TP4 – Algorithmes et listes : correction

Ada Lovelace (1815 – 1852)
Pionnière de la science informatique, elle est connue
pour avoir réalisé le premier programme informatique, dans les années 1840

Voici des programmes classiques qui sont à connaître.

On trouvera aussi quelques compléments de cours et commentaires.

Ces algorithmes prennent généralement en entrée une liste, qu'on note `l`. La complexité des algorithmes est donnée en fonction de $N := $ `len(l)`.

Exercice TP4.1 — Longueur d'une liste

```
def longueur(l):
    S = 0
    for x in l:
        S = S+1
    return S
```

complexité : $O(N)$

Remarque à lire dans un second temps

Comme la variable de boucle `x` *n'est pas utilisée, on peut l'appeler* `_`.

C'est un nom de variable comme un autre mais on l'utilise en `Python` *quand on souhaite insister sur le fait qu'on n'utilise pas la variable de boucle.*

```
def longueur_BIS(l):
    S = 0
    for _ in l:
        S = S+1
    return S
```

complexité : $O(N)$

Exercice TP4.2 — Somme et moyenne

```
def somme(l):
    S = 0
    for x in l:
        S = S+x
    return S
```

complexité : $O(N)$

La fonction suivante `moyenne(l)` traite le cas particulier où `l == []`.

En général, sauf mention expresse du contraire, quand on code une fonction `Python`, on part du principe que l'utilisateur appellera la fonction avec des arguments du bon type. Quand une fonction prend pour argument une liste, sauf mention expresse du contraire, il faudra toujours que la fonction traiter également le cas de la liste vide `[]`.

```
def moyenne(l):
    n = len(l)

    if n == 0:
        return 0
    else:
        return somme(l)/n
```

complexité : $O(N)$

Dans la fonction `moyenne(l)`, la complexité est cachée dans l'instruction `somme(l)` !

Exercice TP4.3 — Minimum et moyenne

```
def monMin(l):
    if len(l) == 0:
        return 0

    minimum = l[0]
    for x in l:
        if x < minimum:
            minimum = x

    return minimum


def monMax(l):
    if len(l) == 0:
        return 0

    maximum = l[0]
    for x in l:
        if x > maximum:
            maximum = x

    return maximum
```

complexité : $O(N)$ pour chacune des fonctions

Exercice TP4.4 — Modification d'une liste

Quelques remarques :

a) La fonction `modifie(l)` ne renvoie aucun objet mais effectue une action.
On dit que c'est une *procédure*.

b) Voici quelques conventions relatives au nommage des fonctions et des procédure.
- Une fonction qui renvoie un objet sera généralement nommée par un nom, comme `moyenne(l)`, `somme(l)`, `monMin(l)`, *etc.* désignant l'objet renvoyé.
- Une procédure sera nommée par un verbe d'action conjugué à la troisième personne, comme `modifie(l)`, `supprimeDernierElement(l)`, `echange(l,i,j)`, *etc.*
- Une fonction qui renvoie un booléen sera nommée avec le verbe être suivi d'un attribut/complément, comme `isInTheList(l,a)`, `isSorted(l)`, `estTriee(l)`, *etc.*

c) Il y avait un piège pour ce programme.
En effet, si `l` est la liste vide, alors le code `l[0]` provoquera une erreur

```
IndexError: list index out of range
```

Le code `l[-1]` provoquera la même erreur.
Il fallait donc traiter ce cas à part.

d) Dans une procédure, quand on est dans un cas où l'on veut interrompre l'exécution, il suffit d'écrire `return` tout court.

```
def modifie(l):
    if len(l) == 0:
        return

    l[-1] = l[-1] + 1
    l[0] = l[0] - 1
```

complexité : $O(1)$

Exercice TP4.5 — Test d'appartenance

Voici un petit programme très important, à savoir refaire impérativement !!

```
def isInTheList(l, a):
    for x in l:
        if x == a:
            return True

    return False
```

complexité : $O(N)$

Une faute classique

Attention ! Le code suivant est faux !

```
def isInTheList_FAUX(l, a):
    for x in l:
        if x == a:
            return True
        else:
            return False
```

complexité : $O(1)$

Dans ce programme, la boucle ne sera jamais parcourue au-delà du premier élément !
En effet, au premier tour, en notant `x_0` le premier élément de `l`, on a deux cas possibles :

- si `x_0 == a`, on sort de la fonction en renvoyant `True` ;
- sinon, `x_0 != a` et on sort de la fonction en renvoyant `False`.

Dans tous les cas, on sort de la fonction au premier tour de boucle.

À votre avis, le code suivant est-il correct ? Quelle est sa complexité ?

```
def isInTheList_MYSTERE(l, a):
    for x in l:
        if x == a:
            return True
        return False
```

`a in l`

En fait, il existe en `Python` une commande permettant de savoir si l'élément `a` est présent dans la liste `l`. C'est tout simplement :

```
a in l
```

qui renvoie un booléen : `True` ou `False`.

