Algorithmes de parcours séquentiel d'un tableau

Dernière mise à jour le : 25/10/2023

Les algorithmes de parcours séquentiel permettent de parcourir les éléments d'un tableau à l'aide d'une boucle.

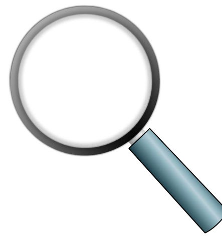
Ils font partie des algorithmes fondamentaux en informatique, leur parfaite maîtrise est donc indispensable car ils sont à la base de beaucoup d'autres algorithmes qui doivent passer en revue tous les éléments d'une collection (tableau, tuple, liste, dictionnaires, etc.).

On détaille dans ce chapitre les trois algorithmes au programme :

- l'algorithme de recherche d'un élément (ou d'une occurrence)
- l'algorithme de recherche d'un extremum (maximum ou minimum)
- l'algorithme de calcul d'une moyenne

Algorithme de recherche d'un élément

On veut écrire un algorithme qui cherche si un élément est présent ou non dans un tableau.



Spécification de l'algorithme

> La **spécification** d'un algorithme précise de manière non ambigüe ce que doit faire un algorithme. En particulier, on y indique : le nom des données manipulées (entrées), le nom des données renvoyées (sorties), le rôle de l'algorithme ainsi que les hypothèses sur les entrées (précondition) et les sorties (postcondition). Un algorithme sera considéré comme *correct* s'il respecte cette spécification.

Voici la spécification de l'algorithme de recherche d'un élément :

- **Entrée(s)** : un tableau `T` de taille n et une valeur `v`
- **Sortie(s)** : un booléen `trouvee`
- **Rôle** : chercher si la valeur `v` est présente dans `T`
- **Précondition(s)** : `v` est du même type que les éléments de `T`
- **Postcondition(s)** : `trouvee` vaut Vrai si `v` appartient à `T`, et Faux sinon

Algorithme

L'idée est la suivante :

- le booléen `trouvee` est initialisé à `Faux`
- on parcourt une à une les valeurs du tableau `T` :
  - si on en trouve une égale à `v`, le booléen `trouvee` prend la valeur `Vrai`
  - sinon on ne fait rien

Voici l'algorithme :

```
trouvee ← Faux
pour i de 0 à n-1 faire :
    si T[i] = v alors :
        trouvee ← Vrai
    fin si
fin pour
```

Coût de l'algorithme

Le **coût d'un algorithme** est le nombre d'opérations élémentaires (arithmétiques ou logiques) ainsi que d'affectations nécessaires à son exécution **dans le pire cas**.
Le coût d'un algorithme dépend toujours de la taille des données d'entrée (ici la taille n du tableau).

Dans tous les cas, notre algorithme va effectuer n tours de boucle, donc dans le pire cas aussi. Ainsi, la comparaison `T[i] = v` sera effectuée n fois.

Si tous les éléments du tableau sont égaux à `v` alors l'affectation `trouvee ← Vrai` est effectuée à chaque tour de boucle, donc n fois.

On dénombre ainsi :

Comparaisons	**Affectations**	**Opérations arithmétiques**
n	$n+1$	0

Le coût de l'algorithme (dans le pire cas) est donc $n+(n+1)+0=2n+1$.

Comme ce coût est de l'ordre de n, on dit que le coût de l'algorithme de recherche d'un élément est **linéaire**.

Écriture en une fonction Python

On peut écrire une fonction `appartient()` possédant deux paramètres `v` et `T` qui implémente cet algorithme. Cette fonction renvoie `True` si `v` est dans `T` et `False` sinon.

```
def appartient(v, T):
    trouvee = False
    for i in range(len(T)):
        if T[i] == v:
            trouvee = True
    return trouvee
```

On peut tester cette fonction :

```
>>> appartient(5, [2, 3, 1, 5, 0])
True
>>> appartient(1, [0, 3, 0, 0])
False
```

Parcours par valeur

Il est aussi possible de parcourir le tableau *par valeur* et non *par indice* car on n'a pas besoin des indices d'après la spécification de l'algorithme. La fonction suivante implémente cette possibilité :

```
def appartient_v2(v, T):
    trouvee = False
    for e in T:
        if e == v:
            trouvee = True
    return trouvee
```

On peut tester :

```
>>> appartient_v2(5, [2, 3, 1, 5, 0])
True
>>> appartient_v2(1, [0, 3, 0, 0])
False
```

Sortie anticipée

L'algorithme ainsi écrit et implémenté par les deux fonctions précédentes peut sembler non optimal. En effet, si le premier élément du tableau `T` est la valeur `v` alors on sait de suite que la réponse est "Vrai" mais notre algorithme va quand même tester toutes les valeurs suivantes du tableau `T`.

