I Jeu de Snort

Le jeu de Snort se joue sur un graphe non orienté G avec deux joueurs 1 et 2 : à tour de rôle, chaque joueur colorie un sommet de leur choix, avec la contrainte qu'un sommet de couleur 1 ne doit pas être voisin d'un sommet de couleur 2. Un joueur ne pouvant pas colorer de sommet a perdu.

1. Écrire une fonction `coups(G, couleurs, j)` qui renvoie la liste des sommets que le joueur `j` peut colorier où :
   - `G` est une liste d'adjacence représentant le graphe (`G[i]` est la liste des voisins du sommet i),
   - `couleurs` est un tableau de taille n où `couleurs[i]` est la couleur du sommet i (0 si non colorié, 1 si colorié par le joueur 1, 2 si colorié par le joueur 2),
   - `j` est un entier représentant le joueur courant.

   Solution :

```
def coups(G, couleurs, j):
    L = []
    for u in range(len(G)):
        if couleurs[u] == 0:
            b = True
            for v in G[u]:
                if couleurs[v] == 3 - j:
                    b = False
            if b:
                L.append(i)
    return L
```

2. On considère une stratégie gloutonne consistant à choisir le sommet de degré maximum parmis ceux possibles. Écrire une fonction `glouton(G, couleurs, j)` qui renvoie le sommet à colorier selon cette stratégie. Si aucun sommet n'est possible, la fonction renvoie -1.

   Solution :

```
def glouton(G, couleurs, j):
    L = coups(G, couleurs, j)
    if L == []:
        return -1
    else:
        max = 0
        for i in range(1, len(L)):
            if len(G[L[i]]) > len(G[L[max]]):
                max = i
        return L[max]
```

3. Écrire une fonction `joueur(couleurs)` qui renvoie le joueur qui doit jouer, sachant que le joueur 1 joue en premier.

   Solution :

```
def joueur(couleurs):
    n = 0
    for c in couleurs:
        if c != 0:
            n += 1
    return n % 2 + 1
```

4. On code une liste `couleurs` $= [c_0, c_1, \ldots, c_{n-1}]$ par un entier $s = \sum_{i=0}^{n-1} c_i 3^i$. Écrire une fonction `encode(couleurs)` qui renvoie l'entier s correspondant.

   Solution :

```
def encode(couleurs):
    s = 0
    for i in range(len(couleurs)):
        s += couleurs[i] * 3 ** i
    return s
```

5. Écrire une fonction `decode(s, n)` effectuant l'opération inverse de `encode`.

Solution :

```
def decode(s, n):
    couleurs = []
    for i in range(n):
        couleurs.append(s % 3)
        s //= 3
    return couleurs
```

6. On souhaite déterminer l'attracteur A du joueur 1. On rappelle que'une situation de jeu appartient à A si on est dans l'un des cas suivants :

   - c'est au tour du joueur 2 et il ne peut pas jouer,
   - c'est au tour du joueur 1 et il peut jouer pour aboutir à une situation de A,
   - c'est au tour du joueur 2 et tous les coups possibles mènent à une situation de A.

   Écrire une fonction `attracteur(G, couleurs)` qui renvoie l'attracteur du joueur 1, sous forme de liste.
   On pourra utiliser un dictionnaire dont chaque clé correspond à un coloriage partiel de G et la valeur associée est un booléen indiquant s'il appartient à l'attracteur du joueur 1. Comme une liste ne peut pas être une clé de dictionnaire, on utilisera `encode` pour obtenir une clé entière.

Solution :

```python
def attracteur(G, couleurs):
    d = {}
    def aux(couleurs): # détermine si couleurs est dans l'attracteur du joueur 1
        s = encode(couleurs)
        if s in d:
            return d[s]
        j = joueur(couleurs)
        L = coups(G, couleurs, j)
        if L == []:
            d[s] = j == 2
        elif j == 1:
            b = False
            for u in L:
                couleurs[u] = j
                if aux(couleurs):
                    b = True
                couleurs[u] = 0
            d[s] = b
        else:
            b = True
            for u in L:
                couleurs[u] = j
                if not aux(couleurs):
                    b = False
                couleurs[u] = 0
            d[s] = b
        return d[s]
    aux(couleurs)
    A = []
    for s in d:
        if d[s]:
            A.append(decode(s, len(couleurs)))
    return A
```

