La typographie informatisée

Corrigé UPSTI

Partie I - Préambule

Question 1 Quel montant est effectivement versé en dollars par Donald Knuth pour une nouvelle erreur trouvée ?

Le nombre $(100)_{16}$ vaut par définition $1 \cdot 16^2 + 0 \cdot 16^1 + 0 \cdot 16^0 = 16^2 = 256$ (cents) $= 2.56$ (dollars).

Question 2 Dessiner ce glyphe. De quel caractère s'agit-il ?

La forme du glyphe est

```
 .


 |
 |
 |
_|
```

On remarque qu'une partie important du glyphe est située en dessous de l'oeil et qu'une partie horizontale est présente en bas ce qui indique la lettre j.

Question 3 Proposer une requête en SQL sur cette base de données pour compter le nombre de glyphes en roman (cf. description précédente).

```
SELECT COUNT(*) FROM glyphe WHERE groman
```

Question 4 Proposer une requête en SQL afin d'extraire la description vectorielle du caractère A dans la police nommée Helvetica en italique.

```
SELECT gdesc FROM glyphe
JOIN Police ON pid
JOIN Caractere ON code
WHERE pnom = "Helvetica" AND NOT(groman) AND car = "A"
```

Question 5 Proposer une requête en SQL pour extraire les noms des familles qui disposent de polices et leur nombre de polices, classés par ordre alphabétique.

```
SELECT fnom,COUNT(DISTINCT Police) FROM Famille
JOIN Police ON fid
GROUP BY fnom HAVING COUNT(Police)>0
ORDER BY fnom ASC
```

Le having est facultatif car la jointure élimine les fnom qui n'ont pas de fid.

Question 6 Implémenter la 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.

```
def points(v):
    L_points = []
    for multipoints in v:
        for point in multipoints:
            L_points.append(point)
    return L_points
## Ou encore
def points(v):
    L_points = []
    for multipoints in v:
        L_points += multipoints
    return L_points
```

Question 7 Implémenter la fonction utilitaire `dim(l:[[float]], n:int)->[float]` qui renvoie la liste des éléments d'indice `n` (en commençant à 0) des sous listes de flottants, dont on supposera qu'ils existent toujours.

```
def dim(l,n):
    L = []
    for elem in l:
        L.append(elem[n])
    return L
```

Question 8 Implémenter la fonction `largeur(v:[[[float]]])->float` qui renvoie la largeur de la description vectorielle `v`. Il faudra utiliser les fonctions utilitaires précédentes ainsi que les fonctions `max` et `min` de Python appliquées à des listes.

```
def largeur(v):
    L_points = points(v)
    val_largeur = max(dim(L_points,0))-(min(dim(L_points,0)))
    return val_largeur
```

Question 9 Implémenter la fonction `obtention_largeur(police:str)->[float]` qui renvoie une liste de largeurs pour toutes les lettres minuscules romanes et italiques (uniquement les 26 lettres non accentuées de a à z) de la police `police` dans l'ordre a roman, a italique, b roman, b italique ...

```
def obtention_largeur(police):
    L = []
    for lettre in "abcdefghijklmnopqurstuvxyz":
        for is_roman in [True,False]:
            val_glyphe = glyphe(lettre,police,is_roman)
            L.append(largeur(val_glyphe))
    return L
```

Question 10 En se basant sur l'exemple de la fonction `applique`, implémenter une fonction utilitaire `transforme(f:callable, v:[[[float]]])->[[[float]]]` 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. La fonction `f` passée en argument transforme un point en un autre point.

```
def transforme(f,v):
    L = []
    for multipoint in v:
        new_multipoint = []
        for point in multipoint:
            new_multipoint.append(f(point))
        L.append(new_multipoint)
    return L
```

Question 11 Expliquer comment est modifiée une description vectorielle `v` par `transforme(zzz, v)`. Préciser l'effet obtenu sur un glyphe.

La transformation proposée divise la largeur d'un glyphe par 2 sans modifier la hauteur. Tous les points sont rapprochés de l'abscisse 0 en les multipliant par 0.5.

Question 12 Implémenter la fonction `penche(v:[[[float]]])->[[[float]]]` qui renvoie une nouvelle description vectorielle correspondant à un glyphe penché vers la droite, obtenue en modifiant comme suit les coordonnées des points (x, y) :

- la nouvelle abscisse est `x + 0.5 * y` ;
- la nouvelle ordonnée reste `y`.

```
def italique(p):
    return [p[0] + 0.5 * p[1],p[1]]

def penche(v):
    return transforme(italique,v)
```

Question 13 Préciser les coordonnées des pixels de l'image encrés (pixels que l'on met en noir) par l'exécution de la ligne 15 ?

