Corrigé des Olympiades de NSI
Académie de Toulouse - 2023

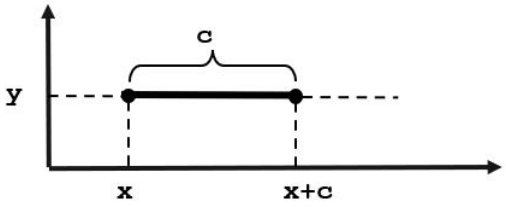
Exercice 1 : Images vectorielles pour le décor d'un jeu vidéo
Cet exercice traite des notions d'images et leur représentation à l'écran.

En informatique, une image vectorielle est une image qui est décrite par les formes géométriques simples qui la composent. Ce principe d'encodage des images permet d'éviter la déformation des images lorsqu'elles sont redimensionnées.

On cherche à étudier cette méthode de représentation sur des images simples. On précise que l'on ne travaillera que sur des images en noir et blanc.

Pour cela, on munit le plan, contenant l'image, d'un repère orthonormé permettant de positionner chaque point à l'aide de son couple de coordonnées `(x,y)` où `x` est l'abscisse du point et `y` son ordonnée. On n'utilisera que des points à coordonnées entières. On fixe l'origine du repère en bas à gauche de l'image. Ainsi, le point dans le coin en bas à gauche de l'image a pour coordonnées (0,0) et les coordonnées de tous les points sont à valeurs positives.

On définit la longueur d'un segment comme la distance entre ses extrémités. Par exemple, un segment réduit à un point est donc de longueur 0 et le segment entre les points `(x,y)` et `(x+c,y)` est de longueur `c`. Dans la suite, de telles longueurs seront représentées grâce à des accolades comme sur le dessin ci-dessous.

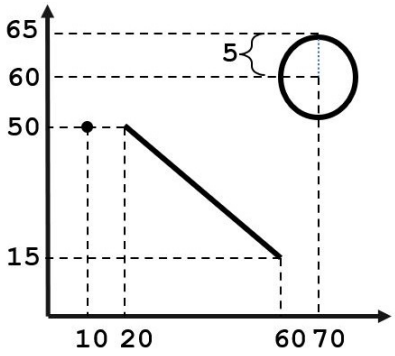


1. Manipulation de formes élémentaires

On choisit de représenter une image par des éléments tracés en noir sur un fond blanc. Par défaut l'image est donc blanche. Pour tracer les éléments noirs, on dispose des fonctions suivantes :

- `point` prenant en paramètres x et `y` deux entiers positifs et qui trace le point de coordonnées `(x,y);`
- `segment` prenant en paramètres `x1`, `y1`, `x2`, `y2` quatre entiers positifs et qui trace un segment entre le point de coordonnées `(x1,y1)` et celui de coordonnées `(x2,y2);`
- `cercle` prenant en paramètres `x`, `y` des entiers positifs et `r` un entier strictement positif et qui trace un cercle dont le centre est le point de coorodonnées`(x,y)` et de rayon `r.`

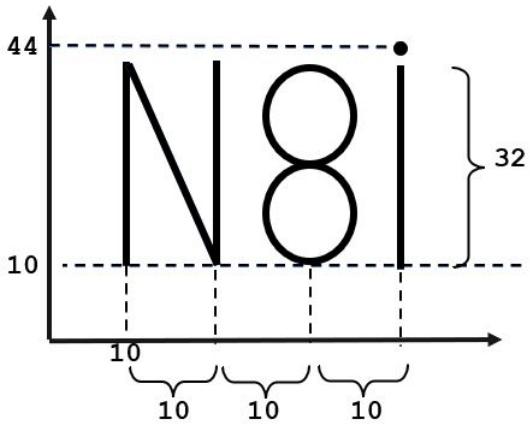
On donne ci-dessous un exemple d'image obtenue à partir d'appels aux fonctions précédentes ainsi que les instructions correspondantes.



Instructions correspondantes :

```
point(10,50)
segment(20,50,60,15)
cercle(70,60,5)
```

Question 1
En utilisant les fonctions précédentes, indiquer les instructions permettant de reproduire l'image ci-contre.



Solution

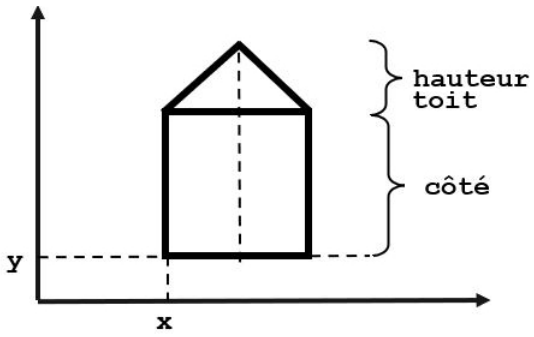
```
segment(10, 10, 10, 42)
segment(10, 42, 20, 10)
segment(20, 10, 20, 42)
cercle(30, 18, 8)
cercle(30, 34, 8)
segment(40, 10, 40, 42)
point(40, 44)
```

Question 2
Écrire le code d'une fonction `carre` prenant en paramètres `x` et `y` deux entiers positifs et `c` un entier strictement positif et qui trace un carré dont les côtés sont horizontaux ou verticaux, dont le coin inférieur gauche a pour coordonnées (`x`,`y`) et dont les côtés sont des segments de longueur `c`.

Solution

```
def carre(x, y, c):
    segment(x, y, x+c, y)
    segment(x+c, y, x+c, y+c)
    segment(x+c, y+c, x, y+c)
    segment(x, y+c, x, y)
```

On souhaite désormais tracer une maison à base carrée dont la forme est donnée ci-dessous. La hauteur de la base de la maison sera toujours égale au double de la hauteur du toit et la pointe du toit à l'aplomb au milieu de la base.



Question 3

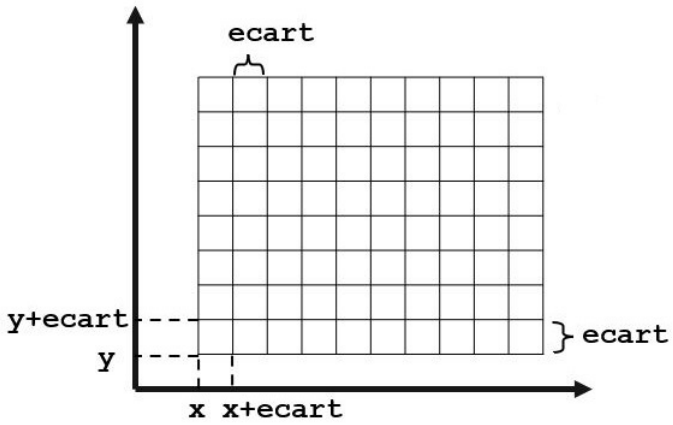
Recopier et compléter le code ci-dessous de la fonction `maison` qui prend en paramètres `x` et `y` deux entiers positifs indiquant les coordonnées du coin inférieur gauche de la maison et `h` la hauteur du toit et qui trace cette maison.

```
def maison(x, y, h):
  # à compléter
```

Solution

```
def maison(x, y, h):
  c = 2 * h
  carre(x, y, c)
  segment(x, y+c, x+h, y+c+h)
  segment(x+h, y+c+h, x+c, y+c)
```

On souhaite tracer un quadrillage, comme sur l'image ci-dessous, qui forme une grille telle que chacune de ses cases est un carré. Dans l'exemple ci-dessous, la grille comporte 8 lignes et 10 colonnes.



