BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2024

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 13 pages numérotées de 1 / 13 à 13 / 13.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

EXERCICE 1 (6 points)

Cet exercice porte sur les listes de listes ainsi que la programmation orientée objet.

Le solitaire bulgare est un jeu de cartes dans lequel on forme des piles de cartes dont seul le nombre de cartes est important. On dispose au départ les cartes sur une seule pile. À chaque tour, on prend une carte de chaque pile et on forme une nouvelle pile à l'aide de ces cartes. Quand une pile a été vidée, on considère qu'elle disparait.

L'ensemble des tailles des piles est appelé l'état et on peut le représenter par une liste triée d'entiers dans l'ordre croissant.

Par exemple, si les piles contiennent respectivement 4, 6 et 10 cartes, elles seront représentées par la liste `[4, 6, 10]`. Pour passer à l'étape d'après, on pioche la carte au sommet de chacune de ces piles. On pioche ainsi trois cartes et les piles sont de tailles respectives 3, 5 et 9. On ajoute les trois cartes sur une nouvelle pile et on obtient alors quatre piles de tailles 3, 3, 5 et 9, représentées par la liste `[3, 3, 5, 9]`.

On pourra noter `[4, 6, 10] → [3, 3, 5, 9]` cet enchainement d'état.

Si le jeu de départ comporte 15 cartes, on commence avec une unique pile représentée par `[15]`. On ne peut piocher qu'une carte, ce qui amène aux deux piles `[1, 14]`. En continuant, on pioche deux cartes, se faisant la pile de taille 1 est vidée alors qu'on rajoute une pile de taille 2. La situation est alors `[2, 13]`. On a donc l'enchainement `[15] → [1, 14] → [2,13]`.

Si on continue, la partie se déroule ainsi :

```
[15] → [1, 14] → [2, 13] → [1, 2, 12] → [1, 3, 11] →
[2, 3, 10] → [1, 2, 3, 9] → [1, 2, 4, 8] → [1, 3, 4, 7] →
[2, 3, 4, 6] → [1, 2, 3, 4, 5] → [1, 2, 3, 4, 5] etc.
```

On constate que l'état `[1, 2, 3, 4, 5]` se réduit sur lui-même : on pioche cinq cartes, on vide ainsi la première pile et on se ramène à des piles de taille 1, 2, 3 et 4 auxquelles on rajoute cette nouvelle pile de cinq cartes. On retombe alors sur `[1, 2, 3, 4, 5]`. On dit que l'état `[1, 2, 3, 4, 5]` est stable.

Un deuxième exemple à la main

1. Vérifier qu'avec 10 cartes, la partie se termine de nouveau avec un état stable.

Par ailleurs, il est également possible que la partie se termine par un cycle comme la partie commençant avec 12 cartes :

```
[12] → [1, 11] → [2, 10] → [1, 2, 9] → [1, 3, 8] → [2, 3, 7] →
[1, 2, 3, 6] → [1, 2, 4, 5] → [1, 3, 4, 4] → [2, 3, 3, 4] →
[1, 2, 2, 3, 4] → [1, 1, 2, 3, 5] → [1, 2, 4, 5] → [1, 3, 4, 4] etc.
```

Avec 12 cartes, la partie aboutit au bout de 12 itérations par un cycle : les piles suivantes sont les mêmes que celles que l'on avait calculées à partir de la 8e itération.

Un troisième exemple à la main

2. Vérifier qu'avec 9 cartes, la partie se termine avec un cycle.

On souhaite maintenant écrire un programme permettant de savoir si le solitaire bulgare se termine par un état stable ou par un cycle.

Une classe pour la partie

```
1 class Partie:
2     def __init__(self, nb_cartes):
3         """
4             self.courant est l'état actuel
5             self.precedents est la liste des états précédents
6         """
7         self.courant = [nb_cartes] # état initial
8         self.precedents = []
9
10    def suivante(self):
11        """
12            place l'état courant dans la liste des états
                précédents
13            calcule le nouvel état courant et le remplace
14        """
15        self.precedents.append(...)
16        valeurs = self.courant
17        suiv = [len(valeurs)]
18        for v in valeurs:
19            if v > 1:
20                suiv.append(v-1)
21        suiv.sort() # tri des valeurs dans l'ordre croissant
22        self.courant = ...
23
24    def est_stable(self):
25        ...
26
27    def est_cyclique(self):
28        """
29            renvoie True si l'état courant a déjà été rencontré
30        """
31        return self.courant in self.precedents
```

3. Compléter les lignes 15 et 22 de la méthode appelée `suivante` qui permet de passer d'un état de la partie à l'état suivant.

