Solution du C.B. d'I.P.T. 2017

Question 1

```
def deplacerParticule(particule, largeur, hauteur):
    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
```

Question 2

```
SELECT NumeroParticule FROM Position WHERE temps=0 AND (x<1 OR y<1)
```

Question 3. On doit faire une jointure entre les deux tables, suivant les deux attributs `NumeroParticule` et `temps`. Ici la valeur du `temps` est prise à 0.

```
SELECT P.NumeroParticule, x,y,vx,vy FROM Position P, Vitesse V
 WHERE P.NumeroParticule=V.NumeroParticule AND P.temps=V.temps AND V.temps=0
```

ou si l'on préfère avec la syntaxe `JOIN` :

```
SELECT P.NumeroParticule, x,y,vx,vy FROM Position P JOIN Vitesse V
  ON P.NumeroParticule=V.NumeroParticule AND P.temps=V.temps WHERE  V.temps=0
```

Question 4. La requête suivant renvoie un nombre, qui est le max cherché.

```
SELECT MAX(SQRT(vx*vx+vy*vy)) FROM Vitesse WHERE temps=0
```

Question 5. Il faut faire attention qu'il peut y avoir plusieurs particules ayant la vitesse maximale au même temps.

```
SELECT NumeroParticule  FROM Vitesse
 WHERE vx*vx+vy*vy= (SELECT MAX(vx*vx+vy*vy) FROM Vitesse WHERE temps=0)
```

Question 6. Il s'agit ici de faire une *auto-jointure* sur la table `Vitesse` car on doit comparer les entrées entre le temps t et le temps $t+1$.

```
SELECT V1.NumeroParticule, V1.temps AS DateRebond FROM Vitesse V1 JOIN  Vitesse V2
ON  V1.NumeroParticule=V2.NumeroParticule AND V2.temps=V1.temps+1
 WHERE V1.vx*V2.vx<0 OR V1.vy*V2.vy<0 -- condition de rebond
```

Question 7. On doit faire attention au problème de la copie des listes. Il ne s'agit pas de copier à chaque fois la même ligne, sous peine qu'une modification d'une ligne ne modifie toutes les lignes. On peut utiliser une double boucle, comme suit :

```
def nouvelleGrille(largeur, hauteur):
    grille = []
    for i in range(largeur):
        ligne = []
        for j in range(hauteur):
            ligne.append(None)
        grille.append(ligne)
    return grille
```

Question 8. La fonction suivante commence par initialiser une nouvelle grille, qu'elle remplit ensuite en faisant agir `deplacerParticule` sur chacune des entrées de `grille` qui ne sont pas `None` tant qu'elle ne détecte pas des collisions, lesquelles se produisent si la case de `nouvelle_grille` qu'on veut affecter a déjà été modifiée.

```
def majGrilleOuCollision(grille):
    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] != None: # S'il y a une particule en (i,j)
                particule_deplacee = deplacerParticule(grille[i][j],largeur,hauteur)
                x, y, vx, vy = particule_deplacee
                if nouvelle_grille[int(x)][int(y)]!= None :
                     return None  # cas de collision
                nouvelle_grille[int(x)][int(y)] = particule_deplacee
    return nouvelle_grille
```

Question 9

```
def attendreCollisionGrille(grille, tMax):
    t = 0
    while t < tMax and grille != None:
        t += 1
        grille = majGrilleOuCollision(grille)
    if t != tMax: return t # collision au temps t
```

Noter que si on n'est pas dans le cas `t!=tMax`, la fonction ne retourne rien, ce qui est la même chose en `Python` qu'un `return None`.

Question 10 Considérons la complexité de chaque fonction intervenant ici :
• La fonction `deplacerParticule` a une complexité en $O(1)$: le nombre d'opérations est le même quelles que soient les valeurs de `largeur, hauteur`.
• La fonction `nouvelleGrille` a une complexité en $O(\texttt{largeur} \times \texttt{hauteur})$: elle effectue en fait exactement `largeur` × `hauteur` fois la fonction `append`.
• La fonction `majGrilleOuCollision` fait :
— un appel à `nouvelleGrille` de complexité $O(\texttt{largeur} \times \texttt{hauteur})$,
— puis encore une double boucle à l'intérieur de laquelle les opérations effectuées sont de complexité constante, donc encore en $O(\texttt{largeur} \times \texttt{hauteur})$.

