Codages optimaux par des mots binaires. *Barème sur 23.5 pts, durée 1h15*

Les chaînes de caractéres en PYTHON (type : `str`) sont de la forme `t = "x0x1...xn-1"`.

Un **texte** est une chaîne de caractères dont les lettres sont celles de l'alphabet usuel.

Un **mot binaire** est une chaîne de caractères dont les lettres sont 0 ou 1.

On peut récupérer la longueur d'une chaîne de caractères s par `n=len(s)` et les caractères par `s[k]`, où $0 \leq k < n$.

Les chaînes de caractères sont des objets non mutables.

L'opérateur `+` permet de concaténer des chaînes de caractères.

Par exemple, `"pre"+"pa"` renvoie `"prepa"` et `"prepa"+""` renvoie `"prepa"`

Un mot binaire v est un préfixe d'un mot binaire u si il existe un mot w tel que $u = v + w$.

Par exemple, les préfixes de `"0010"` sont `"", "0", "00", "001"` et `"0010"`.

Partie I. Occurrences

1) On considère le programme suivant :

```
01.     def tableau(t:str -> list,list) :
02.          L = [] ; M = []
03.          for y in t :
04.               if not(y in L) : L.append(y)
05.          for x in L :
06.               k = 0
07.               for y in t :
08.                    if x == y :
09.                         k += 1
10.               M.append((x,k))
11.          return L,M
```

a) [1.5 pt] Décrire les valeurs renvoyées de L et M.

b) [1.5 pt] Préciser la complexité de `tableau(t)` dans le pire cas en fonction de la longueur n de t.

Donner un exemple où cette complexité est atteinte.

2) a) [2 pts] Écrire une fonction `occurrences` qui étant donné un texte t renvoie le dictionnaire w dont les clés sont les lettres x apparaissant dans t et dont la valeur $w(x)$ est le nombre d'occurrences de x dans t.

On demande une fonction de complexité linéaire en la longueur du texte `t`.

b) [2.5 pts] On suppose que w admet au moins deux clés distinctes.

Écrire une fonction `minima(w)` qui étant donné le dictionnaire des occurrences w renvoie un couple (x, y) des clés tel que $w(x)$ est minimum et que $w(y)$ est minimum (en excluant x).

Par exemple, `minima({"a":5,"p":2,"r":1,"e":3}` renvoie `("r","p")`.

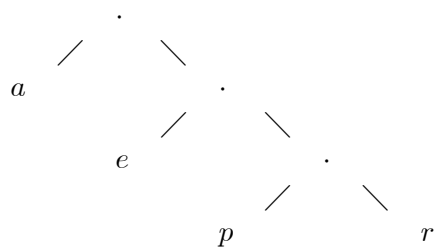
Remarque : On pourra faire intervenir la liste `w.keys()` des clés de `w`.

Partie II. Arbre d'un codage binaire

Un codage possible d'un texte par un mot binaire consiste à attribuer à chaque lettre de l'alphabet un mot binaire (c'est-à-dire une séquence de 0-1) de sorte qu'aucune séquence ne soit un préfixe d'une autre : Par exemple, un texte contenant les quatre lettres a, e, p, r peuvent être codés respectivement par `"0"`, `"10"` et `"110"` et `"111"`. Le mot binaire codant le texte est la concaténation des mots binaires associés à chaque lettre. Par exemple, le texte `"prepa"` est codé par `"110"+"111"+"10"+"110"+"0"`, c'est-à-dire `"110111101100"`.

Remarque : Il est essentiel pour pouvoir décoder un code binaire qu'aucun code d'une lettre ne soit préfixe du code d'une autre lettre : par exemple, si on codait `"e"` par "1" et `r` par "11", le mot binaire "111" représenterait à la fois les textes `"eee", "re"` et `"er"`, ce qui serait problématique.

Un tel codage peut être représenté par un arbre : à chaque niveau, le fils gauche est associé au bit 0 et le fils droit au bit 1 ; à tout chemin dans l'arbre est associé un mot binaire.



Les lettres sont donc associées à des feuilles de l'arbre (il y a une seule feuille sur chaque chemin de l'arbre).

L'arbre du codage est défini en PYTHON par un dictionnaire dont les clés sont les sommets de l'arbre, c'est-à-dire tous les mots associés à ces sommets (qui sont les préfixes des mots codant les lettres).

Dans l'exemple précédent, les clés sont donc les mots `""`, `"0"`, `"1"`, `"10"`, `"11"`, `"110"` et `"111"`.