4. Compléter la méthode `est_stable` qui renvoie `True` si l'état courant est le même que le dernier état précédent et `False` sinon.

5. Écrire une fonction `partie_stable` prenant en paramètre `nb_cartes` et qui renvoie `True` si la partie finie par un état stable et `False` si la partie finie par un état cyclique.
   On pourra suivre l'algorithme suivant : créer une instance de partie avec le bon nombre de cartes puis tant que la partie n'est ni stable ni cyclique, passer à l'état suivant.

6. Proposer des tests pour vérifier si votre fonction `partie_stable` semble correcte.

7. Dans la méthode `suivante`, on trie une liste qui est déjà triée sauf éventuellement la valeur `len(valeurs)`. Donner, en le justifiant, un tri classique qui serait plus efficace dans ce cas.

EXERCICE 2 (6 points)

Cet exercice traite d'architecture réseaux, et de routage

On s'intéresse au réseau informatique d'une entreprise.

Le parc informatique de cette entreprise est constitué des quatre réseaux interconnectés suivants :

- réseau A d'adresse de réseau 192.168.64.0 et de masque 255.255.248.0 ;
- réseau B d'adresse de réseau 192.168.192.0 et de masque 255.255.255.0 ;
- réseau C d'adresse de réseau 192.168.193.0 et de masque 255.255.255.0 ;
- réseau D d'adresse de réseau 192.168.128.0 et de masque 255.255.254.0.

On rappelle qu'un réseau d'adresse 192.168.0.0 et de masque 255.255.0.0 couvre les adresses allant de 192.168.0.1 à 192.168.255.254. L'adresse 192.168.0.0 est réservée en tant qu'adresse de réseau et l'adresse 192.168.255.255 en tant qu'adresse diffusion.

1. Indiquer à quel réseau (A, B, C, D ou AUCUN) appartient chacune de ces quatre adresses de postes de travail :

- 192.168.129.3
- 192.168.194.2
- 192.168.192.7
- 192.168.72.25

2. Indiquer, pour les réseaux A et D le nombre de machines qu'ils peuvent effectivement contenir au maximum.

Les réseaux sont agencés comme indiqués sur la figure 1, et interconnectés par trois routeurs R1 à R3. Les deux adresses IP de R2 sont indiquées, ainsi que l'une des deux adresses de R1 et l'une des deux adresses de R3.

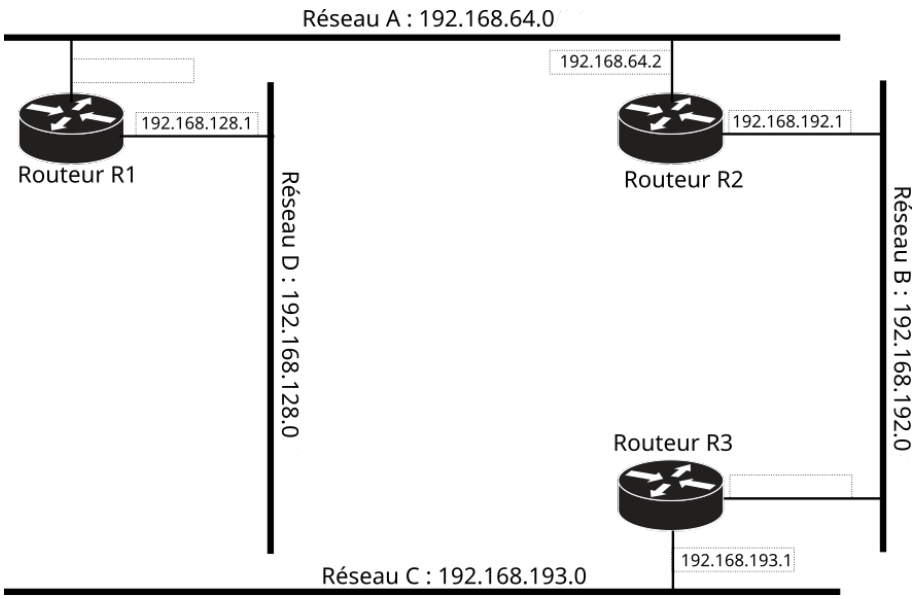

Figure 1. Quatre réseaux interconnectés par trois routeurs.