Ainsi la complexité totale de `majGrilleOuCollision` est $O(\texttt{largeur} \times \texttt{hauteur})$.
• La fonction `attendreCollisionGrille` effectue au plus `tMax` tours de boucle `while`, il y a donc au plus `tMax` appels à `majGrilleOuCollision`. Donc la complexité temporelle de `attendreCollisionGrille` est $O(\texttt{largeur} \times \texttt{hauteur} \times \texttt{tMax})$.

Question 11 Il s'agit bien de considérer la distance euclidienne entre les particules $||p_1 - p_2|| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ et pas $\max(|x_1 - x_2|, |y_1 - y_2|)$!

```
def detecterCollisionEntreParticules(p1, p2):
    x1, y1 = p1[0],p1[1]
    x2, y2= p2[0],p2[1]
    return (x1 - x2)**2 + (y1 - y2)**2 <= 4 * rayon**2
```

Question 12 Il faut bien penser à « extraire » de `particules`, passée en paramètre, ses différentes entrées, pour pouvoir travailler avec. Ensuite c'est un simple parcours de liste.

```
def maj(particules):
    largeur, hauteur, listeParticules = particules
    nouvelle_liste = []
    for particule in listeParticules:
        nouvelle_liste.append(deplacerParticule(particule, largeur, hauteur))
    return largeur, hauteur, nouvelle_liste
```

Question 13. Il faut faire attention qu'on veut detecter les collisions au temps $t + 1$. Donc il faut commencer par faire agir `maj` sur `particules`, en créant ainsi une liste `nouvelles_particules`, sur laquelle on peut chercher à détecter les collisions. Il faut aussi prendre garde de démarrer la boucle intérieure à `i+1` pour ne pas détecter une collision entre une particule et elle-même.

```
def majOuCollision(particules):
    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 None
    return nouvelles_particules
```

Question 14. Du point de vue de la complexité :
• On a déjà dit que `deplacerParticule` a une complexité en $O(1)$,
• `detecterCollisionEntreParticule` a aussi une complexité en $O(1)$: ne dépend pas de `largeur` et `hauteur`.
• `maj` a une complexité en $O(n)$ car on parcourt `listeParticules` de longueur n et à chaque étape on applique des opérations en $O(1)$.
• `majOuCollision` fait
— un appel à `maj` en $O(n)$,
— au plus (exactement si pas de collision) $n(n-1)/2$ appels à `detecterCollisionEntreParticules`, ce qui donne une complexité en $O(n^2)$.
Donc la complexité de `majOuCollision` est $O(n^2)$.

Question 15. La distance maximale entre les deux (centre des) particules lors d'une collision est $2 \times$ `rayon`. La distance maximale qu'elle peuvent parcourir pour se rapprocher l'une de l'autre en un temps de 1 et 2`vmax` $\times 1$.

Elles devront donc se situer à une distance l'une de l'autre d'au plus $2($`rayon` $+$ `vmax`$)$ à l'instant t pour avoir une chance d'entrer en collision à l'instant $t+1$.

Question 16. On commence toujours par la fabrication de la liste `nouvelles_particules` car on cherche à détecter les collisions au temps $t+1$. Pour chaque particule d'indice `i` de la nouvelle liste, on applique `detecterCollision` avec les particules d'indice `j` $>$ `i` *mais on s'arrête dès que la distance entre l'abscisse de la particule j et celle de la particule i est supérieure à* `dmax` *puisque les particules suivantes seront encore plus loin en abscisse.*

```
def majOuCollisionX(particules):
    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: # test d'arrêt qui utilise le
        # caractère ordonné suivant les abscisses.
            if detecterCollisionEntreParticules(nvllisteParticules[i], nvlListeParticules[j]):
                return None # cas de collision
            j += 1
            xj = listeParticules[j][0]
    return nouvelles_particules
```