Octobre 2020

Programmation orientée objet / file et pile

MOINET Dominique

Présentation du travail

Préambule

Le travail demandé pour le bloc 5 portant sur les arbres j'ai fait le choix de travailler sur les listes, les files et les piles. Les parties 1 et 2 étant pratiquement finalisées, m'est venu (trop tardivement) à l'idée un jeu de pion sur des pieux (qui change des tours de Hanoï). Ayant voulu explorer le thème je le propose en partie 3 comme un projet supplémentaire (pas tout à fait finalisé), l'idée me semblant intéressante.
Le tout est donc plus qu'un micro-micro projet, désolé.

Partie 1 programmation objet – récursivité

Le projet (ou suite d'exercices) aborde le thème des polynômes.
A travers la construction de classes on s'interroge sur des implémentations différentes et la possibilité de passer d'une implémentation à une autre. On en profite pour faire un peu de récursivité en appliquant la méthode de Hörner pour l'évaluation d'un polynôme en une valeur de a donnée.
Afin de ne pas augmenter les difficultés (intéressantes mais non essentielles à l'objectif pédagogique de ce projet) on travaillera avec des polynômes à coefficients entiers positifs (évite le problème du « zéro flottant » et l'affichage des négatifs).
Inconvénients (ou avantage) : Ce projet est assez orienté « math »

On peut envisager qu'en classe, aura été vue une classe `Trinome`, permettant de traiter les trinômes du seconde degré (implémentation, affichage, discriminant, racines …)

Les utilisations de `*arg` et `*[L[i] for i in range(len(L))]` auront été aussi abordées, si ce n'est pas le cas l'appel des classes `polynome1` et `polynome2` peut se faire à l'aide de listes.

Partie 2 manipulation et programmation de files (pas de programmation objet)

Cette partie traite de la structure « file » à manipuler à partir d'un algorithme, puis d'une programmation en Python. Les méthodes `append` et `pop` auront été vues en classe.

Partie 3 programmation objet et piles

A partir d'un jeu, travailler la programmation objet et les piles.

Enoncé possible (à travailler) du projet

Partie 1

Question 1 Evaluation d'un polynôme en une valeur de a donnée.

Dans cette question, un polynôme est modélisé par une liste d'entiers correspondant aux coefficients considérés suivant l'ordre croissant des degrés.
Le polynôme P, défini par : $P(x) = 5 + 3x^2 + x^3$ est représenté par la liste L = [5, 0, 3, 1], ainsi L[k] correspond au coefficient du monôme de degré k.
Présentation de la méthode de Hörner : $P(x) = 4x^3 + 5x^2 + 3x + 2 = x\times(x\times(x\times(4) + 5) + 3) + 2$

Travail à effectuer.

1) On considère la fonction `Eval1()` donnée ci-contre prenant pour arguments, une liste L représentant un polynôme P.
Donner les différentes valeurs de la variable locale `res` à la fin de chaque boucle pour `L=[3,0,2,1]` et `a = 2`.
Que renvoie cette fonction ? (on fera une phrase correcte et compréhensible incluant les paramètres de la fonction).

```
def Eval1(L,a):
  res=0
  for k in range(len(L)):
      res=res + L[k]*a**k
  return res
```

On souhaite obtenir le même résultat, de la même façon mais avec une seule ligne de code.
Quelle ligne de code peut-on écrire après le `return` pour que `Eval2()` satisfasse à ces conditions ? (on utilisera la fonction `sum` et une liste par compréhension)

```
def Eval2(L,a):
  return ----------
```

2) Ecrire une fonction `Horner()` prenant pour arguments, une liste L représentant un polynôme P et a un nombre et renvoie la valeur P(a) calculée avec la méthode de Hörner. On donnera une version itérative et une version récursive.

Remarques

Demander d'écrire correctement ce que fait une fonction est un bon exercice pour la rigueur et les spécifications (non présentes ici) d'une fonction.
Thèmes abordés : Récursivité, manipulation de listes, (un peu de programmation fonctionnelle ?).

Question 2 Programmation objet

On souhaite créer une classe `polynome1` permettant de modéliser des polynômes à coefficients entiers positifs dont les attributs sont :
`coeff` qui est la liste des coefficients comme précité et
`deg`, le degré du polynôme

Travail à effectuer.

