Représentations, structures de données. Rappels et compléments de PCSI.

1 Représentation d'objets en informatique

Les objets (par exemples physiques ou mathématiques) doivent être représentés, modélisés en machine en « trahissant », « approchant », « oubliant » une part de leur contenu pour pouvoir « tenir dans la machine ».

En fonction de la nature de l'objet et du type de traitement informatique qu'on souhaite opérer sur lui, on ne considère pas la même type d'objet informatique.

En informatique on fait de même : les unités des objets sont appelés des types.

Vous devez connaître 8 types fondamentaux :

1. trois types simples : entiers (type int), flottants (type float), booléens (type bool).
2. quatre types plus complexes : tuple (type 'tuple'), listes (type 'list'), chaine de caractères (type 'str'), dictionnaires (type 'dict'). On les appelle des **structures de données**.
3. un type à part : 'function'

   à cela il faut rajouter :
4. $\cdots$ tous les types créables à partir de ceux ci, par exemple : list de (list de str) et d'int... Pensez par exemple aux graphes, qui sont représentés par des structures de données plus simples.
5. On pourra de temps à autre faire appel à d'autres structures de données, mais celles–ci sont les seules exigibles selon le programme. Par exemple :
   (a) On modélise plus simplement certaines tâches avec des listes à double queue ;
   (b) pour le concours Centrale, à l'oral, d'autres types issus de la bibliothèque numpy sont parfois rencontrés : complex, polynom.

2 Rappels sur les types basiques

Types à connaître

1. $\mathbb{N}$ et $\mathbb{Z}$ - Les entiers sont représentés par le type **integer**. Cet ensemble est **borné**. Taper `import sys` puis `sys.maxsize` pour obtenir la valeur du plus grand **integer**. En pratique Python permet d'aller au–delà de cette limite (en convertissant directement les **integer** en **long integer**)
2. $\mathbb{R}$ - Les réels sont représentés par des flottants. Le type **float** ne recouvre pas tous les réels. Ne sont pas représentés : $10^{100^{100}}$ `(10.)**(10.**100.)` qui est simplement « trop grand » (Overflow) ; les nombres trop petits de même : 10^{-1000} est représenté par 0. En cas d'opérations entre entiers et flottants, le résultat est un flottant.
3. **Booléens** - Il existe deux booléens : True et False, leur type est **bool**. Ils sont très utiles dans les boucles while et les conditions en if/elif/else. Attention, si `b` est un booléen, le code :

```
while b==True:
    *instruction*
```

   est **maladroit !** Préférez :

```
while b:
    *instruction*
```

   Même remarque avec `if` ou `elif`.
4. $\mathbb{C}$ - Un nombre complexe $a+ib$ peut être représenté par une liste à deux flottants `[a,b]` contenant sa partie réelle a et sa partie imaginaire b (elles-mêmes représentées par des flottants).

Exercice 1 (a) Définir des fonctions `module`, `conjugue`, `somme`, `produit`, `inverse` qui prennent en arguments un ou deux complexes selon le cas et renvoient respectivement le module, le conjugué, la somme, le produit, l'inverse de ceux-ci.
Si $z = 1+i$, calculer et représenter z^n pour $n \in [\![0,4]\!]$

A noter que Python propose la commande `complex` : ainsi le code `z=complex(1,1)` associe à z la valeur $1+i$. Les opérations usuelles entre complexes sont implémentées : `z.conjugate()`, `z.real`, `z.imag`, `module(z)` et bien sûr les opérations algébriques. Reprendre alors les questions de l'exercice précédent.

Exercice 2 Comment le programme suivant se comporte–t-il ? Comment se comporterait-il si on pouvait le faire tourner sur l'ensemble $\mathbb{R}$ et non sur les flottants ?

```python
def crash1():
    x=1.0
    while x>0:
        y=x
        x=x/10.0
    return y
```

Il faut bien se rendre compte que les représentations présentent des infidélités avec l'objet que l'on veut représenter. En fonction du problème que l'on cherche à résoudre, on privilégie une représentation.

