TP4: Rappels

MP2I Option SI: Informatique tronc commun

Pensez à bien tester vos fonctions au fur et à mesure !

Exercice 1.

En Python, l'instruction `assert` fonctionne comme en C : elle prend en entrée un booléen et arrête le programme en levant une erreur si c'est `False`. Elle permet de vérifier les préconditions d'une fonction, ou de faire des tests. Par exemple :

```python
def divise(a, b):
    """
    Renvoie True si a divise b. a doit être non-nul
    """
    assert a != 0
    return (b % a == 0)

# Tests
assert divise(2, 10)
assert divise(7, 21)
assert not divise(3, 10)
```

Dans la suite de l'exercice, lorsque les entrées d'une fonction doivent vérifier certaines conditions, utilisez assert pour vous en assurer.

On rappelle le principe de l'algorithme de **tri rapide** : pour trier une liste L, on choisit un élément $p \in$, et on sépare les autres éléments en deux parties : à gauche les éléments inférieurs à p, à droite les éléments supérieurs à p, et p entre les deux. On trie ensuite récursivement les deux parties.

Le cœur de cet algorithme est la procédure de **partition** qui sépare la liste en deux :

Algorithme 1 : `Partition`(T, a, b)

Entrée(s) : L une liste, $a \leq b$ deux indices valides

Sortie(s) : Un indice $k \in [\![a, b]\!]$ tel que $T[a..k-1]$ contient des éléments inférieurs à $T[k]$ et $T[k+1..b]$ contient des éléments strictement supérieurs à $T[k]$

```
1  i ← a + 1;
2  s ← b;
3  tant que i ≤ s faire
4      si T[i] ≤ T[a] alors
5          i ← i + 1;
6      sinon
7          Échanger T[i] et T[s];
8          s ← s − 1;
9  Échanger T[a] et T[i − 1];
10 retourner i − 1
```

Q1. Appliquer l'algorithme sur la liste $L_0 = [5, 3, 9, 2, 7, 5, 8, 3]$ et vérifier que la sortie satisfait la spécification proposée.

Q2. Justifier que la suite des valeurs prises par $s - i + 1$ est positive, entière, strictement décroissante.

Q3. Majorer le nombre de tours de boucles que peut faire l'algorithme, et en déduire la complexité asymptotique par rapport à a et b.

Q4. Notons T_k, i_k, s_k les valeurs de T, i, s après k tours de boucles. Montrer par récurrence l'invariant suivant :

$$\forall j \in [\![s_k + 1, b]\!], T_k[j] > T[a]$$

Q5. Proposer un invariant similaire sur i_k, et en déduire que l'algorithme est correct.

On peut alors décrire facilement le tri rapide :

Algorithme 2 : `TriRapide`(T, a, b)

Entrée(s) : L une liste, a, b deux indices valides
Sortie(s) : L a été triée dans l'ordre croissant entre a et b (ne fait rien si $a > b$)

```
si a ≥ b alors
    // Un élément ou moins à trier: rien à faire
    retourner
k = Partition(T, a, b);
TriRapide(T, a, k − 1);
TriRapide(T, k + 1, b);
```

Q6. Implémenter le tri rapide en Python, et le tester pour trier quelques listes.

Exercice 2.

Dans cet exercice, essayez de mettre le plus d'assertions possible au début de chaque fonction, pour en vérifier les hypothèses.

On considère une grille G de dimensions $n \times n$ représentant une carte topographique. Chaque case $G[i][j]$ contient un entier indiquant son altitude. Des randonneurs se promenant dans cette grille peuvent se déplacer dans les 4 directions cardinales, mais pas en diagonale, et ne peuvent passer d'une case à une autre que si la différence d'altitude est comprise entre -1 et 1.

Par exemple, considérons la grille G_0 suivante, définie dans le fichier `randonnee.py` de l'archive :

0	1	4	4	4	4	2
6	2	2	0	4	5	2
6	3	2	2	3	5	4
6	2	4	5	1	6	5
0	5	4	6	5	5	5
0	3	3	0	0	4	4
0	1	2	0	0	3	2

FIGURE 1 – Grille G_0

Q1. A la main, déterminer un chemin valide entre la case en haut à gauche $G_0[0][0]$ et celle en bas à gauche $G_0[6][0]$.

