Gestion de Randonnées

Q1. Expliquer en quoi l'attribut `Titre` ne peut probablement pas être une clé primaire pour la table `Randonnee`. Proposer un attribut de la table `Randonnee` qui puisse être une clé primaire.

La clé primaire doit être unique pour chaque enregistrement. Donc on ne peut pas être sûr que le titre soit unique. On peut prendre `Id` et éventuellement `Trace`

Q2. Identifier un attribut qui soit une clé étrangère de la table `Randonnee`.

Un attribut est clé étrangère s'il est clé primaire d'une autre table. Ici, `IdAuteur` est lié à la clé primaire `Id` de la table `Auteur`

Q3. Écrire une requête SQL dont l'évaluation renvoie le titre, les coordonnées GPS du point de départ et la longueur des randonnées à pied.

```
SELECT Titre, Lieu, Distance FROM Randonnee WHERE Type='Pied'
```

Q4. Écrire une requête SQL dont l'évaluation renvoie l'identifiant de l'auteur et son nombre d'activités à pied de niveau 3, classées par ordre décroissant du nombre d'activités de chaque auteur.

```
SELECT  IdAuteur, Count(*) as Activites
FROM  Randonnee
WHERE  Type='Pied' AND Niveau=3
GROUP BY IdAuteur
ORDER BY Activites DESC
```

Q5. Écrire une requête SQL dont l'évaluation renvoie le `Pseudo` de l'auteur et le `Titre` des randonnées stockées dans la base.

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

Q6. Écrire une requête SQL dont l'évaluation renvoie les nom et prénom d'un des auteurs ayant posté le plus de randonnées à cheval.

```
SELECT Auteur.nom, Auteur.Prenom
FROM Auteur
JOIN Randonnee ON Randonnee.Idauteur=Auteur.Id
WHERE Type='Cheval'
GROUP BY Auteur.Id
ORDER BY COUNT(Randonnee.Id) DESC
LIMIT 1
```

Q7. Écrire une ligne de code permettant l'importation du module `gpxpy`.

```
import gpxpy
```

Q8. Donner la valeur numérique que renvoie le code suivant. Donner la signification de cette valeur dans le contexte du sujet.

```
>>> p0 = (47.8741, 1.8758, 100)
>>> p1 = (47.8744, 1.8759, 102)
>>> p2 = (47.8748, 1.8761, 110)
>>> p3 = (47.8750, 1.8759, 108)
>>> l1 = [p0, p1, p2, p3]
>>> mystere(l1)
```

Le valeur renvoyée vaut

$$\frac{100 + 102 + 110 + 108}{4} = \frac{420}{4} = 105$$

La fonction renvoie l'altitude moyenne de la randonnée.

Q9. Donner la complexité temporelle de la fonction `mystere` en fonction de la taille n de la liste passée en argument.

La fonction effectue un unique parcours de la liste passée en argument. À chaque itération, on effectue une extraction d'un élément de la liste ainsi qu'une addition qui s'effectuent toutes deux en temps constant. La complexité est donc linéaire.

Q10. Écrire une fonction `altitude_maximale(iti:itineraire) -> float` qui, étant donné une liste non vide `iti` de points, renvoie l'altitude maximale de l'itinéraire en mètres.

```
def altitude_maximale(iti:itineraire) -> float:
    alt_max = iti[0][2]
    for i in range(1, len(iti)):
        (lat, long, alt) = iti[i]
        if alt > alt_max:
            alt_max = alt
    return alt_max
```

Q11. En utilisant la fonction `altitude_maximale`, écrire une fonction `denivele_global(iti:itineraire) -> float` qui, étant donné une liste `iti` de points, renvoie le dénivelé global de la randonnée.

```
def denivele_global(iti:itineraire) -> float:
    alt_depart = iti[0][2]
    alt_max = altitude_maximale(iti)
    return alt_max - alt_depart
```

Q12. Écrire une fonction `denivele_positif_cumule(iti:itineraire) -> float` qui, étant donné une liste `iti` de points, renvoie le dénivelé positif cumulé de la randonnée.

```
def denivele_positif_cumule(iti:itineraire) -> float:
    dp = 0
    for i in range(len(iti) - 1):
        (lat1, long1, alt1) = iti[i]
        (lat2, long2, alt2) = iti[i+1]
        if alt2 - alt1 > 0:
            dp = dp + alt2 - alt1
    return dp
```

Q13. Écrire une fonction `alt_glissante(liste_alt:list, p:int) -> list` qui étant donné une liste d'altitudes et un entier `p`, crée une nouvelle liste contenant la moyenne glissante des altitudes avec un pas `p`. On pourra utiliser la fonction `min` qui prend deux nombres flottants en entrée et renvoie le plus petit des deux.

```
1 def alt_glissante(liste_alt:list, p:int) -> list:
2     n_liste = []
3     for i in range(len(liste_alt)):
4         s, cpt = 0, 0
5         for j in range(i, min(i+p, len(liste_alt))):
6             s = s + liste_alt[j]
7             cpt = cpt + 1
8         n_liste.append(s/cpt)
9     return n_liste
```

Q14. Évaluer la complexité de la fonction `alt_glissante` en fonction de la taille n de la liste d'altitudes passée en argument et du pas p.

