La typographie informatisée

Mines-Ponts 2023

Bon sujet à donner en DS, avec des BDD et de la programmation dynamique. Mais il est bien trop long pour deux heures.

Attention petite erreur d'énoncé dans Q23. Remplacer `memo = {len(m): 0}` par `memo = {len(lmots), 0}`.

La programmation dynamique à partir de Q23 est intéressante, mais je la trouve mal expliquée (peut-être même l'explication donnée est-elle fausse). Dans mon corrigé, j'ai écrit une nouvelle explication et il faudrait modifier l'énoncé pour la faire apparaître. J'ai aussi rédigé un énoncé de TD de programmation dynamique avec ces idées. On peut même demander la démo par l'absurde de la propriété de sous-structure optimale.

La dernière question (Q26) est très longue. Je répète : l'ensemble est bien trop long pour deux heurs.

Q1. La nombre représenté en hexadécimal par 100 est $16^2 = 256$. Le montant versé est de 256 cents, c'est à dore 2,56$.

Q2. En traçant les lignes à partir des points indiqué, on voit se former la lettre j.

Q3.
```
SELECT COUNT(*) FROM Glyphe WHERE groman = TRUE
```

Q4.
```
SELECT gdesc FROM Glyphe JOIN Caractere
ON Glyphe.code = Caractere.code
JOIN Police ON Police.pid = Glyphe.pid
WHERE pnom = 'Helvetica' AND Caractere.car = 'A'
AND groman = False
```
4

Q5.
```
SELECT fnom, COUNT(*) FROM Famille JOIN Police
ON Famille.fid = Police.fid
GROUP BY fnom
ORDRE BY fnom
```

Q6.
```
def points(v) :
    liste_points = []
    for ligne in v :
        for point in ligne :
            liste_points.append(point)
    return liste_points
```
6

Q7.
```
def dim(l, n) :
    L = []
    for sous_liste in l :
        L.append(sous_liste[n])
    return L
```
4

Q8.
```
def largeur(v) :
    les_points = points(v)
    abscisses =  dim(les_points, 0)
    return max(abscisses) - mins(abscisses)
```

Q9.
```
def obtention_largeur(police) :
```

```
    alphabet = 'abcdefghijklmnopqrstuvwxyz'
    liste_largeurs = []
    for c in alphabet :
        vro = glyphe(c, police, True)
        vit = glyphe(c, police, False)                                    6
        liste_largeurs.append(largeur(vro))
        liste_largeurs.append(largeur(vit))
    return liste_largeurs
```

Q10.
```
def transforme(f, v) :
    nv = []
    for ligne in v :
        n_ligne = []
        for p in ligne :
            n_ligne.append(f(p))                                          6
        nv.append(ligne)
    return nv
```

Q11. On divise par deux l'abscisse de chaque point sans modifier son ordonnée. Le glyphe est donc donc contracté horizontalement pour devenir deux fois moins large.

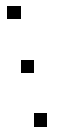
Q12.
```
def penche_point(p):
    x, y = p                                                              2
    return [x + 0.5*y, y]

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

Q13. On encre les pixels (0,0), (1,0), (2,1), (3,1), (4,1), (5,2) et (6,2) et cela donne le résultat ci-dessous.

Q14. Avec les arguments utilisés ligne 16, $dx = -0$ et l'itération prévue ligne 9 n'aura lieu aucune fois, de sorte que la fonction ne tracera rien du tout. On peut ajouter après la ligne 7 : `assert dx > 1`. Néanmoins, la fonction continuera à ne rien tracer ! Elle n'est adapté que si le second point se trouve à droite du premier (à « l'est », d'où le nom de la fonction). Il faudrait en écrire une autre pour un déplacement vers l'ouest, mais ce n'est pas demandé.

Q15. Ici, $dx = 2$ Seuls trois pixels sont encrés : (3,0), (4,4), (5,8). En plus des deux extrémités, seul un point intermédiaire est produit par l'itération. Le résultat visible ci-dessous montre un ligne discontinue insatisfaisant pour la rastérisation. Le problème est que l'écart d'abscisse dx est trop petit par rapport à dy. La fonction convient pour des déplacements orientés principalement vers l'est, mais pas vers le sud.



Q16. Par rapport à la fonction `trace_quadrant_est`, il suffit de permuter les rôles de l'abscisse et de l'ordonné.