Q2. Écrire une fonction `positions_valides(G, i, j)` qui prend en entrée une grille G et une position (i, j), et renvoie la liste des positions voisines valides sur lesquelles on peut se rendre à partir de la case (i, j). Par exemple, sur la grille G_0 précédente, la case $(2, 6)$ contient un 4 : les positions valides adjacentes sont $(2, 5)$, à sa gauche, et $(3, 6)$ en dessous, car elles contiennent des 5. la case au dessus, $(1, 6)$, n'est pas valide car elle contient 2. De plus, on considère comme valide le fait de ne pas bouger. Ainsi, `positions_valides(G_0, 2, 6)` renverra la liste `[(2, 6), (2, 5), (3, 6)]`.

Q3. Écrire une fonction `chemin_valide(G, ch)` qui prend en entrée une grille G et une liste ch de positions, qui détermine si le chemin est valide.

On considère des randonneurs perdus et éparpillés, qui se déplacent aléatoirement. Lorsque deux randonneurs se rencontrent, ils font équipe et se déplacent ensemble. On se demande combien de temps il faut, en moyenne, pour que les randonneurs soient tous réunis.

On rappelle les opérations possibles sur les dictionnaires :

— `dict()` crée un dictionnaire vide

— `d[k] = v` associe à la clé k la valeur v dans le dictionnaire d

— `k in d` renvoie un booléen disant si k est une clé valide dans le dictionnaire d

— `d[k]` renvoie la valeur associée à k dans le dictionnaire d (cette opération nécessite que k soit une clé valide de d)

— `del d[k]` supprime la clé k du dictionnaire d

— `for k in d:` permet de boucler sur toutes les clés de d

On représente les positions des groupes de randonneurs par un dictionnaire, dont les clés sont les noms des groupes, et dont les valeurs sont les positions. Par exemple, supposons qu'Olivier se trouve seul en $G[3][8]$, qu'Adeline et Maxime se trouvent ensemble en $G[12][47]$, et que Guillaume soit seul en $G[25][14]$. Alors, le dictionnaire des positions sera :

```
positions = {
  "Olivier": (3, 8),
  "Adeline_Maxime": (12, 47),
  "Guillaume": (25, 14)
}
```

De telle sorte qu'écrire `positions["Guillaume"]` renvoie le couple $(25, 14)$.

Q4. Écrire une fonction `regrouper(positions, g1, g2)` qui prend en entrée un dictionnaire de positions, et deux noms de groupes g_1, g_2 supposés être à la même position, et :

1. supprime le groupe g_1 et le groupe g_2
2. crée un nouveau groupe à la position commune de g_1 et g_2, dont le nom est obtenu en mettant un "_" entre les noms de g_1 et g_2.

Par exemple, les groupes `"Olivier"` et `"Adeline_Maxime"` fusionneraient en un groupe `"Olivier_Adeline_Maxime"`.

Q5. Écrire une fonction `trouver_fusion(positions)` qui prend en entrée un dictionnaire de positions contenant au moins deux groupes, et renvoie deux groupes se situant à la même position. La fonction renverra `("", "")` si tous les groupes sont à des positions distinctes.

Q6. Que fait la fonction suivante ? Lui donner un meilleur nom :

```
def mystere(positions):
    g1, g2 = trouver_fusion(positions)
    while (g1, g2) != ("" , ""):
        regrouper(positions, g1, g2)
        g1, g2 = trouver_fusion(positions)
```

On rappelle l'existence de la fonction `randint(a, b)` du module `random`, qui renvoie un entier aléatoire entre a et b **inclus**.

Q7. Écrire une fonction `deplacer_groupe(G, positions, g)` qui prend en entrée une grille G, un dictionnaire de positions et un nom de groupe g, et qui déplace aléatoirement g sur une case valide adjacente.

Q8. Écrire une fonction `simuler(G, positions)` qui fait avancer les groupes de randonneurs aléatoirement jusqu'à ce qu'il n'y ait plus qu'un seul groupe, et renvoie le nombre d'étapes de simulation qui ont été nécessaires.

Q9. Vérifier qu'avec les positions de départ suivantes dans G_0, les randonneurs prennent environ $300 \sim 320$ étapes à se rejoindre en moyenne :

```
positions = {
  "Guillaume": (0, 0),
  "Adeline": (6, 0),
  "Olivier_Maxime": (3, 6),
}
```

Attention : la fonction simuler ***modifie*** *le dictionnaire des positions, il faut le réinitialiser avant de relancer une nouvelle simulation.*