Mines informatique 2023 : La typographie informatisée

Arnaud Bégyn PC Déodat de Severac Toulouse

Partie I – Préambule

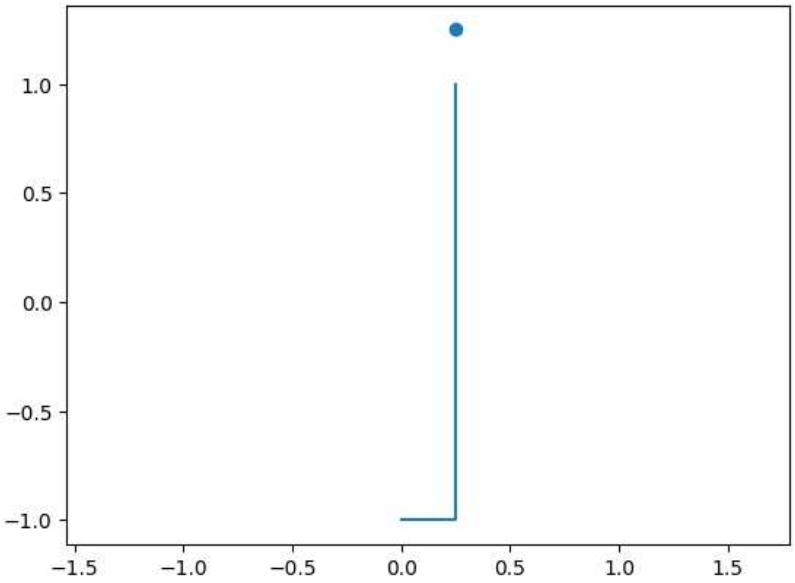
Q1. $\overline{100}^{16} = 16^2 = 256$ cents = 2.56 dollars.

Q2. Il fallait faire le dessin sur sa copie. Ici on utilise Python et on visualise la lettre j.

In [1]:
```
import matplotlib.pyplot as plt

plt.plot([0.25,0.25,0.0],[1.0,-1.0,-1.0])
plt.axis('equal')
plt.scatter([0.25],[1.25])
```

Out[1]:
```
<matplotlib.collections.PathCollection at 0x7fd611ab3f40>
```



Partie II – Gestion de polices de caractères vectorielles

Q3. Requête en SQL pour compter le nombre de glyphes en roman:

In [ ]:
```sql
SELECT COUNT(*)
FROM Glyphe
WHERE groman = True;
```

Q4. Requête en SQL afin d'extraire la description vectorielle du caractère A dans la police nommée Helvetica en italique:

In [ ]:
```sql
SELECT G.gdesc
FROM Glyphe AS G
JOIN Police AS P
JOIN Caractere AS C
ON G.pid = P.pid AND C.code = G.code
WHERE C.car = 'A' AND P.pnom = 'Helvetica' AND G.groman = False;
```

Q5. Requête en SQL pour extraire les noms des familles qui disposent de polices et leur nombre de polices, classés par ordre alphabétique:

In [ ]:
```sql
SELECT F.fnom, COUNT(pid)
FROM Famille AS F
JOIN Police AS P
ON F.fid = P.fid
GROUP BY P.fid
HAVING COUNT(pid) <>0
ORDER BY F.fnom;
```

Partie III – Manipulation de descriptions vectorielles de glyphes

Q6. Fonction utilitaire `points(v:[[[float]]])->[float]` qui renvoie la liste des points qui apparaissent dans les multi-lignes de la description vectorielle `v` d'un glyphe.

In [2]:
```python
def points(v:[[[float]]])->[float] :
    liste = []
    for ligne in v :
        for point in ligne :
            liste.append(point)
    return liste


# test unitaire
v = [ [ [ 0, 0 ], [ 1, 1 ] ], [ [ 0, 1 ], [ 1, 0 ] ] ]
print( points(v) == [[0, 0], [1, 1], [0, 1], [1, 0]] )
```

```
True
```

Q7. Fonction utilitaire qui renvoie la liste des éléments d'indice `n` des sous listes de flottants:

In [3]:
```python
def dim(l:[[float]], n:int)->[float] :
    liste = []
    for point in l :
        liste.append(point[n])
    return liste


l = [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ], [ 7, 8 ] ]
print( dim(l, 1) == [ 2, 4, 6, 8 ] )
```

```
True
```

Q8. Fonction qui renvoie la largeur de la description vectorielle `v` :

In [ ]:
```python
def largeur(v:[[[float]]])->float :
    liste = points(v)
    liste2 = dim(liste, 0)
    return max(liste2)-min(liste2)
```

Q9. Fonction qui renvoie une liste de largeurs pour toutes les lettres minuscules romanes et italiques de la police `police` dans l'ordre a roman, a italique, b roman, b italique...

