```
## Partie 1 : préambule

# Q1
"100 en base 16 fait 1*16**2 = 256 . Le montant versé est donc de 2 dollars et 56 cents"

# Q2
" Le dessin donne un j"
# pour tester dans la suite
varJ = [[[0.25,1],[0.25,-1],[0,-1]],[[0.25,1.25]]]
## Partie 2 : Gestion des polices de caractères vectorielles

# Q3
"""
SELECT COUNT(*) From Glyphe
WHERE groman = "True"
"""

# Q4
"""
SELECT gdesc FROM Glyphe
JOIN Caractere ON Glyphe.code = Caractere.code
JOIN Police ON Police.pid = Glyphe.pid
WHERE car = "A"
AND pnom = "Helvetica"
AND groman = "False"
"""

# Q5
"""
SELECT fnom, COUNT(*) FROM Famille
JOIN Police ON Police.fid = Famille.fid
GROUP BY fnom
ORDER BY fnom
"""

### Partie 3 : Manipulation de descriptions vectorielles de glyphes

# Q6
def points(v : [[[float]]])->[[float]] :
    """ renvoie la liste des points de la multiligne v """
    Lp = []
    for ml in v : # pour chaque multi-ligne
        for pt in ml : # pour chaque point dans une multi-ligne
            Lp.append(pt) # on complète la liste des points
    return Lp
#print(points(varJ))

# Q7
def dim(l : [[float]], n : int)->[float] :
    """ renvoie la liste des éléments d'indice n des éléments de l """
    Lelem = []
    for elem in l : # on gère chacune des listes de l
        Lelem.append(elem[n]) # dont on extrait l'élément d'indice n
    return Lelem
#print(dim(points(varJ),1))

# Q8 (dommage que le sujet ne fasse pas programmer max et min en même temps)
def largeur(v : [[[float]]])->float :
    """ renvoie la largeur d'une glyphe """
    Lxp = dim(points(v),0)  # on récupère les abscisses des points
    return max(Lxp)-min(Lxp)
#print(largeur(varJ))

# Q9  (sans utiliser ord() et chr() c'est assez pénible à écrire)
def obtention_largeur(police : str)->[float] :
    """ Renvoie une liste contenant la largeur de chaque lettre minuscule
    italique de police """
    # lettres = ["a","b",.....,"z"] # long à écrire
    lettres = [chr(ord("a")+i) for i in range(26)]
```

```
    Llarg = []
    for let in lettres :
        Llarg.append(glyphe(let,police,True)) # on ajoute d'abord la lettre romane
        Llarg.append(glyphe(let,police,False)) # puis l'italique
    return Llarg

# Q10
def transforme(f : callable, v : [[[float]]])->[[[float]]] :
    """ applique f à tous les points de v et renvoie une nouvelle liste """
    Lres = []
    for ml in v :
        Llig = [] # une liste pour chaque nouvelle ligne
        for pt in ml :
            Llig.append(f(pt)) # que l'on remplit avec les images des points
        Lres.append(Llig)
    return Lres

# Q11
"""
On divise par 2 l'abscisse de points, cela correspond à une affinité orthogonale
de rapport 0.5 et de direction vect(i) sur l'axe Oy
"""

# Q12
def fp(point : [float])->[float] :
    """ renvoie un nouveau point donc les coordonnées ont été "penchées" """
    return [point[0] + 0.5*point[1]]
def penche(v : [[[float]]])->[[[float]]] :
    """ renvoie une nouvelle description vectorielle de v, penchée à droite """
    return transforme(fp,v)

#print(penche(varJ))

## Partie IV : rasterisation

# Q13
"""
On va de (0,0) à (6,2)
dx = 6, dy = 2 donc dy/dx = 1/3, les points sont
(0,0), (1,0), (2,1), (3,1), (4,1), (5,2), (6,2)
"""

# Q14
"""
On va de (9,8) à (1,9)
cette fois dx <= 0 et on ne rentre pas dans la boucle et il n'y aura pas de point
Il ne faut pas que dx soit négatif, donc :
    assert p0[0] < p0[1]
"""

# Q15
"""
On va de (3,0) à (5,8)
dx vaut 2 et dy vaut 8 donc dy/dx vaut 4
Les points sont (3,0), (4,4), (5,8).
