BOUCLES IMBRIQUÉES, COMPLEXITÉ BIS

1 Tableaux 2D

Une liste peut contenir des objets de différents types : des nombres, des *string*, des booléens... Et on peut même les mélanger, par exemple `L = [ 1 , 'xyz' , True ]`. On peut aller plus loin : une liste peut contenir... d'autres listes.

```
T = [ [1,2,3] , [4,5] , [6,7,8] ]
```

On appelle cela un *tableau à deux dimensions (2D)*. C'est parfois plus lisible sous cette forme :

```
T=[ [1, 2, 3],
    [4, 5   ],  # Attention cette ligne ne contient que deux éléments !
    [6, 7, 8] ]
```

`T` n'est donc rien d'autre qu'une liste qui contient d'autres listes. Tester en console les instructions suivantes :

- `T[0]`
- `T[0][1]`
- `T[1][0]`
- `len(T)`
- `len(T[1])`
- `T[1:]`
- `T[3]` (erreur !)
- `T[1][2]` (erreur !)
- `T[1,1]` (erreur !)

Enfin, on peut mettre une boucle (`for` ou `while`) à l'intérieur d'une autre boucle. Tester le code suivant :

```
for i in range(4) :
    for j in range(3) :
        print('i=',i,' j=',j,' i+j=',i+j)
```

Exercice 1. A partir de la fonction `max` qui retourne le maximum d'une liste (cf TP2), écrire une fonction `max2D` qui prend en argument un tableau 2D et retourne la valeur maximale de ce tableau. Tester avec le tableau `T`.

2 Recherche des deux valeurs les plus proches

On considère une liste de nombres, et on cherche les deux valeurs les plus proches dans cette liste. Par exemple, si

`L = [ -1, 11, 6, 0, -5, 3 ]`

alors on souhaite retourner le couple `(-1,0)`, ou alors `(0,-1)`. Rappel : si $a,b \in \mathbb{R}$, la distance de a à b est donnée par $|a-b|$, ce qu'on peut coder par `abs(a-b)`.

```
def plusProchesValeurs(L):
    dist = abs(L[0]-L[1]) # distance entre L[0] et L[1]
    couple = ...  # couple correspondant à cette distance
    n = len(L)
    for i in range(...):
        for j in range(...):
            # Calculer la distance entre les valeurs des indices i et j. Si
              cette distance est plus petite que la variable dist, alors on
              met à jour dist et couple.
    return couple
```

Exercice 2. Compléter la fonction `plusProchesValeurs` ci-dessus. Tester.

On s'intéresse à la complexité de la fonction `plusProchesValeurs`.

Question 1. On considère que les fonctions `abs` et `len` constituent chacune une opération élémentaire. Quelles sont toutes les opérations élémentaires de la fonction `plusProchesValeurs` ? Entourez-les.

Question 2. Combien d'itérations sont réalisées par la fonction `plusProchesValeurs` ? Par itération on entend une valeur du couple (i, j), ou de manière équivalente une exécution du bloc à partir de la ligne 7.

Question 3. Est-ce qu'il y a un pire cas ? Si oui, lequel ?

Question 4. Déterminer le nombre d'opérations effectuées dans ce pire cas, qu'on note $C(n)$. Montrer que cette fonction a une complexité d'ordre n^2, où n est la longueur de la liste. On parle de coût (ou complexité) quadratique.

3 Recherche d'un mot dans un texte

On dispose de deux chaînes de caractères : un texte (long, avec n caractères) et un mot (court, avec m caractères). On supposera bien sûr $m \leq n$. On cherche la présence du mot complet dans le texte. Contrairement au TP précédent, on ne cherche pas une lettre unique mais un mot complet dans la chaîne :

- On parcourt chaque lettre du texte, qu'on repère par l'indice i. À partir de la lettre d'indice i du texte, on vérifie lettre par lettre si le mot se trouve dans le texte à partir de l'indice i. Il y a donc m vérifications à faire, un par lettre.
- Si (ne serait-ce que) une des lettres ne correspond pas, on passe à la lettre d'indice $i+1$ dans le texte et on recommence la vérification ci-dessus.
- Il faut faire attention à ne pas « sortir » du texte en allant trop loin : si on cherche un mot de longeur $m = 3$ à partir de la lettre $n-1$ du texte, cela risque de causer une erreur de dépassement.

C'est un algorithme essentiel et son implémentation nécessite une certaine attention.

```
1  def rechMot(texte,mot):
2      n = len(texte)
3      m = len(mot)
4      for i in range(...):   # on regarde le texte à partir de la lettre i
5
6          compteur = 0  # compte le nombre de lettres correctes à partir de i
7          for j in range(m):
8              if ... :
9                  compteur = compteur + 1
10         if ... :
11             return True  # le mot a été trouvé
12     return False
```

Exercice 3. Compléter la fonction ci-dessus, et la tester.

Note : l'opérateur `in` permet également de réaliser cette recherche, avec l'instruction `mot in texte`. Par exemple l'instruction `"mal" in "amalgame"` retournera `True`.

On s'intéresse à la complexité de la fonction `rechMot`. Il y a deux arguments-conteneurs : `texte` de longeur n et `texte` de longueur m, donc la complexité va dépendre de n ET m.

