DS Binaire-Circuit-Codage

Durée de l'épreuve : **01h50**

L'usage de la calculatrice n'est pas autorisé.
Le candidat répond sur feuilles doubles numérotées et garde l'énoncé.
Les traces de recherche, même incomplètes ou infructueuses, seront valorisées.
La qualité de la rédaction, la clarté et la précision des raisonnements seront prises en compte.

Exercice 1 — 10 points

1. Coder tous les entiers de 10 à 20 (base 10) en binaire et en hexadécimal. 1 pt

2. Coder 300 (base 10) en binaire. 1,5 pt

3. Coder ABCD (base 16) en binaire. 1,5 pt

4. Implémenter une fonction Python ***decode*** qui : 2 pt

- reçoit un paramètre ***n_bin***, de type chaîne de caractères, représentant un nombre binaire ;
- retourne la valeur de ce nombre en décimal, en type entier.

```
>>> assert decode("10") == 2
>>> assert decode("1111 1111") == 255
```

5. Implémenter une fonction Python ***caracteres*** : 2 pt

- reçoit en paramètre deux caractères ASCII ***c1*** et ***c2***, de type chaîne de caractères ;
- retourne tous les caractères ASCII, sous forme d'une chaîne de caractères, dont le code est compris entre celui de ***c1*** et celui de ***c2***.

```
>>> assert caracteres("a", "e") == "abcde"
>>> assert caracteres("8", "B") == "89:;<=>?@AB"
```

6. Implémenter une fonction Python ***mots*** qui : 2 pt

- reçoit en paramètre un texte ***texte***, de type chaîne de caractères ;
- retourne tous les mots de ce texte, sous forme d'une liste.

```
>>> assert mots("Qui vivra verra") == ["Qui", "vivra", "verra"]
>>> assert mots("Qui as décrypté le message? C'était du Vigenère") == \
       ["Qui", "as", "décrypté", "le", "message", "c", "était", "du", "Vigenère"]
```


Exercice 2 — 5 points

Un comparateur est un circuit combinatoire qui :

- prend en entrée deux entiers en binaire a et b ;
- indique en sortie si :
  - $a = b$: sortie E à 1, les autres sorties à 0 ;
  - $a < b$: sortie I à 1, les autres sorties à 0 ;
  - $a > b$: sortie S à 1, les autres sorties à 0 ;

1. On considère ici un comparateur un bit, chaque nombre est sur un bit, on a donc 2 entrées et 3 sorties.
   - **a.** Établir la table de vérité de ce comparateur : $|\,b\,|\,a\,||\,S\,|\,I\,|\,E\,|$. 1 pt
   - **b.** Établir les fonctions logiques de ses sorties. 1 pt
   - **c.** Tracer le circuit de ce comparateur à l'aide de portes logiques. 1 pt
2. On considère ici un comparateur deux bits, chaque nombre est sur deux bits, on a donc 4 entrées et 3 sorties.
   - **a.** Établir la table de vérité de ce comparateur : $|\,b_1\,|\,b_0\,|\,a_1\,|\,a_0\,||\,S\,|\,I\,|\,E\,|$. 1 pt
   - **b.** Tracer le circuit de la sortie E ce comparateur à l'aide de circuits comparateurs un bit et de portes logiques. 1 pt

Exercice 3 — 5 points

Trow d'emreurs di suntaxm comstote Théou !

Pour y remédier, on va implémenter un début de fonctionnalité de complétion automatique en suggérant une liste de mots *proches* d'une chaîne de caractères.

1. Implémenter une fonction qui vérifie si une lettre est dans un mot : 1 pt

```
def lettreIn(lettre: str, mot: str) -> bool:
    ''' Renvoie True si le caractère lettre est dans \
            la chaîne de caracactère mot, False sinon'''
```

```
assert lettreIn('a', 'travail') == True
assert lettreIn('e', 'travail') == False
```

2. Implémenter une fonction qui vérifie si une chaîne de caractères correspond au début d'un mot : 1 pt

```
def estDebut(chaine: str, mot: str) -> bool:
    '''Renvoie True si chaine est le début de mot, False sinon'''
```

```
assert estDebut('tra', 'travail') == True
assert estDebut('tro', 'travail') == False
assert estDebut('vai', 'travail') == False
```


3. Implémenter une fonction qui : 1 pt

- prend en paramètres :
  - une chaîne de caractères ***chaine*** ;
  - une liste de mots ***listeMots*** ;
- renvoie la liste des mots parmi ***listeMots*** pour lesquels ***chaine*** correspond au début.

```
def motsDebut(chaine: str, listeMots: list) -> list:
    ''' renvoie la liste des mots de listeMots dont chaine est le début'''
```

```
listeMots = ['travail', 'traverser', 'taquiner', 'entraver']
assert motsDebut('t', listeMots) == ['travail', 'traverser', 'taquiner']
assert motsDebut('tra', listeMots) == ['travail', 'traverser']
assert motsDebut('tram', listeMots) == [ ]
```

4. La distance de hamming est le nombre de caractères différents aux mêmes emplacements entre deux chaînes de caractères en tenant (y compris les caractères qui n'existent pas dans une des deux chaînes de caractères).
Implémenter une fonction qui renvoie la distance de hamming entre une chaîne de caractères et un mot : 1 pt

```
def distance(chaine: str, mot: str) -> bool:
    ''' Renvoie la distance de hamming entre chaine et mot'''
```

```
assert distance('travaux', 'travail') == 2
assert distance('tra', 'travail') == 4
assert distance('txavxilx', 'travail') == 3
```

5. Implémenter une fonction qui vérifie si une chaîne de caractères est une sous-chaîne de caractères d'un mot (n'importe où, pas forcément au début) : 1 pt

```
def myIn(chaine: str, mot: str) -> bool:
    '''++ Renvoie True si chaine est une sous-chaîne \
                        de caractères de mot, False sinon'''
```

```
assert myIn('tra', 'travail') == True
assert myIn('tro', 'travail') == False
assert myIn('vai', 'travail') == True
```

NB : Romée, si tu utilises la primitive python ***in***, c'est Zéro ...

Exercice bonus (optionnel)

Programmer une fonction qui décode un nombre écrit en chiffres romains en écriture décimal et une fonction qui code un nombre décimal en chiffres romains.

On a les correspondances suivantes : "I" : 1, "V" : 5, "X" : 10, "L" : 50, "C" : 100, "D" : 500, "M" : 1000.

Et les équivalences ci-après : $(XVII)_R = (17)_{10}$ $(XIV)_R = (14)_{10}$ $(CXLIV)_R = (144)_{10}$

