Correction DS – Mines-Ponts 2024 ITC
Introduction à deux problèmes en communication numérique

Partie I - Compression du message d'Alice : codage arithmétique

1. On propose le codage suivant :

Caractère	Code
`'a'`	`0`
`'b'`	`10`
`'c'`	`11`

Ce codage est **préfixe** : aucun code n'est le préfixe d'un autre. En effet :

- Lorsqu'on lit un `0`, c'est forcément le caractère `'a'`.
- Lorsqu'on lit un `1`, on sait qu'on commence à lire un `'b'` ou un `'c'`, et le bit suivant permet de distinguer les deux cas.

La chaîne `s='abaabaca'` est alors codée par :

`0 10 0 0 10 0 11 0`

soit 11 bits au total.

Remarque : On a bien $0 < 10 < 11$ en binaire (soit $0 < 2 < 3$ en décimal), ce qui respecte la contrainte d'ordre demandée.

I.1 - Analyse du texte source

2. Fonction `nbCaracteres` :

```
1  def nbCaracteres(c, s):
       compteur = 0
       for lettre in s:
           if lettre == c:
5              compteur += 1
       return compteur
```

Complexité : On parcourt une seule fois la chaîne `s` de longueur n, et chaque comparaison est en $O(1)$. La complexité est donc bien $\boxed{O(n)}$, linéaire en la longueur de la chaîne.

3. Pour `s='abaabaca'`, la fonction `listeCaracteres(s)` renvoie `['a', 'b', 'c']`.
Principe de fonctionnement : La fonction parcourt la chaîne caractère par caractère. Pour chaque caractère rencontré :

- S'il n'est pas déjà présent dans la liste `listeCar`, on l'y ajoute.
- Sinon, on passe au caractère suivant.

En fin de parcours, `listeCar` contient exactement la liste des caractères distincts de `s`, dans leur ordre de première apparition.

4. **Analyse de la complexité de `listeCaracteres` :**
   - Les lignes 2, 3, 5 et 7 s'exécutent en temps $O(1)$.
   - La ligne 6 effectue un test d'appartenance `c in listeCar`. Dans le pire des cas, la liste `listeCar` contient k éléments, donc ce test est en $O(k)$.
   - La boucle `for` (lignes 4-7) effectue n itérations.

Un tour de boucle coûte donc $O(k) + O(1) = O(k)$. Comme la boucle est exécutée n fois, la complexité totale est :

$$\boxed{O(nk)}$$

5. **Ce que fait `analyseTexte` :** Cette fonction calcule l'**histogramme** de la chaîne `s`, c'est-à-dire la liste des couples (c, n_c) où c est un caractère distinct de `s` et n_c est son nombre d'occurrences.
   Pour `analyseTexte('babaaaabca')`, la fonction renvoie :

   `[('b', 3), ('a', 6), ('c', 1)]`

   (les caractères apparaissent dans l'ordre de leur première occurrence dans la chaîne)

6. **Complexité de `analyseTexte` :**
   - Ligne 2 : $O(1)$
   - Ligne 3 (`listeCaracteres(s)`) : $O(nk)$ d'après Q4
   - Lignes 4-6 : la boucle effectue k itérations. À chaque itération :
     - Ligne 5 : $O(1)$
     - Ligne 6 : appel à `nbCaracteres(c, s)` qui est en $O(n)$

     Donc la boucle coûte $k \times O(n) = O(kn)$.

   La complexité totale est donc $O(nk) + O(kn) = \boxed{O(kn)}$.

7. Version avec dictionnaire, en complexité $O(n)$:

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

Justification :

- On parcourt la chaîne `s` une seule fois (condition imposée).
- Le test `c in dico` est en $O(1)$ pour un dictionnaire (admis dans l'énoncé).
- L'accès et la modification `dico[c]` sont également en $O(1)$.

La complexité est donc $n \times O(1) = \boxed{O(n)}$, indépendante de k.

Exemple : `analyseTexte('abracadabra')` renvoie `{'a':5, 'b':2, 'r':2, 'c':1, 'd':1}`.

