CORRIGÉ DE L'ÉPREUVE Mines-Ponts 2023 D'ITC

Partie I

Question 1 – $(100)_{16} = 1 \times 16^2 + 0 \times 16 + 0 \times 1 = 256$.
La récompense est donc de $2,56$ dollars.

Question 2 – Après avoir tracé les segments, le caractère dont il s'agit est un j.

Partie II

Question 3 –

```
SELECT COUNT(*)
FROM Glyphe
WHERE groman = True
```

Question 4 –

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

Rem : Peut-être pouvait-on remplacer la condition `C.car = 'A'` par `code = 65` (ce qui évitait une jointure).

Question 5 –

```
SELECT F.fnom , Count(*)
FROM  Famille AS F JOIN Police AS P
                ON P.fid = F.fid
GROUP BY P.fid
ORDER BY F.fnom ASC
```

Rem : `ASC` n'est pas indispensable car choisi par défaut.

Partie III

Question 6 – Remarque : il y a une coquille ici dans l'énoncé au niveau du type de la variable que doit renvoyer la fonction. Il s'agit d'une liste de listes de flottants (`[[float]]`) et non une liste de flottants (`[float]`).
L'exemple pris dans l'énoncé était clair cependant sur ce qui était attendu.

```
def points(v):
    liste = []
    for multiL in v:
        for point in multiL:
            liste.append(point)
    return liste
```

Question 7 –

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

Question 8 –

```
def largeur(v):
    listePoints = points(v)
    listeAbscisses = dim(listePoints,0)
    m , M = min(listeAbscisses) , max(listeAbscisses)
    return M-m
```

Question 9 –

```
def obtention_largeur(police):
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    listeLargeur = []
    for c in alphabet:
        for r in [True , False]:
            v = glyphe(c,police,r)
            l = largeur(v)
            listeLargeur.append(l)
    return listeLargeur
```

Question 10 –

```
def transforme(f,v):
    vTrans = []
    for multiL in v:
        multiLTrans = []
        for point in multiL:
            multiLTrans.append(f(point))
        vTrans.append(multiLTrans)
    return vTrans
```

On peut le faire en compréhension :

```
def transforme(f,v):
    vTrans = [ [f(point) for point in multiL] for multiL in v]
    return vTrans
```

Question 11 – L'abscisse de chaque point est divisée par deux alors que son ordonnée reste inchangée. L'effet visuel est un écrasement horizontal.

Question 12 –

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

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

Partie IV

Question 13 – Les pixels encrés par l'exécution de la ligne 15 sont : $(0,0)$, $(1,0)$, $(2,1)$, $(3,1)$, $(4,1)$, $(5,2)$, $(6,2)$.

Question 14 – Lors de l'exécution de la ligne 16, la variable `dx` utilisée dans la fonction prend la valeur -8. La boucle utilisée dans la fonction est donc appelée par : `for i in range(1,-8)`. Aucun pixel n'est donc encré.
Pour résoudre ce problème, on pourrait mettre l'assertion suivante en début de fonction (après la définition de `dx`) : `assert dx > 0`.
Afin de tracer le segment lorsque `dx` est négatif, il suffira alors d'appeler cette même fonction en échangeant les rôles de `p0` et `p1`.

Question 15 – Les pixels encrés par l'éxécution de la ligne 17 sont : $(3,0)$, $(4,4)$ et $(5,8)$.
Ces pixels sont trop éloignés pour donner l'impression d'une ligne continue. Le problème vient ici du fait que `dy` est plus grand que `dx`. Pour y remédier, il faut écrire une fonction similaire à `trace_cadrant_est` qui échange les rôles des abscisses et des ordonnées.

Question 16 – La fonction suivante sera destinée à tracer les segments pour lesquels `dy` est plus grand que `dx` (en valeur absolue). On introduit également l'assertion invoquée à la question 14.

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

Question 17 – Dans cette fonction, il nous faut éviter les problèmes soulevés lors des questions précédentes :

- sélectionner `trace_cadrant_est` ou `trace_cadrant_sud` selon que `dx` est plus grand ou non que `dy` (en valeur absolue) ;
- s'assurer que `dx` (ou `dy`) est positif en inversant éventuellement les rôles de `p0` et `p1`;
- s'assurer que la fonction trace bien un point si `p0` est égal à `p1`.

```
def trace_segment(im,p0,p1):
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1-x0, y1-y0
    if abs(dx) > abs(dy):
        if x1 > x0:
            trace_cadrant_est(im,p0,p1)
        else :
            trace_cadrant_est(im,p1,p0)
    else :
        if y1 > y0:
            trace_cadrant_sud(im,p0,p1)
        else :
            trace_cadrant_sud(im,p1,p0)
    #le cas o\'u p1==p0 n'a pas engendr\'e de trac\'e dans ces alternatives
    if p1 == p0:
        im.putpixel(p0,0)
```

Partie V

Question 18 – Pour fixer les idées, en notant `pz=(xpz,ypz)` : $(0,0)$ doit être envoyé sur (xpz, ypz), $(1,0)$ doit être envoyé sur $(xpz + taille, ypz)$, $(0,1)$ doit être envoyé sur $(xpz, ypz - taille)$.

```
def position(p,pz,taille):
    xpz , ypz = pz
    xp = xpz + floor(taille * x)
    yp = ypz - floor(taille * y)
    return (xp , yp)
```

Remarque : que se passe-t-il si `taille` vaut 1 ?

