Logique

Écrire un algorithme de simulation de ce jeu qui se terminera par l'affichage du vainqueur ainsi que le nombre de tours complets parcourus par ce vainqueur.
Le lancement du dé sera simulé par l'appel du module sans argument `LancerDé( )` qui retourne une valeur aléatoire entre 1 et 6.

Aide : Définissez la classe `JeuPoursuite`

- Elle permet de représenter
  - le circuit des 50 cases
  - la position des 2 joueurs
  - le nombre de tours effectués par chacun des joueurs
  - qui est le joueur courant
  - Plusieurs possibilités existent ; faites votre choix !
- Le constructeur reçoit la configuration du circuit (pour savoir si les cases contiennent `vrai` ou `faux`)
- La méthode `initialiser()` initialise le jeu (placement des joueurs, ...).
- La méthode `jouer()` lance le jeu jusqu'à son terme et donne le vainqueur et le nombre de tours effectués.
- Vous êtes également fortement invités à définir d'autres méthodes en privé pour modulariser au mieux votre code. Par exemple, on pourrait définir
  - la méthode « `jouerCoup` » qui joue pour un joueur et indique s'il a rattrapé l'autre joueur (sans répétition si on arrive sur une case `vrai`)
  - la même méthode « `jouerTour` » effectue la même tâche mais avec répétition si on arrive sur une case `vrai`. On fera évidemment appel à la méthode ci-dessus.
  - la méthode « `joueurSuivant` » qui permet de passer au joueur suivant.

  Avec ces 3 méthodes, la méthode publique « `jouer` » devient triviale.

1)

classe JeuPoursuite

privé :
- plateau : tableau de booleens de [1à10,1à5]
- joueur1 : entier
- joueur2 : entier
- tourJ1 : entier
- tourJ2 : entier
- joueurCourant : entier

public :
- constructeur JeuPoursuite(jeu : tableau de booleens de [1à10,1à5])
- methode getJoueur1() → entier
- methode getJoueur2 () → entier
- methode getTourJ1() → entier
- methode getTourJ2() → entier
- methode getJoueurCourant() → entier
- methode initialiser()
- methode jouer ()

privée :
- methode jouerCoup(joueur : entier) → booleen
- methode jouerTour(joueur : entier) → booleen
- methode joueurSuivant() → entier

```
constructeur jeuPoursuite (….)

        pour i allant de 1à10 faire
                pour j allant de 1à5 faire
                        plateau[i,j] ← jeu[i,j]
                fin pour
        fin pour
        initialiser()
fin constru

methode initialiser

        joueur1 ← 1
        joueur2 ← 26
        tourJ1 ← 0
        tourJ2 ← 0
        joueurCourant ← 1
fin methode

methode getJoueur1() → entier

        retourner joueur1
fin methode
methode getJoueur2() → entier

        retourner joueur2
fin methode

methode getJoueurCourant() → entier
        retourner joueurCourant
fin methode

methode jouerCoup (joueur : entier) → booleens

        dé : entier
        dé ← lancerDés()
        joueur ← joueur + dé
        si joueur > 50
                joueur ← joueur MOD 50
                si getJoueurCourant = 1
                        tourJ1++
                sinon
                        tourJ2++
                fin si
        fin si
        si joueurCourant = 1
                joueur1<-joueur
                si ((joueur1 > joueur2 et tourJ1 = tourJ2)OU(joueur1<joueur2 et tourJ1>tourJ2))
                        retourner vrai
                sinon
                        retourner faux
                fin
```

```
        sinon

                si ((joueur2 > joueur1 et tourJ1=tourJ2) OU (joeur2<joueur1 et tourJ2>tourJ1))
                        retourner vrai
                sinon
                        retourner faux
                fin
        fin
fin methode

methode jouerTour(joueur ↔  : entier,) → booleen
        x,y : entier
        ok,repeat : booleen
        repeat ← vrai
        ok ← jouerCoup(joueur)
        tant que(!(ok)) ET (repeat)
                si joueur > 10
                        x ← (joueur DIV 10)+ 1
                si joueur MOD 10 = 0
                        y<- 10
                sinon
                        y ← joueurMod10
        sinon
                x ← 1
                si joueur MOD 10 = 0
                        y <- 10
                sinon
                        y<- joueur MOD 10

        fin si
        si plateau [x,y]!= vrai
                repeat ← faux

        sinon
                ok ← jouerCoup(joueur)

        fin si
        fin TQ
        retourner ok
        fin methode

methode joueurSuivant() → entier
        si getJoueurCourant=1
                joueurCourant ← 2
        sinon
                joueurCourant ← 1
        fin si
        retourner joueurCourant
fin methode

methode jouer()
        enJeu : booleens
```

```
        tant que (enJeu)
                enJeu <- jouerTour(joueurCourant)
                joueurCourant ← joueurSuivant()
        fin tant que
        si joueurCourant = 1
                ecrire «  le joueur 1 a gagné «
        sinon
                ecrire « le joueur2 a gagné »
        fin si
fin methode
```

