TP 3 boucles imbriquées et boucles conditionnelles

Comme d'habitude, on lance spyder (Mathématiques→WinPython), on crée un répertoire TP3 dans le répertoire InfoPCSI de votre session, et on sauvegarde le script de l'éditeur de code (fenêtre de gauche) sous le nom `tp3.py`.

On travaillera encore ici sur des listes d'entiers, et on pourra de nouveau introduire dans notre script le code suivant :

```
from random import randrange

def listeAleatoire(n):
    L = []
    for i in range(n):
        L.append(randrange(100000))
    return L
```

1 Recherche des deux valeurs les plus proches dans un tableau

Etant donnée une liste `L` de nombres (qu'on peut aussi appeler tableau à une dimension), on cherche quelles sont les deux valeurs de `L` les plus proches (l'une de l'autre). L'objet sera de renvoyer un couple (v_1, v_2) de deux termes présents dans `L` à des indices distincts tels que pour tout autre couple (v_3, v_4) de termes présents dans `L` à des indices distincts, $|v_1 - v_2| \leqslant |v_3 - v_4|$.

Il n'y a sans doute pas unicité d'un tel couple, mais on fera en sorte que le couple (v_1, v_2) renvoyé par notre fonction soit égal à un couple `(L[i], L[j])` rendant minimal `i` puis `j`. Par exemple pour la liste `[5, 2, 1, 4, 6]` c'est le couple `(5, 4)` qui sera renvoyé.

Question 1. Compléter la fonction suivante pour répondre au problème posé :

```
def valeursProches(L):
    ecartMin = abs(L[1] - L[0]) + 1 # un écart qui ne saurait être optimal
    for i in range(...):
        for j in range(...):
            if abs(...)< ...:
                a, b =
                ... =
    return a, b
```

Question 2. Pour une liste de n termes, au sein des deux boucles for apparaissant dans la fonction précédente, combien d'itérations au total auront été effectuées ?

N.B. : on parlera ici d'un coût quadratique selon n.

Question 3. Vérification expérimentale de la complexité : écrire une fonction d'entête `def tempsValeursProches(n):` qui admet un seul argument entier n, qui créée une liste aléatoire de n termes (à l'aide bien sûr de `listeAleatoire`) et qui mesure alors le temps mis à la recherche des deux valeurs les plus proches de la liste générée, et qui renvoie le temps, en secondes, écoulé par ce traitement. On pourra faire appel à la fonction `perf_counter` du module `time`.

Tester avec $n = 250, 500, 1000, 2000$. Commentaires ?

Question 4. Si la question était plutôt celle de déterminer deux valeurs les plus éloignées d'une liste `L`, montrer qu'un algorithme en coût linéaire permet d'y répondre.

2 Petite digression : boucles conditionnelles et recherche linéaire d'une valeur dans une liste

Le problème est ici le suivant, on cherche dans une liste donnée `L` si une certaine valeur `v` apparaît. L'objet sera de renvoyer le premier indice `i` tel que `L[i]` vaut `v` si `v` est dans `L`, et de renvoyer `None` sinon. On a vu au dernier TP comment avec une boucle `for` répondre à cette question. Il est un autre type de boucle, appelée boucle conditionnelle (qu'on nommera aussi, par abus de langage, une boucle `while`) qui peut être judicieuse pour ce genre de problème. La syntaxe en est :

```
while condition:
    instruction1
    ...
    instructionn
```

Bien sûr, on entend par `condition` une expression qui peut s'interpréter par une valeur booléenne (`True` ou `False`) et, on

le devine si on connaît un tout petit peu d'anglais, que le corps de la boucle est exécuté, de manière répétée, tant que ladite condition est satisfaite (ce qui n'interdit donc pas une boucle infinie…)

Une boucle `while` est à préférer à une boucle `for` lorsqu'on ne connaît pas a priori le nombre d'itérations qui nous seront nécessaires.

Question 5. Soit (u_n) la suite définie par récurrence par $u_0 = 0$ et $\forall n \in \mathbb{N}, u_{n+1} = 2\,u_n + n$. Ecrire alors une fonction d'entête `def jusque(v):` qui affiche tous les termes de (u_n) qui sont au plus égaux à `v`. Par exemple, `jusque(10)` affichera :

```
0
0
1
4
```

car $u_0 = 0$, $u_1 = 1$, $u_2 = 4$ et $u_3 = 11$.

Question 6. Ecrire une fonction d'entête `def indiceDe(L, v):` qui, à l'aide d'une boucle `while` renvoie le premier indice `i` où figure `v` dans `L` s'il en fait partie, et `None` sinon.

3 Recherche de sous-liste

Le problème est le suivant : on dispose de deux listes `L` et `K` comportant respectivement n et p termes, et la question est de savoir si les termes de `K` peuvent se trouver, de manière consécutive, dans `L`. On dira ainsi que `K` figure dans `L` à partir de l'indice `i` pourvu que `L[i]` et `K[0]` soient égaux, ainsi que `L[i+1]` et `K[1]` et ce jusqu'à `L[i+p-1]` et `K[p-1]`.