La valeur est `None` lorsqu'il s'agit d'un noeud interne et la valeur est la lettre lorsqu'il s'agit d'une feuille.

L'exemple précédent est donc défini en PYTHON par le dictionnaire

```
arbre = {"":None,"0":"a","1":None,"10":"e","11":None,"110":"p","111":"r"}
```

3) [1.5 pt] Écrire une fonction `alphabet` qui étant donné un dictionnaire `arbre` définissant l'arbre du codage renvoie le dictionnaire `lettres` dont les clés sont les lettres du dictionnaire de codage et dont la valeur d'une lettre est son code binaire.

Par exemple `alphabet(arbre)` renvoie le dictionnaire `lettres` défini par `{"a":"0":,"e":"10":"p":"110","r":"111"}`

4) a) [1.5 pt] Écrire une fonction `prefixe` qui étant donnés deux mots binaires u et v renvoie `True` ssi le premier mot u est préfixe du second mot v.

Ainsi, `prefixe("10","1010")` renvoie `True` et `prefixe("10","111")` renvoie `False`.

Remarque: Le mot vide `""` est préfixe de tout mot, et chaque mot est préfixe lui-même.

b) [1.5 pt] Écrire une fonction `verif` qui étant donné un dictionnaire `lettres` dont les clés sont des lettres et les valeurs leurs codes binaires renvoie `True` ssi aucun code binaire n'est préfixe d'un autre.

Par exemple, `verif{"a":"0":,"e":"10":"p":"110","r":"111"}` renvoie `True`.

c) [1.5 pt] On suppose connue une fonction `sorted` permettant de trier une liste de textes selon l'ordre lexicographique.

Par exemple, `sorted(["0","10","000",""]` renvoie `["","0","000","10"]`.

Proposer une nouvelle version `verif_bis` de la fonction définie au b) de telle sorte que la complexité (sans tenir compte de `sorted`) soit linéaire en le nombre total de bits (c'est-à-dire la somme des longueurs des mots binaires codant les différentes lettres).

d) [1.5 pt] Pour définir une telle fonction `sorted`, il faut définir au préalable une fonction de comparaison pour l'ordre lexicographique : écrire une fonction `plus_petit` qui étant donnés deux mots binaires u et v renvoie `True` ssi u est inférieur ou égal à v pour l'ordre lexicographique.

Par exemple, `plus_petit("01100","0101111")` renvoie `False`.

Partie III. Codage et décodage

5) [1.5 pt] Écrire une fonction `codage` qui prend comme arguments un texte t (codé par une chaîne de caractère) et un dictionnaire de codage `arbre`, et renvoie le code binaire de t.

Par exemple, `codage("prepa",arbre)` renvoie le mot binaire `"110111101100".`

6) [3.5 pts] Écrire une fonction `decodage` qui prend comme arguments un mot binaire `M` et le dictionnaire de codage `arbre`, et renvoie le texte dont `M` est le codage binaire.

Indication : On parcourt l'arbre à l'aide du dictionnaire en mémorisant le sommet où on se trouve et l'indice du bit du mot en cours de lecture. Lorsqu'on arrive sur une feuille, on récupère la lettre et on repart à la racine de l'arbre.

Partie IV. Codage optimal

On utilisera dans cette question les fonctions définies à la question 2).

7) [3.5 pts] Étant donné un texte t, on peut chercher un codage qui minimise la longueur du mot binaire codant t. L'idée est d'attribuer à chaque lettre x un mot binaire $m(x)$ d'autant plus court que le nombre d'occurrences dans t est élévé. On peut montrer qu'un codage optimal est obtenu par l'algorithme récursif suivant :

- on calcule le nombre d'occurrences $\omega(x)$ de chaque lettre x apparaissant dans t : autrement dit, $\omega(x)$ est le nombre de fois où x apparaît dans t. On suppose dans la suite que t contient au moins deux lettres.

- s'il n'y a que deux lettres x et y apparaissant dans le texte t, un codage optimal est obtenu en codant x par `"0"` et y par `"1"`

- sinon, on détermine les deux lettres x et y pour lesquelles les nombres d'occurrences sont les plus petits

- on considère alors le texte t' obtenu en remplaçant x et y par une même lettre : on peut par exemple remplacer y par x : on obtient ainsi un texte t' contenant une lettre de moins.

- on détermine (par récurrence sur le nombre de lettres) un codage optimal `m_prime` pour t'.

