Entrainement : Boucles imbriquées

Mots clés : *sommes doubles, recherche facteur dans un texte, valeurs les plus proches, tri à bulles*

Table des matières

I Calculs de sommes / produits

Calculer une somme

$$\sum_{k=0}^{n} \sqrt{k}$$

```python
import math # bibliothèque math

def somme(n: int) -> float:
    s = 0      # la somme au départ
               # vaut toujours 0
    for k in range(0, n+1):
        s = s + math.sqrt(k)
    return s
```

Calculer

$$\sum_{i=1}^{n} 2^i$$

```python
def somme(n: int) -> int:
```

Calculer

$$\prod_{k=1}^{n} (2+k)$$

```python
def produit(n: int) -> float:
```

Calculer

$$\sum_{k=1}^{n}\sum_{i=0}^{k} e^{-ik}$$

```
def somme_somme(n: int) -> float:
```

Calculer

$$\sum_{k=1}^{n}\prod_{\ell=0}^{k-1} \sin(k+\ell)$$

```
def somme_prod(n: int) -> float:
```

II Parcours d'un tableau à deux dimensions

Un tableau à deux dimensions peut être représenté comme une liste de listes. Par exemple :

0	0	1	0	1
−1	1	1	1	0
1	1	0	0	0

peut être représenté par `G = [ [0,0,1,0,1], [-1,1,1,1,0], [1,1,0,0,0] ]`

Dans ce cas, compléter ce que renvoient les commandes :

```
>>> G[0]
[0,0,1,0,1]
>>> G[1]

>>> G[1][1]

>>> G[0][1]
```

```
>>> G[1][0]

>>> G[1:]

>>> len(G)

>>> len(G[0])
```

✌ A RETENIR ✌ La valeur de la case (i, j) est accessible par la commande `G[i][j]`

Etant donné un tableau (à deux dimensions) comprenant des entiers, écrire un algorithme qui compte le nombre d'entiers pairs.

```
def nb_pairs(L: list) -> int:
```

```
>>> nb_pairs([ [1,3,6], [7,8,9] ])
2
```

```
>>> nb_pairs([ [1,3,5], [7,13,9] ])
0
```

Etant donné deux entiers n et m représentant respectivement le nombre de lignes et de colonnes, écrire une fonction qui renvoie la grille représentant le tableau :

1	2	3	$\cdots$	m
2	3	4	$\cdots$	$m+1$
$\vdots$	$\vdots$	$\vdots$		$\vdots$
n	$n+1$	$n+2$	$\cdots$	$n+m-1$

On pourra utiliser la fonction suivante :

```
def g_zeros(n: int, m: int) -> list:
    G = []
    for i in range(n): # chaque ligne
        L = []
        for j in range(m): # chaque col
            L.append(0)
        G.append(L)
    return G
```

```
def grille(n: int, m: int) -> list:
    G = g_zeros(n, m) # met dans G une grille
                      # n x m remplie de 0
```

```
>>> grille(2, 3)
[[1,2,3], [2,3,4]]
```

```
>>> grille(3, 2)
[[1,2], [2,3], [3,4]]
```

III Recherche d'un facteur dans un texte

Ecrire un algorithme qui renvoie `True` si `mot` est contenu dans la chaîne de caractères `texte` et `False` sinon.
On pourra procéder ainsi :

- on boucle sur les caractères du `texte`,
- si le caractère du `texte` est égal à la première lettre du `mot` alors :
  - on initialise un compteur à 1 (correspondant au nb de lettres égales)
  - on boucle sur les lettres du mot et à chaque fois qu'on a la bonne lettre on incrémente le compteur
  - si le compteur a la même valeur que le nombre de lettres de `mot` alors on renvoie `True`
- on renvoie `False` à la toute fin

```
def recherche(texte: str, mot: str) -> bool:
```

```
>>> recherche("Poséidon est un voleur", "voleurs")
False

>>> recherche("Poséidon est un voleur", "un")
True

>>> recherche("Poséidon est un voleur", "séid")
True
```

IV Recherche de deux valeurs les plus proches

Ecrire un algorithme qui renvoie la position des deux valeurs les plus proches dans une liste (ie dont l'écart en valeur absolue est minimal).
On pourra procéder ainsi :

- on initialise les deux positions de départ k, ℓ à 0 et 1
- on initialise l'écart dans une variable adaptée
- on parcourt la liste avec un indice i et :
  - pour chaque valeur `L[i]`, on calcule l'écart entre `L[i]` et `L[j]` où $j > i$
  - si l'écart obtenu est inférieur à l'écart initialisé alors on le remplace et on sauvegarde les positions (i, j) dans (k, ℓ).
- on renvoie le couple (k, ℓ).

```
def rech_val_proches(L: list) -> tuple:
```

```
>>> rech_val_proches([11, 4, 15, 5])
(1, 3)

>>> rech_val_proches([5, 3, 10, 8, 25, 18, 11, 14])
(2, 6)
```

V Tri à bulles

Sans ordinateur, décrire en une phrase l'action de la fonction suivante, puis constater le résultat à l'aide de l'ordinateur.

```
def une_remontee(tab: list) -> list:
    n = len(tab)
    for j in range(0, n-1):
        if tab[j] > tab[j+1] :
            tab[j], tab[j+1] = tab[j+1], tab[j]
    return tab
```

On pourra tester sur les deux listes suivantes :
`[9, 5, 4, 6]` et `[6, 2, 3, 9, 1]`

Ecrire un algorithme qui trie une liste de nombres en faisant remonter les plus grands à la fin.
Le principe du tri à bulles est le suivant :

- à chaque étape i, on fait remonter le plus grand élément de $L[:n-i]$ en position $n-1-i$:
  - on compare le premier élément de la liste avec son suivant et s'il est plus grand que son suivant alors on échange les deux
  - on répète cela jusqu'à la position $n-i$
- on renvoie la liste (étape facultative)

```
def tri_a_bulles(L: list) -> list:
```

```
>>> tri_a_bulles([6, 1, 4, 9, 3, 5])
[1, 3, 4, 5, 6, 9]
```