Devoir Commun d'ITC N°2

le 29/05/2026

MPSI & PCSI – Durée : 2 heures

Consignes

- Les codes doivent être présentés en mentionnant explicitement l'indentation, au moyen par exemple de barres verticales sur la gauche.
- Pour qu'un code soit compréhensible, il convient de choisir des noms de variables les plus explicites possibles.
- La numérotation des exercices (et des questions) doit être respectée et mise en évidence. Les résultats (hors questions purement informatiques) doivent être encadrés proprement.
- Il est important de numéroter correctement les pages des copies qui seront données à la correction. Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page.
- Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il convient de le signaler sur la copie et de poursuivre la composition en expliquant les raisons des initiatives qui ont été prises.
- Les candidats ne doivent avoir aucune communication entre eux ou avec l'extérieur durant l'épreuve. Aussi, l'utilisation des téléphones portables et, plus largement, de tout appareil permettant des échanges ou la consultation d'informations, est interdite.
- À l'issue de la durée prévue pour cette épreuve, les candidats doivent déposer le stylo et ne sont plus autorisés à écrire quoi que ce soit sur leur copie. Tout retard donne lieu à une pénalité sur la note finale.
- **L'usage de la calculatrice est interdit.**

- Les signatures des fonctions devront toutes être écrites et reprises sur votre copie. Une docstring devra être indiquée lorsque l'énoncé le demande.
- Vous pouvez (et même devez) utiliser les fonctions des questions précédentes, même si vous ne les avez pas toutes implémentées. Le barème en tiendra compte.

Remarques pour l'ensemble du DS

- Dans les calculs de complexité, on comptera comme opération élémentaire l'emploi des opérateurs `+`, `-`, `*`, `/`, `=` entre entiers ou flottants, ainsi que l'utilisation de `L.append` et `L.pop` pour les listes.
- Si `L` est une liste contenant l'élément `e`, l'instruction `L.remove(e)` enlève à `L` la première occurrence de `e`, comme ci-dessous :

```
>>> L = [2,1,5,1]
>>> L.remove(1)
>>> L
[2, 5, 1]
```

- Dans certains codes de l'énoncé, du document réponse et aussi du corrigé, certaines lignes de code sont affichées sur plusieurs lignes en raison de leur longueur. Il apparait ainsi un symbole de séparation \, suivi à la ligne suivante du symbole ↪. Ceci est juste une indication de coupure, et les deux lignes affichées constituent en réalité une seule ligne de code (sans les symboles).

Problème Gestion de randonnées

Lors d'une randonnée, des applications disponibles sur smartphones permettent d'enregistrer l'itinéraire parcouru. L'utilisation de ces données peut permettre d'analyser a posteriori le parcours effectué (distance, durée, dénivelés notamment) ou de planifier de nouvelles sorties. L'objectif du travail proposé est de découvrir différentes facettes de ces applications. Le sujet abordera les points suivants :

- mise en œuvre de différentes stratégies pour obtenir des informations sur le dénivelé effectué,
- planification de randonnées en utilisant des algorithmes de type Dijkstra.

Les données recueillies lors d'une randonnée sont généralement stockées dans un fichier au format GPX (pour GPS eXchange Format). Ce fichier est appelé trace GPX de la randonnée.

1 QUELQUES CALCULS DE DÉNIVELÉS

Nous allons tout d'abord nous intéresser aux données enregistrées par l'application lors d'une randonnée. En pratique, une application enregistre régulièrement les données fournies par le GPS du téléphone portable comme la latitude et la longitude exprimées en degrés ainsi que l'altitude (élévation) exprimée en mètres.

Le module `gpxpy` de Python permet de lire et extraire simplement les données de ce type de fichier.

1. Écrire une ligne de code permettant l'importation du module `gpxpy` de telle sorte que, si `f` est une fonction du module ayant un unique argument `f(x)`, l'instruction `f(x)` fonctionne .

Après lecture et traitement du fichier à l'aide du module `gpxpy`, l'itinéraire d'une randonnée est représenté par une liste de points où chacun de ces points est un triplet de trois flottants correspondant respectivement à la latitude, la longitude et l'altitude. Un itinéraire (ou une randonnée) est alors une succession de points. Le premier point d'un itinéraire sera le point de départ et le dernier le point d'arrivée.

Pour améliorer la lisibilité de la signature de nos fonctions, nous utiliserons le type `trpt` (pour track point ou point de la trace) pour représenter les triplets de flottants ainsi que le type `itineraire` pour les listes de points. Ainsi, on pourra introduire les variables `p0, p1, p2, p3` qui sont de type `trpt` et la variable `iti` qui est de type `itineraire` :

```
p0 = (43.331146, -1.627655, 261.00) # point de départ
p1 = (43.331055, -1.627895, 267.20) # point intermédiaire
p2 = (43.331047, -1.627927, 267.79) # point intermédiaire
p3 = (43.331040, -1.627966, 268.39) # point d'arrivée
iti = [p0, p1, p2, p3] # itinéraire
```

Rappelons (figure 1) qu'un point de la surface terrestre est défini par :

- sa latitude, notée ϕ, qui est la mesure angulaire entre l'équateur et ce point. Elle est représentée par un angle compris entre $-90°$ et $+90°$;
- sa longitude, notée λ, qui est la mesure angulaire entre le méridien de référence (méridien de Greenwich) et ce point. Elle est représentée par un angle compris entre $-180°$ et $+180°$;
- son altitude qui exprime la hauteur entre le niveau de la mer et le niveau du point. Elle est représentée par un réel et s'exprime en mètres.

