25NSIJ1JA1

Exercice 1 (6 points)

Cet exercice porte sur la décidabilité, l'algorithmique et la programmation en Python.

1. Après l'exécution du programme contenu dans la variable `programme1`, `x` vaut `0` et `y` vaut `20`.
2. Un programme Python est un fichier texte et peut donc être vu comme une chaîne de caractères.
3. Les programmes 3 et 5 terminent, les 4 et 6 ne terminent pas.
4. La fonction `arret_essai1` renvoie `True` si le programme passé en paramètre termine, ce qui permet de répondre partiellement au problème. Cependant si ce programme passé en paramètre ne termine pas alors la fonction `arret_essai1` ne termine pas et ne renvoie donc pas le `False` attendu.
5. Le principe de l'algorithme Boyer-Moore est de chercher un `motif` de longueur `m` dans un `texte` de longueur `t` en alignant le motif avec le début du texte, en position `i=0` puis de comparer `motif[j]` avec `texte[i+j]` en partant de `j=m-1` vers `j=0`. Si on arrive à `j==0` avec des comparaisons vraies, on a trouvé une occurrence du motif et on renvoie `True`. Sinon, en fonction de l'indice `j` où la comparaison échoue, on peut décaler le motif d'un nombre de positions qui dépend de la lettre `texte[i+j]`. On recommence à la nouvelle position ; si on dépasse la fin du texte (si `i>t-m`) alors on renvoie `False`.
6. On écrit la fonction demandée.

```
def arret_essai2(programme):
    return not recherche('while',programme)
```

7. Les programmes 7 et 8 correspondent aux deux cas à mettre en évidence. On a utilisé une fonction récursive dans le programme 7, qui crée l'équivalent d'une boucle infinie.

```
programme7="""
def boucle_infinie():
    boucle_infinie()
boucle_infinie()
"""

programme8="""
n=10
while n>0:
    n=n-1
"""
```

8. On écrit la fonction demandée.

```
def terminaison_inverse(programme):
    if arret(programme):
        boucle_infinie()
```

9. Si `programme_paradoxal` termine, alors `programme_paradoxal` ne termine pas
   puisque `programme_paradoxal="terminaison_inverse(programme_paradoxal)"`, par construction de la fonction `terminaison_inverse`.
   Si `programme_paradoxal` ne termine pas, alors `programme_paradoxal` termine
   puisque `programme_paradoxal="terminaison_inverse(programme_paradoxal)"`, par construction de la fonction `terminaison_inverse`.
   Mais `programme_paradoxal` ne peut pas à la fois terminer et ne pas terminer, c'est absurde.
10. On a prouvé par l'absurde qu'une fonction `arret` telle que décrite ne peut exister.
11. Ce résultat est un résultat général qui n'est pas dû aux limitations de langage Python.

Exercice 2 (6 points)

Cet exercice porte sur les arbres et la compression d'un fichier texte.

Partie A

1. Avec l'encodage ISO8859-1, chaque caractère est codé sur un octet [1] donc `txt="SIX ANANAS"`, qui comporte dix caractères en comptant l'espace, est codé sur dix octets soit quatre-vingt bits.
2. Le codage de `txt` est 53495820414E414E4153.

Partie B

3. Le tableau d'occurrences associé à la chaîne `txt` est :

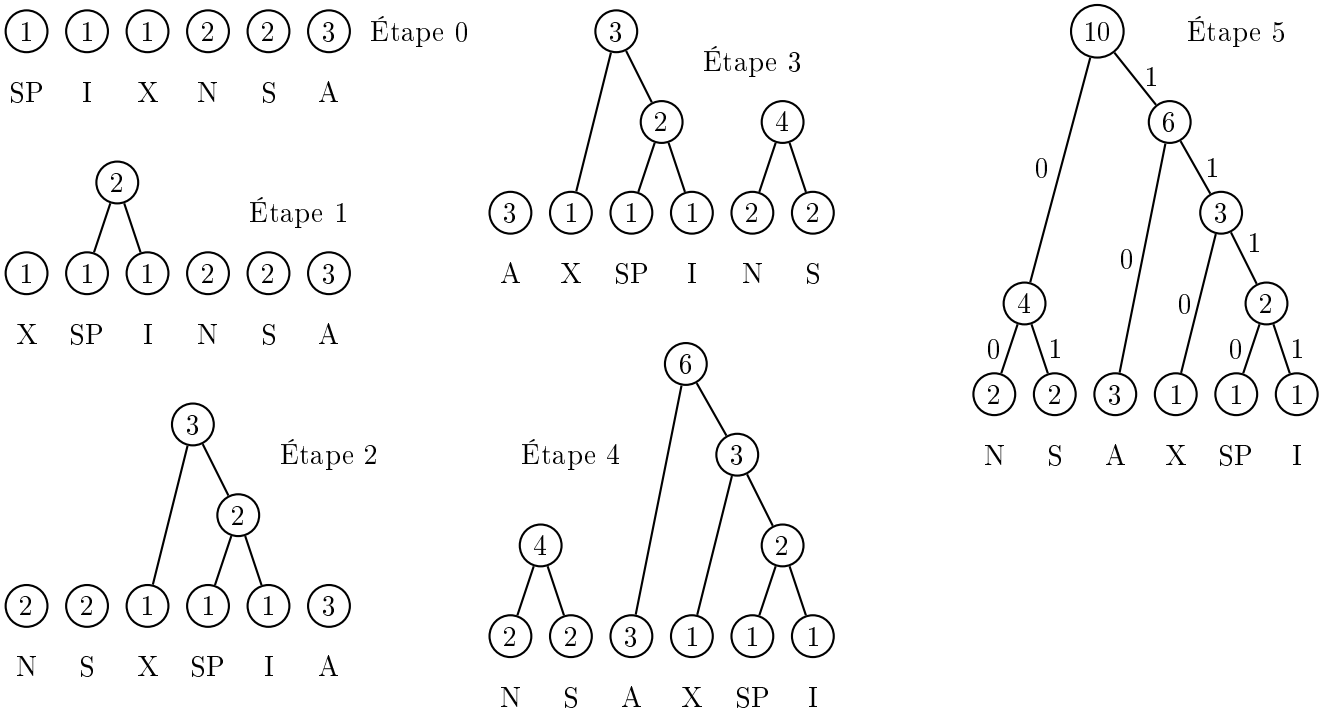
Symbole	SP	A	I	N	S	X
Nombre d'occurrences	1	3	1	2	2	1