3 Listes, tuples, chaines de caractères et dictionaires

3.1 Opérations de base et leur complexité

Les quatre structures de données complexes au programme sont les listes (comme [1,2,3]), les tuple (comme (1, 2, 3)), les chaines de caractères (comme '123') et les dictionnaires (comme { 1 :1, 2 :'2', 3 : 'trois' }).

Les opérations autorisées ne sont pas les mêmes, et n'ont pas les mêmes coûts lorsqu'elles le sont.

Opération	Liste (list)	Tuple (tuple)	Chaîne (str)	Dictionnaire (dict)
Accéder à un élément	`lst[index]`	`tup[index]`	`str[index]`	`dict[key]`
Test d'appartennance d'un élément	`x in lst`	`x in tup`	`str1 in str2`	`key in dict`
Modifier un élément	`lst[index] = value`	(non applicable)	(non applicable)	`dict[key] = value`
Ajouter un élément	`lst.append(value)` `lst.insert(index, value)`	(non applicable)	(non applicable)	`dict[key] = value`

Supprimer un élément	`del lst[index]`
	`lst.remove(value)`
	`lst.pop()`
		(non applicable)	(non applicable)	`del dict[key]`

Longueur (nombre d'éléments)	`len(lst)`	`len(tup)`	`len(str)`	`len(dict)`
Concaténation	`lst1 + lst2`	`tup1 + tup2`	`str1 + str2`	(non applicable)
Itération	`for elem in lst:`	`for elem in tup:`	`for char in str:`	`for key, value in dict.items():`
Vérifier la présence d'un élément	`value in lst`	`value in tup`	`char in str`	`key in dict`
Trier	`sorted(lst)` `lst.sort()`	`sorted(tup)`	(non applicable)	(non applicable)
Slicing	`lst[start:end]`	`tup[start:end]`	`str[start:end]`	(non applicable)
Hashable ?	(non applicable)	`hash(tup)`	`hash(str)`	(non applicable)