Cette fois les pixels ne sont pas voisins, il y a des trous dans le segment,
car dy/dx >2. En fonction de dx cela peut poser un problème dès que dy/dx est supérieur à 1.
"""

# Q16
"""
Pour régler le problème on peut avancer de 1 en 1 en y plutôt qu'en x
"""
def trace_quadrant_sud(im, p0 : (int), p1 : (int))->None :
    """ la fonction renmpli im de pixels pour tracer le segment p0p1 """
    x0,y0 = p0
    x1,y1 = p1
    dx,dy = x1-x0,y1-y0
    im.pupixel(p0,0)  # le pixel de départ est encré
```

```
    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, p0 : (int), p1 : (int))->None :
    """ rempli im de pixel pour tracer le segment p0p1 """
    x0,y0 = p0
    x1,y1 = p1
    dx,dy = x1-x0,y1-y0
    # on gère le cas où les points sont identiques
    if dx == 0 and dy == 0 :
        im.putpixel(p0,0)
    elif dx == 0 :
        if dy > 0 :
            trace_segment_sud(im,p0,P1)
        else :
            trace_segment_sud(im,p1,p0)
    elif dy == 0 :
        if dx > 0 :
            trace_segment_est(im,p0,p1)
        else :
            trace_segment_est(im,p1,p0)
    elif abs(dy/dx) < 1 :
        if dx > 0 :
            trace_segment_est(im,p0,p1)
        else :
            trace_segment_est(im,p1,p0)
    else :
        if dy > 0 :
            trace_segment_sud(im,p0,p1)
        else :
            trace_segment_sud(im,p1,p0)

## Partie V : Affichage du texte

# Q18
"""
je ne comprend pas bien une taille de 1, une lettre de 1 pixel de haut
est ... un pixel !
"""
def position(p : (float), pz : (int), taille : int)->(int) :
    """ renvoie la position du point p dans la page pour une glyphe de taille
    taille et une origine pz """
    # les coordonnées des points du glyphe sont normalisées
    # on commence par une homothétie pour se mettre à taille
    x,y = p
    xh = floor(taille*x)
    # lorsque taille vaut 1 les points dans l'oeil auront un xh à 0 qui est
    # la coordonnée du premier pixel (cf remarque de l'énoncé)
    yh = floor(taille*y)
    # puis on translate vers pz
    xf = pz[0] + xh
    yf = pz[1] + yh
    return (xf,yf)

# Q19
# on suppose que le caractère à une origine en pz dans l'image
# lexemple donné après n'aide pas trop puisque le bas du K n'a pas l'air d'être
# en [10,40]
def affiche_car(page, c : str, police : str, roman : bool, pz : (int), taille : int)->int :
    """ crée le caractère c dans l'image page """
    carac = glyphe(c,police,roman)
    for ml in carac : # on récupère les lignes brisées
        if len(ml) == 1 :
            debut = position(ml[0],pz,taille)
            trace_segment(page,debut,debut)
        else :
```

```
            for i in range(len(ml)-1) :
                debut = position(ml[i],pz,taille)
                fin = position(ml[i+1],pz,taille)
                trace_segment(page,debut,fin)
    return taille*largeur(carac)

# Q20
def affiche_mot(page, mot : str, ic : int, police : str, roman : bool, pz : (int), taille : int)->
int :
    """ crée le mot mot dans l'image page """
    debut = pz
    for car in mot :
        larg = affiche_car(page,car,police,roman,debut,taille)
        debut = [debut[0] + larg + ic,debut[1]]
    return [debut[0]-ic,debut[1]]

## Partie VI : justification d'un paragraphe

# Q21
"""
Pour chaque nouveau mot on regarde s'il reste de la place dans la ligne
si non on ajoute la ligne et on met le mot dans la ligne suivante
si oui on ajoute le mot à la ligne
Cet algorithme est glouton puisque il regarde le problème de la complétion
de ligne localement et pas globalement : chaque ligne est traitée l'une après l'autre
"""

