ISM MP, Informatique
Année 2024/2025

TP Informatique 02

On importera les modules suivants :

```
import numpy as np, numpy.random as rd
```

Copier le fichier `ClassePile.py` dans le répertoire de travail puis, pour chaque programme de l'exercice 1, commencer par réaliser l'importation `from ClassePile import *` pour manipuler la classe `ClassePile`. Celle-ci est munie des opérations primitives suivantes :

- `Pile()` qui crée une pile vide ;
- `P.vide()` qui renvoie `True` si la pile `P` est vide et `False` sinon ;
- `P.empiler(x)` qui empile l'élément `x` dans la pile `P` ;
- `P.depiler()` qui dépile le sommet de la pile non vide `P`.

On dispose également de l'opération `P.affiche()` pour afficher le contenu de la pile `P`.

Exercice 1

La notation *postfixée* permet d'écrire des expressions algébriques sans parenthèse en plaçant les opérateurs après les opérandes. Par exemple, en notation postfixée

$$(1+2)\times 4 \quad \text{devient} \quad 1 \quad 2 \quad + \quad 4 \quad \times$$

La notation *infixée* est une notation complètement parenthésée, incluant les parenthèses habituellement superflues du fait des règles de priorité entre les opérations. En notation infixée

$$1+2\times 5 \quad \text{devient} \quad (1+(2\times 5))$$

Dans ce qui suit, les seules opérations algébriques considérées sont $+$, $-$, $\times$, $/$.

1. Écrire une fonction `eval(L)` d'argument `L` une liste décrivant une expression algébrique sur des nombres en notation postfixée et qui renvoie la valeur de cette expression. Par exemple, l'appel de `eval([1,2,'+',4,'*'])` renvoie 12. On utilisera une pile pour empiler les opérandes.
2. Écrire une fonction `inftopost(E)` d'argument `E` une chaîne décrivant une expression algébrique sur des nombres en notation infixée et qui renvoie une liste de l'expression en notation postfixée. Par exemple, l'appel de `inftopost("(1+(2*5))")` renvoie `[1,2,5,'*','+']`. On utilisera une pile pour les opérateurs. On pourra commencer par des opérandes à valeurs dans $[\![0\,;\,9]\!]$.

Exercice 2

Dans ce problème, on s'intéresse à des implémentations récursives du produit matriciel basées sur le paradigme « diviser pour régner ». Pour simplifier, on ne traitera que le cas de matrices dans $\mathscr{M}_{2^n}(\mathbb{R})$ avec n entier. Soient M_1 et M_2 dans $\mathscr{M}_{2^n}(\mathbb{R})$ avec n entier non nul. On considère leurs écritures par blocs dans $\mathscr{M}_{2^{n-1}}(\mathbb{R})$ données par

$$\mathrm{M}_1 = \left(\begin{array}{c|c} \mathrm{A}_1 & \mathrm{B}_1 \\ \hline \mathrm{C}_1 & \mathrm{D}_1 \end{array}\right) \quad \text{et} \quad \mathrm{M}_2 = \left(\begin{array}{c|c} \mathrm{A}_2 & \mathrm{B}_2 \\ \hline \mathrm{C}_2 & \mathrm{D}_2 \end{array}\right)$$

Alors
$$\mathrm{M}_1 \times \mathrm{M}_2 = \left(\begin{array}{c|c} \mathrm{A}_1\times\mathrm{A}_2+\mathrm{B}_1\times\mathrm{C}_2 & \mathrm{A}_1\times\mathrm{B}_2+\mathrm{B}_1\times\mathrm{D}_2 \\ \hline \mathrm{C}_1\times\mathrm{A}_2+\mathrm{D}_1\times\mathrm{C}_2 & \mathrm{C}_1\times\mathrm{B}_2+\mathrm{D}_1\times\mathrm{D}_2 \end{array}\right)$$

Saisir les fonctions suivantes :

```
def scinde(M):
    n=len(M)//2
    return M[:n,:n],M[:n,n:],M[n:,:n],M[n:,n:]

def fusion(A,B,C,D):
    res1=np.concatenate((A,B),axis=1)
    res2=np.concatenate((C,D),axis=1)
    return np.concatenate((res1,res2),axis=0)
```

La fonction `scinde(M)` prend en argument une matrice $\mathrm{M} \in \mathscr{M}_{2^n}(\mathbb{R})$ avec n entier non nul et renvoie les quatre sous-matrices $\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}$ de $\mathscr{M}_{2^{n-1}}(\mathbb{R})$ telles que $\mathrm{M} = \left(\begin{array}{c|c} \mathrm{A} & \mathrm{B} \\ \hline \mathrm{C} & \mathrm{D} \end{array}\right)$. La fonction `fusion(A,B,C,D)` réalise l'opération inverse et reconstruit la matrice par blocs M à partir de ses quatre blocs.