Opération	Liste (list)	Tuple (tuple)	Chaîne (str)	Dictionnaire (dict)
Accéder à un élément	$O(1)$: Récupère la valeur à une position spécifique via un index.	$O(1)$: Récupère la valeur à une position spécifique via un index.	$O(1)$: Récupère le caractère à une position spécifique via un index.	$O(1)$: Récupère la valeur associée à une clé.
Test d'appartennance d'un élément	$O(n)$	$O(n)$	$O(mn)$	$O(1)$ sauf si collision
Modifier un élément	$O(1)$: Change la valeur à un index spécifique.	(non modifiable) : Les éléments ne peuvent pas être modifiés après la création.	(non modifiable) : Les caractères ne peuvent pas être modifiés après la création.	$O(1)$: Réassigne une valeur à une clé existante.
Ajouter un élément	$O(1)$ (amorti pour append) ou $O(n)$ (insertion) : Insère un nouvel élément ou ajoute à la fin.	(non modifiable) : Impossible d'ajouter des éléments après la création.	(non modifiable) : Impossible d'ajouter des caractères après la création.	$O(1)$ (insertion) : Ajoute une nouvelle clé avec une valeur associée.
Supprimer un élément	$O(n)$ (recherche incluse) : Retire un élément après recherche de sa position. $O(1)$: `lst.pop()` supprime dernier élément	(non modifiable) : Impossible de supprimer des éléments après la création.	(non modifiable) : Impossible de supprimer des caractères après la création.	$O(1)$ (suppression par clé) : Retire un élément en supprimant la clé et sa valeur associée.
Longueur (nombre d'éléments)	$O(1)$: Retourne le nombre d'éléments dans la liste.	$O(1)$: Retourne le nombre d'éléments dans le tuple.	$O(1)$: Retourne le nombre de caractères dans la chaîne.	$O(1)$: Retourne le nombre de paires clé-valeur dans le dictionnaire.
Concaténation	$O(n)$: Combine deux listes en une nouvelle liste.	$O(n)$: Combine deux tuples en un nouveau tuple.	$O(n)$: Combine deux chaînes en une nouvelle chaîne.	(non applicable) : Les dictionnaires ne supportent pas la concaténation.
Itération	$O(n)$: Parcourt chaque élément de la liste.	$O(n)$: Parcourt chaque élément du tuple.	$O(n)$: Parcourt chaque caractère de la chaîne.	$O(n)$: Parcourt chaque paire clé-valeur du dictionnaire.
Vérifier la présence d'un élément	$O(n)$: Vérifie si un élément est dans la liste.	$O(n)$: Vérifie si un élément est dans le tuple.	$O(n)$: Vérifie si un caractère est dans la chaîne.	$O(1)$ (recherche par clé) ou $O(n)$ (valeur) : Vérifie la présence d'une clé ou d'une valeur dans le dictionnaire.
Trier	$O(n \log n)$: Trie les éléments de la liste.	$O(n \log n)$: Trie les éléments du tuple converti en liste.	(non applicable) : Impossible de trier directement les chaînes.	(non applicable) : Impossible de trier directement les dictionnaires.
Slicing	$O(k)$: Crée une sous-liste des éléments spécifiés.	$O(k)$: Crée un sous-tuple des éléments spécifiés.	$O(k)$: Crée une sous-chaîne des caractères spécifiés.	(non applicable) : Impossible de réaliser des opérations de slicing sur les dictionnaires.
Hashable ?	(non hashable) : Les listes ne peuvent pas être utilisées comme clés de dictionnaire.	(hashable) : Un tuple est hashable si, et seulement si, chacune de ses composante l'est.	(hashable) : Les chaînes peuvent être utilisées comme clés de dictionnaire.	non hashable (Rq : : Les clés des dictionnaires doivent être hashables, mais les valeurs peuvent ne pas l'être.)
Hétérogénéïté ?	Oui	Oui	Non	Oui

3.2 Subtilités

Attention

1. Test d'appartenance : autorisé mais coût caché. Attention au contexte.
2. Notion de coût amorti (insertion queue de liste).
3. Hachage et collision (dictionnaire).
4. Copie de liste.

Exercice 3 A l'issue de chacune des éxécutions, donner le contenu des variables l1 et l2.

1.
```
l1=0
l2=l1
l1=19
```

2.
```
l1=[0,1,2,3]
l2=l1
l1[0]=19
```

3.
```
l1=[0,1,2]
l2=[0,1,2]
l1[0]=19
```

3.3 Usage

Méthode

Quelle structure adopter ? Cela dépend du problème que l'on cherche à résoudre !

1. list : polyvalent, mais plus lent,
2. tuple : notation pratique (parenthèses souvent implicites), (HP : utilisable comme clé de hashage), assez rigide,
3. str : optimisé (fonctions et méthodes déjà définies) pour travailler sur du texte.
4. dict : très souple

Exercice 4 Grâce à Python, en choisissant la modélisation la plus adaptée représenter (calculer) :

1. La liste $1,3,5,6,9,\cdots,999$
2. Un vecteur de $\mathbb{K}^n$, en vue de faire des opérations vectorielles : par exemple $(1,1,1)-3(1,2,-1)$
3. $\begin{pmatrix}1&2&3\\3&2&1\\1&1&1\end{pmatrix}$
4. Une séquence (liste, tableau, tuple...) contenant tous les termes de la suite (u_n) telle que $u_0=1$, $u_{n+1}=2^{u_n}-u_n+n$ pour tout $n\in\mathbb{N}$, et qui sont inférieurs à 10^6.
5. La **fonction** $f\circ f\circ f$, où f est la fonction $x\mapsto\arctan(x^3+\cos(x))$
6. La **fonction** polynôme $x\mapsto x^3-5x^2+7$
7. Une application de $[\![0,n-1]\!]$ vers $[\![0,n-1]\!]$
8. L'algorithme de la division euclidienne
9. Le réseau du métro à Paris