3. Choisir les deux adresses IP manquantes pour R1 dans le réseau A pour R3 dans le réseau B. Prendre la plus petite adresse IP disponible qui convienne à chaque fois.

Les tables de routage de R1 et R3 sont les suivantes :

Table de routage de R1 :

Réseau	Masque	Passerelle	Hops/Sauts
192.168.64.0	255.255.248.0	0.0.0.0	0
192.168.192.0	255.255.255.0	192.168.64.2 (R2)	1
192.168.193.0	255.255.255.0	192.168.64.2 (R2)	2
192.168.128.0	255.255.254.0	0.0.0.0	0

Table de routage de R3 :

Réseau	Masque	Passerelle	Hops/Sauts
192.168.64.0	255.255.248.0	192.168.192.1 (R2)	1
192.168.192.0	255.255.255.0	0.0.0.0	0
192.168.193.0	255.255.255.0	0.0.0.0	0
192.168.128.0	255.255.254.0	192.168.192.1 (R2)	2

4. Compléter la table de routage de R2 :

Réseau	Masque	Passerelle	Hops/Sauts
192.168.64.0	255.255.248.0	0.0.0.0	0
……	……	……..	….
……	……	……..	….
……	……	……..	….

Afin de désengorger le réseau, un quatrième routeur (R4) est ajouté (Figure 2) et ses adresses IP sont connues.

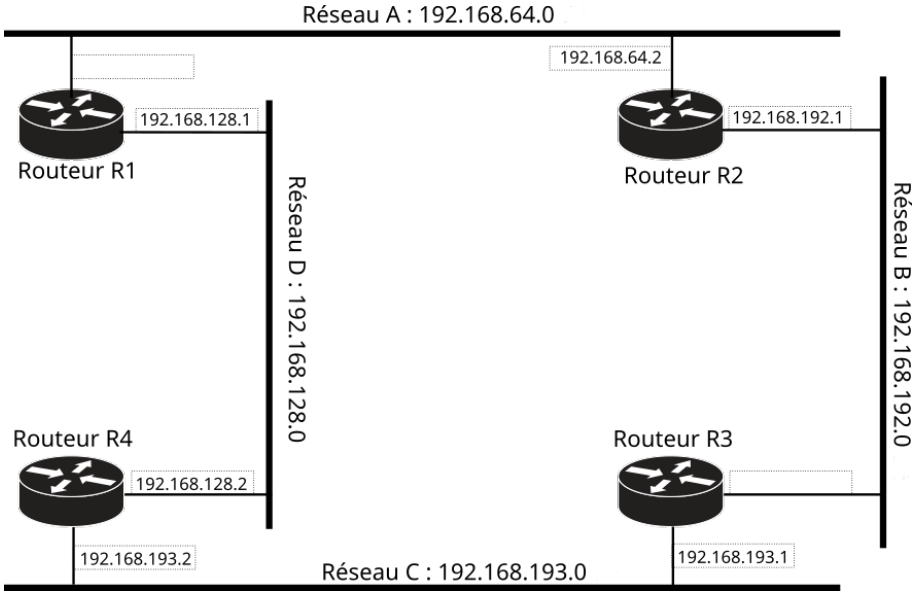

Figure 2. Les réseaux sont maintenant reliés par quatre routeurs.

Initialement, la table de routage de R4 est :

Réseau	Masque	Passerelle	Hops/Sauts
192.168.193.0	255.255.255.0	0.0.0.0	0
192.168.128.0	255.255.254.0	0.0.0.0	0

Le protocole RIP est maintenant utilisé pour construire dynamiquement les tables de routage des quatre routeurs en partant des tables statiques précédentes.

5. Après que chaque routeur aura communiqué une seule fois sa table de routage aux routeurs voisins, le routeur R4 aura reçu les tables de routage de R1 et R3. Donner la nouvelle table de routage de R4 après ces échanges (présenter les tables comme aux questions précédentes).

