Corrigé du DS 4 sujet Mines 2024

Q1. On peut coder `'a'` par `0`, `'b'` par `10` et `'c'` par `11`. Cela permet de déchiffrer sans ambigüité : si on lit `0`, on déchiffre par `'a'`, si on lit `1` on lit aussi le chiffre suivant pour distinguer entre `'b'` et `'c'`. On a alors la chaîne `s` qui est codée sur 11 bits ; la représentation binaire de `'a'` est bien inférieure à celle de `'b'` elle-même inférieure à celle de `'c'`.

Q2.

```
def nbCaracteres(c, s):
    nb = 0
    for x in s:
        if x == c:
            nb = nb + 1
    return nb
```

Q3. `listeCaracteres('abaabaca')` renvoie la liste `['a', 'b', 'c']`. La boucle permet de parcourir les caractères de la chaîne `s`, on teste alors si le caractère considéré a déjà été rencontré et enregistré dans la liste `listeCar` ; si ce n'est pas le cas on l'ajoute à la liste. Ainsi en fin de boucle la liste renvoyée contient chaque caractère utilisée dans `s` une seule fois et dans l'ordre d'apparition.

Q4. Le pire des cas correspond à une liste dont les `k` premiers caractères sont distincts ; la liste `listeCar` est alors de longueur maximale ce qui maximise la complexité du test d'appartenance.

Les lignes 3 et 4 sont de cout constant (3 opérations).
Pour le tour de boucle `i` $\in [\![0\,;n-1]\!]$, la liste `listeCar` est de longueur `i` si `i` $\leqslant$ `k` de longueur `k` sinon. Le tour de boucle `i` contient donc $4+\min($ `i, k`$)$ opérations élémentaires.

Le nombre d'opérations est donc :

$$3+\sum_{i=0}^{k-1}(4+i)+\sum_{i=k}^{n-1}(4+k)=O(k^2)+O(n\times k)=O(n\times k)$$

car $k\leqslant n$.

Donc : la complexité de `listeCaracteres` est en $O(n\times k)$.

Solution 2 : un petit peu moins précis mais acceptable
Les lignes 3 et 4 sont de cout constant (3 opérations).
Pour chaque tour le boucle, `listeCar` est de longueur inférieure à `k`, donc le nombre d'opération est inférieur à 4+ `k`. Il y a n tours de boucle. Donc le nombre d'opérations est inférieur à

$$3+\sum_{i=0}^{n-1}(4+k)=O(n\times k).$$

Q5. La fonction renvoie la liste des couples constitués d'un caractère de `s` et du nombre d'occurence de ce caractère ; les couples sont donnés dans l'ordre de première apparence du caractère.

La valeur renvoyée pour l'exemple donné est : `[('b', 3), ('a', 6), ('c', 1)]`.

Q6. On sait que la complexité de `listeCaracteres` est en $O(n\times k)$, soit $M_1\in\mathbb{R}^+$ tel que cette complexité soit inférieure à M_1nk. Et de même, celle de `nbCaracteres` soit inférieure à M_2n.

On a alors avant la boucle de `analyseTexte` au plus $2+M_1nk$ opérations ;
À chaque tour de boucle, au plus $5+M_2n$ opérations et il y a k tours de boucle.

Donc le nombre d'opérations est inférieur à :

$$2+M_1nk+k\times(5+M_1n)=O(n\times k).$$

La complexité de `analyseTexte` est en $O(n\times k)$.

Q7.

```
def analyseTexte(s):
    R = {}
    for c in s:
        if c in R:
            R[c] = R[c] + 1
        else:
            R[c] = 1
    return R
```

Q8.

```
SELECT DISTINCT auteur FROM corpus;
```

Q9.

```
SELECT symbole,
    SUM(nombreOccurences) / SUM(nombreCaracteres)
FROM occurences AS O JOIN caracteres AS CA JOIN corpus AS CO
    ON O.idCar = CA.idCar AND CO.idLivre = O.idLivre
WHERE langue = 'Français'
GROUP BY O.idCar;
```

Q10.
- `'b'` correspond à l'intervalle $[0.2\,;0.3[$;
- `'ba'` correspond à l'intervalle $[0.20\,;0.22[$;
- `'bac'` correspond à l'intervalle $[0.206\,;0.210[$.