Solution

1. tuple ou liste.
2. tuple.
3. Liste de listes, ou tuple de tuples ; selon que l'on compte ou non modifier la matrice. Il faudra reprogrammer toutes les opérations à la main.
4. Ici si on compte rajouter des termes successivement il faut utiliser une liste
5. Le plus simple est de représenter f par une fonction Python
6. $\mathbb{K}[X]$ – un polynôme à une indéterminée sur le corps $\mathbb{K}$, $\sum_{k=0}^{n}a_kX^k$, est représenté :
   (a) par la liste de ses coefficients : `[a0, a1, ..., an]` ou `[an, ..., a1, a0]`.
   (b) par un array
   (c) en utilisant le type polynom prédéfini

   Evitez de le représenter par une fonction Python.
7. $[\![0,n-1]\!]^{[\![0,n-1]\!]}$ – une application de $[\![0,n-1]\!]$ vers $[\![0,n-1]\!]$ est représenté par la liste des images. Par exemple $[0,1,2]$ représente l'identité et $[0,0,0]$ représente l'application nulle. Evitez de les représenter par une fonction Python.
8. En Python, un **algorithme** est représenté par un **programme** informatique, sous forme de fonction Python dans le cadre du programme. On peut aussi le considérer comme un texte, et représenter ce code sous forme de chaine de caractères.
9. Un graphe non orienté.

4 Exemples d'emploi

4.1 Exemples mathématiques

Exercice 5
1. Proposer une fonction Python qui prend en entrée deux polynomes, représentés par des listes, et retourne en sortie leur somme.
2. Proposer une fonction Python qui prend en entrée deux polynomes, représentés par des listes, et retourne en sortie leur produit.
3. Proposer une fonction Python qui prend en entrée un polynome `P` et un flottant `x`, et retourne $P(x)$.

Exercice 6 On considère la famille F de toutes les fonctions de la forme $x \mapsto a\cos(2x) + b\sin(2x)$.

1. Proposer une représentation Python.
2. Ecrire une fonction Python qui reçoit `[a,b]` et `x` et qui retourne $a\cos(2x) + b\sin(2x)$.
3. Ecrire une fonction qui prend en entrée une fonction f de F, représentée par `f`, et retourne (une représentation de) la fonction f'.
4. Ecrire une fonction qui prend en entrée une fonction f de F, représentée par `f`, un entier `n` et retourne (une représentation de) la fonction $f^{(n)}$.

4.2 Autres exemples, plus complexes

Les objets mathématiques ne sont pas les seuls à devoir être représentés en informatique. Pour chaque exemple, chercher plusieurs représentations, et voir si certaines sont plus économiques en taille que d'autres.

Exemples

1. Jeu d'échecs. Comment représenter une configuration ? et en tenant compte du Roque (effectué ou non) ?
2. Jeu du Mastermind. Comment représenter la configuration secrète, une proposition, la réponse, l'historique de toutes les configurations ? Comment créer une configuration secrète aléatoirement ?
3. Snake. Comment représenter la position du serpent ?
4. Trajet en métro (site ratp par exemple). Comment représenter un trajet en métro, comportant différents changements ?
5. Réseaux sociaux. Qui est ami avec qui. Représentation ? Représentation permettant de savoir qui est ami d'ami, d'ami... rapidement ?
6. Pandémie. Comment représenter les interactions pour détecter/isoler les clusters rapidement ?
7. Problème du sac de voyage/du sac à dos. Vous disposez de N objets, et chaque objet $i \in [\![1, N]\!]$ a une valeur v_i et un poids p_i. Hélas le poids maximum que vous pouvez embarquer est de $P < \sum_{i=1}^{n} p_i$. Quels objets choisir dans votre sac à dos pour respecter cette contrainte, tout en maximisant la somme des v_i ?
8. Correction orthographique. Pour corriger 'ABELLIE' en 'ABEILLE', on a le droit de • supprimer • modifier • insérer une lettre, et on cherche à passer du mauvais au bon mot en un nombre minimum d'étapes. Comment représenter un mot ? Une succession de modifications ? Et comment utiliser des représentations qui permettent de trouver la solution la plus courte ?