Un poste de travail est ajouté dans le réseau B. Son adresse IP est 192.168.192.17 et sa table de routage est statique (pas d'utilisation du protocole RIP sur le poste de travail) :

Réseau	Masque	Passerelle
192.168.192.0	255.255.255.0	0.0.0.0
0.0.0.0	0.0.0.0	192.168.192.1 (R2)

6. Le poste 192.168.192.17 ci-dessus essaie de joindre les nœuds 192.168.193.17, 192.168.129.17, 192.168.65.17 et 192.168.192.27. Dans chaque cas, indiquer quels seront les routeurs traversés, dans l'ordre, par les datagrammes IP.

7. À présent, le réseau D doit être découpé en deux sous-réseaux de tailles égales. Indiquer l'adresse IP et le masque associés à chacun des deux sous-réseaux.

De ces deux sous-réseaux, celui, dont le troisième octet est le plus faible, est relié à R1 et R4. L'autre est uniquement relié à R1.

8. En supposant que les tables de routage aient été retouchées suite à cette modification, indiquer quelles seront les nouvelles tables de routage de R1 et R4 **après qu'elles auront été stabilisées** par le protocole RIP. Si des adresses IP manquent, choisir des adresses possibles.

EXERCICE 3 (8 points)

Cet exercice porte sur la programmation Python (listes…), le traitement de données en table et les bases de données relationnelles.

Les deux parties sont indépendantes.

Partie A

Un **trail** est une course à pied en milieu naturel.

Une **trace** (ou *trace GPS*) est une liste d'au moins deux positions géographiques représentant un parcours.

Dans cet exercice, on souhaite réaliser un programme qui produit des statistiques intéressantes à partir d'une trace de *trail*.

La trace d'une course est stockée dans un fichier au format csv. Chaque ligne du fichier contient, dans cet ordre, la latitude en degrés Nord, la longitude en degrés Est (tous deux des nombres à virgule) puis l'altitude en mètres, et enfin une valeur d'horodatage ou *timestamp* (tous deux des entiers).

Voici un extrait typique de fichier csv :

```
46.03494, 7.09953, 1457, 1689008478
46.03522, 7.09947, 1447, 1689008481
46.03588, 7.09885, 1440, 1689008484
46.03635, 7.09819, 1439, 1689008487
46.03695, 7.09812, 1433, 1689008490
...
```

La lecture du fichier est effectuée par le programme Python suivant (il n'est pas nécessaire de comprendre ce code pour pouvoir faire le sujet) :

```
import csv
def lecture_trace_csv(filename):
    """
    Prend une chaîne de caractère (nom du fichier) en entrée
    et renvoie une liste contenant les données lues
    """
    tab_trace = []
    with open(filename, 'r') as f:
        csvreader = csv.reader(f, delimiter=',')
        for ligne in csvreader:
            tab_trace.append(ligne)
    return tab_trace
```

Ce code peut être utilisé ainsi :

```
>>> tab_trace = lecture_trace_csv('track.csv')
>>> print(tab_trace)
 [['46.03494', '7.09953', '1457', '1689008478'],
  ['46.03522', '7.09947', '1447', '1689008481'],
```

```
 ['46.03588', '7.09885', '1440', '1689008484'],
 ['46.03635', '7.09819', '1439', '1689008487'],
 ['46.03695', '7.09812', '1433', '1689008490']]
```

La variable `tab_trace` est donc une liste contenant des listes de quatre chaînes de caractères.

1. L'altitude du premier point est `'1457'`. Cette chaîne peut être affichée ainsi : `print(tab_trace[0][2])`. Donner l'instruction permettant d'afficher la longitude du dernier point uniquement.

Pour réaliser des calculs sur les données de la trace, il est nécessaire de convertir les différentes valeurs en nombres (`float` pour la latitude et la longitude et `int` pour l'altitude et le *timestamp*).

Pour cela, on écrit une fonction `creation_trace`, prenant en paramètre une liste similaire à `tab_trace` et renvoyant une liste de dictionnaires, chaque dictionnaire contenant une position horodatée (avec les quatre clés `lat`, `lon`, `alt`, `tsp`) :

```
>>> trace = creation_trace(tab_trace)
>>> for k in range(4):
        print(trace[k])
{'lat': 46.03494, 'lon': 7.09953, 'alt': 1457, 'tsp': 1689008478},
{'lat': 46.03522, 'lon': 7.09947, 'alt': 1447, 'tsp': 1689008481},
{'lat': 46.03588, 'lon': 7.09885, 'alt': 1440, 'tsp': 1689008484},
{'lat': 46.03635, 'lon': 7.09819, 'alt': 1439, 'tsp': 1689008487}
```

Noter que dans les dictionnaires, la latitude et la longitude sont des nombres à virgule flottante, alors que l'altitude et le timestamp sont des entiers.