```
def trace_quadrant_sud(im, p0, p1):
    x0, y0 = p0
    x1, y1 = p1
    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. Le plan est partagé en quatre quadrants par les bissectrices des axes d'équation $dy = \pm dx$. Pour savoir vers quel point cardinal le déplacement est le plus orienté, on compare $|dy|$ à $|dx|$, puis on examine le signe de dx ou de dy. Pour traiter les déplacements vers l'ouest et le nord, il suffit d'inverser les rôles des deux points dans les fonctions concernant l'est et le sud.

```
def trace_segment(im, p0, p1) :
    x0, y0 = p0
    x1, y1 = p1
    dx, dy = x1 - x0, y1 - y0
    if abs(dx) >= abs(dy) :
        if dx >= 0 : # est
            trace_quadrant_est(im, p0, p1)
        else : # ouest strict
            trace_quadrant_est(im, p1, p0)
    else :
        if dy >= 0 : # sud
            trace_quadrant_sud(im, p0, p1)
        else : # nord strict
            trace_quadrant_sud(im, p1, p0)
```

Q18. Il convient de décaler les coordonnées de p pour que $(0, 0)$ vienne en pz, et de multiplier les hauteurs par `taille` pour que une hauteur d'œil unitaire prenne la hauteur `taille`. Attention : un lettre vers le haut correspond à une diminution de l'ordonné du pixel.

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

Q19.

```
def affiche_car(page, c, police, roman, pz, taille) :
    v = glyphe(c, police, roman)
    for ligne in v :
        if len(ligne == 0) : # ligne d'un seul point
            p0 = ligne[0]
            p0im = position(0)
            trace_segment(page, p0im, p0im)

        for i in range(len(ligne)-1) :
            p0, p1 = ligne[i], ligne[i+1]
            p0im = position(p0, pz, taille)
            p1im = position(p1, pz, taille)
            trace_segment(page, p0im, p1im)
```

```
    return int(largeur(v)*taille)
```

Q20.
```
def affiche_mot(page, mot, ic, police, roman, pz, taille) :
    qz = pz # pour la lisbilité
    for c in mot :
        d = affiche_car(page, c, police, roman, qz, taille)
        # on déplace le point de base avant le caractère suivant
        qz = (qz[0] + d + ic, qz[1])
    return qz[0] - ic, qz[1]
```

Q21. Le principe est ici de placer les mots à la suite les uns des autres en tenant à jour la longueur occupée sur la ligne (ligne 11). Si l'ajout d'un nouveau mot ferait dépasser la longueur maximale L (ligne 6), on inscrit la ligne en cours (ligne 7) et on en crée un nouvelle qui débute par ce nouveau mot (ligne 8). Il s'agit d'un algorithme glouton puisqu'on regarde à chaque étape ce qui convient mais, une fois la décision de passer à la ligne prise, on ne la remettra pas en question en analysant les mots suivants, même si cela pourrait conduit à une solution plus avantageuse pour l'ensemble du texte.

Q22. Le coût peut être calculé pour tout couple (i, j) avec $j \geqslant (i)$, mais il convient ici de l'appliquer d'un bout à l'autre de chaque ligne.

— Découpage a)

  — ligne 1 : $(i, j) = (0, 2)$, $\Sigma = 8$, $C(0, 2) = 0$
  — ligne 2 : $(i, j) = (3, 3)$, $\sigma = 6$, $C(3, 3) = 4^2$
  — ligne 3 : $(i, j) = (4, 4)$, $\sigma = 6$, $C(4, 4) = 16^2$

  Le coût total est de 32.