5 Graphes

5.1 Qu'est-ce qu'un graphe ?

Un **graphe** $G = (S, A)$ est composé de deux éléments principaux :

- **Sommets** (ou nœuds) éléments de S : Ce sont les points qui représentent les objets dans le graphe. Chaque sommet est identifié par un nom ou un numéro.
- **Arêtes** (ou liens) : Ce sont les connexions entre les sommets. Elles peuvent être dirigées (dans une certaine direction, ce sont alors des éléments de $S \times S$) ou non dirigées (sans direction, parties à deux éléments de S).
- **Voisin** : Un sommet est dit voisin d'un autre si une arête les connecte directement.
- **Composante connexe** : Une composante connexe d'un graphe est un sous-ensemble de sommets tels qu'il existe un chemin entre chaque paire de sommets de ce sous-ensemble, et qui n'est pas relié à d'autres sommets du graphe.

5.2 Représentations possibles en Python

Il existe plusieurs manières de représenter un graphe en Python. Voici les trois représentations les plus courantes :

5.2.1 Liste de listes de voisins

Chaque index de la liste représente un sommet, et chaque élément est une liste de voisins.

```
graphe = [
    [1, 2],     # Sommets voisins de 0
    [0, 2],     # Sommets voisins de 1
    [0, 1]      # Sommets voisins de 2
]
```

5.2.2 Dictionnaire

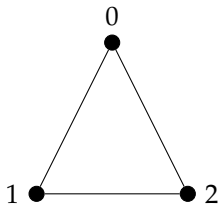
Un dictionnaire a pour clés les sommets et pour valeurs des listes de voisins.

```
graphe = {
    0: [1, 2],
    1: [0, 2],
    2: [0, 1]
}
```

5.2.3 Matrice d'adjacence

C'est une matrice carrée où la case (i, j) est remplie par 1 s'il existe une arête entre les sommets i et j, et 0 sinon.

```
graphe = [
    [0, 1, 1],  # Aretes du sommet 0
    [1, 0, 1],  # Aretes du sommet 1
    [1, 1, 0]   # Aretes du sommet 2
]
```



5.3 Parcours d'un graphe

Parcourir un graphe, c'est énumérer ses sommets en respectant un ordre de parcours lié à sa structure. C'est une opération fondamentale en théorie des graphes et en informatique, avec de nombreuses applications pratiques :

- **Recherche de chemin** : Trouver le chemin le plus court ou le plus long entre deux sommets. Utilisé dans les systèmes de navigation GPS pour trouver le chemin le plus court entre deux points.
- **Détection de cycles** : Détecter la présence de cycles dans un graphe. Utilisé dans les compilateurs pour détecter des dépendances circulaires dans le code source.
- **Connectivité** : Identifier les composantes connexes d'un graphe non orienté. Utilisé dans les réseaux de télécommunications pour identifier les sous-réseaux isolés.
- **Recherche de motifs** : Trouver des motifs fréquents dans des graphes. Utilisé en bio-informatique pour identifier des séquences d'ADN récurrentes.
- **Optimisation** : Résoudre des problèmes de couverture ou de coloration. On verra la notion de graphe biparti en théorie des jeux.
- **Applications pratiques** : Analyser les réseaux sociaux, planifier des itinéraires de transport, optimiser les routes de communication, etc.

