

Devoir surveillé d'informatique

 Devoir
01

Durée : 2h. Aucun document ou appareil électronique autorisé.

Préambule

Le sujet contient deux parties indépendantes et on pourra considérer que chaque partie constitue un fichier Python. Il est sur 4 pages.

Dans chaque partie, une fonction codée précédemment peut être appelée plus tard par vos autres fonctions. Il est autorisé (et même encouragé) d'écrire des fonctions auxiliaires qui ne sont pas demandées explicitement par le sujet, pour les utiliser et faciliter votre écriture de code : on veillera simplement à énoncer clairement leur objectif. L'utilisation d'une fonction non standard de Python (set, count, index, ...) sera pénalisée, et le code considéré comme partiellement faux.

On fera attention à préciser l'indentation dans votre code, à l'aide de barres à gauche de votre code. Un code sans barre d'indentation sera considéré comme partiellement faux par défaut.

Partie 1 : algorithmique et arithmétique

Exercice 1 : nombres premiers et diviseurs

Un nombre entier positif n est dit premier s'il admet exactement deux diviseurs entiers positifs distincts, 1 et lui-même. Par exemple, 17 est premier, mais 18 ne l'est pas, car $18 = 2 \times 9$. On note que 1 admet exactement un seul diviseur entier positif, et n'est donc pas premier.

1. Pour établir la liste des diviseurs d'un nombre n , on initialise une liste vide, puis on parcourt les entiers de 1 à n inclus : si l'un des entiers divise n , on l'ajoute à la liste initiale. À la fin du parcours, on renvoie la liste obtenue jusque-là.
Écrire une fonction `liste_diviseurs(n)` qui renvoie la liste des diviseurs de n . Par exemple, pour $n = 18$, la fonction renverra `[1, 2, 3, 6, 9, 18]`.
2. Écrire une fonction `test_premier_1(n)` qui utilise la fonction `liste_diviseurs` pour tester si n est premier, en vérifiant que la liste des diviseurs de n a les bonnes propriétés.
3. Montrer que n n'est pas premier ssi n admet un diviseur compris entre 2 et $\lfloor \sqrt{n} \rfloor$.
4. Les fonctions `sqrt` et `floor`, présentes dans la bibliothèque `math`, permettent respectivement d'obtenir la racine carrée et la partie entière inférieure d'un flottant. Donner la commande permettant d'utiliser ces deux fonctions dans notre code. Cette commande devrait-elle raisonnablement écrite dans le fichier ou dans l'invite de commandes?
5. À partir de la question précédente, écrire une fonction `test_premier_2(n)` pour tester si n est premier en un nombre proportionnel à $\sqrt{n}$ d'opérations élémentaires (+, −, ×, ÷, mod, √, ⌊, test d'égalité, affectation). On ne justifiera pas la borne sur le nombre d'opérations élémentaires. On vérifiera la cohérence des notations avec la réponse à la question précédente.

Exercice 2 : nombres parfaits

On dit qu'un nombre entier est parfait s'il est la somme de ses diviseurs stricts. Par exemple, les diviseurs stricts de 6 sont 1, 2 et 3, et $1 + 2 + 3 = 6$: donc 6 est parfait.

1. Déterminer si 16 et 28 sont parfaits.
2. Écrire une fonction `est_parfait(n)` qui renvoie `True` si n est parfait, et `False` sinon.

3. Écrire une fonction `petits_parfaits_pairs(n)` qui renvoie une liste contenant les n plus petits nombres parfaits pairs.
4. Les nombres parfaits pairs sont bien connus, grâce au théorème d'Euclide-Euler :

Théorème 1 : théorème d'Euclide-Euler

Un nombre N pair est parfait si, et seulement si, il existe p un nombre premier tel que :

- $2^p - 1$ est premier;
- $N = 2^{p-1} \times (2^p - 1)$.

Écrire une fonction `petits_parfaits_pairs_2(n)` qui renvoie une liste contenant les n premiers nombres parfaits pairs, en utilisant la forme close donnée par le théorème d'Euclide-Euler.

Exercice 3 : factorisation

1. Pour n un entier strictement positif et p un nombre premier, on appelle valuation p -adique de n le nombre de fois que n est divisible par p . Par exemple, $63 = 3 \times 3 \times 7$ est divisible 2 fois par 3, 1 fois par 7 et 0 fois par 11.
Écrire une fonction `valuation(n,p)` qui renvoie la valuation p -adique de n .
2. Écrire une fonction `factorisation(n)` qui renvoie la factorisation de n en facteurs premiers sous forme de liste de couples (nb premier, puissance) : par exemple, `factorisation(63)` renverra `[(3,2), (7,1)]`.
3. Écrire une fonction `defactorisation(l)` qui renvoie un entier n à partir d'une liste de couples l : par exemple, `defactorisation([(3,2), (7,1)])` renverra 63.