I.2 - Exploitation d'analyses existantes (SQL)

8. Liste sans doublon des auteurs :

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

9. Fréquence d'occurrences de chaque caractère en français :

```
1  SELECT ca.symbole,
          SUM(oc.nombreOccurrences) /
          (SELECT SUM(nombreCaracteres)
           FROM corpus
5          WHERE langue = 'Français')
   FROM caractere AS ca
   JOIN occurrences AS oc ON ca.idCar = oc.idCar
   JOIN corpus AS co ON oc.idLivre = co.idLivre
   WHERE co.langue = 'Français'
10 GROUP BY ca.idCar;
```

Explication :

- La sous-requête calcule le nombre total de caractères du corpus français.
- On joint les trois tables pour relier chaque caractère à ses occurrences dans les livres français.
- Le **GROUP BY** permet de calculer la somme des occurrences pour chaque caractère.
- Le rapport donne la fréquence (entre 0 et 1).

Remarque : Le rapport de jury précise « Plusieurs candidats proposent des sous-requêtes alors que le sujet demande explicitement UNE requête. » Je ne sais pas comment le dire autrement : le jury a tort. C'est clairement la réponse précédente qui est la plus adaptée. Cependant, voici une variante sans sous-requête (à condition de supposer que tous les caractères apparaissent dans tous les livres - ou bien que les occurrences sont données même lorsqu'il y a 0 apparition) :

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

I.3 - Compression

10. **Intervalle pour `s='bac'` :**
On utilise la table des fréquences :

Caractère	'a'	'b'	'c'	'd'	'e'
Fréquence	0.2	0.1	0.2	0.4	0.1
Intervalle	$[0;0.2[$	$[0.2;0.3[$	$[0.3;0.5[$	$[0.5;0.9[$	$[0.9;1[$

- Caractère `'b'` : on part de $[0;1[$ et on sélectionne la portion correspondant à `'b'`, soit $[0.2;0.3[$ (de largeur 0.1).
- Caractère `'a'` : on subdivise $[0.2;0.3[$ selon les mêmes proportions. La portion pour `'a'` (les premiers 20%) donne $[0.2;0.2+0.1\times 0.2[=[0.2;0.22[$.
- Caractère `'c'` : on subdivise $[0.2;0.22[$. La portion pour `'c'` commence à 30% et finit à 50% de l'intervalle. Donc :

$$g = 0.2 + 0.02 \times 0.3 = 0.2 + 0.006 = 0.206$$
$$d = 0.2 + 0.02 \times 0.5 = 0.2 + 0.01 = 0.21$$

L'intervalle final est donc $\boxed{[0.206;0.21[}$.

11. Fonction `codage` :

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

I.4 - Décodage

12. **Décodage de $x = 0.123$ après `'ad'` :**

- Premier caractère : $x = 0.123 \in [0;0.2[$ donc c'est `'a'`. Nouvel intervalle : $[0;0.2[$.
- Deuxième caractère : on subdivise $[0;0.2[$. Comme $x = 0.123 \in [0.1;0.18[$ (portion correspondant à `'d'` dans $[0;0.2[$), le caractère est `'d'`.
  Calcul : dans $[0;0.2[$, la portion pour `'d'` commence à 50% et finit à 90%, soit $[0+0.2\times 0.5;0+0.2\times 0.9[=[0.1;0.18[$.
- Troisième caractère : on subdivise $[0.1;0.18[$ (de largeur 0.08). On cherche où se trouve $x = 0.123$.

- Portion `'a'` : $[0.1; 0.1 + 0.08 \times 0.2[= [0.1; 0.116[$
- Portion `'b'` : $[0.116; 0.116 + 0.08 \times 0.1[= [0.116; 0.124[$

Comme $0.123 \in [0.116; 0.124[$, le caractère qui suit `'ad'` est `'b'`.

Le sous-intervalle utilisé est $[0.116; 0.124[$.