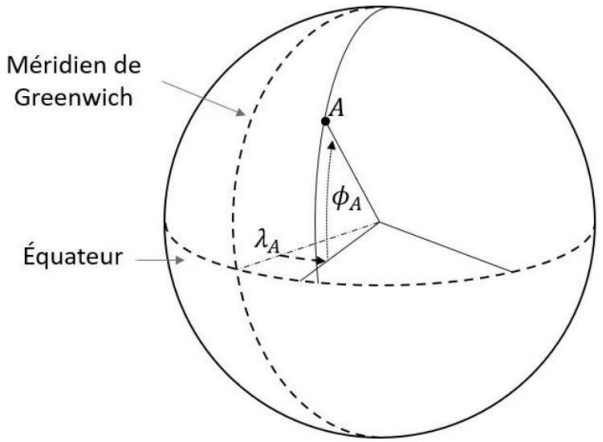

FIGURE 1 : Repérage, sur la surface du globe, d'un point A de latitude ϕ_A et de longitude λ_A

La syntaxe pour extraire les éléments d'un tuple est identique à celle utilisée pour les éléments d'une liste mais les tuples ne sont pas modifiables (ou mutables).

```
>>> p0 = (43.331146, -1.627655, 261.00) # point de départ
>>> p0[0]
43.331146
>>> (a, b, c) = p0
>>> a
43.331146
```

On considère la fonction `mystere`, dont l'argument `iti` est non vide :

```
def mystere(iti :itineraire) -> float:
    s = 0
    for i in range(len(iti)):
        (lat, long, alt) = iti[i]
        s = s + alt
    return s/len(iti)
```

2. 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 = (47.8750, 1.8759, 108)
>>> p2 = (47.8744, 1.8759, 102)
>>> p3 = (47.8748, 1.8761, 110)
>>> iti1 = [p0, p1, p2, p3]
>>> mystere(iti1)
```

3. On cherche à déterminer la complexité temporelle de la fonction `mystere` en fonction de la taille n de la liste `iti` passée en argument. Exprimer cette complexité sous la forme $C(n) = an^\alpha + b$, et préciser les valeurs des entiers a, b et α.
4. Écrire une fonction `altitude_maximale(iti:itineraire) -> float` qui, étant donnée une liste non vide `iti` de points, renvoie l'altitude maximale de l'itinéraire en mètres.

Le dénivelé global d'une randonnée est la différence entre l'altitude maximale et l'altitude du point de départ.

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

1.1 Premier calcul de dénivelé positif

Le dénivelé entre deux points successifs p_1 et p_2 d'un itinéraire est dit positif si la différence entre l'altitude de p_2 et l'altitude de p_1 est positive. On appelle alors dénivelé positif la différence entre ces deux altitudes. Le dénivelé positif cumulé d'une randonnée est la somme de tous les dénivelés positifs entre les points successifs du parcours. Ainsi, le dénivelé positif cumulé de l'exemple `iti1` précédent est égal à 16.

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

La méthode de mesure de l'altitude par le GPS est relativement imprécise en raison par exemple de la présence éventuelle d'une couverture nuageuse, d'un parcours sous des arbres... Or, lors du calcul du dénivelé positif cumulé, de faibles erreurs répétées peuvent induire une erreur conséquente sur le calcul cumulé. Par exemple, si le randonneur effectue une randonnée en bord de mer sur une plage d'altitude constante mais que le GPS effectue des mesures erronées pour donner une liste d'altitudes égale à $[0, -2, 2, -2, 2, -2, 2, -2, 2]$, le dénivelé positif cumulé calculé par la fonction précédente est égal à 16 mètres alors qu'il devrait être nul.

Nous allons envisager deux méthodes pour pallier ces imprécisions : le lissage des altitudes et l'utilisation d'altitudes de référence.

1.2 Lissage des altitudes

Le lissage d'une liste de longueur n des altitudes par moyenne glissante de pas p consiste à remplacer l'altitude du point d'indice i par la moyenne des altitudes des points d'indices $i, i+1, \dots, i+p-1$, pour tous les points ayant un indice $i \in [\![0, n-p]\!]$, les altitudes des points restants étant conservées à l'identique.

Par exemple, lorsque $p = 2$, la liste précédente sera remplacée par :

liste de départ	0	-2	2	-2	2	-2	2	-2	2
calcul	$\frac{0-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	$\frac{2-2}{2}$	$\frac{-2+2}{2}$	2
liste lissée	-1	0	0	0	0	0	0	0	2

Le dénivelé positif est alors de 3 mètres, ce qui est plus proche de la réalité de l'itinéraire.

7. À l'aide de la fonction `mystere`, écrire une fonction `alt_glissante(liste_alt:list, p:int) -> list` qui étant donnés une liste d'altitudes `liste_alt` et un entier `p`, crée une nouvelle liste contenant la moyenne glissante des altitudes avec un pas `p`.
8. Exprimer 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. Montrer que pour $p = n/10$, cette complexité est en $O(n^2)$.
9. Proposer une nouvelle version de `alt_glissante`, nommée `alt_glissante2`, qui réalise le même action que `alt_glissante`, mais dont la complexité est en $O(n)$.

1.3 Utilisation d'altitudes de référence

Une autre stratégie pour améliorer la précision sur les altitudes, une fois la randonnée effectuée et une connexion internet plus stable trouvée, consiste à se connecter à une base de référence qui, étant donné un point du globe, renvoie son altitude. Bien sûr, tous les points ne sont pas stockés dans la base. Nous supposerons que la surface du globe est quadrillée par une liste de latitudes et une liste de longitudes et qu'en chaque point de cette grille l'altitude a été mesurée précisément. Ces altitudes sont stockées dans un dictionnaire nommé `dem` (Digital Elevation Model) dont les clés sont des couples (latitude, longitude) et les valeurs sont les altitudes correspondantes.

On considère le code suivant :

