Partie QCM : n'oubliez pas de reporter vos réponses sur la première page !

QCM sur la partie programmation.

Question 11. Quelle condition est suffisante pour que l'expression `a + b` soit toujours évaluée à une valeur de type `int` ?

(A) Que l'une des deux variables `a` ou `b` soit de type `int`

(B) Que `a` *et* `b` soient les deux de type `int`

(C) Que `a` soit de type `int`, et `b` de type `str` avec une valeur convertissable en `int` (comme `"42"`)

(D) Aucune de ces conditions

Question 12. Laquelle de ces lignes ne va pas forcément afficher le contenu de `a` sur le terminal, étant donné que `a` est de type `str` ?

(A) `print("" + a)`

(B) `print(f"{a}")`

(C) `print(str(a))`

(D) `print(a.__repr__)`

Question 13. Si les expressions `condition1()` et `condition2()` sont toutes les deux évaluées à `True`, qu'affiche le code suivant ?

```
if condition1():
    print("A")
elif condition1() and condition2():
    print("B")
```

(A) `A`

(B) `B`

(C)
```
A
B
```

(D) Rien

Question 14. Si `B` est une sous-classe de `A`, quelle ligne sera forcément indiquée comme erronée par le vérificateur de type (*linter*) ?

(A) `value: A = B()`

(B) `value: B = B()`

(C) `value: B = A()`

(D) Aucune de ces lignes

Question 15. Laquelle de ces boucles n'affichera pas les index des éléments d'une `List[int]` stockée par la variable `numbers` ?

(A)
```
for i in range(len(numbers)):
    print(i)
```

(B)
```
for i, elem in enumerate(numbers):
    print(i)
```

(C)
```
for i in numbers:
    print(i)
```

(D)
```
i: int = 0
while i < len(numbers):
    print(i)
    i += 1
```

Question 16. Laquelle de ces opérations n'est pas possible avec une liste ?

(A) Remplacement d'une série de valeurs contiguës (voisines) par une seule

(B) Obtention de la longueur de la liste

(C) Ajout d'une valeur à la fin de la liste et redimensionnement automatique

(D) Aucune des opérations ci-dessus

Question 17. Avec Tkinter, quel extrait de code créera correctement un `Button` inséré dans `root` qui affiche `action!` sur le terminal lors d'un clic ? La fonction `action` est définie comme suit :

```
def action() -> None:
    print("action!")
```

(A)
```
button = Button(root, command=action())
```

(B)
```
button = Button(root, command=lambda: action)
```

(C)
```
button = Button(root, command=action(button))
```

(D)
```
button = Button(root, command=action)
```

Question 18. Quelle affirmation est erronée, sachant que `d` est un `Dict[int, str]` et que `len(d) == n` ?

(A) `d` contient `n` clés

(B) `d` contient `n` clés, toutes forcément différentes

(C) `d` contient `n` valeurs

(D) `d` contient `n` valeurs, toutes forcément différentes

Question 19. Quel appel de la fonction `my_function` n'est pas forcément possible, sachant qu'elle est définie comme ci-dessous ?

```
def my_function(a: int, b: str, c: bool = True, d: bool = False) -> None:
    print(f"function was called with: {a} {b} {c} {d}")
```

(A) `my_function(0, "4", c=d)`

(B) `my_function(b="4", a=0, c=False)`

(C) `my_function(4, "True")`

(D) `my_function(0, b="4", d=True)`

Question 20. Quelle affirmation sur les threads est incorrecte ?

(A) Les threads permettent d'exécuter plusieurs séquences d'instructions de manière concurrente (voire vraiment parallèle si la machine le permet)

(B) C'est le système d'exploitation qui décide de quand un thread pourra exécuter son code et de quand il sera obligé de faire une pause

(C) Un thread ne peut pas être créé par un autre thread

(D) Dans une application à interface graphique créée avec Tkinter, il y a toujours un thread unique responsable de la gestion des événements

Partie questions ouvertes : répondez sur les feuilles ci-dessous
Pour ces questions ouvertes, vous n'avez pas besoin d'écrire les `imports`*.*
Attention à indenter clairement vos blocs selon les lignes verticales!

Problème 1. (18 points)

a) Compléter ci-dessous la fonction `compute_frequency_map`, qui prend comme paramètre une phrase représentée par la liste de ses mots avec une `List[str]`, et retourne une valeur de type `Dict[str, float]` contenant, comme clés, les mots *distincts* $\mathbf{w}_1, \ldots, \mathbf{w}_n$ de la liste et, comme valeurs liées à ces clés, les *probabilités d'apparition* $p_1, \ldots, p_n$ de ces mots dans la phrase. (Pour simplifier, on peut supposer que chaque mot est écrit entièrement en lettres minuscules.)