In [ ]:
```python
def  obtention_largeur(police:str)->[float] :
    liste = []
    for lettre in 'abcdefghijklmnopqrstuvwxyz' :
        v = glyphe(lettre, police, True)
        liste.append(largeur(v))
        v = glyphe(lettre, police, False)
        liste.append(largeur(v))
    return liste
```

Q10. Fonction utilitaire qui prend en paramètres une fonction `f`, une description vectorielle `v` et qui renvoie une nouvelle description vectorielle construite à partir de `v` en appliquant la fonction `f` à chacun des points et en préservant la structure des multi-lignes:

In [ ]:
```
def transforme(f:callable, v:[[[float]]])->[[[float]]] :
    liste = []
    for ligne in v :
        liste.append( [ f(point) for point in ligne] )
    return liste
```

Q11. Toutes les abscisses des points des multi-lignes sont divisées par 2. Le glyphe est donc déformé horizontalement de telle sorte que sa largeur soit divisée par 2.

Q12. Fonction qui renvoie une nouvelle description vectorielle correspondant à un glyphe penché vers la droite:

In [ ]:
```
def penche(v:[[[float]]])->[[[float]]] :
    def zzz((p:[float])->[float] :
            return [ p[0] + 0.5 * p[1], p[1] ]
    return transforme(zzz, v)
```

Partie IV – Rasterisation

In [6]:
```
from PIL import Image

im = Image.new("1", (50, 100), color=1)

for y in range(60, 65):
    for x in range(5, 45):
        im.putpixel((x, y), 0)
        im.putpixel((x, y-20), 0)

im.save("egal.png")
```

In [ ]:
```python
from math import floor     # renvoie l'entier immédiatement inférieur

def trace_quadrant_est(im, p0:(int), p1:(int)) :
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1-x0, y1-y0
    im.putpixel(p0, 0)
    for i in range(1, dx):
        p = (x0 + i, y0 + floor(0.5 + dy * i / dx))
        im.putpixel(p, 0)
    im.putpixel(p1, 0)


im = Image.new("1", (10, 10), color=1)
trace_quadrant_est(im, (0, 0), (6, 2))
trace_quadrant_est(im, (9, 8), (1, 9))
trace_quadrant_est(im, (3, 0), (5, 8))
im.show()
```

Q13. Ligne 6: `dx` prend la valeur 6 et `dy` prend la valeur 2.
La ligne 7 encre le pixel (0, 0).
Ligne 8: `i` varie de 1 à 5.
Ensuite on répète la ligne 10: on encre les pixels (1, 0) (2, 1) (3, 1) (4, 1) (5, 2)
Ligne 11: on encre le pixel (6, 2).

Q14. Ligne 6: `dx` prend la valeur -8 et `dy` prend la valeur 1.
La ligne 7 encre le pixel (9, 8).
Les lignes 8, 9 et 10 ne sont pas exécutées car **dx $\leq$ 1.**

Ligne 11: on encre le pixel (1, 9).
Le problème vient du fait que `dx < 0`. Il faudrait ajouter en début de fontion `assert dx >= 0`.

Q15. Ligne 6: `dx` prend la valeur 2 et `dy` prend la valeur 8.
La ligne 7 encre le pixel (3, 0).
Ligne 8: `i` varie de 1 à 1.
Ensuite on répète la ligne 10: on encre les pixels (4, 4).
Ligne 11: on encre le pixel (5, 8).
Les pixels ne sont pas adjacents car `dx < dy`.