```
lat_ref = []
long_ref = []
for (lat, long) in dem:
    lat_ref.append(lat)
    long_ref.append(long)
```

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

On considère le code suivant :

```
def auxiliaire(x, y):
    if x == [] : return y
    if y == [] : return x
    if x[len(x)-1] < y[len(y)-1] :
        val = y.pop()
    else :
        val = x.pop()
    z = auxiliaire(x,y)
    z.append(val)
    return z

def principal(x):
    if len(x) <= 1:
        return x
    else :
        m = len(x)//2
        x1 = principal(x[:m])
        y1 = principal(x[m:])
        z = auxiliaire(x1, y1)
        return z
```

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

12. Parmi les qualificatifs ci-dessous, indiquer celui(ceux) qui correspond(ent) au(x) type(s) de programmation utilisé(s) pour coder la fonction `principal` : impératif, récursif, glouton, dichotomique.

13. On admet que la fonction `auxiliaire` termine dans tous les cas. En utilisant un raisonnement par récurrence sur la taille de la liste `x`, démontrer que la fonction `principal` termine.

14. 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.

Par la suite, on suppose que les listes `lat_ref` et `long_ref` sont triées. Il faut maintenant déterminer le point du dictionnaire `dem` le plus proche d'un point donné.

On donne dans le document réponse une implémentation partielle de la fonction `ref(v, liste_ref)` qui, étant donné un flottant `v` et une liste non vide `liste_ref` de flottants triés par ordre croissant tels que `v` est compris entre le premier et le dernier élément de `liste_ref`, renvoie la valeur de la liste `liste_ref` la plus proche de `v`.

15. Sur le document réponse, compléter (en couleur) les lignes incomplètes de la fonction `ref`.

16. Utiliser les données précédentes pour écrire une fonction `standardise(iti: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.

2 ORGANISATION D'UN TREK

Un randonneur souhaite effectuer un long trek, c'est-à-dire une série de randonnées sur plusieurs jours. Il organise son parcours à partir d'un site qui propose différentes randonnées d'une journée. Chaque randonnée est définie par un niveau de difficulté allant de 1 (facile) à 5 (extrême). L'ensemble des données permet au randonneur d'établir un graphe où :

- les sommets représentent les points de départ / arrivée des randonnées,
- les arêtes représentent les randonnées possibles avec comme poids le niveau de la randonnée.

On suppose qu'il y a une unique randonnée qui relie 2 sommets du graphe. On suppose que les randonnées peuvent être effectuées du point de départ vers le point d'arrivée ou du point d'arrivée vers le point de départ sans que la difficulté ne soit modifiée (le graphe représentant les différentes randonnées sera donc non orienté).

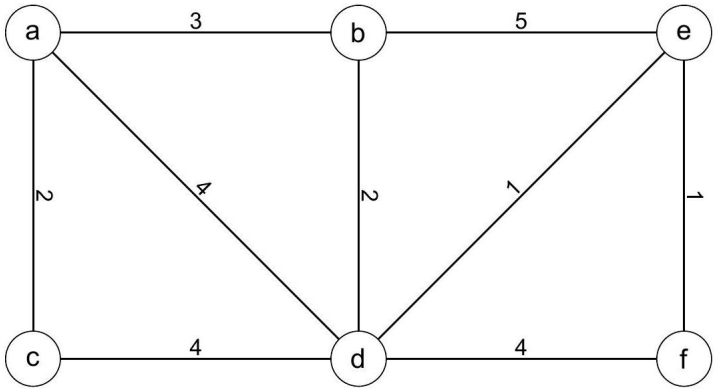

FIGURE 2 : Graphe G

Le graphe G de la figure 2 est représenté par le dictionnaire `G` défini de la manière suivante :

```
G = dict()
G['a'] = {'b':3, 'c':2, 'd':4}
G['b'] = {'a':3, 'd':2, 'e':5}
G['c'] = {'a':2, 'd':4}
G['d'] = {'a':4, 'b':2, 'c':4, 'e':1, 'f':4}
G['e'] = {'b':5, 'd':1, 'f':1}
G['f'] = {'d':4, 'e':1}
```

Le randonneur souhaite aller du point `a` au point `f` . Cependant, comme il n'est pas entraîné, il choisit de trouver le chemin dont la somme des difficultés des randonnées est la plus petite, quel que soit le nombre d'étapes.

Afin d'utiliser le vocabulaire habituellement employé dans les algorithmes de parcours de graphes, on utilisera le terme "distance" au lieu du terme "difficulté" et on cherchera donc à minimiser la "distance" (au sens de "difficulté" du trek).

2.1 Première idée : algorithme intuitif

Pour trouver son chemin, le randonneur exécute la fonction `mystere2` suivante.

