Feuille_info_18 : Tris.

Extrait du programme de BCPST :

- Algorithmes naïfs : tri par insertion, par sélection, par comptage.
- Approfondissement possible : Tri rapide.

La programmation du tri rapide en place n'est pas un objectif du programme.
L'objectif sur les deux années est qu'un étudiant sache programmer un tri de son choix de façon autonome.

`n` désignera dans les descriptions la longueur de la liste `L`. (`n = len(L)`)

On rappelle que pour `L = [L[0], ... , L[n-1]]`,

- `L[a:b]` représente la sous-liste : `[L[a], ... , L[b-1]]`
- `L[:k]` représente le début de la liste : `[L[0], ... , L[k-1]]`
- `L[k:]` représente la fin de la liste : `[L[k], ... , L[n-1]]`
- L'instruction `L[a], L[b] = L[b], L[a]` inverse dans `L` les valeurs d'indice `a` et `b`

Les fonctions natives permettant de trier une liste : `sorted(L)` ou `L.sort()`

Des exercices imaginés dans l'esprit des exercices de l'oral de l'agro (10 mn sans préparation) :

Exercice 1 :

1) Écrire une fonction `ind_min(L, a, b)` qui prend en entrée une liste `L` et deux entiers `a` et `b` et qui renvoie l'indice dans `L` de la valeur minimale de la sous-liste `L[a:b]`

On supposera que `L` est une liste de nombres et que `0 <= a <= b < len(L)`, la fonction n'aura pas à le vérifier.

2) Si la fonction précédente n'a pas été faite avec une seule boucle, la refaire avec une seule boucle, sinon passer à la question suivante.
3) Écrire en utilisant la fonction `ind_min` une fonction `tri_select(L)` qui trie dans l'ordre croissant la liste `L` passée en argument.

Exercice 2 :

1) Écrire une fonction `insere(L, j)` qui prend en entrée une liste `L` et un entier `j` et qui insère `L[j]` dans la sous-liste `L[0:j]` déjà triée de sorte que `L[0:j+1]` soit triée.

On supposera que `L` est une liste de nombres, que `j<len(L)` et que `L[0:j]` est triée, la fonction n'aura pas à le vérifier.

2) Si la fonction précédente a été faite avec uniquement la liste `L` (sans liste auxiliaire), la refaire avec uniquement la liste `L`, sinon passer à la question suivante.
3) Écrire en utilisant la fonction `insere` une fonction `tri_insertion(L)` qui trie dans l'ordre croissant la liste `L` passée en argument.

Exercice 3 :

On considère une liste L dont les valeurs sont prises dans l'intervalle d'entiers $[\![0; k-1]\!]$.

1) Écrire une fonction `occurrence(L, k)` qui prend en entrée une liste `L` et un entier `k` et qui renvoie la liste des nombres d'occurrences des entiers de `0` à `k-1` dans la liste `L`

On supposera que `L` est une liste de nombres et que `0 <= k < len(L)`, la fonction n'aura pas à le vérifier.

2) Si la fonction précédente n'a pas été faite avec une seule boucle, la refaire avec une seule boucle, sinon passer à la question suivante.
3) Écrire en utilisant la fonction `occurrence(L, k)` une fonction `tri_denombre(L)` qui trie dans l'ordre croissant la liste `L` passée en argument.

Exercice 4 :

1) Écrire une fonction `segmente(L, i, j)` qui à partir d'une liste, d'un indice `i` de début et d'un indice de fin `j`, réorganise les éléments entre `i` et `j-1` de sorte que :
   ❶ Tous les éléments strictement inférieurs à un pivot *(choisi comme étant `L[j-1]`)* se retrouvent avant lui.
   ❷ Tous les éléments supérieurs ou égaux au pivot se retrouvent après.
   La fonction doit renvoyer l'indice final du pivot dans la liste après réorganisation.
   Exemple : Avec la liste `L = [8, 2, 1, 7, 0, 10, 5]` l'appel `segmente(L, 0, len(L))` donnera la liste `L = [2, 1, 0, 5, 8, 10, 7]` et la fonction renverra l'indice 3

On supposera que `L` est une liste de nombres et que `0<=i<=j<len(L)` , la fonction n'aura pas à le vérifier.

2) Si la fonction précédente a été faite avec uniquement la liste `L` (sans liste auxiliaire), la refaire avec uniquement la liste `L`, sinon passer à la question suivante.
3) Écrire une fonction récursive `tri_rapide(L, i, j)` qui réorganise `L` de sorte que la sous liste `L[i:j]` soit triée dans l'ordre croissant.

1) **Tri par sélection.**

Principe pour trier la liste de nombre : `L = [L[0], ... , L[n-1]]` :

Pour k allant de 0 à $n-2$, prendre le minimum de `L[k:]` est de l'échanger avec `L[k]`.

a) Écrire une fonction `ind_min(L, k)` qui prend en entrée une liste `L` et un entier `k` et qui renvoie l'indice dans `L` de la valeur minimale de la sous-liste `L[k:]`

b) Écrire en utilisant la fonction `ind_min` une fonction `tri_select(L)` qui trie dans l'ordre croissant la liste L passée en argument.

2) **Tri par insertion.**

Principe pour trier la liste de nombre : `L = [L[0], ... , L[n-1]]` :

Pour j allant de 1 à $n-1$, on insère `L[j]` dans la liste `L[:j]` déjà triée dans l'ordre croissant.

a) Écrire une fonction `insere(L, j)` qui insére `L[j]` dans la liste `L[:j]` déjà triée dans l'ordre croissant.

b) Écrire en utilisant la fonction `insere` une fonction `tri_insertion(L)` qui trie dans l'ordre croissant la liste L passée en argument.

3) **Tri par comptage.**

On considère ici le cas d'une liste `L` de longueur n dont les valeurs sont dans l'intervalle entier $[\![0; k-1]\!]$.

On commence par construire la liste des occurrences des entiers `i` de $[\![0; k-1]\!]$ dans la liste `L`

Puis on reconstruit la liste `L` en rangeant les valeurs de $[\![0; k-1]\!]$ autant de fois que nécessaire.

a) Écrire une fonction `occurrence(L, k)` qui renvoie la liste des occurrences des valeurs de $[\![0; k-1]\!]$.

b) Écrire en utilisant la fonction `occurrence` une fonction `tri_comptage(L, k)` qui trie dans l'ordre croissant la liste `L` ne contenant que des veleurs de $[\![0; k-1]\!]$.

4) **Tri rapide**

On commence par placer un élément choisi arbitrairement (le pivot) à sa place définitive dans la liste.

A l'issue de cette étape la partie gauche et la partie droite de la liste sont triées par l'intermédiaire d'un appel récursif.

a) Que fait la fonction suivante ?

```
def segmente(L, i, j):
    pivot = L[j-1]
    place = i                                # place du pivot
    for k in range(i, j-1):
        if L[k] < pivot:
            L[place], L[k] = L[k], L[place]
            place += 1                       # place = i + le nombre de valeur < pivot
                                             # L[i:place] < pivot
                                             # L[place:k+1] >= pivot
    L[place], L[j-1] = L[j-1], L[place]
    return place
```

b) Que fait la fonction suivante ?

```
def tri_rapide(L, i, j):
    if 1 < j-i:
        place = segmente(L, i, j)
        tri_rapide(L, i, place)
        tri_rapide(L, place + 1, j)
```