On crée pour cela une fonction `quadrillage` prenant en paramètres `x` et `y` deux entiers positifs représentant le point gauche inférieur du quadrillage, un entier strictement positif `ecart` donnant la hauteur des lignes et la largeur des colonnes du quadrillage, `nb_lignes` le nombre de lignes souhaitées et `nb_colonnes` le nombre de colonnes souhaitées.

Question 4

Recopier et compléter le code ci-dessous de la fonction `quadrillage`.

```
def quadrillage(x, y, ecart, nb_lignes, nb_colonnes):
  for i in range(...):
    segment(x+i*ecart,....)
  ...
  ...
```

Solution

```
def quadrillage(x, y, ecart, nb_lignes, nb_colonnes):
  ymax = y + nb_lignes*ecart
  xmax = x + nb_colonnes*ecart
  for i in range(nb_colonnes+1):
    segment(x+i*ecart, y, x+i*ecart, ymax)
  for j in range(nb_lignes+1):
    segment(x, y+j*ecart, xmax, y+j*ecart)
```

Question 5

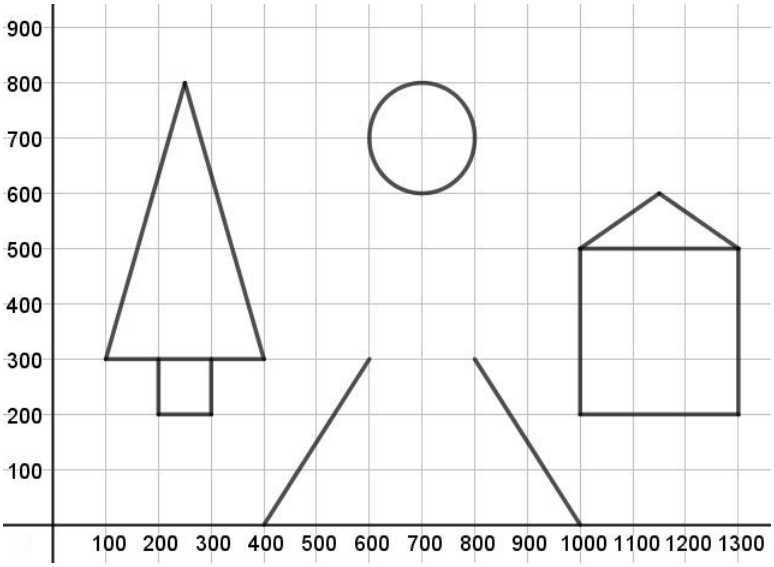
Écrire le code d'une fonction `rectangle_plein` qui prend en paramètres `x` et `y` deux entiers positifs, `larg` et `haut` deux entiers strictement positifs et qui trace un rectangle rempli en noir de largeur `larg` et de hauteur `haut` et dont le coin inférieur gauche a pour coordonnées `(x,y)`. Pour simplifier, on supposera que les côtés de ce rectangle sont horizontaux ou verticaux et que pour remplir une forme, il suffit de mettre en noir tous les points à coordonnées entières qui la composent.

Solution

```
def rectangle_plein(x, y, larg, haut):
  for i in range(haut+1):
    segment(x, y+i, x+larg, y+i)
```

2. Modélisation d'un décor fixe de jeu vidéo

On souhaite dans cette partie modéliser un décor de jeu vidéo dont un exemple est donné ci-dessous.



Ce décor, appelé par la suite "forêt" est créé de manière fixe. Il restera inchangé dans tout l'exercice.

On décompose un décor en éléments graphiques simples que sont : les points, les segments, les cercles, et les carrés correspondant aux quatre fonctions utilisées dans la partie précédente à savoir `point`, `segment`, `cercle` ainsi que `carre`. On n'utilsera pas la fonction `maison` dans la suite.

Chaque élément graphique simple d'un décor est modélisé par un tuple de la forme

```
element_decor = ("nom_fonction", param1, param2, param3, param4)
```

où `"nom_fonction"` est le nom de la fonction graphique de base utilisée sous forme de chaîne de caractères, et `param1, param2, param3` et `param4` sont les valeurs respectives des paramètres attendus pour le tracé de la forme de base.
Un tel tuple a toujours cinq éléments, soit quatre valeurs en plus du nom de la fonction. On parlera donc de quintuplet. Si la fonction graphique de base demande moins de quatre paramètres, les paramètres inutilisés, placés en dernières position du tuple, auront pour valeur `None`.

Par exemple, dans "forêt", le tronc du sapin, est modélisé par le quintuplet suivant.

```
tronc = ("carré", 200, 200, 100, None)
```

Rappel
On rappelle qu'un tuple est une structure de données ordonnée et indicée. Les éléments d'un tuple sont indiqués à l'intérieur de parenthèses, et on accède à un élément grâce à son indice. Les tuples ne sont pas modifiables.
Exemple : `t = (7,45,"a")` crée un tuple à trois éléments.
`t[1]` donne l'élément d'indice 1 de ce tuple, à savoir 45.

Question 6

a) Indiquer l'instruction permettant de créer le tuple `soleil` modélisant le soleil présent dans le décor "forêt".

Solution

```
soleil=("cercle", 700, 700, 100, None)
```

b) Indiquer de même l'instruction permettant de créer le tuple modélisant le bord gauche du sapin dans le décor "forêt".

Solution

```
bord_gauche_sapin = ("segment", 100, 300, 250, 800)
```

Un décor complet est donc modélisé par la liste de tuples réunissant tous les éléments graphiques simples nécessaires pour reconstituer l'image.

```
decor_complet = [tuple_elem_1, tuple_elem_2, tuple_elem_3, ...]
```

Question 7

a) Combien d'éléments graphiques simples sont nécessaires pour modéliser le décor "forêt" ?
Solution
10 éléments graphiques simples sont nécessaires pour modéliser le décor "forêt".

b) Écrire l'instruction permettant de créer la liste de tuples `foret` modélisant la totalité du décor "forêt".
Solution

```
foret = [bord_gauche_sapin,\
("segment",250,800,400,300),\
("segment",400,300,100,300),\
tronc,\
soleil,\
("segment",400,0,600,300),\
("segment",800,300,1000,0),\
("carre",1000,200,300,None),\
("segment",1000,500,1150,600),\
("segment",1150,600,1300,500)]
```

Une fois un décor modélisé à travers ses différents éléments, il faut dessiner ces éléments. On crée pour cela une fonction `dessine_element` qui prend en paramètre un quintuplet modélisant un élément graphique simple et qui exécute la fonction de tracé associée.

