Spé Bio 1 & 2 - Lyc. Champollion Octobre 2024

Informatique - Python - TP N°3
Révision des tris simples et de la récursivité

Objectifs :

- Approcher la notion de **complexité** : mesurer l'efficacité des tris, observer que les algorithmes de tri "lents" sont quadratiques.
- Revoir la **récursivité**, à la base du "tri rapide" (Quicksort) et du tri fusion.

Remarques :

— La taille (nombre d'éléments) d'une liste L s'obtient par `len(L)` ; nous la noterons **N**.

— Philosophie de ce TP : on s'interdit ici l'utilisation des fonctions Python du type `remove()`, `del()`, `pop()`,`insert()`... pour éviter l'effet 'boîte noire'. On privilégie des algorithmes universels pour comparer leur efficacité indépendamment du langage de programmation.

— Les trois tris 'lents' proposés (par insertion, à bulles, par sélection) sont des tris **'en place'** : les fonctions modifient **directement l'ordre des éléments dans la liste de départ**, fournie en paramètre, et n'ont pas de **return**.
On rappelle en effet que les listes python sont ***mutables***, ce qui signifie notamment que toute modification des valeurs de la liste se fait directement en mémoire sur la liste (il n(y a pas de création d'une nouvelle liste).

1. **Le tri par insertion**.
Il s'agit d'insérer successivement chaque élément `L[i]` à la bonne place dans la partie de gauche **déjà triée**, qui s'agrandit progressivement. Déroulement :
   - [L[0]] contient un seul élément : elle est déjà triée ;
   - on y insère `L[1]` à la bonne place (avant ou après `L[0]`)
   - puis, on insère `L[2]` à la bonne place dans la sous-liste formée des deux premiers éléments...
   - et ainsi de suite jusqu'à la fin de la liste.

   Soit une liste `L`
   On suppose que ses éléments d'indices `0` à `i-1` sont **déjà triés** dans l'ordre croissant.
   Pour insérer l'élément `L[i]` dans la partie triée (les éléments à sa gauche, d'indices 0 à i-1), il faut décaler vers la droite les éléments (de la partie déjà triée) strictement supérieurs à `L[i]`, puis recopier `L[i]` à la bonne place.
   Pour cela,
   - on sauvegarde la valeur `L[i]` dans une variable `valeur`,
   - puis on regarde un par un les éléments de la partie déjà triée (de la droite vers la gauche) : si l'élément regardé (d'indice `ind`) est plus grand que `valeur`, on le recopie sur son voisin de droite, puis on recommence avec l'élément à sa gauche, etc. (en décrémentant `ind`).
   - dès que l'élément regardé n'est pas plus grand que `valeur`, ou s'il n'y a plus d'élément à regarder (cas où `ind` < 0), on recopie `valeur` à l'indice `ind`+1.

   (a) Faire fonctionner 'à la main' (sur papier) l'algorithme décrit ci-dessus sur la liste `L=[-5,1,2,5,1,-8,7]` avec `i=4`, puis `i=5`, puis `i=6`.

   (b) Écrire une fonction **`decale_et_insere(L,i)`** qui insère l'élément L[i] à la bonne place dans la partie (déjà triée) à sa gauche, en suivant l'algorithme ci-dessus. (rappel : cette fonction n'a pas de return, elle modifie directement la liste L passée en paramètre)

   (c) Écrire une fonction **`triInsertion(L)`** qui trie la liste `L` en insérant successivement les éléments d'indices `1` à `n-1` (grâce à la fonction précédente).

2. **Le tri par sélection.**
Principe : la liste triée se construit progressivement, de gauche à droite, en suivant l'algorithme suivant :
- Parcourir la liste entière à la recherche **du plus petit élément**, le mettre à la place d'indice 0 qui est sa place définitive ;
- Parcourir les **éléments restants** (indices de 1 à $n-1$) pour trouver le plus petit d'entre eux, le mettre à la place d'indice 1, *etc.*

   La liste d'origine est modifiée à chaque itération par un échange entre le plus petit élément trouvé et l'élément qu'il vient déloger, afin de ne pas perdre cet élément non encore trié.

   (a) Coder une fonction `indice_du_plus_petit(liste,deb)` qui renvoie l'indice du plus petit élément présent dans la partie non encore triée, c'est-à-dire parmi les éléments d'indices `deb` à `N-1`.

   (b) Coder une fonction `TriSelection(L)` qui, pour chaque indice `i` de `0` à `N-2`, échange l'élément d'indice `i` avec le plus petit élément trouvé dans la sous-liste qui commence à l'indice `i`.
   Pourquoi s'arrête-t-on à `N-2` ?

3. **Le tri à bulles**.
Principe : on parcourt la liste de gauche à droite, de l'indice `i = 0` à `i = N - 2`, en intervertissant toute paire d'éléments consécutifs (`L[ i ]`, `L[ i + 1 ]`) qui n'est pas dans le bon ordre : c'est la première "passe" du tri.
On recommence ensuite pour les indices de `0` à `N - 3` (deuxième passe),
puis de `0` à `N - 4` (troisième passe) ...

   On remarque en effet qu'après la 1ère passe, le plus grand élément est rangé à la bonne place (indice `N - 1`) , qu'après la 2ème passe le 2ème plus grand élément est également à la bonne place (indice `N - 2`), etc.

