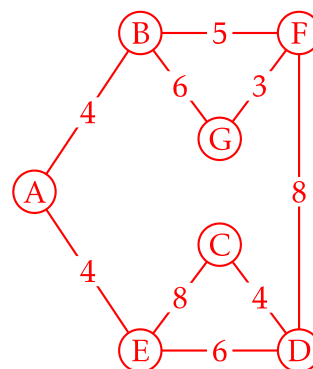
Autotest n°6 – Correction

Exercice 1 : *Cet exercice porte sur les graphes, les algorithmes sur les graphes, les bases de données et les requêtes SQL.*

La société CarteMap développe une application de cartographie-GPS qui permettra aux automobilistes de définir un itinéraire et d'être guidés sur cet itinéraire. Dans le cadre du développement d'un prototype, la société CarteMap décide d'utiliser une carte fictive simplifiée comportant uniquement 7 villes : A, B, C, D, E, F et G et 9 routes (toutes les routes sont considérées à double sens).

Graphe G1



Voici une description de cette carte :

- A est relié à B par une route de 4 km de long ;
- A est relié à E par une route de 4 km de long ;
- B est relié à F par une route de 5 km de long ;
- B est relié à G par une route de 6 km de long ;
- C est relié à E par une route de 8 km de long ;
- C est relié à D par une route de 4 km de long ;
- D est relié à E par une route de 6 km de long ;
- D est relié à F par une route de 8 km de long ;
- F est relié à G par une route de 3 km de long.

1) Représenter ces villes et ces routes ci-dessus en utilisant un graphe pondéré, nommé G1.
2) Déterminer le chemin le plus court possible entre les villes E et F.
   Solution : Le plus court chemin est E – A – B – F pour un total de 13.

3) Définir ci-contre la matrice d'adjacence du graphe G1 (en prenant les sommets dans l'ordre alphabétique). On mettra la distance entre deux sommets s'ils sont reliés et 0 sinon.

↱	A	B	C	D	E	F	G
A	0	4	0	0	4	0	0
B	4	0	0	0	0	5	6
C	0	0	0	4	8	0	0
D	0	0	4	0	6	8	0
E	4	0	8	6	0	0	0
F	0	5	0	8	0	0	3
G	0	6	0	0	0	3	0

Dans la suite de l'exercice, on ne tiendra plus compte de la distance entre les différentes villes et le graphe, non pondéré et représenté ci-dessous, sera utilisé.

Chaque sommet est une ville, chaque arête est une route qui relie deux villes.

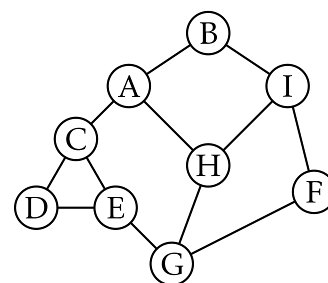


Figure 1 – Graphe G2

4) Proposer une implémentation en Python du graphe G2 à l'aide d'un dictionnaire. On pourra omettre les guillemets pour les noms des sommets.

```python
G2 = {"A": ["B", "C", "H"], "B": ["A", "I"],
      "C": ["A", "D", "E"], "D": ["C", "E"],
      "E": ["C", "D", "G"], "F": ["I", "G"],
      "G": ["E", "F", "H"],
      "H": ["A", "G", "I"],
      "I": ["B", "F", "H"]}
```

5) Proposer un parcours en largeur du graphe G2 en partant de A.
   A ; B, C, H ; I, D, E, G ; F

La société CarteMap décide d'implémenter la recherche des itinéraires permettant de traverser le moins de villes possible. Par exemple, dans le cas du graphe G2, pour aller de A à E, l'itinéraire A-C-E permet de traverser une seule ville (la ville C), alors que l'itinéraire A-H-G-E oblige l'automobiliste à traverser 2 villes (H et G).
Le programme Python suivant a donc été développé (programme p1) :

```
1  tab_itineraires=[]
2  def cherche_itineraires(G, start, end, chaine=[]):
3      chaine = chaine + [start]
4      if start == end:
5          return chaine
6      for u in G[start]:
7          if u not in chaine:
8              nchemin = cherche_itineraires(G, u, end, chaine)
9              if len(nchemin) != 0:
10                 tab_itineraires.append(nchemin)
11     return []
12
13 def itineraires_court(G, dep, arr):
14     cherche_itineraires(G, dep, arr)
15     tab_court = []
16     mini = float('inf')
17     for v in tab_itineraires:
18         if len(v) <= mini:
19             mini = len(v)
20     for v in tab_itineraires:
21         if len(v) == mini:
22             tab_court.append(v)
23     return tab_court
```