La ligne 15 va encrer le pixel (0,0) puis les pixels

(1,0 + floor(0.5 + 1*2/6) = (1,0 + floor(5/6)) = (1,0)

(2,floor(7/6)) = (2,1)

(3,1)

(4,1)

(5,2)

et (6,2)

Question 14 Préciser les coordonnées des pixels de l'image encrés par l'exécution de la ligne 16 ? Indiquer d'où vient le problème rencontré. Proposer une assertion à mettre en début de fonction pour éviter ce problème.

La ligne 16 ne va encrer que le premier (9,8) et le dernier pixel (1,9), car `dx` étant négatif, la boucle for ne comportera aucune itération (il faudrait préciser un pas négatif éventuellement).

On pourrait ajouter la ligne `assert p1[0]>= p0[0]` pour éviter ce problème.

Question 15 Préciser les coordonnées des pixels de l'image encrés par l'exécution de la ligne 17 ? Expliquer le problème rencontré et à quoi il est dû.

La ligne 17 encrera les points

(3,0)

(4,floor(0+ 1* 8/2)) = (4,4)

et (5,8)

Ceci ne fait pas une ligne continue car il n'y a pas de pixel encré aux lignes 1,2,3,5,6,7. C'est dû au fait que l'on ne place qu'un pixel encré par colonne sans se préoccuper du remplissage des lignes.

Question 16 En s'inspirant du code de la fonction `trace_quadrant_est`, implémenter une nouvelle fonction `trace_quadrant_sud` qui règle le problème précédent.

```
def trace_cadrant_sud(im,p0,p1):
    assert p1[1]-p0[1]>=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 + i* dx / dy), y0+i )
        im.putpixel(p,0)
    im.putpixel(p1,0)
```

Question 17 Implémenter la fonction `trace_segment(im:Image, p0:(int), p1:(int))` qui trace un segment continu entre les pixels `p0` et `p1` sur l'image `im` supposée assez grande. Cette fonction devra opérer correctement si les deux points passés en arguments sont égaux.

```
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):
        if x1 >= x0:
            trace_quadrant_est(im, p0, p1)
        else:
            trace_quadrant_est(im, p1, p0)
    else:
        if y1 > y0:
            trace_quadrant_sud(im, p0, p1)
        else:
            trace_quadrant_sud(im, p1, p0)
```

Question 18 Implémenter la fonction `position(p:(float), pz:(int), taille:int)->(int)` 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. Prendre garde à la bonne orientation du glyphe sur la page. Vérifier que, pour la taille limite 1, l'oeil du glyphe fait bien 1 pixel de hauteur.

```
def position(p,pz,taille):
    px = pz[0]+p[0]*taille # les abscisses vont de gauche a droite
    # en representation glyphe et image
    py = pz[1]-p[1]*taille # pour les ordonnees, si p[1] est grand,
    # le glyphe a des pixels sombre en haut alors que dans l'image
    # y est fleche vers le bas donc c'est l'inverse
    ### la fonction demande un entier en sortie il faut donc arrondir
    ### au pixel le plus proche :
    return int(px+0.5),int(py+0.5)
```

Question 19 Implémenter la fonction `affiche_car(page:Image, c:str, police:str, roman:bool, pz:(int), taille:int)->int` 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. Pour rappel, la fonction `glyphe(c:str, police:str, roman:bool)` charge la description vectorielle du caractère `c` dans la police `police` pour la variante `roman`.

```
def affichage_car(page,c,police,roman,pz,taille):
    val_glyphe = glyphe(c,police,roman)
    larg = largeur(val_glyphe)*taille
    for multipoint in val_glyphe:
        if len(multipoint) == 1: # un seul point:
            pi = position(multipoint[0],pz,taille)
            trace_segment(page,pi,pi)
        else:
            for i in range(len(multipoint))-1:
                pip1 = position(multipoint[i+1],pz,taille)
                pi = position(multipoint[i], pz, taille)
                trace_segment(page,pi,pip1)
    return larg
```

Question 20 Implémenter la fonction `affiche_mot(page:Image, mot:str, ic:int, police:str, roman:bool, pz:(int), taille:int)->int` 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.

```
def affichage_mot(page,mot,ic,police,roman,pz,taille):
    for lettre in mot:
        larg = affichage_car(page,lettre,police,roman,pz,taille)
        pz[0] += larg + ic
    return pz[0] - ic  # on enleve le dernier espace pour renvoyer la
    # position finale
```