2. Compléter les lignes 6 à 8 de la fonction `creation_trace` ci-après.

```
 1 def creation_trace(tab_trace):
 2     trace = []
 3     for enregistrement in tab_trace:
 4         pt = {}
 5         pt['lat'] = float(enregistrement[0])
 6         pt['lon'] = ...
 7         ...
 8         ...
 9         trace.append(pt)
10     return trace
```

Une nouvelle fonction doit prendre en paramètre une liste renvoyée par `creation_trace` et renvoyer une liste contenant les altitudes successives. Elle devra satisfaire l'assertion suivante (sur les données d'exemple) :

```
# Calcul des altitudes pour les 4 premiers éléments de trace
seulement
>>> alti = valeurs_altitude(trace)
>>> assert alti[0] == 1457
>>> assert alti[1] == 1447
```

```
>>> assert alti[2] == 1440
>>> assert alti[3] == 1439
```

3. Compléter la fonction suivante, qui prend en paramètre une trace et renvoie la liste des altitudes successives :

```
1 def valeurs_altitude(trace):
2     # Compléter avec autant de lignes que nécessaire
3     ...
4     ...
```

On appelle **segment** d'une trace *t* tout couple *(p1, p2)* où *p1* et *p2* sont 2 points consécutifs de *t*. Ainsi, les *n* points d'une trace déterminent *n - 1* segments.

Le **dénivelé d'un segment** est la différence d'altitude entre le point final d'un segment et le point initial.

On appelle **dénivelé cumulé positif** d'une trace des dénivelés pour les segments dont le dénivelé est positif et **dénivelé cumulé négatif** la somme des valeurs absolues des segments dont le dénivelé est négatif (voir Figure 1).

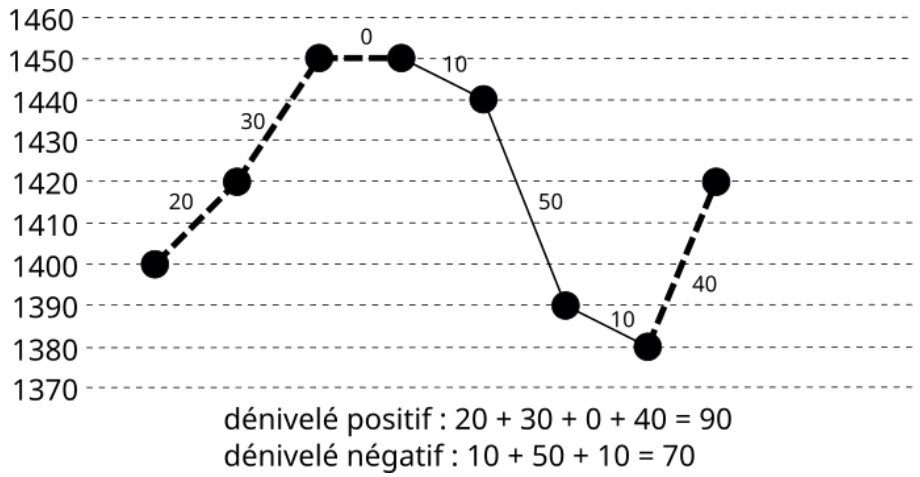

Figure 1. Calcul des dénivelés cumulés positif (pointillés épais) et négatif (traits pleins fins).

4. Donner le code de la fonction `denivele` qui prend en paramètre une liste d'altitudes et renvoie un tuple contenant le dénivelé cumulé positif et le dénivelé cumulé négatif, qui sont tous deux des nombres positifs :

```
>>> denivele([1400, 1420, 1450, 1450, 1440, 1390, 1380, 1420])
 (90, 70)
```

Les valeurs des horodatages (*timestamps*) sont des valeurs en secondes, exprimées à partir d'une date référence (1er janvier 1970). La différence entre deux horodatages donne donc le temps écoulé entre les deux mesures en secondes.

Afin de refléter la difficulté d'un parcours, on souhaite disposer d'une nouvelle fonction qui prendra en paramètre une trace (liste renvoyée par `creation_trace`) et renverra

un tuple de trois valeurs entières contenant : le nombre de secondes d'ascension, le nombre de secondes de descente, et le nombre de secondes sur plat, dans cet ordre.

