Métropole – 2024 – sujet1 - Correction

Exercice 1 (6 points)

---

1. Expliquer la ligne 9 du code

Ligne 9 : `s2.predecesseurs = []`

Cette ligne signifie que le site `s2` n'a aucun prédécesseur, c'est-à-dire qu'aucun autre site ne contient de lien hypertexte vers `site2`.

👉 *Commentaire : un prédécesseur est un sommet ayant une arête orientée vers le sommet étudié.*

---

2. Compléter les lignes 11 et 12

On cherche les prédécesseurs de `s4` et `s5`.

- `s4` reçoit des liens de :
  - `s1` (1 lien)
  - `s2` (2 liens)
- `s5` reçoit des liens de :
  - `s1` (3 liens)
  - `s3` (3 liens)
  - `s4` (6 liens)

Code complété :

```
s4.predecesseurs = [(s1,1), (s2,2)]
s5.predecesseurs = [(s1,3), (s3,3), (s4,6)]
```

👉 *Commentaire : on lit les prédécesseurs en regardant les successeurs des autres sommets.*

---

3. Valeur de l'expression `s2.successeurs[1][1]`

On a :

```
s2.successeurs = [(s1,4), (s3,5), (s4,2)]
```

- `s2.successeurs[1]` → `(s3,5)`
- `[1]` → deuxième élément du tuple → `5`

Valeur : **5**

👉 *Commentaire : il s'agit du nombre de liens de site2 vers site3.*

---

4. Popularité de `site1`

La popularité correspond au **nombre total de liens pointant vers le site**.

Les prédécesseurs de `s1` sont :

```
[(s2,4), (s4,2)]
```

Donc popularité :

4 + 2 = **6**

👉 *Commentaire : on additionne les poids des arêtes entrantes.*

---

5. Méthode `calculPopularite`

```python
def calculPopularite(self):
    total = 0
    for site, poids in self.predecesseurs:
        total += poids
    self.popularite = total
    return self.popularite
```

👉 *Commentaire : on parcourt la liste des prédécesseurs et on additionne les poids.*

---

6. Structure de données utilisée

On utilise :

```
listeS.append(sommet)
listeS.pop(0)
```

Cela correspond à une **file (FIFO)**.

👉 *Commentaire : append ajoute en fin, pop(0) enlève au début → premier entré, premier sorti.*

---

7. Nom du parcours

Il s'agit d'un **parcours en largeur** (BFS).

👉 *Commentaire : l'utilisation d'une file caractérise le parcours en largeur.*

---

8. Valeur de `parcoursGraphe(s1)`

En parcours en largeur à partir de `s1`.

Ordre des successeurs :

- `s1` → `s3`, `s4`, `s5`
- `s3` → `s5`
- `s4` → `s1`, `s5`
- `s5` → `s3`

En tenant compte des couleurs (un sommet traité devient noir et n'est plus revisité), on obtient :

[s1, s3, s4, s5]

👉 *Commentaire : s2 n'est jamais atteint car aucun chemin ne mène de s1 vers s2.*

---

9. Compléter la fonction `lePlusPopulaire`

Lignes 6 et 7 :

```
maxPopularite = site.popularite
siteLePlusPopulaire = site
```

Fonction complète :

```python
def lePlusPopulaire(listeSites):
    maxPopularite = 0
    siteLePlusPopulaire = listeSites[0]
    for site in listeSites:
        if site.popularite > maxPopularite:
            maxPopularite = site.popularite
            siteLePlusPopulaire = site
    return siteLePlusPopulaire
```

👉 *Commentaire : on met à jour le maximum lorsqu'on trouve une popularité plus grande.*

---

10. Valeur de `lePlusPopulaire(parcoursGraphe(s1)).nom`

On a parcouru :
[s1, s3, s4, s5]

Popularités :

- s1 = 6
- s3 = 3 + 5 + 4 = 12
- s4 = 3
- s5 = 3 + 3 + 6 = 12

Le premier maximum rencontré est **s3**.

Valeur renvoyée : **"site3"**

👉 *Commentaire : en cas d'égalité, la fonction conserve le premier maximum rencontré.*

---

11. Adaptation à plusieurs milliers de sites

Oui, ce code est globalement adapté.

- Le parcours en largeur est en complexité **O(n + m)**
  (n = nombre de sommets, m = nombre d'arêtes).
- La recherche du maximum est en **O(n)**.

Ces complexités sont acceptables pour quelques milliers de sites.

Cependant :

- `pop(0)` sur une liste Python est en **O(n)**, ce qui ralentit le parcours. En effet, lorsqu'on supprime le premier élément, il faut décaler tous les autres éléments de la liste.
- Il serait préférable d'utiliser une structure `deque` du module `collections`.

👉 *Commentaire : le point faible vient de l'implémentation de la file avec une liste classique.*