On peut utiliser le mot clé `return` pour stopper l'exécution de notre fonction dès que la valeur `v` est trouvée. Dans le cas, où la boucle `for` a atteint la fin du tableau sans trouver `v`, il faut renvoyer `False`.

```
def appartient_v3(v, T):
    for e in T:
        if e == v:
            return True  # renvoie True dès qu'on trouve v (ce qui stoppe l'exécution de la fonction)
    return False  # après la boucle (si v n'a pas été trouvé)
```

```
>>> appartient_v3(5, [2, 3, 1, 5, 0])
True
>>> appartient_v3(1, [0, 3, 0, 0])
False
```

Remarques :

1. C'est cette dernière version qui est la plus simple et la plus rapide à écrire. On pourra donc l'utiliser sans problème.
2. Si notre algorithme ne devait plus renvoyer Vrai ou Faux mais l'indice de la valeur `v` cherchée, alors il aurait été obligatoire de parcourir le tableau par indice, pour garder trace des indices et renvoyer celui demandé. Le choix du parcours dépend donc de la spécification de l'algorithme à écrire !
3. On aurait également pu écrire une version optimisée (avec arrêt dès que `v` est trouvée) sans utiliser une sortie anticipée (avec `return`) à condition d'utiliser une boucle `while`. Cependant l'écriture est plus longue et compliquée :

```
def appartient_v4(v, T):
    i = 0
    trouvee = False
    while i <= len(T) and trouvee == False:
        if T[i] == v:
            trouvee = True
        i = i + 1
    return trouvee
```

Algorithme de recherche du maximum

On veut écrire un algorithme qui recherche la valeur maximale dans un tableau.

> La recherche du maximum est présentée ici, mais ce qui va être dit est vrai pour la recherche d'un extremum de manière générale, qu'il s'agisse d'un maximum ou d'un minimum).

Spécification de l'algorithme

- **Entrée(s)** : un tableau `T` de taille n
- **Sortie(s)** : un entier `maxi`
- **Rôle** : chercher l'élément maximal de `T`
- **Précondition(s)** : `T` est un tableau *non vide* d'entiers
- **Postcondition(s)** : `maxi` est un entier qui est l'élément maximal de `T`

Algorithme

L'idée est la suivante :

- on initialise la valeur maximale par le premier élément du tableau ;
- on parcourt une à une les valeurs du tableau `T` (à partir de la deuxième) :
    - si on en trouve une strictement supérieure au maximum provisoire, cette valeur devient notre nouveau maximum (provisoire).
    - sinon, on ne fait rien

Voici l'algorithme :

```
maxi ← T[0]
pour i de 1 à n-1 faire :
    si T[i] > maxi alors :
        maxi ← T[i]
    fin si
fin pour
```

Coût de l'algorithme

Dans tous les cas, notre algorithme va effectuer $n-1$ tours de boucle, donc dans le pire cas aussi. Ainsi, la comparaison `T[i] > maxi` sera effectuée $n-1$ fois.

Si les éléments sont rangés dans l'ordre (strictement) croissant, alors la condition `T[i] > maxi` est vraie à chaque tour de boucle et l'affectation `maxi ← T[i]` est effectuée à chaque tour de boucle.

Le *pire cas* est donc un tableau dans lequel chaque élément est strictement supérieur au précédent (par exemple `[1, 2, 3, 4, 5]` pour un tableau de $n=5$ éléments).

On dénombre ainsi dans le pire cas :

Comparaisons	Affectations	Opérations arithmétiques
$n-1$	n	0

Remarque : on n'a pas compté la soustraction du `n-1` (cela ne change rien au coût)

Le coût de l'algorithme (dans le pire cas) est donc $n-1+n+0=2n-1$.

Comme ce coût est de l'ordre de n, on dit que le coût de l'algorithme de recherche d'un extremum est **linéaire**.

Écriture en une fonction Python

On peut écrire une fonction `maximum()`, possédant en paramètre un tableau d'entiers `T` non vide, qui implémente cet algorithme. Cette fonction renvoie la valeur maximale présente dans `T`.

```python
def maximum(T):
    maxi = T[0]
    for i in range(1, len(T)):
        if T[i] > maxi:
            maxi = T[i]
    return maxi
```

On peut tester :

```python
>>> maximum([1, 2, 3, 4, 5])
5
>>> maximum([7, 9, 2])
9
```

Parcours par valeur

Il est aussi possible de parcourir le tableau *par valeur* et non *par indice* car on n'a pas besoin des indices d'après la spécification de l'algorithme. La fonction suivante implémente cette possibilité :

