Sujet bac NSI 2025 jour 2 (18 juin) | Correction

Version pdf du corrigé

Exercice 1

1. Le caractère espace (noté `_` dans l'arbre) sera codé `010`
2. Pour décoder le mot, on descend dans l'arbre jusqu'à arriver à une feuille, puis on recommence avec la suite du mot binaire.

   Ainsi le mot binaire '00011101011111110011001' peut être découpé en

   '00 011 1010 1111 11001 1001' 'e s p i o n'

   Le texte codé est : 'espion'
3. Un ==parcours en largeur d'abord== permet de parcourir les feuilles par ordre de profondeur croissante. C'est donc celui qui permet ici d'obtenir les symboles par taille d'encodage croissante.

   Partie B
4. Le total des occurrences est égal au nombre de symboles de la chaîne (22 dans l'exemple de la figure 2.) Pour avoir deux sous-groupes avec le nombre d'occurrences les plus proches possible, il faut que ces nombres soient le plus proche de la moitié (ici 11).

   Dans le cas de la Figure 2., les deux sous-groupes ont un nombre d'occurrences égal à 11, c'est donc le partage optimal.

   De plus l'ordre des symboles est bien conservé entre les sous-groupes et au sein de chaque sous groupe.
5. La profondeur de la racine étant 0, la profondeur maximale est atteinte pour les feuilles `o` et `d` et elle est de 5. La hauteur de l'arbre de la Figure 3. est 5. Dans le contexte du codage de Shannon-Fano cela correspond donc à la taille maximale en bits du codage d'un symbole.
6. En ASCII, la chaîne de caractères `je pense, donc je suis` est encodée avec 22 octets, c'est à dire 22 x 8 = 176 bits.

   Le codage de Shannon-Faro demanderait ici beaucoup moins de bits :

- 4 bits pour chacun des caractères `i u c p` qui apparaissent chacun 1 fois : 16 bits
- 5 bits pour chacun des caractères `o d` qui apparaissent chacun 1 fois : 10 bits
- 4 bits pour chacun des caractères `n j` qui apparaissent chacun 2 fois : 16 bits
- 3 bits pour le caractère `s` qui apparait 3 fois : 9 bits
- 3 bits pour le caractère `_` qui apparait 4 fois : 12 bits
- 2 bits pour le caractère `e` qui apparait 4 fois : 8 bits

Au total le codage de Shannon-Fano utilise 16 + 10 + 16 + 9 + 12 + 8 = 71 bits. C'est environ deux fois moins que le codage ASCII.

> Cette performance du codage de Shannon-Fano s'explique autant par l'intérêt d'un codage préfixe qui utilise des codages plus courts pour les caractères les plus fréquents que par le nombre réduits de caractères codés dans le codage de Shannon-Fano : 12 au lieu des 128 de l'ASCII.

7. Dessin

Partie C

8.
```python
def creer_dico_occ(texte: str) -> dict:
    dico = {} # dictionnaire vide
    for symbole in texte:
        if symbole in dico:
            dico[symbole] = dico[symbole] + 1
        else:
            dico[symbole] = 1 # première apparation du symbole
    return dico
```

9.
```python
def somme_occ(tab: list) -> int:
    somme = 0
    for symbole, occ in tab:
        somme += occ
    return somme
```

ou par compréhension:

```python
def somme_occ(tab: list) -> int:
    return sum([t[1] for t in tab])
```

10.
```python
def separe(tab: list) -> (list,list):
    moitie = somme_occ(tab) // 2
    somme = 0
    i = 0
    while moitie > somme:
        somme = somme + tab[i][1] # incrémentation du nombre d'occurrence du i-ème symb
        i = i + 1
```

```
    tab1 = [ tab[k] for k in range(0,i)]
    tab2 = [ tab[k] for k in range(i,len(tab))]
    return tab1, tab2
```

Fonction `shannon` complétée :

```
def shannon(symbole, tab):
    if len(tab) == 1:
        return ""
    else:
        t1, t2 = separe(tab)
        if symbole in [elt[0] for elt in t1]:
            return "1" + shannon(symbole, t1)
        else:
            return "0" + shannon(symbole, t2)
```

11. La fonction récursive `shannon` se termine car elle possède une condition d'arrêt (len(tab) == 1) et que les appels récursifs se font sur des tableau de tailles qui décroissent strictement, au dernier appel, le tableau ne contient que le caractère et a donc pour taille 1.

