Chapitre 3 : Recherche dichotomique

Introduction

On a déjà écrit et implémenté un algorithme pour rechercher une valeur v dans un tableau T. Plus précisément, on avait écrit une fonction `recherche_sequentielle(v, T)` qui renvoie `True` si v appartient à T, et `False` sinon.

Complétez les pointillés dans le code suivant qui implémente cette fonction `recherche`.

Recherche séquentielle (rappel)

```
def recherche_sequentielle(v, T):
    """Renvoie True si v appartient au tableau T, False sinon."""
    for elt in T:
        if ................................:
            return ..............
    return ..............
```

Si le tableau T a pour taille n, combien de valeurs du tableau sont parcourues dans le pire des cas avec cet algorithme ?

....

Cet algorithme de recherche a donc un coût de l'ordre de n, on parle de **coût linéaire**. Peut-on faire mieux ? Autrement dit, peut-on effectuer une recherche sans parcourir tous les éléments du tableau ? La réponse est OUI, à condition que le tableau soit trié !

Algorithme de recherche dichotomique

Si un tableau T de n valeurs est **trié** (par ordre croissant), alors il existe un moyen très efficace pour chercher un élément v dans le tableau : il suffit d'appliquer l'algorithme de **recherche dichotomique**

Principe de l'algorithme

Le principe est le suivant : comparer l'élément central du tableau avec la valeur v ; si les valeurs sont égales, alors la tâche est accomplie, sinon on recommence dans la moitié du tableau pertinente.

Autrement dit, l'algorithme consiste à :

Principe

- Trouver la position la plus centrale du tableau (si le tableau est vide, sortir et renvoyer Faux)
- Comparer la valeur de cette case à l'élément recherché v. Trois cas se présentent :
    - Si v est égale à la valeur centrale, alors renvoyer Vrai
    - Si v est strictement inférieure à la valeur centrale, alors reprendre la procédure dans la moitié gauche du tableau
    - Si v est strictement supérieure à la valeur centrale, alors reprendre la procédure dans la moitié droite du tableau

Exemple 1 : Recherche de la valeur `v = 32` dans le tableau `T = [2, 3, 9, 10, 16, 30, 32, 37, 50, 89]`.

```
[2, 3, 9, 10, 16, 30, 32, 37, 50, 89]
```


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

Écriture de l'algorithme

On cherche à écrire une fonction `recherche_dichotomique(v, T)` qui recherche une valeur `v` dans un tableau trié `T` dont voici la spécification.

Spécification de l'algorithme

- *Entrées* : tableau `T` (de taille `n` constitué d'entiers), un entier `v`
- *Sortie* : un booléen
- *Rôle* : renvoie `True` si `v` ∈ `T`, `False` sinon
- *Précondition* : `T` est **trié** par ordre croissant
- *Postcondition* : ∅

> On pourrait écrire une fonction qui ne se contente pas de renvoyer `True` ou `False` pour indiquer que la valeur `v` est trouvée ou non, mais de renvoyer une *position* de la valeur cherchée (et `None` si `v` n'est pas dans `T`). Il n'y aurait quasiment rien à modifier (fait en exo) !

Pour vous aider à écrire et comprendre l'algorithme, on va détailler le déroulé de l'algorithme en utilisant trois variables :

- `m` désigne la position la plus centrale de la zone de recherche
- `debut` et `fin` désignent respectivement les indices de début et de fin de la zone de recherche. On a donc au départ `debut = 0` et `fin = n - 1`

Exemple 2 : Recherche de la valeur `v = 32` dans le tableau `T = [2, 3, 9, 10, 16, 30, 32, 37, 50, 89]`.