Question 8

Recopier et compléter la fonction `dessine_element` ci-dessous.

```
def dessine_element(elmt):
    if elmt[0]=="segment":
        segment(elmt[1],elmt[2],elmt[3],elmt[4])
    elif ....
    # plusieurs lignes à compléter

```

Solution

```
def dessine_element(elmt):
    if elmt[0]=="segment":
        segment(elmt[1],elmt[2],elmt[3],elmt[4])
    elif elmt[0]=="point":
        point(elmt[1],elmt[2])
    elif elmt[0]=="cercle":
        cercle(elmt[1],elmt[2],elmt[3])
    elif elmt[0]=="carre":
        carre(elmt[1],elmt[2],elmt[3])
```

Enfin, pour dessiner un décor en entier, il faut appeler la fonction `dessine_element` sur chaque élément du décor.

Question 9
Écrire la fonction `dessine_decor` prenant en paramètre `decor_complet` une liste de quintuplets représentant le décor à dessiner et dessinant le décor en entier.

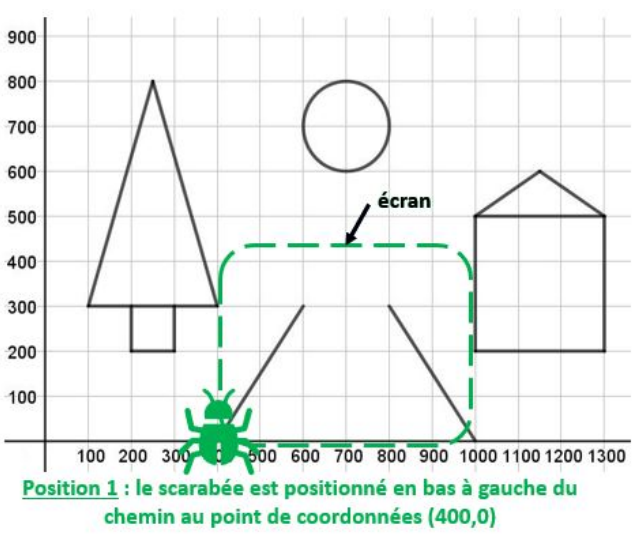
Solution

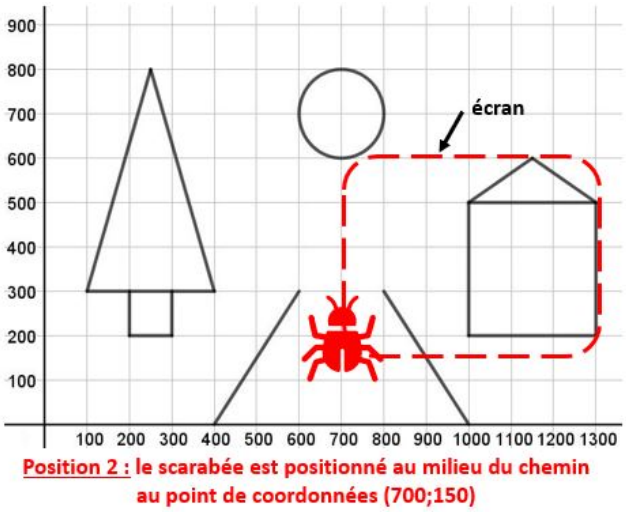
```
def dessine_decor(decor_complet):
    for element in decor_complet:
        dessine_element(element)
```

3. Animation d'un personnage dans le décor

On souhaite désormais faire évoluer un personnage de scarabée dans notre décor de jeu vidéo. On suppose que la taille de l'écran d'affichage réel du jeu vidéo est plus petite que la taille totale du décor. On ne peut donc afficher à l'écran qu'une partie du décor. On simule le déplacement du scarabée en n'affichant à l'écran qu'une partie du décor et en plaçant virtuellement le scarabée en bas à gauche de l'écran (le scarabée ne sera pas représenté à l'écran).

L'image ci-dessous illustre la situation :





On suppose dans un premier temps qu'appeler une fonction traçant une forme partiellement en dehors de la zone de l'écran ne pose pas de problème : seule la partie qui est dans la zone de l'écran est affichée. De même si on appelle une fonction traçant une forme totalement en dehors de la zone de l'écran, cette forme n'est pas affichée.
De plus, du fait du décalage de la zone de l'écran, il n'est plus possible de faire appel directement à la fonction `dessine_element`. En effet,

- dans le cas où le scarabée est en position 1, `dessine_element(chemin1)` par exemple appellerait `segment(400,0,600,300)` alors qu'il faut appeler `segment(0,0,200,300)` puisque l'origine du repère de l'écran est décalée par rapport à celle du décor,
- de même, dans le cas où le scarabée est en position 2, `dessine_element(base_maison)` appelerait `carre(1000,200,300)` alors qu'il faut appeler `carre(300,50,300)`.

Il faut donc opérer un décalage sur les coordonnées avant d'appeler les fonctions `dessine_element` sur chaque élément.

Question 10
Sur chacun des exemples précédents, quel est le lien entre les coordonnées du scarabée et le décalage réalisé ?

Solution
Position 1 :
Pour le scarabée en (400,0) l'appel décalé est par exemple `segment(0,0,200,300)` soit `segment(400-400,0-0,600-400,300-0)`.
Les coordonnées du scarabée ont été soustraites pour réaliser le décalage.

Position 2 :
Pour le scarabée en (700,150) l'appel décalé est `carre(300,50,300)` soit `carre(1000-700,200-150,300)`.
Les coordonnées du scarabée ont été à nouveau soustraites pour réaliser le décalage.

Question 11
De manière générale, compléter la fonction suivante qui affiche le décor de manière décalée en adéquation avec la position du scarabée dont les coordonées sont notées `xS` et `yS`.

```
def affiche_ecran(decor_complet, xS, yS):
  # à compléter
```

Solution

```
def affiche_ecran(decor_complet, xS, yS):
  for elmt in decor_complet :
    (nom, a1, a2, a3, a4) = elmt
    if nom == "segment":
      dessine_element(nom, a1-xS, a2-yS, a3-xS, a4-yS)
    else:
      dessine_element(nom, a1-xS, a2-yS, a3, a4)
```

Dans les deux prochaines questions, on souhaite simuler les déplacements du scarabée dans le décor.
On dispose pour cela de deux fonctions `pause` et `efface` permettant de mettre en œuvre des animations d'images. Ces fonctions ne prennent pas de paramètre et sont telles que :

- l'appel `pause()` a pour effet que le programme attende quelques millisecondes avant d'exécuter l'instruction suivante ;
- l'appel `efface()` a pour effet d'effacer tous les tracés noirs de l'image, qui devient donc blanche.