À chaque passage dans la première boucle, l'algorithme effectue au plus p opérations. La complexité est donc proportionnelle à $n \times p$.

Q15. Indiquer le type des variables `lat_ref` et `long_ref`. Expliquer quel est le contenu de ces variables dans le contexte de ce sujet.

`lat_ref` et `long_ref` sont des listes initialisées en ligne 1 et 2. On parcourt l'ensemble du dictionnaire `dem`. `lat_ref` et `long_ref` représentent donc les listes des latitudes et des longitudes présentes dans le dictionnaire.

Q16. Précisez la signature des fonctions `auxiliaire` et `principal`. La réponse doit être justifiée.

`auxiliaire` prend en argument 2 listes `x,y`. On le voit car ligne 2 on teste si les listes sont vides.
Elle renvoie une liste puisque en ligne 2, 3 elle renvoie `x` ou `y` et en ligne 10, elle renvoie `z`.
`principal` prend donc en argument une liste `x` car en ligne 17 et 18 on effectue une extraction.
Elle renvoie une liste `z` car la fonction `auxiliaire` renvoie des listes.

Q17. Sur le **DR**, cocher la (les) cases qui correspond(ent) au(x) type(s) de programmation utilisé(s) pour coder la fonction `principal`.

Il s'agit d'un algorithme récursif car la définition de la fonction fait appel à cette même fonction.
Il est basé sur le principe du diviser pour régner avec le partage de la liste en 2 parties.

Q18. Justifier brièvement que, étant donné une liste `x`, l'appel `principal(x)` termine.

À chaque appel récursif, la taille de la liste est divisée par 2. L'algorithme va donc finir par atteindre une liste vide ou une liste de taille 1 et tomber sur un cas de base. Ceci à condition que la fonction auxiliaire se termine également ce qui est le cas car ses appels récursifs diminuent la taille de la liste de 1 et finira par aboutir à deux listes vides ou de taille 1 et donc un cas de base.

Q19. L'appel `principal(x)` permet de renvoyer une liste triée par ordre croissant. Proposer un nom qui décrit le type de tri utilisé en justifiant brièvement votre choix.

La fonction `principal` coupe la liste en 2 sous listes puis redécompose en 2 par recursivité. C'est la partition.
La fonction `auxiliaire` reprend les termes de chaque sous listes déjà triées et construit ainsi une liste fusionnée triée. C'est la fusion.
C'est un tri fusion (ou partition/fusion).

Q20. Compléter les lignes 5, 6, 8, 10, 12, 15 et 17 de la fonction `ref`.

```
1  def ref(valeur:float, liste_ref:list) -> float:
2      # on détermine ind_deb et ind_fin tel que
3      # liste_ref[ind_deb]<valeur<=liste_ref[ind_fin]
4      # avec une méthode par dichotomie
5      ind_deb = 0
6      ind_fin = len(liste_ref)-1
7      while ind_deb < ind_fin - 1:
8          k = (ind_fin + ind_deb)//2
9          if liste_ref[k] >= valeur:
10             ind_fin = k
11         else:
12             ind_deb = k
13     # on détermine le plus proche
14     if liste_ref[ind_fin]-valeur<valeur-liste_ref[ind_deb]:
15         return liste_ref[ind_fin]
16     else:
17         return liste_ref[ind_deb]
```

Q21. Utiliser les données précédentes pour écrire une fonction `standardise(liste_parcours:itineraire) -> itineraire` qui, étant donné un itinéraire, renvoie un nouvel itinéraire où l'altitude de chaque point a été remplacée par l'altitude issue du dictionnaire `dem`.
Pour chaque point de l'itinéraire, on cherchera la latitude ϕ_r de référence la plus proche de sa latitude, la longitude λ_r de référence la plus proche de sa longitude et on remplacera son altitude par l'altitude du point de référence de coordonnées (ϕ_r, λ_r).
Les variables `lat_ref`, `long_ref` et `dem` sont définies globalement en dehors de la fonction et peuvent être utilisées directement.

```
1  def standardise(liste_parcours:list) -> list:
2      n_parcours = []
3      for i in range(len(liste_parcours)):
4          lat, long, alt = liste_parcours[i]
5          lat_plus_proche = ref(lat, lat_ref)
6          long_plus_proche = ref(long, long_ref)
7          n_alt = dem[(lat_plus_proche, long_plus_proche)]
8          n_parcours.append((lat, long, n_alt))
9      return n_parcours
```

Q22. Expliquer le fonctionnement de la boucle itérative comprise entre les lignes 15 à 19 et en déduire le nom du type d'algorithme utilisé.

