BOUCLES IMBRIQUÉES

On parle de **boucles imbriquées** lorsqu'une boucle (`for` ou `while`) se trouve dans le corps d'une autre boucle. À chaque itération de la boucle extérieure, la boucle intérieure est alors exécutée intégralement. Ce TP présente quelques exemples d'utilisation de boucles imbriquées.

1 Premiers exemples

1. On se donne la fonction suivante :

```
def f(n):
  r = 0
  for i in range(n):
    for j in range(n):
      r = r + j
  return r
```

   (a) Déterminer le résultat de l'appel `f(4)`.

   (b) Combien d'additions sont effectuées lors d'un appel `f(n)`, en fonction de n ?

   (c) Modifier la fonction `f` pour calculer le même résultat en utilisant moins d'opérations arithmétiques.

2. Écrire une fonction `somme_double` prenant en argument deux entiers positifs `n` et `m` et renvoyant $\sum_{i=0}^{n-1}\sum_{j=0}^{m-1} \ln(i+j^2+1)$.

   Vérifier que `somme_double(3,5)` renvoie `26.17033192699271`.

3. Combien de fois la fonction `log` est-elle appelée à chaque appel de `somme_double`, en fonction de `n` et `m` ?

2 Maximums et minimums

4. Écrire une fonction `maximum` prenant en argument une liste de nombres et renvoyant son maximum. Par exemple, `maximum([4, 1, 7, 2])` doit renvoyer `7`.

5. Écrire une fonction `minmax` prenant en argument une liste de listes de nombres, et renvoyant le minimum des maximums de ces listes.

   Par exemple, `minmax([[4, 1, 7, 2], [3, 1, 8], [5, 6]])` doit renvoyer `6`.

   La fonction `minmax` pourra faire appel à la fonction `maximum`. Cela reste une situation de boucles imbriquées, dans laquelle la boucle intérieure est celle de la fonction `maximum`.

6. Combien de comparaisons sont effectuées lors d'un appel à `minmax` ?

3 Recherche des deux valeurs d'une liste les plus proches

Dans cette partie, on s'intéresse à déterminer les deux éléments les plus proches dans une liste de nombres, au sens de la valeur absolue.

7. Écrire une fonction `valeurs_plus_proches` prenant en argument une liste de nombres `L`, qu'on supposera de longueur au moins 2, et renvoyant les deux valeurs les plus proches.

   Ainsi, `valeurs_plus_proches([-2, 3, 8, 5, -9]` doit renvoyer `(3, 5)`.

8. Combien de fois la fonction `abs` est-elle appelée à chaque appel de `valeurs_plus_proches`, en fonction de `n` la longueur de la liste argument ?

   Remarque : D'après le calcul précédent, le temps de calcul de la fonction a un terme dominant en n^2. On dit qu'elle est de **coût quadratique**.

4 Recherche d'un facteur dans un texte

On s'intéresse dans cette partie au problème de la recherche d'un mot dans un texte.

9. On veut pouvoir interrompre prématurément la boucle intérieure. Nous allons donc, comme dans l'exercice 2, écrire une fonction contenant cette boucle intérieure, qui pourra ensuite être appelée dans la boucle extérieure.

   Écrire une fonction `facteur_position` prenant en argument une chaîne de caractères `texte`, une chaîne de caractères `mot` et un indice `i`, et testant si `mot` apparaît dans `texte` à partir de l'indice `i`.

   Par exemple :

   `facteur_position('hello world', 'world', 6)` doit renvoyer `True`

   `facteur_position('hello world', 'world', 5)` doit renvoyer `False`

10. En utilisant la fonction précédente, écrire une fonction `facteur` prenant en argument une chaîne de caractères `texte` et une chaîne de caractères `mot`, et testant si `mot` apparaît dans `texte`.

11. Écrire une fonction `liste_occurrences` prenant en argument une chaîne de caractères `texte` et une chaîne de caractères `mot`, et renvoyant la liste des indices où commencent chaque occurrence de `mot` dans `texte`.

    Ainsi, `liste_occurrences('blablabla', 'bla')` doit renvoyer `[0,3,6]`.

12. Combien de comparaisons de caractères sont effectuées au maximum lors d'un appel de la fonction précédente, en fonction de `n` la longueur de `texte` et `m` la longueur de `mot` ?

13. En Python, on peut obtenir la **tranche**, c'est-à-dire la sous-chaîne, de l'indice `i` inclus à l'indice `j` exclus de la chaîne `s` en écrivant `s[i:j]`. Écrire des variantes de `facteur` et `liste_occurrences` utilisant les tranches plutôt que la fonction `facteur_position`.