1) Ecrire la classe `polynome1` représentant un polynôme P.
On écrira le constructeur donnant les attributs `coeff` et `deg` ainsi qu'une méthode `__repr__` permettant d'afficher le polynôme.
`polynome1(5,0,3,1)` modélise le polynôme P : $P(x) = 5 + 3x^2 + x^3$
Pour `P = Polynome1(5,0,3,1)` on a `P.coeff = [5,0,3,1]` et `P.deg = 3`.
`print(P)` affichera : `5+3X^2+X^3` (bien gérer la constante, le degré 1 et les éventuels coefficients nuls ou égaux à 1)

2) Compléter la classe `polynome1` avec les méthodes suivantes :
- `evalue` qui renvoie la valeur du polynôme en une valeur a donnée.
  On utilisera une des fonctions de la question 1. Avec notre exemple `P.evalue(2)` renvoie P(2) = 25.
- `plus` qui renvoie la somme de deux polynômes.
- `derive` renvoyant le polynôme dérivé (on pourra utiliser une construction par compréhension)
- `fois` qui renvoie le produit de deux polynômes (facultatif : pour les plus passionnés).

3) Commenter la classe, afin d'avoir une documentation à l'aide de `polynome1.__doc__`

4) Faire les jeux d'essai suivants :

Données :

```
P = polynome1(5,0,3,1)
Q = polynome1(0,3,1,6,2,3)
```

Les exécutions :
Afficher `P` et `Q`
Créer et afficher les polynômes dérivés de `P` et `Q`
Evaluer `P` en $a = 0$, $a = 2$ et $a = -1$
Créer et afficher `S = P + Q`
Déterminer le nombre dérivé S'(1).

Question 3 Programmation objet

L'utilisation de la classe `polynome1`, n'est pas forcément la plus adaptée pour des polynômes du type $P(x) = 3x^5 + x^{25}$ ou tout simplement $P(x) = x^{50}$.
On souhaite alors créer une nouvelle classe `polynome2` où les polynômes sont représentés par une liste de tuples du type (degré, coefficient) représentant chaque monôme.
Ainsi $P(x) = 3x^5 + x^{25}$ est représenté par la liste $[(5,3), (25,1)]$ (les tuples seront donnés suivant l'ordre croissant des degrés).
La classe `polynome2` permettant de modéliser des polynômes comme somme de monômes aura pour attributs :

- `monomes` qui est la liste des monômes comme précité et
- `deg`, le degré du polynôme

Le polynôme nul est représenté par `[(0,0)]`.

Travail à effectuer.

1) Ecrire la classe `polynome2` représentant un polynôme P comme somme de monômes.
On écrira le constructeur donnant les attributs `monomes` et `deg` ainsi qu'une méthode `__repr__` permettant d'afficher le polynôme.
`polynome2((5,3),(25,1))` modélise le polynôme P : $P(x) = 3x^5 + x^{25}$
Pour `P = polynome2((5,3),(25,1))` on a `P.monomes = [(5,3),(25,1)]` et `P.deg = 25`.

2) Compléter la classe `Polynome2` avec les méthodes suivantes :

- une méthode d'affichage.
- `derive` renvoyant le polynôme dérivé (on essaye en une seule ligne !)

3) Proposer des jeux d'essai.

4) La somme de deux polynômes semble plus compliquée (moins intuitive) qu'avec l'implémentation précédente. Les méthodes `evalue` et `derive` (et autres si besoin) étant déjà disponibles dans la classe `polynome1`, on souhaite pouvoir passer d'une implémentation à une autre pour ne pas avoir à tout écrire.

a) Compléter le constructeur de la classe `polynome2` pour obtenir un attribut `poly1` qui est la liste ordonnée croissante de tous les coefficients.

b) Compléter le constructeur de la classe `polynome1` pour obtenir un attribut `poly2` qui est la liste ordonnée croissante des tuples représentant chaque monôme.

c) Compléter alors la classe `polynome2` avec une méthode permettant d'ajouter deux polynômes en utilisant la méthode `plus` de la classe `polynome1` ainsi que les attributs `poly1` et `poly2`.

