CCINP 2025

1. Le titre n'est pas nécessairement unique (plusieurs randonnées peuvent avoir le même nom), il ne peut donc pas être une clé primaire. Un attribut possible pour clé primaire est `Id`.
2. L'attribut `IdAuteur` est une clé étrangère (référence à la table Auteur).

3. ```sql
   SELECT Titre, Lieu, Distance
   FROM Randonnee
   WHERE Type = 'Pied';
   ```

4. ```sql
   SELECT IdAuteur, COUNT(*) AS nb
   FROM Randonnee
   WHERE Type = 'Pied' AND Niveau = 3
   GROUP BY IdAuteur
   ORDER BY nb DESC;
   ```

5. ```sql
   SELECT Auteur.Pseudo, Randonnee.Titre
   FROM Auteur
   JOIN Randonnee ON Auteur.Id = Randonnee.IdAuteur;
   ```

6. ```sql
   SELECT Nom, Prenom
   FROM Auteur
   WHERE Id = (
       SELECT IdAuteur
       FROM Randonnee
       WHERE Type = 'Cheval'
       GROUP BY IdAuteur
       ORDER BY COUNT(*) DESC
       LIMIT 1
   );
   ```

7. ```python
   import gpxpy
   ```

8. Valeur : 105. Signification : altitude moyenne des points de l'itinéraire.
9. Complexité en $O(n)$ où n est la taille de la liste.

10. ```python
    def altitude_maximale(iti):
        alt_max = iti[0][2]
        for p in iti:
            if p[2] > alt_max:
                alt_max = p[2]
        return alt_max
    ```

11. ```python
    def denivele_global(iti):
        return altitude_maximale(iti) - iti[0][2]
    ```

12. ```python
    def denivele_positif_cumule(iti):
        s = 0
        for i in range(len(iti)-1):
            diff = iti[i+1][2] - iti[i][2]
            if diff > 0:
                s += diff
        return s
    ```

13.
```python
def alt_glissante(liste_alt, p):
    n = len(liste_alt)
    res = []
    for i in range(n):
        j = min(p, n - i)
        s = 0
        for k in range(j):
            s += liste_alt[i+k]
        res.append(s / j)
    return res
```

14. Complexité en $O(n\,p)$.

15. `lat_ref` et `long_ref` sont des listes. Elles contiennent respectivement les latitudes et longitudes des points de référence du dictionnaire.

16. `auxiliaire : (list, list) -> list` `principal : (list) -> list` Ces fonctions prennent des listes et renvoient une liste triée.

17. Il s'agit d'un algorithme récursif qui utilise la méthode diviser pour régner.

18. La fonction termine car à chaque appel récursif, la taille de la liste diminue strictement jusqu'au cas de base ($|x| \leqslant 1$).

19. Il s'agit d'un **tri fusion**.

20.
```python
ind_deb = 0
ind_fin = len(liste_ref) - 1
k = (ind_deb + ind_fin) // 2
ind_fin = k
ind_deb = k
return liste_ref[ind_fin]
return liste_ref[ind_deb]
```

21.
```python
def standardise(liste_parcours):
    res = []
    for (lat, long, alt) in liste_parcours:
        lat_r = ref(lat, lat_ref)
        long_r = ref(long, long_ref)
        alt_r = dem[(lat_r, long_r)]
        res.append((lat, long, alt_r))
    return res
```

22. La boucle parcourt tous les voisins `sommet` du sommet courant `sTraite` non encore visités. Elle conserve dans `sInter` le voisin dont le poids de l'arête (`graph[sTraite][sommet]`) est le plus faible, et mémorise ce poids minimum dans `d`.

    Il s'agit d'un **algorithme glouton** (stratégie du plus proche voisin) : à chaque étape, on choisit localement l'arête de coût minimal sans reconsidérer les choix passés.

23. On suit l'exécution pas à pas :

    - **Étape 1** – depuis `a` : voisins `b`(3), `c`(2), `d`(4). Minimum : `c` (coût 2). `sTraite = c`, `diffChemin = 2`.
    - **Étape 2** – depuis `c` : voisins non visités : `d`(4). (a est déjà visité). Minimum : `d` (coût 4). `sTraite = d`, `diffChemin = 6`.
    - **Étape 3** – depuis `d` : voisins non visités : `b`(2), `e`(1), `f`(4). Minimum : `e` (coût 1). `sTraite = e`, `diffChemin = 7`.
    - **Étape 4** – depuis `e` : voisins non visités : `b`(5), `f`(1). Minimum : `f` (coût 1). `sTraite = f`, `diffChemin = 8`.