On parcourt tous les sommets liés dans le graphe à `sTraite`. On détermine celui dont la difficulté de la randonnée est le plus petit. On recommence en prenant à chaque étape le sommet non visité de niveau le plus petit.

À chaque étape, on minimise la difficulté de la randonnée suivante, il s'agit d'un algorithme **glouton**.

Q23. Donner ce que renvoie l'instruction `mystere2(G, "a", "f")`. On ne demande pas d'indiquer toutes les étapes de l'algorithme.

`mystere2(G, "a", "f")` affiche `(["a", "c", "d", "e", "f"], 8)`

Q24. Justifier si ce programme permet au randonneur de trouver le chemin de difficulté cumulée minimale.

Un algorithme glouton ne permet pas d'obtenir le meilleur résultat à priori. Ici, la liste de randonnées `["a", "d", "e", "f"]` permet d'avoir une difficulté cumulée de 6. L'algorithme glouton ne permet donc pas de répondre à la question.

Q25. On effectue l'appel `dijkstra(G, "a", "f")` où le graphe G est défini dans la **figure ??**. Dans le tableau du **DR** est représenté le contenu de certaines variables de l'algorithme `dijkstra` en fonction de l'étape de l'itération (comme si un `print` était effectué après la ligne 27). À partir des cases déjà remplies, compléter les cases vides du tableau. Lorsque la clé n'est pas définie dans le dictionnaire, la case du tableau contient un X.

		distance						
Étape	sTraite	a	b	c	d	e	f	aVisiter
Initialisation		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"]

Q26. Indiquer le contenu des lignes 8, 9, 10 et 15 du code précédent.

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

Q27. Expliquer comment pourrait être diminué le nombre de tests d'appartenance à une liste, notamment des lignes 14 et 15, de l'algorithme de Dijkstra.

Les parcours de listes pour vérifier les appartenances des sommets sont de complexité linéaire. On pourrait stocker le caractère "à visiter" ou "visité" dans des dictionnaires dont les clés sont les sommets et les valeurs sont des booléens. Le test d'appartenance s'effectuerait alors en temps constant.

On pourrait également utiliser une file de priorité pour diminuer le temps de calcul du minimum (certainement hors programmme...).

Q28. Lister les sommets visités par l'appel `dijkstra(G1, "a1", "j1")`, puis par l'appel `dijkstra(G1, "j1", "a1")`.

- Appel `dijkstra(G1, "a1", "j1")`.
  L'algorithme de Dijkstra va commencer par visiter tous les sommets à distance 1 de `"a1"`, soit : `"b1"`, `"c1"`, `"d1"`.
  Il visite ensuite tous les sommets à distance 2, soit : `"e1"`, `"f1"`, `"g1"`, `"h1"`, `"i1"`.
  Enfin, il visite le sommet à distance 3, soit : `"j1"` et il termine.
- Appel `dijkstra(G1, "j1", "a1")`.
  L'algorithme de Dijkstra va commencer par visiter tous les sommets à distance 1 de `"j1"`, soit : `"g1"`.
  Il visite ensuite tous les sommets à distance 2, soit : `"c1"`.
  Enfin, il visite le sommet à distance 3, soit : `"a1"` et il termine.

Q29. Compléter la dernière ligne des tableaux du **DR** qui recense les étapes successives de l'algorithme de Dijkstra bidirectionnel sur le graphe G de la **figure ??** avec `a` comme sommet de départ et `f` comme sommet d'arrivée. À chaque étape, on précise si le sommet est visité par la recherche forward (F) ou par la recherche backward (B). Dans chaque case sont précisées les valeurs de `distanceF` et `distanceB`. Pour simplifier la lisibilité, le tableau est découpé en deux parties.

