BOUCLES IMBRIQUÉES

1 Tableaux 2D

Dans une liste on peut mettre ce qu'on veut : des nombres, des chaînes de caractères, des booléens... Et on peut même les mélanger, par exemple `L = [ 1 , 'xyz' , True ]`. On peut aller plus loin encore : une liste peut contenir... d'autres listes.

```
T = [ [2,4,1] , [3,11] , [7,1,5] ]
```

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

```
T=[ [2,4  ,1],
    [3,11  ],    # Attention cette ligne ne contient que deux éléments !
    [7,1  ,5]]
```

La syntaxe est la même que celle des listes : compiler le code ci-dessus et tester en console les instructions `T[0]` puis `T[0][1]` et `T[1][0]`. Attention à l'ordre !

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(L)` qui retourne le maximum d'une liste `L` (cf TP2), écrire une fonction qui détermine le maximum d'un tableau 2D. Tester avec le tableau `T` défini plus haut.

2 Recherche des deux valeurs les plus proches

On se place dans le plan, où un point est repéré par un *couple* `(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), (2,1), (4,1), (4,3) ]
```

Un couple est presque comme une liste à deux éléments. La syntaxe est identique : si `a=(-2,1)` alors `a[0]` renvoie `-2`, etc. La seule différence est qu'une fois créé, on ne peut plus modifier le couple (changer les valeurs, en ajouter ou en supprimer).

On se donne également 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
```

L'objectif est de déterminer la paire de points de `L` dont la distance est minimale. Pour cela, une méthode simple est de comparer toutes les paires possibles. Si on a n points dans `L`, il y a donc $\frac{n(n-1)}{2}$ paires à tester.

Exercice 2. Compléter la fonction suivante, puis la tester avec `L`. Faites un dessin.

```
def plus_proches(L,distance):
    n = len(L)
    paire = [0,1]                       # une première paire de points
    d = distance(L[0],L[1])             # la distance associée

    for i in range(...):                # Compléter les ...
        for j in range(...):            # Compléter les ...
            dij = distance(L[i],L[j])
            if dij < d :
                paire = [i,j]
                d = dij
    return paire
```

On notera que la fonction `plus_proches` prend en argument la fonction `distance` ! Une fonction peut donc être un argument d'une autre fonction.

Exercice 3. Montrer que l'algorithme ci-dessus a une complexité d'ordre n^2. 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 cherche la présence du mot complet dans le texte. Contrairement au TP précédent, on ne cherche pas une valeur mais une suite de valeurs dans la chaîne :

- Il faut chercher la première lettre du mot, puis la seconde, etc.
- Il faut commencer par la première lettre du texte, puis la seconde, etc.
- Il faut faire attention à ne pas « sortir » du texte en allant trop loin : pas besoin de chercher un mot de longeur `m = 3` à partir de la lettre `n-1`.

C'est un algorithme essentiel et son implémentation nécessite une certaine rigueur. On souhaite renvoyer la position de la première lettre de la première occurence du mot.

```
def rech_mot(texte,mot):
    n = len(texte)
    m = len(mot)
    for i in range(n-m+1):    # on regarde le texte à partir de la lettre i
        j = 0
        while j < m and mot[j]==texte[i+j]:    # on teste la j-ième lettre du mot
            j = j + 1
        if ... :              # Compléter les ...
            return i
    return None
```

Exercice 4. Compléter la fonction ci-dessus, et la tester.

Exercice 5. Écrire une fonction `rech_toutes_pos_mot` qui renvoie toutes les positions de départ d'un mot dans un texte sous forme d'une liste. Si le mot n'est pas présent, on retournera une liste vide.

4 Exercices d'approfondissement

Exercice 6. Quand est-ce que le pire cas arrive dans la fonction `rech_toutes_pos_mot` ? En déduire un ordre de grandeur de la complexité en fonction de `n` et `m`. A-t-on une complexité linéaire ou quadratique si $m = 1$? Si $m = n$? Si n est pair et $m = \frac{n}{2}$?

Exercice 7. 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) , (0,1) )` est deux fois plus grande que `distance( (0,0) , (1,0) )`. Modifier la fonction `distance` pour refléter cela. Que retourne la fonction `plus_proches` dans ce cas ?