    - `sTraite == "f"` : la boucle s'arrête.

`mystere2(G, "a", "f")` $\longrightarrow$ `(["a","c","d","e","f"], 8)`

24. **Non.** L'algorithme glouton ne garantit pas l'optimalité globale.
Contre-exemple : le chemin $a \to b \to d \to e \to f$ a pour coût $3+2+1+1=7$, ce qui est strictement inférieur à 8 (le résultat trouvé). L'algorithme a choisi `c` dès la première étape parce que l'arête (a,c) est de coût minimal, mais ce choix local conduit à un chemin global sous-optimal.

25.

Étape	sTraite	a	b	c	d	e	f	aVisiter
Init.	–	(0,a)	X	X	X	X	X	["a"]
1	a	(0,a)	(3,a)	(2,a)	(4,a)	X	X	["b","c","d"]
2	c	(0,a)	(3,a)	(2,a)	(4,a)	X	X	["b","d"]
3	b	(0,a)	(3,a)	(2,a)	(4,a)	(8,b)	X	["d","e"]
4	d	(0,a)	(3,a)	(2,a)	(4,a)	(5,d)	(8,d)	["e","f"]
5	e	(0,a)	(3,a)	(2,a)	(4,a)	(5,d)	(6,e)	["f"]

Remarque : à l'étape 3, `cherche_min` sélectionne `b` (distance $3<4$). À l'étape 4, depuis `d` on met à jour `e` : $4+1=5<8$, et `f` : $4+4=8$. À l'étape 5, depuis `e` on met à jour `f` : $5+1=6<8$. Comme `sTraite == "f"`, la boucle s'arrête après l'étape 5.

26.
```
# Ligne 8 :
while s != sInit:
# Ligne 9 :
    s = distance[s][1]
# Ligne 10 :
    chemin.append(s)
# Ligne 15 :
print("La difficulte minimale est de :", distance[sFin][0])
```

Explication : on remonte le dictionnaire `distance` en suivant les prédécesseurs depuis `sFin` jusqu'à `sInit`. On renverse ensuite la liste pour obtenir le chemin dans le bon sens.

27. Remplacer les listes `aVisiter` et `dejaVisites` par des **ensembles** (`set` Python). Le test `v not in dejaVisites` (ligne 15) et `v not in avisiter` (ligne 14) passent alors de $O(n)$ à $O(1)$ en moyenne (table de hachage). On utilise `aVisiter.discard(s)` et `dejaVisites.add(s)` pour les mises à jour.

28. Le graphe G1 est un graphe « en ligne » : $a_1 - c_1 - g_1 - j_1$ avec des ramifications b_1, d_1, e_1, f_1, h_1, i_1 (tous de poids 1, voisins triés alphabétiquement).

- **`dijkstra(G1, "a1", "j1")`** : l'algorithme explore en largeur depuis `a1`. Les sommets visités (dans l'ordre) sont : `a1, b1, c1, d1, e1, f1, g1, h1, i1, j1`.
- **`dijkstra(G1, "j1", "a1")`** : symétrique, l'algorithme part de `j1`. Les sommets visités sont : `j1, i1, g1, h1, f1, c1, e1, d1, b1, a1`.

Dans les deux cas, **tous les sommets** du graphe sont visités avant d'atteindre la destination, ce qui illustre l'intérêt de l'algorithme bidirectionnel.

29. À l'étape 3, on a $F_{\min}=2$, $B_{\min}=2$, $BF_{\min}=6$. Comme $BF_{\min} \leqslant F_{\min}+B_{\min}$ ($6 \leqslant 4$) est **faux**, on continue. $F_{\min}=B_{\min}=2$ et $|\texttt{aVisiterF}|=3>|\texttt{aVisiterB}|=2$: on choisit une étape **Backward** depuis `d` (qui réalise $B_{\min}=2$).
Depuis `d` (backward) : voisins non visités backward : `a`(4), `b`(2), `c`(4). On met à jour : $d_B(a)=\min(\infty, 2+4)=6$, $d_B(b)=\min(6, 2+2)=4$, $d_B(c)=\min(\infty, 2+4)=6$.

