Chapitre 2 : Boucles imbriquées opérant sur un structure séquentielle

Certains algorithmes nécessitent d'effectuer une deuxième boucle à chaque passage dans une boucle. On parle de boucles imbriquées avec une première boucle appelée boucle externe et une seconde dite interne. Nous allons découvrir leur utilité dans certaines recherches dans des structures séquentielles et découvrir leur rôle dans un algorithme de tri. Par ailleurs, nous allons introduire la notion de coût (appelée complexité d'un algorithme) qui sera détaillée dans un prochain chapitre.

1. Recherche des deux points les plus proches

1.1.Principe

Soit une liste contenant `n` éléments. On cherche ici une paire de valeurs dans la distance est minimale. La notion de distance pourra être différente d'un exercice à l'autre mais nous considèrerons ici que la distance entre deux éléments `x` et `y` sera définie par la valeur absolue `abs(x-y)`. Un algorithme naïf consiste à tester toutes les paires possibles.

1.2.Algorithme naïf

Pour effectuer cette recherche, il est nécessaire de parcourir la liste en associant tour à tour chaque élément `i` les différents autres éléments `j` de la liste afin de former tous les couples `(i;j)` possibles (attention, on devra nécessairement avoir `i` et `j` différents).
On peut noter que deux boucles non-conditionnelles seront nécessaires :

- Une première boucle externe afin de parcourir tous les éléments `i` de la liste
- Une seconde boucle interne afin de parcourir une seconde fois la liste pour les éléments `j` différents de `i`

✓ Écrire à l'aide de deux boucles `for` une fonction `plus_proches1` renvoyant les rangs des deux valeurs les plus proches d'une liste `L`

```
```

On peut remarquer que cet algorithme n'est pas optimal car il teste toutes les paires deux fois.

✓ Modifier le programme précédent afin d'écrire une fonction `plus_proches2` optimisée.

1.3.Coût

S'intéresser au coût de cet algorithme revient à évaluer, pour une liste de longueur `n`, le nombre de fois où la distance entre deux points (opérations la plus coûteuse en terme de temps) est calculée.
Pour une valeur de `i` fixée, on a `n-1-i` itérations de la boucle interne et donc `n-1-i` calculs de distance. Le nombre exact de calculs de distance est donc égale à `n*(n-1)/2`.
On parle ainsi de complexité quadratique car, pour les grandes valeurs de `n`, le coût évolue comme le carré de la longueur de la liste.

2. Tri à bulles

2.1.Introduction

Trier les données d'une structure séquentielle consiste à les organiser suivant un ordre préétabli tel que l'ordre alphabétique pour des caractères/mots ou par ordre croissant pour des nombres. Cette organisation implique que la structure doit être indexable et mutable. Ainsi, seules les listes, dictionnaires et chaînes de caractères pourront faire l'objet d'une telle procédure.
Cette année, nous verrons différents algorithmes permettant d'effectuer cette tâche dont le tri à bulles qui nous intéresse ici.

2.2.Principe

L'algorithme du tri à bulles consiste à parcourir la structure séquentielle autant de fois que nécessaire pour qu'elle soit finalement triée. Pour chaque parcours, on compare deux éléments consécutifs et on les inverse dans le cas où ils ne sont pas bien ordonnés. Le processus devra prendre fin dès lors qu'un parcours aura lieu sans qu'aucune modification ne soit effectuée.

Exemple : on considère la liste suivante `L=[15,19,8,7,2]`. On a alors :
- Premier passage : `L=[15,8,19,7,2]` puis `L=[15,8,7,19,2]` puis `L=[15,8,7,2,19]`
- Deuxième passage : `L=[8,15,7,2,19]` puis `L=[8,7,15,2,19]` puis `L=[8,7,2,15,19]`
- Troisième passage : `L=[7,8,2,15,19]` puis `L=[7,2,8,15,19]`
- Quatrième passage : `L=[2,7,8,15,19]`
- Cinquième passage : `L=[2,7,8,15,19]`

L'algorithme s'arrête après le cinquième passage puisqu'aucune permutation n'est effectuée. On remarque que les plus grands nombres remontent ainsi dans la liste tels des bulles d'air dans un liquide (d'où le nom de cette méthode de tri).