On peut notamment tester les différents codes en utilisant la fonction glyphe ci-dessous :

```
def glyphe(c, p, r):
    if c == "i":
        return [[[0.1, 1.0], [0.1, 0], [0.0, 0]], [[0.1, 1.25]]]
    if c == "e":
        return [[[0, 0.5], [1, 0.5]], [[0, 0.5], [1, 0]], [[0, 0.5], [1, 1]]]
    if c == "o":
        return [[[0, 1], [1, 1]], [[0, 1], [0.5, 0]], [[1, 1], [0.5, 0]]]
```

```
    if c == "ö":
        return [[[0, 1], [1, 1]], [[0, 1], [0.5, 0]], [[1, 1], [0.5, 0]] , [[0.25, 1.25]],
         [[0.75, 1.25]]]
    if c == "j":
        return [[[0.25, 1.0], [0.25, -1.0], [0.0, -1.0]], [[0.25, 1.25]]]
    if c == "K":
        return [[[0, 0], [0, 2]],  [[0, 1], [2, 0]], [[0, 1], [2, 2]]]
    if c == "G":
        return [[[0, 1], [2, 0]],  [[2, 0], [2, 1]], [[2, 1], [1, 1]], [[0, 1], [2, 2]]]
    if c == "d":
        return [[[1, 0], [1, 2]],  [[0, 0.5], [1, 0]], [[0, 0.5], [1, 1]]]
    if c == "l":
        return [[[0, 0], [0, 2]],  [[0, 0], [1, 0]]]
    if c == ".":
        return [[[0, 0]]]
    return [[[0, 0],[1, 1]], [[0, 1], [1, 0]]]
```

Question 21 Expliquer en une ou deux phrases le principe de l'algorithme et pourquoi il est dit glouton.

L'algorithme remplit chaque ligne au fur et à mesure en essayant de la remplir au maximum et effectue un retour à la ligne dès qu'il y a un dépassement. Si on souhaite justifier le texte i.e. avoir des lignes de longueurs sensiblement égales, ça ne semble pas être le meilleur choix.

L'algorithme est glouton car les décisions sont prises en ne traitant que la ligne en cours sans se préoccuper de ce qui se passera dans la suite du problème.

Question 22 Évaluer pour les deux découpages a) et b) de l'exemple, ce que renvoie la fonction coût pour chacune des lignes en précisant les indices `i` et `j` pour chaque ligne (`lmots=[2,4,2,6,6]`). Conclure sur l'algorithme qui donne la solution la plus harmonieuse en sommant les différents coûts par ligne pour le découpage a) puis pour le découpage b).

On remarque que le code utilise l'extraction par slicing `lmots[i:j+1]` pour extraire le nombre de caractères des mots `i` à `j` inclus. On commence par discuter les coûts de la colonne de gauche (cas a) :

- La première ligne contenant les mots d'indice 0,1 et 2, on prendra i=0 et j=2. Dans ce cas pour L=10, la fonction calcule le coût $(10-(2-0)-(2+4+2)^2=0$. Ici la ligne est parfaitement remplie.
- Pour la deuxième ligne, on a le mot d'indice 3 donc i=3 et j =3, la fonction calcule le coût $(10-6)^2=4^2=16$.
- Pour la troisième ligne, on a le mot d'indice 4 donc i=4 et j=4 et la fonction calcule le coût $(10-6)^2=16$

Le coût total pour la colonne de gauche est donc $0+16+16=32$.

Pour la colonne de droite :

- 1ere ligne : i=0 et j=1 d'où le coût $(10-(1-0)-(2+4))^2=3^2=9$;
- 2e ligne : i=2 et j=3 d'où le coût $(10-(1-0)-(2+6))=1^2=1$; 3e ligne : i=4 et j=4 d'où le coût $(10-6)^2=16$

Le coût total pour la colonne de gauche est donc $9+1+16=26$

La second répartition est donc la meilleure pour cette fonction de coût.

Question 23 Proposer une modification de la fonction `algo_recursif` pour rendre celle-ci plus efficace en introduisant une mémoïsation. On définira la nouvelle fonction récursive
`progd_memo(i:int,lmots:[int],L:int,memo:{int:int})` avec la variable `memo`, dictionnaire initialisé en

dehors de la fonction par `memo={len(m):0}`

```
def progd_memo(i:int,lmots:[int],L:int,memo:{int:int}):
    if i==len(lmots):
        return 0
    ### Ajout ###
    if i in memo:
        return memo[i] # d(i) deja calcule
    ###
    else:
        mini=float("inf")
        for j in range(i+1,len(lmots)+1):
            d=progd_memo(j,lmots,L,memo)+cout(i,j-1,lmots,L)
            if d<mini:
                mini=d
        ### Ajout ###
        memo[i] = mini # stockage d(i)
        ###
        return mini
```

Question 24 Analyser, en fonction de `n` nombre de mots, la complexité temporelle asymptotique associée à l'algorithme récursif naïf (fonction `algo_recursif`) ainsi qu'à l'algorithme de programmation dynamique de bas en haut (fonction `progd_bashaut`) et conclure sur l'intérêt de l'algorithme de programmation dynamique. On ne comptera que les opérations de type additions/soustractions. Il n'est pas attendu de développements mathématiques poussés, les résultats peuvent être donnés sans justification.

