Informatique PCSI
Prérequis TP 2 : boucles imbriquées

Les programmes rencontrés dans le TP 1 contiennent des boucles simples. Mais il est possible de placer plusieurs boucles `while` ou `for` à l'intérieur d'une boucle `while` ou `for`. On parle alors de boucles imbriquées.

Par exemple :

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

Les valeurs successives des variables `i` et `j` sont :

`i = 0` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite
`i = 1` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite
`i = 2` et `j = 0`, puis `j = 1`, puis `j = 2`, ensuite
`i = 3` et `j = 0`, puis `j = 1`, puis `j = 2`.

Pour chacune des quatre valeurs de `i`, (0, 1, 2, 3), `j` prend trois valeurs, (0, 1, 2), et nous aurons donc douze affichages avec la fonction `print`.

Voici deux exemples pour créer des listes emboîtées avec une définition en compréhension :

```
>>> liste = [[x, y] for x in range(3) for y in range(3)]
>>> liste
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
>>> liste = [[x, y] for y in range(3) for x in range(3) if x != y]
>>> liste
[[1, 0], [2, 0], [0, 1], [2, 1], [0, 2], [1, 2]]
```

1 Recherche des deux points les plus proches

On cherche une paire de points dont la distance est minimale dans un ensemble fini de points.

Algorithme naïf : on teste toutes les paires. Si l'ensemble contient n éléments, il y a $n(n-1)/2$ paires à tester. On peut utiliser des nombres avec pour distance entre x et y, `abs(x-y)`, ou se placer dans le plan avec la distance euclidienne.

```
def plus_proches(liste, dist):
    n = len(liste)
    paire = [0, 1]
    d = dist(liste[0], liste[1]) # distance à définir
    for i in range(n-1):
        for j in range(i+1, n):
            dij = dist(liste[i], liste[j])
            if dij < d:
                d = dij
                paire = [i, j]
    return paire
```

Pour chaque valeur de `i`, `j` prend les valeur de `i+1` à `n-1`. L'algorithme consiste à calculer toutes les distances entre deux points sans répétitions. Nous avons donc $n(n-1)/2$ distances à calculer. Nous disons que la complexité de l'algorithme en fonction de n est de l'ordre de n^2.

2 Recherche textuelle

La question est de déterminer la présence ou l'absence d'un motif dans un texte. Le texte et le motif sont représentés en Python par des chaînes de caractères (type str). Ils sont donc composés de caractères qui peuvent être des lettres, des signes de ponctuations, différents symboles et types d'espace. Si le motif est constitué d'un unique caractère, il s'agit d'une simple recherche séquentielle traitée au chapitre 1. Dans ce cas, le coût de la recherche est linéaire en la longueur de la chaîne.

Recherche naïve

On parle d'algorithme naïf quand il s'agit de la mise en œuvre d'une idée simple, l'une des premières idées à laquelle on peut penser. Le principe est le suivant :

- on cherche la présence du premier caractère du motif dans le texte ;
- si on le trouve, on vérifie si les caractères suivants du motif coïncident avec ceux du texte.
- si tous les caractères coïncident, le motif est trouvé, sinon on reprend la recherche du premier caractère.

Cet algorithme est implémenté dans le programme suivant :

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

À partir de l'indice `n-m-1`, ce n'est plus la peine de chercher car il ne reste plus assez de place pour caser le motif à la fin du texte.

indice j	...	...	...	k	k+1	k+2	k+3	...	...
texte				e	x	e	m		
				↕	↕	↕			
motif				e	x	e	r		
indice i				0	1	2	3		

Étude du coût

On note n la longueur du texte et m la longueur du motif. La boucle externe `for` est parcourue $(n-m+1)$ fois. Dans le pire des cas, la boucle interne `while` est parcourue au plus m fois, pour tester chaque caractère du motif. Le nombre total de comparaisons entre caractères est donc majoré par $m(n-m+1)$. En particulier, si $m = n/2$, la valeur de ce produit vaut $(n^2+2n)/4$.