TP-2 : boucles imbriquées

1 Premiers exemples et implémentations

1. Il est parfois nécessaire de mettre en place un premier parcours séquentiel, avec une boucle, et en fixant ce premier item, de parcourir une deuxième boucle. De manière artificielle on peut faire

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

   Combien de fois la fonction print est-elle appelée dans cette boucle ?

   Dans le même esprit on peut créer un tableau rectangulaire (une matrice) avec une liste de listes grâce à deux boucles imbriquées, par exemple pour une matrice de **Vandermonde**, dont l'élément de la ligne $i \in [\![0, n-1]\!]$ et de la colonne $j \in [\![0, p-1]\!]$ est i^j, on peut faire

```
def vandermonde(n, p):
    M = []
    for i in range(n):
        L = []
        for j in range(p):
            L.append(i**j)
        M.append(L)
    return M
```

   puis, dans la console, `vandermonde(6,4)` pour voir le résultat.

   Quel est la complexité – le coût – de la fonction `vandermonde` ?

2. On peut dans certains cas n'avoir besoin que d'un tableau triangulaire, c'est-à-dire qu'une borne de l'itérateur qui permet de décrire la boucle interne (la deuxième) dépend de l'indice qui permet de décrire la première boucle. Par exemple

```
for i in range(4):
    for j in range(i):
        print(i, j)
```

```
def tableau_triangulaire(n):
    #crée un tableau triangulaire de taille n avec m[i][j]=(i+1)*(j+1)
    M = []
    for i in range(n):
        L = []
        for j in range(i): # j décrit [[0, i-1]]
            L.append((i+1)**(j+1))
        M.append(L)
    return M
```

   suivi de `tableau_triangulaire(5)` .

   Quel est la complexité de la fonction `tableau_triangulaire` ?

2 Recherche des deux valeurs les plus proches

Pour une liste de longueur au moins égale à deux d'éléments comparables deux à deux, on voudrait déterminer les deux valeurs les plus proches. Pour faire simple, on va prendre une liste d'entiers et la distance de a à b sera $|a-b|$ obtenue en Python avec abs(a−b).

On peut implémenter cela dans une fonction qui prend en argument une liste conforme à ce qui précède et qui retourne le couple des indices de deux valeurs parmi les plus proches.

On vous propose l'algorithme suivant :

Fonction plus proches
Entrées : liste L d'au moins 2 nombres
Sorties : paire d'indices des items les plus proches dans L

```
1  début
2      soit n le nombre d'éléments de L,
3      soit m la meilleure paire trouvée (pour l'instant). m est initialisée avec la paire (0, 1)
         c'est à dire les rangs des deux premiers items
4      soit d la plus faible distance trouvée. d est initialisée avec la distance entre les items de
         rangs 0 et 1
5      pour i allant de 0 à n − 2 faire
6          pour j allant de i + 1 à n − 1 faire
7              si les items de rangs i et j ont une distance inférieur à d alors
8                  d prend la valeur de la distance entre les items de rangs i et j
9                  m prend la valeur (i, j)
10             fin
11         fin
12     fin
13     renvoyer m
14 fin
```

1. Faites l'implémentation Python de `plus_proches(L)`
2. Si la liste est de longueur n, combien y a-t-il de comparaisons ?
   La complexité est-elle linéaire ? On parle de complexité quadratique.
3. Proposez une fonction `ppd` (plus proches distincts) qui pour une liste contenant au moins deux valeurs distinctes, renvoient les rangs des deux items les plus proches **et distinctes**.
   La fonction renverra également la distance entre ces deux items.

3 Recherche d'un motif

On dispose d'un texte, d'une chaîne de caractères et on cherche dans cette chaîne, un motif – par ex. un mot. Il s'agit de savoir si le motif apparaît dans la chaîne et si oui, éventuellement, où et combien de fois ? On implémente ici un algorithme naïf qui consiste à chercher l'occurrence du premier caractère du motif dans le texte (ce que l'on est sensé savoir faire) puis lorsque on est dans ce cas, regarder ce qu'il en est du deuxième caractère du motif avec le caractère suivant du texte et ainsi de suite, en cas de succès, jusqu'à épuisement du motif. Dans la version la plus simple, on retourne un booléen qui représente la présence ou pas du motif dans le texte. On peut faire

```
def recherche(texte, motif):
    n, p = len(texte), len(motif)
    if n < p:
        return False
    for i in range(n-p+1):
        if texte[i] == motif[0]:
            j = 1
            while j < p and texte[i+j] == motif[j]:
                j += 1
            if j == p:
                return True
    return False
```

1. Justifier le `n-p+1` de la cinquième ligne ?

   Dans le même esprit, faire des commentaires qui semblent pertinents pour chaque boucle et chaque `return`.
2. Quels sont les coûts de cet algorithme, dans le meilleur des cas et dans le pire des cas ?
3. Modifier la fonction `recherche` pour qu'en cas de présence du motif dans le texte, elle retourne le nombre d'occurrences de `motif` dans `texte` et tous les indices d'apparition.