   (a) Écrire une fonction python **`echange_voisins`**, de paramètres `L` et `k`, qui teste l'ordre des éléments de chaque paire de la forme (`L[ i ]`, `L[ i + 1 ]`) et intervertit ces deux éléments lorsque `L[ i ] > L[ i + 1 ]`, tant que `i < k` .

   (b) Écrire une fonction **`triBulle`** de paramètre une liste `L`, qui la trie en appelant la fonction précédente pour $k = N-1, N-2, \cdots, 1$.

4. Comparons : **tri insertion** *vs.* **tri à bulles** *vs.* **tri sélection**

• Ces trois algorithmes de tri utilisent des méthodes différentes. A priori, ils ne mettent pas le même temps pour trier une liste donnée. Nous allons comparer expérimentalement l'efficacité des tris insertion, sélection et bulles.

• Pour générer des listes à trier, on utilisera la fonction random de la bibliothèque random. L'appel de random() renvoie un nombre de type flottant (réel) dans l'intervalle [0, 1[suivant une loi uniforme. On peut créer une liste de longueur N directement 'en compréhension' par l'instruction : L=[random() for i in range(N)]

• Écrire une fonction tempsTri(L,f) qui prend en paramètres une liste et le nom f de la fonction de tri à chronométrer.
Elle doit créer une copie de la liste puis trier cette copie et renvoyer le temps de tri chronométré.
On utilisera pour les chronométrages la fonction default_timer( ) du module timeit, qui renvoie un nombre correspondant à l'heure qu'il est : ainsi, par différence entre deux temps relevés, on peut mesurer la durée du tri.

• Tester avec un programme principal bref : il doit créer une liste de N valeurs aléatoires, puis afficher (print) et stocker dans une liste (append) les temps obtenus avec le tri insertion pour N = 2000, 4000, 8000, 16000. Observer... commentaire ?

• Modifier (Save As ...) le programme principal pour qu'il chronomètre les 3 tris codés pour ces valeurs de N, et représente les temps en fonction de N sur le même graphique matplotlib.

• Modifier à nouveau (Save As...) pour réaliser un tracé en échelle logarithmique, c'est-à-dire log(temps) en fonction de log(N).
Que constate-t-on ?

5. **Rappel sur les fonctions récursives.**
Une **fonction récursive** est une fonction **qui contient un appel à elle-même**.
Quelques exemples (dont un à compléter) :

5.a) la fonction factorielle récursive

```
def factorielle(n) :
    if n==0 :     # cas de base
        resu = 1
    else :
        resu = n*factorielle(n-1)    # appel à elle-même
    return resu
```

Déroulons les événements suite à un appel à factorielle(3) :

— n = 3 ⇒ resu ← 3*factorielle(2) : nouvel appel à factorielle.
— n = 2 ⇒ resu ← 2*factorielle(1) : encore un nouvel appel
— au 4ème appel, n == 0 ⇒ resu ← 1 et comme il n'y a pas de nouvel appel (**cas de base**), un enchaînement de return se met en place.

Ces return se succèdent jusqu'au return de la fonction factorielle appelée en premier, et c'est ce dernier return qui fournit le résultat :
• le 1er return exécuté (celui du 4eme et dernier appel) renvoie 1.
• le 2eme return renvoie une autre variable resu locale à la fonction n°3 : 1*1.
• le 3eme return renvoie une variable resu locale à la fonction n°2.
Remarque : Python gère (empile) les variables locales successives et ne les confond pas. Ici resu vaut 2*1.
• le quatrième et dernier return va renvoyer un resu local à la fonction appelée en n°1 et qui vaudra 3*2 : c'est le résultat de factorielle(3)

N.B. : pour toute fonction récursive, il **faut s'assurer que les appels ne s'enchaînent pas à l'infini**. Le **cas de base** a donc une grande importance.

5.b) Calcul récursif des coefficients du binôme $\binom{n}{p}$

La fonction (une fois complétée !) réalise un calcul récursif des coefficients du binôme, grâce à la propriété (relation de Pascal) $\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$:

```
def binom(n,p) :
    """calcule récursivement les coefficients du binôme"""
    if _ _ _ _ :        # 1er cas de base : si n et p sont égaux
        return _
    if _ _ _ _ _ _ _ _ :        # 2ème cas de base : si p est nul
        return _
    return binom( _ _, _ _) + binom( _ _, _ _)
```

Compléter la fonction et tester. Par exemple binom(8,4) doit renvoyer 70.

5.c) Un arbre fractal construit par récursivité

Avant de faire tourner le script, deviner dans quel ordre vont se tracer les différents segments, en tenant compte des appels récursifs. Modifier la fonction pour obtenir des arbres plus ou moins ramifiés et déséquilibrés.

```
import turtle as t # tortue graphique
def branche(longueur) : # fonction recursive
    """Dessine 1 branche et 2 plus petites, en Y. """
    if longueur > 5.0 :
        t.forward(longueur)
        t.left(25) # tourne à gauche de 25°
        branche(longueur*0.56)
        t.right(60) # tourne à droite de 60°
        branche(longueur*0.64)
        t.left(35)
        t.backward(longueur)
# programme principal
t.color("green")
t.goto(0,-169)
t.left(90)
branche(170)
```