X 2016 PSI-PT - Correction

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)
```

2)
```
def nouvelleGrille(largeur, hauteur):
    return [[None for i in range(hauteur)] for j in range(largeur)]
```

3)
```
def majGrilleOuCollision(grille):
    largeur = len(grille)
    hauteur = len(grille[0])
    res = nouvelleGrille(largeur, hauteur)
    for ligne in grille:
        for case in ligne:
            if case:
                particule = deplacerParticule(case,largeur, hauteur)
                x,y,vx,vy = particule
                i,j = int(x),int(y)
                if res[i][j]==None:
                    res[i][j] = particule
                else:
                    return None
    return res
```

4)
```
def attendreCollisionGrille(grille, tMax):
    t=0
    while t<tMax :
        t+=1
        grille = majGrilleOuCollision(grille)
        if grille==None:
            return t
    return None
```

5) Dans la fonction `attendreCollisionGrille(grille, tMax)`, la boucle `while` s'effectue au pire `tMax` fois. Les autres opérations sont en temps constant sauf l'appel à la fonction `majGrilleOuCollision(grille)`.
Dans la fonction `majGrilleOuCollision(grille)`, on a :

- d'une part l'appel à `nouvelleGrille(largeur, hauteur)` dont la complexité sera évaluée ci-dessous ;
- d'autre part deux boucles imbriquées, qui ont une complexité en $\mathcal{O}$ (`largeur` $\times$ `hauteur`), la fonction `deplacerParticule(case,largeur, hauteur)` s'effectuant en temps constant.

La fonction `nouvelleGrille(largeur, hauteur)` contient deux boucles imbriquées et s'effectue donc en $\mathcal{O}$ (`largeur` $\times$ `hauteur`).
Finalement, la complexité de la fonction `attendreCollisionGrille(grille, tMax)` est donc en $\mathcal{O}$ (`tMax` $\times$ `largeur` $\times$ `hauteur`)

6)
```
def detecterCollisionEntreParticules(p1, p2):
    x1,y1,vx,vy = p1
    x2,y2,vx,vy = p2
    if (x1-x2)**2+(y1-y2)**2<=4*rayon**2:
        return True
    return False
```

7)
```
def maj(particules):
    L,H,ps = particules
    nouvp = []
    for p in ps:
        p = deplacerParticule(p,L,H)
        nouvp.append(p)
    return (L,H,nouvp)
```

8)
```
def majOuCollision(particules):
    L,H,ps = maj(particules)
    for i in range(len(ps)-1):
        for j in range(i+1,len(ps)):
            if detecterCollisionEntreParticules(ps[i],ps[j]):
                return None
    return (L,H,ps)
```

9)
```
def attendreCollision(particules, tMax):
    t=0
    while t<tMax :
        t+=1
        particules = majOuCollision(particules)
        if particules==None:
            return t
    return None
```

Dans la fonction `attendreCollision(particules, tMax)`, la boucle `while` s'effectue au pire `tMax` fois. Les autres opérations sont en temps constant sauf l'appel à la fonction `majOuCollision(particules)`.
Dans la fonction `majOuCollision(particules)`, on a :

- d'une part l'appel à `maj(particules)` dont la complexité sera évaluée ci-dessous ;

- d'autre part deux boucles imbriquées, qui dans le pire des cas ont une complexité en $\mathcal{O}\left(\frac{n(n-1)}{2}\right) = \mathcal{O}\left(n^2\right)$ où n est le nombre de particules, la fonction `detecterCollisionEntreParti` s'effectuant en temps constant.

Dans la fonction `maj(particules)`, la boucle `for` s'effectue n fois, et à l'intérieur de cette boucle `deplacerParticule(p,L,H)` est en temps constant, et `nouvp.append(p)` est dans le pire des cas en $\mathcal{O}(n)$ mais, en moyenne, s'effectue en temps constant. On peut donc considérer que la complexité de `maj(particules)` est en $\mathcal{O}(n)$ (même s'il peut arriver ponctuellement qu'elle soit en $\mathcal{O}(n^2)$).
Finalement, la complexité de la fonction `attendreCollision(particules, tMax)` est en $\mathcal{O}(n^2 \times \texttt{tMax})$.

10) Pour que les particules entrent en collision à l'instant $t+1$, elles doivent se trouver, ***à cet instant*** $t+1$, à une distance inférieure à $2 \times$ `rayon`.
Or, elles ont, chacune, parcouru une distance au maximum égale `vMax` $\times \mathrm{d}t =$ `tMax` donc se trouvait, à l'instant t, à une distance au plus égale à $2 \times$ (`rayon+vMax`)

11)
```
def majOuCollisionX(particules):
    L,H,ps = maj(particules)
    for i in range(len(ps)-1):
        x1 = particules[2][i][0]
        for j in range(i+1,len(ps)):
            x2 = particules[2][j][0]
            if x2-x1>2*(rayon+vMax):
                break
            if detecterCollisionEntreParticules(ps[i],ps[j]):
                return None
    return (L,H,ps)
```

12)
```
def scm(s):
    d=0
    res=[]
    i=1
    while i<len(s):
        if s[i]<s[i-1]:
            res.append((d,i-1))
            d=i
        i+=1
    res.append((d,i-1))
    return res
```

13)
```
def fusionner(s,r1,r2):
    i,a = r1
    j,b = r2
    c1 = s[i]
    c2 = s[j]
    while i<=a and j<=b:
        if c2 < c1:
            s[i+1:a+2]= s[i:a+1]
            s[i] = c2
            a = a+1
            j = j+1
            i = i+1
            if j>b:
                return
            c2 = s[j]
        else:
            i = i+1
            c1 = s[i]
```

14)
```
def depileFusionneRemplace(s,pile):
    a = pile.pop()
    b = pile.pop()
    fusionner(s,b,a)
    pile.append((b[0],a[1]))
```

15)
```
def alphaTri(s):
    S = scm(s)
    pile=[]
    while S:
        pile.append(S.pop(0))
        while len(pile)>1 and len(pile[-2]))<2*len(pile[-1]):
            depileFusionneRemplace(s, pile)
    while len(pile)>1:
        depileFusionneRemplace(s, pile)
```

Erratum : voici une bonne version de la fonction :

```
def long(L):
    return L[1]-L[0]


def alphaTri(s):
    S = scm(s)
    pile=[]
    while S:
        pile.append(S.pop(0))
        while len(pile)>1 and long(pile[-2])<2*long(pile[-1]):
```

```
            depileFusionneRemplace(s, pile)
    while len(pile)>1:
        depileFusionneRemplace(s, pile)
```