teaching sciences for innovation

# Proposition de corrigé

Concours : Concours Commun INP

Année : 2025

Filière : PSI

Épreuve : Informatique

Ceci est une proposition de corrigé des concours de CPGE, réalisée bénévolement par des enseignants de Sciences Industrielles de l'Ingénieur et d'Informatique, membres de l'UPSTI (Union des Professeurs de Sciences et Techniques Industrielles).



L'équipe UPSTI

# Gestion de randonnées

Corrigé UPSTI

## 1 Gestion des randonnées dans une base de données

### Question 1

L'attribut `Titre` ne peut probablement pas être une clé primaire pour la table `Randonnee` car plusieurs randonnées peuvent avoir le même nom. L'attribut `Id` peut être une clé primaire.

### Question 2

L'attribut `IdAuteur` est une clé étrangère de la table `Randonnee` car elle peut être utilisée pour faire une jointure avec la table `Auteur`.

### Question 3

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

### Question 4

```
SELECT IdAuteur, Count(*) AS C FROM Randonnee
WHERE Niveau = 3 AND Type='Pied'
GROUP BY IdAuteur
SELECT BY C DESC ;
```

ORDER

### Question 5

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

### Question 6

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

## 2 Quelques calculs de dénivelés

### Question 7

```
Import gpxpy
```

**Question 8**

La valeur numérique renvoyée par le code est 105. Il s'agit de l'atitude moyenne de l'itinéraire.

**Question 9**

La compléxité de la fonction mystere est $\mathcal{O}(n)$ avec $n$ la longueur de la liste `iti`.

**Question 10**

```
def altitude_maximale(iti) :
    maxi=iti[0][2]
    for i in range(1,len(iti)) :
        if iti[i][2]>maxi:
            maxi=iti[i][2]
    return maxi
```

**Question 11**

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

**Question 12**

```
def denivele_positif_cumule(iti):
    den=0
    for i in range(len(iti)-1):
        if iti[i + 1][2]-iti[i][2]>0 :
            den + =iti[i + 1][2]-iti[i][2]
    return den
```

**Question 13**

```
def alt_glissante(liste_alt,p) :
    L_lissee=[]
    n=len(liste_alt)
    for i in range(n):
        moy=0
        for j in range(min(p-1,n-i-1) + 1) :
            moy + =liste_alt[i + j]
        L_lissee.append(moy/(j + 1))
    return L_lissee
```

**Question 14**

La complexité de la fonction est $\mathcal{O}(np)$.

**Question 15**

Les variables `latref` et `longref` sont des listes.
La liste `latref` contient la liste des latitudes des points de référence et la liste `longref` celle des longitudes des point de références.
Les élements de ces deux listes constituent les clés du dictionnaires `dem`.

**Question 16**

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

La fonction `auxliaire` prend en argument deux listes triées, puis les fusionne. Elle renvoie la fusion des deux listes pour ne former qu'une liste triée. La fonction `principal` prend une liste en argument, et renvoie la liste triée.

**Question 17**

Récursif et Diviser pour mieux régner

**Question 18**

La fonction `principal` est composée d'un cas général et d'un cas terminal. Le cas terminal est utilisé lors que la liste ne contient aucun ou un seul élément. Dans le cas général, la fonction principal s'appelle elle-même avec des intervalles de plus en plus petit. Au bout d'un moment, elle s'appelle avec une liste avec 0 ou 1 élément qui correspond au cas terminal.

**Question 19**

Il s'agit d'un tri fusion. La liste est coupé en deux. Chaque sous-liste est triée à l'aide du tri fusion puis les 2 listes triées sont fusinnées.

**Question 20**

5 : `ind_deb=0`
6 : `ind_fin=len(liste_ref)-1`
8 : `k=(ind_fin + ind_deb)//2`
10 : `ind_fin=k`
12 : `ind_deb=k`
15 : `return liste_ref[ind_fin]`
17 : `return liste_ref[ind_deb]`

**Question 21**

```
def standardise(liste_parcours) :
    itineraire_s=[]
    for(lat,long,alt) in liste_parcours :
        lat_s=ref(lat,lat_ref)
        long_s=ref(lat,long_ref)
        alt_s=dem[(lat,long)]
        itineraire_s.append([lat_s,long_s,alt_s])
    return itineraire_s
```

**Question 22**

La fonction étudie tous les sommets voisins du sommet `sTraite` (le sommet à traiter) et choisi celui qui est le plus proche.
Il s'agit d'un algorithme glouton.

**Question 23**

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

**Question 24**

Le programme ne permet pas de trouver le chemin de dificulté cumulée minimale. Dans le cas de l'exemple, le chemin de difficulté cumulée minimale est `['a', 'b', 'd', 'e', 'f']` avec une difficulté de 7.

**Question 25**

| | | distance | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Etape | 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 | b | 0,a | 3,a | 2,a | 4,a | X | X | ['b','d'] |
| 3 | c | 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'] |

**Question 26**

8 : `while chemin[-1]!= sInit :`
9 : `chemin.append(distance[chemin[-1]][1])`
10 : non nécessaire
15 : `distance[sFin][0]`

**Question 27**

Il est possible de supprimer le test d'appartenance de la ligne 14. Dans ce cas, chaque sommet non visité `v` voisin du sommet `s` sera ajouté dans la liste `avisiter` et il sera visité plusieurs fois, ce qui n'a pas d'incidence sur le dictionnaire créé.