Q11.

```
def codage(s):
    g, d = 0, 1
    for c in s:
        g, d = codeCar(c, g, d)
    return g, d
```

Q12. `'ad'` correspond à l'intervalle $[0.10\,;0.18[$, donc :

- `'ada'` correspond à l'intervalle $[0.100\,;0.116[$;
- `'adb'` correspond à l'intervalle $[0.116\,;0.124[$

Donc : le caractère suivant `'ad'` est `'b'` car $0.123 \in [0.116\,;0.124[$.

Q13. • `'b'` correspond à l'intervalle $[0.2\,;0.3[$;

- `'ba'` correspond à l'intervalle $[0.2\,;0.22[$;

donc les chaînes `'b'` et `'ba'` peuvent correspondre au flottant `0.2`. L'ambiguïté vient du fait que rien ne marque la fin d'une chaîne et que si on ajoute une (ou des) lettre(s) à la suite d'une chaîne `s`, on obtient une chaîne `s'` qui correspond à un intervalle qui est inclus dans l'intervalle de `s` et donc tout flottant représentant `s'` représente également la chaîne `s`.

Q14.

```
def decodage(x):
    s = ''
    g, d = 0, 1
    c = ''  # initialisation pour le test
    while c != '#':
        c = decodeCar(x, g, d)
        g, d = codeCar(c, g, d)
        s = s + c
    return s
```

Q15. • Pour chacune des N observations il y a K états possibles.
Donc, le nombre de sommets est : $N \times K$.

- Les arrêtes sont caractérisées par
  - le choix d'un sommet de départ dont le numéro d'observation est entre 1 et $N-1$ et pour chaque observation il y a K états possibles ; donc $(N-1)K$ sommets de départ possible ;
  - pour chaque sommet de départ il y a K sommets d'arrivée possible : chacun des K sommets de l'obervation suivante.

  Donc, le nombre d'arrêtes est : $(N-1)K^2$.

Q16. Les maths sont fausses ; mais vous devez remplir en appliquant les instructions données.

Q17. À chaque étape de σ à obs_{n-1} du chemin il y a K possibilités pour l'étape suivante, une seule possibilité pour obs_{N-1} à τ. Les chemins correspondent aux N-listes d'élémetns de Σ. Donc le nombre de chemins entre σ et τ est K^N.

Un algorithme d'exploration exhaustive aurait une complexité supérieur à $\mathbb{K}^N$ (exponentielle en N) ; ce n'est donc pas envisageable.

Q18.

```
def maximumListe(liste):
    ind = 0
    for j in range(1, len(liste)):
        if liste[j] > liste[ind]:
            ind = j
    return liste[ind], ind
```

Q19. Il y a des fautes de frappe dans le sujet, pour `initialiserGlouton`, le paramètre `Obs` est une lsite d'entiers et non une liste de listes d'entiers. Il faut donc lire `initialiserGlouton(Obs:[int], E:[[float]], K:int)-> int`.

Et dans la définition de cette fonction, en ligne 2 :
`probasInitiales = [E[Obs[0][i]] for i in range(K)]` doit être remplacé par :
`probasInitiales = [E[Obs[0]][i] for i in range(K)]`

```
def glouton(Obs, P, E, K, N):
    ind_sommet = initialiserGlouton(Obs, E, K)
    res = [ind_sommet]
    for j in range(1, N):
        probas = [E[Obs[j]][i] * P[ind_sommet][i] for i in range
            (K)]
        x, indice_sommet = maximumListe(probas)
        res.append(indice_sommet)
    return res
```

Q20. On commence par évaluer la complexité de la fonction `initialiserGlouton` :

- ligne 2 : $O(K)$ opérations
- ligne 3 : l'appel de `maximumListe` coûte $O(K)$ opérations car la liste `probasInitiales` est de longueur K.