Schéma en début d'itération	`m` à la fin de l'itération	`T[m]` à la fin de l'itération	Poursuite recherche	`debut` à la fin de l'itération	`fin` à la fin de l'itération
				0	9
debut (↓ sur 2), fin (↓ sur 89) [2, 3, 9, 10, 16, 30, 32, 37, 50, 89] ↑ m (sur 16)	4	16	à droite	5	9
debut (↓ sur 30), fin (↓ sur 89) [~~2, 3, 9, 10, 16~~, 30, 32, 37, 50, 89] ↑ m (sur 37)	7	37	à gauche	5	6
debut (↓ sur 30), fin (↓ sur 32) [~~2, 3, 9, 10, 16~~, 30, 32, ~~37, 50, 89~~] ↑ m (sur 30)	5	30	à droite	6	6
debut=fin (↓ sur 32) [~~2, 3, 9, 10, 16, 30~~, 32, ~~37, 50, 89~~] ↑ m (sur 32)	6	32	Valeur trouvée ! Fin de l'algo	(6)	(6)

> - On a eu besoin d'examiner 4 valeurs pour trouver la valeur cherchée, alors qu'on aurait dû en examiner 7 avec la recherche séquentielle.
> - On remarque que la variable `m` à l'issue de l'algorithme représente une position de la valeur cherchée.

CC BY NC SA

Exercice 1 :

1. Appliquez l'algorithme de recherche dichotomique pour chercher la valeur `v = 9` dans le tableau `T = [1, 2, 2, 5, 6, 6, 7, 9, 9, 11, 12, 13, 13, 15]`.

Schéma en début d'itération	`m` à la fin de l'itération	`T[m]` à la fin de l'itération	Poursuite recherche	`debut` à la fin de l'itération	`fin` à la fin de l'itération
				0	13
[1, 2, 2, 5, 6, 6, 7, 9, 9, 11, 12, 13, 13, 15]					
[1, 2, 2, 5, 6, 6, 7, 9, 9, 11, 12, 13, 13, 15]					
[1, 2, 2, 5, 6, 6, 7, 9, 9, 11, 12, 13, 13, 15]					

2. Quelle est la position de l'élément cherché ? ……………………………………………………………………………
3. Combien de valeurs du tableau `T` ont été examinées ? ………………………………………………………………..

Algorithme

Compléter la fonction `recherche_dichotomique` ci-dessous :

Recherche dichotomique

```
def recherche_dichotomique(v, T):
    """renvoie True si v est dans le tableau T, supposé trié,
    et False sinon."""
    debut = 0
    fin = len(T) - 1
    while debut <= fin:
        m = ...............................
        if v == T[m]:
            return ...............................
        elif v > T[m]:
            debut = m + 1
        else:
            fin = ...............................
    return ...............................
```

Explications (si nécessaire)

On note `debut` et `fin` les indices définissant la zone de recherche de `v` dans le tableau `T`.
Au départ, on a donc `debut = 0` et `fin = len(T) - 1` :

```
          debut                          fin
     T  [                                    ]
```

Après un certain nombre d'étapes, on se trouve dans la situation suivante :

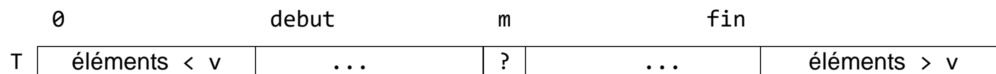
	0	debut	fin
T	éléments < v	...	éléments > v

CC BY NC SA

Le programme doit répéter ce principe de dichotomie tant que la zone de recherche n'est pas vide, donc tant que `debut` ≤ `fin`. L'utilisation d'une boucle « tant que » est toute indiquée !

```
while debut <= fin:
```

Il faut, à chaque étape, comparer l'élément `v` avec l'élément central de la zone de recherche. On va noter `m` l'indice de l'élément central. On a la situation suivante :



Pour calculer la position centrale `m`, il suffit de faire la moyenne entière de `debut` et `fin` :

```
m = (debut + fin) // 2
```

Il faut faire attention, il s'agit d'une division entière pour être certain de ne pas obtenir de valeurs décimales (`m` est nécessairement un entier).

