DICHOTOMIE, TESTS, TERMINAISON (TP EN UNE SÉANCE)

1 Recherche dans un tableau trié

Principe et algorithme. Au TP 2, on a vu un algorithme pour vérifier si une valeur donnée se trouve dans un tableau. Lorsque le tableau est de taille n, cela donnait une complexité d'ordre n dans le pire cas, car il faut tester les éléments du tableau un par un. Mais, lorsque le tableau est *trié* (on supposera par ordre croissant), on peut aller beaucoup plus vite.

Si on recherche un nombre x dans un tableau trié, la *méthode par dichotomie* consiste à comparer x avec un élément m pris au milieu du tableau. Si $x = m$, alors on a fini. Si $x > m$, alors x ne peut pas se trouver dans la première moitié du tableau et si $x < m$, alors x ne peut pas se trouver dans la deuxième moitié. On a donc « éliminé » la moitié des éléments du tableau avec juste une comparaison. Puis on recommence avec la moitié restante.

Exemple. On recherche 9 dans `T = [0,1,3,5,6,9,12,14,17]`.

0 | 1 | 3 | 5 | **6** | 9 | 12 | 14 | 17	le milieu du tableau est 6,
| | | | | 9 | 12 | 14 | 17	on cherche $9 > 6$, on ne garde que la deuxième moitié,
| | | | | 9 | **12** | 14 | 17	le "milieu" du tableau est 12,
| | | | | 9 | | |	on cherche $9 < 12$, on ne garde que la première moitié,
| | | | | **9** | | |	on cherche $9 = 9$, fin de l'algorithme !

A la troisième ligne, le milieu du tableau ne tombe pas juste car il y a un nombre pair d'éléments : quand ça arrive, on prend l'élément juste avant le milieu, donc 12. C'est un choix, on aurait pu faire le contraire. Voici l'algorithme correspondant :

```
def dichotomie(T,x):
    g,d = 0,len(T)-1
    while g <= d :
        m = (g+d)//2
        if T[m] == x:
            return True
        if T[m] < x:
            g = m+1
        else:
            d = m-1
    return False
```

Exercice 1. Comprendre et commenter la fonction.

Une étape importante en programmation est de vérifier si un algorithme fonctionne. Pour en être absolument sûr, il faudrait tester ici toutes les valeurs possibles de T et x et vérifier qu'on obtient bien le résultat voulu. C'est évidemment impossible en pratique. On fait alors des tests sur des cas particuliers, qu'il faut choisir astucieusement pour tester aussi bien quelques cas généraux typiques que des cas extrêmes.

Exercice 2. Proposer un jeu de tests comportant au moins 4 cas, puis tester la fonction.

Terminaison de l'algorithme. La fonction `dichotomie` contient une boucle `while`. Contrairement aux boucles `for`, rien ne permet d'assurer qu'une boucle `while` n'est pas répétée à l'infini, par exemple :

```
i = 0
while i>=0:
    i += 1    # équivaut à i = i+1
```

Astuce : pour interrompre un programme (par exemple si on est coincé dans la boucle ci-dessus), il faut redémarrer la console avec Shell > Redémarrer ou avec le raccourci Ctrl + K. Malheureusement, cela efface aussi le Workspace, donc il faut tout recompiler.

Sur les tests ci-dessus, la boucle `while` n'est répétée qu'un nombre fini de fois : l'algorithme se termine. Nous allons le montrer dans le cas général.

Démonstration. On note d_k, g_k les valeurs des variables `d`, `g` après k itérations. Supposons par l'absurde que l'algorithme ne se termine pas : les suites (d_k) et (g_k) sont donc définies pour tout $k \in \mathbb{N}$ et on a $g_k \leq d_k$ pour tout k. De plus, la condition `T[m]==x` n'est jamais vérifiée, car la commande `return` met fin à la fonction. Soit p le plus petit entier tel que $d_0 - g_0 \leq 2^p$. Pour tout k, on remarque que

$$d_{k+1} - g_{k+1} \leq \frac{d_k - g_k}{2}$$

ce qui permet de déduire par récurrence immédiate que $d_k - g_k \leq 2^{p-k}$. En particulier, après $k = p+1$ itérations, on a $d_{p+1} - g_{p+1} < 1$. Comme $d_{p+1} - g_{p+1}$ est un entier positif par hypothèse, on en déduit que $d_{p+1} - g_{p+1} = 0$. L'itération $p+2$ conduit alors à $g_{p+2} > d_{p+2}$, ce qui est une contradiction. Donc l'algorithme se termine toujours : après un nombre fini d'itérations, on sort de la boucle `while`. □

Complexité. On s'intéresse à la complexité de la fonction `dichotomie`. Plutôt que de compter les opérations élémentaires, on va compter le nombre d'itérations de la boucle `while` [1] :

Exercice 3. Modifier la fonction `dichotomie` pour qu'elle renvoie, en plus de `True`/`False`, la taille de `T` ainsi que le nombre d'itérations de la boucle `while`. Tester le pire cas et remplir le tableau ci-desous. *On admet que le pire cas arrive lorsque l'élément* `x` *recherché n'est pas dans* `T` *et est strictement supérieur à tous les éléments de* `T`.

$n =$ `len(T)`	n en binaire	Valeur de p	Nombre d'itérations de `dichotomie`
1	1		
2	10		
3	11		
4	100		
7	111		
8	1000		
15	1111		
16	10000		

où p est le plus petit entier tel que $n \leq 2^p - 1$

On constate deux choses : primo, p est égal au nombre de chiffres de l'écriture en binaire de n. Ce n'est pas un hasard. La règle est identique en base 10 : pour tout $n \in \mathbb{N}^*$, si q est le plus petit entier tel que $n \leq 10^q - 1$, alors q est le nombre de chiffres de n. On y reviendra au second semestre. Une autre façon de le dire est que $p = \lfloor \log_2(n) \rfloor + 1$, où $\log_2(n)$ est le logarithme en base 2 de n.

Secundo, le nombre d'itérations est égale à p. Si maintenant on compte les opérations élémentaires, on trouve une complexité de l'ordre de $\log_2(n)$. C'est ce qu'on appelle la *complexité logarithmique*. Ce coût est nettement inférieur au coût linéaire qu'on a déjà vu, lorsque `T` n'est pas trié :

n	Coût linéaire (ordre n)	Coût logarithmique (ordre $\log_2(n)$)
100	100	7
1000	1000	10
10^5	10^5	14
10^6	10^6	17

On comprend aisément l'importance d'un ordre de complexité plus faible !

2 Exercices d'approfondissement **(je vous encourage vraiment à le faire !)**

Exercice 4. On a vu dans l'exemple que lorsqu'un tableau est de taille paire, le milieu ne « tombe pas juste » : avec l'instruction `m=(g+d)//2`, on choisit l'élément juste à gauche. On veut modifier le code pour prendre à la place l'élément juste à *droite* :

- Un élève modifie la ligne `m=...` par `m=(g+d)//2+1`. Tester la fonction avec un jeu de test.
- Quelle est la bonne modification à faire ? Vérifier avec un jeu de test.

1. Dans le pire cas, chaque itération entraîne 6 opérations élémentaires, donc compter les itérations suffit pour avoir l'ordre de la complexité.