TESTS ET INVARIANTS

Quelques fonctions python seulement utilisables pour tester votre code

- Créer une nouvelle liste triée :

```
a=sorted(B)
```

a est une nouvelle liste; cette liste est la liste B triée.

Pour traduire qu'une liste t1 est triée, il suffira d'écrire : **`t1==sorted(t1)`**

Pour traduire que deux listes t1 et t2 contiennent exactement les mêmes valeurs (t1=permut(t2)), il suffira d'écrire : **`sorted(t1)==sorted(t2)`**

- Créer une nouvelle liste avec suppression des doublons :

```
a=list(set(A))
```

Exercice 1 : Les gardiens de musée :

On considère une galerie de peinture rectiligne : par exemple, les tableaux sont placés aux positions croissantes **`LT={1,2,4,5.2,7,8.4}`**. Un gardien peut surveiller tous les tableaux de LT situés au maximum à 2m de lui à gauche et à droite. Par exemple un gardien placé à la position 6 pourrait surveiller les 2 tableaux de LT placés en 4, 5.2 et 7.

1. Préliminaires :

a) Soit la fonction **`tableau_surveille`** suivante :

```
def tableau_surveille(LG,T):
    assert True ,"PE"
    i=0
    while i<len(LG) and abs(LG[i]-T)>2 :
          i+=1
    return i<len(LG)
```

et ses tests :

```
assert tableau_surveille([0,1],0)
assert tableau_surveille([0,2,1],4)
assert tableau_surveille([2],5)==False
assert tableau_surveille([3,0],2)
assert tableau_surveille([],2)==False
```

Que fait cette fonction ? Répondre en langage naturel et si vous pouvez en langage de la logique des prédicats.

b) L'objectif de cette question est d'écrire une fonction **`bs(LG,LT)`** qui prend deux listes : LG liste des positions des gardiens et LT liste des positions des tableaux et qui renvoie un booléen vrai si chacun des tableaux de LT est surveillé par au moins un gardien de LG.

Dans cette question les 2 listes LG et LT ne sont pas nécessairement croissantes.

- Commencer par écrire un jeu de test (assert) pour la fonction **`bs`** en envisageant divers cas significatifs et particuliers.
  On écrit maintenant 2 versions de cette fonction que l'on teste avec nos asserts.

- Version impérative : appeler la fonction **`tableau_surveille`** ci-dessus dans une boucle…

- Version récursive : La spécification récursive est la suivante

bs(LG, LT)=
Si LT est vide : True
Sinon Si LG est vide : False
Sinon : LT[0] est surveillé par LG[0] ou par un gardien de LG[1:]
et
les tableaux de LT[1:] sont surveillés par un gardien de LG

2. On se place maintenant dans le cas d'une liste **LT non vide et strictement croissante de positions de tableaux.** On cherche où placer les gardiens pour pouvoir surveiller tous les tableaux en minimisant le nombre de gardiens. On souhaite mettre en œuvre **un algorithme itératif glouton** de résolution de ce problème. Pour cela répondre aux questions suivantes

La spécification en triplet de Hoare est :

#PE: len(LT)>0 and (LT, <)

#musee(LT,lg)

#PS: tous les tableaux sont protégés par un gardien. (on ne traduit pas qu'ils sont en nombre minimal)

1) Écrire le PS en utilisant la fonction **bs**
2) Pour respecter l'aspect glouton, on choisit de placer le premier gardien 2m après le premier tableau. Puis on place un nouveau gardien de telle sorte que le premier tableau non surveillé soit en début de zone de surveillance du nouveau gardien. On s'arrête lorsque tous les tableaux sont surveillés. Calculer le résultat pour la liste `LT={1,2,4,5.2,8.4}` ?
3) Écrire un jeu de tests de la fonction demandée **musee**
4) L'algo s'écrit :

Initialisations

INV : les gendarmes déjà placés surveillent les tableaux déjà traités

Tant qu'il reste un tableau à traiter : CC

(corps)

si le tableau à traiter n'est pas protégé :

placer un nouveau gendarme (choix glouton)

