ACADÉMIE
DE REIMS
Liberté
Égalité
Fraternité

NUMERIQUE
et
SCIENCES
INFORMATIQUES

Épreuve de l'enseignement de spécialité

Sujet d'entraînement

Partie Pratique

Classe Terminale de la voie générale

Le candidat doit traiter les 2 exercices

Le sujet comporte 4 pages

Exercice 1 (4 points)

Écrire une fonction `tri_insertion` qui prend en paramètre une liste `tab` de nombres entiers et qui renvoie le tableau trié par ordre croissant.

On utilisera l'algorithme suivant :

- Le premier élément de la liste à trier est considéré comme formant une liste à un élément triée,
- Le deuxième élément de la liste initiale est placé à sa place ordonnée dans le sous-tableau déjà trié : ces deux éléments forment désormais un sous-tableau trié à deux éléments,
- Le troisième élément de la liste initiale est rangé à sa place ordonnée dans le sous-tableau déjà trié : ces trois éléments forment désormais un sous-tableau trié à trois éléments,
- ...
- Le dernier élément de la liste initial est rangé à sa place dans le sous-tableau déjà trié : on a désormais trié la liste complètement.

Ainsi, à chaque étape la liste est découpée est deux sous-tableaux :

- un premier trié où tous les éléments sont déjà positionnés par ordre croissant,
- un second non encore trié où tous les éléments ont conservé leur place initiale ; c'est le premier élément de ce sous-tableau que l'on insère dans le sous-tableau déjà trié. Suite à des décalages successifs, on insère cet élément de sorte que le sous-tableau déjà trié reste trié par ordre croissant après ajout de cet élément.

Le processus est répété jusqu'à ce que le sous-tableau des éléments non encore triés soit vide.

Ci-dessous un schéma de la situation pendant le processus de tri par sélection :

```
            élément à insérer dans le sous-tableau trié
                      |
                      v
 ---------------------------------------------
| sous-tableau trié | sous-tableau non trié  |
 ---------------------------------------------
```

Exemples :

```
>>> tri_selection([2,1,0])
[0,1,2]
>>> tri_selection([2])
[2]
>>> tri_selection([5,2,6,8,1,2,8,3])
[1,2,2,3,5,6,8,8]
```

Exercice 2 (4 points)

Vous programmez un robot d'exploration. Ce robot doit se déplacer en autonomie sur une surface de sorte à se positionner en haut d'une colline depuis laquelle il pourra prendre une vue panoramique de la zone.
La zone que le robot peut parcourir est schématisée par un tableau 2D de taille $n \times n$ dans lequel chaque nombre d'une case représente la hauteur moyenne de l'endroit.
Voici un exemple de tableau modélisant une zone à explorer :

```
surface = [[ 4,6, 7, 9,10],
           [ 3,8,10,12,12],
           [14,9, 6,11,13],
           [12,9, 8,14,11],
           [ 8,6, 9,10, 6]]
```

Le robot ne peut se déplacer que dans 4 directions : haut `'H'`, bas `'B'`, gauche `'G'` et droite `'D'`. Ces déplacements reviennent aux déplacements suivant sur les cases de la surface :

- haut revient à passer si possible de ligne i à celle i-1 en restant sur la même colonne,
- bas revient à passer si possible de ligne i à celle i+1 en restant sur la même colonne,
- gauche revient à passer si possible de colonne j à celle j-1 en restant sur la même ligne,
- droite revient à passer si possible de colonne j à celle j+1 en restant sur la même ligne.

Pour se déplacer, le robot suit la méthode gloutonne suivante : il évalue parmi les au plus 4 cases voisines celle qui est qui est la plus haute.
Si cette hauteur est plus grande que celle de la case actuellement occupée, le robot monte sur cette case.
Si aucune case voisine n'est plus grande, il s'arrête sur la case actuelle (pour prendre ensuite ses photos).
Si plusieurs cases voisines ont la même plus grande hauteur, il prend une direction permettant d'atteindre une de ces cases plus hautes, peu importe laquelle.

Pour le suivi à distance du robot, ce robot stocke dans une liste `déplacements` la succession des directions prises depuis son lancement, directions modélisées par les lettres `'H'`, `'B'`, `'G'` et `'D'`.

Pour gérer le déplacement du robot suivant cette méthode, vous avez commencé à créer une fonction `grimper` qui prend en paramètre :

- une liste de listes `surface` contenant les hauteurs de la zone,
- un entier naturel `i_deb` correspondant à l'index de la ligne donnant la position de départ du robot dans la zone,
- un entier naturel `j_deb` correspondant à l'index sur cette ligne de la position de départ du robot dans la zone.

Par exemple, pour la surface précédemment énoncée, la case de hauteur `3` est celle correspondant à `i_deb=1` et à `j_deb=0`.

Cette fonction renvoie la liste `déplacements` donnant la succession des directions prises entre la position de départ et celle où le robot s'arrête.

Par exemple :

```
>>> grimper(surface,0,0)
['D', 'B', 'D', 'D']
>>> grimper(surface,2,4)
[]
>>> grimper(surface,4,4)
['H', 'G']
```

Compléter la fonction `grimper` commencée ci-dessous :

```
def grimper(surface:list,i_deb:int,j_deb:int)->list:
    """surface est une liste de listes d'entiers de taille carrée donnant la hauteur de chaque case.
    i_deb et j_deb donne la ligne et la colonne de la case de départ du robot
    Fonction renvoyant la succession des déplacements du robot avant son arrêt."""
    longueur = len(surface)
    i_pos = i_deb  # index de la ligne donnant la position du robot
    j_pos = j_deb  # index sur cette ligne de la position du robot
    hauteur_pos = surface[i_pos][j_pos] # hauteur de la position du robot
    deplacement_possible = ...
    deplacements = []
    while deplacement_possible:
        # booléen qui informe si une case de hauteur plus grande a été trouvée parmi les 4 possibles
        trouve = False
        # étude de la case au-dessus de la position actuelle
        if i_pos>0 and surface[i_pos-1][j_pos]>hauteur_pos:
            hauteur_pos = surface[i_pos-1][j_pos]
            direction =...
            # mémorisation de la position plus intéressante
            i_temp = i_pos-1
            j_temp = j_pos
            trouve = ...
        # étude de la case en dessous de la position actuelle
        if i_pos... and surface[i_pos+1][j_pos]>hauteur_pos:
            hauteur_pos = surface[i_pos+1][j_pos]
            direction ='B'
            i_temp = i_pos-1
            j_temp = j_pos
            trouve = True
        # étude de la case à droite de la position actuelle
        if ... and surface[...][...]>hauteur_pos:
            hauteur_pos = surface[...][...]
            direction ='D'
            i_temp = ...
            j_temp = ...
            trouve = True
        # étude de la case à gauche de la position actuelle
        if ... and surface[...][...]>hauteur_pos:
            hauteur_pos = surface[...][...]
            direction ='G'
            i_temp = ...
            j_temp = ...
            trouve = True
        if ...:
            deplacement_possible = False
        else:
            deplacements.append(...)
            i_pos = ...
            j_pos = ...
    return deplacements
```