TP n°5 : algorithmes dichotomiques

1 Recherche d'un élément dans un tableau

On considère une liste nommée `t` de nombres (ou de chaînes de caractères). On voudrait écrire un algorithme qui détermine si une valeur donnée `x` est un élément de cette liste.

1.1 Recherche séquentielle dans un tableau

On regarde si le premier élément de la liste est égal à `x`, sinon, on passe au deuxième...

1. Écrire une fonction booléenne `est_dans_liste(x: int, t: [float])-> bool` qui prend en argument un entier x et un tableau t d'entiers et teste l'appartenance de x à t.

   Remarque : l'instruction `est_dans_liste(x, t)` est à peu près équivalente à l'instruction `x in t` de Python.

2. Donner dans le pire des cas, le nombre de comparaisons. Comment qualifieriez-vous la complexité temporelle de cette fonction ?

1.2 Recherche dichotomique pour un tableau trié

On suppose cette fois-ci que l'on dispose d'un tableau trié. Nous allons voir dans ce cas une méthode de recherche beaucoup plus efficace.

Il y a une situation concrète, dans laquelle nous sommes emmenés à chercher un mot dans une liste triée : la recherche d'un mot dans un dictionnaire (comme le petit Larousse). Il est clair que nous ne cherchons pas le mot comme précédemment. Nous pouvons utiliser une stratégie dite de dichotomie, «l'art de diviser en deux».

Prenons par exemple $t = [2, 4, 7, 12, 15, 20, 25, 28, 31, 36]$ avec $x = 25$.

On prend l'élément médian (ou juste inférieur) du tableau, c'est-à-dire 15 . Notre nombre $x = 25$ est supérieur à 15, donc si x est présent, il se trouve dans le sous-tableau à droite de 15, c'est-à-dire dans $[20, 25, 28, 31, 36]$. On recommence, 28 est le milieu du sous-tableau, mais $25 < 28$ donc 25 s'il est présent est dans le sous-tableau à gauche de 28, c'est-à-dire dans $[20, 25]$. On continue ce procédé, jusqu'à ce que l'on tombe sur 25, ou jusqu'à ce que l'on tombe sur un tableau vide, auquel cas, la valeur x est absente du tableau.

3. Écrire en pseudo-code (en français) une fonction `recherche_dichotomie` qui prend en argument un nombre x et un tableau t et

   et renvoie vrai si l'élément x appartient à t. On pourra démarrer ainsi :

```
Algo: recherche_dichotomie
Données: x un nombre et t une liste triée
Résultat: vrai si x est un élément de t et faux sinon
n = longueur de t
```

```
a = 0  # indice de la borne  gauche du tableau
b = n-1 # indice de la borne droite
Tant que l'intervalle d'indice [a,b] contient au moins une valeur
    c = indice médian entre a et b, arrondi par défaut
    si t[c] = ...
        on renvoie True  # l'élément x a été trouvé
    sinon si ....
         a = c + 1
    sinon
        ....
# Arrivé ici, on est sûr que ....
on renvoie ....
```

4. Implémenter votre fonction en Python et tester là avec le tableau précédent et les valeurs $x = 25$ et $x = 24$. Faîtes d'autres tests en prenant pour x des valeurs extrêmes comme le premier ou le dernier élément du tableau, puis faire des tests avec des tableaux de longueur 1 ou 2.

5. Si l'on admet que la quantité $b - a$ est au moins divisée par 2 à chaque itération (est-ce le cas de votre algorithme ?), combien faudra-t-il d'itérations au pire pour une liste de longueur 2^p ? Commenter.

1.3 Recherche d'un élément dans un dictionnaire

Nous avons vu dans le dernier TP une nouvelle structure de données appelée dictionnaire qui permet aussi de stocker des objets, avec pour spécificité que les index des objets ne sont pas forcément des entiers, on parle alors de clé. Par exemple dans le dictionnaire `dico = {'vache':500, 'brebis':10, 'cochon':100 }` qui contient 3 éléments, l'élement `'vache':500` a pour clé `'vache'` et pour valeur 500 . On peut tester si une **clé** (attention et pas une valeur) est présente dans un dictionnaire avec l'opérateur `in`.

```
>>> 'vache' in dico
True
>>> 10 in dico
False
```

Cette opération se fait dans un temps **constant** (en moyenne), donc ne dépend pas de la taille du dictionnaire (c'est une conséquence des tables de hashage, notion hors-programme). C'est un des intérêts fondamentaux du dictionnaire.

On se propose de vérifier expérimentalement ceci.

6. Écrire une fonction `liste_to_dico(t)` qui transforme une liste `t` de nombres en un dictionnaire. Le nombre `t[k]` sera stocké dans le dictionnaire avec une clé égale à l'indice k.

