


Mines-Ponts - Informatique commune - 2023

Corrigé	1
Commentaires	4

Corrigé

1. Le nombre 100 interprété en base 16 vaut en base décimale $16^2 = 256$. Le montant versé par Donald Knuth pour une nouvelle erreur trouvée est donc 2.56\$.

2. 

Il s'agit du caractère j.

3.

```
SELECT COUNT(*)
FROM Glyphe
WHERE groman = TRUE;
```
4.

```
SELECT g.gdesc
FROM Glyphe g
JOIN Caractere c ON g.code = c.code
JOIN Police p ON g.pid = p.pid
WHERE c.car = 'A'
      AND p.pnom = 'Helvetica'
      AND g.groman = FALSE;
```
5.

```
SELECT f.fnom, COUNT(p.pid)
FROM Famille f
JOIN Police p ON f.fid = p.fid
GROUP BY f.fnom
ORDER BY f.fnom;
```
6.

```
def points(v:[[[float]])->[[float]]:
    l = []
    for multi_ligne in v:
        for point in multi_ligne:
            l.append(point)
    return l
```
7.

```
def dim(l:[[float]], n:int)->[float]:
    return [l[i][n] for i in range(len(l))]
```
8.

```
def largeur(v:[[[float]])->float:
    p = points(v)
    abscisses = dim(p,0)
    return max(abscisses) - min(abscisses)
```
9.

```
def obtention_largeur(police:str)->[float]:
    largeurs = []
    for c in "abcdefghijklmnopqrstuvwxyz":
        largeurs.append(largeur(glyphe(c, police, True)))
        largeurs.append(largeur(glyphe(c, police, False)))
    return largeurs
```
- 10.

```
def transforme(f:callable, v:[[[float]])->[[[float]]]:
    return [[f(v[i][j]) for j in range(len(v[i]))] for i in range(len(v))]
```

11. L'appel `transforme(zzz, v)` divise les abscisses de tous les points par 2, sans changer les ordonnées ; ou autrement dit effectue une homothétie de rapport $\frac{1}{2}$ selon l'axe des abscisses. Les glyphes sont « compressés » horizontalement.

12.

```
def f(p:[float])->[float]:
    return [p[0] + 0.5 * p[1], p[1]]
```

```
def penche(v:[[[float]])->[[[float]]]:
    return transforme(f, v)
```

13. Ici, `dx = 6` et `dy = 2`. Les pixels encrés ont pour coordonnées (0,0), (1,0), (2,1), (3,1), (4,1), (5,2), (6,2).

14. Ici, `dx = -8` et `dy = 1`. Le seul pixel encré a pour coordonnées (9,8) (on n'entre jamais dans la boucle `for`). Le problème est que `dx` est strictement négatif. Pour éviter ce problème, on peut ajouter `assert dx >= 0` après la ligne 7.

15. Ici, `dx = 2` et `dy = 8`. Les pixels encrés ont pour coordonnées (3,0), (4,4), (5,8). On n'obtient pas une ligne continue mais seulement trois pixels non adjacents. Le problème vient du fait que `dy > dx`.

16.

```
def trace_quadrant_sud(im:img, p0:(int), p1:(int)):
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1-x0, y1-y0
    assert dy >= 0
    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)
```

17.

```
def trace_segment(im:img, p0:(int), p1:(int)):
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1-x0, y1-y0
    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)
```

18.

```
def position(p:(float), pz:(int), taille:int)->int:
    x, y = p
    zx, zy = pz
    return (zx + floor(x * taille), zy + floor(y * taille))
```

19.

```
def affiche_car(page:Image, c:str, police:str, roman:bool, pz:(int), taille:int)->int:
    v = glyphe(c, police, roman)
    for multi_ligne in v:
```

```

    positions = [position(p, pz, taille) for p in multi_ligne]
    for i in range(len(positions)-1):
        trace_segment(page, positions[i], positions[i+1])
    return largeur(v) * taille

```

20. `def affiche_mot(page:Image, mot:str, ic:int, police:str, roman:bool)->int:`
 `for c in mot:`
 `largeur = affiche_car(page, c, police, roman, pz, taille)`
 `pz = pz + largeur + ic`
 `return pz - ic`

21. L'algorithme glouton consiste simplement à ajouter les mots sur la ligne tant qu'il y a de la place, avec un espace minimal entre deux mots, puis à passer à la ligne quand le mot considéré est trop long. L'algorithme est dit glouton car à chaque étape, on fait ce qui semble le mieux à l'instant présent sans considérer ce qu'il va se passer ensuite : on ne prend pas en compte les mots suivants.