Remarque : le dictionnaire créé est le même mais ça semble assez bête de visiter plusieurs fois le même sommet. Je ne vois pas d'autres réponses à part celle-là

**Question 28**

Lors de l'appel `dijkstra(G1,'a1','j1')` tous les sommets sont visités. Lors de l'appel `dijkstra(G1,'j1','a1')`, seuls les sommets `j1`, `g1`, `c1` et `a1` sont visités.

**Question 29**

| | | distanceF\|distanceB | | | | | |
|---|---|---|---|---|---|---|---|
| Etape | 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\|1,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 |

| Etape | 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'] |

**Question 30**

Renvoyer la quantité `BFmin` dès qu'un sommet a été atteint par les recherches backward et forward permet de trouver le chemin de longueur minimale.
L'algorithme permet de connaitre le chemin le plus court de `Sd` (sommet de départ) au sommet intermédiaire ainsi que le chemin le plus court de `Sf` (sommet final) à ce même sommet intermédiaire. Cela permet donc de connaitre le chemin le plus court de `Sd` à `Sf` en passant par le sommet intermédiaire. Le fait de toujours partir

du sommet avec le chemin le plus "faible" (lors des étapes successives) avec les variables `Fmin` et `Bmin` pour avancer dans le graphe nous permet d'être sûr que ce chemin est bel et bien le plus court pour passer de `Sd` à `Sf`.
Ce résultat peut être retrouvé lors du remplissage du tableau de la question 29. En s'arrêtant dès qu'un sommet a été atteint par les recherches backward et forward, on obtient le chemin `'a'->d->e->f` de distance totale 6 qui est effectivement le chemin qui réalise la plus petite distance entre `a` et `f`.

**Question 31**

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

**Question 32**

Supposons par l'absurde que $s_i$ n'appartient ni à `dejaVisitesF` ni à `dejaVisitesB`.
Comme $s_i$ n'appartient pas à `dejaVisitesF`, $d(s_0, s_i)$ est supérieure à la plus grande des distances déjà calculées donc $d(s_0, s_i) \geq$`Fmin`.
De la même manière, comme $s_i$ n'appartient pas à `dejaVisitesB`, $d(s_i, s_n) \geq$`Bmin`.
Donc $d(s_0, s_n) = d(s_0, s_i) + d(s_i, s_n) \geq$`Fmin`+`Bmin`$\geq$`BFmin`. Or, $d(s_0, s_n) <$`BFmin`.
Donc $s_i$ appartient soit à `dejaVisitesF` soit à `dejaVisitesB`.

**Question 33**

Si $s_0 \neq s_t$, alors $s_0$ appartient à `dejaVisitesF` et $s_t$ appartient à `dejaVisitesB`.
Quelque soit $i$, $s_i$ appartient soit à `dejaVisitesF` soit à `dejaVisitesB` (cf Question 32).
Supposons par l'absurde qu'il n'existe pas d'indice $s_{i_0}$ tel que $s_{i_0}$ soit visité par la recherche forward et $s_{i_0+1}$ par la recherche backward. Alors le fait que $s_0$ appartienne à `dejaVisitesF` implique que $s_1$ appartienne à `dejaVisitesF` qui implique que $s_2$ appartienne à `dejaVisitesF` et ainsi de suite. Donc $s_t$ appartient à `dejaVisitesF`. Cela est en contraction avec le fait que $s_0$ appartient à `dejaVisitesF` et $s_t$ appartient à `dejaVisitesB`.

**Question 34**

Nous cherchon ici à montrer la correction partielle de l'algorithme de Dijkstra bidirectionnel ; cela revient à justifier que lorsque l'algorithme se termine, le chemin trouvé est le plus court chemin.
Pour cela, il a été supposé par l'abusrde que lorsque l'algorithme se termine, un chemin plus court que celui trouvé existe. Les résultats des questions 32 et 33 découle de cette hypothèse. Nous cherchons ici à montrer que la proposition obtenue à l'issue de la question 33 est en contradiction avec le critère d'arrêt de l'algorithme birectionnel détermine à la question 30.

On remarque d'abord que comme l'entend la question 30, l'algorithme s'arrête dès qu'un sommet a été atteint par les recherchers forward et backward. On précise que "atteint" signifie qu'il est à la fois dans `aVisiterF` et `aVisiterB`, mais ni dans `dejaVisiteF` ni dans `dejaVisiteB`.
On note `v` le sommet nouvellement atteint qui réalise la distance minimale `BFmin`. En accord avec la question 33, le chemin ainsi obtenu de distance totale `BFmin` est tel que tous les sommets avant `v` sont dans `dejaVisiteF`, tous les sommets après `v` sont dans `dejaVisiteB`, et `v` n'est dans aucun des deux.
D'après le critère d'arrêt de l'algorithme bidirectionnel, il ne peut donc pas exister de séquence $(s_0, s_1, ..., s_n)$ tel qu'il existe $s_{i_0}$ avec $s_{i_0}$ appartenant à `dejaVisiteF` et $s_{i_0+1}$ appartenant à `dejaVisiteB`. Cela est en contradiction avec le résultat de la question 33.
Donc il n'existe pas de séquence $(s_0, ..., s_n)$ de distance strictement plus petite que `BFmin`. Cela est en contra-

diction avec la supposition effectuée. La supposition est donc fausse.
Nous venons donc de montrer par l'absurde que lorsque l'algorithme de Dijkstra bidirectionnel termine, la distance `BFmin` est la plus petite distance reliant le sommet de départ au sommet d'arrivée.