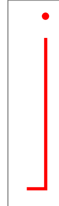
Corrigé : la typographie informatisée (Mines PC 2023)

Partie I. Préambule

❑ **Q1** – $(100)_{16} = 16^2 = 256$ donc chaque erreur trouvée rapporte 2,56 dollars.

❑ **Q2** – Il s'agit du caractère j :



Partie II. Gestion de polices de caractères vectoriels

❑ **Q3** –

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

❑ **Q4** –

```
SELECT gdesc FROM Glyphe NATURAL JOIN Police NATURAL JOIN Caractere
   WHERE car = 'A' AND pnom = 'Helvetica' AND groman = False ;
```

❑ **Q5** –

```
SELECT fnom, COUNT(*) FROM Famille NATURAL JOIN Police GROUP BY fid ORDER BY fnom ASC ;
```

Partie III. Manipulation de descriptions vectorielles de glyphes

❑ **Q6** – Notons que le type correct de cette fonction est `points(v:[[[float]]]) -> [[float]]` (erreur dans l'énoncé).

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

❑ **Q7** –

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

❑ **Q8** –

```
def largeur(v):
    ordonnees = dim(points(v), 1)
    return max(ordonnees) - min(ordonnees)
```

❑ **Q9** –

```
def obtention_largeur(police):
    lst = []
    for c in 'abcdefghijklmnopqrstuvwxyz':
        lst.append(largeur(glyphe(c, police, True)))
        lst.append(largeur(glyphe(c, police, False)))
    return lst
```

❑ **Q10** –

```
def transforme(f, v):
    vv = []
    for multiligne in v:
        vv.append([f(point) for point in multiligne])
    return vv
```

❑ **Q11** – Les abscisses des points de `v` sont divisées par 2 par `transforme(zzz, v)` ; ainsi, les glyphes voient leurs chasses divisées par deux sans que leurs corps ne soit modifiés.

❑ **Q12** –

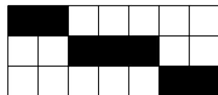
```
def xxx(p):
    return [p[0] + 0.5 * p[1], p[1]]

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

Partie IV. Rasterisation

❑ **Q13** – L'exécution de la ligne 15 trace un segment de droite entre les points $p_0 = (0,0)$ et $p_1 = (6,2)$. On a $(d_x, d_y) = (6,2)$ donc outre les deux extrémités, cinq pixels seront encrés : les pixels de coordonnées

– $(1, \lfloor 1/2 + 1/3 \rfloor) = (1,0)$;
– $(2, \lfloor 1/2 + 2/3 \rfloor) = (2,1)$;
– $(3, \lfloor 1/2 + 3/3 \rfloor) = (3,1)$;
– $(4, \lfloor 1/2 + 4/3 \rfloor) = (4,1)$;
– $(5, \lfloor 1/2 + 5/3 \rfloor) = (5,2)$.



❑ **Q14** – Lors de l'exécution de la ligne 16, on a $d_x = -8 < 0$ donc seuls les deux points d'extrémités $p_0 = (9,8)$, $p_1 = (1,9)$ sont encrés. Pour éviter se problème, on peut ajouter `assert p0[0] > p1[0]` au début de la fonction.

❑ **Q15** – Lors de l'exécution de la ligne 17, on a $(d_x, d_y) = (2,8)$ donc outre les deux points extrémités, un seul point sera encré, le point $(1,4)$. La fonction n'est utilisable que lorsque le segment $[p_0, p_1]$ appartient au secteur angulaire $[-\pi/4, \pi/4]$.

❑ **Q16** – Il suffit de permuter les rôles des abscisses et des ordonnées :

```
def trace_quadrant_sud(im, p0, p1):
    assert p0[1] < p1[1]
    dx, dy = x1 - x0, y1 - y0
    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)
```

❑ **Q17** –

```
def trace_segment(im, p0, p1):
    if p0 == p1:
        im.putpixel(p0, 0)
    else:
        dx, dy = p1[0] - p0[0], p1[1] - p0[1]
        if abs(dx) <= abs(dy):
            if dx > 0:
                trace_quadrant_est(im, p0, p1)
            else:
                trace_quadrant_est(im, p1, p0)
        else:
            if dy > 0:
                trace_quadrant_sud(im, p0, p1)
            else:
                trace_quadrant_sud(im, p1, p0)
```

Partie V. Affichage de texte

❑ **Q18** –

```
def position(p, pz, taille):
    x = pz[0] + int(taille * p[0])
    y = pz[1] - int(taille * p[1])
    return (x, y)
```

❑ **Q19** –

```
def affiche_car(page, c, police, roman, pz, taille):
    v = glyphe(c, police, roman)
    for multiligne in v:
        p0 = multiligne[0]
        for p1 in multiligne[1:]:
            trace_segment(page, position(p0, pz, taille), position(p1, pz, taille))
            p0 = p1
    return int(largeur(v) * taille)
```

❑ **Q20** – Par « position du dernier pixel » j'imagine qu'il s'agit de son abscisse.

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

Partie VI. Justification d'un paragraphe

❑ **Q21** – L'algorithme `glouton` cherche à remplir au maximum chacune des lignes, au fur et à mesure ; il est dit glouton car il cherche à obtenir un optimum local (pour chaque ligne), dans l'espoir d'obtenir ainsi un résultat approchant de l'optimum global (pour le texte dans son entier).

❑ **Q22** – Pour le découpage glouton :

- **ligne 1** les mots d'indices 0 à 2, pour un coût de 0 ;
- **ligne 2** le mot d'indice 3, pour un coût de 16 ;
- **ligne 3** le mot d'indice 4, pour un coût de 16.

Soit un coût total de 32.

Pour le découpage dynamique :

- **ligne 1** les mots d'indices 0 à 1, pour un coût de 9 ;
- **ligne 2** les mots d'indices 2 à 3, pour un coût de 1 ;

ligne 3 le mot d'indice 4, pour un coût de 16.

Soit un coût total de 26.

Le découpage par programmation dynamique est le plus harmonieux, comme on pouvait le prévoir.

❏ **Q23** – Ajoutons un dictionnaire pour stocker les résultats déjà calculés :

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

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

On notera là encore une erreur d'énoncé sur l'initialisation du dictionnaire `memo`.

❏ **Q24** – Pour simplifier le raisonnement, je vais supposer le calcul de la fonction `cout` de complexité constante (en toute rigueur, connaître la complexité de la fonction `sum` est de toute façon hors-programme, même si on se doute qu'elle est de complexité linéaire).

Notons $C(n)$ la complexité temporelle de l'algorithme récursif. On a $C(n) = K + \sum_{i=0}^{n-1} C(i)$ où K est une constante donc $C(n) - C(n-1) = C(n-1)$, ou encore $C(n) = 2C(n-1)$. On a donc $C(n) = O(2^n)$; la complexité est exponentielle.

Notons $C'(n)$ la complexité temporelle de bas en haut. On a $C'(n) = O(n^2)$ donc la complexité est quadratique.

Oh surprise, l'algorithme récursif sans mémoïsation est de complexité bien supérieure !

❏ **Q25** – Contrairement à ce que pourrait laisser penser l'énoncé, cette fonction ne dépend pas de la valeur de `L` :

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

❏ **Q26** – Question un peu problématique : comment répartir « au mieux » k espaces entre p mots lorsque $\frac{k}{p-1}$ n'est pas entier ?