X - 2016 - COMPOSITION D'INFORMATIQUE - (XCR)
PSI, PT

J. Larochette - Lycée Carnot, Dijon - pour serveur UPS.

Partie I. Simulation du mouvement des particules

Question 1.

```
def deplacerParticule(particule, largeur, hauteur):
    """prend en paramètre une particule à l'instant t ainsi que les dimensions du
rectangle et renvoie la particule à l'instant t + 1, en tenant compte des rebonds."""
    x, y, vx, vy = particule
    if x + vx <= 0 or x + vx >= largeur: vx = -vx
    if y + vy <= 0 or y + vy >= hauteur: vy = -vy
    return x + vx, y + vy, vx, vy
```

Partie II. Représentation par une grille

Question 2. Attention à la copie de listes!

```
def nouvelleGrille(largeur, hauteur):
    """renvoie une nouvelle grille vide de dimensions largeur x hauteur."""

    grille = []
    for _ in range(largeur):
        ligne = []
        for _ in range(hauteur): ligne.append(None)
        grille.append(ligne)

    return grille
```

Ou avec des listes en compréhension (autorisées?) :

```
def nouvelleGrille(largeur, hauteur):
    """renvoie une nouvelle grille vide de dimensions largeur x hauteur."""
    return [ [ None for _ in range(hauteur) ] for _ in range(largeur) ]
```

Question 3.

```
def majGrilleOuCollision(grille):
    """prend en paramètre une grille contenant des particules à l'instant t et renvoie
une nouvelle grille contenant ces particules à l'instant t + 1 s'il n'y a pas eu de
collision. Si une collision survient, la fonction renvoie None."""
    largeur, hauteur = len(grille), len(grille[0])

    nouvelle_grille = nouvelleGrille(largeur, hauteur)
    for i in range(largeur):
        for j in range(hauteur):
            if grille[i][j] is not None: # S'il y a une particule en (i,j)
                particule_deplacee = deplacerParticule(grille[i][j], largeur, hauteur)
                x, y, _, _ = particule_deplacee
                if nouvelle_grille[int(x)][int(y)] is not None : return None #collision

                nouvelle_grille[int(x)][int(y)] = particule_deplacee

    return nouvelle_grille
```


Question 4.

```
def attendreCollisionGrille(grille, tMax):
    """prend une grille de particules en paramètre et renvoie le temps où a eu lieu
la première collision entre deux particules.
S'il n'y a pas de collision avant le temps tMax, la fonction renvoie None."""
    t = 0
    while t < tMax and grille is not None:
        t += 1
        grille = majGrilleOuCollision(grille)
    if t != tMax: return t # collision au temps t
```

(Inutile de mettre le `return None` ici.)

Question 5. La fonction `attendreCollisionGrille` effectue au plus `tMax` – 1 tours de boucle `while`, il y a donc au plus `tMax` appels à `majGrilleOuCollision` appliquée à des grilles de format `largeur` × `hauteur`, le reste étant en $O(\texttt{tMax})$.

Or `majGrilleOuCollision` fait un appel à `nouvelleGrille` qui a une complexité en $O(\texttt{largeur}\times\texttt{hauteur})$ et au plus `largeur`×`hauteur` fois un nombre majoré d'opérations (`deplacerParticule` s'exécute en $O(1)$), soit une complexité en $O(\texttt{largeur}\times\texttt{hauteur})$.

Finalement, la complexité temporelle de `attendreCollisionGrille` est $O(\texttt{largeur}\times\texttt{hauteur}\times\texttt{tMax})$.

Partie III. Représentation par une liste de particules

Question 6. Il y a collision lorsque la distance euclidienne entre les deux particules est au plus égale au double du rayon.

```
def detecterCollisionEntreParticules(p1, p2):
    """prend en paramètre deux particules et renvoie True si les particules sont en
collision et False sinon."""
    x1, y1, _, _ = p1
    x2, y2, _, _ = p2
    return (x1 - x2)**2 + (y1 - y2)**2 <= 4 * rayon**2
```

Question 7.

```
def maj(particules):
    """prend en paramètre un ensemble de particules à l'instant t et renvoie un
ensemble contenant les particules à l'instant t + 1, sans s'occuper des collisions
éventuelles."""
    largeur, hauteur, listeParticules = particules
    nouvelle_liste = []
    for particule in listeParticules:
        nouvelle_liste.append(deplacerParticule(particule, largeur, hauteur))
    return largeur, hauteur, nouvelle_liste
```

Question 8.

```
def majOuCollision(particules):
    """prend en paramètre un ensemble de particules à l'instant t et qui renvoie un
ensemble contenant les particules à l'instant t + 1, s'il n'y a pas eu de collision à
l'instant t + 1. S'il y a eu une collision, la fonction renvoie None."""
    nouvelles_particules = maj(particules)
    listeParticules = nouvelles_particules[2]
    n = len(listeParticules)
    for i in range(n-1):
        for j in range(i+1, n): # on prend des particules différentes
            if detecterCollisionEntreParticules(listeParticules[i], listeParticules[j]):
                return
    return nouvelles_particules
```

Question 9.

```
def attendreCollision(particules, tMax):
    """prend un ensemble de particules et un temps tMax en paramètres et renvoie le
temps où a eu lieu la première collision entre deux particules. S'il n'y a pas de
collision avant le temps tMax, la fonction renvoie None."""
    t = 0
    while t < tMax and particules is not None:
        t += 1
        particules = majOuCollision(particules)
    if t != tMax: return t # collision au temps t
```

Au plus `tMax` appels de `majOuCollision(particules)` dans laquelle `maj(particules)` a un coût en $O(n)$ et faisant appel au plus $\frac{n(n-1)}{2} = O(n^2)$ fois à `detecterCollisionEntreParticules` qui a un coût constant, soit une complexité en $O(n^2 \times \texttt{tMax})$.