Ainsi, la suite d'instructions ci-dessous réalise par exemple l'animation d'images permettant de simuler le déplacement horizontal d'un carré de deux points vers la droite.

```
carre(100,200,10)
pause()
efface()
carre(101,200,10)
pause()
efface()
carre(102,200,10)
```

Question 12

On souhaite simuler un déplacement horizontal point par point du scarabée dans le décor "forêt". Pour cela, on décale au fur et à mesure la zone du décor qui est affichée à l'écran. Écrire les instructions nécessaires à l'animation d'un tel déplacement du point `(400,0)` au point `(600,0)`.

Solution

```
for i in range(201):
    affiche_ecran(foret,400+i,0)
    pause()
    efface()
```

Question 13

Écrire une fonction `deplacement_vertical` qui prend en paramètres `decor_entier` une liste de quintuplets représentant le décor, `x` et `y` deux entiers représentant la position initiale du scarabée et `h` un entier strictement positif et qui simule le déplacement vertical entre les points de coordonnées `(x,y)` et `(x,y+h)`. À la fin de l'appel de la fonction le décor devra être visible.

Solution

```
def deplacement_vertical(decor_entier, x, y, h):
    affiche_ecran(decor_entier,x,y)
    for j in range(1,h+1):
        pause()
        efface()
        affiche_ecran(decor_entier,x,y+j)
```

4. Gestion des formes hors zone d'affichage

Dans cette partie, nous allons travailler sur les formes qui sortent de la zone d'affichage de l'écran afin de s'assurer qu'elles ne soient effectivement pas affichées lors de l'exécution des fonctions vues précédemment, comme cela avait été admis dans la partie 3.

Un écran est caractérisé par la position de son coin inférieur gauche dans le décor donnée par un couple `(x,y)` de coordonnées entières positives, par sa longueur et sa hauteur qui sont deux entiers strictement positifs. On représente ainsi un écran par un dictionnaire qui rassemble ces trois données en les associant respectivement aux clés `"position"`, `"largE"` et `"haut"`.

Exemple : `ecran1 = {"position":(0,0), "largE":600, "hautE":450}`

Question 14

Écrire une fonction `est_dans_ecran_point` qui prend en paramètres `x` et `y` deux entiers positifs représentant la position d'un point et `ecran` un dictionnaire représentant un écran comme expliqué précédemment et qui renvoie `True` si le point de coordonnées `(x,y)` est dans la zone de l'écran, `False` sinon.

Solution

```
def est_dans_ecran_point(x, y, ecran):
  return x >= ecran["position"][0] and \
  x <= ecran["position"][0] + ecran["largE"] and \
  y >= ecran["position"][1] and \
  y <= ecran["position"][1] + ecran["hautE"]
```

Dans un premier temps, on cherche à écrire une fonction `est_dans_ecran_element` qui prend en paramètres un quintuplet `element` représentant un point, un segment ou un carré et un dictionnaire `ecran` comme décrit précédemment. Cette fonction doit renvoyer `True` si l'élément graphique est **intégralement** dans l'écran, `False` sinon.

Question 15

a) En supposant la fonction `est_dans_ecran_element` créée, quelle est la valeur de la variable `verif1` après l'exécution des instructions suivantes ?

```
elmt1 = ("segment", 100, 200, 500, 400)
ecran1 = {"position":(0,0), "largE":600, "hautE":450}
verif1 = est_dans_ecran_element(elmt1,ecran1)
```

Solution

`True`

b) De même, quelle est la valeur de la variable `verif2` après l'exécution des instructions suivantes ?

```
elmt2 = ("carre", 500, 300, 250)
ecran2 = {"position":(100,250), "largE":600, "hautE":450}
verif2 = est_dans_ecran_element(elmt2,ecran2)
```

Solution

`False`

c) Écrire le code de la fonction `est_dans_ecran_element` en traitant uniquement le cas des formes `point`, `segment` et `carre`.

Solution

```
def est_dans_ecran_element(elt, ecran):
  if elt[0]=="point":
    return est_dans_ecran_point(elt[1],elt[2],ecran)
  if elt[0]=="segment":
    verif1 = est_dans_ecran_point(elt[1],elt[2],ecran)
    verif2 = est_dans_ecran_point(elt[3],elt[4],ecran)
    return verif1 and verif2
  if elt[0]=="carre":
    verif1 = est_dans_ecran_point(elt[1],elt[2],ecran)
    verif2 = est_dans_ecran_point(elt[1]+elt[3],elt[2]+elt[3],ecran)
    return verif1 and verif2
```

Question 16

On souhaite désormais intégrer cette fonction dans la fonction `affiche_ecran` de la question 11,

de manière à afficher à l'écran uniquement les formes **intégralement** dans l'écran. Expliquer les modifications à apporter pour cela à la fonction `affiche_ecran`. On ne traitera pas dans cette question le problème des formes partiellement hors écran.

Solution
Rajouter un test sur les éléments du décor pour que les lignes 3 à 7 de la fonction `affiche_ecran` ne soient exécutées que sur les éléments qui sont intégralement dans l'écran, `(xS, yS)` correspondant à la valeur associée à la clé `"position"` du dictionnaire représentant l'écran utilisé.

Question 17
Comment faire en sorte qu'une forme partiellement hors écran soit en partie tracée pour sa partie dans l'écran et non tracée pour sapartie hors écran ? *Dans cette question ouverte, il vous est demandé d'expliquer avec clarté et précision votre démarche. Toute tentative de raisonnement cohérent même partiel sera valorisée.*

Solution
Question ouverte... Réponse ouverte :)

Exercice 2 : Course à pieds sur un circuit

Dans le corrigé de cet exercice on utilise des annotations de type du module `Typing` pour préciser le type des paramètres et du résultat des fonctions.
Ainsi `def maFonction(n:int, l:List[float], c:str) -> Tuple[int, float]:` signifie que la fonction `maFonction` prend quatre paramètres, le premier (`n`) est un entier, le deuxième (`l`) une liste de nombres à virgule flottante, le troisième (`c`) une chaîne de caractères et qu'elle renvoie un couple dont le premier élément est un entier et le deuxième un nombre à virgule flottante.
Pour utiliser de telles annotations il faut une version de Python supérieure à 3.5 et importer le module (en pratique `from typing import List, Tuple, Dict` est souvent suffisant).
La documentation de ce module est disponible sur https://docs.python.org/3/library/typing.html.
Sans entrer dans les détails, on peut retenir que les noms des types (hors types de base) prennent une majuscule, et que leurs paramètres sont toujours entre crochets et séparés par des virgules (même pour les tuples et les dictionnaires, ça n'a rien à voir avec la syntaxe de l'objet Python).
On peut définir des nouveaux types avec la syntaxe `NouveauType = expression_de_type`.
On peut aussi définir des variables de type, mais cette possibilité n'est pas illustrée dans cet exercice.
Ces annotations de types ont d'abord un rôle pédagogique : elles encouragent à écrire un code propre et préparent les élèves à utiliser des langages de programmations typés. Pour vérifier la cohérence des annotations de types de manière automatisée il faut utiliser un outil externe, car elles sont ignorées par Python : un code mal typé tourne sans erreur s'il est par ailleurs correct. On peut par exemple utiliser MyPy, dont la documentation est disponible sur https://mypy-lang.org/.