C'est le même problème que serait celui de chercher un mot ou une phrase dans un texte donné (et l'algorithme que l'on va implémenter permettra aussi de donner une réponse à ce problème).

Remarque 1. On peut extraire d'une liste une sous-liste à l'aide de « tranche » (slice en anglais). Ainsi avec la syntaxe `L[a:b]` on extrait de la liste `L` une nouvelle liste formée de tous les termes de `L` dont l'indice `i` satisfait aux inégalités `a<=i<b`.

Avec `L[a:]` on obtient la liste formée des termes de `L` d'indice `i>=a` (jusqu'à la fin de la liste `L` donc), avec `L[:b]` la liste formée des termes de `L` d'indice `i<b`. Avec `L[:]` on obtient une nouvelle liste formée des mêmes termes que `L` : c'est aussi ce qu'on appelle une copie superficielle de `L`. C'est très différent de ce qu'on obtiendrait avec l'affectation `K = L` qui conduirait à ce que `L` et `K` désignent la même liste, et que toute modification de l'une (ajout ou retrait d'un élément par exemple) soit aussi une modification de l'autre.

On peut aussi, comme pour les `range` mettre un pas dans les tranches : ainsi avec `L[a:b:c]` on construit une nouvelle liste dont les indices `i` prennent la forme `a,a+c,a+2*c` tant que la valeur `b` n'est pas atteinte ou dépassée. Par exemple `L[::-1]` créée la liste formée des mêmes termes que `L`, mais en sens inverse.

La question posée, de savoir si `K` (de longueur `p`) figure dans `L` à partir de l'indice `i` peut alors trouver une réponse à l'aide du test suivant : `L[i:i+p] == K`. Cela étant dit, on ne se permettra pas d'utiliser les tranches dans la fonction qu'on va écrire à la question 8.

Question 7. Avec les données précédentes (listes de n et p termes), indiquer jusqu'à quel indice `i` on cherchera la présence de `K` dans `L`.

Question 8. Sachant qu'on ne cherche pas toutes les occurrences de `K` dans `L`, mais qu'on s'arrêtera dès lors qu'on aura trouvé `K` dans `L`, proposer quels types de boucles on pourrait choisir pour réaliser ce travail. (Deux réponses sont acceptables)

Question 9. Ecrire une fonction d'entête `def sousListe(L, K):` qui admet deux listes en argument `L` et `K` et qui renvoie, si `K` est présente dans `L`, le premier indice `i` à partir duquel `K` apparaît dans `L`, et qui renvoie `None` si `K` n'est pas présente dans `L`.

4 Tri à bulles

On apprendra bientôt comment, à l'aide d'une dichotomie, rendre beaucoup plus efficace la recherche d'une valeur particulière dans une liste, pourvu que celle-ci soit triée. L'idée est de définir une zone de recherche et, en testant une valeur médiane, de diviser la zone de recherche par deux à chaque itération. Ainsi, en n itérations, on peut réaliser une recherche sur une liste de 2^n termes, ce qui est, il va sans dire, beaucoup plus efficace que de parcourir un par un les 2^n termes de cette liste !

(Exemple : en 10 tests, on peut donc chercher une valeur dans une liste de $2^{10} = 1024$ termes, en 16 tests, on cherche une valeur dans une liste de $2^{16} = 65536$ termes !)

Encore faut-il disposer d'une liste triée, ainsi nous étudierons plusieurs algorithmes permettant de trier une liste, et en voici le premier, dont le nom évoque l'idée d'une bulle d'air coincée entre deux feuilles de plastique et que par appui sur celle-ci on déplace de gauche à droite.

4.1 Premier balayage

Considérons la liste suivante, de 9 termes (indicés donc de 0 à 8 y compris) :

0	1	2	3	4	5	6	7	8
3	2	5	1	7	10	-1	4	0

Un premier balayage fonctionne ainsi : on compare les deux premiers termes (d'indices 0 et 1). S'ils sont dans le bon ordre (croissant), on ne change rien et on passe à la case suivante, sinon (comme c'est le cas ici), on les échange et on passe aux deux cases suivantes, d'indices 1 et 2, et ainsi de suite jusqu'à parvenir en fin de liste.

Question 10. Recopier la liste obtenue à la fin du balayage précédent. La liste obtenue est-elle triée ? Le plus petit terme de la liste est-il désormais bien placé (à l'indice 0 donc) ? le plus grand ?

Question 11. Réécrire la liste obtenue après un second balayage similaire au précédent, puis après un troisième.