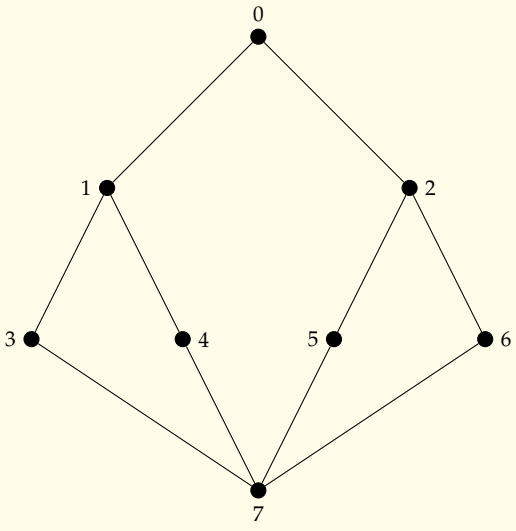
5.4 Parcours en largeur et en profondeur

Deux algorithmes fondamentaux de recherche dans les graphes sont :

Parcours en largeur (Breadth-First Search, BFS) Ce parcours explore les sommets voisins avant de descendre plus profondément dans le graphe. Il utilise une file d'attente pour garder une trace des sommets à explorer.

Parcours en profondeur (Depth-First Search, DFS) Dans ce parcours, on s'enfonce dans un sommet aussi loin que possible avant de reculer. Il peut être réalisé avec une pile ou de manière récursive.

Exemple



Prenons l'exemple du graphe suivant :
Parcours en largeur (BFS)

- Débuter au sommet 0.
- Explorer les sommets adjacents : 1 et 2.

- Explorer les sommets adjacents de 1 : 3 et 4.
- Explorer les sommets adjacents de 2 : 5 et 6.
- Explorer les sommets adjacents de 3, 4, 5, et 6 : 7.

L'ordre de visite des sommets serait : 0, 1, 2, 3, 4, 5, 6, 7.

Parcours en profondeur (DFS)

- Débuter au sommet 0.
- Explorer le sommet adjacent 1.
- Explorer le sommet adjacent de 1 : 3.
- Explorer le sommet adjacent de 3 : 7.
- Revenir en arrière à 1 et explorer le sommet adjacent 4.
- Explorer le sommet adjacent de 4 : 7 (déjà visité).
- Revenir en arrière à 0 et explorer le sommet adjacent 2.
- Explorer le sommet adjacent de 2 : 5.
- Explorer le sommet adjacent de 5 : 7 (déjà visité).
- Revenir en arrière à 2 et explorer le sommet adjacent 6.
- Explorer le sommet adjacent de 6 : 7 (déjà visité).

L'ordre de visite des sommets serait : 0, 1, 3, 7, 4, 2, 5, 6.

6 Bibliothèques

En pratique, la majorité des méthodes que nous étudierons en cours sont classiques et sont préprogrammées sous Python. Elles sont « rangées » dans des bibliothèques. Deux grandes catégories :

1. Les bibliothèques citées dans le programme officiel, en particulier la bibliothèque `collections`.

Remarque — Rappels

On utilise les instructions `import` *nom_de_la_bibliothèque*
et
`import` *nom_de_la_bibliothèque* `as` *nom_par_lequel_je_veux_l_appeler*
`from` *nom_de_la_bibliothèque* `import` *fonctions_que_je_souhaite_appeler_facilement*

```
import math
from math import *  #toutes les commandes de maths sont importees: pi au lieu de math.pi, etc
#centrale, TIPE:
import numpy as np
import scipy
import scipy.optimize
import scipy.integrate
from scipy.integrate import odeint
#centrale, TIPE, TP avec graphismes:
import matplotlib
import matplotlib.pyplot as plt
plot=matplotlib.pyplot.plot
```

Remarque

Citons d'autres bibliothèques pratiques, en particulier pour vos TIPE : `pylab`, qui mime l'environnement matlab ; `tkinter` qui permet d'autres graphismes (création d'interfaces graphiques) ; `sympy` pour le calcul formel (factorisation, primitivation...) ; `time`, qui permet de mesurer le temps et en particulier d'évaluer la rapidité d'exécution des programmes.

