DS 3 : lundi 5 janvier

Correction

Mines 2023 - MP-PC-PSI

Partie I – Préambule

Q1. En base 16, 100 correspond à 16^2 en base 10, ie à 256 cents, ie à 2.56 dollars.

Q2. Le glyphe (ne pas oublier de le dessiner) correspond à un 'j'

Partie II — Gestion de polices de caractères vectorielles

Q3.
```
SELECT COUNT()
FROM Glyphe
WHERE groman=True
```

Q4. Une solution possible, on peut joindre la table Caractere (entre Caractere.code et Glyphe.code) plutôt que de faire une sous-requête.

```
SELECT gdesc
FROM Glyphe
    JOIN Police ON Glyphe.pid=Police.pid
WHERE code = (SELECT code FROM Caractere WHERE car="A") AND groman=False AND pnom="Helvetica"
```

Q5. Inutile de demander nbr>0, en effet les familles qui n'ont pas de police n'apparaissent pas à la jointure

```
SELECT fnom, COUNT() AS nbr
FROM Famille
    JOIN Police ON Police.fid=Famille.fid
GROUP BY fid
ORDER BY fnom
```

Partie III — Manipulation de descriptions vectorielles de glyphes

Q6. Si un point est présent dans plusieurs multi-lignes il apparaitra plusieurs fois (rajouter un `if pt not in res` avant le rajout si on ne le désire pas)

```
def points(v:[[[float]]])->[[float]]:
    res=[]
    for mligne in v:
        for pt in mligne:
            res.append(pt)
    return res
```

Q7.
```
def dim(l:[[float]], n:int)->[float]:
    res=[]
    for elt in l:
        res.append(elt[n])
    return res
```

Q8.
```
def largeur(v:[[[float]]])->float:
    abs=dim(points(v),0) # La liste des abscisses des points
    return max(abs)-min(abs)
```

Q9.
```
def obtention_largeur(police:str)->[float]:
    res=[]
    for lettre in "abcdefghijklmnopqrstuvwxyz":
        res.append(largeur(glyphe(lettre,police,True)))
        res.append(largeur(glyphe(lettre,police,False)))
    return res
```

Q10.
```
def transforme(f:callable, v:[[[float]]])->[[[float]]]:
    res=[]
    for mligne in v:
        newmligne=[f(pt) for pt in mligne]
        res.append(newmligne)
    return res
```

Q11. Toutes les abscisses sont divisées par deux, cella correspond à déformer horizontalement le glyphe en le réduisant de moitié.

Q12.
```
def penche(v:[[[float]]])->[[[float]]]:
    def f(pt:[float])->[float]:
        x,y=pt
        return [x+0.5*y,y]
    return transforme(f,v)
```

Partie IV — Rasterisation

Q13. La ligne 15 doit tracer le segment reliant $(0,0)$ à $(6,2)$, ainsi $dx=6$ et $dy=2$ donc $\frac{dy}{dx}=\frac{1}{3}$, le premier pixel encré est $(0,0)$ le second est $(1,0+\lfloor 0.5+\frac{1}{3}\rfloor)=(1,0)$, le suivant est $(2,0+\lfloor 0.5+\frac{2}{3}\rfloor)=(2,1)$, etc. La liste des pixels encrés est donc $(0,0)$, $(1,0)$, $(2,1)$, $(3,1)$, $(4,1)$, $(5,2)$ et $(6,2)$.

Q14. La ligne 16 doit tracer le segment reliant $(9,8)$ à $(1,9)$, ainsi $dx=-8$ et $dy=1$, ainsi on ne rentre pas dans la boucle, les pixels encrés sont donc uniquement les deux pixels qui le sont à l'extérieur de la boucle, ie $(9,8)$ et $(1,9)$. il faudrait donc mettre un : `assert dx>0`

Q15. La ligne 17 doit tracer le segment reliant $(3,0)$ et $(5,8)$, ainsi $dx=2$ et $dy=8$, les pixels encrés sont donc $(3,0)$, $(4,4)$ et $(5,8)$, le tracé ne ressemble pas vraiment à un segment (les pixels ne se touchent pas), cela proviens du fait que $\frac{dy}{dx}>1$

Q16. Pour régler ce dernier problème, il suffit d'inverser les rôles de dx et de dy, et donc de tracer le segment de "haut en bas" (mettre éventuellement un `assert dy>0`).

```
def trace_quadrant_sud(im:Image, 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, dy):
        p = (x0 + floor(0.5 + dx * i / dy), y0 + i)
        im.putpixel(p, 0)
    im.putpixel(p1, 0)
```

Q17.
```
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): # segment plutôt vertical
        if dy<0:
            trace_quadrant_sud(im,p1,p0)
        else:
            trace_quadrant_sud(im,p0,p1)
    else: #segment plutôt horizontal
        if dx<0:
            trace_quadrant_est(im,p1,p0)
        else:
            trace_quadrant_est(im,p0,p1)
```

Partie V — Affichage de texte

Q18. J'ai considéré que `p` correspondait à des coordonnées tel que donnés dans la partie I (donc avec l'axe vertical orienté vers le haut), que `pz` était un point de l'image (donc l'axe vertical orienté vers le bas).

```
def position(p:(float), pz:(int), taille:int)->(int):
    x,y=p
    x0,y0=pz
    xn=x0+floor(taille*x)
    yn=y0-floor(taille*y)
    return [xn,yn]
```