- un codage optimal de t est obtenu en définissant la fonction `m` par

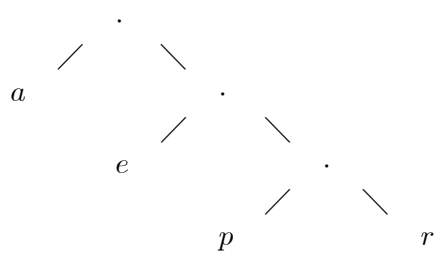
$$\begin{cases} \texttt{m(x) = m\_prime(x) + "0"} \\ \texttt{m(y) = m\_prime(x) + "1"} \\ \texttt{m(z) = m\_prime(z)} \text{ pour toute lettre } z \notin [x, y\} \end{cases}$$

On considère un texte t admettant au moins deux lettres distinctes. Écrire une fonction récursive `code_optimal(w)` qui étant donné un dictionnaire des occurrences ω du texte t renvoie un dictionnaire `lettres` représentant un codage optimal du texte t.

Par exemple, `arbre_optimal({"a":5,"p":2,"r":2,"e":3})` peut renvoyer le dictionnaire

`lettres = {"a":"0","p":"110","r":"111","e":"10"}`

qui est associé au codage optimal représenté par l'arbre :



8) (★) *Question supplémentaire*

Montrer que l'algorithme décrit au 7) renvoie bien un code minimisant la longueur du mot binaire codant t.

Exercice. Bases de données

On considère une base de données d'un ensemble de magasins de fleurs. La base de données est constituée de trois tables, dont le modèle relationnel est le suivant :

- `Magasins (idm, nom, ville)`
- `Ventes (idv, idm, idf, prix)`
- `Fleurs (idf, nom, couleur)`

avec :

— `idm`, `idv`, `idf` sont des entiers, soulignés lorsqu'il s'agit de clés primaires de la table, non soulignés lorsqu'il s'agit de clés étrangères.

- Dans la table `Magasins`, l'attribut `nom` est le nom du magasin, et l'attribut `ville` le nom de la ville dans laquelle il se situe. Ce sont deux chaînes de caractères.
- Dans la table `Ventes`, `prix` est le prix (flottant) de la fleur vendue dans ce magasin-là
- Dans la table `Fleurs`, `nom` est le nom de la fleur, `couleur` est la couleur de la fleur. Ce sont des chaînes de caractères.

Q1. Est-ce que `nom` peut jouer le rôle de clé primaire dans la table `Magasins` ? Pourquoi ?

Écrire en langage SQL les requêtes qui donnent :

Q2. Les couleurs existantes des tulipes

Q3. Pour chaque ville le nombre de magasins

Q4. Le prix moyen des tulipes

Q5. Le nom des magasins de Paris qui vendent des roses blanches

Q6. Le nom des villes ayant au moins trois fleuristes

Q7. Le nom des couples de villes ayant au moins deux fleuristes vendant des fleurs de la même couleur. Il s'agit de renvoyer les couples (i, j) d'identifiants de magasins, avec $i < j$ (pour l'ordre lexicographique déjà implémenté en SQL).

Corrigé

Q1. La clé primaire est le choix d'une clé par le créateur de la base de données choisie pour servir de moyen d'identification pour la suite du problème.
`nom` ne peut pas jouer le rôle de clé primaire dans la table `Magasins` car il peut y avoir plusieurs magasins de même nom, et `nom` ne permettrait pas de distinguer chaque ligne de façon unique.

Q2.

```
SELECT couleur
FROM Fleurs
WHERE nom = 'tulipe'
```

Q3.

```
SELECT ville , COUNT (idm)
FROM Magasins
GROUP BY ville
```

Q4.

```
SELECT AVG (prix)
FROM Ventes
JOIN Fleurs ON Fleurs.idf = Ventes.idf
WHERE Fleurs.nom = 'Tulipe '
```

Q5.

```
SELECT Magasins.nom
FROM Magasins
JOIN Ventes ON Ventes.idm = Magasins.idm
JOIN Fleurs ON Fleurs.idf = Ventes.idm
WHERE Fleurs = 'Rose' AND couleur = 'blanc' AND ville = 'Paris '
```

Q6.

```
SELECT ville
FROM Magasins
GROUP BY ville
HAVING COUNT(idm) >=3
```

Q7.

```
SELECT DISTINCT Mag1.idm,Mag2.idm
FROM (Magasins JOIN Ventes ON Magasins.idm = Ventes.idm AS Mag1)
JOIN (Magasins JOIN Ventes ON Magasins.idm = Ventes.idm AS Mag2)
ON Mag1.idf = Mag2.idf
WHERE Mag1.idm < Mag2.idm
```