12. ```
    def encode_shannon(texte: str) -> str:
        '''Retourne l'encodage binaire de texte avec le codage de Shannon-Fano'''
        dico = creer_dico_texte(texte) # création du dictionnaire
        tab = creer_tab_trie(dico) # liste triée suivant les nombres d'occurrences
        codage = {} # dictionnaire des codages des caractères

        # ajout des codages des caractères au dictionnaire codage
        for symbole in dico:
            codage[symbole] = shannon(symbole, tab)


        encodage_binaire = "" # chaine pour contenir le codage binaire

        for s in texte:
            encodage_binaire += codage[s]

        return encodage_binaire
    ```

    On peut envisager une fonction sans le dictionnaire codage. Cependant ce dernier est utile pour ne pas avoir à rappeler la fonction `shannon` a chaque occurrence d'un symbole.

Exercice 2

1. Une clé primaire doit respecter une contrainte d'unicité. Il peut y avoir des homonymes parmi les adhérents. L'attribut `nom` ne peut donc pas servir de clés primaire.
    ```

2. La requête SQL va renvoyer les noms des jeux et des éditeurs pour tous les jeux de la relation `jeu`. Ils seront triés par ordre alphabétique des noms des jeux.

3. ```sql
   SELECT nomJeu
   FROM emprunt
   WHERE dateRendu is NULL
   ```

4. ```sql
   SELECT nom, prenom
   FROM adherent
   NATURAL JOIN emprunt
   NATURAL JOIN jeu
   WHERE nomJeu = 'Catan'
   ```

5. ```sql
   UPDATE emprunt
   SET dateRendu = '2025-06-03'
   WHERE idEmprunt = 1538
   ```

6. ```sql
   SELECT nomJeu, categorie
   FROM jeu
   WHERE anneeSortie >= 2010 AND ageMinimum < 10
   ```

7. Les deux autres attributs de la relation `participation` sont des clés étrangères : `idAdherent` qui fait référence à la clé primaire de `adherent` et `nomEvenment` qui fait référence à la clé primaire de la relation `evenement`

8. Voici la fonction `dict_emprunts` qui sera appelée avec la liste des jeux créees ci-dessus

   ```python
   def dict_emprunts(liste):
       dico = {}
       for jeu in liste:
           if jeu in dico:
               dico[jeu] += 1
           else:
               dico[jeu] = 1
       return dico
   ```

9. Question plus difficile

   ```python
   def gen_podium(emprunts: dict) -> [list]:
       '''

       # ÉTAPE 1 : création de la liste de tous les nombres d'emprunts distinct triée en d
       Retourne la liste des listes des jeux sur le podium des plus empruntés. La première
       '''
       nombres_emprunts = [] # liste qui va contenir les différents nombres d'emprunts par

       for nb in emprunts.values():
           if nb not in nombres_emprunts:
   ```

```
            nombres_emprunts.append(nb) # ajout de nb a sa première apparition uniqueme

    nombres_emprunts.sort() # trie de la liste en ordre croissant

    # ÉTAPE 2: Création des 3 listes des emprunts bronze, argent et or à partir de la l
    jeux_or = [ jeu for jeu,nb_emprunts in emprunts.items() if nb_emprunts == nombres_e
    jeux_argent = [ jeu for jeu,nb_emprunts in emprunts.items() if nb_emprunts == nombr
    jeux_bronze = [ jeu for jeu,nb_emprunts in emprunts.items() if nb_emprunts == nombr

    return [jeux_bronze, jeux_argent, jeux_or]
```

Il est possible d'écrire une algorithme en complexité linéraire qui parcours le dictionnaire `emprunts` et qui pour chaque jeu, soit ne fait rien si le jeu a un nombre d'emprunts inférieur à ceux du podium potentiel, soit ajoute à la liste d'une des 3 places du podium si le nombre d'emprunts est égal au nombre de cette place, soit crée une nouvelle liste singleton avec le jeu uniquement si le nombre d'emprunts est supérieur strictement au nombre d'une des places du podium sans être égal à un autre du podium.