22. • Découpage a) :
- Ligne 1 : $\text{cout}(0, 2) = (10 - 2 - 2 - 4 - 2)^2 = 0$
 - Ligne 2 : $\text{cout}(3, 3) = (10 - 0 - 6)^2 = 16$
 - Ligne 3 : $\text{cout}(4, 4) = (10 - 0 - 6)^2 = 16$
- Découpage b) :
- Ligne 1 : $\text{cout}(0, 1) = (10 - 1 - 2 - 4)^2 = 9$
 - Ligne 2 : $\text{cout}(2, 3) = (10 - 1 - 2 - 6)^2 = 1$
 - Ligne 3 : $\text{cout}(4, 4) = (10 - 0 - 6)^2 = 16$

La somme des coûts par ligne vaut 32 pour le découpage a) et 26 pour le découpage b) : le découpage b) est plus harmonieux.

23. `def progd_memo(i:int, lmots:[int], L:int, memo:{int:int})->int:`
 `if i in memo:`
 `return memo[i]`
 `elif i == len(lmots):`
 `return 0`
 `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`
 `memo[i] = mini`
 `return mini`

24. Déjà, l'appel à $\text{cout}(i, j-1, \text{lmots}, L)$ effectue $j - i + 3$ additions et soustractions.

- Algorithme récursif naïf : notons $c_i(n)$ le nombre d'additions et soustractions effectués par l'appel à $\text{algo_récursif}(i, \text{lmots}, L)$. On a $c_n(n) = 0$ et pour tout $i \in \llbracket 0, n-1 \rrbracket$, $c_i(n) = 2 + \sum_{j=i+1}^n (c_j(n) + j - i + 3)$. On a donc clairement pour tout $i \in \llbracket 0, n-1 \rrbracket$, $c_i(n) \geq d_i(n)$ où $d_i(n)$ est définie par $d_n(n) = 1$ et $d_i(n) = 1 + \sum_{j=i+1}^n d_i(n)$.

Or pour tout i , $d_i(n) = 2^{n-i}$. Ainsi, la complexité temporelle associée à l'algorithme récursif naïf est $c_0(n) \geq d_0(n) = 2^n$: elle est (au moins) exponentielle.

- Programmation dynamique de bas en haut : il y a deux boucles imbriquées, soit $O(n^2)$ itérations. Chaque itération a une complexité en $O(n)$. Tout le reste se fait en temps constant. La complexité est donc $O(n^3)$.

En pratique, n vaut au moins quelques dizaines donc il est nécessaire d'utiliser l'algorithme de programmation dynamique au lieu de l'algorithme récursif naïf.

```
25. def lignes(mots:[str], t:int, L:int):
    l = []
    i = 0
    while i < len(mots):
        l.append(mots[i:t[i]])
        i = t[i]
    return l

26. def formatage(lignesdemots:[[str]], L:int)->str:
    texte = ""
    for ligne in lignesdemots:
        nb_mots = len(ligne)
        if nb_mots == 1:
            texte = texte + ligne[0] + " "*(L-len(ligne[0])) + "\n"
        else:
            espace_dispo = L - sum(len(ligne[i]) for i in range(nb_mots))
            espace_par_mot = espace_dispo // (nb_mots - 1)
            r = espace_dispo % (nb_mots - 1)
            for i in range(r):
                texte = texte + ligne[i] + " "*(espace_par_mot+1)
            for i in range(r, nb_mots):
                texte = texte + ligne[i] + " "*espace_par_mot
            texte = texte + "\n"
    return texte
```

Commentaires

- **Polices de caractères vectorielles.** La représentation vectorielle des glyphes proposée par le sujet est rudimentaire puisqu'elle permet uniquement de tracer des droites. En pratique, on peut aussi tracer des courbes représentées par des courbes de Bézier. Mais le principe général des polices vectorielles est celui présenté dans l'énoncé.
- **Justification d'un paragraphe.** Ici aussi, l'algorithme présenté dans le sujet est rudimentaire. En particulier, il ne prend pas en compte la possibilité d'ajouter des césures (trait d'union qui coupe un mot en fin de lignes). Certains algorithmes de justification récents jouent aussi sur l'espacement entre les caractères pour obtenir un résultat plus satisfaisant.