EX2)

Une commune gère la liste de ses couples mariés dans un fichier **UNIONS**. Ce fichier est classé dans l'ordre chronologique des mariages et chacun de ses enregistrements, de type *Union*, contient les champs suivants :

- DATE Date date du mariage
- NOM1 chaine nom du premier conjoint
- NUM1 chaine numéro de registre national du premier conjoint
- NOM2 chaine nom du second conjoint
- NUM2 chaine numéro de registre national du second conjoint

Les demandes de divorces durant une certaine année ont été stockées dans le fichier **DIVORCES** et ses enregistrements sont similaires à ceux de **UNIONS** : chaque enregistrement contient également les noms et numéros des deux conjoints demandant la séparation, et la date est cette fois-ci celle du divorce. Le fichier est également classé par ordre chronologique des demandes de divorces. On peut supposer l'absence d'erreur, c'est-à-dire que tous les enregistrements de **DIVORCES** ont été répertoriés dans **UNIONS**.

Sur base de ces données, écrire **un** algorithme qui :

1. crée un fichier **UNIONSUPDATE** résultant de la mise à jour du fichier **UNIONS**. Le nouveau fichier ne contiendra plus les couples divorcés renseignées dans le fichier des divorces. Pour comparer les contenus des fichiers en entrée, on se fiera uniquement aux numéros de registre national (car il est possible que des personnes différentes puissent avoir le même nom). Attention, il est possible que les noms des conjoints soient inversés dans **DIVORCES** par rapport à l'ordre de **UNIONS**. Veillez aussi à comparer les deux noms avant de supprimer un enregistrement de **UNIONS**, car on peut imaginer le cas de personnes se remariant et divorçant plusieurs fois dans l'année !
2. donne le mois de l'année où le nombre de divorce a été le plus élevé.

```
structure Union
date : date
num1:entier
nom1 : chaine
num2 : entier
nom2 : chaine
fin structure

module Divorces( Divorces, Unions , fichier en entrée d'Union
        marié,divorcé,update : Liste d'Union
        tailleUnions,tailleDivorces,i,j : entier
        mariage,divorce : Union
        Unions.ouvrir
        mariage ← Unions.lire()
        marié ← nouvelle liste d'Union ()
        tant que non eof(Unions)
                marié.ajouter(mariage)
                tailleUnion++
```

```
            mariage ← Unions.lire()
        fin TQ
        Unions.fermer()
        Divorces.ouvrir()
        divorce ← Divorces.lire()
        divorcé ← nouvelle liste d'Union()
        tant que non eof(Divorces)
            divorcé.ajouter(divorce)
            tailleDivorce++
            divorce ← Divorce.lire()
        fin tant que
        Divorces.fermer()

        update ← nouvelle Liste d'Unions
        tant que (i <= marié.taille())
            ok ← recherche(marié.get(i).num1,marié.get(i).nom1,Divorces)
            si!(ok)
                update.ajouter(marié.get(i))
            fin si
        fin Tant que
        record ← rechercheMoisRecord(Divorces)
        UpdateUnions : fichier sortie d' Union
        UpdateUnion.ouvrir()
        enr : Union
        tant que i<= update.taille()
            UpdateUnion.ecrire(Divorces .get(i))
        fin tantque
        UpdateUnion.fermer()
fin module



module rechercheMoisRecord(liste : liste d'Union)
        calendrier : nouveau tableau de [1à12] d'entier
        tant que i<= liste.taille()
            tab[liste.get(i).date.getMois()]++
        fin tant que
        max ← tab[1]
        maxI ← 1
        pour i allant de 2 à tab.taille()
            si tab[i] > max
                i ← maxI
            fin si
        fin pour
        retourner maxI
fin module

module recherche(num1:entier, nom1 : chaine, liste : liste d'union) → booleen
        ok : booleen
        ok ← faux
        pour i allant de 1 a liste.taille()
            si liste.get(i).num1 = num1 ou liste.get(i).num2 = num1
```

```
                    si liste.get(i).nom1 = nom1 ou liste.get(i).nom2 = nom1
                        ok ← vrai
                fin si
        fin pour
        retourner ok
fin module
```