RECHERCHE DANS UNE LISTE OU UNE CHAÎNE

1 Recherche d'un élément, complexité

On cherche à savoir si une valeur `a` est présente dans une liste `L`. La seule méthode est de regarder successivement tous les éléments de `L` et de voir si l'un d'eux est égal à `a`. Dès que c'est le cas, on arrête de chercher et on renvoie `True` (Vrai). Sinon, on renvoie `False` (Faux). Les valeurs `True` et `False` sont ce qu'on appelle des *booléens*.

Exemple. On donne deux exemples : l'un avec une boucle `while`, l'autre avec `for`.

```python
def rech_while(L,a) :
    n = len(L)
    i = 0
    while i<n :
        if L[i] == a :
            return True
        i = i+1
    return False
```

```python
def rech_for(L,a) :
    n = len(L)
    for i in range(n) :
        if L[i] == a :
            return True
    return False
```

Question 1. Recopier les scripts ci-dessus. Les tester et ajouter des commentaires pour expliquer leur fonctionnement. Que peut-on en déduire pour la commande `return` ?

Question 2. Tester les scripts pour chercher un caractère dans une chaîne de caractère, par exemple `'c'` dans `'abc'`.

Définition (Complexité)

On appelle *opération élémentaire* une des opérations suivantes :

- des comparaisons, avec `< > ==`, ou encore `<= >= !=` pour $\leq$ $\geq$ $\neq$ respectivement,
- des calculs arithmétiques simples, par exemple avec `+ - * /`.

La *complexité* (ou *coût*) d'un algorithme est déterminée par le nombre d'opérations élémentaires exécutées par l'algorithme. On regarde la complexité dans le « pire cas » et on l'exprime en fonction de la « taille » des arguments de l'algorithme (cf la remarque ci-dessous).

Remarque (Complexité, pire cas, ordre)**.** Si l'élément `a` est au tout début de la liste `L`, on ne fait que très peu d'opérations élémentaires : le résultat est immédiat, et la notion de complexité n'a que peu d'intérêt. On regarde donc généralement la complexité dans le *pire cas*.

Ici, le pire cas arrive lorsque `a` n'est pas dans `L` : on fait alors `n` opérations élémentaires avec le script `for` et `3n` avec le script `while` (la condition `i < n` est évaluée à chaque tour de boucle). En pratique, peu importe que ce soit `n`, `2n`, `3n` : on dira que ces deux scripts ont une complexité d'ordre `n`.

En d'autres termes, si `L` est de grande taille (i.e. si `n` est grand), on sait que le temps d'exécution des deux algorithmes sera (quasiment) proportionnel à `n`. On parle de complexité *linéaire*.

Pour vérifier si un élément `a` est dans une liste `L`, on peut aussi écrire « `a in L` ».

2 Recherche de la position

Maintenant, si l'élément `a` se trouve dans `L`, on veut retourner la position de la première occurence de `a` dans `L`.

```python
def rech_pos(L,a) :
    for i in range(len(L)) : # on peut se passer de n !
        if L[i] == a :
            return i
    return None              # None est une valeur vide : on ne retourne rien
```

Exercice 1. Tester l'algorithme ci-dessus lorsque `L` est une liste puis une chaîne de caractères. Quelle est sa complexité ?

Exercice 2. Écrire une fonction qui retourne la position de la dernière occurence de `a`.

Exercice 3. Écrire une fonction qui retourne TOUTES les positions de `a` sous forme d'une liste. Si `a` n'est pas dans `L`, on retournera une liste vide.

3 Recherche du maximum

A présent, on suppose que `L` est une liste de nombres, dont on cherche la valeur maximale ainsi que sa position.

```
def maximum(L) :
    max = L[0]
    for v in L :
        if v > max :
            max = v
    return max
```

```
def pos_max(L) :
    pos = 0
    for i in range(len(L)) :
        if L[i] > L[pos] :
            pos = i
    return pos
```

Exercice 4. Tester et commenter la fonction `maximum`. Quelle est sa complexité ? *Note : pour obtenir le maximum d'une liste* `L`*, on peut aussi écrire* `max(L)`.

Exercice 5. Tester et commenter la fonction `pos_max`. Quelle position est retournée lorsque le maximum est présent plusieurs fois ? Comment modifier la fonction pour que la dernière position du maximum soit renvoyée ?

4 Dictionnaire et comptage d'éléments

Exemple. Un dictionnaire est une association entre des clés et des valeurs. Les clés et les valeurs peuvent être de n'importe quel type (nombres, chaînes de caractères, etc.) :

```
score = { 'Paolo': 12 , 'Tom': 7 , 'Ella': 25 }  # les clés sont Paolo, Tom et Ella

on_double = { 1:2 , 3:6 , 'bla': 'blabla' }      # les clés sont 1, 3 et bla
```

Les clés sont à gauche du « : » et les valeurs à droite. Les clés doivent être uniques, mais les valeurs peuvent être répétées.

Question 3. Recopier le dictionnaire `score`, puis tester en console les commandes suivantes :

```
score['Paolo']
score['Tom'] = 77
score['Moi'] = 'heu...'
score                       # Qu'est-ce qui a changé dans score ?

'Moi' in score
'Le prof' in score          # Qu'est-ce qui est vérifié par cette instruction ?
```

Exercice 6. Écrire une fonction qui prend en argument un mot (chaîne de caractères) et renvoie un dictionnaire dont les clés sont les caractères du mot et les valeurs associées sont le nombre d'occurences du caractère dans ce mot.

5 Exercices d'approfondissement

Exercice 7. Écrire une fonction qui renvoie TOUTES les positions du maximum d'une liste.

Exercice 8. Écrire une fonction qui renvoie la valeur du second maximum d'une liste.