5) Ecrire une fonction `Saisie()` prenant pour argument un entier `cas` prenant pour valeur 1 ou 2 et permettant de saisir les données pour un polynôme, `cas = 1` pour la liste des coefficients et `cas = 2` pour la liste des tuples représentant chaque monôme.

6) Proposer des jeux d'essai permettant d'illustrer le travail effectué.

Partie 2

Question 4 Manipulation de files

1) On considère l'algorithme suivant :

```
u ← defiler(A)
v ← defiler(B)
tant que u ≠ 1 et v ≠ 1
    si u > v
        enfiler(u, A)
        u ← defiler(A)
    sinon si u < v
        enfiler(v, B)
        v ← defiler(B)
    sinon
        u ← defiler(A)
        v ← defiler(B)
si u = 1
    enfiler(u, A)
    tant que v ≠ 1
        enfiler(v, B)
        v ← defiler(B)
    enfiler(v, B)
sinon
    enfiler(v, B)
    tant que u ≠ 1
        enfiler(u, A)
        u ← defiler(A)
    enfiler(u, A)
```

Appliquer cet algorithme aux deux files A et B données ci-dessous. On donnera l'état des variables u et v ainsi que des files A et B à la fin de chaque passage de boucle.
Puis ce qui se passe après la boucle tant que.
On pourra compléter le schéma / tableau donné ci-dessous.

Etat initial (sens de la file --->)

A	1	2	2	3	5	5	7
B	1	2	3	3	5	11	

Avant l'entrée dans la boucle

u	7					
v	11					
A	1	2	2	3	5	5
B	1	2	3	3	5	

Etat après passage n°1

u	7					
v	5					
A	1	2	2	3	5	5
B	11	1	2	3	3	

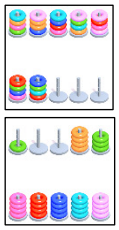
2) On considère les nombres $a = 2100$ et $b = 990$ ainsi que $f = \frac{a}{b}$.

A quoi correspondent les files A et B pour les nombres a et b ?
A quoi pourrait servir cet algorithme ?

3) Faire une programmation Python de cet algorithme.

Partie 3

Faire une recherche sur internet « jeu sort hoop » et faire quelques parties.



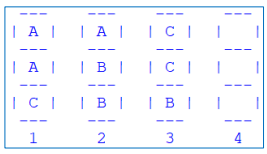
Le principe du jeu en ligne

On ne peut déplacer un pion que sur un pion de même couleur ou sur un pieu vide.
Un pieu plein ne peut plus recevoir de pion.
C'est gagné lorsque les mêmes couleurs sont regroupées sur un même pieu.

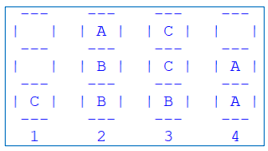
Le projet

L'idée est de construire une classe `pile` ayant pour attributs `contenu` qui est la liste des éléments et `taille` qui le nombre maximal d'éléments (modélise un pieu)
Les méthodes classiques :
`EstVide`, `EstPlein`, `depiler`, `empiler` auxquelles on ajoute `sommet`, `NbrElt` et `depalce` qui prend pour argument une pile et déplace le sommet d'une pile vers l'autre.
Un premier travail pourrait être la création de cette classe.

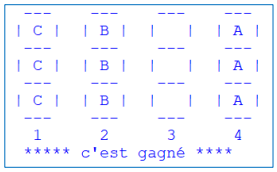
On adapte le jeu de la façon suivante : on choisit un nombre de couleurs n entre 3 et 7.
Le jeu est constitué alors de n pieux contenant n pions de valeurs comprises entre 0 et $n-1$.
On ajoute de 1 à 3 pieux vides. On affiche des lettres au lieu des valeurs (des couleurs) :
Pour $n = 3$



