# Exercice 1 (6 points)

Correction

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | Structures de données, interface et implémentation - POO - Graphes | balise12 = Balise(12, ['vert', 'noir']) |
| 2 | 1 | Structures de données, interface et implémentation - POO - Graphes | balise9.voisines = [balise8, balise11, balise12] |
| 3 | 1 | Structures de données, interface et implémentation - POO - Graphes - Tableau donné en compréhension | [2, 5, 6] |
| 4 | 1 | Structures de données, interface et implémentation - POO - Graphes - Tableau donné en compréhension | ['noir', 'vert'] |
| 5 | 2 | Algorithmes sur les graphes, paradigmes de programmation | 4 while balise.num\_balise != balise\_fin.num\_balise: 5 for b in balise.voisines:6 if (couleur in b.couleurs\_balise) and (b not in chemin): 7 balise = b 8 chemin.append(balise) |
| 6 | 2 | Algorithmes sur les graphes | 1 – 2 – 4 – 5 – 10 – 7 – 6 – 3 – 11 – 9 – 8 – 12 |
| 7 | 1 | p-uplets, tableau indexé | balise4.voisines = [(balise2, 13), (balise5, 21), (balise6, 15)] |
| 8 | 2 | Constructions élémentaires, paradigmes de programmation | 5 (numéro de la balise la plus proche de la balise numéro 10) |
| 9 | 2 | Algorithmes sur les graphes - Algorithmes gloutons | 1 – 2 – 4 – 6 – 11 – 9 – 12 |
| 10 | 3 | Algorithmes sur les graphes - Algorithmes gloutons |  5 while balise\_fin not in chemin: 6 prochaine = mystere(balise)  7 if prochaine != None: 8 prochaine.visitee = True 9 chemin.append(prochaine) 10 balise = prochaine11 else:12 return None13 return [b.num\_balise for b in chemin] |
| 11 | 1 | Algorithmes gloutons | **Proposition A** : algorithme glouton |
| 12 | 1 | Algorithmes gloutons | Inconvénient : un tel algorithme ne fournit pas toujours la solution optimale, voire peut ne pas fournir de solution.Avantage : l’exécution de tels algorithmes se fait en des temps raisonnables car ils construisent une solution assez simple, sans effectuer trop de calculs. |

# Exercice 2 (6 points)

Correction

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | langages\_programmation | 6 |
| 2 | 1 | langages\_programmation | 1 def plateau\_init(n, m, cartes):2 shuffle(cartes)3 plateau = [] 4 for i in range(n): 5 plateau.append([cartes[i+j\*n] for j in range(m)])6 return plateau  |
| 3 | 2 | langages\_programmation |  1 def cartes\_voisines(n, m, i, j): 2 voisines = []  3 for i2 in range(i-1, i+2): 4 for j2 in range(j-1, j+2):  5 if (i2, j2) != (i, j) and \ 6 i2 in range(n) and \ 7 j2 in range(m):  8 voisines.append( (i2, j2) )  9 return voisines  |
| 4 | 2 | langages\_programmation | e1 = 30 e2 n’est pas une chaine (cartes non voisines) e3 n’est pas une chaine (on sort du tableau) |
| 5 | 2 | langages\_programmation | 1 def chaine\_evalue(plateau, chemin):2 s = 03 for i, j in chemin:4 s += plateau[i][j]5 return s |
| 6 | 2 | langages\_programmation | Seul le parcours en largeur permet d’obtenir la chaine la plus courte possible. |
| 7 | 2 | langages\_programmation |  1 def explore(plateau, cible): 2 n, m = len(plateau), len(plateau[0]) 3 a\_visiter = file\_init() 4 for i in range(n): 5 for j in range(m): 6 file\_ajoute(a\_visiter, [(i,j)]) 7  8 while not file\_est\_vide(a\_visiter):  9 chemin = file\_retire(a\_visiter)10 if chaine\_evalue(plateau, chemin) == cible:11 return chemin12 dernier = chemin[-1]13 i0, j0 = dernier14 for i, j in cartes\_voisines(n, m, i0, j0):15 if (i, j) not in chemin: 16 file\_ajoute(a\_visiter,17 chemin + [ (i,j) ])18 19 # Pas de solutions20 return None |
| 8 | 2 | langages\_programmation | on peut définir une liste chemins, à la ligne 11, on remplace le return par un chemins.append(chemin) |

# Exercice 3 (8 points)

Correction

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | clé et valeur d'un dictionnaire | enregistrement['latitude'] |
| 2 | 2 | Etude d'un algo | La fonction renvoie ('2024-06-27', '23:36:01') |
| 3 | 1 | Fonction len et structure d'une liste et d'un dictionnaire | * len(frames) renvoie 4
* len(frames[1]) renvoie 5
 |
| 4 | 3 | Écrire une fonction | def detecter\_anomalie(enregistrement): return enregistrement['altitude'] > 35000 or enregistrement['altitude']<0 |
| 5 | 3 | Écrire une fonction | def liste\_num\_serie(frames) : liste=[] for elt in frames: if elt['num\_serie'] not in liste: liste.append(elt['num\_serie']) return liste |
| 6 | 3 | Écrire une fonction | 1 def distance\_totale(dep):2 total = 03 for i in range(len(dep)-1):4 total += distance\_haversine(dep[i],dep[i+1])5 return total  |
| 7 | 1 | Création d'une instance de classe (d'un objet) | sonde623=Sonde(623, 38.38825, 27.09004, '2024-06-27', None, None) |
| 8 | 1 | insertion d'un noeud dans un ABR |  |
| 9 | 2 | Parcours infixe d'un arbre | Avec un parcours infixe de l’arbre, on obtient la liste des numéros de série triée dans l’ordre croissant. 300->500->575->623->650->700->900 |
| 10 | 2 | recherche dans un arbre |  1 def rechercher(self, numero):  2 if self.est\_vide(): 3 return None  4 if numero == self.num\_serie:  5 return self 6 elif numero < self.num\_serie: 7 return self.gauche.rechercher(numero)  8 else: 9 return self.droit.rechercher(numero) 10  |
| 11 | 1 | définition fonction récursive | La méthode rechercher est dite récursive car elle s’appelle elle-même. |
| 12 | 1 | Définition d'une clé primaire | id\_abonne est une clé primaire car c’est une donnée unique par abonné. On peut admettre que l’email peut être aussi une clé primaire |
| 13 | 1 | Définition d'une clé étrangère | num\_serie est une clé étrangère car elle établit la liaison avec l’attribut num\_serie de la relation sondes. id\_abonne est une clé étrangère car elle établit la liaison avec l’attribut id\_abonne de la relation abonnes. |
| 14 | 1 | Analyse d'une requête SQL | ‘Girard’ ‘Antoine’‘Détoile’ ‘Diane’ |
| 15 | 1 | Ecriture d'une requête SQL | INSERT INTO infosRecuperation VALUES (14,480,24,'2024/07/10',47.230,12.244) |
| 16 | 1 | Ecriture d'une requête SQL | SELECT sondes.num\_serie,abonnes.nom,infosRecuperation.date\_recupFROM infosRecuperationJOIN SondesON infosRecuperation.num\_serie=Sondes.num\_serieJOIN AbonnesON infosRecuperation.id\_abonne=Abonnes.id\_abonne |