13. **Ambiguïté pour le flottant 0.2 :**
Les chaînes `'b'` et `'ba'` correspondent toutes deux au flottant 0.2.
Explication : Le flottant 0.2 est exactement la borne gauche de l'intervalle $[0.2; 0.3[$ associé à `'b'`. Lorsqu'on encode ensuite `'a'`, le sous-intervalle est $[0.2; 0.22[$, dont la borne gauche est toujours 0.2. Cette ambiguïté vient du fait que la borne gauche d'un intervalle appartient à cet intervalle, et que le sous-intervalle pour `'a'` conserve cette même borne gauche.

14. Fonction `decodage` :

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

Principe : On décode caractère par caractère. À chaque étape :

(a) On trouve le caractère correspondant à x dans l'intervalle courant $[g, d[$.
(b) Si ce n'est pas `'#'`, on l'ajoute à la chaîne et on met à jour l'intervalle.
(c) On répète jusqu'à trouver le caractère de fin `'#'`.

Remarque : Variante récursive :

```
def decodage(x):
    def aux(g, d, s):
        car = decodeCar(x, g, d)
        if car == '#':
            return s + '#'
        else:
            g_new, d_new = codeCar(car, g, d)
            return aux(g_new, d_new, s + car)
    return aux(0, 1, '')
```

Partie II - Décodage par l'algorithme de Viterbi

II.1 - Modélisation du canal de communication par un graphe

15. **Nombre de sommets et d'arcs :**

- **Sommets :** Il y a K symboles possibles et N observations, donc K sommets par couche et N couches. Le nombre total de sommets (hors σ et τ) est :

$$\boxed{KN}$$

- **Arcs :** Chaque sommet de la couche j (pour $j < N-1$) est relié à tous les sommets de la couche $j+1$. Il y a donc $K \times K = K^2$ arcs entre deux couches consécutives, et $N-1$ transitions entre couches. Le nombre total d'arcs (hors ceux depuis σ et vers τ) est :

$$\boxed{(N-1)K^2}$$

16. **Graphe pour `Obs` $= [2,0]$ avec $K=3$ et $N=2$:**

Les matrices sont :

$$E = \begin{pmatrix} 0.7 & 0.2 & 0.3 \\ 0.2 & 0.7 & 0.1 \\ 0.1 & 0.1 & 0.6 \end{pmatrix}, \quad P = \begin{pmatrix} 0.3 & 0.2 & 0.5 \\ 0.4 & 0.4 & 0.2 \\ 0.2 & 0.3 & 0.5 \end{pmatrix}$$

Arcs depuis σ : pondérés par $E_{\texttt{obs}_0,i} = E_{2,i}$ (première observation $= 2$)

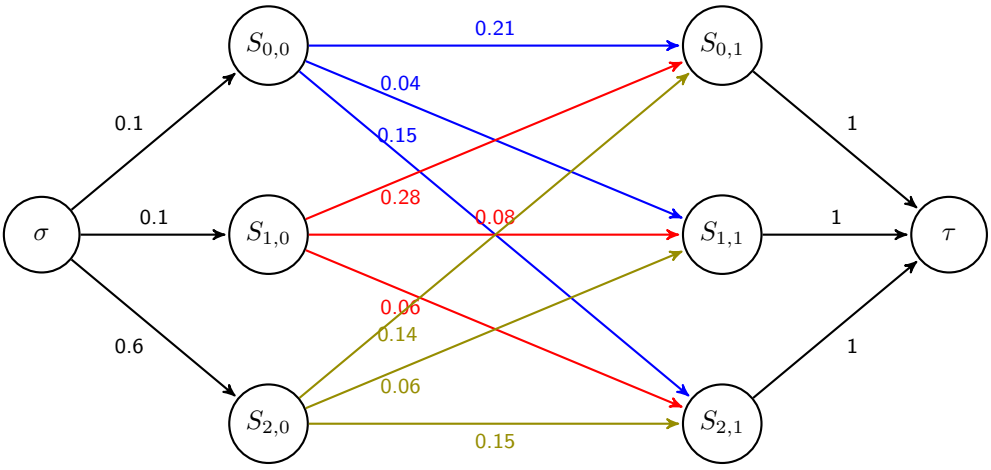
- $\sigma \to S_{0,0}$: $E_{2,0} = 0.1$
- $\sigma \to S_{1,0}$: $E_{2,1} = 0.1$
- $\sigma \to S_{2,0}$: $E_{2,2} = 0.6$

