I Modélisation sommaire d'un circuit

I.A Validité de la représentation d'un circuit

Q 1.

```
1 def longueur1(c:[str], d:int) -> int:
2     return c.count('A') * d
```

Q 2. `representation_minimale(["A","A","G","D","G","G","G","A"])` renvoie `['A','A','D','A']`.

La variable `nbg` représente le nombre de virages à gauche modulo 4 que l'on devra faire avant d'avancer (*e.g.* une succession de trois `"D"` sans avancer revient à un virage à gauche, soit un `"G"`).

Q 3. La fonction `representation_minimale` permet de réduire la représentation d'un circuit et faire en sorte qu'un même circuit n'ait qu'une seule représentation.

À noter toutefois que le sujet semble supposé qu'un circuit commence par une ligne droite et se termine par une ligne droite (ce qui simplifie grandement le code). Puisque un circuit, à une rotation initiale près, peut débuter par une ligne droite vers la droite on supposera que ce sera le cas. On supposera également que le dernier caractère représentera également une ligne droite (`'A'`), ce qui fait sens lorsque l'on considère un départ de Formule 1 où une vingtaine de voitures alignés sur la grille doivent pouvoir voir les feux de départ. Ainsi la longueur minimale d'un circuit convenable sera $6d$ et non $4d$.

On traitera uniquement dans les deux prochaines questions, les cas où la liste `c` ne commence pas ni se termine par un `'A'` (cas général).

Q 4. Dans une représentation minimale, il suffit de vérifier qu'entre deux `'A'`, il n'existe jamais plus de deux `'G'`.

◇ Cas général : le circuit peut commencer en allant à gauche

```
1  def contient_demi_tour1(c:[str]) -> bool:
2      res = representation_minimale(c)
3      nbg = 0
4      for i in range(len(res)) :
5          if c[i] == 'A' :
6              nbg = 0
7          elif c[i] == 'G' :
8              nbg += 1
9          if i > 1 and nbg > 1 : # si i>1, on a passé le cas du départ ['G','G',A
               ',...]
10             return True
11     return False
```

◇ Cas simplifié :

```
1  def contient_demi_tour1(c:[str]) -> bool:
2      res = representation_minimale(c)
3      nbg = 0
4      for e in res :
5          if e == 'A' :
6              nbg = 0
7          elif e == 'G' :
8              nbg += 1
9          if nbg > 1 :
10             return True
11     return False
```

Q 5. Pour toutes les questions suivantes on supposera que la liste `c` fournie en paramètre, quitte à écrire `c = representation_minimale(c)`, est la représentation minimale d'un circuit.

À l'instar de la fonction `representation_minimale`, on compte dans un tableau `avancees` les avancées dans chacune des directions. Le circuit est fermé si et seulement si toutes les avancées horizontales (resp. verticales) sont égales et que la direction est celle initiale.

◇ Cas général :

```
def est_ferme1(c:[str]) -> bool:
    avancees = [0,0,0,0] # droite, haut, gauche, bas
    n = len(c)
    nbg_initiale = 0
    nbg = 0
    i = 0
    while i < n and c[i] != 'A' :
        if c[i] == 'G' :
            nbg = (nbg + 1) % 4
        elif c[i] == 'D' :
            nbg = (nbg - 1) % 4
        i += 1
    nbg_initiale = nbg   # orientation initiale
    while i < n :
        if c[i] == 'G' :
            nbg = (nbg + 1) % 4
        elif c[i] == 'D' :
            nbg = (nbg - 1) % 4
        elif c[i] == 'A' :
            avancees[nbg] += 1
        i += 1
    return nbg == nbg_initiale and avancees[0] == avancees[2] and avancees[1]
       == avancees[3]
```

◇ Cas simplifié :

```
def est_ferme1(c:[str]) -> bool:
    avancees = [0,0,0,0] # droite, haut, gauche, bas
    nbg = 0
    for e in c :
        if e == 'G' :
            nbg = (nbg + 1) % 4
        elif e == 'D' :
            nbg = (nbg - 1) % 4
        elif e == 'A' :
            avancees[nbg] += 1
    return nbg == 0 and avancees[0] == avancees[2] and avancees[1] == avancees
        [3]
```

Dorénavant, on supposera que la liste `c` fournie en paramètre débute et se termine par `'A'` (cas simplifié).

