1. Si l'on code a par `0`, b par `10` et c par `11`, il n'y aura pas d'ambiguïté de lecture : si l'on lit un `1`, on sait que l'on commence à lire un b ou un c, le bit suivant indiquant lequel. Les `1` sont donc toujours au début d'un bloc de longueur 2 et ne sont pas interprétés seuls, jamais les `0`.

   La chaîne s est alors codée par `01000100110`, donc sur 11 bits.

2.

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

3. Sur cette exemple, la fonction renvoie la liste `['a','b','c']`.

   En effet, cette fonction parcourt les caractères de la chaîne s, et pour chaque caractère :

   — s'il n'a pas déjà été rencontré, donc s'il n'est pas dans la liste `listCar`, on l'ajoute à `listCar` ;

   — sinon, on le passe.

   Cette fonction renvoie ensuite la liste `listCar`, qui contient bien la liste des caractères de s, sans doublon.

4. Les instructions des lignes 2, 3, 5 et 7 s'effectuent en temps $O(1)$.

   Le test de la ligne 6 s'effectue en temps (au plus) $O(k)$. Ainsi, un tour de la boucle `for` s'effectue en temps $O(k) + O(1) = O(k)$. Cette boucle effectue n tours, donc a une complexité en $nO(k) = O(nk)$.

   La complexité totale de la fonction est donc en $O(nk) + O(1) = O(nk)$.

5. Cette fonction calcule la liste des caractères de s (ligne 3), et pour chacun de ces caractères c elle calcule le nombre n_c d'occurences de c dans s (ligne 6), puis renvoie la liste des (c, n_c).

   Cette fonction renvoie donc *l'histogramme* de la liste s, c'est-à-dire la liste des couples (c, n_c), où les c sont triés par ordre d'apparition dans s.

   Sur cet exemple, la fonction renvoie la liste `[('b',3), ('a',6), ('c',1)]`.

6. La ligne 3 a une complexité en $O(kn)$. La ligne 6 a une complexité en $O(n)$ (appel de la fonction `nbCaracteres`), et est itérée k fois. Ainsi, la boucle `for` de cette fonction s'effectue en temps au plus $kO(n) = O(kn)$.

   La complexité totale de cette fonction est donc $O(kn) + O(kn) = O(kn)$.

7.

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

8.

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

9. On va calculer à part le nombre de total de caractères du corpus.

```sql
SELECT symbole, SUM(nombreOccurences)/nbTotal
FROM caractere, (SELECT SUM(nombreCaracteres) AS nbTotal
                 FROM corpus
                 WHERE langue = 'Francais')
```

```sql
JOIN occurences ON caracteres.idCar = occurences.idCar
JOIN corpus ON occurences.idLivre = corpus.idLivre
WHERE langue = 'Francais'
GROUP BY caractere.idCar
```

10. Lettre b : intervalle $[0,2;0,3[$, de largeur $\frac{1}{10}$.
Pour la lettre a, on considère ensuite le premier cinquième de cet intervalle : $[0,2;0,22[$, de largeur $\frac{1}{5}\times\frac{1}{10}=\frac{1}{50}$.
Pour la lettre c, on subdivise cet intervalle en dix intervalles :

$$[0,2;0,202;0,204;0,206;0,208;0,210;0,212;0,214;0,216;0,218;0,220],$$

et l'on considère les quatrième et cinquième intervalles, ce qui donne $[0,206;0,210]$. Cet intervalle est bien de largeur $\frac{1}{5}\times\frac{1}{50}=\frac{1}{250}=0,004$.

11.

```python
def codage(s) :
    g, d = 0, 1
    for caractere in s :
        g, d = codCar(caractere , g , d)
    return g, d
```

12. Le caractère `a` donne l'intervalle $[0;0,2[$. Le caractère `d` donne ensuite l'intervalle $[0,1;0,18[$. On le subdivise ensuite en 10 intervalles de largeur $\frac{8}{100}$, ce qui donne la subdivision

$$[0,1;0,108;0,116;0,124;0,132;0,140;0,148;0,156;0,164;0,172;0,18].$$

Comme x appartient au troisième intervalle, le troisième caractère est un `b`.

13. Les chaînes `b` et `ba` sont toutes les deux représentées par $0,2$.
Cela vient du fait que la borne de gauche est autorisée pour `a`. Si on prenait le codage de la chaîne à l'intérieur de l'intervalle considéré, il n'y aurait pas ce problème.

14. On peut récupérer le premier caractère de la chaîne par la fonction `decodeCar`, puis calculer les bornes suivantes par la fonction codage. On itère ensuite ce procédé, tant que le dernier caractère décodé n'est pas `#`. On peut donc écrire la fonction suivante.

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

On peut aussi en donner une version récursive.

```python
def decodage(x):
    def aux(s, g, d):
        if s[-1] == '#' :
            return s
        else :
            c = decodeCar(x, g, d)
            s2 = s+c
            g, d = codage(s2)
```

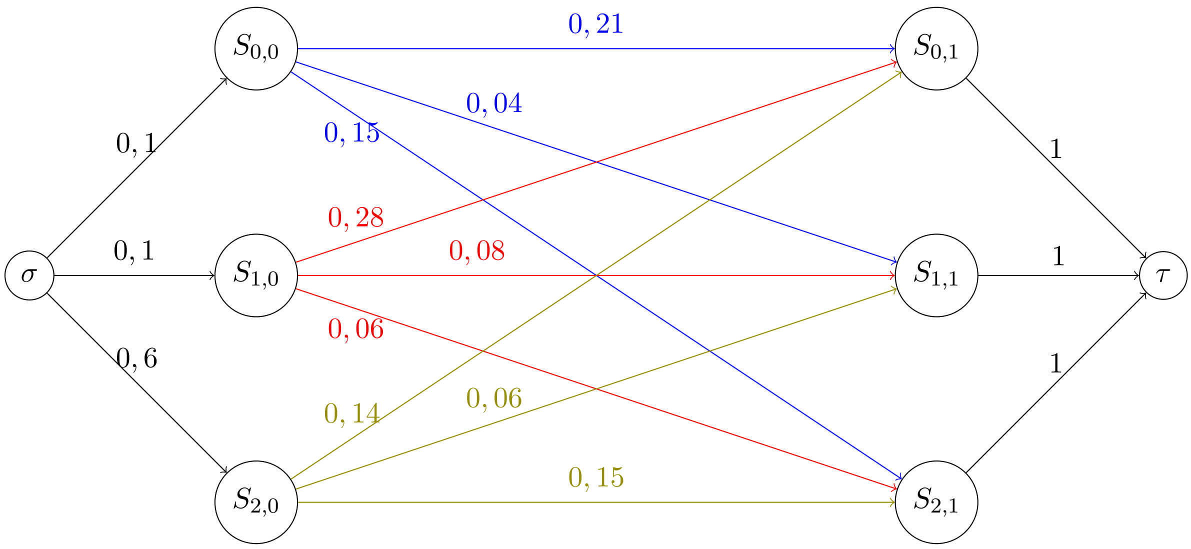
```
            return aux(s2, g, d)
    return aux("", 0, 1)
```

15. Il y a N observations, et pour chaque observation un sommet pour chacun des K caractères : il y a donc KN sommets.

    Pour chacune des $N-1$ observations, sauf la dernière, et chacun des K caractères, il y a K arêtes dans le graphe. Il y a donc $(N-1)K^2$ arête dans le graphe.

16. Le premier symbole est `2`, les probabilités partant de la source sont donc données par la dernière ligne de E : $(0,1; 0,1; 0,6)$.

    Le deuxième symbole est `0`. Pour calculer la probabilité de la transition de $S_{i,0}$ vers $S_{k,1}$, on multiplie coefficient par coefficient chaque ligne de P par la première ligne de E.



17. Choisir un chemin entre σ et τ revient, pour chacune des N obsevations, à choisir un symbole. Il y a donc K^N tels chemins exactement (pas besoin de O...).

    Il n'est pas possible de mettre en œuvre un algorithme exhaustif dès que K ou N sont grands : il y a trop de chemins.

18.

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

19. Les spécifications de la fonction `initialiserGlouton` ne sont pas correctes (`Obs` est une liste d'entiers), et le code est erroné (lire `E[Obs[0]][i]`), mais c'est sans incidence pour la suite.

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

20. La fonction `initialiserGlouton` a *a priori* une complexité en $O(K)$, vu qu'elle consiste en un parcours simple d'une liste de longueur K.

    Le calcul de la liste `probas` s'effectue en temps $O(K)$, pour la même raison, et le calcul de k aussi. Ainsi, un tour de la boucle `for` s'effectue en temps $O(K)$, et cette boucle est réalisée $N-1 \leq N$ fois.

    La complexité de la fonction `glouton` est donc en $O(K) + NO(K) = O(NK)$.

21. L'algorithme glouton choisit d'abord le caractère `0`, puis encore le caractère `0`. Le produit des probabilités sur ce chemin donne $0,3$.

    Or, le chemin `10` a une probabilité de $0,36$. L'algorithme glouton n'est donc pas optimal.

22. En transformant chaque probabilité de transition $p \in [0,1]$ en $1-p \in [0,1]$, on transforme le plus long chemin en le plus court chemin, toujours dans un graphe pondéré à poids positifs.

    On pourrait alors appliquer l'algorithme de Dijkstra.

23.

```python
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)]
            max, imax = maximumListe(liste)
            T[i][j] = max
            argT[i][j] = imax
    return T, argT
```

24. On a ici $N = 8$ et $K = 3$. En partant de la dernière colonne, on revient en arrière en choisissant la probabilité maximale (ici, $1,8 \times 10^{-5}$), qui correspond au symbole `0`

    Il suffit ensuite de remonter la liste des prédécesseurs dans le deuxième tableau : pour chaque observation $1 \leq i \leq 8$, si le chemin optimal passe par k, alors le prédécesseur est donné par $argT_{k,i}$. On obtient successivement 0, 1, 1, 2, 0, 0, 2. La séquence d'états la plus probables est donc $\sigma - 2 - 0 - 0 - 2 - 1 - 1 - 0 - 0 - \tau$.

25. Complexité temporelle. L'initalisation du tableau s'effectue clairement en temps $O(KN)$. Un tour de la boucle sur i a une complexité en $O(K)$ (recherche de maximum pour une liste de taille K), cette boucle comporte K tours et a donc une complexité en $O(K^2)$. Un tour de la boucle sur j a donc une complexité en $O(K^2)$ et est répétée $N-1$ fois, donc cette boucle a une complexité temporelle en $O(NK^2)$.

    La complexité temporelle de cet algorithme est donc en $O(NK^2)$.

    Complexité spatiale. On crée deux nouveaux tableaux de taille $N \times K$. La complexité spatiale est donc en taille $O(NK)$.