DS5info

17 mai 2025

La calculatrice est interdite. L'usage de tout document est interdit. La rigueur, le soin, la présentation seront fortement pris en compte dans la notation. Les résultats de chaque question seront soulignés ou encadrés. Veuillez à bien indenter avec deux grands carreaux (ou quatre petits) par indentation.

L'épreuve est à traiter en langage Python. Les différents algorithmes doivent respecter les éléments de syntaxe du langage (les brouillons ne sont pas acceptés).

La réponse ne doit pas se limiter à la rédaction de l'algorithme sans explication, les programmes doivent être expliqués et commentés de manière raisonnable.

PROBLÈME - Gestion de randonnées

Les fonctions sont définies avec leur signature dans le sujet : `ma_fonction(arg1:type1, arg2:type2) -> type3`. Cette notation permet de définir une fonction qui se nomme `ma_fonction` qui prend deux arguments en entrée, `arg1` de type `type1` et `arg2` de type `type2`. Cette fonction renvoie une valeur de type `type3`. Il ne faut pas recopier les signatures des fonctions dans votre copie, il faut écrire directement :

```
def ma_fonction(arg1, arg2):
    # liste d'instructions
```

Introduction

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...) ou de planifier de nouvelles sorties. 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.
Nous allons maintenant nous intéresser plus particulièrement 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. La **figure 1** présente un extrait du fichier trace GPX.

```
<?xml version="1.0" encoding="UTF-8"?>
<gpx
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1 http://www.topografix.com/GPX/11.xsd"
  xmlns="http://www.topografix.com/GPX/1/1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <trk>
    <name>Randonne</name>
    <type>Marche</type>
    <trkseg>
      <trkpt lat="43.331146650016307830810546875" lon="-1.627655038610100746154785156 25">
        <ele>261</ele>
        <time>2022-08-01T07:37:39.000Z</time>
      </trkpt>
      <trkpt lat="43.331055035814642906188964843 75" lon="-1.627895934507250785827636718 75">
        <ele>267.20001220703125</ele>
        <time>2022-08-01T07:37:52.000Z</time>
      </trkpt>
      <trkpt lat="43.331047911196947097778320312 5" lon="-1.627927953377366065979003906 25">
        <ele>267.79998779296875</ele>
        <time>2022-08-01T07:37:54.000Z</time>
      </trkpt>
      <trkpt lat="43.331040535122156143188476562 5" lon="-1.627966091036796569824218 75">
        <ele>268.399993896484375</ele>
        <time>2022-08-01T07:37:57.000Z</time>
      </trkpt>
```

Figure 1 - Exemple de fichier trace GPX

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`.

Un *parcours* (ou une randonnée) est alors une succession de points. 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. 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 points ainsi que le type `itineraire` pour les listes de points. Ainsi, à partir du fichier de la **figure 1** 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 depart
p1 = (43.331055,-1.627895,267.20) # point intermediaire
p2 = (43.331047,-1.627927,267.79) # point intermediaire
p3 = (43.331040,-1.627966,268.39) # point d arrivee
iti = [p0, p1,p2, p3] # itineraire
```

Rappelons (**figure 2**) 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.

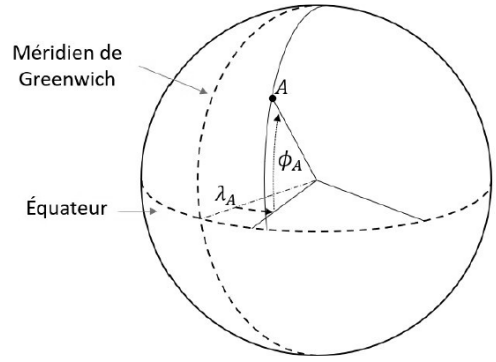

Figure 2 - Repérage, sur la surface du globe, d'un point A de latitude ϕ_A et de longitude λ_A

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

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

4. 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.

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 donné une liste `iti` de points, renvoie le dénivelé global de la randonnée.

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.

6. 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.

La méthode de mesure de l'altitude par le GPS est relativement imprécise due à 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.

Lissage des altitudes

Le *lissage* d'une liste de longueur n des altitudes par moyenne glissante de pas p consiste à remplacer l'altitude au point numéroté i par la moyenne des altitudes des points numérotés $i, i+1, \dots, i+j$ où $j = \min\{p-1, n-i-1\}$. 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. 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.
8. 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.

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)
```

On supposera dans la suite que les valeurs -90 et 90 sont dans `lat_ref` et que les valeurs -180 et 180 sont dans `long_ref`.

9. 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[0 :m])
        y1 = principal(x[m:len(x)])
        z = auxiliaire(x1, y1)
        return z
```

10. Précisez la signature des fonctions `auxiliaire` et `principal`. La réponse doit être justifiée.
11. Sans justification, parmi la liste d'éléments ci-dessous, donner les éléments qui correspond(ent) au(x) type(s) de programmation utilisé(s) pour coder la fonction `principal`.

▫ Itératif	▫ Parcours en profondeur\largeur	▫ Récursif
▫ Programmation dynamique	▫ Algorithme glouton	▫ Diviser pour régner

12. Justifier brièvement que, étant donné une liste `x`, l'appel `principal(x)` termine.
13. 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 une implémentation partielle de la fonction `ref(valeur, liste_ref)` qui, étant donné un flottant valeur et une liste non vide de flottants triés par ordre croissant `liste_ref` tels que valeur est strictement compris entre le premier et le dernier élément de `liste_ref`, renvoie la valeur de la liste `liste_ref` la plus proche de valeur.

```
1 def ref(valeur:float, liste_ref:list) -> float:
2     # on détermine ind_deb et ind_fin tels que
3     # liste_ref[ind_deb]<valeur<=liste_ref[ind_fin]
4     # avec une méthode par dichotomie
5     ind_deb = .......
6     ind_fin = .......
7     while ind_deb < ind_fin - 1:
```

```
8           k = .....
9           if valeur <= liste_ref[k] :
10              ind_fin = .....
11          else :
12              .....
13      # on détermine le plus proche
14      if liste_ref[ind_fin]-valeur<valeur-liste_ref[ind_deb]:
15          return .....
16      else :
17          return .....
```

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