Q 6. Puisque toutes les lignes droites sont de longueur égales à un, les croisements éventuels se font à l'une des extrémités d'une ligne droite. On modifie la fonction `est_ferme1` afin de vérifier que le circuit ne soit pas fermé avant la fin de la lecture de la liste circuit `c` (on pourrait également appelé la fonction `est_ferme1` sur chaque sous tableau `c[:i]` avec $0 < i < len(c)$).

On pouvait penser à ce code simple mais peu efficace en terme de complexité temporelle (quadratique selon `len(c)`).

```python
def circuit_convenable1(c:[str]) -> bool:
    if contient_demi_tour1(c) :
        return False
    for i in range(1, len(c)-1) :
        if est_ferme1(c[:i]) :   # circuit fermé prématurément
            return False
    return est_ferme1(c)
```

On préfèrera

```python
def circuit_convenable1(c:[str]) -> bool:
    if contient_demi_tour1(c) :
        return False
    avancees = [0,0,0,0] # droite, haut, gauche, bas
    nbg = 0
    for i in range(len(c)) :
        if c[i] == 'G' :
            nbg = (nbg + 1) % 4
        elif c[i] == 'D' :
            nbg = (nbg - 1) % 4
        elif c[i] == 'A' :
            avancees[nbg] += 1
            if i < len(c)-1 and avancees[0] == avancees[2] and avancees[1] ==
                avancees[3] : # circuit fermé prématurément
                return False
    return nbg == 0 and avancees[0] == avancees[2] and avancees[1] == avancees
        [3]
```

I.B Tracé d'un circuit

Q 7.

```python
def dessine_circuit1(c:[str], d:int) -> None:
    for e in c :
        if e == 'A' :
            turtle.forward(d)
        elif e == 'G' :
            turtle.left(90)
        elif e == 'D' :
            turtle.left(-90)
```

Rq : pour chacune des fonctions `dessine_circuit`, on peut initialiser le code avec `turtle.pendown()` et le clore avec `turtle.penup()`.

II Modélisation plus réaliste d'un circuit

II.A Validation

Q 8.

```python
def element_valide2(e : object) -> bool:
    if isinstance(e,int) :
        return e > 0
    elif isinstance(e,tuple) :
```

```
5         if len(e) != 2 :
6             return False
7         if isinstance(e[0], int) and isinstance(e[1], int) : # on ne considère
              que des angles entiers
8             return e[0] > 0 and -360 < e[1] < 360
9     return False
```

II.B Première méthode de tracé à l'écran

Q 9. On suppose que la séquence `c` ne contient que des éléments valides.

```
1  def dessine_circuit2(c:list, echelle:float) -> None:
2      for e in c :
3          if isinstance(e, int) :
4              turtle.forward(e*echelle)
5          else : # tuple
6              if e[1] < 0 :
7                  turtle.circle(-e[0]*echelle,-e[1])
8              else :
9                  turtle.circle(e[0]*echelle,e[1])
```

II.C Tracé pixel par pixel

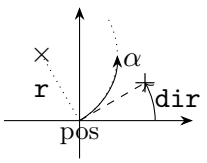
Q 10. Une matrice de rotation est de la forme $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$

```
1  def matrot(t:int) -> np.ndarray:
2      theta = t / 360 * math.pi
3      return np.array( [[math.cos(theta), -math.sin(theta)],\
4                        [math.sin(theta), math.cos(theta) ]] )
```

Q 11. Les paramètres de la fonction `cc` :

- ◇ `pos` représente la position (couple `[i,j]` du pixel considéré) et est de type `np.array`,
- ◇ `dir` représente l'angle degré de la direction de la voiture par rapport l'axe horizontal et est de type `int`,
- ◇ `r` représente le rayon de courbure et est de type `float`,
- ◇ `alpha` représente un angle degré et est de type `int`.

Ainsi la fonction `cc` calcule la position du centre du virage de rayon de courbure `r` d'angle `alpha` à partir d'une voiture à la position `pos` se dirigeant selon la direction `dir`.