— Découpage b)

  — ligne 1 : $(i, j) = (0, 1)$, $\Sigma = 7$, $C(0, 1) = 3^2$
  — ligne 2 : $(i, j) = (2, 3)$, $\Sigma = 9$, $C(2, 3) = 1^2$
  — ligne 3 : $(i, j) = (4, 4)$, $\Sigma = 6$, $C(4, 4 = 16^2$

  Le coût total est de 26.

Avec le critère adopté, le découpage b) est plus harmonieux.

Programmation dynamique

Je trouve que l'idée de programmation dynamique est mal expliquée dans l'énoncé et je vais tenter de la clarifier. Le coût d'un découpage est défini comme la somme des coûts de chacune de ses lignes, ce que l'énoncé ne dit pas clairement mais qu'on comprend à la lecture de la question 22 où le calcul de la somme est demandé. L'énoncé appelle $d(i)$ « le problème de placement optimal de changement de ligne jusqu'à l'indice i ». En réalité, $d(i)$ est un nombre, et plus précisément : *$d(i)$ est le coût optimal du placement du texte formé des mots de l'indice i jusqu'au dernier, d'indice $n-1$ avec n =* len(lmots). On pose par convention $d(n) = 0$ (il n'y a aucun mot d'indice n ou supérieur). Le but est bien sûr de trouver $d(0)$, c'est à dire le coût optimal pour écrire tous les mots du texte.

Considérons un découpage optimal $\mathcal{D}(i)$ du texte formé des mots de l'indice i jusqu'au dernier, de coût minimal $d(i)$. Dans ce découpage optimal, la première ligne commence par `mots[i]` et se poursuit jusqu'à un certain mot indice $j-1$, la ligne suivante commençant à l'indice j (avec $j > i$). Comme le découpage est défini par une somme sur les lignes, on a

$$d(i) = cout(i, j-1) + d(j) \quad .$$

La propriété de sous-structure optimale s'exprime comme suit.

Dans le découpage optimal $\mathcal{D}(i)$, les mots à partir de l'indice j sont placés de manière optimale.

Supposons en effet qu'il existe une meilleure manière de placer les mots à partir de l'indice j en commençant par `mots[j]`, avec un coût $d'(j) < d(j)$. En disposant les mots d'indices i à $j-1$ comme dans $\mathcal{D}(i)$, puis les suivants de cette nouvelle manière, on obtiendrait un coût $d'(i) = cout(i, j-1) + d'(j) < d(i)$. Mais alors $\mathcal{D}(i)$ ne serait pas optimal. La propriété est donc démontrée par l'absurde.

Lorsqu'on cherche $\mathcal{D}(i)$ et $d(i)$, on ne connaît pas encore j et on envisage donc toutes les possibilités.

— $j = i+1$: on écrit `mots[i]` seul sur une ligne, puis les suivants de manière optimale, $d(i) = cout(i,i) + d(i+1)$;

— $j = i+2$, on écrit `mots[i], mots[i+1]` sur une ligne, puis les suivants de manière optimale, $d(i) = cout(i, i+1) + d(i+2)$;

— $j = i+3$: on écrit `mots[i], mots[i+1], mots[i+2]` sur une ligne, puis les suivants de manière optimale, $d(i) = cout(i, i+2) + d(i+3)$;

— etc, jusqu'à écrire `mots[i], mots[i+1], ... mots[n-1]` sur une ligne, $d(i) = cout(i, n-1) + 0$.

Parmi toutes ces possibilités, c'est la plus avantageuse (ou l'une des plus avantageuses) qu'il faut retenir. À partir de $d(n) = 0$, on peut calculer $d(n-1)$, puis $d(n-2)$, etc, jusqu'à $d(0)$. La programmation dynamique « montante » va donc procéder par indices décroissants, et au contraire le codage récursif conduit à calculer $d(0)$ en à partir de $d(1)$, $d(2)$, $d(3)$, etc.

Q23. La fonction `algo_recursif` peut être appelée pour tout i compris entre 0 et `len(lmots)`, mais il faut l'appeler avec $i = 0$ pour définir le découpage du texte entier. Dans la fonction `progd_bashaut`, on observe d'ailleurs que c'est `M[0]` qui est renvoyé.

