucle “tant que”, et en renvoyant un indice.

```
Rechercher(x : élément, E : Ensemble) : booléen
    trouvé ← −1
    i ← 0
    tant que trouvé = −1 et i < E.longueur faire
        si x = E.tab[i] alors
            trouvé ← i
        i ← i+1
    retourner trouvé
```

Notez que cette fois, s’il y a plusieurs fois l’élément cherché dans l’ensemble, on retourne l’indice de la *première* occurrence.

Complexité : L’initialisation et la finalisation sont en $O(1)$. Le corps de boucle effectue une comparaison et parfois une affectation. Dans tous les cas, c’est un nombre constant d’opérations soit $O(1)$, et la boucle est exécutée n fois. L’algorithme a donc une complexité algorithmique en $O(n)$.

En effet, la quantité d’instructions à exécuter augmente linéairement en fonction de la taille de l’ensemble à traiter (il y a n comparaisons : une pour chacun des éléments de l’ensemble, et au plus n affectations (si tous les éléments sont identiques à celui recherché).

La solution “plus efficace” a une complexité algorithmique du même ordre $O(n)$ mais en moyenne, elle n’effectue que $n/2$ opérations et est donc plus efficace (si l’élément recherché est dans l’ensemble). Dans le meilleur cas, la valeur recherchée correspond à celle du premier élément testé et une seule opération est donc suffisante. En revanche, dans le pire cas, l’algorithme exige quand même n opérations (si la valeur recherchée est trouvée dans le dernier élément testé ou si elle n’est pas présente).