2.3.Algorithme

Afin d'optimiser dès le début le programme, nous pouvons remarquer qu'il n'est pas nécessaire de parcourir la structure séquentielle jusqu'à la fin à chaque parcours. En effet, après un passage, l'élément le plus grand est forcément en dernière position. Ainsi, au deuxième parcours, il ne sera pas la peine de le comparer à son prédécesseur qui est nécessairement plus petit. Idem pour l'avant dernier élément lors du troisième passage et ainsi de suite.

Écrire à l'aide d'une boucle `while` et d'une boucle `for` une fonction `tri_bulles` permettant le tri d'une liste `L`.

2.4.Coût

On s'intéresse ici au nombre de permutations à effectuer pour trier la liste `L` de longueur `n`. L'utilisation d'une boucle conditionnelle ne permet pas de connaître le nombre exact de permutations puisqu'on ne connaît pas le nombre de passage dans la boucle. On examine ainsi le meilleur et le pire des cas de figure pouvant se présenter.

- Dans le meilleur des cas, la liste est déjà bien ordonnée. Il n'y a donc aucune permutation et `n-1` comparaisons à effectuer.
- Dans le pire des cas, la liste est ordonnée dans le mauvais sens (par exemple par ordre décroissant au lieu de croissant s'il s'agit de nombres).
  Supposons qu'on est la liste `L=[n-1,n-2,…,1,0]`.
  - Premier passage : on aura `n-1` permutations pour amener le terme `n-1` en dernière position.
  - Deuxième passage : on aura `n-2` permutations pour amener le terme `n-2` en avant-dernière position
  - ...
  - Dernier passage avec permutation : on aura une permutation entre le `1` et le `0`.

  Au final, on aura dans le pire des cas `n*(n-1)/2` permutations dans le pire des cas.

On parle une fois de plus de complexité quadratique mais dans le pire de cas.

3. Recherche textuelle

3.1.Principe

Il s'agit ici de rechercher la présence d'une suite de caractères (lettres, signes de ponctuation, espace...) de longueur `m` au sein d'un texte (chaîne de caractères) de longueur `n`.
Si la suite ne contient qu'un unique caractère alors il s'agit d'une recherche d'occurence déjà vue au chapitre précédent. On se place ici donc dans le cas où `m>1`.

3.2.Algorithme naïf

L'idée la plus simple qui vient à l'esprit est la suivante :
- On commence par chercher la présence du premier caractère du motif
- Dès lors que le premier caractère du motif apparait dans le texte, on regarde les éléments suivants pour les comparer à ceux du motif
- Si tous les caractères correspondent, alors le motif est trouvé. Dans le cas contraire, on recommence à chercher le premier caractère.

À noter qu'à partir du rang `n-m-1`, ce n'est plus la peine de chercher le motif car il ne reste plus assez de place dans le texte pour pouvoir le rencontrer.

Écrire à l'aide d'une boucle `for` et d'une boucle `while` une fonction `recherche` permettant de rechercher la chaîne `motif` dans le texte `texte`.

3.3.Amélioration de l'algorithme

Avec le précédent algorithme, certaines comparaisons sont effectuées de manière systématique alors que la connaissance du motif recherché permet de savoir qu'elles sont inutiles.

Considérons par exemple le motif `'abcd'` recherché dans le texte `'abcfkthfabcd'`.

Une amélioration possible consiste donc à éviter ces étapes inutiles. Pour cela, on a recours à un pré-traitement du motif.