Pendant la pandémie de 2020, différents marathons ont été annulés. Plusieurs ont été remplacés par des courses, toujours à pieds, sur des circuits automobiles. C'est ce qui s'est passé dans la ville d'Albi en 2020.

Dans cet exercice, on va d'abord travailler avec une modélisation particulière d'un circuit. On va ensuite expliquer ce qu'on appelle la validité d'un circuit et écrire une fonction qui vérifie si un circuit est valide ou non.

Dans tout cet exercice on suppose que les bibliothèques `math` et `turtle` sont rendues accessibles grâce à l'instruction : `import math, turtle`

1. Représentation d'un circuit

Pour simplifier le problème, nous considérons un circuit constitué uniquement de segments de droite et de virages à angle droit.
Un tel circuit est représenté par une liste de chaînes de caractères dont les éléments peuvent être seulement `'A'`, `'G'` ou `'D'` et où :

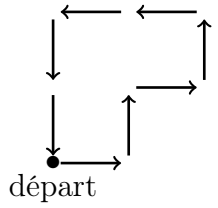
- `'A'` représente une portion de ligne droite de longueur déterminée d,
- `'G'` représente un virage à 90° à gauche et
- `'D'` représente un virage à 90° à droite.

Pour simplifier l'écriture et **pour toute la suite du sujet**, cette liste ne contiendra pas les sous-listes `['G', 'D']`, `['D', 'G']`, `['D', 'D', 'D']`, `['G', 'G', 'G']`.

En effet, les sous-listes `['G', 'D']`, tourner à droite puis à gauche en restant sur place, ou `['D', 'G']`, tourner à droite puis à gauche en restant sur place, ne modifient pas le circuit.
Exemple : `['A', 'A', 'D', 'G', 'G']` et `['A', 'A', 'G']` correspondent au même circuit.

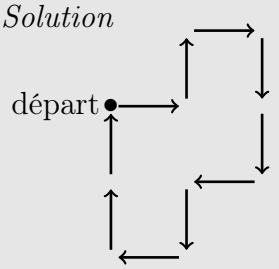
De même les sous-listes `['D', 'D', 'D']`, tourner trois fois consécutivement à droite en restant sur place, revient à tourner à gauche et peut être remplacé par la sous-liste `['G']` sans changer le circuit, tandis que `['G', 'G', 'G']` , tourner trois fois consécutivement à gauche en restant sur place, revient à tourner à droite et peut être remplacé par la sous-liste `['D']` sans changer le circuit.
Exemple : `['A', 'D', 'D', 'D', 'A']` et `['A', 'G', 'A']` correspondent au même circuit.

De plus, pour représenter un circuit, on suppose que **l'orientation au départ est toujours vers la droite**. Ainsi le circuit ci-contre est représenté par la liste suivante :
`['A', 'G', 'A', 'D', 'A', 'G', 'A', 'G', 'A', 'A', 'G', 'A', 'A']`

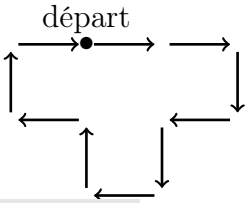


Question 1
Tracer le circuit correspondant à la liste suivante.
`['A', 'G', 'A', 'D', 'A', 'D', 'A', 'A', 'D', 'A', 'G', 'A', 'D', 'A', 'D', 'A', 'A']`

Solution



Question 2
À quelle liste correspond le circuit ci-contre ?



Solution
`['A', 'A', 'D', 'A', 'D', 'A', 'G', 'A', 'D', 'A', 'D', 'A', 'G', 'A', 'D', 'A', 'D', 'A']`

2. Longueur du parcours

Question 3
Écrire une fonction `longueur(c, d)` qui prend en paramètres `c` une liste représentant un circuit et `d` la longueur en mètres d'une portion de ligne droite et qui renvoie la longueur totale du circuit en mètres.
Solution

```
from typing import List, Tuple, Dict

Circuit = List[str]

def longueur(c:Circuit, d:float) -> float :
    """ hypothèses : d >= 0
        retourne la longueur totale de c pour des tronçons de longueur d
    """
    dist = 0.
    for el in c :
        if el == 'A' :
            dist = dist + d
    return dist
```

Le parcours est constitué d'un ou plusieurs tours de circuit, suivant la distance souhaitée par les organisateurs et la longueur du circuit. On souhaite calculer le nombre de tours pour faire une course d'une longueur donnée. Le résultat sera le nombre de tours effectués entièrement. Par exemple pour faire 10km sur un circuit de 3km et 600m, il faut faire 2 tours entiers.

Question 4

Écrire une fonction `nb_tours(c, d, d_totale)` qui prend en paramètres `c` une liste non vide représentant un circuit, `d` la longueur non nulle en mètres d'une portion de ligne droite et `d_totale` la distance totale de la course en mètres et qui renvoie le nombre de tours entiers à effectuer.

Solution

```
def nb_tours(c:Circuit, d:float, d_totale:float) -> float :
    """ hypothèses : c != [] and d > 0 and d_totale >= 0
        retourne le nombre de tours entiers de c à réaliser pour
        parcourir une distance d_totale
    """
    return d_totale // longueur(c, d)
```

3. Dessiner un circuit

Rappel

La bibliothèque `turtle` simule une « tortue » robot qui se déplace sur l'écran. Une trace de ces déplacements est laissée sur l'écran. Au début du programme, la tortue est située au centre de l'écran et est orientée vers la droite. On dispose alors des trois fonctions suivantes pour déplacer la tortue :

- `turtle.forward(n)` fait avancer la tortue de `n` pixels.
- `turtle.right(a)` fait pivoter la tortue sur la droite de `a` degrés.
- `turtle.left(a)` fait pivoter la tortue sur la gauche de `a` degrés.

Par exemple le dessin ci-dessous est le résultat du code ci-contre.

```
import turtle
turtle.forward(40)
turtle.left(90)
turtle.forward(20)
turtle.right(90)
turtle.forward(20)
```

Question 5

En utilisant la bibliothèque `turtle`, écrire une fonction `tracer_circuit(c, d)` qui prend en paramètres `c` une liste représentant un circuit, `d` la longueur en pixels d'une portion de ligne droite et qui dessine le circuit `c`.

