Corrigé du DS n°1
D'après ITC Mines Ponts MP-PC-PSI - 2023

La typographie informatisée

Partie I – Préambule

Q1 Le montant versé par Donald Knuth sera de :

$$m = 1 \times 16^2 + 0 \times 16^1 + 0 \times 16^0 = 256 \text{ cents} = 2{,}56 \text{ \$}$$

Q2 On obtient le caractère j minuscule :

Partie II – Gestion de polices de caractères vectorielles

Q3 On peut proposer :

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

Q4 On a :

```
SELECT gdesc FROM Glyphe JOIN Caractere
                            ON Glyphe.code = Caractere.code
                            JOIN Police
                            ON Glyphe.pid = Police.pid
WHERE Caractere.car = 'A' AND Police.nom = 'Helvetica' AND groman = False
```

Q5 On peut proposer :

```
SELECT fnom, COUNT(pnom)  FROM Famille JOIN Police
                                      ON Famille.fid = Police.fid
GROUP BY fnom
ORDER BY fnom
```

Partie III – Manipulation de descriptions vectorielles de glyphes

Q6 On a :

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

Q7 On a :

```
def dim(l,n) :
  assert n >=0 and n <= 1
  L = []
  for p in l :
    L.append(p[n])
  return L
```

Q8 On a :

```
def largeur(v) :
  Liste_points = points(v)
  Liste_abscisses = dim(Liste_points,0)
  abscisse_max = max(Liste_abscisses)
  abscisse_min = min(Liste_abscisses)
  return max - min
```

Q9 On peut proposer :

```
def obtention_largeur(police) :
  alphabet1 = ["a", "b","c","d","e","f","g","h","i","j","k","l","m","n"]
  alphabet2 = ["o","p","q","r","s","t","u","V","w","x","y","z"]
  alphabet = alphabet1 + alphabet2
  L = []
  for car in alphabet :
    v_rom = glyphe(car, police, True)
    v_ital = glyphe(car, police, False)
    L.append( largeur(v_rom) )
    L.append( largeur(v_ital) )
  return L
```

Q10 On a :

```
def transforme(f,v) :
  nouv_v = []
  for multiligne in v :
    L = []
    for p in multiligne :
      L.append(f(p))
    nouv_v.append(L)
  return nouv_v
```

Q11 L'abscisse de chaque point est divisée par 2 mais son ordonnée n'est pas changée. Cela va donc diviser la largeur du glyphe par 2 sans changer sa hauteur.

Q12 On peut proposer :

```
def penche(v) :
  def f(p) :
    return [ p[0] + 0.5*p[1], p[1] ]
  return transforme(f,v)
```

Partie IV – Rasterisation

Q13 Ligne 15 : on a $x_0 = 0$, $y_0 = 0$, $x_1 = 6$ et $y_1 = 2$. Il s'ensuit que $\mathrm{d}x = 6$ et $\mathrm{d}y = 2$. L'ordonnée de chaque pixel p dans la boucle `for` ligne 9 est :

$$y = \mathrm{floor}(0.5 + i/3) \quad \text{pour } i \text{ variant de 1 à 5}$$

ce qui donne 7 pixels en tenant compte de p_0 et p_1. Leurs coordonnées entières sont données ci-dessous :

$$(0,0)\ (1,0)\ (2,1)\ (3,1)\ (4,1)\ (5,2)\ (6,2)$$

Q14 Aucun pixel de la boucle `for` ligne 9 ne sera affiché car $x_0 = 9$ et $x_1 = 8$, ce qui donne $\mathrm{d}x = -1$. En effet, des instructions du type :

```
for i in range(a,b) : ...
```

ne sont exécutées que si les deux entiers vérifient $b > a$.

On peut conclure que, dans le cas général où $\mathrm{d}x \leqslant 0$, seuls les pixels p_0 et p_1 seront dessinés.

Remarques :

- Dans le cas particulier où $\mathrm{d}x = 1$, la boucle `for` n'est pas exécutée non plus à cause du `range(1,1)`, **mais ce n'est pas un problème** ici car il n'y a alors que deux pixels à tracer, à savoir p_0 et p_1, ce qui est fait aux lignes 8 et 12.
- Le cas particulier où $\mathrm{d}x = 0$ (ligne verticale) devra donc être traité à part.

L'énoncé propose de placer une assertion à placer au début de la fonction, ce qui pourrait inciter les candidats à écrire :

```
assert p0[0] < p1[0]
```

Ce code assure que le programme ne sera exécuté que si $x_0 < x_1$. Cependant, cela ne résout pas vraiment le problème car dans le cas contraire le programme s'arrête immédiatement avec une erreur d'exécution, ce qui n'est pas non plus ce qu'on recherche.