Q 12. On suppose que la séquence `c` ne contient que des éléments valides.

```
1  def dessine_circuit3(s:np.ndarray, c:list, echelle:float) -> None:
2      imax, jmax = s.shape
3      pos = np.array( [imax//2, jmax//2] )  # on se place au milieu de l'écran
4      dir = 0       # angle degré, initialement mouvement vers la droite
5      for e in c :
6          if isinstance(e, int) :
7              posdbebut = pos
```

```
8            theta = dir/360*math.pi
9            pos += np.array( [math.cos(theta), math.sin(theta)] ) * e * echelle
10           ligne(s, posdebut, pos)
11       else : # tuple
12           r = e[0] * echelle
13           alpha = e[1]
14           poscentre = cc(pos, dir, r, alpha)
15           arc(s, pos, poscentre, alpha)
16           deplacement = poscentre-pos + matrot(alpha+dir) @ np.array([0.,-r*
                 sign(alpha)])
17           pos += deplacement # pos = poscentre + matrot(alpha+dir-90) @ np.
                 array([r,0])
18           dir += alpha
```

III Le parcours d'une voiture

III.A Formules de calcul des vitesses et temps de parcours

III.A.1)

Q 13. On a $\frac{d\vec{v}}{dt} = \vec{a}$, ainsi avec une accélération constante et de manière rectiligne, on obtient $v_{max} - v_0 = a_{max} t_m$ soit $t_m = \frac{v_{max} - v_0}{a_{max}} = \frac{100 - 0}{10} = 10$ s.

On remarque de suite qu'à accélération constante, on a $dt = \frac{dv}{a_{max}}$.

Q 14. On convertit préalablement les vitesses : $v_1 = 300$ km $\cdot$ h^{-1}= $\frac{500}{6}$ m $\cdot$ s^{-1}

$v_2 = 120$ km $\cdot$ h^{-1}= $\frac{100}{3}$ m $\cdot$ s^{-1}

Dans le cadre d'une accélération positive : $t_m = \frac{v_2 - v_1}{a_{max}}$

Pour une décélération, l'accélération est donnée par f_{max}, ainsi $t_m = \frac{v_2 - v_1}{f_{max}} = \frac{\frac{100}{3} - \frac{500}{6}}{-20} = 2,5$ s

Q 15. Notons x la position de la voiture, on a alors $\frac{d\vec{x}}{dt} = \vec{v}$.
Dans le cas d'un mouvement linéaire et une accélération constante positive,

$$d = x_2 - x_1 = \int_{t_1}^{t_2} v(t)dt = \int_{v_1}^{v_2} v \frac{dv}{a_{max}} = \frac{v_2^2 - v_1^2}{2\ a_{max}}$$

Q 16. Dans le cas d'une décélération, $d = \frac{v_2^2 - v_1^2}{2\ f_{max}} = \frac{200^2 - 500^2}{-40 \times 36} = \frac{7000}{48} \approx \frac{700}{5} = 140$ m (ce qui est cohérent avec $2,5$ s de décélération et une vitesse entre 83 et 33 m/s).

III.A.2)

Q 17. Selon la **Q 15.**, pour atteindre v_{max}, il faut accélérer sur une distance $d_{acc} = \frac{v_{max}^2 - v_1^2}{2\ a_{max}}$. On doit alors décélérer jusqu'à v_2 sur une distance $d_{dec} = \frac{v_2^2 - v_{max}^2}{2\ f_{max}}$.

Ainsi $d_{min} = d_{acc} + d_{dec} = \frac{v_{max}^2 - v_1^2}{2\ a_{max}} + \frac{v_2^2 - v_{max}^2}{2\ f_{max}}$

Q 18. Dans le cas où la ligne droite permet d'atteindre la vitesse maximale, il y aura trois phases : une phase d'accélération sur une distance d_{acc}, une phase à vitesse constante maximale puis une phase de décélération sur une distance d_{dec}.

Selon la question précédente, $d_{acc} = \dfrac{v_{max}^2 - v_1^2}{2\ a_{max}}$, qui requiert un temps $t_{acc} = \dfrac{v_{max} - v_1}{a_{max}}$ (**Q 14.**).

De manière similaire, pour la décélération nous avons $d_{dec} = \dfrac{v_2^2 - v_{max}^2}{2\ f_{max}}$ et $t_{dec} = \dfrac{v_2 - v_{max}}{f_{max}}$.