Il reste ensuite à comparer `v` avec `T[m]`. Trois cas de figure se présentent selon que `v` soit plus petite, plus grande ou égale à `T[m]`.

Si `v` est égale à `T[m]` c'est qu'on a trouvé la valeur `v` en position `m`, il suffit de renvoyer `True`.

Si en revanche elle est plus grande, on se restreint à la moitié droite et on modifie la valeur de `debut` en conséquence :

```
if v > T[m]:
    debut = m + 1
```

Sinon se restreint à la moitié gauche en modifiant la valeur de `fin` en conséquence :

```
else:
    fin = m - 1
```

Il se peut également qu'on finisse par sortir de la boucle car la condition `debut <= fin` devient fausse. Cela signifie que la valeur `v` ne se trouve pas dans `T` et dans ce cas on renvoie `False`.

Efficacité de l'algorithme

Pour calculer le coût de cet algorithme on peut compter le nombre d'itérations de la boucle `while`, c'est-à-dire le nombre de valeurs du tableau `T` qui ont été examinées.
On se place dans le pire des cas, donc dans le cas où la valeur `v` ne se trouve pas dans `T` puisque c'est ce qui va occasionner le plus grand nombre d'itérations.

Exemple

Si on cherche de cette façon un nombre dans un tableau trié de taille n = 100 000. Après 1 comparaison, on le cherche dans un tableau d'au plus 50 000 nombres ; après 2 comparaisons, on le cherche dans un ensemble d'au plus 25 000 nombres ; etc. On arrive très vite au nombre cherché ou on constate qu'il ne se trouve pas dans le tableau.

En combien d'itérations au pire ?

cc BY NC SA

Réponse :

N° itération	Taille de la zone de recherche à la fin de chaque itération
1	$\frac{n}{2} = 50000$
2	$\frac{n}{2^2} = 25000$
3	$\frac{n}{2^3} = 12500$
…	

Coût de l'algorithme

Si on considère un tableau T de taille n, alors le nombre maximal d'itérations correspond au nombre de fois qu'il faut diviser n par 2 pour obtenir un nombre inférieur ou égal à 1.

Exercice 2 : Complétez le tableau ci-contre qui donne le nombre maximal d'itérations k pour une taille de tableau n de départ.

n	k
10	
100	
1000	
1 000 000	
1 000 000 000	

On voit qu'il s'agit d'un algorithme extrêmement efficace puisque même pour rechercher une valeur dans un tableau de taille 1 milliard, il suffit d'examiner 30 valeurs dans le pire des cas : la recherche dichotomique sera donc instantanée. A titre de comparaison, avec l'algorithme de recherche séquentielle il faudrait examiner 1 milliard de valeurs dans le pire des cas ! Bien entendu, il faut garder à l'esprit que le tableau doit être trié pour faire une recherche dichotomique.

Hors programme

Chercher le nombre de fois qu'il faut diviser n par 2 pour obtenir un nombre inférieur ou égal à 1 revient à chercher le plus petit entier k tel que $\frac{n}{2^k} \leq 1$. Cette inéquation est résolue dans l'encadré ci-dessous.

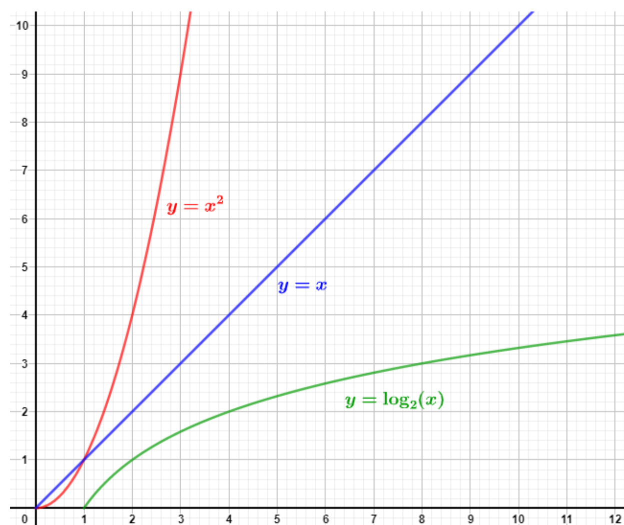
$$\frac{n}{2^k} \leq 1$$
$$\Leftrightarrow n \leq 2^k$$
$$\Leftrightarrow \log(n) \leq \log(2^k)$$
$$\Leftrightarrow \log(n) \leq k \log(2)$$
$$\Leftrightarrow \frac{\log(n)}{\log(2)} \leq k$$