La fonction `itineraires_court` prend en paramètre un graphe `G`, un sommet de départ `dep` et un sommet d'arrivée `arr`. Cette fonction renvoie une liste Python contenant tous les itinéraires pour aller de `dep` à `arr` en passant par le moins de villes possible. Exemple (avec le graphe G2) :

```
>>> itineraires_court(G2, 'A', 'F')
[['A', 'B', 'I', 'F'], ['A', 'H', 'G', 'F'], ['A', 'H', 'I', 'F']]
```

On rappelle les points suivants :

- la méthode `append` ajoute un élément à une liste Python ; par exemple, `tab.append(el)` permet d'ajouter l'élément `el` à la liste Python `tab` ;
- en python, l'expression `['a'] + ['b']` vaut `['a', 'b']` ;
- en python `float('inf')` correspond à l'infini.

6) Expliquer pourquoi la fonction `cherche_itineraires` peut être qualifiée de fonction récursive.
   Solution : La fonction s'appelle elle-même à la ligne 8.
7) Expliquer le rôle de la fonction `cherche_itineraires` dans le programme p1.
   Solution : Elle sert à trouver tous les chemins allant de `start` à `end`.
8) Compléter la fonction `itineraires_court`.

Les ingénieurs sont confrontés à un problème lors du test du programme p1. Voici les résultats obtenus en testant dans la console la fonction `itineraires_court` deux fois de suite (sans exécuter le programme entre les deux appels à la fonction `itineraires_court`) :

```
>>> # exécution du programme p1
>>> itineraires_court(G2, 'A', 'E')
[['A', 'C', 'E']]
>>> itineraires_court(G2, 'A', 'F')
[['A', 'C', 'E']]
```

alors que dans le cas où le programme p1 est de nouveau exécuté entre les 2 appels à la fonction `itineraires_court`, on obtient des résultats corrects :

```
>>> # exécution du programme p1
>>> itineraires_court(G2, 'A', 'E')
[['A', 'C', 'E']]
>>> # exécution du programme p1
>>> itineraires_court(G2, 'A', 'F')
[['A', 'B', 'I', 'F'], ['A', 'H', 'G', 'F'], ['A', 'H', 'I', 'F']]
```

9) Donner une explication au problème décrit ci-dessus. Vous pourrez vous appuyer sur les tests donnés précédemment.

Solution : Le problème vient du fait que la liste `tab_itineraires` n'est réinitialisée qu'au début du programme p1.

Donc si on appelle plusieurs fois de suite `itinieraires_court`, qui appelle `cherche_itineraire`, les itinéraires précédents seront conservés.

Dans l'exemple, on sait que l'appel `itineraires_court(G2, 'A', 'F')` devrait renvoyer 3 chemins de longueurs 4.

Or, après l'appel à `itineraires_court(G2, 'A', 'E')`, il y a un chemin de longueur 3 (`['A', 'C', 'E']`). Les chemins de longueur 4 sont ignorés.

Il n'y a aucune vérification que les chemins de `tab_itineraires` sont des chemins correspondant bien aux deux sommets considérés.

La société CarteMap décide d'ajouter à son logiciel de cartographie des données sur les différentes villes, notamment des données classiques : nom, département, nombre d'habitants, superficie, . . . , mais également d'autres renseignements pratiques, comme par exemple, des informations sur les infrastructures sportives proposées par les différentes municipalités.
Dans un premier temps, la société a pour projet de stocker toutes ces données dans un fichier texte. Mais, après réflexion, les développeurs optent pour l'utilisation d'une base de données relationnelle.

10) Expliquer en quoi le choix d'utiliser un système de gestion de base de données (SGBD) est plus pertinent que l'utilisation d'un simple fichier texte.

Solution : Avec un SGBD, on peut assurer que l'intégrité des données. Les recherches sont également plus simples.

L'utilisation de plusieurs tables permet également de mieux gérer l'ajout ou la suppression de données.

Enfin, avec un SGBD, on peut gérer les droits de différents utilisateurs pour décider qui peut lire et qui peut modifier les données.