Etat initial

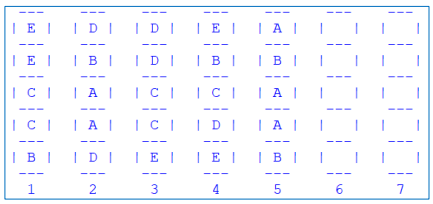


Etat intermédiaire

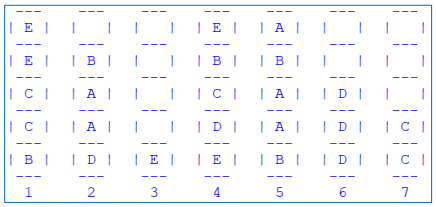


Etat final

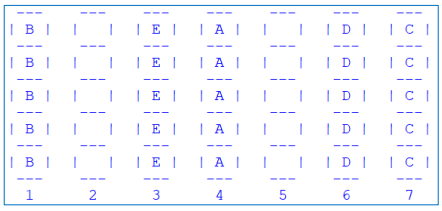
Pour $n = 5$



Etat initial



Etat intermédiaire



Etat final

On crée alors une classe `JeuHoop`
Les attributs sont :
`LesPieux`, une liste de piles qu'il faut initialisées de façon aléatoire.
`NbPieu` le nombre de pieux
`hauteur` le nombre maximal de pions par pieux
Une méthode d'affichage et une méthode `gagnant`.
On aborde là des choses un peu plus complexes.

On complète avec une fonction de menu et lecture du pion à déplacer.
On peut compléter avec la possibilité de reprendre une partie au début ou d'annuler de un à plusieurs coups, une méthode de gestion du stock des états (une pile de jeu) est alors nécessaire.

Je n'ai pas étudié le nombre de pieux vides à ajouter suivant la valeur de n ou s'il existe des situations sans solution. Si on joue mal certaine situation semble ne pas avoir de solution. Non traitée dans le programme.

Eléments de correction

Partie 1

Une version Python : diu_bloc4_moinet_grB_Part1.py

Question 1 Evaluation d'un polynôme en une valeur de a donnée.

1) On a

`L=[3,0,2,1]` et `a = 2`

`Res = 0`

`Len(L) = 4`

`k = 0  res=res+L[0]*2**0 = 0 + 3 = 3`

`k = 1  res=res+L[1]*2**1 = 3 + 0 = 3`

`k = 2  res=res+L[2]*2**2 = 3 + 2*2**2 = 11`

`k = 3  res=res+L[3]*2**3 = 11 + 1*2**3 = 19`

Ici L représente le polynôme $P(x) = 3 + 2x^2 + x^3$ et pour $a = 2$, $P(a) = P(2) = 3 + 2\times2^2 + 2^3 = 3 + 8 + 8 = 19$.
La fonction `evalue1` renvoie la valeur d'un polynôme représenté par la liste `L` de ses coefficients classés par ordre croissant en la valeur `a`.

2) La fonction `sum` ajoute tous les éléments d'une liste. Ici on doit ajouter la valeur de `a` à une puissance donnée multipliée par le coefficient correspondant. On peut donc écrire :

```
def Eval2(L,a):
  return sum([L[k]*a**k for k in range(len(L))])
```

3) `Horner1` est la version itérative et `Horner2` la version récursive.

```
# Question 1
#-----------
def Eval1(L,a):
  res=0
  for k in range(len(L)):
      res=res + L[k]*a**k
  return res

def Eval2(L,a):
  return sum([L[k]*a**k for k in range(len(L))])

def Horner1(L,a):
    res=0
    n=len(L)-1
    for i in range(n):
       res=a*(res+L[n-i])
    return(res+L[0])

def Horner2(L,a,i):
  # Données : a un nombre
  #           L une liste de nombres (coeff du polynôme par ordre croissant des degrés)
  #           i un entier (indice et degré)
  # P(x)= 4x^3 + 5x^2 + 3x +2 = x*(x*(x*(4) + 5) + 3) + 2
  # Evalue un polynome en x=a donné par la méthode de Horner
  if i==len(L)-1: # on atteint le plus haut degré
      return L[i]
  else:
      return(a*Horner1(L,a,i+1)+L[i])
```

```
# Jeux d'essai pour la question 1
# -------------------------------
print("Jeux d'essai de la quetion 1")
txt=[" P(x) = 3 et a = 2","P(x) = 1+2X+3X^2 et a = 1","P(x) = 1+X+3X^3 et a = 2"]
Pol=[[3],[1,2,3],[1,1,0,3]]
Val=[2,1,2]
n=len(txt)
for k in range(n):
  print("Essai n°",k+1,txt[k])
  print("avec Eval1 :",Eval1(Pol[k],Val[k]))
  print("avec Eval2 :",Eval2(Pol[k],Val[k]))
  print("Avec Horner1 :",Horner1(Pol[k],Val[k]))
  print("Avec Horner2 :",Horner2(Pol[k],Val[k],0))
  print("------")
```

```
Jeux d'essai de la quetion 1
Essai n° 1  P(x) = 3 et a = 2
avec Eval1 : 3
avec Eval2 : 3
Avec Horner1 : 3
Avec Horner2 : 3
------
Essai n° 2 P(x) = 1+2X+3X^2 et a = 1
avec Eval1 : 6
avec Eval2 : 6
Avec Horner1 : 6
Avec Horner2 : 6
------
Essai n° 3 P(x) = 1+X+3X^3 et a = 2
avec Eval1 : 27
avec Eval2 : 27
Avec Horner1 : 27
Avec Horner2 : 27
------
```