Le mieux est de modifier légèrement la fonction en permutant les rôles de p_0 et p_1 dans le cas où $x_0 > x_1$. On obtient alors le code suivant :

```
1  def trace_quadrant_est(im:img, p0:tuple, p1:tuple) :
2    if p0[0] > p1[0] :
3      p0,p1 = p1,p0
4    x0, y0 = p0
5    x1, y1 = p1
6    dx, dy = x1-x0, y1-y0
7    im.putpixel(p0, 0)
8    for i in range(1,dx) :
9      p = (x0 + i, y0 + floor(0.5 + dy * i / dx))
10     im.putpixel(p, 0)
11   im.putpixel(p1, 0)
```

Q15 Ligne 17 : on a maintenant $x_0 = 3$, $y_0 = 0$, $x_1 = 5$ et $y_1 = 8$. Il en résulte que $\mathrm{d}x = 2$ et $\mathrm{d}y = 8$. L'ordonnée du pixel p dans la boucle `for` ligne 9 vérifie :

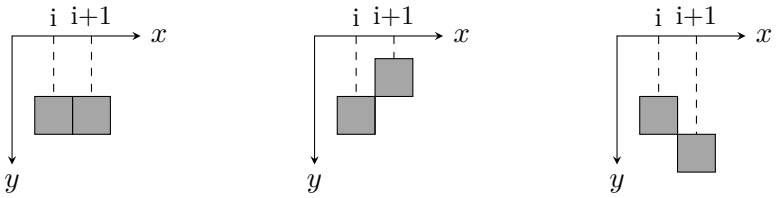
$$y = \mathrm{floor}(0.5 + 4 * i),\ \ i \text{ ne prenant que la valeur } 1$$

Ceci donne 3 pixels en tenant compte de p_0 et p_1, à savoir :

$$(3,0)\ (4,4)\ (5,8)$$

Ici, le problème est que ces 3 pixels ne se "touchent" pas. L'effet visuel ne sera pas celui d'une ligne d'un seul tenant mais plutôt un effet d'une ligne en pointillés. C'est la pente du segment qui est trop grande et il vaut mieux dans ce cas faire une boucle faisant varier les ordonnées par pas de 1 plutôt que les abscisses.

Pour que les pixels puissent se toucher, on doit se trouver dans un des trois cas ci-dessous :



Il faut donc que la pente $\mathrm{d}y/\mathrm{d}x$ soit strictement comprise entre -1 et 1. Afin de ne pas avoir à gérer le cas $\mathrm{d}x = 0$, ni les signes de $\mathrm{d}x$ et de $\mathrm{d}y$, on peut reformuler cette condition par $|\mathrm{d}y| \leqslant |\mathrm{d}x|$.

Q16 La boucle `for` se fera sur l'ordonnée y. Afin de gérer correctement le cas où $\mathrm{d}y < 0$ (cas analogue à celui de la question **Q14**) on permute p_0 et p_1 dans le cas où `p0[1] > p1[1]`. On propose :

```
1  def trace_quadrant_sud(im:img, p0:tuple, p1:tuple) :
2    if p0[1] > p1[1] :
3      p0,p1 = p1,p0
4    x0, y0 = p0
5    x1, y1 = p1
6    dx, dy = x1-x0, y1-y0
7    im.putpixel(p0,0)
8    for j in range(1,dy) :
9      p = (floor(0.5 + dx * j/dy), y0 + j)
10     im.putpixel(p,0)
11   im.putpixel(p1,0)
```

Q17 On doit tenir compte de tous les cas qui peuvent survenir. On a donc :

```
1  def trace_segment(im:img, p0:tuple, p1:tuple) :
2    if p0 == p1 :
3      im.putpixel(p0,0)
4    else :
5      x0,y0 = p0
5      x1,y1 = p1
6      dx,dy = x1-x0,y1-y0
7      if dx == 0 : trace_quadrant_sud(im,p0,p1) :
8      elif dy == 0 : trace_quadrant_est(im,p0,p1) :
9      elif abs(dy) <= abs(dx) : trace_quadrant_est(im,p0,p1) :
10     else : trace_quadrant_sud(im,p0,p1) :
```

Partie VI – Justification d'un paragraphe

Q21 La fonction `glouton` examine les mots de `lmots` les uns à la suite des autres et en place le maximum sur chaque ligne. Il est glouton parce qu'il recherche un optimum local (ie

le maximum de mots dans une ligne) au fur et à mesure de sa progression, sans jamais revenir sur les lignes (étapes) déjà traitées.