On note $\log_2(n)$ le quotient $\frac{\log(n)}{\log(2)}$ et ce nombre s'appelle le **logarithme en base 2 de n**.

Le nombre d'itérations est donc égale au plus petit entier supérieur ou égal à $\log_2(n)$.

Le *coût* de la recherche dichotomique est dit *logarithmique*, noté $O(\log_2(n))$. Il s'agit d'un coût nettement inférieur à celui de la recherche séquentielle qui est linéaire ($O(n)$).

En effet, pour comparer l'efficacité de différents algorithmes, on a tracé ci-contre les courbes d'équations $y = x$ (linéaire), $y = x^2$ (quadratique) et $y = \log_2(x)$ (logarithmique).



 Germain BECKER & Sébastien POINT, Lycée Mounier, ANGERS

Terminaison de l'algorithme

Il nous reste à justifier pourquoi cet algorithme termine nécessairement (qu'il ne « tourne » pas à l'infini). On appelle cela : « prouver la *terminaison de l'algorithme* ».

Traitons tout de suite le cas où T est un tableau vide. Dans ce cas, `debut = 0` et `fin = -1` et donc `debut > fin` donc on n'entre pas dans la boucle `while` et donc l'algorithme termine (la fonction renvoie la valeur None directement).

Montrons que l'algorithme termine également si T n'est pas vide. Il faut donc montrer que la boucle while termine. Pour cela on va considérer la quantité `fin - debut`.

- La quantité `fin - debut` est toujours un nombre entier au cours de l'algorithme (car `debut` et `fin` le sont).
- Au départ `fin - debut` est positive et on entre dans la boucle `while`
- À chaque itération de la boucle `while`, trois cas se présentent :
  - Soit la valeur `v` est trouvée et le `return` termine l'algorithme
  - Soit `fin` diminue d'au moins une unité (instruction `fin = m - 1`) et donc `fin - debut` également ;
  - Soit `debut` augmente au moins d'une unité (instruction `debut = m + 1`) et donc `fin - debut` diminue au moins d'une unité.

Ainsi, on sait que la quantité `fin - debut` est un nombre entier qui diminue strictement à chaque itération donc cette quantité deviendra forcément strictement négative donc on finira par avoir `debut > fin` et donc par sortir de la boucle `while`, si on n'a pas trouvé la valeur `v` avant bien sûr.

La quantité `fin - debut` s'appelle un **variant de boucle**. C'est une quantité entière qui diminue strictement à chaque itération et qui reste positive. Le variant ne peut donc prendre qu'un nombre fini de valeurs, ce qui prouve que le nombre d'itérations de la répétitive « tant que » est fini, donc celle-ci termine

Bilan

- La recherche dichotomique permet de rechercher une valeur dans un tableau trié.
- Le principe consiste à couper en deux à chaque fois la portion du tableau dans laquelle s'effectue la recherche.
- Cet algorithme est très efficace. Il ne faut que quelques dizaines de comparaisons pour chercher une valeur dans un tableau en contenant des milliards.

Références

- Équipe éducative du DIU EIL, Université de Nantes.
- Livre *Numérique et Sciences Informatiques*, niveau Première, T. BALABONSKI, S. CONCHON, J.-C. FILLIATRE, K. NGUYEN, éditions Ellipses.

CC BY NC SA