1 Compression du message d'Alice : codage arithmétique

Q1. On code `'a'` par `0`, `'b'` par `10` et `'c'` par `11`. Lorsqu'on lit un `0` c'est le caractère `'a'` et lorqu'on lit un `1` on lit le caractère suivant pour savoir si c'est un `'b'` ou un `'c'`. La chaîne `'abaabaca'` est codée par `0 10 0 0 10 0 11 0`.

1.1 Analyse du texte source

Q2.

```
1  def nbCaracteres(c, s) :
2      nb = 0
3      for x in s :
4          if x == c :
5              nb += 1
6      return nb
```

Q3. `listeCaracteres('abaabaca')` renvoie `['a', 'b', 'c']`.

On parcourt la chaîne caractère par caractère, si le caractère courant n'est pas déjà dans la liste `listeCar` alors on l'ajoute à cette liste. En fin de boucle on renvoie la liste `listeCar` qui réprésente alors la liste des caractères utilisés.

Q4. Dans le pire des cas la liste `listeCar` est de longueur `k`, donc les lignes 5 à 7 ont une complexité en `O(k)`. En ajoutant la ligne 4 la complexité est donc en `O(nk)`. Les autres lignes sont de complexité constante donc au total la complexité est en `O(nk)`.

Q5. Cette fonction crée une liste de tuples, chaque tuple correspondant à un des caractères de la chaîne donnée en entrée et au nombre d'occurences de ce caractère.

La commande `analyseTexte('babaaaabca')` renvoie `[('b', 3), ('a', 6), ('c', 1)]`.

Q6. La ligne 5 est en `O(1)` et la ligne 6 en `O(n)` donc les lignes 4 à 6 sont en `O(nk)`. La ligne 2 est en `O(1)` et la ligne 3 en `O(nk)` d'après **Q4.** donc au total la complexité de la fonction `analyseTexte` est en `O(nk)`.

Q7.

```
1  def analyseTexte(s) :
2      d = {}
3      for c in s :
4          if c in d :
5              d[c] += 1
6          else :
7              d[c] = 1
8      return d
```

Comme la commmande `c in d` a une complexité en `O(1)` la complexité de cette fonction analyseTexte est en `O(n)`.

1.2 Exploitation d'analyses existantes

Q8.
```
SELECT DISTINCT auteur FROM corpus ;
```

Q9.
```
SELECT ca.symbole, SUM(oc.nombreOccurences) /
   ( SELECT SUM(nombreCaracteres) FROM corpus WHERE langue = 'Français' )
FROM caractere AS ca
JOIN occurences AS oc
JOIN corpus AS co
ON ca.idCar = oc.idCar AND oc.idLivre = co.idLivre
WHERE langue = 'Français'
GROUP BY ca.symbole ;
```

La sous-requete `SELECT SUM(nombreCaracteres) FROM corpus WHERE langue = 'Français'` donne le nombre total de caractères des textes de tout le corpus.

Le rapport du jury indique : « Plusieurs candidats proposent des sous-requêtes alors que le sujet demande explicitement UNE requête ». La requête ci-dessus n'a donc pas rapporté le nombre maximal de points.

En faisant l'hypothèse que la base recense le nombre d'apparitions de chaque caractère dans tous les livres même si ce nombre est nul, on peut proposer :

```
SELECT ca.symbole, SUM(oc.nombreOccurrences)/SUM(co.nombreCaracteres)
FROM caractere as ca
JOIN occurrences AS oc
JOIN corpus as co
On ca.idCar = oc.idCar AND co.idLivre = oc.idLivre
WHERE langue = 'Français'
GROUP BY car.idCar ;
```

1.3 Compression

Q10. On obtient d'abord l'intervalle $[0.2;0.3[$ correspondant au caractère `'b'` ; le caractère `'a'` détermine alors le sous-intervalle $[0.20;0.22[$; le caractère `'c'` détermine enfin l'intervalle $[0.206;0.21[$.

Q11.

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

1.4 Décodage

Q12. Puisque x est dans l'intervalle $[0.116;0.124[$ le caractère qui suit `'ad'` est `'b'`.

Q13. Les chaînes `'b'` et `'ba'` peuvent correspondre au flottant `0.2`.
Cette ambiguité vient du fait que lorsque le caractère est `'a'`, le sous-intervalle a la même borne de gauche que l'intervalle.

Q14.

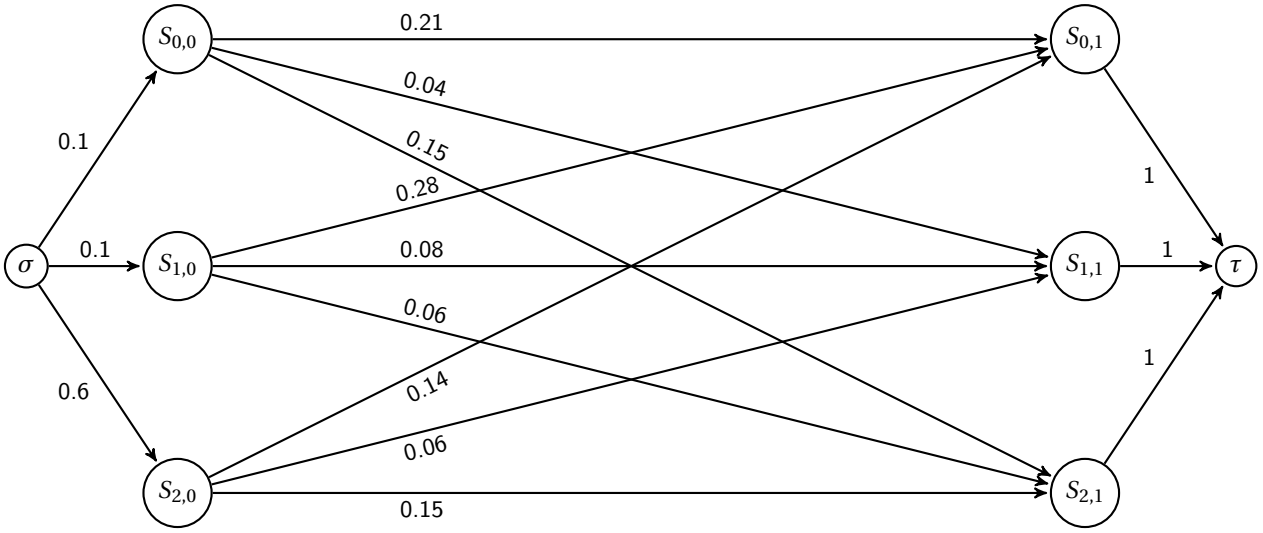
```
def decodage(x) :
    s = ''
    g = 0
    d = 1
    car = decodeCar(x, g, d)
    while car != '#' :
        s = s + car
        g, d = codage(s)
        car = decodeCar(x, g, d)
    s = s + '#'
    return s
```