```
1  def mystere2(graph:dict, Sd:str, Sf:str) -> tuple:
2      '''
3      Renvoie un chemin permettant de relier Sd à Sf ainsi que \
       ↪ la somme des difficultés d'un tel chemin.

4      graph : graphe représentant les randonnées disponibles
5      Sd : sommet de départ dans le graphe
6      Sf : sommet d'arrivée dans le graphe
7      '''
8      dejaVisites = [] # Liste des sommets déjà visités
9      sTraite = Sd # Initialisation du sommet à traiter
10     chemin = [sTraite] # Chemin choisi initialisé
11     diffChemin = 0 # Difficulté du chemin choisi
12     # Construction iterative du chemin
13     while sTraite != Sf:
14         dejaVisites.append(sTraite)
15         d = float('inf') # valeur représentant l'infini
16         for sommet in graph[sTraite]:
17             if sommet not in dejaVisites :
18                 if graph[sTraite][sommet] < d:
19                     sInter = sommet
20                     d = graph[sTraite][sommet]
21         diffChemin = diffChemin + d
22         chemin.append(sInter)
23         sTraite = sInter
24     return chemin, diffChemin
```

17. Expliquer le fonctionnement de la boucle itérative comprise entre les lignes 16 à 20. Parmi les qualificatifs ci-dessous, indiquer celui(ceux) qui correspond(ent) au(x) type(s) de programmation utilisé(s) pour coder la fonction `mystere2` : impératif, récursif, glouton, dichotomique.

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

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

2.2 Deuxième idée : l'algorithme de Dijkstra

Pour résoudre son problème, le randonneur décide d'appliquer l'algorithme de Dijkstra. Il réalise ainsi une fonction `dijkstra(graph :dict, Sd:str, Sf:str)`

`-> dict` qui prend en arguments :

- `graph`, représentation du graphe sous forme de dictionnaire de dictionnaires ;
- `Sd`, sommet de départ sous forme d'une chaîne de caractères ;
- `Sf`, sommet d'arrivée sous forme d'une chaîne de caractères.

et renvoie un dictionnaire dont les clés sont les sommets du graphe et les valeurs sont des couples dont :

- la première composante est la distance totale minimale du point de départ au sommet clé ;
- la seconde composante est le sommet précédent dans le graphe qui permet de réaliser la distance minimale entre le point de départ et le sommet clé.

L'implémentation rappelée ci-dessous de l'algorithme de Dijkstra utilise les quatre variables :

- `aVisiter` : liste des sommets qui doivent être visités ;
- `dejaVisites` : liste des sommets déjà visités ;
- `distance` : le dictionnaire qui sera renvoyé par la fonction ;
- `sTraite` : sommet dont on étudie les voisins pour mettre à jour les distances au point de départ.

On rappelle par ailleurs que la méthode `L.remove(elt)` permet de supprimer la première apparition de `elt` dans la liste `L`. Enfin, la fonction `Djikstra` fait appel à deux fonction `cherche_min` et `unpas`, détaillées ci-dessous.

```
1 def cherche_min(dico :dict, liste :list) -> str:
2     d = float("inf") # Initialisation
3     for s in liste: # parcours des éléments de la liste
4         if s in dico and dico[s][0] < d: # recherche de la \
      ↪ valeur minimale
5             d = dico[s][0]
6             sTraite = s
7     return sTraite
```

```
1 def unpas(graph :dict, s:str, distance :dict, \
  ↪ dejaVisites:list, aVisiter:list) -> None:
2     aVisiter.remove(s) # Mise a jour des sommets a visiter
3     dejaVisites.append(s) # Mise a jour des sommets déjà \
      ↪ visités
4     for v in graph[s]: # Mise a jour des distances au point de \
      ↪ depart
5         if v not in dejaVisites :
6             if v not in aVisiter:
7                 aVisiter.append(v)
8             ndistance = distance[s][0] + graph[s][v]
9             if v not in distance or ndistance < \
              ↪ distance[v][0]:
10                distance[v] = (ndistance, s)
```