L'algorithme récursif est lancé avec $i = 0$ (pour placer de manière optimale les retours à la ligne de la fin au début). Ainsi la boucle for parcours tous les indice j de 1 à n. Et lance un appel à `progd_memo(j)` ainsi que le calcul de `cout(i,j-1,lmots,L)`. La fonction cout fait la somme des éléments de lmots entre i et j son coût est donc en $O(j-i)$.Si on note $C(i)$ la complexité de `progd_memo(i)`, on a alors

$$C(i) = \sum_{j=i+1}^{n} (C(j) + O(j-i))$$

or $\sum_{j=i+1}^{n} O(j-i) = O(\frac{(j-i)(j-i+1)}{2}) = O((j-i)^2)$ et donc $C(i) = (\sum_{j=i+1}^{n} C(j)) + O((j-i)^2)$. Le recouvrement des appels conduit à une complexité exponentielle en n mais on peut détailler le calcul :

$$\begin{aligned}
C(0) &= O(n^2) + \sum_{j=1}^{n}(C(j)) = O(n^2) + O((n-1)^2) + 2\sum_{j=2}^{n}(C(j)) \\
&= O(n^2) + O((n-1)^2) + 2O((n-2)^2) + 2^2\sum_{j=3}^{n}(C(j)) \\
&= O(n^2) + O((n-1)^2) + 2O((n-2)^2) + 2^2O((n-3)^2) + 2^3 \sum_{j=4}^{n}(C(j)) \\
&= O(n^2) + \sum_{k=1}^{n} 2^{k-1}O((n-k)^2) + 2^{n-1}O(1) \\
&= O(2^n)
\end{aligned}$$

En revanche si on applique la méthode bottom up, on voit deux boucles imbriquées qui appellent la fonction `cout(i,j-1)`, la complexité vaut donc $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n} O(j-i) = \sum_{i=0}^{n-1} O((n-i)^2) = O(n^3)$

Question 25 Proposer une implémentation de cette fonction `lignes(mots:[str],t:[int],L:int)`

Le tableau `t` contient à l'indice `i` l'indice du mot auquel il faudrait s'arrêter en partant du mot `i` pour remplir de manière optimale la ligne. Ainsi la première ligne doit contenir les mots d'indice allant de 0 à `t[0]` (exclus). Comme `t[0]` = 2, on placera en 2ième ligne les mots d'indice 2 à `t[2]` (exclus) et ainsi de suite. D'où le code.

```
def lignes(mots:[str],t:[int],L:int)->[[str]]:
    L_mots = []
    indice_debut = 0
    indice_fin = t[0]
    while indice_fin<len(t):
        L_mots += [mots[indice_debut:indice_fin]]
        indice_debut = indice_fin
        indice_fin = t[indice_debut]
    # il faut traiter le dernier cas ou indice_fin == len(t)
    L_mots += [mots[indice_debut:indice_fin]]
    return L_mots
```

Question 26 Proposer une implémentation de cette fonction `formatage(lignesdemots:[[str]],L:int)->str` qui renvoie la chaîne de caractères justifiée.

Il faut traiter plusieurs cas :

- Un seul mot : il suffit de l'écrire et ajouter un retour à la ligne
- Plus d'un mot : il faut alors répartir convenablement les espaces. On peut compter les espaces en calculant le nombre de caractères des mots à placer et en faisant la différence avec le nombre de caractères max d'une ligne. Ensuite on divise ce nombre de caractères par le nombre de mots -1 pour savoir combien d'espace placer après chaque mot (le dernier mot n'est pas suivi d'espaces d'où le -1. Si ce nombre ne tombe pas juste, certains mots (on a choisi ici arbitrairement les premiers) recevront un espace de plus.

```
def formatage(lignesdemots:[[str]],L:int):
    s = ""
    for ligne in lignesdemots:
        if len(ligne) == 1:
            s+= ligne[0] + "\n"
        else: # cas plus d'un mot :

            taille_ligne = 0
            for mot in ligne :
                taille_ligne += len(mot)

            espaces = L - taille_ligne
            nb_espaces_apres_chaque_mot = espaces//(len(ligne)-1)
            # nombre a mettre apres chaque mot
            nb_espaces_en_plus = espaces - nb_espaces_apres_chaque_mot
            # espaces eventuels a mettre sur certains mots (il y en a
            # pas assez pour en placer devant tous les mots donc on en
            # mettra un de plus sur les premiers mots jusqu'a ce qu'ils
            # soient tous places)
            ligne_suivante = ""
            for i in range(len(ligne)):
                ligne_suivante += ligne[i]
                ligne_suivante += nb_espaces_apres_chaque_mot*" "
                if nb_espaces_en_plus>0:
                    ligne_suivante+= " "
                    nb_espaces_en_plus -= 1

            s += ligne_suivante + '\n'
    return s
```