```
# ============================== #
# PSI*                           #
# DS d'informatique du 18/11/2017 #
# Corrigé de l'exercice 1        #
# ============================== #

## Importations

import numpy as np

## La classe MatInt

class MatInt():
    """Classe pour représenter et manipuler des matrices d'entiers sous
forme de
    listes de listes
    """
    def __init__(self,L):
        self.coeff = list(L)
        self.row = len(L)
        self.col = len(L[0])

    def __repr__(self):
        """Méthode __repr__ de représentation des objets MatInt.
        On introduit un saut de ligne... entre chaque ligne pour obtenir un
        affichage globalement satisfaisant...
        """
        s = str(self.coeff[0])
        for i in range(1,len(self.coeff)):
            s += "\n" + str(self.coeff[i])
        return(s)

    def __mul__(self,B):
        """Surcharge de l'opérateur __mul__ pour pouvoir effectuer des
produits
        matriciels avec les objets MatInt en utilisant le symbole *
        """
        # On s'assure d'abord que B est bien du type MatInt
        if type(B) != MatInt:
            raise TypeError("Le deuxième facteur n'est pas du type MatInt
!")
        # On vérifie ensuite que les dimensions sont compatibles
        elif self.col != B.row:
            raise ValueError("Les dimensions des matrices ne permettent pas
d'effecter le calcul !")
        # On peut effectuer le calcul
        else:
            P = []
            for i in range(self.row):
                LigneCourante = []
                for j in range(B.col):
                    NewCoeff = 0
                    for k in range(self.col):
                        NewCoeff += self.coeff[i][k] * B.coeff[k][j]
                    LigneCourante.append(NewCoeff)
                P.append(LigneCourante)
            return(MatInt(P))

    def __pow__(self,n):
        """Surcharge de l'opérateur __pow__ pour pouvoir effectuer des
```

```
        exponentiations de matrices carrées avec les objets MatInt en
utilisant
        le symbole **
        On impose à l'exposant d'être positif (on ne tient donc pas compte
de
        l'inversibilité éventuelle de self et on met en oeuvre la méthode
        d'exponentiation rapide
        """
        # n doit être un entier naturel
        if type(n) != int:
            raise TypeError("L'exposant n'est pas entier !")
        elif n < 0:
            raise ValueError("L'exposant doit être positif !")
        # La matrice doit être carrée
        elif self.row != self.col:
            raise ValueError("La matrice doit être carrée !")
        else:
            dim = self.row
            # Si n est nul, on renvoie l'identité
            if n == 0:
                I = []
                for i in range(dim):
                    I.append([0] * dim)
                for i in range(dim):
                    I[i][i] = 1
                return(MatInt(I))
            # n est égal à 1
            elif n == 1:
                return(self)
            # n est égal à 2
            elif n == 2:
                return(self * self)
            # n est strictement supérieur à 2
            else:
                (q,r) = divmod(n,2)
                P = (self**q)**2
                if r == 1:
                    P = P * self
                return(P)

## Question 1

def A_build(ALPHA):
    p = len(ALPHA)
    # Création d'un tableau carré d'ordre p contenant des 0 (entiers)
    A = np.zeros((p,p),dtype=int)
    # Remplacement de la première ligne de A par les éléments de ALPHA
    A[0] = ALPHA
    # Remplissage de la diagonale sous la diagonale principale : j = i-1
    for i in range(1,p):
        A[i,i-1] = 1
    # Renvoi de A
    return(A)

## Question 2

# Cf. le chapitre sur la récursivité

## Question 3
```

```sage
def A_build2(ALPHA):
    # Longueur de la liste passée en argument
    n = len(ALPHA)
    # A est une liste de liste. Son premier élément est la liste ALPHA
    A=[ALPHA]
    # On ajoute n-1 listes de n 0 à A
    for i in range(n-1):
        A.append(n*[0])
    # Remplissage de la diagonale sous la diagonale principale : j = i-1
    for i in range(1,n):
        A[i][i-1] = 1
    # Renvoi de A
    return(MatInt(A))

## Question 4

def SRL(ALPHA,INIT,n):
    L = len(ALPHA)
    if n <= L-1:
        return(INIT[n])
    else:
        # Construction de la matrice colonne des premiers termes de la suite
        INIT2 = []
        for i in range(L):
            INIT2.append([INIT[-1-i]])
        Vinit = MatInt(INIT2)
        # Renvoi de la valeur de u(n)
        return((A_build2(ALPHA)**(n-L+2)*Vinit).coeff[1][0])

## Pour tester...

# Suite de FIBONACCI
# ALPHA = [1,1]
# INIT = [0,1]

# La suite définie par u(n+3) = u(n+2) -2 u(n+1) + 3 u(n) et u(0) = -5, u(1)
= 7
# et u(2) = 2
ALPHA = [1,-2,3]
INIT = [-5,7,2]

n = int(input("Veuillez saisir le rang du terme à calculer : "))
print("u(",n,") =",SRL(ALPHA,INIT,n))
```