Informatique PCSI
Prérequis TP 4 : algorithmes dichotomiques

Introduction

Le mot *dichotomie* vient du grec et signifie *division en deux parties*. Une méthode dichotomique est une stratégie de type *diviser pour régner*.

Le coût d'une recherche linéaire d'un élément dans un tableau de taille n est linéaire en n dans le pire des cas qui consiste à tester tous les éléments du tableau.

L'intérêt de la méthode dichotomique, est d'accélérer grandement la recherche. Mais cette méthode n'est applicable qu'à une condition : que le tableau soit trié. *Le principe est de comparer l'élément avec celui se trouvant au milieu du tableau. S'ils sont égaux, c'est terminé, sinon on recommence avec la partie du tableau pouvant contenir l'élément.*

Voici une application de cette méthode sur un exemple. On cherche l'élément 28 qui est compris entre les valeurs extrêmes du tableau 15 et 33. Les « milieux » utilisés pour les tests sont grisés et désignés par les flèches verticales. Les extrémités des sous-tableaux sont soulignées.

				↓						
15	16	18	19	23	24	28	29	31	33	$28 > 23$
							↓			
15	16	18	19	23	24	28	29	31	33	$28 < 29$
					↓					
15	16	18	19	23	24	28	29	31	33	$28 > 24$
						↓				
15	16	18	19	23	24	28	29	31	33	$28 = 28$

Si on cherche l'élément 27, le processus est le même, seule la conclusion change.

				↓						
15	16	18	19	23	24	28	29	31	33	$27 > 23$
							↓			
15	16	18	19	23	24	28	29	31	33	$27 < 29$
					↓					
15	16	18	19	23	24	28	29	31	33	$27 > 24$
						↓				
15	16	18	19	23	24	28	29	31	33	$27 \neq 28$

Programmation

Les tableaux sont représentés en Python par des listes. Nous écrivons le code d'une fonction qui prend en paramètres une liste triée et l'élément cherché. La fonction peut renvoyer `True` si l'élément est trouvé ou son indice. La difficulté est de prévoir les cas, qu'il vaut mieux tester après l'écriture d'un code, d'une liste à deux éléments, un élément, ou d'une liste vide, avec la recherche d'un élément qui appartient à la liste ou pas.

Une recherche par dichotomie s'effectue dans une liste triée. Si la liste n'est pas triée, nous disposons en Python de la fonction `sorted` qui prend en argument une liste et renvoie la liste triée. La syntaxe est `lst2 = sorted(lst1)`. La liste initiale `lst1` n'est pas modifiée. Python propose aussi la méthode `sort` qui trie en place la liste à laquelle elle s'applique. La syntaxe est `lst.sort()`.

- **Principe de l'algorithme** : à chaque étape, on coupe le tableau en deux et on effectue un test pour savoir dans quelle partie se trouve l'élément cherché.

Première version

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

Preuve de la terminaison

Nous utilisons l'expression $d-g$ qui à chaque étape représente la taille de la sous-liste qui peut contenir x. Cette expression est *un variant de boucle*.

Soit p le plus grand entier tel que la taille n du tableau est inférieure à 2^p. Avant d'entrer dans la boucle $d-g \leq 2^p$.

Après k itérations, $d-g \leq \dfrac{2^p}{2^k}$, soit $d-g \leq 2^{p-k}$. La suite des valeurs de l'expression $d-g$ est strictement décroissante et après p étapes, $d-g \leq 1$, donc $g \geq d-1$ et la boucle est terminée.

Par exemple, il faut sept étapes pour une taille de tableau égale à 100 et 10 étapes pour une taille égale à 1000.

Ceci prouve aussi que le coût de cette recherche dichotomique est de l'ordre du nombre de chiffres dans l'écriture binaire de n, donc de l'ordre de $\log_2(n)$. Ce coût est nettement inférieur à celui d'une recherche linéaire. On dit que la complexité est *logarithmique*.

Avant l'entrée dans la boucle, `g` a pour valeur 0 et `d` a pour valeur `len(liste)`. Si $x <$ `liste[0]`, ou si $x >$ `liste[d-1]`, alors x n'est pas dans la liste et la recherche n'aboutira pas.

Il est possible de vérifier avant de commencer la recherche que le nombre cherché est bien compris entre les deux bornes de la liste. On utilise pour cela une *assertion* qui arrête le programme si la condition à vérifier n'est pas remplie. Ce procédé est développé au chapitre 9.

La syntaxe est `assert liste[g] <= x <= liste[d-1]`, à placer avant la boucle.

Si l'assertion est vérifiée, alors `liste[g]` $\leq x \leq$ `liste[d-1]` avant l'entrée dans la boucle.

Preuve de la correction

Nous ajoutons un élément en fin de liste, strictement supérieur au dernier élément, par exemple égal à `liste[d-1]+1`. Nous allons démontrer que la propriété `liste[g]` $\leq x <$ `liste[d]` qui est vraie avant l'entrée dans la boucle reste encore vraie après chaque passage dans la boucle. Cette propriété est *un invariant de la boucle*.

La propriété est vraie avant l'entrée dans la boucle d'après ce qui précède.

Supposons qu'elle soit vraie avant un passage dans la boucle : `liste[g]` $\leq x <$ `liste[d]`.

D'après le choix de `k`, `liste[g]` $\leq$ `liste[k]` $\leq$ `liste[d]` puisque la liste est triée.

Donc, si $x <$ `liste[k]`, on obtient `liste[g]` $\leq x <$ `liste[k]`. Dans ce cas la nouvelle valeur de `d` est `k`, et donc `liste[g]` $\leq x <$ `liste[d]` après le passage dans la boucle.

Sinon, `liste[k]` $\leq x$, et on obtient `liste[k]` $\leq x <$ `liste[d]`. Dans ce cas la nouvelle valeur de `g` est `k`, et donc `liste[g]` $\leq x <$ `liste[d]` après le passage dans la boucle.

Il reste à examiner l'état des variables après la sortie de la boucle.

Lorsque le dernier passage dans la boucle est effectué, `d-g` ≥ 2. Donc `g` $<$ `k` $<$ `d`. Si `d` prend la valeur `k` ou si `g` prend la valeur `k`, alors `d-g` > 0. Mais s'il n'y a plus de passage dans la boucle, cela signifie que `d-g` ≤ 1. La seule possibilité est donc `d-g` $= 1$.

Or `liste[g]` $\leq x <$ `liste[d]`, donc `liste[g]` $\leq x <$ `liste[g+1]`. En conclusion, nous n'avons que deux possibilités : soit x est la valeur de `liste[g]`, soit x n'est pas dans la liste.

Remarques

1. Il est intéressant de tester la fonction avec plusieurs types de listes. Une liste contenant un seul élément, deux éléments, un nombre pair d'éléments et un nombre impair d'éléments avec une valeur cherchée présente ou non dans chaque liste.
2. Le principe de dichotomie s'applique en mathématiques pour trouver une solution approchée d'une équation du type $f(x) = 0$ sur un intervalle $[a; b]$ à epsilon près. La fonction f est supposée monotone continue.

```
def solution(f, a, b, epsilon):
    while b - a > epsilon:
        m = (a + b) / 2
        if f(a) * f(m) <= 0:
            b = m
        else:
            a = m
    return (a + b) /2
```

À chaque étape, l'amplitude de l'intervalle contenant la solution est divisée par deux. Ceci signifie que la précision gagne un chiffre dans l'écriture binaire. (10 chiffres dans l'écriture binaire correspondent à environ 3 chiffres dans l'écriture décimale).

Deuxième version

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