dans l'ordre alphabétique des symboles.

4. La somme des nombres d'occurrences est le nombre de caractères de la chaîne, donc le nombre d'octets utilisé pour coder cette chaîne avec l'encodage ISO8859-1.
5. On a complété le code.

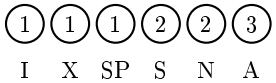
```
def occurrence(texte):
    dico={}
    for lettre in texte:
        if lettre in dico:
            dico[lettre] = dico[lettre]+1
        else:
            dico[lettre] = 1
    return dico
```

6. On construit l'arbre de Huffman, en partant du tableau de la question 3 [2].



7. Le poids de la racine de cet arbre, 10, est la longueur de la chaîne.

1. Ce n'est pas le cas en général, p.ex UTF8 code chaque caractère sur un à quatre octets.

2. L'ordre initial aurait été 

avec le `dict` obtenu par la fonction de la question 5.

8. Il s'agit d'un parcours en profondeur pour lequel on stockera un couple `symbole:code` pour chaque feuille parcourue.
9. La table de codage est donnée par ce tableau.

Symbole	N	S	A	X	SP	I
Code	00	01	10	110	1110	1111

10. Le symbole N est codé sur deux bits, le symbole I sur quatre bits, donc le codage de Huffman est de longueur variable.
11. La chaîne `txt` est codée par 011111110111010001000 1001, sur 25 bits.
12. Le taux de compression est $\frac{80-25}{80} = 0,6875$ soit 68,75 % (malheureusement il faut aussi garder trace de l'arbre de décodage).

Exercice 3 (8 points)

Cet exercice porte sur les dictionnaires et leurs algorithmes associés, le traitement de données en table, la sécurisation des communications et la programmation en général.

Partie A

1. La requête `SELECT "id-kimono" FROM location WHERE fin='';` convient.
2. La requête `SELECT COUNT("id-kimono") FROM kimono WHERE "taille-kimono"=130;` convient.
3. La requête `SELECT nom, prenom FROM adherent JOIN location ON adherent."numero-licence"=location."numero-licence" WHERE "id-kimono"=42 AND fin='';` convient.
4. La requête `UPDATE adherent SET "taille-adherent"="taille-adherent"+10 WHERE "taille-adherent"<160;` convient.
5. Les requêtes `DELETE FROM location WHERE "id-kimono"=25;` puis `DELETE FROM kimono WHERE "id-kimono"=25;`, dans cet ordre, conviennent.

NOTA BENE L'utilisation de - dans les noms des champs oblige à utiliser des *identificateurs délimités* par des guillemets doubles dans les requêtes ci-dessus.

Partie B

6. Eddie Nirrer peut avoir comme numéro de licence M12102021NIRRE01 par exemple.
7. Cet adhérent est né le 23 septembre 1974, il peut s'appeler Marti.
8. L'expression `tab_adherents[1]['prenom']` permet d'accéder à la chaîne `'STEPHANIE'`.
9. On accède à `'F03071997DUPON01'` par `tab_adherents[0]['numero-licence']`.
10. On complète la fonction demandée.

```
def nombre_adherents(table, annee):
    compteur = 0
    for adherent in table:
        if adherent['annee'] == annee:
            compteur = compteur + 1
    return compteur
```

11. On écrit la fonction demandée.

```
def adherent_plus_age(table):
    res = [table[0]]
    annee = table[0]['annee']
    for i in range(1, len(table)):
        adherent = table[i]
        if adherent['annee'] == annee:
            res.append(adherent)
        if adherent['annee'] < annee:
            res = [adherent]
            annee = adherent['annee']
    return res
```

12. On écrit la fonction demandée.

```
def verification_licence(adherent):
    licence = adherent['numero-licence']
    if extraire(licence, 0, 1) != adherent['sexe']:
        return False
    if extraire(licence, 1, 3) != adherent['jour']:
        return False
    if extraire(licence, 3, 5) != adherent['mois']:
        return False
    if extraire(licence, 5, 9) != adherent['annee']:
        return False
    if extraire(licence, 9, 14) != extraire(adherent['nom'], 0, 5):
        return False
    return True
```

ou bien

```
def verification_licence(adherent):
    licence = adherent['numero-licence']
    return extraire(licence, 0, 1) == adherent['sexe'] \
        and extraire(licence, 1, 3) == adherent['jour'] \
        and extraire(licence, 3, 5) == adherent['mois'] \
        and extraire(licence, 5, 9) == adherent['annee'] \
        and extraire(licence, 9, 14) == extraire(adherent['nom'], 0, 5) \
```