Q16. On inverse le rôle de `dx` et `dy` :

In [ ]:
```
def trace_quadrant_sud(im, p0:(int), p1:(int)) :
    assert dy >= 0
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1-x0, y1-y0
    im.putpixel(p0, 0)
    for i in range(1, dy):
        p = (x0 + floor(0.5 + dx * i / dy), y0 + i)
        im.putpixel(p, 0)
    im.putpixel(p1, 0)


im = Image.new("1", (10, 10), color=1)
trace_quadrant_sud(im, (3, 0), (5, 8))
im.show()
```

Q17. Fonction qui trace un segment continu entre les pixels `p0` et `p1` sur l'image `im` :

In [ ]:
```
def trace_segment(im:Image, p0:(int), p1:(int)) :
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1-x0, y1-y0
    if abs(dx)>=abs(dy) :   # si on va vers l'est ou l'ouest
        if dx < 0 :  # on se ramène au cas où on va vers l'est
            p0, p1 = p1, p0
```

```
            x0, y0 = p0
            x1, y1 = p1
            dx, dy = x1-x0, y1-y0
        im.putpixel(p0, 0)
        for i in range(1, dx):
            p = (x0 + i, y0 + floor(0.5 + dy * i / dx))
            im.putpixel(p, 0)
    else :  # on va vers le nord ou le sud
        if dy < 0 :  # on se ramène au cas où on va vers le sud
            p0, p1 = p1, p0
            x0, y0 = p0
            x1, y1 = p1
            dx, dy = x1-x0, y1-y0
        im.putpixel(p0, 0)
        for i in range(1, dy):
            p = (x0 + floor(0.5 + dx * i / dy), y0 + i)
            im.putpixel(p, 0)
    im.putpixel(p1, 0)


im = Image.new("1", (10, 10), color=1)
trace_segment(im, (0, 0), (6, 2))
trace_segment(im, (9, 8), (1, 9))
trace_segment(im, (3, 0), (5, 8))
trace_segment(im, (3, 0), (3, 0))
im.show()
```

Partie V – Affichage de texte

Q18. Fonction qui renvoie les coordonnées du point `p` (point d'un glyphe de coordonnées flottantes) en un point dans une page (pixel de coordonnées entières) de manière à ce que le point (0, 0) de la description vectorielle soit en position `pz` sur la page, et que l'oeil de taille normalisée 1 du glyphe fasse `taille` pixels de hauteur:

In [ ]:
```
from math import floor

def position(p:(float), pz:(int), taille:int)->(int) :
```

```
    x, y = p
    xz, yz = pz
    x2 = floor(x*taille) + xz
    y2 = -floor(y*taille) + yz
    return (x2, y2)
```

Q19. Fonction qui affiche dans l'image `page` le caractère `c` dans la police `police`, en roman ou italique selon la valeur du booléen `roman`, et renvoie la largeur en pixel du glyphe affiché :

In [ ]:
```
def affiche_car(page:Image, c:str, police:str, roman:bool, pz:(int), taille:int)->int :
    v = glyphe(c, police, roman)
    for ligne in v :
        [x ,y] = ligne[0]
        x, y = position((x, y), pz, taille)
        trace_segment(page, (x, y), (x, y) ) # cas où la ligne n'a qu'un point
        for k in range(len(ligne)-1) :
            x1, y1 = ligne[k]
            x1, y1 = position((x1, y1), pz, taille)
            x2, y2 = ligne[k+1]
            x2, y2 = position((x2, y2), pz, taille)
            trace_segment(page, (x1, y1), (x2, y2) )
        return taille*largeur(v)
```

Q20. Fonction qui affiche la chaîne de caractères `mot` dans les mêmes conditions, chaque glyphe étant séparé du suivant par `ic` pixels, et renvoie la position du dernier pixel de la dernière lettre dans la page :

In [ ]:
```
def affiche_mot(page:Image, mot:str, ic:int, police:str, roman:bool, pz:(int), taille:int)->int :
    x, y = pz
    positionPixel = x
    for k in range(len(mot)) :
        c = mot[k]
        largeur = affiche_car(page, c, police, roman, [positionPixel, y], taille)
        positionPixel += largeur + ic
    return (positionPixel - ic, y)
```

Partie VI – Justification d'un paragraphe

Q21. L'algorithme ajoute des mots à la ligne tant que c'est possible. Si l'ajout d'un mot provoque un dépassement alors il est placé sur la ligne suivante. Il est **glouton** car il se contente de vérifier localement si l'ajout d'un mot est possible sur la ligne courante.

