

NOM :PRENOM :

CPGE PSI* 2025/2026
Lycée La Fayette

CORRIGÉ

Informatique
Nathalie Planche

Concours Blanc – Document réponse

Q1 - Explication de pourquoi l'attribut Titre ne peut probablement pas être une clé primaire pour la table Randonnee. Proposition d'un attribut de la table Randonnee qui puisse être une clé primaire

- Deux randonnées différentes peuvent avoir le même titre, donc l'attribut "Titre" ne permet pas de repérer de manière unique une randonnée
- Id est une clé primaire

Q2 - Identification d'un attribut qui puisse être une clé étrangère de la table Randonnee

IdAuteur est une clé étrangère de la table Randonnee, puisqu'elle fait référence à la clé primaire Id de la table Auteur

Q3 - 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 - 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(Id) AS Nombre  
FROM Randonnee  
WHERE Type = "Pied" AND Niveau = 3  
GROUP BY IdAuteur  
ORDER BY Nombre DESC
```


Q5 - Requête SQL dont l'évaluation renvoie le Pseudo de l'auteur et le Titre des randonnées stockées dans la base

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

} jointure entre
2 tables

Q6 - Requête SQL dont l'évaluation renvoie les nom et prénom d'un des auteurs ayant posté le plus de randonnées à cheval

```
SELECT A.nom, A.prenom  
FROM Randonnee AS R JOIN Auteur AS A  
ON R.IdAuteur = A.Id  
WHERE R.Type = "Cheval"  
GROUP BY A.Id # fait référence au COUNT  
ORDER BY COUNT(R.Id) DESC  
LIMIT 1 # on ne fait afficher que le premier
```

Q7 - Instruction pour l'importation du module gpxpy

```
import gpxpy
```

Q8 - Valeur numérique renvoyée par `mystere(I1)`. Signification de cette valeur

Valeur retournée: $(100 + 102 + 110 + 108) / 4 = 420 / 4 = \underline{105}$
C'est l'altitude moyenne de la randonnée

Q9 - Complexité temporelle de `mystere` en fonction de la taille n de la liste passée en argument

$C(n) = O(n)$ C'est une complexité linéaire
car on a une boucle de complexité constante qui s'exécute n fois.

Q10 - Fonction altitude_maximale(iti:itineraire) -> float

```
def altitude_maximale(iti):  
    alt_max = iti[0][2] #initialisation avec la  
                        #première altitude  
    n = len(iti)  
    for i in range(1, n):  
        if iti[i][2] > alt_max:  
            alt_max = iti[i][2]  
    return alt_max
```

Q11 - Fonction denivele_global(iti:itineraire) -> float

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

Q12 - Fonction denivele_positif_cumule(iti:itineraire) -> float

```
def denivele_positif_cumule(iti):  
    d = 0  
    n = len(iti)  
    for i in range(n-1): # pas n, à cause du i+1 dans  
                        # la boucle  
        diff = iti[i+1][2] - iti[i][2]  
        if diff > 0:  
            d = d + diff  
    return d
```


Q13 - Fonction alt_glissante(liste_alt:list, p:int) -> list

```
def alt_glissante(liste_alt, p):  
    liste_lissee = []  
    n = len(liste_alt)  
    for i in range(n):  
        j = min(p-1, n-i-1)  
        moy = 0  
        for k in range(i, i+j+1): # calcul de la moyenne glissante pour le point i  
            moy = moy + liste_alt[k]  
        moy = moy / (j+1)  
        liste_lissee.append(moy)  
    return liste_lissee
```

Q14 - Complexité de la fonction alt_glissante(liste_alt:list, p:int) -> list

On va prendre pour valeur maximale pour j à chaque fois p-1 (cas le plus probable, car min(p-1, n-i-1) sert seulement à ne pas dépasser l'indice de fin de liste)

Donc

$$C(n, p) = 2 + n \times (2 + 2 \times p + 4) = \underline{O(np)}$$

boucle extérieure ↗ ↖ boucle intérieure

Q15 - Type des variables lat_ref et long_ref. Explication du contenu de ces variables dans le contexte de ce sujet

lat_ref et long_ref sont des listes de flottants

lat_ref contient toutes les latitudes de référence apparaissant dans le "Digital Elevation Model" et long_ref contient de même toutes les longitudes.

Q16 - Signature des fonctions auxiliaire et principal. Justifier

def auxiliaire(x:list, y:list) -> list

def principal(x:list) -> list

Justification: les opérations utilisées dans les fonctions ne peuvent convenir qu'à des listes

Q17 - Type(s) de programmation utilisé(s) pour coder la fonction principal

- | | |
|---|---|
| ☐ Itératif | ☐ Algorithme glouton |
| ☒ Récursif | ☐ Programmation dynamique |
| ☐ k plus proches voisins | ☒ Diviser pour régner |

Q18 - Justification de la terminaison de la fonction principal

- Tout d'abord la fonction principale fait appel à la fonction auxiliaire, et la fonction auxiliaire se termine bien : l'appel récursif s'effectue sur des listes dont les longueurs diminuent à chaque appel, donc on aboutira forcément à l'un des tests d'arrêts $x = []$, ou $y = []$
- La fonction principale se termine car les 2 appels récursifs se font sur des listes de taille strictement inférieure à la taille de x , donc on aboutira forcément au test d'arrêt $\text{len}(x) = 0$ ou 1

Q19 - Nom décrivant le type de tri utilisé. Justification de votre choix

Il s'agit d'un tri fusion : la fonction auxiliaire permet de faire une fusion entre 2 listes triées, et la fonction principale permet de scinder récursivement une liste en deux, avant d'effectuer des fusions.

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

Ligne 5 :

$\text{ind_deb} = 0$

il s'agit d'un algorithme de recherche par dichotomie

Ligne 6 :

$\text{ind_fin} = \text{len}(\text{liste_ref}) - 1$

Ligne 8 :

$k = (\text{ind_deb} + \text{ind_fin}) // 2$ # l'indice milieu

Ligne 10 :

$\text{ind_fin} = k$

Ligne 12 :

$\text{ind_deb} = k$

Ligne 15 :

$\text{return liste_ref}[\text{ind_fin}]$

Ligne 17 :

$\text{return liste_ref}[\text{ind_deb}]$

Q21 - Fonction `standardise(liste_parcours:itineraire) -> itineraire`