Cette fonction pourra être utilisée ainsi (elle indique ici 190 secondes de montée, 233 secondes de descente et 22 secondes de plat) :

```
>>> asc_desc(trace)
(190, 233, 22)
```

5. Donner le code de la fonction `asc_desc`.

Soient deux points A et B définis par leur latitude et leur longitude. La distance qui les sépare à la surface du globe (sans tenir compte de l'altitude) est donnée par la formule :

$$d = R \times \arccos\big(\sin(lat_A) \times \sin(lat_B) + \cos(lat_A) \times \cos(lat_B) \times \cos(long_A - long_B)\big)$$

avec R le rayon de la Terre en mètres (on pourra prendre 6371000 m), tous les angles exprimés en radians et $\arccos$ la fonction *arccosinus* (fonction réciproque de *cosinus*).

6. Écrire une fonction nommée `distance(p1: dict, p2: dict)` qui prend en paramètre deux *points* (deux dictionnaires donc) et renvoie la distance qui les sépare en mètres.

   Les fonctions suivantes, du module `math` peuvent être utiles :

   – `math.cos(a)` : renvoie le cosinus de l'angle `a` exprimé en radians
   – `math.sin(a)` : renvoie le cosinus de l'angle `a` exprimé en radians
   – `math.acos(v)` : renvoie l'arc cosinus de `v`, en radians
   – `math.radians(a)` : renvoie la valeur en radians d'un angle `a` donné en degrés

7. En utilisant le théorème de Pythagore, proposer le code d'une fonction `distance_precise(p1: dict, p2: dict)` qui tient en plus compte de l'altitude des deux points pour le calcul de la distance (voir Figure 2).

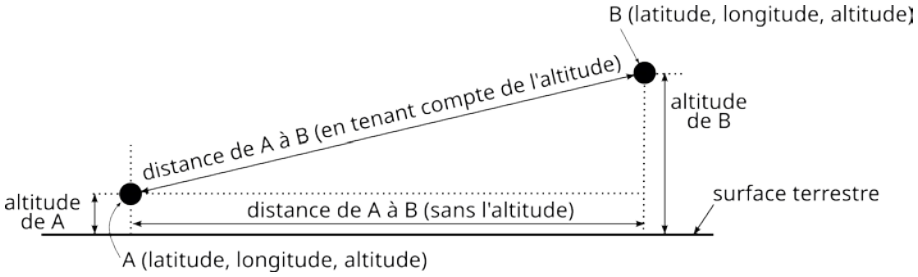

Figure 2. Calcul de la distance en tenant compte de l'altitude.

Partie B

Des fichiers `csv` et les différentes fonctions écrites précédemment sont utilisés pour alimenter une base de données relationnelle. Cette base ne contient qu'une relation, nommée `trails`, dont voici un échantillon (les dénivelés sont en mètres, la distance est en kilomètres, et le temps est en heures) :

Relation `trails`					
`id`	`lieu`	`dist`	`temps`	`denivele_p`	`denivele_n`
1	Mont Blanc	21.3	3	1919	1898
5	Saint Pardoux	24	2	40	40
2	Mont Blanc	171	30	10000	10000
6	Ile de Ré	92	7.5	320	320
…	…	…	…	…	…

8. Donner le résultat de la requête suivante dans le cas où la relation ne contient que les quatre enregistrements ci-dessus :

```
SELECT lieu, dist FROM trails WHERE denivele_p > 1000;
```

9. Donner la requête SQL permettant d'obtenir uniquement les `id` et les lieux (colonne `lieu`) des *trails* dont la distance dépasse 50 km.

10. On suppose que si les dénivelés positifs et négatifs sont égaux, alors la course est une **balade parfaite**. En utilisant cette idée, donner une requête SQL permettant d'afficher le lieu et la distance, et uniquement cela, de toutes les courses **qui ne sont pas** des balades parfaites.

11. Donner la requête SQL affichant la distance cumulée de tous les *trails* dont le dénivelé cumulé positif est supérieur à 1000 m (sur l'échantillon de tableau ci-dessus, la réponse serait 192.3).

N.B. : on rappelle que si `c` est une colonne d'une relation contenant des données numériques, la requête suivante : `SELECT sum(c) FROM rel` renvoie la somme des valeurs figurant dans la colonne `c` de la relation `rel`.