Q22 **Découpage a)** :

Ligne 1 : $i = 0$, $j = 2$; cout(0,2)=$(10\text{-}2\text{-}8)^2$=0

Ligne 2 : $i = 3$, $j = 3$; cout(3,3)=$(10\text{-}0\text{-}6)^2$=16

Ligne 3 : $i = 4$, $j = 4$; cout(4,4)=$(10\text{-}0\text{-}6)^2$=16

Le coût total est donc égal à 32.

Découpage b) :

Ligne 1 : $i = 0$, $j = 1$; cout(0,1)=$(10\text{-}1\text{-}6)^2$=9

Ligne 2 : $i = 2$, $j = 3$; cout(2,3)=$(10\text{-}1\text{-}8)^2$=1

Ligne 3 : $i = 4$, $j = 4$; cout(4,4)=$(10\text{-}0\text{-}6)^2$=16

Le coût total est donc égal à 26.

L'algorithme de programmation dynamique est donc celui qui donne un coût total moindre.

Q23 Question classique du cours de programmation dynamique et facile pour les candidats qui ont travaillé ce point. Les clés du dictionnaire `memo` sont les indices i des mots dans lmots et ses valeurs sont les $d(i)$. Dans la méthode de mémoïsation, il est nécessaire de commencer par examiner si le calcul d'un $d(i)$ n'a pas déjà été fait avant de lancer la récursion ; si ce n'est pas le cas, la récursion est réalisée mais on n'oublie pas de noter le résultat dans le diictionnaire avant de le renvoyer. Il vient :

```
1  def progdmemo(i:int, lmots:[int], L:int) -> int :
2    if i in memo.keys() :
3      return memo[i]
4    elif i == len(lmots) :
5      memo[i] = 0
6      return 0
7    else :
8      mini = float("inf")
9      for j in range(i+1,len(lmots)+1):
10       d = progdmemo(j, lmots, L, memo) + cout(i, j-1, lmots, L)
11       if d < mini :
12         mini = d
13     memo[i] = mini
14     return mini
```

Q24 Cette question est **tout sauf simple**, surtout pour des candidats peut rompus aux calculs de complexité. Un seule conseil : sauter la question ! !

Il faut d'abord évaluer la complexité de la fonction `cout(i,j,lmots,L)`, en ne tenant compte que des additions et des soustractions : on attribue donc 1 point à chacune de ces opérations. Cette fonction réalise $3 + (j - i + 1) = 4 + j - i$ opérations donc sa complexité est

$$T_{\text{cout}(i,j)} = 4 + j - i$$

- Pour l'algorithme récursif naïf :

  Il calcule le coût total minimal lorsqu'on lance `algo_recursif(0,lmots,L)` (ce qui correspond à $i = 0$: on travaille sur le paragraphe commençant au mot d'indice 0 et se terminant au mot d'indice $n-1$). Soit $T(0)$ sa complexité qu'on cherche à calculer.

  Notons $T(i)$ la complexité de `algo_recursif(i,lmots,L)` et cherchons la relation entre $T(i)$ et $T(i-1)$.

  `algo_recursif(i-1,lmots,L)` exécute la boucle `for` de la ligne 6 une fois de plus que `algo_recursif(i,lmots,L)` et comme c'est cette boucle qui va donner la complexité en opérations +/- on remarque que :

$$T(i-1) = T(i) + T_{\text{cout}(i-1,i-1)} + T(i) = 2\,T(i) + 4$$

  On a donc :

$$T(i) = \frac{1}{2}\,T(i-1) - 2$$

  Cherchons une solution constante : $x = \frac{x}{2} - 2$, ce qui donne $x = -4$. Posons alors : $T(i) = U(i) + x = U(i) - 4$. La relation de récurrence sur $U(i)$ s'écrit :

$$U(i) = T(i)+4 = \frac{1}{2}\,T(i-1)-2+4 = \frac{1}{2}\,T(i-1)+2 = \frac{1}{2}\,U(i-1)-2+2 = \frac{1}{2}\,U(i-1)$$

  ce qui implique :

$$U(n) = \frac{1}{2}\,U(n-1) = \frac{1}{2^2}\,U(n-2) = \ldots = \frac{1}{2^n}\,U(0)$$

  Ceci permet de trouver la relation entre $T(0)$ et $T(n)$:

$$T(0) + 4 = 2^n\ (\,T(n) + 4)$$

  Or $T(n) = 0$ puisque dans ce cas, seules les lignes 2 et 3 sont exécutées, sans aucune opération +/-. Il s'ensuit que :

$$\boxed{T(0) = 2^{n+2} - 4 = O(2^n)}$$

  On a affaire à une complexité exponentielle, qui n'a rien de mirifique, mais on sait qu'on n'a rien à attendre d'un algorithme récursif naïf en programmation dynamique (c'est l'intérêt de la mémoïsation).