Il reste donc une distance $d_{vmax} = d - d_{acc} - d_{dec}$ à vitesse maximale durant $t_{vmax} = \dfrac{d_{vmax}}{v_{max}}$.

Ainsi le temps de parcours de la ligne droite est :

$$\begin{aligned} t_{min} &= t_{acc} + t_{vmax} + t_{dec} \\ &= \frac{v_{max} - v_1}{a_{max}} + \frac{d_{vmax}}{v_{max}} + \frac{v_2 - v_{max}}{f_{max}} \end{aligned}$$

Q 19. Dans le cas où $d < d_{min}$, il n'y a pas de phase à vitesse maximale.

On a alors pour la phase d'accélération $d_{acc} = \dfrac{v_3^2 - v_1^2}{2\ a_{max}}$, durant $t_{acc} = \dfrac{v_3 - v_1}{a_{max}}$ secondes. Pour la phase de décélération, $d_{dec} = \dfrac{v_2^2 - v_3^2}{2\ f_{max}}$ et $t_{dec} = \dfrac{v_2 - v_3}{f_{max}}$.

Soit un temps de parcours de : $t_{min} = t_{acc} + t_{dec} = \dfrac{v_3 - v_1}{a_{max}} + \dfrac{v_2 - v_3}{f_{max}}$

III.B Implantation en Python

III.B.1) Temps pour un tour

Q 20. Puisqu'une ligne droite débute par un virage (à moins d'être au départ), nous aurons les vitesses recommandées d'entrée et de sortie de ligne droite (la vitesse dans un virage est constante égale à celle recommandée).

Autrement dit,

- ◇ pour une droite, la vitesse maximale d'entrée possible (v_{mep}) est limitée
  - **-** par la relation $d \geqslant \dfrac{v_r^2 - v_{mep}^2}{2 f_{max}}$ (où v_r est la vitesse d'entrée du virage de fin de droite)

    soit $2 f_{max} d \leqslant v_r^2 - v_{mep}^2$, ainsi $v_{mep} \leqslant \sqrt{v_r^2 - 2 f_{max} d}$,
  - **-** et par la vitesse maximale de la voiture (v_{max}).
- ◇ pour une courbe, la vitesse étant constante, elle est donnée par le minimum entre $vr(R)$ et v_{mep} de la ligne droite suivante.

On parcourt le circuit à rebours afin de ne jamais dépasser les vitesses recommandées.

```
1  def vitesses_entree_max(c:list, vf:float) -> [float]:
2      lvmax = [vf]
3      for i in range(len(c)-1, -1, -1) :
4          if isinstance(c[i], int) :
5              lvmax.append( min( (lvmax[-1]**2-2*FMAX*c[i])**.5, VMAX) )
6          else : # virage
7              lvmax.append( min( vr( c[i][0] ), lvmax[-1]) )
8      return lvmax[:0:-1]    # on inverse en retirant vf
```

Q 21. (??) La gestion des exceptions n'est jamais mentionnée dans le BO d'IPT... Sinon on utilise les résultats de la partie **III.A**

```
1  def temps_droite(d:int, v1:float, v2:float) -> (float, float):
2      vmax = vmax_droite(d, v1, v2)          # v1 <= vmax <= max(v2,VMAX)
3      tacc = (vmax - v1) / AMAX              # accélération (nulle si vmax=v1)
4      dacc = (vmax**2 - v1**2) / (2*AMAX)
5      if vmax < v2 :                         # on accélère tout le temps
```

```
6           return ( vmax, tacc )
7       # vmax==v2 -> accélération, (palier si v2==VMAX);
8       # vmax>=v2 -> (palier si vmax==VMAX), décélération
9       tdec = (v2 - vmax) / FMAX           # décélération (nulle si vmax==v2)
10      ddec = (v2**2 - vmax**2) / (2*FMAX) # nulle si vmax==v2
11      if ddec > d :       # longueur insuffisante pour décélérer de v1 à v2
12          raise ValueError # hors programme IPT
13      # ddec < d ; il est possible d'atteindre en sécurité le virage
14      else :              # (accélération), (palier), (décélération)
15          dvmax = d - dacc - ddec
16          tvmax = 0
17          if dvmax > 0 :   # palier non nul
18              tvmax = dvmax / VMAX
19          return ( v2, tacc + tvmax + tdec )
```