Question 2 Programmation objet

On souhaite créer une classe `polynome1` permettant de modéliser des polynômes à coefficients entiers positifs dont les attributs sont :

`Coeff` qui est la liste des coefficients comme précité et
`deg`, le degré du polynôme

Travail à effectuer.

1) 2) 3) Ecrire la classe `polynome1` représentant un polynôme P.

```
class polynome1:
    '''Classe modélisant un polynome à coefficient entiers positifs par :
       - la liste de ces coefficients par ordre croissant des degrés
       - son degré
       Polynome(1,0,3,5) modélise le polynome P : P(x) = 1 + 3x^2 + 5x^3
       Son degré (deg) est 3.

       Les fonctionnalités :
       - Affichage
       - Dérivée
       - Ajout avec un autre polynome
       - Produit avec un autre polynome (non fait)
       - Evaluation pour une valeur de a donnée '''

    '''Le constructeur'''
    # Les coefficients sont donnés dans l'ordre croissant des coefficients
    def __init__(self, *args):
       self.Coeff=[args[i] for i in range(len(args))]
       self.deg=len(self.Coeff)-1


    '''Les méthodes'''
    # Ecriture du polynôme
    def __repr__(self):
        if self.Coeff[0]==0:
            txt=""
        else:
            txt=str(self.Coeff[0])
        for k in range(1,len(self.Coeff)):
            if self.Coeff[k]!=0:
               if txt=="":
                   if self.Coeff[k]==1:
                     if k==1:
                        txt=txt+"X"
                     else:
                        txt=txt+"X^"+str(k)
                   else:
                     if k==1:
                        txt=txt+str(self.Coeff[k])+"X"
                     else:
                        txt=txt+str(self.Coeff[k])+"X^"+str(k)
               else:
                   if self.Coeff[k]==1:
                     if k==1:
                        txt=txt+"+X"
                     else:
                        txt=txt+"+X^"+str(k)
                   else:
                     if k==1:
                        txt=txt+"+"+str(self.Coeff[k])+"X"
                     else:
                        txt=txt+"+"+str(self.Coeff[k])+"X^"+str(k)
        return txt


    # renvoie le polynome dérivé
    def derive(self): # Donne le polynome dérivé
        if len(self.Coeff)==1:
            return Polynome(0)
        else:
            return polynome1(*[(i+1)*self.Coeff[i+1] for i in range(len( self.Coeff)-1)])

    # Renvoie la somme de deux polynômes
    def plus(self,Q):
        n=self.deg
        m=Q.deg
        S=[]
        for i in range(0,min(m,n)+1):
            S.append(self.Coeff[i]+Q.Coeff[i])
        if n<m:
           S=S+Q.Coeff[n+1:]
        else:
           S=S+self.Coeff[m+1:]
        return polynome1(*[S[i] for i in range(len(S))])

    # Evalue le polynôme en une valeur de a donnée, avec la méthode de Horner
    def evalue(self,a):
       L=self.Coeff
       return Horner2(L,a,0)
```