Question 10. La distance maximale entre les deux particules lors d'une collision étant $2\times$`rayon`, et la vitesse maximale de chacune permettant d'atteindre cette distance étant `vmax`, elles devront se situer à une distance l'une de l'autre d'au plus $2 \times (\texttt{rayon} + \texttt{vmax} \times 1) = 2(\texttt{rayon} + \texttt{vmax})$ à l'instant t pour avoir une chance d'entrer en collision à l'instant $t+1$.

Question 11. La comparaison se fait sur les abscisses avant déplacement car celles-ci sont triées par ordre croissant.

```
def majOuCollisionX(particules):
    """prend en paramètre un ensemble de particules dont la liste des particules est
    triée par abscisses croissantes ; renvoie un ensemble contenant les particules à
l'instant t+1, sauf si une collision survient entre deux particules, auquel cas la
fonction renvoie None."""
    nouvelles_particules = maj(particules)
    listeParticules = particules[2]
    nvlListeParticules = nouvelles_particules[2]
    n = len(listeParticules)
    dmax = 2*(rayon + vMax) # distance max au temps t pour risque de collision

    for i in range(n-1):
        xi = listeParticules[i][0] # abscisse de la particule i
        j = i + 1
        xj = listeParticules[j][0] # abscisse de la particule j

        while j < n and xj - xi <= dmax:
            if detecterCollisionEntreParticules(nvllisteParticules[i],
                                                nvlListeParticules[j]): return
            j += 1
            xj = listeParticules[j][0]
    return nouvelles_particules
```

Question 12.

```
def scm(s):
    """prend une liste s en paramètre et renvoie la liste ordonnée des couples
d'indices correspondant au partitionnement en scm de s."""
    d = 0
    liste_scm = []
    n = len(s)
    for i in range(n-1):
        if s[i+1] < s[i]: # fin de la scm en cours
            liste_scm.append( (d, i) )
            d = i + 1
    liste_scm.append( (d, n-1) )
    return liste_scm
```


Question 13.

```
def copier(s, debut, fin):
    return s[debut:fin + 1]

def fusionner(s, r1, r2):
    """prend une liste s en paramètre ainsi que deux scm consécutives encodées
par leurs indices de début et de fin, et les fusionne en une seule scm :
si r1 = (d1, f1) et r2 = (d2, f2), alors après l'appel à la procédure,
la partie de s située entre les indices d1 et f2 dans s doit être triée.
Cette procédure ne crée pas une nouvelle liste, elle modifie la liste s."""
    d1, f1 = r1
    d2, f2 = r2 # d2 = f1 + 1 car les scm sont consécutives.
    s1 = copier(s, d1, f1)
    s2 = copier(s, d2, f2)
    i1, i2, i = 0, 0, d1 # indices de parcours de s1, s2 et s

    for _ in range(f2 - d1 + 1): # les scm sont consécutives.
        if i2 == n2 or (i1 < n1 and s1[i1] < s2[i2]):
            s[i] = s1[i1]
            i1 += 1
        else:
            s[i] = s2[i2]
            i2 += 1
        i += 1
        # Si les scm n'étaient pas consécutives, il faudrait passer de i = f1 à i = d2.
```

Question 14.

```
def depileFusionneRemplace(s, pile):
    """prend en paramètre une liste s ainsi qu'une pile de scm (sous la forme de
couples d'indices de début et de fin). Cette procédure devra retirer les deux scm
au sommet de la pile, les fusionner dans la liste s et replacer les indices de la
scm fusionnée au sommet de la pile."""
    r2 = (d2, f2) = pile.pop()
    r1 = (d1, f1) = pile.pop()
    fusionner(s, r1, r2)
    pile.append( (d1, f2) )
```

Question 15.

```
def longueur(r):
    """Renvoie la longueur d'une scm r."""
    d, f = r
    return f - d + 1

def alphaTri(s):
    """prend en paramètre une liste s et trie cette liste en utilisant l'algorithme
alpha-tri. Attention, cette procédure ne crée pas une nouvelle liste, elle modifie la
liste passée en paramètre."""
    liste_scm = scm(s)
    pile = []
    # Première phase
    for i in range(len(liste_scm)):
        pile.append( liste_scm[i] )
        while len(pile) > 1 and longueur(pile[-2]) < 2*longueur(pile[-1]):
            depileFusionneRemplace(s, pile)
     # Deuxième phase
    while len(pile) > 1 : depileFusionneRemplace(s, pile)
```