On donne les deux tables suivantes :

Table ville				
id	nom	num_dep	nombre_hab	superficie
1	Annecy	74	125 694	67
2	Tours	37	136 252	34.4
3	Lyon	69	513 275	47.9
4	Chamonix	74	8 906	246
5	Rennes	35	215 366	50.4
6	Nice	06	342 522	72
7	Bordeaux	33	249 712	49.4

Table sport				
id	nom	type	note	id_ville
1	Richard Bozon	piscine	9	4
2	Bignon	terrain multisport	7	5
3	Ballons perdus	terrain multisport	6	1
4	Mortier	piscine	8	2
5	Block'Out	mur d'escalade	8	2
6	Trabets	mur d'escalade	7	4
7	Centre aquatique du lac	piscine	9	2

Dans la table ville, on peut trouver les informations suivantes :

- l'identifiant de la ville (`id`) : chaque ville possède un id unique ;
- le nom de la ville (`nom`) ;
- le numéro du département où se situe la ville (`num_dep`) ;
- le nombre d'habitants (`nombre_hab`) ;
- la superficie de la ville en km^2 (`superficie`).

Dans la table sport, on peut trouver les informations suivantes :

- l'identifiant de l'infrastructure (`id`) : chaque infrastructure a un id unique ;
- le nom de l'infrastructure (`nom`) ;
- le type d'infrastructure (`type`) ;
- la note sur 10 attribuée à l'infrastructure (`note`) ;
- l'identifiant de la ville où se situe l'infrastructure (`id_ville`).

En lisant ces deux tables, on peut, par exemple, constater qu'il existe une piscine Richard Bozon à Chamonix.

11) Donner le schéma relationnel de la table `ville`. On précisera les types parmi `INT`, `TEXT`, `BOOL` et `FLOAT`, ainsi que la clef primaire.

Solution :
`ville(id: INT, nom: TEXT, num_dep: INT, nombre_hab: INT, superficie: FLOAT)`.
La clef primaire est soulignée.

12) Expliquer le rôle de l'attribut `id_ville` dans la table `sport`.

Solution : C'est une clef étrangère qui fait référence à la clef primaire de la table `ville`. Cela permet de faire les jointures.

13) Donner le résultat de la requête SQL suivante :

```sql
SELECT nom FROM ville
WHERE num_dep = 74 AND superficie > 70;
```

Solution : On obtient les noms des villes du département 74 dont la superficie est supérieure à 70 km^2. Dans les tables considérées, il n'y a que Chamonix qui correspond à ces critères.

14) Écrire une requête SQL permettant de lister les noms de l'ensemble des murs d'escalade présents dans la table `sport`.

Solution :

```
SELECT nom FROM sport WHERE type="mur d'escalade";
```

Suite à de bons retours d'utilisateurs, la note du terrain multisport "Ballons perdus" est augmentée d'un point (elle passe de 6 à 7).

15) Écrire une requête SQL permettant de modifier la note du terrain multisport "Ballons perdus" de 6 à 7.

Solution :

```
UPDATE sport SET note=7 WHERE nom = "Ballons perdus";
```

16) Écrire une requête SQL permettant d'ajouter la ville de Toulouse dans la table `ville`. Cette ville est située dans le département de la Haute-Garonne (31). Elle a une superficie de 118 km^2. En 2023, Toulouse comptait 471941 habitants. Cette ville aura l'identifiant 8.

Solution :

```
INSERT INTO ville VALUES (8, "Toulouse", 31, 471941, 118);
```

17) Écrire une requête SQL permettant de lister les noms des piscines disponibles dans le département 26.

Solution :

```
SELECT sport.nom FROM sport
JOIN ville on ville.id=sport.id_ville
WHERE type="piscine" AND num_dep=26;
```

Exercice 2 : *Cet exercice porte sur l'algorithmique, la programmation orientée objet et la méthode diviser-pour-régner.*

L'objectif de cet exercice est de trouver les deux points les plus proches dans un nuage de points pour lesquels on connaît les coordonnées dans un repère orthogonal.

On rappelle que la distance entre deux points A et B de coordonnées $(x_A ; y_A)$ et $(x_B ; y_B)$ est donnée par la formule : $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

Les coordonnées d'un point seront stockées dans un tuple de deux nombres réels.

Le nuage de points sera représenté en Python par une liste de tuples de taille n, n étant le nombre total de points. On suppose qu'il n'y a pas de points confondus (mêmes abscisses et mêmes ordonnées) et qu'il y a au moins deux points dans le nuage.

Pour calculer la racine carrée, on utilisera la fonction sqrt du module `math`, pour rappel :

```
1 >>> from math import sqrt
2 >>> sqrt(16)
3 4.0
```

1) Cette partie comprend plusieurs questions générales :

   a) Donner le rôle de l'instruction de la ligne 1 du code précédent.