1. Écrire une fonction `produit(M1,M2)` qui réalise récursivement le produit matriciel $\mathrm{M}_1 \times \mathrm{M}_2$. Les matrices M_1 et M_2 seront fournies au format `np.array` et on pourra utiliser `len(M1)` qui renvoie le nombre de lignes de M_1.
2. Tester `produit(M1,M2)` sur des matrices générées aléatoirement. L'instruction `rd.random((n,n))` fournit une matrice de taille $n \times n$ avec des coefficients aléatoires dans $]\,0\,;1\,[$. On pourra comparer avec le résultat fourni par `np.dot(M1,M2)` qui calcule le produit matriciel $\mathrm{M}_1 \times \mathrm{M}_2$.
3. L'algorithme de Strassen consiste à effectuer

$$\mathrm{M}_1 \times \mathrm{M}_2 = \left(\begin{array}{c|c} \mathrm{X}_1 + \mathrm{X}_4 - \mathrm{X}_5 + \mathrm{X}_7 & \mathrm{X}_3 + \mathrm{X}_5 \\ \hline \mathrm{X}_2 + \mathrm{X}_4 & \mathrm{X}_1 - \mathrm{X}_2 + \mathrm{X}_3 + \mathrm{X}_6 \end{array}\right)$$

   avec

$$\mathrm{X}_1 = (\mathrm{A}_1 + \mathrm{D}_1)(\mathrm{A}_2 + \mathrm{D}_2) \quad \mathrm{X}_2 = (\mathrm{C}_1 + \mathrm{D}_1)\mathrm{A}_2 \quad \mathrm{X}_3 = \mathrm{A}_1(\mathrm{B}_2 - \mathrm{D}_2) \quad \mathrm{X}_4 = \mathrm{D}_1(\mathrm{C}_2 - \mathrm{A}_2)$$

$$\mathrm{X}_5 = (\mathrm{A}_1 + \mathrm{B}_1)\mathrm{D}_2 \quad \mathrm{X}_6 = (\mathrm{C}_1 - \mathrm{A}_1)(\mathrm{A}_2 + \mathrm{B}_2) \quad \mathrm{X}_7 = (\mathrm{B}_1 - \mathrm{D}_1)(\mathrm{C}_2 + \mathrm{D}_2)$$

   Écrire une fonction `strassen(M1,M2)` qui réalise récursivement le produit $\mathrm{M}_1 \times \mathrm{M}_2$ selon l'algorithme de Strassen.
4. Tester `strassen(M1,M2)` sur des matrices générées aléatoirement. On pourra comparer avec le résultat fourni par `np.dot(M1,M2)`
5. Saisir les lignes de code suivantes :

```
import time
print("\nComparaison produit récursif naïf VS Strassen")
N=2**8
M1=rd.random((N,N));M2=rd.random((N,N))
deb=time.time();produit(M1,M2);fin=time.time()
print("produit récursif naïf = ",fin-deb)
deb=time.time();strassen(M1,M2);fin=time.time()
print("Strassen = ",fin-deb)
```

   Ces lignes comparent le temps d'exécution de `produit(M1,M2)` et `strassen(M1,M2)` sur des matrices M_1 et M_2 de taille $2^8 \times 2^8$. Faire l'expérimentation. Que constate-t-on ?
6. Proposer une explication au comportement précédemment observé.

Exercice 3

On s'intéresse au phénomène de *percolation* : on imagine un matériau poreux sur lequel on verse de l'eau et on cherche à savoir si l'eau traverse le milieu.

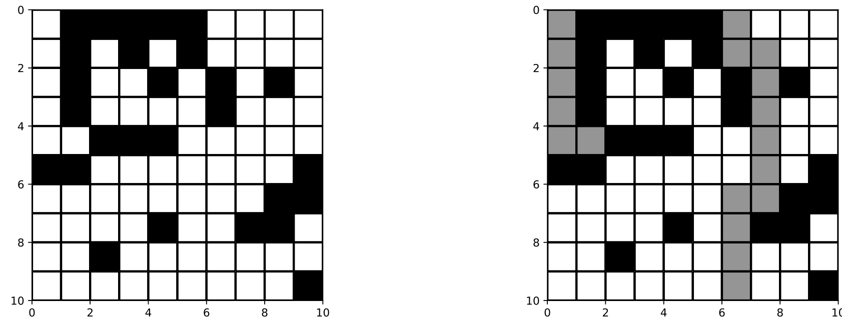

Figure 1 – Phénomène de percolation

En s'inspirant de la démarche de résolution d'un labyrinthe (fichier `labyrinthe.py`), on se propose d'implémenter une simulation de percolation dans une grille de taille $n \times n$ modélisée par une liste de listes. On exécutera le fichier `perco.py` qui génère un milieu poreux avec des cases qui sont murées avec probabilité $1 - p$ où $p \in]\,0\,;1\,[$.

1. Écrire une fonction `percole(milieu,deb)` d'arguments `milieu` une liste de listes et `deb` une position. La fonction `percole` renvoie `True` s'il y a percolation depuis `deb` et `False` sinon et modifie les cases visitées de `milieu` en les passant à l'état visité.
2. Écrire une fonction `percolation(milieu)` qui balaye l'ensemble des positions de la première ligne de `milieu` jusqu'à arriver en butée ou détecter une percolation. La fonction renvoie `True` s'il y a percolation dans `milieu` et `False` sinon.
3. Pour n variant dans `[10,20,30,40]`, tracer pour p variant de 0 à 1 la proportion de percolations sur $\mathrm{N} = 100$ milieux générés aléatoirement avec probabilité de murage de $1 - p$. Quel phénomène observe-t-on ?