2 Décodage du message reçu par Bob à l'aide de l'algorithme de Viterbi

2.1 Modélisation du canal de communication par un graphe

Q15. Il y a K strates et N couches donc le nombre total de sommets est NK.
De chaque sommet part K arêtes, sauf pour les sommets de la dernière couche pour lesquels on ne compte pas les arêtes. Donc au total le nombre d'arêtes est $(N-1)K \times K = (N-1)K^2$.

Q16.



Q17. Depuis le sommet σ on a K arêtes, puis pour chaque couche de obs_0 à obs_{N-2} encore K arêtes possibles, puis pour la dernière couche obs_{N-1} seulement une arête qui mène au sommet τ.
Le nombre total de chemin est donc $K \times K^{N-1} = K^N$.
Ce nombre est beaucoup trop grand pour espérer une exploration exhaustive.

2.2 Stratégie gloutonne

Q18.

```
1  def maximumListe(liste) :
2      maxi = liste[0]
3      ind = 0
4      for k in range(1, len(liste)) :
5          if liste[k] > maxi : # strict pour avoir le plus petit indice qui donne le max
6              maxi = liste[k]
7              ind = k
8      return (maxi, ind)
```

Q19. *Erreur d'énoncé c'est* `E[Obs[0]][i]` *et non pas* `E[Obs[0][i]]`.

```
1  def Glouton(Obs, P, E, K, N) :
2      chemin = []
3      i = initialiserGlouton(Obs, E, K)
4      chemin.append(i)
5      for j in range(N-1) :
6          probas = [ E[Obs[j+1]][k]*P[i][k] for k in range(K) ]
7          p, i = maximumListe(probas)
8          chemin.append(i)
9      return chemin
```

Q20. Les lignes 6 et 7 ont une complexité en `O(K)` et la ligne 8 en `O(1)` donc la boucle des lignes 5 à 8 a une complexité en `O(NK)`. Les lignes 1 et 3 ont une complexité en `O(1)` et la ligne 2 en `O(K)`.
Donc au total la complexité de la fonction est en `O(NK)`.

Q21. L'algorithme glouton renvoie le chemin `[0, 0]` de probabilité égale à $0.6 \times 0.5 \times 1 = 0.3$.
Ce n'est pas optimal car le chemin `[1, 0]` est de probabilité égale à $0.4 \times 0.9 \times 1 = 0.36$.

2.3 Stratégie de programmation dynamique

Q22. En remplaçant chaque probabilité p par $-\ln(p)$ (qui est encore positif) comme poids des arêtes, chercher le chemin de probabilité maximale revient à cherche le plus court chemin par rapport à ces poids positifs. On peut donc utiliser l'algorithme de Dijkstra.

Q23. La formule de récurrence donne qu'on doit remplir le tableau `T` colonne par colonne.

```
1  def construireTableauViterbi(Obs, P, E, K, N) :
2      T, argT = initialiserViterbi(E, Obs[0], K, N)
3      for j in range(1, N) :
4          for i in range(K) :
5              liste = [ T[k][j-1] * P[k][i] * E[Obs[j]][i] for k in range(K) ]
6              T[i][j], argT[i][j] = maximumListe(liste)
7      return T, argT
```

Q24. Dans cet exemple `K=3` et `N=8`.
La dernière colonne de `T` donne les probabilités maximales entre l'état source et les états $S_{i,7}$. La plus grande probabilité est obtenue en arrivant au sommet $S_{0,7}$.
Avec le tableau `argT` on obtient la liste des prédecesseurs de 0 : 0, 1, 1, 2, 0, 0, 2.
Le chemin est donc `[2, 0, 0, 2, 1, 1, 0, 0]`.

Q25. ▷ **Complexité temporelle.**
La ligne 2 a une complexité temporelle en `O(NK)` à cause de la création des tableaux `T` et `argT`.
Les lignes 5 et 6 ont chacune une complexité temporelle en `O(K)` donc les boucles des lignes 3 à 6 ont une complexité en $O(NK^2)$.
Donc au total la complexité temporelle de la fonction est en $O(NK^2)$.
▷ **Complexité spatiale.**
La ligne 2 a une complexité spatiale en `O(NK)` à cause de la création des tableaux `T` et `argT`.
Les lignes 5 et 6 ont chacune une complexité spatiale en `O(K)` donc les boucles des lignes 3 à 6 ont une complexité en `O(K)` (c'est la même variable `liste` qui est utilisée à chaque itération).
Donc au total la complexité spatiale de la fonction est en `O(NK)`.