# Q22
" le calcul du coût de cout() est longueur de la ligne moins les espaces moins les mots "
"""
lmots = [2,4,2,6,6]
a)
le découpage est (0,2),(3,3), (4,4)
quand on somme les coûts on a 0 + 16 + 16 = 32
b)
le déocupage est (0,1), (2,3), (4,4)
la somme des coûts donne 9 + 1 + 16 = 26

Le coût de b) est donc moindre, le découpage par programmation dynamique
est considéré comme plus harmonieux
"""

# Q23 (c'est du cours)
#memo = {len(lmots) : 0}  ## là il y a une petite confusion sur ce qu'est lmots
    ## c'est probablement la description du texte mais c'est aussi un paramètre de progd_memo()
def progd_memo(i : int, lmots : [int], L : int, memo : {int : int})->None :
    if i in memo :
        return memo[i]
    else :
        mini = float("inf")
        for j in range(i+1,len(mots)+1) :
            d = progd_memo(j,lmots,L) + cout(i,j-1,lmots,L)
            if d < mini :
                mini = d
        memo[i] = mini
    return mini

# Q24
"""
Pour l'algorithme récursif. Une réponse intuitive est que la complexité est en
O(e**n) en raison des appels récursifs imbriqués.
En détaillant un peu (merci à mes collègues Axel rogue et Martin Canals avec qui j'ai
échangé pour construire le raisonnement).
soit C_k le coût de algo_recursif(k,lmots,L).
On voit que C_0 = somme de i =1 à n de C_i + les coûts de la fonction cout()
on a donc C_0 > somme de i =1 à n de C_i , si les C_n valent 1 on sait que cette somme
vaut 2**(n-1).
On a donc un nombre d'opération qui est (très) supérieur à 2**(n-1).
La complexité est donc au moins en O(e**n)
```

```
Pour l'algo bashaut, on se retrouve avec la somme des couts des fonctions cout(), ce qui
revient plus ou moins à faire la somme des entiers (à un décalage près). La complexité
est donc en O(n**2).
"""


# Q25
def lignes(mots : [str], t : [int], L : int)->[[str]] :
    """ mots contient les mots du texte, t contient des indcations pour un placement optimal
    et L est la longueur d'une ligne. La fonction renvoie la liste des lignes du texte """
    i = 0  #i est indice qui va parcourir t de manière non régulière
    Lres= []
    while i<len(t) :
        Lres.append(mots[i:t[i]]) # une phrase commençant à i fini à t[i]-1 inclus
        i = t[i] # on décale l'indice de départ
    return Lres
#print(lignes(["Ut","enim","ad","minima","veniam"],[2,3,4,4,5],10))
# Q26
def formatage(lignesdemots : [[str]], L : int)->str :
    """ lignesdemot est une liste de liste de mot composant des lignes, L est la longueur
    d'une ligne.
    La fonction va renvoyer une chaîne de caractère correspondant au texte justifié """
    texteJ = ""
    for lig in lignesdemots :
        lmots= 0 # calcul la longueur de tous les mots de la ligne
        for mot in lig :
            lmots += len(mot)
        nbEspace = L - lmots  # le nombre d'espaces à insérés
        if len(lig) == 1 :
            ligneTexte = lig[0] + " "*nbEspace + "\n"
        else :
            nbEspaceMini = nbEspace//(len(lig)-1) # le nombre minimum d'espace à placer à chaque
fois
            espaceSup = nbEspace%(len(lig)-1) # le rab d'espaces
            # on construit un liste contenant le nombre d'espace à insérer entre les mots
            Lespaces = [nbEspaceMini+1]*espaceSup + [nbEspaceMini]*(len(lig)-1-espaceSup)
            # construciton de la ligne
            ligneTexte = ""
            for i in range(len(lig)-1) :
                ligneTexte += lig[i]+" "*Lespaces[i]
            ligneTexte += lig[len(lig)-1] + "\n"
        # completion du texte
        texteJ += ligneTexte
    return texteJ

res = formatage([["Ut","enim"],["ad","minima"],["veniam"],["et","non","or"]],10)
```