Question 19 –

```
def affiche_car(page,c,police,roman,pz,taille):
    c_vect = glyphe(c,police,roman)
    l = floor(largeur(c_vect)*taille)
    #on doit tracer chaque multiligne de c_vect :
    for multiL in c_vect:
        nbPoints = len(multiL)
        # on traite le cas o\'u il n'y a qu'un seul point
        if nbPoints == 0:
                p0 = position(multiL[0] , pz , taille)
                trace_segment(page , p0 , p0)
        else :
                # on trace chacun des segments de multiL
                for indicePoint in range(1,nbPoints):
                        # on r'ecup'ere les coordonn\'ees enti\'eres des points
                        p0 = position(multiL[indicePoint-1] , pz , taille)
                        p1 = position(multiL[indicePoint] , pz , taille)
                        trace_segment(page , p0 , p1)

    return l
```

Question 20 –

```
def affiche_mot(page,mot,ic,police,roman,pz,taille):
        for c in mot:
                l = affiche_car(page,c,police,roman,pz,taille)
                pz += l + ic
        pz -= ic
        return pz
```

Partie VI

Question 21 – Cet algorithme construit chaque ligne de la manière suivante : il empile les mots sur une ligne (`nligne`) jusqu'à ce que le mot à traiter ne puisse pas être empilé pour cause de dépassement (condition `if (c+l) > L`) ; lorsque c'est le cas, la ligne est ajoutée au texte et on passe à la ligne suivante.
Cet algorithme peut être qualifié de glouton pour la raison suivante : la décision d'empiler un mot (ou non) sur une ligne est prise uniquement en considérant la place restant sur cette ligne et la taille du mot, et non en considérant la situation dans sa généralité. Cette optimisation est locale et non globale.

Question 22 – Pour le découpage a) :

- ligne 1 (i=0 et j=2) : 0
- ligne 1 (i=3 et j=3) : 16
- ligne 1 (i=4 et j=4) : 16

Pour le découpage b) :

- ligne 1 (i=0 et j=1) : 9
- ligne 1 (i=2 et j=3) : 1
- ligne 1 (i=4 et j=4) : 16

Avec le découpage a), on a une somme des coûts de 32 et avec le découpage b), on a une somme des coûts de 26. Ce second découpage est donc meilleur avec ce critère.

Question 23 – Remarque : que désigne `m` dans l'énoncé ? J'écrirais plutôt : `memo = {len(lmots):0}` en supposant la variable `lmots` définie auparavant.

```
memo = {len(lmots):0}
def progd_memo(i,lmots,L,memo):
        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
```

Cette fonction ne renvoie rien ; on peut ensuite récupérer la valeur de $d(i)$ par la commande `memo[i]`.

Question 24 – Notons $C(n,i)$ le nombre d'opérations à effectuer pour cette fonction lorsque le nombre de mots est n et l'indice i. Pour $i=0$ (ce qui est l'objectif final), la boucle permettant de trouver le minimum donne la relation suivante :

$$C(n,0) = \sum_{j=1}^{n} C(n,j) + K \times j \ ;$$

où K est une constante. La deuxième partie de chaque terme de cette somme est due aux opérations faites par la fonction `cout`.
Pour résoudre le problème de taille n, le coût est donc supérieur à la somme des coûts pour résoudre les problèmes de tailles 1 à $n-1$. Cela donne une complexité exponentielle.

Étudions maintenant l'algorithme programmé de bas en haut. La boucle externe est de taille n, la boucle interne de taille inférieure ou égale à n. Au sein de cette boucle interne, on fait appel à la fonction `cout` dont la complexité peut être majorée (grossièrement) par une constante fois n.
On obtient donc une complexité majorée par un $\mathcal{O}(n^3)$ (ce qui peut peut-être être amélioré en affinant l'analyse de l'utilisation de la fonction `cout`).
Cette complexité est bien meilleure car elle est polynomiale.

Question 25 –

```
def lignes(mots,t,L):
        listeLignes = []
        i = 0
        while i < len(lmots):
                listeLignes.append(mots[i:t[i]]):
                i = t[i]
        return listeLignes
```

Question 26 – Le nombre d'espaces à répartir pour chaque ligne est égal à la somme des nombres de lettres de chaque mot de la ligne. Il faut répartir ces espaces entre les mots. S'il y a n mots sur la ligne, cela fait $n-1$ espaces. Que fait-on si le nombre d'espaces n'est pas un multiple de $n-1$? Il faudrait pouvoir obtenir des espaces de taille non entière.
On écrit la fonction qui suit sans tenir compte de ce problème.

```
def formatage(lignedemots,L):
        chaine = ""
        for ligne in lignedemots:
                espaceTotal = L - sum([len(mot) for mot in ligne])
                nombreEspaceEntreDeuxMots = espaceTotal // (len(ligne)-1)
                espaceEntreDeuxMots = nombreEspaceEntreDeuxMots * " "
                chaine += ligne[0]
                for i in range(1,len(ligne)):
                        chaine += espaceEntreDeuxMots + ligne[i]
                chaine += "\n"

        return chaine
```