ISM MP, Informatique
Année 2024/2025

TP Informatique 01

On importera le module suivant :

```
import numpy.random as rd
```

Tous les programmes devront être testés.

Exercice 1

Pour x, y réels, on définit la distance entre x et y par $\mathrm{d}(x,y) = |x-y|$.

1. Écrire une fonction `voisins1(L)` d'argument `L` une liste de nombres et qui renvoie la distance minimale entre deux éléments d'indices distincts de `L`. On n'utilisera pas de tri sur cette liste.
2. Déterminer la complexité temporelle de `voisins1`. On notera n la taille de `L`.
3. En commençant par trier la liste avec l'instruction `sorted`, écrire une nouvelle version `voisins2` de `voisins1`.
4. On rappelle que `sorted(L)` est de complexité temporelle en $\mathrm{O}(n \log n)$. Quelle version est préférable ? On pourra commencer par tester les instructions suivantes :

```
>>> L=rd.random(10000)  # génère 10000 nombres au hasard dans [0,1]
>>> voisins1(L)
>>> voisins2(L)
```

Exercice 2

1. Écrire une fonction `seq_cons(T)` d'argument une liste d'entiers `T` qui renvoie la liste des indices des premiers entiers de chaque séquence d'entiers consécutifs.
   Par exemple, pour la liste `[1,2,3,4,2,3]`, le programme renvoie la liste d'indices `[0,1,2,4]` correspondant respectivement aux séquences d'entiers consécutifs `[1,2]`, `[2,3]`, `[3,4]`, `[2,3]`.

```
>>> seq_cons([1,2,3,4,2,3])
[0, 1, 2, 4]
```

2. Écrire une fonction `seq_prem(T)` d'argument une liste d'entiers `T` qui renvoie l'indice du premier entier de la première suite d'entiers consécutifs et `False` si `T` ne contient aucune suite d'entiers consécutifs.
   Par exemple, pour la liste `[1,3,5,6,8,10,11]`, la première séquence d'entiers consécutifs est `[5,6]` et `seq_prem` renvoie l'indice de l'élément `5`. Avec la liste `[1,3,5]` qui ne contient pas de séquence d'entiers consécutifs, la fonction renvoie `False`.

```
>>> seq_prem([1,3,5,6,8,10,11])
2
>>> seq_prem([1,3,5])
False
```

Exercice 3

1. Écrire une fonction `compte(L,elt)` qui renvoie le nombre d'occurrences de `elt` dans la liste `L`.
2. Écrire une fonction `freqmax1(L)` qui renvoie l'élément le plus fréquent de la liste `L` (élément au choix s'il n'est pas unique). On n'utilisera pas de tri sur cette liste et les types `range` et `list` sont les seuls types composés autorisés ici.
3. Déterminer la complexité temporelle de `freqmax1`. On notera n la taille de la liste `L`.
4. En commençant par trier la liste avec l'instruction `sorted`, écrire une nouvelle version `freqmax2` de `freqmax1`. Les types `range` et `list` sont de nouveau les seuls types composés autorisés ici.
5. On rappelle que `sorted(L)` est de complexité temporelle en $O(n \log n)$. Quelle version est préférable ? On pourra commencer par tester les instructions suivantes :

```
>>> L=rd.randint(0,10,10000) # génère 10000 nombres au hasard dans [[0,9]]
>>> freqmax1(L)
>>> freqmax2(L)
```

6. Sans contrainte de type imposé, proposer une version `freqmax3` qui améliore les deux versions étudiées ci-avant. On précisera la complexité temporelle de cette dernière version.

Exercice 4

1. Écrire une fonction `fragmente1(L)` qui fragmente la liste fournie en argument en renvoyant une liste de listes où chacune de ces sous-listes contient tous les indexes d'un élément ayant plusieurs occurrences dans `liste` et n'utilisant que les types `range` et `list` comme types composés.
   Par exemple, l'appel `fragmente1([1,2,3])` renvoie `[[0],[1],[2]]`. En effet, chaque élément de la liste de départ est présent une unique fois donc chaque sous-liste contient son unique index.
   En revanche, l'appel `fragmente1([1,2,1,5,2])` renvoie `[[0, 2], [1, 4], [3]]` : l'entier `1` est présent aux indexes `[0,2]`, l'entier `2` est présent aux indexes `[1,4]` et l'entier `5` est présent uniquement à l'index `3`.
2. Étudier la complexité temporelle de `fragmente1`. On notera n la taille de `L`.
3. Sans contrainte de type imposé, proposé un version `fragmente2` qui améliore la version précédente. On précisera la complexité temporelle de cette dernière version.