RECHERCHE D'UN ÉLÉMENT DANS UNE LISTE / STRING, COMPLEXITÉ

1 Recherche d'un élément, chaine de caractères

On cherche à savoir si une valeur `a` est présente dans une liste `L`. On peut utiliser l'instruction « `a in L` », mais il faut aussi savoir le coder par soi-même. La seule méthode générale est un algorithme de first-match :

Exercice 1. Écrire une fonction `recherche` qui à une liste `L` et une valeur `a` retourne `True` si `a` est dans `L`, et retourne `False` sinon. Tester.

En Python, les chaines de caractères (ou *string* en anglais) sont délimitées par des crochets simples `'...'` ou doubles `"..."`. La syntaxe est très similaire à celle des listes : si `S` est une chaine de caractères, `S[0]` est le premier caractère, `S[-1]` le dernier, on peut faire du slicing avec `S`, on peut concaténer deux chaines `S1` et `S2` avec `S1 + S2`, etc.

Question 1. Que va retourner `recherche("AZERTYUIOP","Z")` ? Que va retourner `recherche("AZERTYUIOP","o")` ?

- Attention, les *méthodes* associées aux listes ne fonctionnent pas sur des chaines de caractères : les instructions `S.append("Q")` ou encore `S.reverse()`entrainent une erreur.
- De plus, les chaines sont *immuables* : on ne peut donc pas les modifier une fois créées (par exemple avec `S[0]="Q"`). Par contre, on peut toujours créer une nouvelle chaine par concaténations successives.

Exercice 2. Écrire une fonction `espace` qui à une chaine de caractères retourne la même chaine mais avec un espace entre chaque caractère. Par exemple `espace("Leon")` doit retourner `"L e o n"`.

2 Recherche de la position

On souhaite écrire une fonction `firstPos` qui prend en argument une liste `L` et un élément `a` : si `a` est dans `L`, on retourne la position de la première occurence de `a` dans `L`. Sinon, on ne retourne rien (`None`).

Une méthode native sur les listes permet de réaliser cette opération : il s'agit de `L.index(a)`. Cependant, si `a` n'est pas dans `L`, cette instruction cause une erreur. De plus, il faut aussi savoir le coder par soi-même.

Question 2. Un élève écrit la fonction suivante, mais il a fait plusieurs erreurs. Trouvez-les, puis écrivez une fonction `firstPos` corrigée qui réalise l'opération demandée. Tester.

```
def firstPos(L,a) :
    for i in range(n) :
        if L[i] == a :
            print(i)
        return None
```

Question 3. À votre avis, la fonction `firstPos` fonctionne-t-elle si `L` est une chaine de caractères et que `a` est un caractère ? Essayer de répondre avant de tester.

Exercice 3. Écrire une fonction `lastPos` qui à une liste `L` et un élément `a` retourne la position de la *dernière* occurence de `a` dans `L`. Si `a` n'est pas dans `L`, alors on ne retourne rien.

3 Recherche du maximum

A présent, on suppose que `L` est une liste de nombres, dont on cherche la valeur maximale. On peut utiliser l'instruction `max(L)`, mais il faut aussi savoir le coder par soi-même.

Exercice 4. Écrire une fonction `maximum` qui à une liste de flottants retourne la valeur maximale de cette liste.

On souhaite maintenant savoir à quelle(s) position(s) dans `L` est-ce que le maximum est atteint. Toute la difficulté est que ce maximum peut être atteint en plusieurs positions.

Question 4. Que retourne `L.index(max(L))` ?

Exercice 5. Compléter la fonction suivante.

```
def maxPos(L) :  # retourne une position du maximum d'une liste L

    pos = 0  # contiendra une position du maximum de L
    for i in range(len(L)) :  # on boucle sur les indices des éléments de L
        if ... > ... :
            pos = i
    return pos
```

Question 5. Quelle position est retournée lorsque le maximum est présent plusieurs fois ?

Exercice 6. Écrire une fonction `maxLastPos` qui sera une copie légèrement modifiée de `maxPos`, de sorte qu'elle retourne la *dernière* position du maximum lorsqu'il est présent plusieurs fois.