Question 5. On considère que la fonction `len` constitue une opération élémentaire. Quelles sont toutes les opérations élémentaires de la fonction `rechMot` ? Entourez-les.

Question 6. Combien d'itérations sont réalisées par la fonction `rechMot` ? Par itération on entend une valeur du couple (i, j).

On note $C(n,m)$ le nombre d'opérations effectuées dans le pire cas. Ce pire cas n'est pas évident à trouver : à partir de la lettre d'indice i, chaque lettre correcte donne une opération élémentaire en plus. Mais si toutes les lettres sont correctes, alors on ne passera pas à l'itération de l'indice $i+1$.

Dans les faits, trouver exactement les arguments qui mènent au pire cas n'est pas toujours nécessaire. Il suffit parfois de minorer et majorer $C(n,m)$ assez finement pour que, quand on regarde *l'ordre* de la complexité, le résultat revient au même.

Question 7 (Minoration par un « moins pire » cas)**.** On suppose qu'aucune lettre du mot se trouve dans le texte, par exemple

`texte =` $\underbrace{\texttt{"aaaa...a"}}_{n \text{ lettres}}$ `mot =` $\underbrace{\texttt{"bb...b"}}_{m \text{ lettres}}$

Quel est alors le nombre d'opérations élémentaires ? En déduire une minoration de $C(n,m)$.

Question 8 (Majoration par un « plus pire » cas théorique)**.** Donner une majoration simple de $C(n,m)$. On pourra considérer par exemple que

`texte =` $\underbrace{\texttt{"aaaa...a"}}_{n \text{ lettres}}$ `mot =` $\underbrace{\texttt{"aa...a"}}_{m \text{ lettres}}$

MAIS, bien que toutes les lettres soient correctes, on ne réalise pas pour autant l'instruction `return True` (d'où l'aspect théorique).

Question 9. En déduire la complexité de `rechMot`.

4 Exercices d'approfondissement

Exercice 4. On reprend la fonction `rechMot`. Quelle est sa complexité (en n) si on cherche un mot de longueur 1 ? De longueur n ? De longueur $\frac{n}{2}$?

Exercice 5. Écrire une fonction `valeurCommune` qui prend en argument deux listes (de tailles éventuellement distinctes n et m) et qui retourne `True` s'il y a une valeur commune dans ces listes, et `False` sinon. Quelle est sa complexité ?

Exercice 6. Écrire une fonction `valeurCommune2D` qui prend en argument deux tableaux 2D de même taille $n \times n$ (i.e. n lignes et n colonnes), et qui retourne `True` s'il y a une valeur commune, et `False` sinon. Quelle est sa complexité ?

Exercice 7. Écrire une fonction `rechPosMot` qui prend en argument un texte T et un mot M et renvoie dans une liste tous les indices qui correspondent aux débuts du mot M dans T. Par exemple avec

$T =$ `"KwaKwaKwa"` $\qquad M =$ `"Kwa"`

la liste retournée devra être `[0,3,6]`.

Exercice 8. La fonction `rechMot` peut être améliorée. En effet, si on trouve qu'une lettre du mot ne correspond pas, il est inutile de regarder si les lettres suivantes du mot correspondent. Remplacer la boucle `for` sur `j` par une boucle `while` avec deux conditions :

```
1 while ... and ...
```

de sorte que dès qu'une lettre du mot ne correspond pas, la boucle `while` s'arrête.

Exercice 9 ($\star$)**.** Déterminer la complexité de la fonction `rechMot` améliorée par l'exemple précédent. On pourra supposer que m divise n pour simplifier (avec n la longueur du texte et m la longueur du mot).

Comme vu en TP, on fera une majoration et minoration du coût $C(n,m)$.

Exercice 10. On se place dans le plan, où un point est repéré par un couple, càd un tuple à deux éléments `(x,y)` qui correspond à ses coordonnées. On se donne une série de points qu'on stocke dans une liste

```
L = [ (-2,1), (1,2), (4,1), (2,1), (4,3) ]
```

L'objectif est de déterminer la paire de points de `L` dont la distance est minimale.

On définit une fonction « distance » qui prend en argument deux couples et renvoie la distance entre les points correspondants :

```
def distance(A,B) :
    xA,yA = A            # Équivaut à xA = A[0] et yA = A[1]
    xB,yB = B
    return ( (xB-xA)**2 + (yB-yA)**2 )**0.5
```

Écrire une fonction `pointsProches` qui prend en argument une liste de couples et retourne l'indice des couples les plus proches (pour la fonction `distance`) dans cette liste. Par exemple pour la liste `L` ci-dessus, la fonction doit retourner `[1,3]`.

Exercice 11. On suppose qu'on est dans une vigne horizontale : se déplacer dans la direction x est deux fois plus facile que dans la direction y, et donc (par exemple) `distance( (0,0) , (1,0) )` est deux fois plus petite que `distance( (0,0) , (0,1) )`. Modifier la fonction `distance` pour refléter cela. Que retourne la fonction `pointsProches` appliquée à la liste `L` dans ce cas ?