passer au tableau suivant

Compléter le code (...) qui suit dans l'ordre : **CC , Initialisations , corps**

```
def musee (LT) :
    assert len(LT)>0 and LT==sorted(list(set(LT))) , " PB PE"
    N=len(LT)
    i=...       #indice du dernier tableau traité
    lg=[...]    #choix glouton
    j=...       #indice du dernier gendarme placé
    #INV : les gendarmes de lg[:j+1] déjà placés surveillent les tableaux de LT[:i+1]
   assert bs(lg[:j+1], LT[:i+1]) , " PB init"
    while   .... :
        #INV et CC : les gendarmes de lg[:j+1] déjà placés surveillent les tableaux de LT[:i+1]   et CC
        if ... :
            lg.append(...)   #placer un nouveau gendarme (choix glouton)
          j+=1
        i+=1  #passer au tableau suivant
        #INV : les gendarmes de lg[:j+1] déjà placés surveillent les tableaux de LT[:i+1]
        assert bs(lg[:j+1], LT[:i+1] ) , " PB fin d'itération"
    #INV et non(CC)
    assert bs(lg[:j+1], LT[:i+1]) and i==N-1 , " PB sortie de boucle"
    #PS
    assert bs(lg,LT) , " PB PS"
   return lg
```

Exercice 2: Un exemple de tri quadratique, le tri par insertion:

Dans cet exercice, on note T une liste de **N>1** entiers.

On parcourt la liste et on insère l'élément courant dans la partie déjà triée à gauche, le reste de la liste n'est pas modifié. Au début, le premier élément est bien placé car le slice T[0 :1] n'a qu'un seul élément donc est déjà trié. On peut visualiser ce tri ici : animation tri du jeu de cartes.

Pour implémenter ce tri, on commence par une fonction nommée **insertion (t,K)** qui prend en entrée :

- un indice K vérifiant 0<K<N
- une liste T de N nombres **telle que le slice T[:K] est déjà trié**

et qui insère la valeur T[K] dans la partie déjà triée T[:K] de sorte que, après l'exécution de la fonction, le slice t[:K+1] soit trié.

La spécification en triplet de Hoare est :

#PE: N>1 ET 0<K<N et (T[:K],<=)

#insertion(t,K)

#PS: (t[:K+1],<=) et t[:K+1]=permut(T[:K+1])

1) Envisager divers cas significatifs ou particuliers et écrire un jeu de tests de la fonction **insertion(t,K)**

2) Dans cette question on va écrire la fonction **insertion,** la version demandée prendra la forme d'une seule boucle While . (Dans **insertion,** on modifie la liste et on ne renvoie rien).

Pour cela, répondre successivement aux questions suivantes pour compléter l'algorithme commenté ci-dessous :

a) Écrire une fonction booléenne correspondant à l'invariant : **inv(j,K,t,T,x)**

b) remplacer les 5 lignes : **1 2 3 4 5** par des asserts utilisant si nécessaire la fonction **inv**

c) compléter les lignes manquantes **...** (en tout 4 affectations)

d) tester .

```
def insertion( t, K ) :
    N=len(t)
    T=t[:]  #T est une copie utilisée pour garder les valeurs initiales
    #PE 1
       ...
       ...
    #INV: 0<=j<=K et  x==T[K] et t[:j]==T[:j] et  (t[j:K+1], <=)  ← INIT 2
     while j!=0 and t[j-1]>x:
            #INV et CC
            ...
            ...
            #INV   ← FIN d'ITERATION 3
    #INV ET   (j==0 ou t[j-1]<=x)  ← SORTIE DE BOUCLE 4
    t[j]=x
    #PS 5
```

3) Écrire ensuite la fonction **tri_insertion(t)** qui renvoie la liste t triée : il s'agit d'une seule boucle for dans laquelle vous appelez **insertion**.

Les tests de tri de listes sont faciles en python grâce à la fonction sorted() qui renvoie une liste triée en conservant la liste initiale. La fonction **test_tri** fournie ci-dessous permet de tester tous les algorithmes de tri que vous aurez l'occasion d'écrire cette année.

Elle prend en paramètre **le nom de la fonction** de tri à tester **tri_a_tester** et produit un affichage en cas de problème :

```
from copy import deepcopy
from random import randrange

#E : une fonction de tri : tri_a_tester
#génère 1000 listes et vérifie qu'elles sont bien triées par tri_a_tester
#S : affichage d'un message

def test_tri( tri_a_tester ) :

    pb=False
    for i in range(1000) : #nombre de tris
        nb=randint(2,10) #longueur de la liste a trier
        #génération d'une liste aléatoire de nb nombres de [-10,10]
        T=[randint(-10,10) for _ in range(nb)]
        t=deepcopy(T)
        if sorted(T) != tri_a_tester(t) : #liste triée avec la fonction à tester
                pb=True
                print('Pb :')
            print(T)
            print(t)

    if not(pb) :

            print( "Ok")

#.....

#appel de la fonction test_tri pour le tri_insertion

test_tri(tri_insertion)
```

Attention : Pour que la fonction test_tri fonctionne, la fonction tri_a_tester doit prendre pour seul argument la liste à trier et renvoyer la liste triée !