4.2 Algorithme de tri à bulles

En vous aidant du paragraphe précédent, concevoir un algorithme permettant de trier une liste `L` de n valeurs numériques. Il est utile de se poser les questions suivantes :

- Combien de balayages seront a priori nécessaires dans le pire des cas (en partant d'une liste triée en sens inverse)
- Au k-ième balayage, quels seront les indices des deux derniers termes de `L` que l'on comparera et échangera si nécessaire ?

Question 12. Ecrire une fonction d'entête `def triBulles(L):` qui admet un argument `L` formé d'une liste de nombres, et qui modifie cette liste pour la trier. Notre fonction pourra renvoyer la liste `L` ainsi obtenue, ou renvoyer `None` au choix.

Question 13. En analysant le code écrit ci-dessus, combien de comparaisons auront été effectuées pour trier les n valeurs d'une liste `L` ?

Remarque 2. python sait déjà trier une liste, à l'aide d'un algorithme beaucoup plus efficace que le tri à bulles appelé le tri fusion et qu'on étudiera un peu plus tard dans l'année, implémenté dans la méthode `sort` d'une liste. Ainsi, si `L` est la liste `[3, 1, 2]` et qu'on invoque `L.sort()`, rien n'apparaît dans la console (ce qui indique que c'est la valeur `None` qui est renvoyée) mais la liste `L` a été modifiée pour être désormais la liste triée `[1, 2, 3]`.

5 Pour ceux qui ont encore un peu de temps

Le problème va être, étant donnée une liste `L` de nombres, de déterminer la longueur de la plus longue sous-liste en progression croissante (au sens large). Dans un premier temps, on en cherchera une sous-liste formée de termes consécutifs, puis dans un deuxième temps on autorisera que les termes issus de `L` ne se suivent pas directement. Exemple, avec `L = [24, 64, 12, 10, 53, 85, 25, 63, 53, 78, 13, 75]`, alors la plus grande sous-liste en progression croissante ne comporte que trois termes, il s'agit de `[10, 53, 85]`.

En revanche, si on se permet de sauter un ou plusieurs termes de `L`, on peut construire des sous-listes triées dans le sens croissant comportant plus que 3 termes. C'est le cas de `[24, 25, 53, 75]`, `[12, 53, 53, 78]` et on trouve de nombreux autres exemples de sous-listes croissantes de 4 termes.

Question 14. Ecrire une fonction d'entête `def longueurSousListeCroissante(L):` qui prend en argument une liste de nombres et renvoie la plus grande longueur d'une sous-liste de `L` croissante formée de termes consécutifs de `L`.

Question 15. Ecrire une fonction d'entête `def longueurSousListeCroissante2(L):` qui réalise le même travail, mais pour renvoyer la plus grande longueur d'une sous-liste de `L` croissante formée de termes non forcément consécutifs de `L` (mais tout de même placés les uns par rapport aux autres dans le même ordre que dans `L`)

6 Annexe : quelques éléments de syntaxe python

1. **Boucles `for` et `range`** : `range` est un constructeur qui nous sert à définir un ensemble d'entiers sur lesquels itérer, et qu'on utilise essentiellement pour créer des boucles `for`. Avec un seul argument `n`, `range(n)` va permettre d'itérer sur les valeurs de `0` à `n-1`. Avec deux arguments `a` et `b`, `range(a, b)` va itérer sur les entiers de `a` à `b-1`, et enfin on peut rajouter un troisième argument, lequel est le pas (par défaut de 1, mais on peut aller de 2 en 2, à l'envers avec un pas de -1). Ainsi par exemple, `range(10,2,-2)` va itérer sur les entiers de 10 à 2 de -2 en -2, l'indice final n'étant pas compris (il ne l'est jamais) c'est-à-dire sur les entiers 10, 8, 6, 4.

2. Les mots-clés `break` et `return` : dans un corps de boucle (`for` ou `while`), si l'instruction `break` est rencontrée, on sort alors aussitôt du corps de cette boucle (mais juste de celui-ci), que tous les tours de boucle aient été effectués ou non. Avec l'instruction `return` dans une fonction, où que cette instruction soit exécutée, on sort aussitôt de la fonction, avec la valeur de retour qui peut être précisée, ou non. (En l'absence d'une valeur de retour, c'est `None` qui est renvoyé)

3. **Listes** : étant donnée une liste `L`, `len(L)` donne le nombre de termes de `L`. Si `n` est ce nombre, alors les indices des termes de `L` sont numérotés de `0` à `len(L)-1` et on accède, en lecture et en écriture, au terme d'indice `i` par la syntaxe `L[i]`.

   Deux méthodes à connaître : `append` et `pop` : l'instruction `L.append(val)` rajoute à la fin de `L` le terme `val`, tandis que `L.pop()` extrait le dernier terme de `L` (le retire) et en renvoie la valeur

4. `perf_counter` : voici exemple d'utilisation :

```
from time import perf_counter
start = perf_counter()
unTraitementLong()
stop = perf_counter()
tempsEcoulé = stop - start # contient le temps écoulé dans unTraitementLong() en secondes
```