```
1 def dijkstra(graph :dict, Sd:str, Sf:str) -> dict:
2     aVisiter = [Sd] # Liste des sommets à visiter
3     dejaVisites = [] # Liste des sommets déjà visités
4     distance = {Sd:(0, Sd)} # Dictionnaire des distances
5     sTraite = Sd # Premier sommet a visiter
6     while sTraite != Sf:
7         sTraite = cherche_min(distance, aVisiter)
8         unpas(graph, sTraite, distance, dejaVisites, aVisiter)
9     return distance
```

20. On effectue l'appel `dijkstra(G, 'a', 'f')` où le graphe G est défini dans la figure 2. Dans le tableau du document réponse 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 8). 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 .

21. Compléter (en couleur) le code fourni dans le document réponse qui a pour but de reconstruire un chemin qui réalise la distance optimale entre le point de départ `'a'` et le point d'arrivée `'f'`. On rappelle que la variable globale `G` a été définie précédemment.

22. Expliquer comment pourrait être diminué le nombre de tests d'appartenance à une liste, notamment des lignes 5 et 6, de la fonction `unpas`.

Pour la fin de cette sous-partie, on considère le graphe de la figure 3 où, pour simplifier, toutes les randonnées sont supposées de difficulté 1 . On suppose qu'une représentation de ce graphe par un dictionnaire est fournie dans une variable globale `G1` et que la liste des voisins est triée par ordre alphabétique.

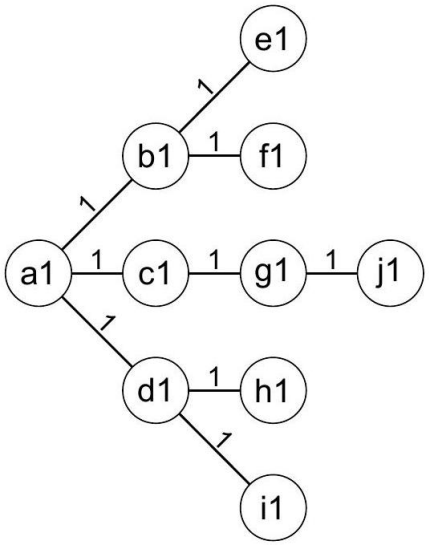

FIGURE 3 : Graphe G1

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

2.3 Troisième idée : l'algorithme de Dijkstra bidirectionnel

On constate à l'aide des deux questions précédentes que, en fonction de la structure du graphe, il est parfois préférable de chercher un chemin qui part du sommet d'arrivée vers le sommet de départ plutôt qu'un chemin qui part du sommet de départ vers le sommet d'arrivée.

L'algorithme de Dijkstra bidirectionnel combine ces deux stratégies : au cours de l'algorithme, on va effectuer successivement soit une recherche depuis le point de départ (appelée étape forward) soit une recherche depuis le point d'arrivée (appelée étape backward). À chaque étape, on choisit ainsi un sommet qui réalise le minimum de la distance soit au sommet de départ, soit au sommet d'arrivée et on y applique l'algorithme de Dijkstra classique.

Précisons l'algorithme. Notons $\mathscr{S}$ l'ensemble des sommets du graphe. Pour tout sommet $i \in \mathscr{S}$, on note $d\mathrm{F}(i)$ (respectivement $d\mathrm{B}(i)$) la distance minimale actuellement calculée entre le sommet de départ (respectivement d'arrivée) et le sommet i. Ces distances sont actualisées au cours de l'algorithme et valent initialement l'infini. Nous allons maintenir, c'est-à-dire mettre à jour pendant l'exécution de l'algorithme, plusieurs variables :

- `aVisiterF`, `dejaVisitesF` : associées à l'algorithme de Dijkstra partant du point de départ (partie Forward),
- `aVisiterB`, `dejaVisitesB` : associées à l'algorithme de Dijkstra partant du point d'arrivée (partie Backward),
- `Fmin` $= \min\{d\mathrm{F}(i), i \in$ aVisiterF$\}$: distance minimale du sommet de départ à un sommet non encore visité par l'algorithme forward,
- `Bmin` $= \min\{d\mathrm{B}(i), i \in$ aVisiterB$\}$: distance minimale du sommet d'arrivée à un sommet non encore visité par l'algorithme backward,
- `BFmin` $= \min\{d\mathrm{F}(i) + d\mathrm{B}(i), i \in \mathscr{S}\}$: longueur minimale d'un chemin reliant le sommet de départ au sommet d'arrivée,
- `distanceF` (respectivement `distanceB`) : dictionnaire dont les clés sont les sommets et les valeurs sont des couples dont la première composante est la plus petite distance d'un chemin qui relie le sommet depuis le point de départ (respectivement depuis le point d'arrivée) et la seconde est le sommet précédent dans un chemin qui réalise cette distance.

À chaque étape :

- Si `BFmin` $\leq$ `Fmin` + `Bmin`, l'algorithme termine : tout chemin qui réalise le minimum `BFmin` est un chemin optimal.
- Sinon,
  - si `Fmin` < `Bmin`, on choisit un sommet qui réalise `Fmin` et on applique une étape forward de Dijkstra depuis ce sommet, c'est-à-dire qu'on appelle la fonction `unpas` avec les variables forward en mettant à jour les variables `dejaVisitesF`, `aVisiterF` et `distanceF`,
  - si `Bmin` < `Fmin`, on choisit un sommet qui réalise `Bmin` et on applique une itération de l'algorithme de Dijkstra backward depuis ce sommet, c'est-à-dire qu'on appelle la fonction `unpas` avec les variables backward en mettant à jour les variables `dejaVisitesB`, `aVisiterB` et `distanceB`,
  - si `Bmin` = `Fmin`, on choisit un sommet qui réalise ce minimum dans l'ensemble qui contient le moins d'éléments parmi `aVisiterF` et `aVisiterB` et on réalise une étape de Dijkstra à partir de ce sommet. Dans le cas où les deux ensembles contiennent le même nombre d'éléments, on préférera une recherche forward.
- On actualise `Fmin`, `Bmin` et `BFmin` en parcourant l'ensemble des sommets pour lesquels un chemin entre ce sommet et les sommets d'arrivée et de départ a déjà été calculé.

24. Compléter la dernière ligne des tableaux du document réponse qui recense les étapes successives de l'algorithme de Dijkstra bidirectionnel sur le graphe G de la figure 2 avec `'a'` comme sommet de départ et `'f'` comme sommet d'arrivée, en expliquant en quelques lignes la démarche suivie. 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.

25. 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.

On donne dans le document réponse une implémentation partielle de 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.

26. Compléter (en couleur) la fonction `dijkstra_bidirectionnel` fournie dans le document réponse.

NB : la fonction `min(L:list)` renvoie l'élément minimal d'une liste `L`.

Document réponse

Nom :

Classe :

Question 15