Exercice 3

1. `11 (L)  8 (I)  1 (B) 17 (R)  4 (E)`

   `4 (E) 24 (Y) 16 (Q) 12 (M) 19 (T)`

   en sommant et en appliquant le modulo 26

   `15 (P) 6 (G) 17 (R)  3 (D) 23 (X)`

   `LIBRE` sera codé `PGRDX` à l'aide de la clé `EYQMT`

2. ```
   def indice(L: list, element) -> int:
       for i in range(len(L)):
           if L[i] == element:
               return i
   ```

   Il est aussi possible d'utiliser la méthode native Python `index` qui s'applique aux listes.

3. ```
   def lettres_vers_indices(texte):
       return [ indice(lettre) for lettre in texte ]
   ```

   Utiliser `ord(lettre) - ord('A')` permettrait d'éviter la recherche séquentielle dans alphabet pour chaque lettre, donc de gagner en complexité temporelle.

4. ```
   def chiffrement(msg, cle):
       assert len(cle) >= len(msg), 'impossible'
       indices_msg = lettres_vers_indices(msg)
       indices_cle = lettres_vers_indices(cle)
   ```
   ```

```
    n = len(msg)
    indices_msg_chiffre = []
    for k in range(n):
        ind = indices_msg[k] + indices_cle[k] # somme des indices
        if ind >= 26:
            ind = ind % 26 # valeur modulo 26
indices_msg_chiffre.append(ind)
    msg_chiffre = indices_vers_lettres(indices_msg_chiffre)
    return msg_chiffre
```

5. L'appel `chiffrement('RESEAU', 'GFTZ')` va provoquer une erreur de type AssertionError et afficher le message `'impossible'` précisé dans assert. La clé est en effet trop courte.

6. `6 (G)  12 (M)  4 (E) 3 (D)  7 (H)`

   `5 (F) 21 (V)  4 (E) 8 (I) 19 (T)`

   en soustrayant les indices de la clé et en appliquant le modulo 26

   `1 (B) 17 (R) 0 (A) 21 (V) 14 (O)`

   Le message `GMEDH` est déchiffré en `BRAVO` avec la clé `FVEIT`

7. Pour déchiffrer le message on applique un procédé analogue au chiffrement, sauf qu'au lieu de sommer les indices de la clé on les soustrait (pour revenir à l'indice de départ). On applique aussi de nouveau le calcul du modulo 26.

8. ```
    def dechiffrement(msg, cle):
            assert len(cle) >= len(msg), 'impossible'
            indices_msg = lettres_vers_indices(msg)
            indices_cle = lettres_vers_indices(cle)
            n = len(msg)
            indices_msg_dechiffre = []
            for k in range(n):
                    ind = indices_msg[k] - indices_cle[k] # différence des indices
                    if ind < 0:
                            ind = ind % 26 # valeur modulo 26
                    indices_msg_dechiffre.append(ind)
            msg_dechiffre = indices_vers_lettres(indices_msg_dechiffre)
            return msg_dechiffre
    ```

Partie B - Sécururisation des communications

9. Un **chiffrement symétrique** utilise la même clé pour chiffrer et déchiffrer. L'exemple du masque jetable utilise un chiffrement symétrique. Cette clé unique impose qu'elle soit le même du côté emission et du côté réception.

   Un **chiffrement asymétrique** utilise une clé pour chiffer et une autre pour déchiffrer. Cela demande à chaque utilisateur de posséder un couple
    ```

de clé (une publique et un privée). RSA est un exemple d'algorithme de chiffrement asymétrique.

Le chiffrement symétrique propose des algorithmes très sûrs mais nécessite un échange de clé entre les parties, ce qui est parfois une difficulté à sa mise en oeuvre.

10. Avec un algorithme de chiffrement asymétrique tel que RSA, Bob utilisera sa clé privée pour déchiffrer le message envoyé par Alice et qu'elle a chiffré avec la clé publique de Bob.

11. L'utilisation de la seule clé publique de Bob ne suffit pas à garantir l'authenticité de la personne qui émet. Ainsi une tierce personne peut remplacer le message d'Alice par un autre qui sera lui aussi chiffré avec la clé publique de Bob. Pour garantir l'authenticité des parties dans un échange asymétrique, il faut utiliser une signature qui sera chiffré avec la clé privé de la personne qui émet, garantissant ainsi son identité.