Donc : `initialiserGlouton` a une complexité en $O(K)$.

Pour la fonction `glouton`

- avant la boucle, l'appel à `initialiserGlouton` est en $O(K)$, les autres opérations sont de complexité bornée.

- à chaque tour de boucle, la création de la liste `probas` est de complexité en $O(K)$, puis l'appel à la fonction `maximumListe` sur cette liste de longueur K est en $O(K)$. Il y a $N-1$ tours de boucle.

La complexité de `glouton` est donc en $O(N \times K)$.

Q21. Le résultat renvoyé par la fonction `glouton` est `[0, 0]` car à la première étape le maximum est 0.6 ce qui fait choisir le symbole 0, puis à la deuxième étape le maximum est 0.5, donc le symbole est 0. La probabilité de ce chemin est $0.6 * 0.5 * 1 = 0.3$.

Or le chemin `[1, 0]` a une proba : $0.4 * 0.9 * 1 = 0.36$ qui est strictement supérieure. Donc l'approche gloutonne n'est pas optimale.

Q22. On cherche ici à maximiser $p = \prod_{i=0}^{N} p_i$ où p_i est la probabilité attachée à l'arrête choisie, ce qui revient (par stricte croissance sur $\mathbb{R}_+^*$ de la fonction ln) à maximiser

$$\ln(p) = \sum_{i=0}^{N} \ln(p_i)$$

ou à minimiser

$$-\ln(p) = \sum_{i=0}^{N} -\ln(p_i)$$

avec $-\ln(p_i) > 0$, car $p_i \in]0\,;1[$.

Ainsi en attribuant les poids $-\ln(p_i)$ aux arrêtes, on est ramené à un problème de plus court chemin dans un graphe à poids positifs. On peut alors appliquer l'algorithme de Dijkstra.

Q23. Il faut faire attention à l'ordre des deux boucles (comme souvent dans une stratégie bottom up). L'ordre nous est imposé car on a besoin de `T[k][j - 1]` pour tout `k` afin de construire `T[i][j]`.

```
def construireTableauViterbi(Obs, P, E, K, N):
    T, argT = initialiserViterbi(E, Obs[0], K, N)
    for j in range(1, N):
        for i in range(K):
            probas = [E[Obs[j]][i] * P[k][i] * T[k][j - 1] for k
                in range(K)]
            maxi, indice = maximumListe(probas)
            T[i][j] = maxi
            argT[i][j] = indice
    return T
```

Q24. En partant du sommet le plus probable en fin de parcours : $S_{0,7}$ et en remontant suivant `argT`, on obtient le chemin suivant : `[2, 0, 0, 2, 1, 1, 0, 0]`.

Q25. Complexité temporelle :

pour la fonction `initialiserViterbi` :

- ligne 2 : $O(K)$ (liste de longueur K)
- lignes 3 et 4 : $O(N \times K)$ (liste de K liste de longueurs N)
- à chacun des K tours de boucle : 3 opérations ; donc $O(K)$ opérations pour la boucle.

Donc, la fonction `initialiserViterbi` est de complexité en $O(NK)$.

pour la fonction `construireTableauViterbi` :

- l'initialisation : $O(NK)$
- chaque tour de boucle coute $O(K)$ (création de la liste `probas` de longueur K et appel à `maximumListe` sur cette liste, les autres opérations en $O(1)$) et il y a $(N-1) \times K$ tours de boucle.

Donc la complexité temporelle de `construireTableauViterbi` est en $O(N \times K^2)$.

Complexité spaciale :

- les tableaux `T` et `argT` sont de taille $N \times K$
- la liste `probas` est de taille K

Donc la complexité spaciale de `construireTableauViterbi` est en $O(N \times K)$.