4 Premiers tris

Dans toute activité scientifique, le classement est essentiel. Il s'agit de ranger les éléments dans un certain ordre, afin de pouvoir s'y retrouver plus facilement, plus rapidement. Cet ordre peut-être arbitraire, mais quand on a affaire à des nombres on utilise en général l'ordre naturel dans $\mathbb{N}$, ou dans $\mathbb{Z}$, ou dans $\mathbb{R}$, c'est-à-dire en informatique, dans l'ensemble des flottants (en prenant des précautions). Quand on a des caractères on prend l'ordre alphabétique et quand on a des chaînes de caractères, on prend l'ordre lexicographique, celui des dictionnaires de la vie courante – *à ne pas confondre avec les dictionnaires Python, il n'y a pas d'ordre pertinent dans les clés ... on y reviendra en deuxième année*. Tout ceci est déjà implémenté dans Python.

Nous allons tout au long de l'année étudier plusieurs fonctions, procédures, de tri et les comparer de divers points de vue. Pour simplifier cette première approche, on va se contenter de trier des tableaux (en Python ce sera des listes) d'entiers. Pour commencer on étudie ici deux tris par comparaison (on compare les nombres) qui utilisent des boucles imbriquées. Il y en a d'autres du même type (au sens courant du mot) et de types différents que l'on verra plus tard. Évidemment il y a au moins une fonction (et une méthode), disponible dès l'ouverture de Python, qui fait le tri d'une liste ; nous verrons aussi cela plus tard.

4.1 Tri par sélection

Au cours de la progression de l'algorithme, le tableau est séparé en 2 zones : une zone déjà triée et une zone à trier.

La méthode consiste à sélectionner le plus petit item parmi les non-triés et à le placer à la suite de la zone triée. Ainsi la zone triée finie par contenir la totalité des items.

Fonction tri par sélection
Entrées : liste L

```
1  début
2      Soit n le nombre d'items de L,
3      pour Pour i allant de 0 à n − 2 faire
4          Au début de la i^eme répétition, les i − 1 premiers items sont triés et sont inférieurs
5          aux items suivants. Il s'agit de trouver le rang du plus petit à partir du rang i
6          Soit i_min, initialisé à i,        représente indice du min trouvé
7          pour j allant de i + 1 à n − 1 faire
8              si item de rang j < item de rang i_min alors
9                  i_min prend la valeur j
10             fin
11         fin
12             À ce stade, i_min est le rang du plus petit des items de rang ⩾ i
13         Permuter les items de rang i et de rang i_min
14     fin
15 fin
```

Remarques : Il n'y pas de **sortie**. On ne renvoie pas de nouvelle liste, copie de la liste en entrée mais triée. On trie directement la liste fournie en entrée. On dit que c'est un tri **en place**.

La fonction est une **procédure** – pas de sortie – qui produit un effet de bord – traduction littérale de *side effect* – c'est à dire qu'elle modifie l'objet sur lequel elle travaille.

On étudiera la validité de l'algorithme (terminaison, correction) et le coût-complexité plus tard.

Faites l'implémentation de cet algorithme dans une fonction tri_selection (L).

4.2 Tri à bulles

L'algorithme du tri à bulles d'une liste L de n élément, avec $n > 1$, consiste
— à prendre successivement tous les couples d'indices $(i-1, i)$ pour $1 \leqslant i \leqslant n-1$,
— à comparer $L[i]$ et $L[i-1]$ et à les permuter s'ils ne sont pas dans le bon ordre.

Prenons l'exemple de L = [2, 3, 1, 7, 5, 1] on obtient les états successifs :
— i = 1 : L[0] < L[1] sont dans le bon ordre,
— i = 2 : L[1] > L[2] → permutation -> L = [2, 1, 3, 7, 5, 1],
— i = 3 : L[2] < L[3] → rien à faire,
— i = 4 : L[3] > L[4] → permutation → L = [2, 1, 3, 5, 7, 1],
— i = 5 : L[4] > L[5] → permutation → L = [2, 1, 3, 5, 1, 7]

À l'issue de cette boucle, on est certain que le plus grand élément est bien à droite. En revanche, le reste du tableau n'est pas trié. Il faut recommencer, mais il n'est plus nécessaire d'aller jusqu'au dernier élément. On peut parcourir $1 \leqslant i \leqslant n-2$.

Suivra une troisième boucle avec $1 \leqslant i \leqslant n-3$, et ainsi de suite jusqu'à une dernière avec $1 \leqslant i \leqslant 1$.

Fonction tri à bulles
Entrées : liste L

```
1  début
2  |  Soit n le nombre d'items de L,
3  |  pour Pour sup allant de n − 1 à 1 faire
4  |  |     On fait remonter une bulle avec 1 ⩽ i ⩽ sup
5  |  |  pour i allant de 1 à sup faire
6  |  |  |  si item de rang i − 1 > item de rang i alors
7  |  |  |  |  Permuter les deux éléments
8  |  |  |  fin
9  |  |  fin
10 |  fin
11 fin
```

1. Grâce à une deux boucles imbriquées, implémenter cet algorithme en Python en une procédure tri_bulles d'argument une liste de nombres qui modifie en place la liste pour qu'elle soit classée en sortie de fonction – de procédure – par ordre croissant des éléments.
2. Pourquoi « tri à bulles » ?
3. Préciser le nombre de comparaisons effectuées.
   On parle de coût (complexité) quadratique, comment justifier cela ?
4. Comment justifier – prouver – le fait que cet algorithme est conforme à sa spécification, c'est-à-dire ici qu'il classe la liste ?