Solution

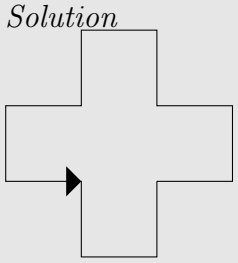
```
import turtle
def tracer_circuit(c:Circuit, d:int) -> None :
    """ hypothèses : d >= 0
        trace c dans une fenêtre turtle pour des tronçons d pixels
    """
    for el in c :
        if el == 'A' :
            turtle.forward(d)
        elif el == 'G' :
            turtle.left(90)
        else :
            turtle.right(90)
```

Question 6

En prenant pour échelle 10 pixels = 1 centimètre, dessiner la trace de la tortue lorsqu'on exécute le code suivant.

```
for k in range(4):
    tracer_circuit(['D', 'A', 'G', 'A', 'G', 'A'], 20)
```

Solution



4. Virages et demi-tours

Lorsqu'on tourne deux fois à gauche, ou deux fois à droite, sans avancer entre les deux virages, on revient sur ses pas. Ainsi, dans une liste représentant un circuit, on appellera *demi-tour* deux occurrences consécutives de `'G'`, ou deux occurrences consécutives de `'D'`. Par exemple la liste `['G','A','G','G','A','D','D','A','A','D']` présente deux demi-tours, le premier à l'indice 2, le second à l'indice 5.

Question 7

Écrire une fonction `detection_demi_tour(c)` qui prend en paramètre `c` une liste représentant un circuit et qui renvoie `True` si le circuit a un demi-tour et `False` sinon.

Solution

```
def detection_demi_tour(c:Circuit) -> bool :
  """ teste si le circuit c présente des demi-tours
  """
  for k in range(len(c) - 1) :
   if (c[k]=='G' and c[k+1]=='G') or (c[k]=='D' and c[k+1]=='D') :
     return True
  return False
```

Question 8

Écrire une fonction `distance_1_demi_tour(c, d)` qui prend en paramètres `c` une liste représentant un circuit et `d` la longueur en mètres d'une portion de ligne droite et qui renvoie la distance en mètres entre le début du circuit et le premier demi-tour. S'il n'y a pas de demi-tour, la fonction renvoie -1.

Solution

```
def distance_1_demi_tour(c:Circuit, d:float) -> float :
  """ renvoie à quelle distance du départ est le premier demi-tour
      d c (pour des tronçons de longueur d) s'il y en a, -1 sinon
  """
  dist = 0.
  demi_tour = False
  k = 0
  while not demi_tour and k < len(c) - 1:
      if c[k] == 'A' :
          dist = dist + d
      elif (c[k] == 'G' and c[k+1] == 'G') \
      or (c[k] == 'D' and c[k+1] == 'D'):
          demi_tour = True
      k = k + 1
  if demi_tour :
      return dist
  else:
      return -1
```

Question 9
Écrire une fonction `sans_demi_tours(c)` qui prend en paramètre une liste `c` représentant un circuit et qui renvoie la liste `c` sans ses demi-tours.

Solution

```
def sans_demi_tours(c:Circuit) -> Circuit :
    """
    renvoie le cicruit c privé de ses demi-tours
    """
    res = []
    k = 0
    while k < len(c)-1 :
        if (c[k]+c[k+1] != 'GG') and (c[k]+c[k+1] != 'DD') :
            res.append(c[k])
            k = k + 1
        else :
            k = k + 2
    if k == len(c)-1 :
        res.append(c[-1])
    return res
```

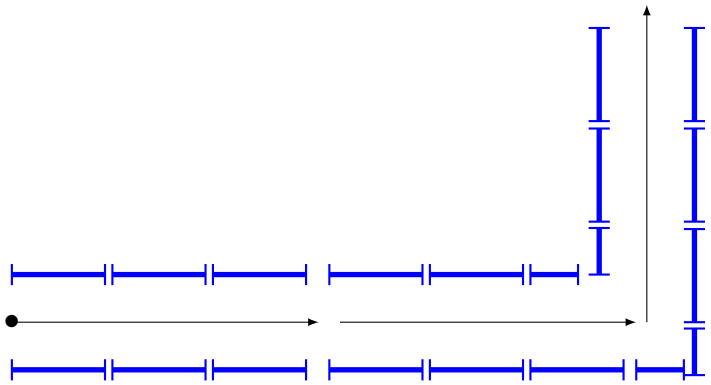
Dans toute la suite du sujet, on suppose que le circuit n'a pas de demi-tour.

Question 10
On sait que pour faciliter la course, il est important de minimiser le nombre de virages.
Écrire une fonction `nb_virages(c)` qui prend en paramètre `c` une liste représentant un circuit et qui renvoie le nombre de virages qu'il y a dans ce circuit.

Solution

```
def nb_virages(c:Circuit) -> int :
    """
    renvoie le nombre de virages dans c
    """
    nb = 0
    for el in c :
        if (el == 'G' or el == 'D') :
            nb = nb + 1
    return nb
```

On cherche maintenant à déterminer le nombre de barrières nécessaires pour sécuriser le parcours. Chaque portion de ligne droite nécessite 3 grandes barrières de chaque côté, à moins qu'elle ne soit suivie ou précédée d'un virage, auquel cas il faut plutôt des petites barrières pour gérer le coin, comme on peut le voir sur la figure ci-dessous qui illustre les barrières sécurisant le circuit représenté par la liste `['A','A','G','A']`.



Question 11
Définir une fonction `nb_barrieres(c)` qui prend en paramètre `c` une liste représentant un circuit et qui calcule les nombres de barrières nécessaires pour sécuriser ce circuit. Cette fonction renverra un couple (p, g) où p est le nombre de petites barrières, et g le nombre de grandes barrières.
Pour simplifier, on suppose que le circuit ne termine pas par un virage, et ne revient jamais sur lui-même, de sorte qu'il n'y pas de croisement ni de jonction à prendre en compte.