4 Opération élémentaire

Sur le site, tout en bas, téléchargez le fichier `(...) complexite.py` et ouvrez-le. On y trouve les deux fonctions suivantes, qui retournent toutes les deux la position du maximum d'une liste `L`.

```
def maxPos1(L) :
    for i in range(len(L)) :
        if L[i] == max(L) :
            return i
```

```
def maxPos2(L) :
    m = max(L)
    for i in range(len(L)) :
        if L[i] == m :
            return i
```

Dans la cellule suivante, on importe le module `time`. On verra ultérieurement ce que cela signifie : pour le moment, il suffit de se dire que cela nous donne accès à la fonction `time.time()` qui retourne le nombre de secondes écoulées depuis l'Epoch, qui dans Python correspond au 1er janvier 1970. Ainsi, on peut mesurer un temps d'exécution en faisant la différence de deux temps mesurés :

```
t1 = time.time()
...
... # une suite d'instructions
...
t2 = time.time()

print(t2-t1) # temps qu'a mis l'ordinateur pour exécuter les instructions
```

Les cellules qui suivent servent à mesurer le temps d'exécution de `maxPos1` et de `maxPos2` pour des listes `L` « typiques ».

Question 6. Exécuter le code pour mesurer le temps d'exécution. Au vu des résultats, laquelle des deux fonctions vous semble la plus rapide ? Pourquoi, à votre avis ?

Réponse *(mieux vaut écrire au crayon à papier en attendant la correction)* :

On comprend ainsi que, **pour une même opération à réaliser** (comme trouver la première position du maximum d'une liste), **tous les algorithmes ne se valent pas**. Certains mettent par exemple plus de temps que d'autres à s'exécuter pour un même argument. Or, il va de soi qu'on préfère un algorithme le plus rapide possible.

Il convient ainsi de se donner un outil qui permette d'estimer le temps de calcul nécessaire pour qu'un algorithme s'exécute. Il s'agit de la complexité (temporelle).

Définition 4.1 (Opération élémentaire)

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 définition ci-dessus n'est en fait pas universelle. Selon les situations, on peut inclure également dans les opérations élémentaires (l'énoncé vous le précisera) :

- Les opérations d'affectation : `a = 4`
- Les instructions qui récupèrent un élément d'une liste : `L[2]`
- L'exécution d'une fonction, comme l'instruction `print(i)` ou `len(L)`, peut également constituer une opération élémentaire, voire plusieurs.
- etc.

Les opérations élémentaires sont concrètement des opérations très simples qui mettront un temps minimal à être exécutées (on ne peut pas faire plus court). Un ordinateur avec un processeur avec une fréquence de 1 GHz ($= 10^9$ Hertz) pourra faire, *grosso modo*, 10^9 opérations élémentaires par seconde.

Question 7. On définit une opération élémentaire suivant la définition 4.1. De plus, on considère que si `L` est une liste de taille n, alors l'instruction `max(L)` compte pour n opérations élémentaires et `len(L)` pour une seule. Si `L` est une liste de taille n, quel est le nombre d'opérations élémentaires réalisées par `maxPos1(L)` et `maxPos2(L)` ?

Réponse pour `maxPos1` :

Réponse pour `maxPos2` :

On comprend ainsi pourquoi, lorsque la taille n de la liste `L` est très grande, ces deux algorithmes n'ont pas le même temps de calcul.

5 Complexité

Dans certains cas, il peut arriver que même un argument de grande taille ne conduit pas à un temps de calcul élevé :

Question 8. Dans `(...) complexite.py`, ajouter l'instruction `L = L[::-1]` juste en-dessous des deux lignes qui définissent `L` (20 et 30). Compiler. Que remarque-t-on ? Comment expliquer cela ?

Commenter les lignes `L = L[::-1]` que vous venez d'ajouter. On fera la suite de ce TP sans ces lignes.

Définition 4.2 (Complexité)

La complexité (temporelle) (ou coût) d'un algorithme est déterminée par le nombre d'opérations élémentaires exécutées par l'algorithme.

- On l'exprime selon la « taille » du ou des arguments-conteneurs (liste, string, tuples, etc.) de l'algorithme, généralement notées n, m, ...
- On l'exprime sous la forme d'un « ordre » de grandeur en puissance de n (et m, ... le cas échéant).
- On regarde la complexité dans le « pire cas » : à taille fixée, on choisit l'argument qui conduira au plus grand nombre d'opérations élémentaires.