   Solution : Elle sert à importer la fonction `sqrt` du module `math`.

   b) Expliquer le résultat suivant :

   Solution : Les nombres à virgules sont stockés avec un nombre fini de chiffres en Python, selon la norme de la virgule flottante. Or ces trois nombres ne peuvent pas s'écrire avec un nombre fini de chiffres après la virgule. Ce sont donc des valeurs approchées qui sont utilisées, d'où l'erreur.

```
>>> 0.1 + 0.2 == 0.3
False
```

   c) Expliquer l'erreur suivante :

   Solution : Les tuples sont des objets immuables. On ne peut pas changer un élément. Il faut créer un nouveau tuple si on veut le faire.

```
>>> point_A = (3, 4)
>>> point_A[0]
3
>>> point_A[0] = 2
Traceback (most recent call last):
File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

2) On définit la classe Segment ci-dessous :

```
1 from math import sqrt
2 class Segment:
3     def __init__(self, point1, point2):
4         self.p1 = point1
5         self.p2 = point2
6         self.longueur = ............................
```

a) Recopier et compléter la ligne 6 du constructeur de la classe `Segment`.

Solution :

```
self.longueur = sqrt((self.p2[0]-self.p2[0])**2+(self.p2[1]-self.p2[1])**2)
```

La fonction `liste_segments` donnée ci-dessous prend en paramètre une liste de points et renvoie une liste contenant les objets `Segment` qu'il est possible de construire à partir de ces points. **On considère les segments [AB] et [BA] comme étant confondus et on ajoutera un seul objet dans la liste.**

```
def liste_segments(liste_points):
    n = len(liste_points)
    segments = []
    for i in range(n-1):   # n  est aussi une réponse valide
        for j in range(i+1, n):
            # On construit le segment à partir des points i et j.
            seg = Segment(liste_points[i], liste_points[j])
            segments.append(seg)   # On l'ajoute à la liste.
    return segments
```

b) Recopier la fonction sans les commentaires et compléter le code manquant.

c) Donner en fonction de `n` la longueur de la liste segments. Le résultat peut être laissé sous la forme d'une somme.

Solution : $(n-1)+(n-2)+\ldots+2+1=\dfrac{n(n-1)}{2}$

d) Donner, en fonction de `n`, la complexité en temps de la fonction `liste_segments`.

Solution : La complexité est quadratique puisqu'on a une double boucle qui dépend de n.

3) L'objectif de cette partie est d'écrire la fonction de recherche des deux points les plus proches en utilisant la méthode diviser-pour-régner.

On dispose de deux fonctions : `moitie_gauche` (respectivement `moitie_droite`) qui prennent en paramètre une liste et qui renvoient chacune une nouvelle liste contenant la moitié gauche (respectivement la moitié droite) de la liste de départ. Si le nombre d'éléments de celle-ci est impair, l'élément du centre se trouve dans la partie gauche.

Exemples :

```
>>> liste = [1, 2, 3, 4]
>>> moitie_gauche(liste)
[1, 2]
>>> moitie_droite(liste)
[3, 4]
```

```
>>> liste = [1, 2, 3, 4, 5]
>>> moitie_gauche(liste)
[1, 2, 3]
>>> moitie_droite(liste)
[4, 5]
```

Rechercher les deux points les plus proches revient à rechercher le plus court segment parmi ceux qui ont été construits à partir du nuage de points.

Écrire la fonction `plus_court_segment` qui prend en paramètre une liste d'objets `Segment` et renvoie l'objet `Segment` dont la longueur est la plus petite. On procédera de la façon suivante :

- Tester si le cas de base est atteint, c'est-à-dire lorsque la liste contient un seul segment ;
- Découper la liste en deux listes de tailles égales (à une unité près) ;
- Appeler récursivement la fonction pour rechercher le minimum dans chacune des deux listes ;
- Comparer les deux valeurs récupérées et renvoyer la plus petite des deux.

Solution :

```
def plus_court_segment(segments):
    if len(segments) == 1:
        return segments[0]
    else:
        segments_g = moitie_gauche(segments)
        segments_d = moitie_droite(segments)
        seg_g = plus_court_segment(segments_g)
        seg_d = plus_court_segment(segments_d)
        if seg_g.longueur < seg_d.longueur:
            return seg_g
        else:
            return seg_d
```

4) On considère les trois points $A(3;4)$, $B(2;3)$ et $C(-3;-1)$.

a) Donner l'instruction Python permettant de construire la variable `nuage_points` contenant les trois points A, B et C.

Solution :

```
A = (3, 4)
B = (2, 3)
C = (-3, -1)
nuage_points = [A, B, C]
```

b) En utilisant les fonctions de l'exercice, écrire les instructions Python qui affichent les coordonnées des deux points les plus proches du nuage de points `nuage_points`.

Solution :

```
segments = liste_segments(nuage_points)
seg = plus_court_segment(segments) print(seg.p1, seg.p2)
```