```
def ref(valeur :float, liste_ref:list) -> float:
    ''' Renvoie l'élément de 'liste_ref' le plus proche de 'v' '''
    # on détermine le plus petit intervalle [ind_deb,ind_fin] tel que
    # liste_ref[ind_deb] <= v <= liste_ref[ind_fin]
    # par une méthode de dichotomie
    ind_deb = _______________
    ind_fin = _______________
    while ind_deb < ind_fin-1:
        k = _________________
        if valeur <= liste_ref[k] :
            ind_fin = _________________
        else :

            _________________
    # on détermine la valeur de 'liste_ref' la plus proche de 'v'
    if liste_ref[ind_fin]-valeur < valeur-liste_ref[ind_deb]:
        return __________
    else :
        return __________
```

Question 20

		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	'c'	0,'a'	3,'a'	2,'a'	4,'a'	x	x	['b','d']
3								
4								
5								

Question 21

```
# On applique l'algorithme de Dijkstra :
sInit, sFin = 'a', 'f'
distance = dijkstra(G, sInit, sFin)
# On construit la liste du chemin en partant de la fin
s = sFin
chemin = [s]
while ____________:
    ________________
    ____________________
# On remet le chemin dans l'ordre du début vers la fin
chemin.reverse()
print("Un chemin de ", sInit, " à ", sFin, " est : ", chemin)
print("La difficulté minimale est de : ", ____________________)
```

Question 24

		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'|∞,'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							

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					

Question 26

```python
def dijkstra_bidirectionnel(graph :dict, Sd:str, Sf:str) -> float:
    aVisiterF = [Sd] # Liste des sommets à visiter Forward
    dejaVisitesF = [] # Liste des sommets déjà visités Forward
    aVisiterB = _____________ __# Liste des sommets à visiter Backward
    dejaVisitesB = _____________ # Liste des sommets déjà visités Backward
    # Dictionnaire des distances Forward
    distanceF = {s:(float('inf'),Sd) for s in graph}
    # Dictionnaire des distances Backward
    distanceB = {s:(float('inf'),Sf) for s in graph}
    distanceF[Sd] = (0, Sd)
    distanceB[Sf] = (0, Sf)
    Fmin, Bmin, BFmin = 0, 0, float("inf")
    while _________________________:
        if Fmin < Bmin or \
                (Fmin == Bmin and len(aVisiterF) <= len(aVisiterB)):
            sTraite = cherche_min(distanceF, aVisiterF)
            unpas(graph, sTraite, distanceF, dejaVisitesF, aVisiterF)
        else :
            sTraite = cherche_min(distanceB, aVisiterB)
            unpas(graph, sTraite, distanceB, dejaVisitesB, aVisiterB)
        Fmin = min([_____________[v][0] for v in aVisiterF if v in distanceF])
        Bmin = min([_____________[v][0] for v in aVisiterB if v in distanceB])
        L = [_____________________ for v in distanceB if v in distanceF]
        if L == [] :
            BFmin = float("inf")
        else :
            BFmin = min(L)
    return BFmin
```

Correction
du Devoir Commun d'ITC N°2
MPSI & PCSI

Solution

1. La ligne qui permet l'importation est : `from gpxpy import *`
2. La fonction `mystere` renvoie la moyenne des altitudes, soit ici $(100+102+110+108)/4 = 105$
3. On a une affectation, puis n itérations avec 5 opérations élémentaires (trois affectations, une somme et une affectation), et enfin une division. On a donc $C(n) = 1+5n+1 = 5n+2$. On a donc $a = 5, b = 2, \alpha = 1$.
4.
```python
def altitude_maximale(iti:itineraire)->float:
    alt_max = iti[0][2]
    for p in iti[1:]:
        if p[2] > alt_max:
            alt_max = p[2]
    return alt_max
```
5.
```python
def denivele_global(iti:itineraire) -> float:
    alt_max = altitude_maximale(iti)
    return alt_max-iti[0][2]
```
6.
```python
def denivele_positif_cumule(iti:itineraire) -> float:
    den_pos = 0
    n = len(iti)
    for i in range(n-1):
        if iti[i+1][2]-iti[i][2] > 0:
            den_pos += iti[i+1][2]-iti[i][2]
    return den_pos
```
7.
```python
def alt_glissante(liste_alt:list, p:int) -> list:
    n = len(liste_alt)
    alt_liss = []
    # boucle pour les points lissés
    iti = [(0,0,liste_alt[i]) for i in range(n)]
    for i in range(n-p+1):
        # calcul de la moyenne des altitudes des 'p' points \
        ↪ après le point d'indice 'i'
        moy = mystere(iti[i:i+p])
        alt_liss.append(moy)
    # on complète la liste avec les points non lissés restants
    alt_liss += liste_alt[n-p+1:]
    return alt_liss
```
8. On a deux initialisations, et $3n$ affectations pour l'initialisation de la liste `iti`, puis dans chaque itération sur i, un appel à la fonction `mystere` sur une liste de taille p (soit $5p+2$ opérations) et deux ajouts, et enfin $p-1$ ajouts dans la liste (en dehors de la boucle). On a donc $C = 2+3n+\sum_{i=0}^{n-p}(5p+4)+p-1$, soit après calcul $C = 5np-5p^2+7n+2p+5$. Pour $p = n/10$, on a bien $C = O(n^2)$.
9. La forte complexité vient du fait que le calcul de la moyenne par l'appel à `mystere` conduit à des calculs de sommes redondants. On peut faire évoluer la somme des p points intervenant dans le lissage en faisant uniquement deux opérations (au lieu de p précédemment) à chaque itération. On peut ainsi proposer
```python
def alt_glissante2(liste_alt:list, p:int) -> list:
    n = len(liste_alt)
    alt_liss = []
    # somme des 'p' premières altitudes
    s = 0
    for i in range(p):
        s = s+liste_alt[i]
    # boucle pour les points lissés
    for i in range(n-p+1):
        # ajout de l'altitude moyennée
        alt_liss.append(s/p)
        # mise à jour de la somme
        s = s- liste_alt[i]+liste_alt[i+p]
    # on complète la liste avec les points non lissés restants
    alt_liss += liste_alt[n-p+1:]
    return alt_liss
```
Pour cette fonction, on a dans chaque itération de la boucle `for` uniquement 5 opérations, d'où en tenant compte du premier calcul de s et des ajouts finaux, $C = 3+2p+5(n-p+1)+p-1 = 7+5n-2p$. Comme $p \leq n$, on a bien $C = O(n)$.
10. Vues les initialisations, `lat_ref` et `long_ref` sont des listes. Elles contiennent respectivement l'ensemble des latitudes et longitudes de référence qui constituent les clés de `dem`.
11. Les tests de début de la fonction `auxiliaire` montrent que `x,y` sont des listes. La fonction `auxiliaire` renvoie `z`, issu d'un appel récursif. L'examen des cas terminaux (deux premières lignes) montrent que c'est une liste. Ainsi la signature est :

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

De même, dans la fonction `principal`, `x` est une liste, et renvoie le résultat de la fonction `auxiliaire`, qui est donc une liste. Ainsi, la signature est :