Soit n = len(lmots). L'appel pour $i = 0$ déclenche n nouveaux appels pour $j = 1, 2, ..., n$. Chacune d'eux provoque de nouveaux appels pour toutes les valeurs de k suivant chaque valeurs de j. Il n'est pas facile d'établir une relation de récurrence sur la complexité (ce serait un récurrence forte), mais ces multiples appels emboîtés laissent deviner une complexité exponentielle en n.

Dans la version itérative de bas en haut, la double boucle provoque un nombre d'itération de l'ordre de n^2. L'appel à `cout(i, j-1)` a une complexité proportionnelle à $j - i$, d'où finalement une complexité en $O(n^3)$.

Q24. Même avec les exemples fournis, le rôle de la liste `t` n'est pas immédiat et on ne va pas utiliser toutes les valeurs qui y sont inscrites dans cette liste. En effet, un placement optimal s'obtient en plaçant sur une même ligne les mots d'indice i jusqu'à `t[i` exclu ; autrement dit il convient d'ouvrir une nouvelle ligne pour écrire le mot `t[i]`. Pour écrire tout le texte, il faut d'abord utiliser la valeur de `t[0]`, puis certain autres éléments de la liste qui se déduisent les uns des autres.

Dans l'exemple `(["Ut","enim","ad","minima","veniam"],[2,3,4,4,5],10)`, on lit `t[0]=2`, donc passe à la ligne pour le mot d'indice 2 (c'est `ad`). La valeur `t[1]= 3` ne sert à rien ! En repartant de $i = 2$, on lit `t[2]=4`, donc on va à la ligne pour le mot d'indice 4 (`veniam`). On lit ensuite `t[4]=5` ce qui termine l'écriture.

Dans le second exemple, `t[0]=2` (passage à la ligne pour `dolor`), puis `t[2]=5` (passage à la ligne pour `consectetur`), puis `t[5]=6` (passage à la ligne pour `adipiscing`), puis `t[6]=7` (passage à la ligne pour `elit`, puis `t[7] = 9` (passage à la ligne pour `non`), puis enfin `t[9]=11` (c'est terminé).

Q25.
```
def lignes(mots, t, L) :
    lignes = []
    i = 0
    while i < len(mots) :
        ind_coupure = t[i]
        nligne = [mots[k] for k in range(i, ind_coupure)]
        # idem : mots[i:ind_coupure]
```
5

```
        lignes.append(nligne)
        i = ind_coupure
    return lignes
```

Q26. La difficulté réside dans la gestion des blancs excédentaires et dans la manière de les répartir entre les mots. Dans chaque ligne, je dénombre ces blancs excédentaires, je calcule le quotient a et le reste b de la division entière de ce nombre par le nombre d'espace entre mots sur la ligne. J'ajoute a blancs aux premiers espaces, et un de plus aux b premiers.

```
def formatage(lignesdemots, t, L) :
    chaine = ''
    for ligne in lignesdemots :

        if len(ligne) == 1 :
            mot = ligne[0]
            for lettre in mot :
                chaine += lettre
            chaine += '␣' * (L - len(mot)) # blancs à la fin
            chaine += '\n'

        else :
            ##########################################################
            # Ce passage concerne la gestion des blancs excédentaires
            longueur = 0
            n_mots = len(ligne)
            for mot in ligne :
                longueur += len(mot)

            n_excedent = L - longueur - (n_mots - 1)
            a, b = divmod(n_excedent, n_mots-1)
            ##################################################
            for i, mot in enumerate(ligne[:-1]) :
                for lettre in mot :
                    chaine += lettre
                chaine += '␣' + a * '␣' # on intercale des blancs
                if i <= b : # et encore d'autres blancs
                    chaine += '␣'

            for lettre in ligne[-1] : # dernier mot de la ligne
                chaine += lettre
            chaine += '\n'
    return chaine
```