Complexité, pire cas. Si le maximum est au tout début de la liste `L`, on a vu qu'on ne fait que très peu d'opérations élémentaires : le temps d'exécution est très court. Il s'agit d'une situation exceptionnelle qui n'est pas représentative du cas général. Le pire cas peut ne pas être représentatif non plus, mais il a le mérite de donner une idée du temps d'exécution maximal d'un algorithme. Dans notre exemple, le pire cas arrive lorsque le maximum est à la dernière position dans `L`.

Complexité, taille et opération élémentaire. *Une fois le pire cas déterminé,* on se fixe une taille générique pour chaque argument-conteneur, en général notées n, m, ... et on compte le nombre d'opérations élémentaires nécessaires. C'est ce que l'on a fait à la question 7, car les listes `L` sont triées par ordre croissant. On a ainsi trouvé que `maxPos1` réalise n^2+n+1 opérations élémentaires tandis que `maxPos2` en réalise $2n+1$.

Complexité, ordre. Une fois calculé le nombre d'opérations élémentaires dans le pire cas, on ne regarde qu'un ordre de grandeur quand n est grand : on ne conserve que la plus grande puissance de n, et on ignore tout coefficient multiplicatif.

- `maxPos1` réalise n^2+n+1 opérations : on dit qu'il a une complexité d'ordre n^2, i.e. une complexité quadratique.
- `maxPos2` réalise $2n+1$ opérations : on dit qu'il a une complexité d'ordre n, i.e. une complexité linéaire.

En d'autres termes, si `L` est une liste de taille $n \ggg 1$, le temps d'exécution de `maxPos2` sera (à peu près) proportionnel à n, tandis que le temps d'exécution de `maxPos1` sera (à peu près) proportionnel à n^2.

Question 9. Assurez-vous d'avoir bien commenté les instructions `L = L[::-1]` écrites précédemment. Compiler pour avoir les temps d'exécution. Doubler la taille de chaque liste `L` puis compiler à nouveau. Que remarque-t-on ?

6 Exercices d'approfondissement

Exercice 7. Écrire une fonction `palindrome` qui teste si une chaine de caractères est un palindrome, c'est-à-dire que si on la lit de droite à gauche on obtient exactement la même chaine que si on la lit de gauche à droite (par exemple le mot « radar » ou la chaine « nez zen »).

Exercice 8. Écrire une fonction `allPos` qui à une liste `L` et à une valeur `a` retourne TOUTES les positions de `a` dans `L` sous forme d'une liste. Si `a` n'est pas dans `L`, on retournera une liste vide.

Exercice 9. Écrire une fonction `doublon` qui prend en argument une chaine de caractères `S` et un caractère `c`, et qui renvoie `True` si ce caractère est présent au moins deux fois dans la chaine, et renvoie `False` sinon.

Exercice 10. Écrire une fonction `allMax` qui renvoie TOUTES les positions du maximum d'une liste de flottants.

Exercice 11. Écrire une fonction `secondMax` qui renvoie la valeur du second maximum d'une liste, i.e. la plus grande valeur strictement inférieure au maximum de la liste. S'il n'y en a pas, on ne retourne rien.

Exercice 12. On reprend la fonction `recherche` de l'exercice 1. Déterminer sa complexité : on commencera donc par déterminer le (ou un) pire cas, puis on calcule le nombre d'opérations élémentaires, et enfin on donne l'ordre de la complexité.

Exercice 13. Même exercice que le précédent avec la fonction `maxPos` de l'exercice 5.

Exercice 14. À la fin du fichier `(...) complexite.py`, écrire une suite d'instruction qui permet de mesurer le temps d'exécution de l'instruction `p = L.index(max(L))`. Comparer avec `maxPos2` pour une liste de taille 10^7. Laquelle est la plus rapide ? Tester encore avec une liste de taille 2×10^7 puis 4×10^7. Compte tenu de ces résultats, quelle serait la complexité de l'instruction `p = L.index(max(L))` ?

Exercice 15 ($\star$)**.** Un mauvais génie vous impose d'écrire une fonction `minimum` qui renvoie le minimum d'une liste de flottants. Mais ce génie est sournois : il vous interdit de faire une boucle ni d'utiliser la fonction `min`. Avec un sourire narquois, il vous autorise toutefois à utiliser la fonction `max`.

Saurez-vous écrire cette fonction `minimum` pour le renvoyer définitivement dans sa lampe ?