12. HTTPS est un acronyme : HyperText Transfer Protocol Secure.

    Il ajoute à HTTP deux propriétés essentielles à la sécurisation des communications :

    - le chiffrement des données qui garantit la confidentialité des échanges.
    - l'authenticité du serveur grace à un certificat d'authenticité délivré par une autorité de certification.

    Le protocole HTTPS fonctionne avec le protocole TLS qui implique plusieurs étapes:

    - Vérification de l'authenticité du serveur par le client.
    - Création de la clé commune du chiffrement symétrique à l'aide d'un algorithme de chiffrement asymétrique offrant des garanties contre l'attaque de l'homme du milieu (man in the middle).
    - Échange des données sécurisé à l'aide du chiffrement symétrique avec la clé créee.

13. HTTPS permet d'utiliser un chiffrement symétrique très sur et rapide: AES. Il garantit de plus l'authenticité du serveur. HTTPS offre donc des fonctions supplémentaires à un algorithme de chiffrement asymétrique en combinant : authenticité, rapidité et confidentialité.

Partie C

14. Les adresses du réseau privé `192.168.110.0/24` s'obtiennent en vérifiant si la partie réseau (préfixe) est la même que celle de l'adresse `192.168.110.0` sur la longueur du masque de 24 bits : `255.255.255.0`. Elles commencent donc toutes par `192.168.110`.

L'adresse précisée dans la commande ping : `192.168.100.115` ne fait donc pas partie de ce réseau. Sans routeur qui connecte les deux sous-réseau, cela explique que Marc n'obtienne aucun paquet réponse aux 4 envoyés et donc l'erreur obtenue.

Pour vérifier la connexion avec le poste de travail 115 du même sous-réseau, Marc peut taper la commande :

`ping 192.168.110.115`

15. Si le masque de sous-réseau devient `11111111.11111111.11111111.11100000`, il sera sur 27 bits. Pour trouver son écriture décimale, nous allons convertir le dernier octet,`11100000`, en décimal : 224 (128 + 64 + 32).

    Le masque de sous-réseau sera alors en décimal : `255.255.255.224`.

16. Le masque `255.255.255.224` correspond à 27 bits pour la partie réseau.

    Il reste 32 - 27 = 5 bits pour la partie hôte.

    Le nombre total d'adresses = 2^5 = 32 adresses

    2 adresses sont réservées :

    - 1 pour l'adresse du sous-réseau : `192.168.110.96`
    - 1 pour l'adresse de diffusion (broadcast) : `192.168.110.127`

    Il y 30 adresses IPv4 utilisables sur ce sous-réseau.

17. Pour avoir la la représentation binaire du nombre 134, nous pouvons utiliser la méthode des divisions euclidiennes successives par 2 ou trouver par une méthode gloutonne les puissances de 2 qui composent 134. 134 = 128 + 4 + 2.

    Donc 134 s'écrit en binaire sur un octet : '10000110'.

18. Si Zoé possède l'adresse IPv4 `192.168.110.134` avec un masque de sous-réseau `255.255.255.224`, les machines qui sont connectées au même sous-réseau auront une adresse IPv4 avec les mêmes 27 premiers bits que ceux de l'adresse de Zoé donc une adresse en 192.168.110 et le dernier octet qui commence par les 3 bits `100`ceux de 134.

    - Commande N°1 : 115 à une écriture binaire qui commence par `0` et non `1`. L'adresse `192.168.110.115` n'est donc pas sur le même sous-réseau que Zoé.
    - Commande N°2 : 153 s'écrit `10011001` qui commence bien par `100`. L'adresse `192.168.110.153` est donc bien sur le même sous-réseau que Zoé.

    La commande N°2 `ping 192.168.110.153` permet donc la réponse `4 packets transmitted, 4 received, 0% packet loss, time 3002ms`, même sans routeur.

Il est indiqué que les sous-réseaux sont réliés par des routeurs. Aussi avec un routeur qui fonctionne, la commande N°1 pourrait aussi donner la même réponse.