7. Exécuter le code suivant et commenter :

```
def temps_liste(taille_liste):
    n = taille_liste
    t = [0 for k in range(n)] # liste de zéros

    # on teste 1000 fois si 1 appartient à la liste t et on chronomètre
    t0 = time.time()
    for _ in range(1000):
        est_dans_liste(1, t)
    t1 = time.time()

    delta = t1 -t0
    print(f"temps de calcul en ms: {int(1000*delta)}")

for k in range(2,7):
    temps_liste(10**k)
```

8. Exécuter le code suivant et commenter :

```
def temps_dico(taille_dico):
    n = taille_dico
    t = [0 for k in range(n)]
    dico = liste_to_dico(t)

    # on teste 1000 fois si 1 appartient au dico et on chronomètre
    t0 = time.time()
    for _ in range(1000000):
        'toto' in dico  # la clé 'toto' n'est pas dans le dico
    t1 = time.time()

    delta = t1 -t0
    print(f"temps de calcul en microsecondes: {int(10**6*delta)}")


for k in range(2,7):
    temps_dico(10**k)
```

Pour comprendre comment fonctionne un dictionnaire, on peut regarder les deux vidéos suivantes sur Internet :

- Cours Python, 3.3 tables de Hash

  `https://www.youtube.com/watch?v=IhJo8sXLfVw`

- Cours Python, 3.4 dictionnaires

  `https://www.youtube.com/watch?v=VnhBoQAgIVs`

1.4 Une application des dictionnaires

Exercice 1 (Intersection de listes d'éléments deux à deux distincts)

1. Écrire une fonction `intersection(liste1: [int], liste2:[int]) -> [int]` qui prend en argument deux listes d'entiers deux à deux distincts et renvoie une liste qui contient les éléments communs aux deux listes (l'ordre des éléments n'est pas important). Par exemple si `liste1 = [4, 10, 25, 13, 21, 3]` et `liste2 = [2, 3, 50, 10, 47]`, la fonction pourrait renvoyer la liste `[10, 3]`. Dans cette question, on n'utilisera pas de dictionnaires.
2. Déterminer la complexité de cette fonction. On pourra supposer que les deux listes ont une même longueur n.
3. Comment améliorer la complexité de cet algorithme à l'aide de dictionnaires ?

2 Recherche du zéro d'une fonction par dichotomie

La fonction Python `bisect` du module `scipy.optimize` permet de résoudre numériquement des équations du type $f(x) = 0$ où f est une fonction réelle, par la méthode mathématique dite de dichotomie. On se propose ici de reprogrammer une telle fonction.

Exercice 2 (Implémentation de la dichotomie) Écrire une fonction `dichotomie` qui renvoie la valeur approchée du zéro α dans $[a, b]$ d'une fonction continue f selon la méthode de dichotomie :

- les arguments de `dichotomie` devront être : la fonction f, des réels a et b, avec $a < b$ et un réel $\varepsilon > 0$ qui peut être égal à 10^{-10} par défaut.
- le résultat renvoyé doit être une valeur approchée de α à ε près.
- on testera au préalable si $f(a)f(b) > 0$, et dans ce cas, on renverra `None`.

Exercice 3 (Utilisation)

1. On considère la fonction f définir sur $\mathbb{R}$ par $f(x) = x^7 + 23x^5 + 2x^3 - 2$. Prouver que f admet une unique racine a dans $[0, 1]$ puis déterminer une valeur approchée de a à 10^{-6}
2. Déterminer une valeur approchée à 10^{-6} près des éventuelles racines de la fonction $x \mapsto 2x^3 - 9x^2 + 12x - \frac{9}{2}$.
3. Comment utiliser la fonction `dichotomie` pour obtenir une valeur approchée du nombre $\sqrt[3]{2}$ à 10^{-4} près ?
4. Comment utiliser la fonction `dichotomie` pour obtenir une valeur approchée du nombre π ?
5. On note f la fonction définie par $f(x) = (x-2)^2 - 1$. Les racines de f sont 1 et 3. On prend $a = 0$ et $b = 4$. Réfléchir un instant, et deviner la valeur qui sera renvoyée. Vérifier.

6. On modifie notre fonction précédente em remplaçant la condition `if f(g)*f(m) <= 0: d = m` par `if f(g)*f(m) < 0: d = m`. On note f la fonction définie par $f(x) = x - 1$. On prend $a = 0$ et $b = 2$. Réfléchir un instant, et deviner la valeur qui sera renvoyée. Vérifier.

Exercice 4 (Terminaison et complexité) On souhaite maintenant prouver la terminaison et déterminer la complexité de notre algorithme. On note a_k et b_k les valeurs des variables `a` et `b` après la k-ième itération pour $k \in \mathbb{N}^*$. On pose aussi $a_0 = a$ et $b_0 = b$, qui correspondent aux valeurs des variables `a` et `b` avant la première itération.

1. Justifier que tant que l'algorithme boucle, on a $b_k - a_k = \frac{b-a}{2^k}$.
2. En déduire que l'algorithme se termine et que le nombre d'itérations p vérifie $p \leqslant \log_2(n)+1$. Comment qualifie-t-on cette complexité ?

3 Le juste prix

Exercice 5 (Le juste prix) Alice choisit un entier au hasard, par exemple entre 1 et $n = 1000$. Bob doit deviner le nombre choisi par Alice avec un minimum de réponses. Pour chaque réponse de Bob, Alice lui indique si sa réponse est trop grande, trop petite ou correcte.

Bob pense pouvoir trouver le juste prix en moins de dix réponses. Qu'en pensez-vous ?