- Pour l'algorithme itératif (de bas en haut) :

  La complexité est due principalement à la double boucle `for`. Chaque itération de la boucle sur j donne lieu à $5 + j - i$ opérations additives ou soustractives, j variant de i à n. On a donc pour cette boucle centrale ligne 5 :

$$\text{nombreOp} = \sum_{j=i}^{n}(5 + j - i) = 5\,(n - i + 1) + \sum_{k=0}^{n-i} k$$

  (dans le dernier terme on a posé $k = j - i$). On a donc :

$$\text{nombreOp} = 5\,(n - i + 1) + \frac{(n-i)(n-i+1)}{2}$$

Maintenant, dans la première boucle `for`, i varie de $n-1$ à 0, ce qui fait que le nombre total d'opérations satisfait à l'équation :

$$\text{nombreTotalOp} = \sum_{i=0}^{n-1} \left(5\,(n-i+1) + \frac{(n-i)(n-i+1)}{2} \right)$$

On pose alors $k = n - i$ et on transforme la somme précédente en :

$$\text{nombreTotalOp} = \frac{1}{2} \sum_{k=1}^{n} (k+1)\,(k+10)$$

En ne retenant que les termes prépondérants dans cette somme, c'est à dire $\sum_{k=1}^{n} k^2$, on obtient un nombre total d'opérations en $\mathrm{O}(n^3)$, ce qui est un peu mieux que la complexité exponentielle mais qui n'a rien d'extraordinaire non plus.

Q25 On revient à quelquechose d'abordable. On peut proposer la fonction ci-dessous :

```
1  def lignes(mots,t) :
2      i = 0
3      res = []
4      while i < n :
5          UneLigne = []
6          for j in range(i,t[i]+1) :
7              UneLigne.append(mots[j])
8          res.append(UneLigne)
9          i = t[i] + 1
10     return res
```

Q26 Dernière question du problème : difficile et ouverte (sûrement pour occuper les candidats arrivés jusqu'à ce point). Soit p le nombre de mots d'une ligne donnée.

- Si $p = 1$, on place le mot en début de ligne et, dans le cas où il reste de la place, on complète par des caractères d'espacement pour avoir au total L caractères. C'est ce qui arrive avec "veniam" dans l'exemple de l'énoncé.
- Si $p > 1$, alors il y a au minimum $p-1$ caractères d'espacement qui viennent se placer après chaque mot, du premier jusqu'à l'avant-dernier. Si `longueur` est le nombre total de caractères de tous les mots d'une ligne, alors il y a éventuellement $L-(p-1)-$ longueur caractères d'espacement supplémentaires à placer de façon la plus équitable possible derrière chacun de $p-1$ premiers mots suivis de leur caractère d'espacement.

  Le mieux est peut-être de faire *un tirage au sort avec une loi uniforme* pour attribuer ces derniers caractères d'espacement à chacun des $p-1$ premiers mots. Dans ce but :

  * Si on a $L-(p-1)-$ longueur $\geqslant 0$ caractères d'espacement supplémentaires à répartir, on construit une liste `L` de $p-1$ éléments entiers donc les indices correspondent aux $p-1$ premiers mots de la ligne étudiée et on la remplit initialement de 0.
  * On réalise $L-(p-1)-$ longueur tirages au sort dans l'ensemble $\{0, 1, ..., p-2\}$. Si i est le résultat d'un tirage au sort, on augmente `L[i]` d'une unité.

* À la fin des tirages au sort, `L[i]` contient le nombre de caractères d'espacement supplémentaires à placer derrière le mot d'indice i dans la ligne.

Remarque :

L'algorithme de programmation dynamique nous assure que $L-(p-1)-\text{longueur} \geqslant 0$ (il est éventuellement égal à 0).

On peut donc proposer :

```
    import numpy as np

1   def formatage(lignesdemots,L) :
2     ch = ""      # chaîne à construire
3     for ligne in lignesdemots :   # parcourt de toutes les lignes
4       p = len(ligne)
5       if p == 1 :
6         ch += ligne[0] + (L-len(ligne[0])*" " + "\n"
7       else :
8         longueur = 0   # nombre total de caractères des mots de la ligne
9         for mot in ligne :
10          longueur += len(mot)
11        nombreEspacesSupp = L -(p-1)-longueur
12        if nombreEspacesSupp > 0 :
13          L = (p-1)*[0]
14          for k in range(nombreEspacesSupp) :
15            i = int( np.random.uniform(0,p-2) )
16            L[i] += 1
17        for i in range(0,p-1) :
18          ch += ligne[i] + " " + L[i]*" "
19        ch += ligne[p-1] + "\n"
20      return ch
```