TP 26- Proposition de solutions

Solution 1 Commençons par un algorithme qui part d'un sommet s donné. Il utilise la notion de *file*. Les outils sont :

- une liste `parents` qui contient soit `False` pour un sommet non atteint, soit un sommet entrant du sommet considéré ;
- une file `file` des sommets (voisins) intermédiaires restant à parcourir.

```
1  def detection_cycle_GNO_S(s,M): ## graphe NO
2      n=len(M)
3      file=[s]
4      parent=n*[False]
5      parent[s]=s # valeur arbitraire
6      while len(file)>0:
7          p=file.pop(0)
8          for i in range(n):
9              if M[p][i]>0:
10                 if parent[i]==False:
11                     file.append(i)
12                     parent[i]=p
13                 elif parent[p]!=i:
14                     return True
15     return False
```

Le critère de la ligne 13 consiste a éviter que la cycle soit constitué d'une seule arrête (aller-retour) ce qui ne constitue pas un cycle dans les graphes non orientés.

On note que le cycle trouvé, lorsqu'il y en a un ne contient pas forcément le sommet s, mais est un cycle de la composante connexe de s. Ainsi pour généraliser le travail, il suffit de travailler une seule fois sur chaque composante connexe. Après un premier parcours, on repère parmi les sommets non encore atteint le début d'une nouvelle composante connexe.

```
def detection_cycle_GNO(M): ## graphe non oriente
    n=len(M)
    parent=n*[False]
    for s in range(n):
        if parent[s]==False:
# nouvelle comp. connexe
            file=[s]
            parent[s]=s # valeur arbitraire
            while len(file)>0:
                p=file.pop(0)
                for i in range(n):
                    if M[p][i]>0:
                        if parent[i]==False:
                            file.append(i)
                            parent[i]=p
                        elif parent[p]!=i:
                            return True
    return False
```

Solution 2 1. Le résultat de 1 2 3 4 5 / + × − est

$$1-2\times\left(3+\frac{4}{5}\right)$$

2. Une écriture en notation polonaise inversée de

$$\frac{3}{4*5+1}+2$$

est 3 4 5 × 1 + / 2 +

3. Évaluation d'une expression écrite en notation polonaise inversée :

```
def eval_npi(L):
    op=['+','-','*','**','/']
    pile=[]
    for i in range(len(L)):
        if L[i] in op:
            a=str(pile.pop())
            b=str(pile.pop())
            pile.append(eval(b+L[i]+a))
        else:
            pile.append(L[i])
    return(pile.pop())
```

Exemple :

```
>>> E1=[1,2,'+',3,4,'/','-']
>>> print(eval_npi(E1))
2.25
```

4. Généralisation :

npi.py

```
def eval_npi(L):
    op=['+','-','*','**','/']
    pile=[]
    for i in range(len(L)):
        if type(L[i]) in [float,int]:
            pile.append(L[i])
        elif L[i] in op:
            a=str(pile.pop())
            b=str(pile.pop())
            pile.append(eval(b+L[i]+a))
        else:
            a=pile.pop()
            pile.append(L[i](a))
    return(pile.pop())
```

Exemple :

```
>>> E2=[1,2,sqrt,'+']
>>> print(eval_npi(E2))
2.41421356237
```

Solution 3 Parenthèses

1. Pour tester si une expression est bien parenthésée il suffit de :
- Compter algébriquement, +1 pour une parenthèse ouvrante et -1 pour une fermante.
- Vérifier que le compteur est toujours positif et que sa valeur finale est nulle.

Avec un compteur

```
def p(c):
    compte=0
    for i in c:
        if i=='(':
            compte+=1
        elif i==')':
            if compte>0:
                compte-=1
            else:
                return(False)
    if compte==0:
        return(True)
    else:
        return(False)
```

2. Cette fois, en plus de dénombrer les parenthèses ouvrantes, il faut retenir leurs position. Une pile est l'outil ad-hoc ! On empile les positions des parenthèses ouvrantes et on dépile à chaque parenthèse fermante rencontré.

Avec une pile

```
def p2(c):
    couple=[]
    pile=[]
    for i in range(len(c)):
        if c[i]=='(':
            pile.append(i)
        elif c[i]==')':
            if len(pile)>0:
                couple.append((pile.pop(),i))
            else:
                return(False,i)
    if len(pile)==0:
        return(True,couple)
    else:
        return(False,pile.pop())
```

3. Afin de généraliser à plusieurs types de parenthèses, on peut gérer une pile de couples : la position et la type de parenthèse.

Avec une pile de couples

```
def p3(c):
    couple=[]
    pile=[]
    for i in range(len(c)):
        if c[i] in '({[':
         pile.append((i,c[i]))
        elif c[i]==')':
         if len(pile)==0 or pile[-1][1]!='(':
          return(False,'(',i)
         else:
          couple.append(('(',pile.pop()[0],i))
        elif c[i]=='}':
         if len(pile)==0 or pile[-1][1]!='{':
          return(False,'{',i)
         else:
          couple.append(('{',pile.pop()[0],i))
        elif c[i]==']':
         if len(pile)==0 or pile[-1][1]!='[':
          return(False,'[',i)
         else:
          couple.append(('[',pile.pop()[0],i))
    if len(pile)==0:
        return(True,couple)
    else:
        return(False,pile.pop())
```