Arcs internes : pondérés par $E_{\texttt{obs}_1,k} \times P_{i,k} = E_{0,k} \times P_{i,k}$ (deuxième observation $= 0$)

Depuis $S_{0,0}$: $S_{0,0} \to S_{0,1} : 0.7 \times 0.3 = 0.21$; $S_{0,0} \to S_{1,1} : 0.2 \times 0.2 = 0.04$; $S_{0,0} \to S_{2,1} : 0.3 \times 0.5 = 0.15$

Depuis $S_{1,0}$: $S_{1,0} \to S_{0,1} : 0.7 \times 0.4 = 0.28$; $S_{1,0} \to S_{1,1} : 0.2 \times 0.4 = 0.08$; $S_{1,0} \to S_{2,1} : 0.3 \times 0.2 = 0.06$

Depuis $S_{2,0}$: $S_{2,0} \to S_{0,1} : 0.7 \times 0.2 = 0.14$; $S_{2,0} \to S_{1,1} : 0.2 \times 0.3 = 0.06$; $S_{2,0} \to S_{2,1} : 0.3 \times 0.5 = 0.15$



17. **Nombre de chemins entre σ et τ :**

À chaque couche, on choisit l'un des K symboles. Comme il y a N couches, le nombre de chemins est exactement :

$$\boxed{K^N}$$

Ce nombre croît **exponentiellement** avec N. Pour des valeurs raisonnables de K et N (par exemple $K = 26$ pour l'alphabet et $N = 100$ pour un message), K^N devient astronomique. Une exploration exhaustive n'est donc **pas envisageable** en pratique.

II.2 - Stratégie gloutonne

18. Fonction `maximumListe` :

```python
def maximumListe(liste):
    maxi = liste[0]
    ind = 0
    for k in range(1, len(liste)):
        if liste[k] > maxi:  # strict pour le plus petit indice
            maxi = liste[k]
            ind = k
    return (maxi, ind)
```

Remarque : L'inégalité stricte `liste[k] > maxi` garantit qu'en cas d'égalité, on conserve le plus petit indice (celui rencontré en premier).

19. **Erreur dans l'énoncé :** Dans `initialiserGlouton`, la ligne 2 devrait être `E[Obs[0]][i]` et non `E[Obs[0][i]]`.
Fonction `glouton` :

```python
def glouton(Obs, P, E, K, N):
    chemin = []
    i = initialiserGlouton(Obs, E, K)
    chemin.append(i)
    for j in range(N - 1):
        probas = [E[Obs[j+1]][k] * P[i][k] for k in range(K)]
        _, i = maximumListe(probas)
        chemin.append(i)
    return chemin
```

Principe : À chaque étape, on choisit le sommet suivant qui maximise la probabilité de l'arc sortant du sommet courant. C'est un choix **localement optimal**.

20. **Complexité de l'approche gloutonne :**
    - `initialiserGlouton` : parcourt une liste de taille K, donc $O(K)$.
    - La boucle s'exécute $N-1$ fois. À chaque itération :
        - Construction de `probas` : $O(K)$
        - Appel à `maximumListe` : $O(K)$

      Donc chaque itération coûte $O(K)$, et la boucle coûte $(N-1) \times O(K) = O(NK)$.

    La complexité totale est $O(K) + O(NK) = \boxed{O(NK)}$.

21. **Application à la Figure 4 :**
    - Depuis σ : arc vers symbole 0 avec probabilité 0.6, vers symbole 1 avec probabilité 0.4. L'algorithme glouton choisit le symbole **0** (probabilité maximale).
    - Depuis $S_{0,0}$: arc vers symbole 0 avec probabilité 0.5, vers symbole 1 avec probabilité 0.1. L'algorithme glouton choisit le symbole **0**.