Solution

```
def nb_barrieres(c:Circuit) -> Tuple[int,int] :
    """
    renvoie les nombres de petites et de grandes barrières qu'il faut
    pour encadrer le circuit c
    """
    nbv = nb_virages(c) #nombre de virages
    nbd = len(c) - nbv  #nombre de portions de ligne droite
    return (4*nbv,6*nbd-2*nbv)
```

5. Validité du circuit

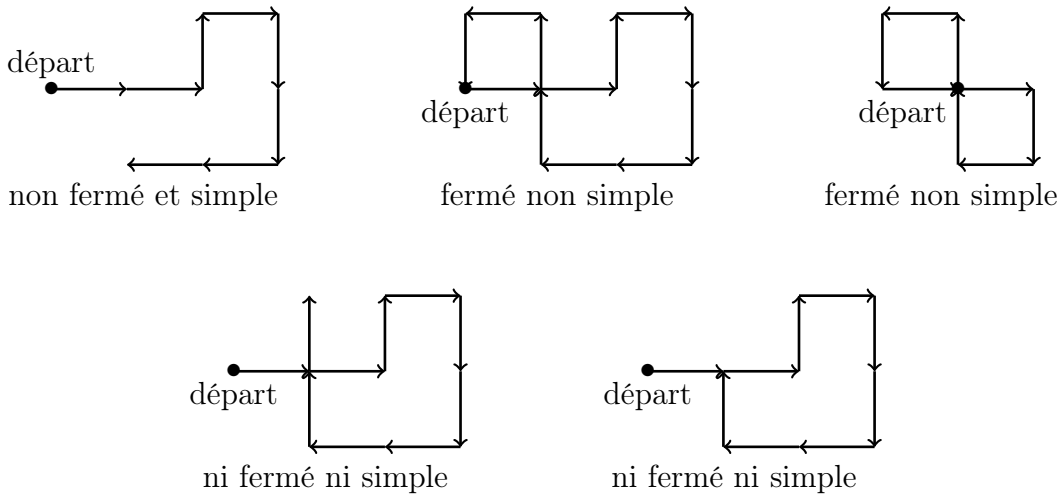
Dans la suite du sujet, on suppose que la longueur d'un tronçon de ligne droite vaut 1.

Le but de cette partie est de tester si un circuit est valide.

Validité. Pour être **valide**, un circuit doit vérifier deux conditions :

- être **fermé**, c'est-à-dire que le point d'arrivée du circuit est le même que celui de départ (c'est d'ailleurs ce qui justifie qu'on parle de circuit et non de chemin) ;
- être **simple**, c'est-à-dire qu'il ne s'auto-intersecte pas en dehors du point de départ et d'arrivée, autrement dit lors du parcours du circuit, le seul moment où il est permis de revenir sur ses pas est la fin du parcours, et dans ce cas le point sur lequel on revient doit être le point de départ.

On donne ci-après 5 exemples de circuits non valides, les circuits tracés dans les sections 1 à 4 de cet exercice en revanche étaient tous valides.



États courants. Pour tester la validité d'un circuit, il suffit de connaître la position courante à chaque étape du parcours (disons qu'une étape correspond à une lettre de la liste représentant le circuit). En revanche pour connaître la position à une étape, il ne suffit pas de connaître la position à l'étape précédente et la lettre lue, il faut aussi connaître l'orientation à l'étape précédente.
En effet, la lecture d'un `'A'` indique qu'il faut avancer d'une unité de longueur, mais pas dans quelle direction, car celle-ci dépend de l'orientation au point de départ et de tous les virages réalisés depuis...

On appelle donc **état** la donnée de deux informations :

- la **position**, qui dit où on est ;
- l'**orientation**, qui donne la direction dans laquelle on s'apprête à aller.

Puisque ce qui nous importe ici est la forme du circuit et non pas son emplacement géographique réel, on utilisera seulement des coordonnées cartésiennes dans un repère orthonormé centré sur le point de départ. Ainsi la position initiale a pour coordonnées $(0, 0)$.
De plus, comme la longueur d'un tronçon de ligne droite est d'une unité, les positions atteintes au fil d'un circuit sont toujours à coordonnées entières, ainsi **une position est représentée par un couple d'entiers**.
Ce repère étant fixé, les 4 orientations possibles sont représentées par les 4 vecteurs suivants :

- `(0,1)` pour $\uparrow$;
- `(0,-1)` pour $\downarrow$;
- `(1,0)` pour $\rightarrow$;
- `(-1,0)` pour $\leftarrow$.

Ainsi **une orientation est représentée par l'un des 4 couples :** `(0,1)`, `(0,-1)`, `(1,0)`, `(-1,0)` .

On fixe arbitrairement l'orientation initiale vers la droite. Ainsi l'état initial est le suivant : (position : $(0, 0)$, orientation : $(1, 0)$).

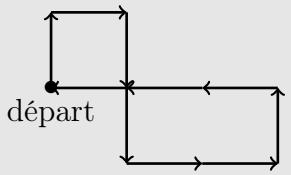
Un exemple. On donne ci-après les états successivement atteints lors du parcours du circuit représenté par : `['G','A','D','A','D','A','A','G','A','A','G','A','G','A','A','A']`.

Dernière lettre lue	Position	Orientation
-	$(0, 0)$	$(1, 0)$
`'G'`	$(0, 0)$	$(0, 1)$
`'A'`	$(0, 1)$	$(0, 1)$
`'D'`	$(0, 1)$	$(1, 0)$
`'A'`	$(1, 1)$	$(1, 0)$
`'D'`	$(1, 1)$	$(0, -1)$
`'A'`	$(1, 0)$	$(0, -1)$
`'A'`	$(1, -1)$	$(0, -1)$
`'G'`	$(1, -1)$	$(1, 0)$
`'A'`	$(2, -1)$	$(1, 0)$
`'A'`	$(3, -1)$	$(1, 0)$
`'G'`	$(3, -1)$	$(0, 1)$
`'A'`	$(3, 0)$	$(0, 1)$
`'G'`	$(3, 0)$	$(-1, 0)$
`'A'`	$(2, 0)$	$(-1, 0)$
`'A'`	$(1, 0)$	$(-1, 0)$
`'A'`	$(0, 0)$	$(-1, 0)$

Question 12
Dessiner le circuit considéré dans l'exemple précédent. Ce circuit est-il valide ? Justifier.

Solution

Ce circuit n'est pas valide car comme on peut le voir sur le dessin ci-contre il s'auto-intersecte en un point différent du départ et de l'arrivée.



Question 13
Donner les états successivement atteints lors du parcours du circuit représenté par la liste suivante. `['A','A','D','A','D','A','G']`.

Solution

Dernière lettre lue	Position	Orientation
-	$(0,0)$	$(1,0)$
`'A'`	$(1,0)$	$(1,0)$
`'A'`	$(2,0)$	$(1,0)$
`'D'`	$(2,0)$	$(0,-1)$
`'A'`	$(2,-1)$	$(0,-1)$
`'D'`	$(2,-1)$	$(-1,0)$
`'A'`	$(1,-1)$	$(-1,0)$
`'G'`	$(1,-1)$	$(0,-1)$

L'état initial étant fixé, les lettres du circuit peuvent être vues comme des instructions qui modifient l'état courant

- `'A'` indique qu'il faut avancer d'une unité dans la direction donnée par l'orientation courante, c'est-à-dire modifier la position courante,
- `'G'` (resp. `'D'`) indique qu'il faut changer de direction, c'est-à-dire modifier l'orientation courante.