4) Faire les jeux d'essai suivants :

```
# Jeux d'essai pour la question 2
# ------------------------------
print("Jeux d'essai de la question 2")
print(30*"-")
P=polynome1(5,0,3,1)
Q=polynome1(0,3,1,6,2,3)
print("Affichage des polynomes P et Q")
print("   P(x) =",P)
print("   Q(x) =",Q)
print("Création et affichages des polynomes dérivés")
dP=P.derive()
dQ=Q.derive()
print("   P'(x) =",dP)
print("   Q'(x) =",dQ)
print("Evaluation de P en a = 0, a = 2 et a = - 1")
print("   P(0)=",P.evalue(0))
print("   P(2)=",P.evalue(2))
print("   P(-1)=",P.evalue(-1))
print("Création et affichages de la somme P+Q")
S=P.plus(Q)
print("   S(x)=P(x)+Q(x)=",S)
print("Affichages de S'(x) et S'(1)")
res=S.derive().evalue(1)
print("   S'(1)=",res)
print()
print("********")
print()
```

```
Jeux d'essai de la question 2
------------------------------
Affichage des polynomes P et Q
   P(x) = 5+3X^2+X^3
   Q(x) = 3X+X^2+6X^3+2X^4+3X^5
Création et affichages des polynomes dérivés
   P'(x) = 6X+3X^2
   Q'(x) = 3+2X+18X^2+8X^3+15X^4
Evaluation de P en a = 0, a = 2 et a = - 1
   P(0)= 5
   P(2)= 25
   P(-1)= 7
Création et affichages de la somme P+Q
   S(x)=P(x)+Q(x)= 5+3X+4X^2+7X^3+2X^4+3X^5
Affichages de S'(x) et S'(1)
   S'(1)= 55
```

Question 3 Programmation objet

Travail à effectuer.

1) Ecrire la classe `polynome2` représentant un polynôme P comme somme de monômes.

```
class polynome2:

   '''Le constructeur'''
   def __init__(self,*args):
     # monomes : la liste des tuple représentant chaque monomes
     self.monomes=[]
     for i in range(len(args)):
         self.monomes.append((args[i][0],args[i][1]))
     # deg : le degré du polynome
     self.deg=self.monomes[len(self.monomes)-1][0]
```

2) Compléter la classe `polynome2` avec les méthodes suivantes : (affichage et dérivation)

```
def __repr__(self):
  txt=""
  for m in self.monomes:
    if m[0]==0:
        if m[1]==0:
          txt="0"
        else:
          txt=str(m[1])
    elif m[0]==1:
        if txt=="":
            txt=str(m[1])+"X"
        else:
            txt=txt+"+"+str(m[1])+"X"
    else:
        if txt=="":
            txt=str(m[1])+"X^"+str(m[0])
        else:
            txt=txt+" + "+str(m[1])+"X^"+str(m[0])
  return txt
```

```
def derive(self):
   return polynome2(*[(m[0]-1,m[0]*m[1]) for m in self.monomes if m[0]>0])
```

3) Proposer des jeux d'essai.
Pas de correction proposée

4) La somme de deux polynômes semble plus compliquée (moins intuitive) qu'avec l'implémentation précédente. Les méthodes `evalue` et `derive` (et autres si besoin) étant déjà disponibles, on souhaite pouvoir passer d'une implémentation à une autre pour ne pas avoir à tout écrire.

a) Compléter le constructeur de la classe `polynome2` pour obtenir un attribut `poly1` qui est la liste ordonnée croissante de tous les coefficients.

```
# poly1 : la liste de tous les coefficients, de la constante jusqu'au degré
A=[] # liste des degrés de coefficient non nul
for k in range(len(self.monomes)):
    A.append(self.monomes[k][0])
self.poly1=[]
rang=0 # indice du tuple (monome) dans la liste des monomes
for k in range(self.deg+1): # on parcourt tous les degrés jusqu'au plus haut
    if k in A: # Si c'est un degré de coeff non nul
        self.poly1.append(self.monomes[rang][1]) # on ajoute le coeff correspondant
        rang+=1 # on passe au rang du monome de coeff non nul suivant à rencontrer
    else:
        self.poly1.append(0) # Si le degré n'est pas présent, le coeff est nul
```

b) Compléter le constructeur de la classe `polynome1` pour obtenir un attribut `poly2` qui est la liste ordonnée croissante des tuples représentant chaque monôme.

```
A=[]
for k in range(self.deg+1):
    if self.Coeff[k]!=0:
        A.append((k,self.Coeff[k]))
self.poly2=A
```

c) Compléter alors la classe `polynome2` avec une méthode permettant d'ajouter deux polynômes en utilisant la méthode `plus` de la classe `polynome1` ainsi que les attributs `poly1` et `poly2`.

```
def plus(self,Q):
    P1=polynome1(*[self.poly1[i] for i in range(len(self.poly1))])
    Q1=polynome1(*[Q.poly1[i] for i in range(len(Q.poly1))])
    S=P1.plus(Q1)
    return polynome2(*[S.poly2[i] for i in range(len(S.poly2))])
```