Q22.

- Découpage a): coût total de `32`.
  - 3 mots de `i=0` à `j=2` donc `coût = (10-(2-0)-(2+4+2))**2 = 0`.
  - 1 mot de `i=3` à `j=3` donc `coût = (10-(3-3)-6)**2 = 16`.
  - 1 mot de `i=4` à `j=4` donc `coût = (10-(4-4)-6)**2 = 16`.

- Découpage b): coût total de `26`.
  - 2 mots de `i=0` à `j=1` donc `coût = (10-(1-0)-(2+4))**2 = 9`.
  - 2 mots de `i=2` à `j=3` donc `coût = (10-(3-2)-(2+6))**2 = 1`.
  - 1 mot de `i=4` à `j=4` donc `coût = (10-(4-4)-6)**2 = 16`.

C'est donc l'algorithme dynamique qui donne la solution la plus harmonieuse.

Q23.

In [ ]:
```python
#variable globale
memo = { len(lmots) : 0 }   # coquille dans l'énoncé: m au lieu de lmots

def progd_memo(i:int, lmots:[int], L:int, memo:{int:int})
    if i not in memo :
        mini = float("inf")
        for j in range(i+1, len(lmots)+1) :
            d = progd_memo(j, lmots, L) + cout(i, j-1, lmots, L)
            if d < mini :
                mini = d
        memo[i] = mini
    return memo[i]
```

Q24. On note C_n la complexité temporelle pour n mots.

- Pour l'algorithme récursif naïf on a << à la louche >> que $C_n = \sum_{j=1}^{n} \left(C_{n-j} + j^2 \right)$ donc $C_n \geq C_0 + C_1 + \cdots + C_{n-1} = 2C_{n-1}$. Par récurrence on a donc $C_n \geq 2^n C_0$. Donc la complexité temporelle est au minimum exponentielle.
- Pour l'agorithme de programmation dynamique de bas en haut on a

$$C_n = \sum_{i=0}^{n-2} \left(\sum_{j=i+1}^{n} (j - i + 3) \right) = \sum_{i=0}^{n-2} \left(\sum_{j=4}^{n-i+3} j \right) = \sum_{i=0}^{n-2} \frac{(n-i+3)(n-i+4)}{2} = O(n^3)$$

L'algorithme de programmation dynamique de bas en haut est donc bien plus performant.

Q25.

In [4]:
```python
def lignes(mots:[str], t:[int], L:int)->[[str]] :
    i = 0
    liste = []
    while i<len(t) :
        ligne = mots[i:t[i]]
        liste.append(ligne)
        i = t[i] # on a déjà placé les mots[k] pour k allant de i à t[i]-1
    return liste

# Test unitaire
print( lignes(["Ut","enim","ad","minima","veniam"],[2,3,4,4,5],10)
       == [["Ut","enim"],["ad","minima"],["veniam"]] )
```

```
True
```

Q26.

In [5]:
```python
def formatage(lignesdemots : [[str]], L : int)->str :
    texte = ""
    for ligne in lignesdemots :
        lmots = 0 # calcul la longueur de tous les mots de la ligne
        for mot in ligne :
            lmots = lmots + len(mot)
```

```
        nbEspace = L - lmots  # le nombre d'espaces à insérer
        if len(ligne) == 1 :
            ligneTexte = ligne[0] + " "*nbEspace + "\n"
        else :
            nbEspaceMini = nbEspace // (len(ligne)-1) # le nombre minimum d'espace à placer à chaque fois
            EspacesSup = nbEspace % (len(ligne)-1) # le rab d'espaces
            # on construit un liste contenant le nombre d'espace à insérer entre les mots
            LesEspaces = [nbEspaceMini+1]*EspacesSup + [nbEspaceMini]*(len(ligne)-1-EspacesSup)
            # construction de la ligne
            ligneTexte = ""
            for i in range(len(ligne)-1) :
                ligneTexte = ligneTexte + ligne[i] + " "*LesEspaces[i]
            ligneTexte = ligneTexte + ligne[len(ligne)-1] + "\n"
        # completion du texte
        texte = ligneTexte + texte
    return texte

print( formatage([["Ut","enim"],["ad","minima"],["veniam"]],10) )
```

```
veniam
ad  minima
Ut    enim
```

In [ ]: