Introduction à deux problèmes de communication numérique

Corrigé UPSTI

1 Compression du message d'Alice : codage arithmétique

□ **Q1** – Proposer une telle représentation en expliquant pourquoi celle-ci pourra être décodée sans ambiguïté. Vous ferez en sorte que la représentation binaire de 'a' soit inférieure à celle de 'b', elle-même inférieure à celle de 'c'.

C'est le principe du **codage de Huffman**, algorithme de compression sans pertes, où les caractères les plus fréquents sont codés avec un minimum de bits, en respectant la séquence suivante :

- le caractère le plus fréquent est codé **0** ;
- le suivant est codé **10** ;
- le suivant est codé **110** ;
- ... ;
- ainsi de suite, jusqu'au dernier (le moins fréquent) qui est codé **1111 ··· 1** ;

Cette représentation assure l'unicité de l'écriture de chaque caractère, puisqu'il suffit de compter le nombre de 1 successifs pour identifier de manière unique un caractère : 0 pour le premier caractère, 1 pour le second, 2 pour le troisième, ..., $n-1$ pour le n^{e} caractère.

Ainsi, dans l'exemple donné où `s='abaabaca'`, on obtient la table de correspondance ci-contre, d'écriture binaire associée **0 10 0 0 10 0 11 0**, soit une taille de 11 bits $<$ 16 bits obtenus par codage sur 2 bits de tous les caractères.

Caractère	Nombre d'occurences	Code binaire associé
'a'	5	0
'b'	2	10
'c'	1	11

Remarque : ce résultat est à relativiser, car il est nécessaire en vue du décodage de transmettre également la table de correspondance des codes, propre à chaque message, qui rajoutera donc des bits à la taille totale du message. Néanmoins, pour de longs messages, on obtient quand même des gains en compression appréciables.

1.1 Analyse du texte source

□ **Q2** – Écrire une fonction nommée `nbCaracteres(c:str,s:str)->int` qui prend comme argument un caractère `c`, une chaîne `s` et qui renvoie le nombre d'occurrences (c'est-à-dire le nombre d'apparitions) de `c` dans `s`. La fonction doit avoir une complexité linéaire en `n`, la longueur de la chaîne `s`.

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

□ **Q3** – Pour déterminer la liste des caractères utilisés à l'intérieur d'une chaîne `s` on utilise la fonction définie ci-dessous :

```
1 def listeCaracteres(s):
2     listeCar = []
3     n = len(s)
4     for i in range(n):
5         c = s[i]
6         if not (c in listeCar):
7             listeCar.append(c)
8     return listeCar
```

Que renvoie cette fonction lorsque `s='abaabaca'` ? Expliquer succinctement le principe de fonctionnement de cette fonction.

```
>>> listeCaracteres('abaabaca')
>>> ['a', 'b', 'c']
```

Cette fonction ajoute dans la liste `listeCar` tous les caractères de `s` qui n'y sont pas déjà, en balayant la chaîne `s` du premier au dernier caractère. Ainsi, l'ordre d'apparition dans la liste des caractères de `s` sera :

- `'a'` (`s[0]`) : premier caractère de `s` ;
- `'b'` (`s[1]`) : second caractère de `s` différent de `'a'` ;
- `'c'` (`s[6]`) : troisième caractère de `s` différent de `'a'` et `'b'`.

□ **Q4** – En fonction de la longueur n de la chaîne et du nombre k de caractères distincts dans celle-ci, déterminer la complexité asymptotique dans le pire des cas de la fonction de la question **Q3**. Par exemple pour `s='abaabaca'`, on a $n = 8$ et $k = 3$. On négligera la complexité des `append` mais pas celle des tests d'appartenance de la forme `i in L`.

Autrement dit, la ligne 7 est considérée comme étant de complexité constante et la ligne 6 de complexité linéaire en la longueur de la liste `listeCar`.

Les deux premières lignes sont à coût constant. Puis on boucle pour chaque élément de `s`, soit n fois. Dans cette boucle, l'instruction ligne 5 est à coût constant, l'instruction ligne 6 est à coût en $\mathcal{O}(k)$ maximum, et réalise (ou non, suivant le résultat du test) la ligne 7, qui est à coût constant.

Le pire des cas est celui où les k caractères distincts présents dans `s` sont en début de chaîne, ce qui entraîne que pour $i < k$, la complexité asymptotique de la ligne 6 est en $\mathcal{O}(i)$ et la ligne 7 est exécutée, puis pour $i \geq k$ elle est en $\mathcal{O}(k)$ et la ligne 7 n'est pas exécutée, soit au total :

$$\begin{aligned} C_1(k,n) &= 1 + \sum_{i=0}^{k-1}(1+i+1) + \sum_{i=k}^{n-1}(1+k) \\ &= 1 + \frac{k\cdot(k-1)}{2} + 2\,k + (k+1)\cdot(n-k) \\ &= 1 + \frac{3}{2}\,k - \frac{k^2}{2} + n + k\cdot n \\ &= \boldsymbol{\mathcal{O}(k\cdot n)} \end{aligned}$$

car $1 \leq k \leq n$ et donc $\mathcal{O}(1) \leq \mathcal{O}(k) \leq \mathcal{O}(k^2) \leq \mathcal{O}(k\cdot n)$ ainsi que $\mathcal{O}(1) \leq \mathcal{O}(k) \leq \mathcal{O}(n) \leq \mathcal{O}(k\cdot n)$.

Remarque : peut se démontrer de façon moins formelle, bien évidemment.

□ **Q5** – On définit alors une fonction `analyseTexte(s:str)->list`

```
1 def analyseTexte(s):
2     R = []
3     l = listeCaracteres(s)
4     for i in range(len(l)):
5         c = l[i]
6         R.append((c, nbCaracteres(c, s)))
7     return R
```

Expliquer ce que fait cette fonction et donner la valeur renvoyée par la commande `analyseTexte('babaaaabca')`.

Cette fonction renvoie une liste de k tuples de deux données, k étant le nombre de caractères distincts présents dans `s`, telles que :

- la première donnée est un caractère de `s` ;
- la seconde est son nombre d'occurrence dans `s`.

Cette fonction permet donc d'obtenir sous forme de liste de tuples toutes les occurrences des caractères présents dans `s`.

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

□ **Q6** – En fonction de la longueur n de `s` et du nombre k de caractères distincts présents dans `s`, (autrement dit k est la longueur de `listeCaracteres(s)`), donner une estimation de la complexité asymptotique dans le pire des cas de la fonction `analyseTexte`.

La ligne 2 est à coût constant, la 3 est dans le pire des cas en $\mathcal{O}(k \cdot n)$ (voir **Q4**). Puis on boucle pour chaque élément de `l`, soit k fois. Dans cette boucle, l'instruction ligne 5 est à coût constant, l'instruction ligne 6 est de complexité celle de la fonction `nbCaracteres`, soit en $\mathcal{O}(n)$, soit donc au total :

$$\begin{aligned} C_2(k,n) &= 1 + k \cdot n + \sum_{i=0}^{k-1}(1+n) \\ &= 1 + k \cdot n + k \cdot (n+1) \\ &= 1 + k + 2\,k \cdot n \\ \\ &= \boldsymbol{\mathcal{O}(k \cdot n)} \end{aligned}$$

pour les mêmes raisons que celles données en **Q4**.

Remarque : peut encore une fois se démontrer de façon moins formelle...

□ **Q7** – Adapter la fonction de la question **Q5** pour qu'elle utilise (et renvoie) un dictionnaire. Elle devra avoir une complexité :

i. linéaire en la longueur n de s ;
ii. indépendante de k nombre de caractères distincts présents dans `s`.

De plus, cette fonction devra impérativement ne parcourir qu'une seule fois la chaîne de caractères. On admettra qu'un test d'appartenance d'une clé à un dictionnaire se fait à coût constant. Par exemple,

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

```
def analyseTexte(s):
    d = {}
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

1.2 Exploitation d'analyses existantes

Q8.1 -

SELECT DISTINCT auteur FROM Corpus

Q8.2 -

SELECT titre FROM Corpus WHERE nombreCaracteres BETWEEN (100000, 300000)

Q8.3 -

SELECT COUNT(DISTINCT auteur) , langue FROM Corpus GROUP BY langue

Q9 -
Cette requête est bien plus compliquée, car elle nécessite de récupérer dans un premier temps le nombre total d'occurrences, afin ensuite de pouvoir en calculer la fréquence pour chaque symbole. Cela nécessite dans chaque cas des jointures : 1 seule pour la récupération du nombre total d'occurrences, 2 pour joindre les 3 tables dans la requête finale.
On donne dans un premier temps la requête permettant de récupérer le nombre total d' occurrences de tous les caractères des livres en langue française.

```
SELECT SUM(O.nombreOccurrences) AS 'Total' FROM occurrences AS O
JOIN corpus AS CO ON O.idLivre = CO.idLivre
WHERE CO.langue = 'Français' ;
```

Dont on déduit la requête complète :

```
SELECT CA.symbole AS 'Symbole', SUM(O.nombreOccurrences) /
    (SELECT SUM(O.nombreOccurrences) FROM occurrences AS O
        JOIN corpus AS CO ON O.idLivre = CO.idLivre
     WHERE CO.langue = 'Français' ;) AS 'Fréquence'
FROM occurrences AS O
    JOIN corpus AS CO ON O.idLivre = CO.idLivre
    JOIN caractere AS CA ON O.idCar = CA.idCar
WHERE CO.langue = 'Français'
GROUP BY CA.symbole ;
```

1.3 Compression

□ **Q10** – En considérant la table des fréquences précédente, proposer l'intervalle correspondant à la chaîne `s='bac'`.

Pour `s='bac'` :

- on obtient d'abord l'intervalle $[0.2; 0.3[$ correspondant au caractère `'b'` ;
- le caractère `'a'` détermine alors le sous-intervalle $[0.2; 0.228[$ de $[0.2; 0.3[$ correspondant à la portion associée au caractère `'a'` ($= [0.2 + 0 * (0.3 - 0.2); 0.2 + 0.2 * (0.3 - 0.2)]$) ;
- le caractère `'c'` détermine enfin l'intervalle $[0.206; 0.21[$ ($= [0.2+0.3*(0.228-0.2); 0.2+0.5*(0.22-0.2)]$).

Au final, le sous-intervalle obtenu est donc **[0.206; 0.21[**.

□ **Q11** – Écrire une fonction `codage(s:str)->(float,float)` prenant en argument la chaîne `s` et fournissant en réponse le tuple `(g,d)` constitué des deux extrémités de l'intervalle $[g, d[$ produit par l'algorithme de codage précédent.

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

1.4 Décodage

□ **Q12** – Déterminer le caractère qui suit 'ad' dans la chaîne codée par `x=0.123` en spécifiant le sous-intervalle qui a permis de décoder ce caractère.

Le troisième caractère est donc dans l'intervalle $[0.1; 0.118[$.

On en déduit les intervalles possibles pour chaque caractère :

Caractère	`'a'`	`'b'`	`'c'`	`'d'`	`'e'`
Intervalle	$[0; 0.2[$	$[0.2; 0.3[$	$[0.3; 0.5[$	$[0.5, 0.9[$	$[0.9; 1[$
Sous-intervalle de $[0.1; 0.18[$	$[0.1; 0.116[$	$[0.116; 0.124[$	$[0.124; 0.14[$	$[0.14; 0.172[$	$[0.172; 0.18[$

dont on déduit que **le troisième caractère est** `'b'` puisque 0.123 appartient à l'intervalle $[0.116; 0.124[$.

□ **Q13** – Dans le cadre de l'exemple de cette partie, indiquer deux chaînes qui peuvent correspondre au flottant 0.2. Expliquer par une phrase ce qui est à l'origine de cette ambiguïté.

Les chaînes contenant 0.2 sont obligatoirement celles :

- démarrant par `'b'` puisque seul caractère dont l'intervalle inclus 0.2, qui est la première valeur de cet intervalle ;
- suivi de `a` en un nombre illimité, seul caractère assurant que le sous-intervalle commence toujours par la valeur 0.2 incluse.

Les chaînes correspondantes sont donc

`'b', 'ba', 'baa', 'baaa', ... , 'baaaaaaaa', ...`

Cette multiplicité est très gênante, car un même nombre flottant peut donc coder pour une infinité de chaînes !

Cette ambiguïté vient de la présence du 0 dans l'intervalle du caractère `'a'`, qui permet ainsi d'inclure la première valeur d'un intervalle dans les sous-intervalles obtenus avec l'ajout de ce caractère, et donc générer de la redondance possible de code arithmétique.

□ **Q14** – Écrire une fonction `decodage(x:float)->str` produisant la chaîne de caractères `s` déterminée par la valeur de `x` (avec le caractère '#' compris).

```
def decodage(x):
    g, d = 0, 1
    s = ""
    fini = False

    while not fini:
        c = decodeCar(x, g, d)
        s += c
        g, d = codeCar(c, g, d)
        if c == '#':
            fini = True
    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** – En fonction de N et de K, donner le nombre de sommets et d'arcs du graphe illustré par la Figure 3. On ne comptera pas les sommets source σ et cible τ, ni les arcs partant du sommet source σ ni ceux arrivant à la cible τ.

Sans les sommets d'entrée et de sortie de graphe, et en notant $G = (S, A)$ le graphe G d'ensemble de sommets S et d'ensemble d'arcs A, on a :

- K sommets par observation, pour N observations : $Card(S) = K \cdot N$;

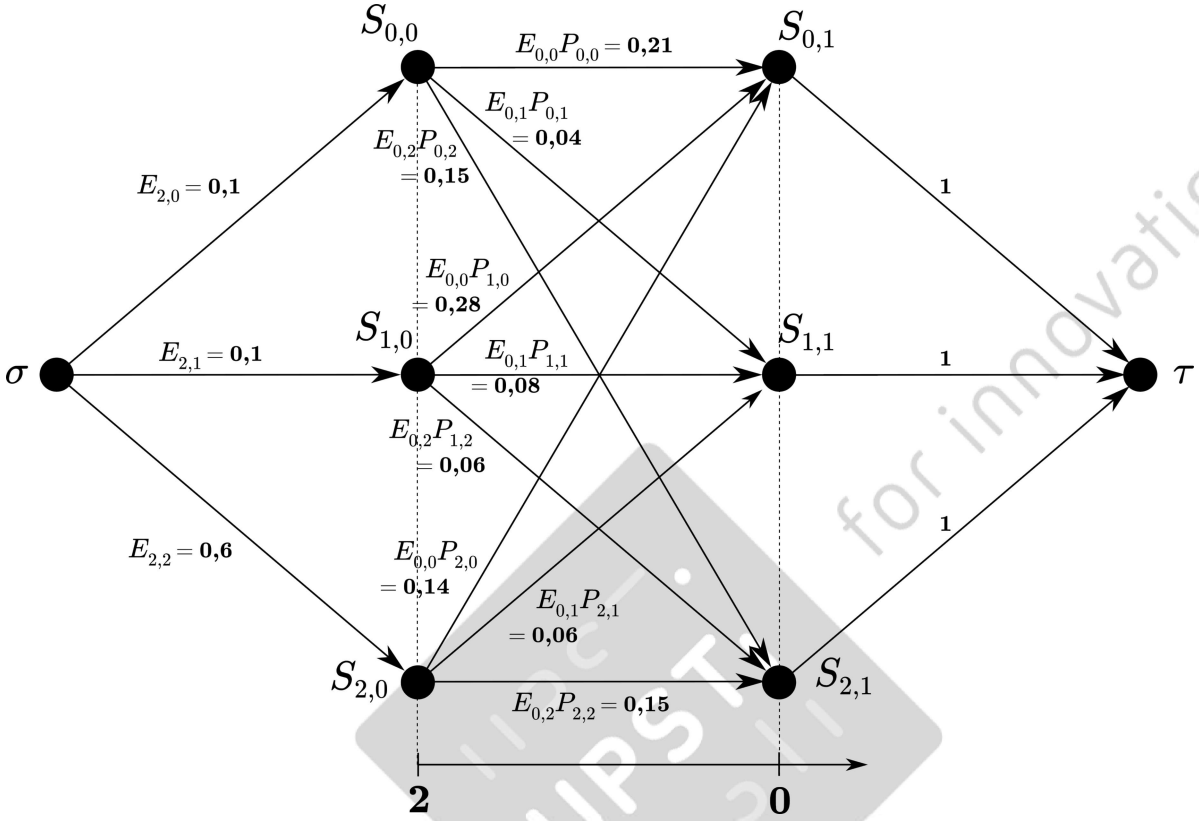
$$\Rightarrow \qquad \boldsymbol{Card(S) = \mathcal{O}(K \cdot N)}$$

;

- K arcs par sommet, hormis pour les K derniers : $Card(A) = K \cdot (\underbrace{K \cdot N}_{=Card(S)} - K) = K^2 \cdot (N-1)$.

$$\Rightarrow \qquad \boldsymbol{Card(A) = \mathcal{O}(K^2 \cdot N)}$$

□ **Q16** – On suppose que Bob a observé la séquence `[2,0]`. En utilisant les matrices `E` et `P` données dans l'énoncé (avec $K = 3$), construire le graphe pondéré associé à ce message de longueur $N = 2$. Les arcs entre les sommets devront être pondérés par les probabilités correspondantes.



□ **Q17** – On revient dans le cas général, N et K sont désormais quelconques. Indiquer combien il existe de chemins entre σ et τ (un ordre de grandeur utilisant la notation $\mathcal{O}$ ou Θ est accepté). Préciser si un algorithme d'exploration exhaustive est envisageable dans ce cas.

Il y a K chemins initiaux depuis σ, puis K chemins possibles pour chaque observation j, avec $0 \leq j < N-1$, enfin pour $j = N-1$ il n'y a qu'un chemin possible menant à τ, soit donc :

$$\textbf{Nombre de chemins } = K \times (K^{N-1}) \times 1 = K^N \textbf{ !!!}$$

Un parcourt de graphe exhaustif aura donc une complexité asymptotique minimale en $\mathcal{O}(K^N)$, soit une **complexité exponentielle** : **une recherche exhaustive est donc inenvisageable** (explosion combinatoire).

2.2 Stratégie gloutonne

□ **Q18** – Pour une liste `liste`, on appelle argument du maximum et on note `argMax` tout indice i tel que `liste[i]` soit maximal. Proposer une fonction `maximumListe(liste:[float])->(float,int)` qui prend en entrée une liste de nombres et qui renvoie la valeur du maximum de la liste ainsi que le plus petit argument du maximum, i.e. le premier indice auquel cette valeur maximale apparaît.

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

□ **Q19** – Proposer une fonction `glouton(Obs:[int],P:[[float]],E:[[float]],K:int,N:int)->[int]` qui renvoie la liste d'états obtenue par l'approche gloutonne. Même si cela n'est pas nécessaire, K, N seront des arguments de cette fonction.

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

□ **Q20** – En fonction de K et de N , quelle est, en ordre de grandeur, la complexité temporelle asymptotique de l'approche gloutonne ?

La fonction `glouton` :

- appelle `InitialiserGlouton` qui a une complexité en $\mathcal{O}(K)$, puisque balayant les K éléments pour calculer les probas + appel de la fonction verb !maximumListe ! de complexité en $\mathcal{O}(K)$ également ;
- puis boucle $N-1$ fois :
  - ◇ en bouclant K fois pour le calcul de probabilités ;
  - ◇ en appelant la fonction `maximumListe` de complexité en $\mathcal{O}(K)$, puisque balayant la liste des probas de longueur K ;
  - ◇ puis ajoutant le symbole dans la liste résultat à coût constant.

soit donc :

$$\boldsymbol{C_3(K,N) = \mathcal{O}(K) + (N-1) \times \mathcal{O}(K) = \mathcal{O}(K \cdot N)}$$

□ **Q21** – Indiquer le chemin renvoyé par l'algorithme glouton appliqué à la Figure 4. Conclure quant à l'optimalité de l'approche.

L'algorithme choisira :

- le symbole 0 initialement, de plus grande probabilité $= 0.6$;
- puis à nouveau le symbole 0, de plus grande probabilité $= 0.5$.

L'algorithme renverra donc les symboles `[0, 0]` pour une probabilité totale du chemin de $0,5 \times 0,6 = 0,3$.

On constate facilement que ce chemin n'est pas optimal, le chemin optimal étant obtenu pour les symboles `[1,0]` de probabilité totale $0,4 \times 0,9 = 0,36 > 0,3$: **cet algorithme est bien un algorithme glouton puisqu'il renvoie une solution sous-optimale** (au mieux optimale, mais pas dans tous les cas).

2.3 Stratégie de programmation dynamique

□ **Q22** – Expliquer en quoi rechercher un chemin de probabilité maximale pourrait se transformer en un problème de recherche de plus court chemin dans un graphe pondéré à poids positifs. Préciser alors quel algorithme pourrait être utilisé.

La recherche d'un plus court chemin, à savoir le chemin de distance minimale, où la distance est la somme des pondérations des arcs du chemin, est un algorithme de recherche de solution optimale en terme de distance minimale. Un tel algorithme peut très bien être adapté à la recherche de chemin le plus probable, où la probabilité est le produit des pondérations des arcs de ce chemin, où si l'on applique la fonction log au calcul des probabilités (fonction monotone strictement croissante), les extremums sont obtenus pour les mêmes valeurs, où maintenant la recherche est celle d'un chemin le plus «long». L'algorithme de recherche d'un plus «court» chemin peut donc très bien être adapté à la recherche d'un chemin le plus probable.

Un des algorithmes le plus connu est **l'algorithme de Dijkstra**, ou encore l'**algorithme A^*** qui est une variante de Dijkstra basée sur une heuristique permettant de ne pas parcourir le graphe dans sa totalité (mais c'est en ce sens un algorithme glouton puisque sous-optimal).

□ **Q23** – Proposer une fonction (méthode de bas en haut de programmation dynamique)

`construireTableauViterbi(Obs:[int], P:[[float]], E:[[float]], K:int, N:int)`

`->([[float]],[[int]])` qui prend comme arguments la liste des observations `Obs`, la matrice des probabilités de transition `P` et la matrice des probabilités `E` et renvoie les deux listes de listes `T` et `argT` de taille $K \times N$.

```
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):
            maxT, argmaxT = 0, 0
            for k in range(K):
                proba = T[k][j-1]*P[k][i]*E[Obs[j]][i]
                if proba > maxT:
                    maxT = proba
                    argmaxT = k
            T[i][j] = maxT
            argT[i][j] = argmaxT

    return T, argT
```

□ **Q24** – L'algorithme de Viterbi codé en Python et appliqué à un message en entrée donne les tableaux `T` et `argT` suivants. Indiquer la séquence d'états la plus probable.

$$T = \begin{pmatrix} 0.1 & 0.084 & 0.018 & 0.00053 & 0.00021 & 8.9e-05 & 8.7e-05 & 1.8e-05 \\ 0.1 & 0.036 & 0.0054 & 0.00041 & 0.0011 & 0.00031 & 2.5e-05 & 3.5e-06 \\ 0.6 & 0.09 & 0.014 & 0.0053 & 0.00026 & 2.2e-05 & 1.9e-05 & 1.3e-05 \end{pmatrix}$$

$$argT = \begin{pmatrix} -1 & 2 & 0 & 0 & 2 & 1 & 1 & 0 \\ -1 & 2 & 2 & 2 & 2 & 1 & 1 & 0 \\ -1 & 2 & 0 & 0 & 2 & 1 & 1 & 0 \end{pmatrix}$$

Pour retrouver la séquence la plus probable, il faut appliquer l'algorithme suivant :

- on se place à la dernière colonne ($j = N-1$) et on cherche la ligne de valeur maximale dans `T` : le dernier symbole sera l'indice de ligne obtenu ;
- Ensuite on récupère le chemin en cherchant le prédécesseur dans le tableau `argT`, et on reconstruit ainsi de proche en proche en remontant tous les prédécesseurs, jusqu'à la valeur -1 du nœud initial σ.

Ainsi, dans l'exemple donné :

- La valeur maximale dans la dernière colonne est `T[0][7] = 1.8e-05` donc dernier symbole = 0, et `argT[0][7] = 0` donc symbole précédent = 0 ;
- puis prédécesseur du 0 de la colonne 6 est `argT[0][6] = 1` ;
- puis prédécesseur du 1 de la colonne 5 est `argT[1][5] = 1` ;
- puis prédécesseur du 1 de la colonne 4 est `argT[1][4] = 2` ;
- puis prédécesseur du 2 de la colonne 3 est `argT[2][3] = 0` ;
- puis prédécesseur du 0 de la colonne 2 est `argT[0][2] = 0` ;
- puis prédécesseur du 0 de la colonne 1 est `argT[0][1] = 2` ;
- puis prédécesseur du 2 de la colonne 0 est `argT[2][0] = -1`, et le chemin est complet.

Le chemin le plus probable est donc **{2-0-0-2-1-1-0-0}**.

Remarque *: il semble qu'il y ait une légère erreur dans le tableau argT à la colonne 2, car en appliquant l'algorithme de Viterbi à la séquence* `[2,0,0,2,1,1,0,0]` *on retrouve exactement le tableau T (aux arrondis près), ainsi que le tableau argT avec comme seule différence la colonne 2 qui vaut* $\begin{pmatrix} 0 \\ 2 \\ 2 \end{pmatrix}$ *et non* $\begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}$*, mais cela n'a aucune incidence sur le traitement de la question...*

□ **Q25** – En fonction de K et de N, donner l'ordre de grandeur de la complexité temporelle de l'approche de programmation dynamique, ainsi que la complexité spatiale.

- **Complexité temporelle** :

  Elle est principalement liée aux 3 boucles imbriquées de la fonction `construireTableauViterbi` qui respectivement bouclent $N-1$ fois, K fois et encore K fois ; pour une complexité temporelle asymptotique en :

  $$\boldsymbol{\mathcal{O}(K^2 \cdot N)}$$

- **Complexité spatiale** :

  Les structures de données utilisées sont :
  - ◇ la liste `Obs` de longueur N ;
  - ◇ les matrices `P` et `E` de dimensions $K \times K$;
  - ◇ les matrices `T` et `argT` de dimensions $K \times N$.

  N et K pouvant être quelconques (mais entiers et ≥ 1), on en déduit une complexité spatiale asymptotique en :

  $$\boldsymbol{\mathcal{O}(\max\{K^2, K \cdot N\})}$$

FIN DU CORRIGÉ