Tableau distanceF | distanceB – Étape 4 :

Étape	Traité	a	b	c	d	e	f
4	d, B	$(0,a)\|(6,d)$	$(3,a)\|(4,d)$	$(2,a)\|(6,d)$	$(4,a)\|(2,e)$	$(\infty,a)\|(1,f)$	$(\infty,a)\|(0,f)$

Tableau Fmin/Bmin/BFmin – Étape 4 :

Étape	Fmin	Bmin	BFmin	aVisiterF	aVisiterB
4	2	4	6	[“b”,“c”,“d”]	[“b”,“c”]

Calcul de BFmin : pour les sommets présents dans les deux dictionnaires, $d_F(v)+d_B(v)$ vaut : a : $0+6=6$, b : $3+4=7$, c : $2+6=8$, d : $4+2=6$, e : ∞, f : $\infty+0=\infty$. Donc $BF_{\min}=6$.
Condition d’arrêt : $BF_{\min}=6 \leqslant F_{\min}+B_{\min}=2+4=6$ **vrai**. L’algorithme **termine** à l’étape 4 et renvoie $BF_{\min}=6$.

30. **Non, cela ne suffit pas.** Dès qu’un sommet v est atteint par les deux recherches, on aurait $d_F(v)+d_B(v)$ comme candidat, mais le chemin passant par v n’est pas nécessairement optimal : un autre chemin passant par un sommet différent peut être plus court. La condition correcte est $BF_{\min} \leqslant F_{\min}+B_{\min}$, qui garantit qu’aucun chemin non encore exploré ne peut améliorer $BF_{\min}$.

31.
```
# Ligne 4 :
aVisiterB = [Sf]
# Ligne 5 :
dejaVisitesB = []
# Ligne 13 :
while BFmin > Fmin + Bmin:
# Ligne 21 :
Fmin = min([distanceF[v][0] for v in aVisiterF if v in distanceF])
# Ligne 22 :
Bmin = min([distanceB[v][0] for v in aVisiterB if v in distanceB])
# Ligne 23 :
L = [distanceF[v][0] + distanceB[v][0] for v in distanceB if v in distanceF]
```

32. Supposons par l’absurde qu’il existe un sommet s_i du chemin optimal $s=s_0,s_1,\ldots,s_n=t$ (de distance $d<BF_{\min}$) qui n’appartient ni à `dejaVisitesF` ni à `dejaVisitesB`.
    - Si $s_i \notin$ `dejaVisitesF`, alors s_i n’a pas encore été visité par la recherche forward. Par la propriété de Dijkstra, $d_F(s_i) \geqslant F_{\min}$. Or la distance du chemin optimal de s à s_i est $\geqslant d_F(s_i) \geqslant F_{\min}$.
    - De même, si $s_i \notin$ `dejaVisitesB`, la distance de t à s_i est $\geqslant B_{\min}$.
    - Donc $d \geqslant F_{\min}+B_{\min} \geqslant BF_{\min}$, ce qui contredit $d<BF_{\min}$.

    Ainsi tout sommet s_i du chemin optimal appartient à `dejaVisitesF` ou à `dejaVisitesB`.
33. D’après Q32, chaque s_i appartient à `dejaVisitesF` ou à `dejaVisitesB`. Considérons la séquence : $s_0=s$ est le point de départ, donc $s_0 \in$ `dejaVisitesF`. $s_n=t$ est le point d’arrivée, donc $s_n \in$ `dejaVisitesB`. La séquence commence dans `dejaVisitesF` et se termine dans `dejaVisitesB`. Il existe donc un indice $i_0 \in \{0,\ldots,n-1\}$ tel que $s_{i_0} \in$ `dejaVisitesF` et $s_{i_0+1} \in$ `dejaVisitesB` (premier indice où l’appartenance “bascule” de F vers B).
34. Par Q33, il existe i_0 tel que $s_{i_0} \in$ `dejaVisitesF` et $s_{i_0+1} \in$ `dejaVisitesB`.
    - Par correction de Dijkstra classique (admis), la distance calculée `distanceF`$[s_{i_0}][0]$ vaut $d(s,s_{i_0})$.
    - De même, `distanceB`$[s_{i_0+1}][0]$ vaut $d(t,s_{i_0+1})$.
    - Donc la valeur `distanceF`$[s_{i_0}][0]$ + `distanceB`$[s_{i_0+1}][0]$ + `graph`$[s_{i_0}][s_{i_0+1}]$ est une borne sur d. Or cette quantité est $\geqslant BF_{\min}$ (par définition de $BF_{\min}$ comme minimum sur tous les sommets).

    Ainsi $d \geqslant BF_{\min}$, ce qui contredit $d<BF_{\min}$.
    Conclusion : l’hypothèse par l’absurde est fausse. Quand l’algorithme termine, $BF_{\min}$ est bien la distance minimale reliant s à t. L’algorithme de Dijkstra bidirectionnel est donc **partiellement correct**.