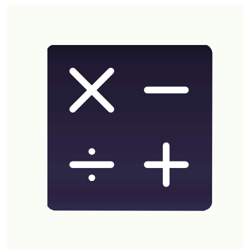
```
def maximum_v2(T):
    maxi = T[0]
    for e in T:
        if e > maxi:
            maxi = e
    return maxi
```

Remarques :

1. Cette version a le (petit) désavantage de parcourir le premier élément du tableau (au premier tour de boucle) alors que ce n'est pas utile, mais cela ne change rien au coût, qui reste linéaire, car cela n'ajoute qu'une comparaison et qu'une affectation (le coût est donc $2n+1$, donc toujours linéaire).
2. Si notre algorithme ne devait plus renvoyer la valeur maximale mais une position de cette valeur maximale, alors il aurait été obligatoire de parcourir le tableau par indice, pour garder trace des indices et renvoyer celui demandé. Le choix du parcours dépend donc de la spécification de l'algorithme à écrire !

Algorithme de calcul d'une moyenne

On veut écrire un algorithme qui calcule la moyenne des nombres présents dans un tableau.



Spécification de l'algorithme

- **Entrée(s)** : un tableau `T` de taille n
- **Sortie(s)** : un réel `m`
- **Rôle** : calculer la moyenne des éléments de `T`
- **Précondition(s)** : `T` est un tableau *non vide* d'entiers
- **Postcondition(s)** : `m` est la moyenne des éléments de `T`

Algorithme

L'idée est de calculer la somme des valeurs de `T` puis de calculer la moyenne en divisant la somme par le nombre d'éléments de `T` :

- on initialise la somme des valeurs à 0 ;
- on parcourt une à une les valeurs du tableau `T` :
  - on ajoute chaque valeur à notre somme
- la réponse est la somme divisée par `n` (le nombre d'éléments de `T`)

Voici l'algorithme :

```
s ← 0
pour i de 0 à n-1 faire :
    s ← s + T[i]
fin pour
m ← s/n
```

Coût de l'algorithme

Dans tous les cas, notre algorithme va effectuer n tours de boucle, donc dans le pire cas aussi. Ainsi, l'affectation `s ← s + T[i]` est effectuée à chaque tour de boucle.

N'importe quel tableau est donc un *pire cas* pour notre algorithme.

On dénombre ainsi :

Comparaisons	Affectations	Opérations arithmétiques
0	$n+2$	$n+1$

Remarque : il y a n additions (`s ← s + T[i]`) et *une* division (`s/n`) donc $n+1$ opérations arithmétiques.

Le coût de l'algorithme (dans le pire cas) est donc $0+(n+2)+(n+1)=2n+3$.

Comme ce coût est de l'ordre de n, on dit que le coût de l'algorithme de calcul d'une moyenne est **linéaire**.

Écriture en une fonction Python

On peut écrire une fonction `moyenne()`, possédant en paramètre un tableau d'entiers `T` non vide, qui implémente cet algorithme. Cette fonction renvoie la moyenne des valeurs présentes dans `T`.

```
def moyenne(T):
    s = 0
    for i in range(len(T)):
        s = s + T[i]
    return s / len(T)
```

On peut tester :

```
>>> moyenne([10, 14, 6])
10.0
>>> moyenne([8, 15, 12, 14, 19, 11])
13.166666666666666
```

Parcours par valeur

Il est aussi possible de parcourir le tableau *par valeur* et non *par indice* car on n'a pas besoin des indices. La fonction suivante implémente cette possibilité :

```
def moyenne_v2(T):
    s = 0
    for e in T:
        s = s + e
    return s / len(T)
```

Remarque : Cette fonction est légèrement plus simple à écrire et peut donc être privilégiée pour la calcul de la moyenne des éléments d'un tableau.

Bilan

- La **spécification** d'un algorithme définie de manière non ambigüe ce qu'un algorithme doit faire, les données avec lesquelles il travaille (les *entrées*) et les données qu'il renvoie (les *sorties*).
- Un algorithme de **parcours séquentiel** d'un tableau parcoure les éléments du tableau à l'aide d'une boucle. Ces algorithmes sont à la base de beaucoup d'autres algorithmes.
- Les trois algorithmes de parcours séquentiel étudiés ont tous un **coût linéaire**, c'est-à-dire un coût de l'ordre de la taille n du tableau parcouru.
- Le type de parcours, par indice ou par valeur, dépend de l'algorithme et de l'exercice.
- Nous verrons d'autres algorithmes au cours de l'année, dont certain n'ont pas un coût linéaire.

Références :

- Documents ressources de l'équipe éducative du DIU EIL, Université de Nantes, Christophe JERMANN et Christophe DECLERCQ.

Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS cc BY NC SA