    Le chemin renvoyé est donc $\boxed{[0,0]}$ avec une probabilité de $0.6 \times 0.5 \times 1 = 0.3$.
    Cependant, le chemin $[1,0]$ a une probabilité de $0.4 \times 0.9 \times 1 = 0.36 > 0.3$.
    Conclusion : L'algorithme glouton **n'est pas optimal**. Il peut manquer la solution globalement optimale en faisant des choix localement optimaux qui s'avèrent sous-optimaux à long terme.

II.3 - Stratégie de programmation dynamique

22. **Transformation en plus court chemin :**
    On souhaite maximiser le produit des probabilités le long d'un chemin. En appliquant la fonction $x \mapsto -\ln(x)$ (qui est décroissante et définie pour $x > 0$), on transforme :
    - le produit en somme : $-\ln(p_1 \times p_2 \times \cdots) = -\ln(p_1) - \ln(p_2) - \cdots$
    - la maximisation en minimisation (car $-\ln$ est décroissante)
    - les poids sont positifs (car $0 < p_i \leqslant 1$ implique $-\ln(p_i) \geqslant 0$)

On obtient ainsi un problème de **plus court chemin dans un graphe à poids positifs**, que l'on peut résoudre avec l'**algorithme de Dijkstra**.

Remarque : Personne n'utiliserait l'algorithme de Dijkstra pour cela. L'algorithme de Viterbi, présenté dans le sujet, est plus efficace car il exploite la structure en couches du graphe (graphe orienté acyclique avec un ordre topologique naturel).

23. Fonction `construireTableauViterbi` :

```
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):
            liste = [T[k][j-1] * P[k][i] * E[Obs[j]][i]
                     for k in range(K)]
            T[i][j], argT[i][j] = maximumListe(liste)
    return T, argT
```

Principe : On remplit le tableau colonne par colonne (de gauche à droite). Pour chaque état $S_{i,j}$, on calcule la probabilité maximale pour y arriver en considérant tous les prédécesseurs possibles $S_{k,j-1}$.

24. **Lecture de la séquence optimale :**
Avec les tableaux donnés ($K = 3$, $N = 8$) :

(a) On cherche le maximum dans la dernière colonne de T : c'est 1.8×10^{-5} en position $(0, 7)$, donc le dernier état est $S_{0,7}$ (symbole **0**).

(b) On remonte les prédécesseurs avec `argT` :

- `argT[0][7]` $= 0 \rightarrow$ prédécesseur : symbole 0
- `argT[0][6]` $= 1 \rightarrow$ prédécesseur : symbole 1
- `argT[1][5]` $= 1 \rightarrow$ prédécesseur : symbole 1
- `argT[1][4]` $= 2 \rightarrow$ prédécesseur : symbole 2
- `argT[2][3]` $= 0 \rightarrow$ prédécesseur : symbole 0
- `argT[0][2]` $= 0 \rightarrow$ prédécesseur : symbole 0
- `argT[0][1]` $= 2 \rightarrow$ prédécesseur : symbole 2
- `argT[2][0]` $= -1 \rightarrow$ source σ

La séquence d'états la plus probable est donc :

$$\boxed{[2, 0, 0, 2, 1, 1, 0, 0]}$$

25. **Complexité de l'algorithme de Viterbi :**

▷ **Complexité temporelle :**

- Initialisation (`initialiserViterbi`) : création de deux tableaux $K \times N$, donc $O(KN)$.
- Double boucle : $N - 1$ itérations sur j, K itérations sur i. À chaque itération intérieure :
  - Construction de `liste` : $O(K)$
  - Appel à `maximumListe` : $O(K)$

  Donc la double boucle coûte $(N - 1) \times K \times O(K) = O(NK^2)$.

La complexité temporelle totale est $\boxed{O(NK^2)}$.

▷ **Complexité spatiale :**

- Les tableaux T et `argT` ont chacun $K \times N$ éléments.
- La liste temporaire `liste` a K éléments (réutilisée à chaque itération).

La complexité spatiale est $\boxed{O(NK)}$.

Remarque : Par rapport à l'exploration exhaustive en $O(K^N)$, l'algorithme de Viterbi offre une complexité polynomiale, ce qui le rend utilisable en pratique.