N° Passage Boucle externe	N° Passage Boucle interne	j	i	Test effectué	Réponse du test
1		0	0	'a'=='a'	True
	1	0	1	'b'=='b'	True
	2	0	2	'c'=='c'	True
	3	0	3	'd'=='f'	False
2		1	0	'a'=='b'	False
3		2	0	'a'=='c'	False
4		3	0	'a'=='f'	False
5		4	0	'a'=='k'	False
6		5	0	'a'=='t'	False
7		6	0	'a'=='h'	False
8		7	0	'a'=='f'	False
9		8	0	'a'=='a'	True
	1	8	1	'b'=='b'	True
	2	8	2	'c'=='c'	True
	3	8	3	'd'=='d'	True
	4	8	4		
		8			
Fin du programme					

Le pré-traitement du motif consiste à créer un tableau de distances tel que :

- Si un caractère `'c'` apparait dans le motif, la distance est définie comme le nombre de caractères entre la dernière occurence de `'c'` (exceptée la dernière place) et la fin du motif
- Si un caractère `'c'` n'apparait pas dans le motif ou seulement en dernière position, la distance sera définie comme la longueur (nombre de caractères) du motif

L'ensemble de ces informations est stocké dans un dictionnaire pour lequel les clés sont les différents caractères possibles et les valeurs, les distances.

Considérons le motif `'arnaud'` ; on aurait alors :

```
{'a' : 2 , 'r' : 4 , 'n' : 3 , 'u' : 1 , 'd' : 5 }
```

Plutôt que de limiter ce dictionnaire aux caractères apparaissant dans le motif, on l'étend à tous les caractères majuscules, minuscules, de ponctuation... Pour cela, on utilise les `256` caractères codés par l'ASCII étendu.

Écrire à l'aide d'une boucle `for` une fonction `distances` renvoyant un dictionnaire contenant tous les caractères en affectant les distances relatives à leur apparition dans la séquence `motif`.

Comme d'autres informaticiens, Boyer et Moore ont eu l'idée du pré-traitement du motif mais leur approche est différente car la comparaison entre le motif et une partie du texte s'effectue de droite à gauche, c'est-à-dire en commençant par la fin du motif.

Le script suivant présente une version simplifiée de leur algorithme.

```
def boyer_moore(texte,motif):
    m=len(motif)
    n=len(texte)
    d=distances(motif) # on effectue le pré-traitement du motif
    s=m-1  # on commence par le dernier caractère du motif
    sol=[] # on crée une liste afin de stocker les rangs des occurences du motif dans le texte
    while s<n:
        i=m-1
        while i>=0 and motif[i]==texte[s]:# tant qu'il y a correspondance entre les caractères du motif et du texte
            s=s-1                        # on recule d'un cran afin de comparer le caractère précédent du motif
            i=i-1                        # au caractère précédent du texte
        if i<0:                          # si le motif est trouvé (i=-1 car m passages dans la boucle précédente)
            sol.append(s+1)              # on stocke dans la liste sol le rang dans le texte du premier terme du motif
        s=s+max(d[texte[s]],m-i)
    return(sol)
```

Cette fonction répond aux critères suivants :

- La comparaison entre le motif et le texte s'effectue de la droite vers la gauche à partir du dernier caractère du motif.
- Si le motif est trouvé, on poursuit la recherche en décalant le motif d'un cran vers la droite et on stocke l'indice marquant le début d'une occurence dans une liste.
- Si le motif n'est pas trouvé, on poursuit la recherche en décalant le motif vers la droite de manière à faire coïncider le caractère fautif du texte avec sa dernière occurence dans le motif. S'il n'y en a pas, on décale de la longueur du motif.
- La fonction renvoie la liste des indices trouvés.

Appliquons cette fonction au motif `'abcd'` recherché dans le texte `'afcdkthfabcd'`.

Dans le cas du motif considéré, on a comme résultat du pré-traitement :

`{'a' : 3 , 'b' : 2 , 'c' : 1 , 'd' : 4 }`

N° Passage Boucle externe	N° Passage Boucle interne	S	i	sol	Test effectué	Réponse du test
1		3	3	[]	'd'=='d'	True
	1	2	2	[]	'c'=='c'	True
	2	1	1	[]	'b'=='f'	False
		5	1	[]		
2		5	3	[]	'd'=='k'	False
	1	5	3	[]		
		9	3	[]		
3		9	3	[]	'd'=='b'	False
	1	9	3	[]		
		11	3	[]		
4		11	3	[]	'd'=='d'	True
	1	10	2	[]	'c'=='c'	True
	2	9	1	[]	'b'=='b'	True
	3	8	0	[]	'a'=='a'	True
	4	8	-1	[]		
		12	-1	[8]		
Fin du programme						