```
principal(x:list)->list
```

12. `principal` est de nature récursif, dichotomique.

13. la fonction `principal` étant récursive, on peut faire un raisonnement par récurrence sur la taille de la liste `x`. Soit la propriété « P_n : la fonction `principal` termine si elle est appliquée à une liste de n éléments ». Pour $n = 0$ et $n = 1$, la propriété est vraie, car cela correspond au cas terminal. Supposons que le fonction termine pour toute liste le longueur $k \leq n$ (récurrence forte), et considérons un appel de la fonction sur une liste de taille $n+1$. Dans ce cas, les sous listes `x[:m]` et `x[m:]` sont de taille $n//2$ ou $n//2-1$ suivant la parité de n, ce qui est toujours inférieur à n. D'après l'hypothèse de récurrence `principal(x[m:])` et `principal(x[m:])` terminent donc `x1,y1` sont bien définies. Comme la fonction `auxiliaire` termine, `z` existe et finalement la fonction `principal` termine.

14. La fonction `auxiliaire` permet de réaliser la fusion de listes triées. Ainsi, on a affaire à un tri fusion.

15.
```
def ref(v:float, liste_ref:list) -> float:
    ''' Renvoie l'élément de 'liste_ref' le plus proche de \
    ↪ 'v' '''
    # on détermine le plus petit intervalle [ind_deb,ind_fin] \
    ↪ tel que
    # liste_ref[ind_deb] <= v <= liste_ref[ind_fin]
    # par une méthode de dichotomie
    ind_deb = 0
    ind_fin = len(liste_ref)-1
    while ind_deb < ind_fin-1:
        k = (ind_deb+ind_fin)//2
        if v <= liste_ref[k] :
            ind_fin = k
        else :
            ind_deb = k
    # on détermine la valeur de 'liste_ref' la plus proche de \
    ↪ 'v'
    if liste_ref[ind_fin]-v < v-liste_ref[ind_deb]:
        return liste_ref[ind_fin]
    else :
        return liste_ref[ind_deb]
```

16.
```
def standardise(iti:itineraire)->itineraire:
    n_iti = [] # nouvel itinéraire
    for p in iti:
        lat, long, alt = p
        # coordonnées du point de référence le plus proche de \
        ↪ 'p'
        n_lat = ref(lat,lat_ref)
        n_long = ref(long,long_ref)
        # altitude du point de référence le plus proche de 'p'
        n_alt = dem[(n_lat,n_long)]
        # modification de l'altitude de 'p'
        n_iti.append((lat,long,n_alt))
    return n_iti
```

17. Les lignes indiquées permettent de trouver, pour un sommet `sTraite`, le sommet voisin le plus proche parmi les voisins de `sTraite` non encore visités. A la fin de cette boucle, `sInter` contient ce plus proche voisin, et `d` contient la distance entre `sTraite` et `sInter`. Cet algorithme choisit à chaque étape le voisin le plus proche, sans vision globale du meilleur itinéraire : il s'agit d'un algorithme impératif, glouton.

18. En parcourant le graphe en se déplaçant à chaque fois vers plus proche voisin du sommet `sTraite`, on obtient le chemin `['a','c','d','e','f']`, de difficulté 8.

19. Pour le graphe `G`, le chemin `['a','d','e','f']` est possible, et sa difficulté est de 6, inférieure à 8. La fonction `mystere2` ne renvoie pas forcément le chemin de difficulté minimale.

20. On complète la tableau en appliquant l'algorithme de Djikstra :

		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	'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']

La dernière itération affecte le sommet `'f'` à `sTraite`, ce qui entraine la sortie de la boucle `while`.

21. Le code permettant de reconstruire le chemin ainsi que l'affichage du résultat est :