Le minimum est atteint à la fois pour le sommet `c` en forward et pour le sommet `d` en backward. Comme le nombre de sommets à visiter pour le backward est plus faible, on choisit donc cette option.

		distanceF | distanceB					
Étape	Traité	a	b	c	d	e	f
Init.		0, a | ∞, f	∞, a | ∞, f	∞, a | ∞, f	∞, a | ∞, f	∞, a | ∞, f	∞, a | 0, f
1	a, F	0, a | ∞, f	3, a | ∞, f	2, a | ∞, f	4, a | ∞, f	∞, a | ∞, f	∞, a | 0, f
2	f, B	0, a | ∞, f	3, a | ∞, f	2, a | ∞, f	4, a | 4, f	∞, a | 1, f	∞, a | 0, f
3	e, B	0, a | ∞, f	3, a | 6, e	2, a | ∞, f	4, a | 2, e	∞, a | 1, f	∞, a | 0, f
4	d, B	0, a | 6, d	3, a | 4, d	2, a | 6, d	4, a | 2, e	∞, a | 1, f	∞, a | 0, f

si erreur en choisissant Forward on obtient :

		distanceF | distanceB					
Étape	Traité	a	b	c	d	e	f
4	c, F	0, a | ∞, f	3, a | 6, e	2, a | ∞, f	4, a | 2, e	∞, a | 1, f	∞, a | 0, f

Étape	Fmin	Bmin	BFmin	aVisiterF	aVisiterB
Init.	0	0	∞	`["a"]`	`["f"]`
1	2	0	∞	`["b", "c", "d"]`	`["f"]`
2	2	1	8	`["b", "c"; "d"]`	`["d", "e"]`
3	2	2	6	`["b", "c", "d"]`	`["d", "b"]`
4	2	4	6	`["b", "c", "d"]`	`["b", "a", "c"]`

ou si erreur :

Étape	Fmin	Bmin	BFmin	aVisiterF	aVisiterB
Init.	0	0	∞	`["a"]`	`["f"]`
4	3	2	6	`["b", "d"]`	`["d", "b"]`

Q30. Justifier si renvoyer la quantité `BFmin` dès qu'un sommet a été atteint par les recherches forward et backward permet de trouver le chemin de longueur minimale.

Non car le premier sommet à être visité par les deux recherches est le sommet `d` lors de l'étape 2, mais on constate que le chemin `a`, `d`, `f` ainsi obtenu est de longueur 8 et n'est donc pas de longueur minimale.

Q31. Compléter la fonction `dijkstra_bidirectionnel` qui, étant donné un graphe `graph`, un sommet de départ `Sd` et un sommet final `Sf`, renvoie la distance minimale reliant `Sd` à `Sf` en utilisant l'algorithme de Dijkstra bidirectionnel. Indiquer précisément le contenu des lignes 4, 5, 13, 21, 22 et 23.
La fonction **`min(L:list)`** renvoie l'élément minimal d'une liste `L`.

```
ligne 4 : aVisiterB = [Sf]
ligne 5 : dejaVisitesB = []
ligne 13: while BFmin > Fmin + Bmin:
ligne 21: ....... distanceF[v][0]...
ligne 22: ....... distanceB[v][0]...
ligne 23: ....... distanceF[v][0]+distanceB[v][0]...
```

Q32. Montrer que s_i appartient soit à `dejaVisitesF` soit à `dejaVisitesB`.

Par l'absurde, si $s_i \notin dejaVisitesF \cup dejaVisitesB$, alors d'après la propriété de l'algorithme de Dijkstra classique,

$$\begin{cases} d(s, s_i) \geq \max\limits_{x \in dejaVisitesF} d(s, x) \geq Fmin, \\ d(s_i, t) \geq \max\limits_{x \in dejaVisitesB} d(x, t) \geq Bmin. \end{cases}$$

Ainsi, $d = d(s, s_i) + d(s_i, t) \geq Fmin + Bmin \geq BFmin$ puisque l'algorithme a terminé. On obtient ainsi une contradiction.

Q33. En déduire qu'il existe un indice i_0 tel que s_{i_0} a été visité par la recherche forward et s_{i_0+1} l'a été par la recherche backward.

Comme s a été visité par la recherche forward et t l'a été par la recherche backward, il existe nécéssairement une arête (s_{i_0}, s_{i_0+1}) pour laquelle s_{i_0} a été visité par la recherche forward et s_{i_0+1} l'a été par la recherche backward.

Q34. En déduire la correction partielle de l'algorithme de Dijkstra bidirectionnel.

Comme s_{i_0} a été visité, la distance forward à s_{i_0+1} a été mise à jour et

$$distanceF[s_{i_0+1}][0] \leq d(s, s_{i_0}) + d(s_{i_0}, s_{i_0+1}).$$

Lorsque s_{i_0+1} a été visité, d'après la propriété rappelée de l'algorithme de Dijkstra,

$$distanceB[s_{i_0+1}][0] = d(s_{i_0+1}, t).$$

Ainsi, d'après la définition de `BFmin`,

$$BFmin \leq distanceF[s_{i_0+1}] + distanceB[s_{i_0+1}] \leq d(s, s_{i_0}) + d(s_{i_0}, s_{i_0+1}) + d(s_{i_0+1}, t) = d,$$

ce qui contredit l'optimalité de d.

FIN