```
def standardise(liste_parcours):  
    new_liste = []  
    n = len(liste_parcours)  
    for i in range(n):  
        (lat, long, alt) = liste_parcours[i]  
        lat_proche = ref(lat, lat_ref)  
        long_proche = ref(long, long_ref)  
        new_alt = dem[(lat_proche, long_proche)]  
        new_liste.append((lat, long, new_alt))  
    return new_liste
```

Q22 - Explication des lignes 15 à 19 de la fonction `mystere2`. Nom du type d'algorithme utilisé

ligne 15: on se place au sommet courant `sTraite`, et on va examiner tous ses voisins qui n'ont pas encore été visités (ligne 16). lignes 17-18-19: va permettre de stocker dans `sInter` le sommet voisin le plus proche de `sTraite`. difficulté la + faible

Il s'agit d'un algorithme glouton car on cherche sur une étape le chemin de difficulté minimale, mais pas de vision globale.

Q23 - Résultat de l'instruction `mystere2(G, "a", "f")`

`["a", "c", "d", "e", "f"], 8`

Q24 - Ce programme permet-il au randonneur de trouver le chemin de difficulté cumulée minimale? Vous justifierez votre réponse.

Ce n'est pas le chemin de difficulté minimale, par exemple $a \xrightarrow{3} b \xrightarrow{2} d \xrightarrow{1} e \xrightarrow{1} f$ a une difficulté 7

Q25 - Tableau ci-dessous à compléter

Étape	sTraite	distance						aVisiter
		a	b	c	d	e	f	
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 - Compléter les lignes 8, 9, 10 et 15 du code

Ligne 8 :

`s! = sInit`

Ligne 9 :

`s = distance[s][1]` #on remplace s par son prédécesseur

Ligne 10 :

`chemin.append(s)`

Ligne 15 :

`distance[sfin][0]`

Q27 - Proposition pour diminuer le nombre de tests d'appartenance à une liste de l'algorithme de Dijkstra

- On peut créer un dictionnaire avec clé: sommet, valeur: booléen indiquent si le sommet a déjà été visité.
Pour savoir si un sommet a déjà été visité, la complexité sera constante au lieu de linéaire.

Q28 - Liste des sommets visités :

- par l'appel `dijkstra(G1, "a1", "j1")`

`a1, b1, c1, d1, e1, f1, g1, h1, i1, j1` #on les fait tous!

- par l'appel `dijkstra(G1, "j1", "a1")`

`j1, g1, c1, a1`

Q29 - Dernière ligne de chacun des tableaux ci-dessous à compléter.

		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

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"]

Q30 - Justification de la proposition pour trouver le chemin de longueur minimale

Si le "atteint" de la question veut dire que le sommet est dans "déjà visité", alors oui les distances seront minimisées et on peut renvoyer BFmin

Q31 - Compléter les lignes 4, 5, 13, 21, 22 et 23 de la fonction dijkstra_bidirectionnel

Ligne 4 :

avisiterB = [sf]

Ligne 5 :

dejavisitesB = []

Ligne 13 :

while BFmin > Fmin + Bmin

Ligne 21 :

Fmin = min([distanceF[v][0] ...

Ligne 22 :

Bmin = min([distanceB[v][0] ...

Ligne 23 :

L = [distanceF[v][0] + distanceB[v][0] ...