Partie 2 : le métro de Paris

Le métro de Paris est constitué, à l'heure actuelle, de 321 stations. L'objectif de cette partie est d'écrire plusieurs fonctions permettant d'extraire des données sur le métro de Paris. *Aucune connaissance spécifique sur le métro de Paris n'est requise pour cet exercice.*

On suppose que les données du métro sont enregistrées dans une variable globale `metro`, qui est une liste de tuples, chaque tuple représentant une station de métro. Une station est enregistrée sous forme d'un tuple (nom, abscisse, ordonnée, lignes) : le nom est une chaîne de caractères, l'abscisse et l'ordonnée sont en mètres par rapport au point zéro de Paris sur le parvis de Notre-Dame, et les lignes sont une liste d'entiers. Par exemple, la station Odéon est représentée par le tuple `("Odéon", -652, -186, [4, 10])`. Voici le début de `metro` :

```
metro = [("Odéon", -652, -186, [4, 10]), ("Concorde", -1913, 1411, [1, 8, 12]), ("Temple", 927, 1489, [3]), ...]
```

On a alors les résultats suivants :

- `metro[2]` vaut `("Temple", 927, 1489, [3])`;
- `metro[2][0]` vaut `"Temple"`;
- `metro[2][1]` vaut 927, l'abscisse en mètres (et `metro[2][2]` l'ordonnée en mètres);
- `metro[2][3]` vaut `[3]`, la liste des stations de métro disponibles à la station (ici, seulement la ligne 3).

Exercice 4

Écrire une fonction `est_dedans(elem, tab)` qui renvoie `True` si `elem` est un élément du tableau `tab`, et `False` sinon.

Exercice 5

On vous donne le code suivant, qui est censé renvoyer le nom de la station la plus à l'ouest de Paris. Ce code ne fonctionne pas comme souhaité : il renvoie l'ensemble des données de la station la plus au nord. Réécrire la fonction pour qu'elle fonctionne comme escomptée.

```
def plus_a_louest():
    ind = 0
    for i in range(len(metro)) :
        if metro[i][2] > metro[ind][2] :
            ind = i
    return metro[ind]
```

Exercice 6

1. Écrire une fonction `noms_stations()` qui renvoie un tableau avec les noms de toutes les stations.
2. Écrire une fonction `nom_le_plus_long()` qui renvoie le nom de station le plus long.
3. Écrire une fonction `indice_station(nom)` qui, étant donné une chaîne de caractères, renvoie l'indice correspondant à la station de métro en question. Par exemple, `indice_station("Temple")` renverra 2. Si aucune station n'a de nom correspondant, on renverra -1.
4. Écrire une fonction `distance(i1, i2)` qui renvoie la distance en mètre entre les stations de métro d'indices `i1` et `i2`.
5. Les stations "Boulogne-Pont de Saint Cloud" et "Gallieni" sont particulièrement éloignées (15,5 km) : on se demande s'il y a plus éloigné encore.
Écrire une fonction `les_plus_eloignees()` qui renvoie les noms des deux stations les plus éloignées.

Exercice 7

Pour se repérer sur le métro de Paris, il est important de savoir si on peut aller facilement d'une station à l'autre.

1. Écrire une fonction `direct(s1, s2)` qui renvoie `True` si on peut aller sans changement de la station `s1` à la station `s2` (et `False` sinon) (`s1` et `s2` sont les noms des stations).
2. Écrire une fonction `en_un_changement(s1, s2)` qui renvoie `True` si on peut aller en au plus un changement de la station `s1` à la station `s2` (et `False` sinon).

Exercice 8

1. Écrire une fonction `liste_lignes()` qui renvoie la liste de toutes les lignes répertoriées dans `metro` (sans doublons, pas forcément triée).
2. Écrire une fonction `ligne(k)` qui renvoie la liste des noms des stations de la ligne `k` dans un ordre quelconque.
3. On cherche à reconstituer les lignes de métro à partir des données de `metro`. Pour retrouver l'ordre sur une ligne de métro précise, nous allons procéder selon l'algorithme suivant :
 - (1) on sélectionne la station de la ligne la plus éloignée du centre de Paris, qui sera considérée comme un des terminus de la ligne, et l'ajouter comme premier élément d'une liste;
 - (2) on détermine ensuite quelle station de la ligne, parmi celles qui ne sont pas encore dans la liste, est la plus proche de la dernière station ajoutée, et on l'ajoute à la liste;
 - (3) on répète l'étape (2) jusqu'à ce que toutes les stations de la ligne aient été ajoutées à la liste.

Par exemple, considérons (une partie de) la ligne 6 ["Pasteur", "Duplex", "Place d'Italie", "Glacière", "Daumesnil", "Nation", "Étoile"]. On commence par déterminer que "Nation" est la station la plus éloignée du zéro de Paris : ce sera le premier terminus de notre ligne. Puis on calcule que "Daumesnil" est la station de la ligne la plus proche de "Nation" : ce sera la station suivante. Puis on calcule que "Place d'Italie" est la plus proche de "Daumesnil" *qui n'a*

pas encore été considérée : on l'ajoute à notre liste. En procédant ainsi, on obtient la liste ["Nation", "Daumesnil", "Place d'Italie", "Glacière", "Pasteur", "Dupleix", "Étoile"].

- a) Écrire une fonction `terminus(k)` qui renvoie le nom de la station de métro de la ligne `k` la plus éloignée du zéro de Paris.
- b) Écrire une fonction `suiivante(k, nom_s, l)`, où `k` est le numéro de la ligne, `nom_s` est le nom d'une station, `l` est une liste de noms de stations, et qui renvoie le nom de la station située sur la ligne `k` la plus proche de `nom_s` et qui ne soit pas déjà stocké dans `l`.

Pour simplifier le programme, on estimera qu'il n'y a pas deux statios éloignées de plus de 100 km.

- c) Écrire une fonction `ligne_ordonnee(k)` qui renvoie la liste des noms des stations de la ligne `k` dans l'ordre à l'aide des fonctions précédentes.
- d) Proposer un dessin montrant que l'algorithme précédent pourrait ne pas trouver le bon ordre des stations sur une ligne. *On ne demande pas que le dessin décrive une situation réelle du métro de Paris.*