```
# On applique l'algorithme de Dijkstra :
sInit, sFin = 'a', 'f'
distance = dijkstra(G, sInit, sFin)
# On construit la liste du chemin en partant de la fin
s = sFin
```

```
chemin = [s]
while chemin[-1} != sInit:
    s = distance[s][1]
    chemin.append(s)
# On remet le chemin dans l'ordre du début vers la fin
chemin.reverse()
print("Un chemin de ", sInit, " à ", sFin, " est : ", chemin)
print("La difficulté minimale est de : ", distance[sFin][0])
```

22. La recherche de l'appartenance d'un élément dans une liste de taille n a une complexité en $O(n)$. Il serait préférable de stocker l'information sur le caractère « déjà visité » ou « à visiter » de chaque sommet dans des dictionnaires dont les clés sont les sommets et les valeurs associées un booléen. L'accès à cette information serait alors de complexité $O(1)$.

23. L'appel `dijkstra(G1, 'a1', 'j1')` entraine la visite des sommets de distance 1 de `'a1'`, soit `'b1'`,`'c1'`,`'d1'`, puis il visite les sommets de distance 2, soit `'e1'`,`'f1'`,`'g1'`,`'h1'`,`'i1'`, et enfin on termine par `'i1'`. Tous les sommets sont visités. Pour l'appel `dijkstra(G1, 'j1', 'a1')`, les seuls sommets visités sont `'g1'`,`'c1'`,`'a1'`. Ce deuxième appel est beaucoup plus rapide.

24. Lors de l'itération n°4, pour la recherche forward, parmi les sommets `'b'`,`'c'`,`'d'`, la distance minimale au sommet de départ vaut 2 (pour le sommet `'c'`), et pour la recherche backward, parmi les sommets `'d'`,`'b'`, la distance minimale au sommet d'arrivée vaut également 2 (pour le sommet `'d'`). On choisit donc la recherche backward car la liste `aVisiterB` est plus petite que `aVisiterF`. Les voisins de `'d'`, non visités par la recherche bavkward sont `'a'`,`'b'`,`'c'`. On applique la mise à jour des distances et on obtient pour le tableau distance :

		distanceF|distanceB					
Etape	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'	∞,'a'|1,'f'	∞,'a'|0,'f'

et pour le tableau des autres variables :

Etape	Fmin	Bmin	BFmin	aVisiterF	aVisiterB
4	2	4	6	['b','c','d']	['b','a','c']

25. A l'issu de l'itération n°2, le sommet `'d'` a été atteint par les deux recherches, pour une valeur `BFmin` égale à 8, mais le chemin obtenu `'a'`,`'d'`,`'f'` n'est pas de distance minimale (car le chemin `'a'`,`'d'`,`'e'`,`'f'` est plus court).

26. Code complété :

```
def dijkstra_bidirectionnel(graph :dict, Sd:str, Sf:str) \
    -> float:
    aVisiterF = [Sd] # Liste des sommets à visiter Forward
    dejaVisitesF = [] # Liste des sommets déjà visités \
    Forward
    aVisiterB = [Sf] # Liste des sommets à visiter Backward
    dejaVisitesB = [] # Liste des sommets déjà visités \
    Backward
    # Dictionnaire des distances Forward
    distanceF = {s:(float('inf'),Sd) for s in graph}
    # Dictionnaire des distances Backward
    distanceB = {s:(float('inf'),Sf) for s in graph}
    distanceF[Sd] = (0, Sd)
    distanceB[Sf] = (0, Sf)
    Fmin, Bmin, BFmin = 0, 0, float("inf")
    while BFmin > Fmin + Bmin :
        if Fmin < Bmin or \
                (Fmin == Bmin and len(aVisiterF) <= \
                len(aVisiterB)):
            sTraite = cherche_min(distanceF, aVisiterF)
            unpas(graph, sTraite, distanceF, dejaVisitesF, \
            aVisiterF)
        else :
            sTraite = cherche_min(distanceB, aVisiterB)
            unpas(graph, sTraite, distanceB, dejaVisitesB, \
            aVisiterB)
        Fmin = min([distanceF[v][0] for v in aVisiterF if \
        v in distanceF])
        Bmin = min([distanceB[v][0] for v in aVisiterB if \
        v in distanceB])
        L = [distanceF[v][0]+distanceB[v][0] for v in \
        distanceB if v in distanceF]
        if L == [] :
            BFmin = float("inf")
        else :
            BFmin = min(L)
    return BFmin
```