5) Ecrire une fonction `Saisie()` prenant pour argument un entier `cas` prenant pour valeur 1 ou 2 et permettant de saisir les données pour un polynôme, `cas = 1` pour la liste des coefficients et `cas = 2` pour la liste des tuples représentant chaque monôme.

```
def Saisie(cas):
# Donnée : cas un entier valant 1 ou 2
# Résultat : une liste d'entiers ou une liste de tuples d'entiers
  P=[]
  if cas==1:
    n=int(input("Degré du polynôme : "))
    for k in range(n+1):
      print("Coefficicient du monome de degré",k," : ",end="")
      coeff=int(input(" : "))
      P.append(coeff)
  else:
    n=int(input("Nombre de monômes à saisir : "))
    for k in range(n):
      print("degré du monome n°",k+1," : ",end="")
      d=int(input(" : "))
      print("Coefficicient du monome de degré",d," : ",end="")
      c=int(input(" : "))
      P.append((d,c))
  return P
```

6) Proposer des jeux d'essai permettant d'illustrer le travail effectué.
Pas de correction proposée

Partie 2

Une version Python : diu_bloc4_moinet_grB_Part1.py

Question 4 Manipulation de files

1) Exécuter un algorithme sur les files.

Etat initial (sens de la file --->)

A	1	2	2	3	5	5	7
B	1	2	3	3	5	11	

Avant l'entrée dans la boucle

u 7
v 11

A	1	2	2	3	5	5
B	1	2	3	3	5	

Etat après passage n°1

u 7
v 5

A	1	2	2	3	5	5
B	11	1	2	3	3	

Etat après passage n°2

u 5
v 5

A	7	1	2	2	3	5
B	11	1	2	3	3	

Etat après passage n°3

u 5
v 3

A	7	1	2	2	3
B	11	1	2	3	

Etat après passage n°4

u 3
v 3

A	5	7	1	2	2
B	11	1	2	3	

Etat après passage n°5

u 2
v 3

A	5	7	1	2	
B	11	1	2		

Etat après passage n°6

u 2
v 2

A	5	7	1	2	
B	3	11	1		

Etat après passage n°7

u 2
v 1

A	5	7	1		
B	3	11			

Fin de l'exécution

u 2
v 1

A	5	7	1		
B	1	3	11		

u 1
v 1

A	2	5	7		
B	1	3	11		

Etat final

u 1
v 1

A	1	2	5	7	
B	1	3	11		

2) On considère les nombres $a = 2100$ et $b = 990$ ainsi que $f = \frac{a}{b}$.

Les files A et B donnent la décomposition en facteurs premiers des nombres a et b.
A l'issue de l'algorithme le produit des valeurs vaut 70 et celles de b vaut 33 et $f = \frac{a}{b} = f = \frac{2100}{990} = \frac{70}{33}$

Cet algorithme pourrait permettre de simplifier des fractions de deux entiers dont on connait la décomposition en facteurs premiers.

3) Faire une programmation Python de cet algorithme.

```
def Splf(M,N):
    A=M[:]
    B=N[:]
    u=A.pop()
    v=B.pop()
    while u!=1 and v!=1:
        if u>v:
            A=[u]+A
            u=A.pop()
        elif u<v:
            B=[v]+B
            v=B.pop()
        else:
            u=A.pop()
            v=B.pop()
        print("u =",u)
        print("v =",v)
        print("A =",A)
        print("B =",B)
    if u==1:
        A=[u]+ A
        while v!=1:
            B=[v]+B
            v=B.pop()
        B=[v]+ B
    else:
        B=[v]+ B
        while u!=1:
            A=[u]+A
            u=A.pop()
        A=[u]+ A
    return[A,B]
```

Partie 3

Une version Python : Jeu_Hoop.py