Attention : cette instruction est relativement dangereuse. En effet, sa syntaxe extrêmement simple pourrait laisser croire que c'est une commande qui s'exécute instantanément, ie *en* $O(1)$ *opérations.*

C'est en fait faux : si N est la longueur de `l`, alors la commande

```
a in l
```

a une complexité en $O(N)$ opérations.

Exercice TP4.5 — Premier indice si l'élément est présent

```
def firstIndex(l, a):
    n = len(l)
    for i in range(n):
        if l[i] == a:
            return i
    return -1
```

complexité : $O(N)$

Exercice TP4.5 — Nombre d'occurrences d'un élément dans une liste

```
def numberOfOccurences(l, a):
    nb = 0
    for x in l:
        if x == a:
            nb = nb + 1
    return nb
```

complexité : $O(N)$

Exercice TP4.6 — Test de tri

Pour tester si une liste `l` est triée, il suffit de vérifier à chaque rang `i` si on a bien `l[i] <= l[i+1]`.

Il y avait un piège dans cet exercice, puisque la définition de « est triée » incitait à faire deux boucles imbriquées. Dans ce cas, la complexité aurait été en $O(N^2)$, ce qui est beaucoup moins bien.

```
def isSorted(l):
    n = len(l)
    for i in range(n-1):
        if l[i] > l[i+1]:
            return False

    return True
```

complexité : $O(N)$

Exercice TP4.7 — Échange de deux éléments

Dans le programme qui suit, on va utiliser un élément de syntaxe spécifique à `Python` qui permet d'échanger les valeurs de deux variables. Plus précisément, si `x` et `y` sont des variables, pour échanger les valeurs de `x` et `y`, on fait

```
x, y = y, x
```

Cette syntaxe est en fait très générale. Par exemple, considérons le code suivant

```
x = 100
y = 50
x, y = x+y, x-y
```

Alors, après exécution, les valeurs de `x` et de `y` seront, respectivement `150` et `50`.

Cette syntaxe peut être appliquée à des éléments d'une liste. Par exemple, on pourra faire

```
l[i], l[j] = l[j], l[i]
```

pour échanger les valeurs d'indice `i` et `j` de la liste `l`.

Voilà la réponse à la question posée :

```
def echange(l, i, j):
    if i >= n or j >= n:
        return

    if i < -n or j < -n:
        return

    l[i], l[j] = l[j], l[i]
```

complexité : $O(1)$

Exercice TP4.8 — Recherche par dichotomie dans une liste triée

L'algorithme qui suit a une rapidité fulgurante.

Sa complexité est en $\mathrm{O}(\ln N)$.

Pour le voir, il suffit de comprendre ce qui se passe pour une liste dont la longueur N égale 2^p, où $p \in \mathbb{N}$. Pour une telle liste, au pire des cas, après p tours de boucles, l'algorithme aura terminé. Ainsi, la complexité est $\mathrm{O}(p)$. Or, on a

$$p = \log_2(2^p) = \frac{\ln(N)}{\ln(2)}.$$

La constante $\frac{1}{\ln(2)}$ peut être intégrée au $\mathrm{O}(\cdot)$: on a $p = \mathrm{O}(\ln(N))$. On voit que, en général, la complexité de cet algorithme est en $\mathrm{O}(\ln(N))$.

Recherche par dichotomie dans une liste triée

```
def index_by_dichotomy(l, a):
    n = len(l)
    if n == 0:
        return -1

    i_min = 0
    i_max = n-1

    while i_min < i_max:
        i_milieu = (i_min + i_max)//2
        b = l[i_milieu]
        if a == b:
            return i_milieu
        elif a < b:
            i_max = i_milieu - 1
        else:
            i_min = i_milieu + 1

    if l[i_min] == a:
        return i_min
    else:
        return -1
```

complexité : $\mathrm{O}(\ln(N))$

Programme enrichi pour compter le nombre de boucles

```
def index_by_dichotomy_BIS(l, a):
    nb_boucles = 0

    n = len(l)
    if n == 0:
        return -1, nb_boucles

    i_min = 0
    i_max = n-1

    while i_min < i_max:
        nb_boucles = nb_boucles + 1
        i_milieu = (i_min + i_max)//2
        b = l[i_milieu]
        if a == b:
            return i_milieu, nb_boucles
        elif a < b:
            i_max = i_milieu - 1
        else:
            i_min = i_milieu + 1

    if l[i_min] == a:
        return i_min, nb_boucles
    else:
        return -1, nb_boucles
```

complexité : $O(\ln(N))$

Version récursive

```
def index_by_dichotomy_RECURSIF(l, a):

    def index_aux(i_min, i_max, N):
        if i_min == i_max:
            if l[i_min] == a:
                return i_min
        else:
            return -1

        i_milieu = (i_min + i_max)//2
        b = l[i_milieu]
        if a == b:
            return i_milieu
        elif a > b:
            return index_aux(i_milieu + 1, i_max)
        else:
            return index_aux(i_min, i_milieu - 1)

    n = len(l)
    return index_aux(0, n-1)
```

complexité : $O(\ln(N))$

Exercice TP4.9 — Distance minimale

L'algorithme qu'on propose ci-dessous est un *algorithme naïf*. En informatique, ce terme n'est pas péjoratif : cela veut dire que c'est l'algorithme naturel qui répond à la question, sans être forcément l'algorithme le plus rapide.