Q 22. La fonction `temps_droite` fournit la vitesse de sortie de ligne droite sans dépasser la vitesse maximale possible préalablement calculée par la fonction `vitesses_entree_max`.

```
1  def temps_tour(c:list, v0:float, vf:float) -> float:
2      ttotal = 0
3      vmaxcourse = vitesses_entree_max(c, vf) + [vf]
4      v = v0
5      for i in range(len(c)) :
6          if isinstance(c[i], int) : # ligne droite
7              v, t = temps_droite(c[i], v, vmaxcourse[i+1])
8              ttotal += t
9          else :                  # virage
10             dv = c[i][0] * math.pi * abs(c[i][1]) / 180 # distance du virage
11             ttotal += dv / v  # v est constante égale à la vitesse d'entrée
12     return ttotal
```

III.B.2) Temps de course

Q 23. Il est possible de ne pas concaténer n fois le circuit, mais oblige à calculer la vitesse de fin de tour afin de débuter le prochain tour avec. Ce faisant, on réécrirait plus ou moins la fonction `temps_tour`.

```
1  def temps_course(c:list, n:int) -> float:
2      # permet d'avoir une mise à jour correcte de la vitesse de début de tour
3      return temps_tour(c*n, 0, VMAX)
```

Rq : avec le circuit de la figure 4, l'appel

- ◇ `temps_course(circuit4, 1)` renvoie $\approx 24,0s$
- ◇ `temps_course(circuit4, 2)` renvoie $\approx 46,8s$, soit $22,8s$ au tour à partir du deuxième (le deuxième tour (et les suivants) est entamé avec une vitesse d'environ 133 km/h).

Q 24. Les fonctions

- ◇ `vitesses_entree_max` possède une complexité temporelle en $\mathcal{O}(len(c))$ (l'appel `append` est en coût constant, l'inverse de la liste est également en $\mathcal{O}(len(c))$).
- ◇ `temps_droite` possède une complexité temporelle en $\mathcal{O}(1)$
- ◇ `temps_tour` possède une complexité temporelle en $\mathcal{O}(len(c))$ puisqu'il fait `len(c)` appels à `temps_droite`

Ainsi la fonction `temps_course` faisant n appels à des fonctions en $\mathcal{O}(len(c))$, la complexité temporelle dans le pire des cas de la fonction `temps_course` est $\mathcal{O}(n \cdot len(c))$.

IV Gestion des résultats

Q 25. Deux pilotes peuvent avoir le même nom (*c.f.* les frères Schumacher ou les père/fils Villeneuve).

Q 26. La table `Tour` possède entre 40 et 80 lignes par pilote (retenons 60). Il y a environ 20 pilotes à concourir et 20 circuits chaque année, ainsi la table possèdera environ 24 000 lignes pour chaque année de championnat.

Depuis 1 950, il y a eu environ 70 années de championnats, soit un total d'environ 1 700 000 lignes.

Q 27.

```
SELECT gp.gp_date, ci.ci_nom FROM Circuit AS ci
    INNER JOIN GrandPrix AS gp ON gp.ci_id = ci.ci_id
    WHERE ci_pays = 'France'
    ORDER BY gp.gp_date ;
```

Q 28.

```
SELECT ci.ci_nom, pi.pi_nom, SUM(to.to_temps) FROM Participation AS pa
    INNER JOIN Tour AS to ON to.pa_id = pa.pa_id
    INNER JOIN Pilote AS pi ON pi.pi_id = pa.pi_id
    INNER JOIN GrandPrix AS gp ON gp.gp_id = pa.gp_id
    INNER JOIN Circuit AS ci ON gp.ci_id = ci.ci_id
    GROUP BY ci.ci_nom, pi.pi.nom
    HAVING pa.pa_cla = 1
        AND EXTRACT(year FROM gp.gp_date) = 2021 ;
```

Q 29. La sous-requête enregistrée sous le nom `rdc` permet de récupérer le meilleur temps de chaque circuit associé au nom du circuit et son identifiant.

Ainsi la requête générale permet pour chacun des meilleurs temps des circuits, de récupérer le nom du circuit, le nom du pilote ayant couru, la date de la course ainsi que le dit-meilleur temps, le tout trié selon l'ordre alphabétique du nom des circuits.