7 les Piles, les Files, et Python

En informatique, les **files** et les **piles** sont deux structures de données fondamentales qui permettent de stocker et de manipuler des ensembles d'éléments. Elles se distinguent principalement par la manière dont les éléments sont ajoutés et retirés. Ces concepts sont essentiels pour comprendre différents algorithmes, notamment les parcours de graphes. Dans ce cours, nous utiliserons Python pour implémenter ces structures, en modélisant les **files** avec `collections.deque` et les **piles** avec des **listes**.

La Pile (Stack)

Une **pile** fonctionne selon le principe **LIFO** (Last In, First Out), ce qui signifie que le dernier élément ajouté sera le premier à être retiré. C'est comme une pile de livres où vous ne pouvez retirer que le livre qui est sur le dessus.

En Python, une pile peut être facilement implémentée avec une liste :

```

pile = []


pile.append(1)
```

```
pile.append(2)
pile.append(3)


element = pile.pop()  # element = 3
```

La File (Queue)

Une **file** fonctionne selon le principe **FIFO** (First In, First Out), ce qui signifie que le premier élément ajouté sera le premier à être retiré. C'est similaire à une file d'attente au cinéma, où la première personne arrivée est la première à entrer.

En Python, une file est efficacement implémentée avec `collections.deque`, qui permet d'ajouter et de retirer des éléments **des deux extrémités** en temps constant :

```
from collections import deque


file = deque()


file.append(1)
file.append(2)
file.append(3)


element = file.popleft()  # element = 1
```

8 Parcours en Profondeur et en Largeur : implémentations

```
def parcours_profondeur(graphe, depart):
    # Initialiser une pile avec le sommet de depart
    pile = [depart]
    # Ensemble pour garder trace des sommets visites
    visites = {}

    while pile:
        # Prendre le sommet au sommet de la pile
        sommet = pile.pop()

        # Si ce sommet n'a pas ete visite
        if sommet not in visites:
            # Le marquer comme ayant ete visite
            print(sommet)  # Action a realiser lors de la visite (ici, l'impression du sommet)
            visites[sommet]=True

            # Ajouter tous les voisins non visites du sommet a la pile
            for voisin in graphe[sommet]:
                if voisin not in visites:
                    pile.append(voisin)

# Exemple d'utilisation
graphe = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

parcours_profondeur(graphe, 'A')




from collections import deque
def parcours_largeur(graphe, depart):
    # Initialiser une file avec le sommet de depart
    file = deque([depart])
    # Dictionnaire pour garder trace des sommets visites
    visites = {}

    while file:
        # Prendre le sommet au debut de la file
        sommet = file.popleft()

```

```
48          # Si ce sommet n'a pas ete visite
49          if sommet not in visites:
50              # Le marquer comme visite
51              print(sommet)  # Action a realiser lors de la visite (ici, l'impression du sommet)
52              visites[sommet] = True
53
54              # Ajouter tous les voisins non visites du sommet a la file
55              for voisin in graphe[sommet]:
56                  if voisin not in visites:
57                      file.append(voisin)
58
59  # Exemple d'utilisation
60  graphe = {
61      'A': ['B', 'C'],
62      'B': ['D', 'E'],
63      'C': ['F'],
64      'D': [],
65      'E': ['F'],
66      'F': []
67  }
68
69  parcours_largeur(graphe, 'A')
```

9 Résumé

1. On choisit une représentation en fonction du problème posé. **Toutes les représentations ont leur points forts et points faibles**.
2. La complexité de certaines opérations dépend aussi des représentations qui ont été choisies.

10 Solutions

Solution

Solution 2 Dans $\mathbb{R}$, la suite de terme général $\frac{1}{10^n}$ tend vers 0 mais **reste strictement positive**. Ici il existe un integer n tel que `x/10**n` a la valeur `0.0`. Le programme calcule la dernière valeur non nulle de la suite, à savoir `y`. On trouve la valeur `1e-323`, c'est-à-dire 10^{-323}.