Ici, on va regarder tous les couples d'indices (`i`,`j`) tels que `i` $\neq$ `j` et trouver le couple optimal (`i_0`, `j_0`) tel que $\left|$`l[i_0]-l[j_0]`$\right|$ soit minimal.

Ceci serait vraiment l'algorithme naïf. En fait, on va faire un peu plus malin et ne regarder que les couples d'indices (`i`,`j`) tels que `i` $<$ `j`.

En `Python`, pour calculer la valeur absolue, on utilise la commande `abs(x)`. Remarquez que cette commande ne nécessite pas d'avoir chargé le module `math`.

```
def distanceMin(l):
    n = len(l)
    if n == 0 or n == 1:
        return -1

    best_i = 0
    best_j = 1

    bestDistance = abs(l[best_j] - l[best_i])

    for i in range(n):
        for j in range(i,n):
            distance = abs(l[j] - l[i])
            if distance < bestDistance:
                best_i = i
                best_j = j
                bestDistance = distance

    return (best_i, best_j)
```

complexité : $O(N^2)$

La complexité de cet algorithme est en $O(N^2)$: en effet, il y a deux boucles imbriquées.

Remarque

En fait, il existe un algorithme avec une complexité bien meilleure pour répondre à cette question. En effet, si on trie la liste `l` *alors on est assuré que la distance minimale séparant deux termes de* `l` *sera nécessairement atteinte pour deux termes successifs. Il suffit donc de parcourir la liste triée, ce qui fait en* $O(N)$ *opérations.*

Or, il existe des algorithmes qui trient une liste en $O(N\ln(N))$ *opérations.*

Conclusion : cet algorithme trouve la valeur cherchée en $O(N\ln(N))$ *opérations, ce qui est une complexité bien meilleure.*

Exercice TP4.10 — Élément le plus proche

```
def indexOfNearestElement(l, a):
    n = len(l)
    if n == 0:
        return -1

    best_i = 0
    bestDistance = abs(l[best_i] - a)

    for i in range(n):
        distance = abs(l[i] - a)
        if distance < bestDistance:
            best_i = i
            bestDistance = distance

    return best_i
```

complexité : $O(N)$

Exercice TP4.11 — Palindromes

Le programme qui suit est très classique.

On parcourt la « moitié gauche » de la liste pour regarder si chaque élément est égal à l'élément correspondant dans la « moitié droite ».

Si `l` possède un nombre impair d'élément, alors on n'a pas besoin de tester l'élément central de la liste. Cela nous permet de voir rapidement que la boucle doit aller de `i = 0` à `(n//2 - 1)`. En effet, pour une liste de longueur `n = 3`, on veut s'arrêter à l'indice `0` :

- ça ne peut pas être `((n-1) // 2)` qui dans ce cas-là vaut `1` ;
- *a fortiori*, ça ne peut pas être `n//2` ;
- en revanche, `(n//2 - 1)` fonctionne.

D'ailleurs, pour une liste de longueur `n = 4`, on doit s'arrêter à l'indice `i = 1`, qu'on retrouve bien avec l'opération `(n//2 - 1)`.

```
def isPalindrome(l):
    n = len(l)

    for i in range(n//2):
        if l[i] != l[(n-1) - i]:
            return False

    return True
```

complexité : $O(N)$

Exercice TP4.12 — Renverser une liste

On s'inspire fortement de la fonction précédente.

```
def renverse(l):
    n = len(l)

    for i in range(n//2):
        l[i], l[(n-1) - i] = l[(n-1) - i], l[i]
```

complexité : $O(N)$

Exercice TP4.13 — Rotation d'une liste

On commence par définir une fonction qui décale tous les termes d'une liste d'un cran.

```
def shift_aux(l):
    n = len(l)

    for i in range(n-1):
        l[i], l[i+1] = l[i+1], l[i]
```

complexité : $O(N)$

On peut alors coder :

```
def shift(l, p):
    for i in range(p):
        shift_aux(l)
```

complexité : $O(N \times p)$

Il est possible de coder une fonction `shift(l, p)` qui serait de complexité en $O(N)$.

Exercice TP4.14 — Retirer un élément d'une liste

```
def retireElement(l, i):
    for j in range(i, n-1):
        l[j], l[j+1] = l[j+1], l[j]

    l.pop()
```

complexité : $O(N)$

Il existe en `Python` une commande qui permet de retirer de la liste `l` son élément d'indice `i`. C'est :

```
l.pop(i)
```