Q19.
```
def affiche_car(page:Image, c:str, police:str, roman:bool, pz:(int), taille:int)->int:
    v=glyphe(c,police,roman)
    for mligne in v: # pour chaque multiligne
        p0=mligne[0]
        p0n=position(p0,pz,taille)
        if len(mligne)==1:
            trace_segment(page,p0n,p0n)
        else:
            for i in range(1,len(mligne)):
                p0=mligne[i-1]
                p0n=position(p0,pz,taille)
                p1=mligne[i]
                p1n=position(p1,pz,taille)
                trace_segment(page,p0n,p1n)
    return taille*largeur(v)
```

Q20.
```
def affiche_mot(page:Image, mot:str, ic:int, police:str, roman:bool, pz:(int), taille:int)->int:
    x0,y0=pz
    pz0=[x0,y0]
    for c in mot:
        dpz=affiche_car(page,c,police,roman,pz0,taille)
        x0 += dpz + ic
        pz0=[x0,y0]
    return [x0-ic,y0]
```

Partie VI — Justification d'un paragraphe

Q21. L'algorithme rajoute des mots à la ligne tant que ce rajout ne provoque pas un dépassement de la largeur maximal, si l'ajout provoque un dépassement alors on met le mot dans une nouvelle ligne et on continue.
C'est un algorithme glouton car il essai de mettre un maximum de mot dans chaque ligne sans considérer la ligne suivante (optimisation local), dans certains cas il peut être préférable de ne pas rajouter un mot dans la ligne (alors qu'on le peut) pour pouvoir mieux remplir les lignes suivantes.

Q22. Pour le premier découpage, la première ligne coute $cout(0,2)$, la deuxième coute $cout(3,3)$ et la troisième coute $cout(4,4)$, soit un total de $(10-(2-0)-(2+4+6))^2+(10-(3-3)-6)^2+(10-(4-4)-6)^2 = 0+16+16 = 32$.
Pour le deuxième découpage, la première ligne coute $cout(0,1)$, la deuxième coute $cout(2,3)$ et la troisième coute $cout(4,4)$, soit un total de $(10-(1-0)-(2+4))^2+(10-(3-2)-(2+6))^2+(10-(4-4)-6)^2 = 9+1+16 = 26$.
Ainsi la solution par programmation dynamique donne une solution plus harmonieuse.

Q23.
```
memo={len(m):0} # len(lmots) aurait été plus parlant IMHO
def progd_memo(i:int,lmots:[int],L:int,memo:{int:int}):
    if i in memo: # déjà calculé
        return memo[i]
    # sinon (else inutile à cause du return) on calcul
    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 mini
```

Q24. On remarque que $cout(i,j)$ a une complexité de l'ordre de $j-i+1$

Pour l'algorithme récursif naïf, en notant C_i la complexité de l'appel avec i, on a $C_i = \sum_{j=i+1}^{n}(C_j+j-i) = \frac{(n-i)(n-i+1)}{2} + \sum_{j=i+1}^{n} C_j$. Ainsi si $C_n = 1$ alors $C_{n-1} = C_n+1 = 2$ et $C_{n-2} = C_{n-1}+1+C_n+2 = 8$, par récurrence directe on a donc que pour tout i on a $C_{i-1} \geq 2C_i$ (et largement), ainsi $C_0 \geq 2^n$, ce qui montre que la complexité est au moins exponentielle (sans doute trop fastidieux d'être plus précis). Cette complexité n'est donc pas raisonnable.

Pour l'algorithme de bas en haut la complexité est $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n}(j-i) = \sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2}$, on est donc en $O(n^3)$. Ce qui est bien meilleurs que pour l'algorithme récursif naïf.

Q25.
```
def lignes(mots:[str],t:[int],L:int):
```

```
    i=0
    res=[]
    while i<len(t):
        res.append(mots[i:t[i]])
        i=t[i]
    return res
```

Q26.
```
def formatage(lignesdemots:[[str]],L:int)->str:
    res=""
    for ligne in lignesdemots:
        lmots=sum([len(mot) for mot in ligne])
        nbesp=L-lmots
        if len(ligne)==1:
            res += ligne[0]+" "*nbesp+"\n"
        else:
            #Nombre d'espaces entre chaque mot (pb cela ne tombe pas juste, d'où reste)
            nbespmotsmin=nbesp//(len(ligne)-1)
            nbespmotsreste=nbesp%(len(ligne)-1)
            lesespaces=[nbespmotsmin]*(len(ligne)-1)

            # pas claire sur comment répartir ces espaces supplémentaires, on doit rajouter
            # +1 à nbespmotsreste éléments de cette liste, on procède de manière régulière
            if nbespmotsreste>0:
                eespsup=(len(ligne)-1)//nbespmotsreste #un esp de plus tous les eespsup mots
                for k in range(nbespmotsreste):
                    i=k*eespsup
                    lesespaces[i]+=1

            for i in range(len(ligne)-1):
                res += ligne[i] + " "*lesespaces[i]
            res += ligne[len(ligne)-1] + "\n"
    return res
```