5 Tri à bulles

Le tri à bulles est un algorithme de tri très simple dont le principe est de faire remonter à chaque étape le plus grand élément du tableau à trier, comme les bulles d'air remontent à la surface de l'eau (d'où le nom de l'algorithme).

Commençons par un exemple du fonctionnement de l'algorithme. On souhaite trier la liste de nombres : $[2,5,4,3,1]$.
Premier passage :
$[2,5,4,3,1]$, on compare 2 et 5 et on ne fait rien car $5 > 2$;
$[2,5,4,3,1]$, on compare 5 et 4 et on les échange car $5 < 4$;
$[2,4,5,3,1]$, on compare 5 et 3 et on les échange ;
$[2,4,3,5,1]$, on compare 5 et 1 et on les échange ;
$[2,4,3,1,5]$, fin du premier passage.
Deuxième passage :
$[2,4,3,1,5]$, on compare 2 et 4 et on ne fait rien ;
$[2,4,3,1,5]$, on compare 4 et 3 et on les échange ;
$[2,3,4,1,5]$, on compare 4 et 1 et on les échange ;
$[2,3,1,4,5]$, fin du deuxième passage.
Troisième passage :
$[2,3,1,4,5]$, on compare 2 et 3 et on ne fait rien ;
$[2,3,1,4,5]$, on compare 3 et 1 et on les échange ;
$[2,1,3,4,5]$, fin du troisième passage.
Quatrième passage :
$[2,1,3,4,5]$, on compare 2 et 1 et on les échange ;
$[1,2,3,4,5]$, fin du quatrième passage.

14. Appliquer « à la main » l'algorithme à la liste $[5,1,2,3,4,5]$.
15. Écrire une fonction `echanger(L,i,j)` prenant en argument une liste et échangeant ses éléments situés aux indices i et j. Cette fonction modifiera donc son argument mais ne renverra rien (une fonction qui ne renvoie rien est appelée une **procédure**).
16. Écrire une fonction `tri_bulles` implémentant cette version de l'algorithme de tri à bulles en Python et la tester en donnant en entrée une liste aléatoire de nombres entiers.
17. Déterminer, en fonction de la longueur de la liste argument, le nombre de comparaisons effectuées par cet algorithme.
18. À partir de l'exemple de la question 14, proposer une condition d'arrêt de la boucle principale permettant de gagner du temps dans les meilleurs cas.
19. Implémenter cette optimisation par une fonction `tri_bulles2`.
20. En comptant le nombre de comparaisons, qualifier le coût en temps dans le pire et dans le meilleur cas de la fonction `tri_bulles2`.
21. Une optimisation du temps dans le meilleur cas n'est pas forcément une optimisation dans le cas moyen. Comparer les temps d'exécution des fonctions `tri_bulle` et `tri_bulle2` sur une liste aléatoire de 10 000 éléments à l'aide de la fonction `time`.
22. Pour analyser la correction de l'algorithme, on commence par remarquer qu'on ne travaille que par transposition, ce qui justifie qu'à la fin on obtient une permutation du tableau initial. Pour justifier que cette permutation correspond à un tri, on utilise un **invariant de boucle** : une proposition portant sur les variables du programme et qui reste vraie après chaque passage dans la boucle principale. Identifier une telle propriété, qui permette de déduire qu'à l'issue du dernier passage dans la boucle le tableau est trié.

Aide Python :

- Pour obtenir le logarithme népérien : `log`, après avoir importé la bibliothèque `math` avec `from math import *`.
- pour obtenir la valeur absolue : `abs`.
- Pour générer une liste `L` et `n` nombres entiers aléatoires compris dans l'intervalle `[a,b[` on peut écrire `L = random.sample(range(a,b),n)`. Pour cela, il ne faut pas oublier d'importer la bibliothèque `random` en faisant `import random`.
- Pour calculer le temps d'exécution d'une fonction, on peut utiliser la fonction `time` du module `time` . Cette dernière renvoie le nombre de secondes écoulées depuis le 1 er janvier 1970 (pour les systèmes UNIX, le 1er janvier 1970 00 :00 :00 (UTC), appelé *epoch*, correspond au moment où le temps commence) et, en capturant cette valeur avant et après l'exécution d'une partie de programme, on va pouvoir connaitre le temps qu'elle a pris pour s'exécuter.