On remarque d'ailleurs dans l'exemple ci-dessus qu'à chaque étape, c'est soit la position qui est modifiée, soit l'orientation, jamais les deux simultanément. On va donc créer deux fonctions différentes, celle qui gère les modifications de la position, appelée `avance` et définie ci-dessous, et une autre qui gère les modifications de l'orientation.

```
def avance (pos, ori):
    """ hypothèse : ori est un vecteur unité
    retourne la position courante après la lecture d'un 'A'
    depuis l'état défini par la position pos et l'orientation ori
    """
    x1,y1 = pos
    x2,y2 = ori
    return (x1+x2, y1+y2)
```

Question 14
Que renvoie `avance((1,1),(0,1))` ?
Que renvoie `avance((2,1),(-1,0))` ?

Solution
(1, 2)
(1, 1)

Question 15

Écrire une fonction `tourne(ori, lettre)` qui prend en paramètres une orientation `ori` et un caractère `lettre` qui peut être `'G'` ou `'D'`, et qui renvoie l'orientation après la lecture de cette lettre.

Solution 1

```
Orientation = Tuple[int,int]

def tourne (ori:Orientation, lettre:str) -> Orientation :
  """ hypothèse : lettre =='G' or lettre =='D'
                  ori est une orientation valide
      retourne l'orientation courante après la lecture de lettre
      depuis un état d'orientation ori
  """
  if lettre =='G':
    if ori == (0,1): #haut
      return (-1,0) #gauche
    elif ori == (0,-1): #bas
      return (1,0) #droite
    elif ori == (-1,0): #gauche
      return (0,-1) #bas
    elif ori == (1,0): #droite
      return (0,1) #haut
  elif lettre =='D':
    if ori == (-1,0): #gauche
      return (0,1)#haut
    elif ori == (1,0): #droite
      return (0,-1)#bas
    elif ori == (0,-1): #bas
      return (-1,0)#gauche
    elif ori == (0,1): #haut
      return (1,0)#droite
```

Solution 2

```
def tourne (ori:Orientation, lettre:str) -> Orientation :
    if lettre == 'G':
        return (-ori[1], ori[0])
    else:
        return (ori[1], -ori[0])
```

Question 16

Écrire une fonction `etat_suivant(pos, ori, lettre)` qui prend en paramètres une position `pos`, une orientation `ori` et un caractère `lettre` qui peut être `'A'`, `'G'` ou `'D'`, et qui donne la position et l'orientation après la lecture de la lettre.

Solution

```
1
2  Position = Tuple[int,int]
3  Etat = Tuple[Position,Orientation]
4
5  def etat_suivant(pos:Position, ori:Orientation, lettre:str) -> Etat :
6    """ hypothèse : lettre =='G' or lettre =='D' or lettre =='A'
7        retourne l'état obtenu en lisant lettre depuis l'état (pos, ori)
8    """
9    if lettre == 'A' :
10     nv_pos = avance(pos,ori)
11     nv_ori = ori
12   else :
13     nv_pos = pos
14     nv_ori = tourne(ori,lettre)
15   return (nv_pos,nv_ori)
```

Question 17

Écrire une fonction `est_ferme(c)` qui prend en paramètre `c` une liste représentant un circuit et qui renvoie `True` si le circuit est fermé et `False` sinon.

Solution

```
1
2  def est_ferme(c:Circuit) -> bool :
3    """ hypothèse : les éléments de c sont 'A', 'D' ou 'G'
4        teste si le point d'arrivée de c est le point de départ
5    """
6    position = (0,0)
7    orientation = (1,0)
8    for el in c:
9      position,orientation= etat_suivant (position,orientation,el)
10   return position == (0,0)
```

Question 18

Écrire une fonction `est_simple(c)` qui prend en paramètre `c` une liste représentant un circuit et qui renvoie `True` si le circuit est simple et `False` sinon.

Solution

```
def est_simple(c:Circuit) -> bool :
  """ hypothèse : les éléments de c sont 'A', 'D' ou 'G'
      teste si le circuit est sans auto-intersection
      (version de complexité pire cas en O(len(c)^2))
  """
  if len(c) == 0 :
    return True
  position = (0,0)
  orientation = (1,0)
  liste_des_positions = [position]
  trouve_intersection = False
  i = 0
  while not trouve_intersection and i < len(c)-1:
      position,orientation = etat_suivant (position,orientation,c[i])
      if position in liste_des_positions :
          trouve_intersection = True
      liste_des_positions.append(position)
      i = i + 1
  return not trouve_intersection
```

Question 19

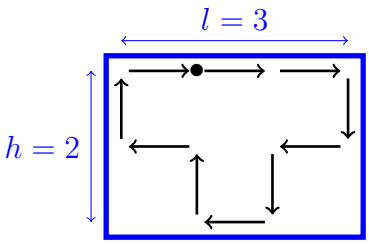
Écrire une fonction `est_valide(c)` qui prend en paramètre `c` une liste représentant un circuit et qui renvoie `True` si le circuit est valide et `False` sinon.

Solution

```

def est_valide(c:Circuit) -> bool :
  """ hypothèse : les éléments de c sont 'A', 'D' ou 'G'
      teste si le circuit c est valide
  """
  return est_ferme(c) and est_simple(c)
```

Dans la question suivante, on souhaite implanter un circuit dont la forme est déjà fixée. On veut alors trouver les dimensions minimales d'un terrain sur lequel cela est possible.
Pour simplifier le problème, on suppose que le terrain est rectangulaire et que le circuit sera placé de sorte que les axes du repère sous-jacent soient parallèles aux côtés du rectangles.
Autrement dit on attend un couple (l, h) où l est la *largeur* du circuit et h sa *hauteur*.
On donne ci-dessous un exemple du plus petit rectangle pouvant contenir le circuit présenté à la question 2.



Question 20

Écrire une fonction `dimensions(c)` qui prend en paramètre `c` une liste représentant un circuit et qui calcule les dimensions (largeur puis hauteur) nécessaires pour implanter ce circuit dans un espace rectangulaire.

Solution

```
1
2  def dimensions(c:Circuit) -> Tuple[int,int] :
3    """ hypothèse : les éléments de c sont 'A', 'D' ou 'G'
4        renvoie les dimensions minimales d'un rectangle contenant c
5    """
6    x_min = 0
7    x_max = 0
8    y_min = 0
9    y_max = 0
10   position = (0, 0)
11   orientation = (1, 0)
12   for el in c:
13     position, orientation = etat_suivant(position, orientation, el)
14     x,y = position
15     if x > x_max :
16       x_max = x
17     elif x < x_min :
18       x_min = x
19     if y > y_max :
20       y_max = y
21     elif y < y_min :
22       y_min = y
23   return (x_max - x_min), (y_max - y_min)
```