Exemple : Si la phrase d'entrée est donnée par la liste `["un", "chat", "est", "un", "chat"]`, alors la valeur retournée doit relier, dans le `Dict` retourné, `"un"` à la valeur `0.4`, `"chat"` à la valeur `0.4` et `"est"` à la valeur `0.2`.

Votre réponse :

```
def compute_frequency_map(words: List[str]) -> Dict[str, float]:
```

b) Compléter ensuite la fonction `compute_entropy`, qui prend un seul paramètre de type `Dict[str, float]` comme construit en a) et retourne l'entropie H correspondante (calculée ici mot par mot et non lettre par lettre). Vous pouvez utiliser l'expression `math.log2(p)` pour calculer $\log_2(p)$.

Exemple : L'expression suivante une fois évaluée donnerait une valeur d'environ 1.52193 :
`compute_entropy(compute_frequency_map(["un", "chat", "est", "un", "chat"]))`

Votre réponse :

```
def compute_entropy(probs: Dict[str, float]) -> ______:
```

On suppose maintenant qu'on dispose des deux fonctions suivantes (*notez bien que vous n'avez pas à les écrire !*) :

— `def build_huffman_code(probs: Dict[str, float]) -> Dict[str, str]`,
qui permet d'exécuter l'algorithme de Huffman et donc qui, en se basant sur les probabilités d'apparition $p_1, \ldots, p_n$ ci-dessus transmises dans le `Dict` en paramètre, construit une autre `Dict` qui associe à chaque mot $\mathbf{w}_i$ un mot de code c_i constitué des caractères `'0'` et `'1'` ;

— `def encode_words(words: List[str], code: Dict[str, str]) -> str`,
qui à partir de la liste de mots d'origine et du code produit, retourne un string binaire S, composé de caractères `'0'` et `'1'`, qui représente l'encodage de la phrase.

Exemple : La valeur de la variable `code` dans ces lignes est le `Dict` représentée ci-dessous, et `message` vaudra `"10100101"` :

```
words: List[str] = ["un", "chat", "est", "un", "chat"]
probs: Dict[str, float] = compute_frequency_map(words)
code: Dict[str, str] = build_huffman_code(probs)
message: str = encode_words(words, code)
```

mot (clé dans le Dict)	`"un"`	`"chat"`	`"est"`
mot de code (valeur dans le Dict)	`"1"`	`"01"`	`"00"`

c) Compléter la fonction `decode_string`, qui accepte comme paramètres (1) un tel string binaire S et (2) le `Dict` *inversé* de celui construit par `build_huffman_code` (donc où chaque clé est un mot de code c_i et chaque valeur associée est le mot $\mathbf{w}_i$ correspondant), et retourne la liste de mots d'origine décodés dans une `List[str]`.

Votre réponse :

```
def decode_string(bin_string: str, inverse_code: Dict[str, str]) -> List[str]:
```

d) Quelle relation existe-t-il entre l'entropie H de la phrase d'origine et le nombre de bits de la séquence S ?

Votre réponse :

Résultat :

a) (6 pts)	**b)** (4 pts)	**c)** (6 pts)	**d)** (2 pts)	**Total** (18 pts)

(Laisser en blanc)

Partie questions ouvertes: répondez sur les feuilles ci-dessous
Pour ces questions ouvertes, vous n'avez pas besoin d'écrire les `imports`*.*
Attention à indenter clairement vos blocs selon les lignes verticales!

Problème 2. (12 + 2 points)

a) Complétez la fonction `hamming_distance`, qui prend en paramètres deux strings binaires $\mathbf{c}_1$ et $\mathbf{c}_2$ (donc formés uniquement des caractères `'0'` et `'1'`) de même longueur n, et qui calcule la *distance de Hamming* $d(\mathbf{c_1}, \mathbf{c}_2)$ entre ces 2 mots de code (pour rappel, celle-ci est donnée par le nombre de positions où les bits des mots $\mathbf{c}_1$ et $\mathbf{c}_2$ diffèrent).

Votre réponse:

```
def hamming_distance(c1: str, c2: str) -> ______:
```

b) Complétez la fonction `min_distance`, qui prend en paramètre, dans une `List[str]`, M mots de code $\mathbf{c}_1, \ldots, \mathbf{c}_M$ de longueur n et calcule la *distance minimale* entre ceux-ci. Faites pour cela appel à votre fonction `hamming_distance`. Pour rappel, la distance minimale est donnée par:

$$d = \min_{i \neq j} d(\mathbf{c}_i, \mathbf{c}_j)$$

Votre réponse:

```
def min_distance(code_words: List[str]) -> ______:
```

Considérons maintenant un ensemble de M mots de code $\mathbf{c}_1, \ldots, \mathbf{c}_M$ dont la distance minimale $d \geq 3$ et considérons un autre mot de code $\mathbf{c}$ dont on sait qu'il diffère en au plus une position d'un des mots de code $\mathbf{c}_1, \ldots, \mathbf{c}_M$.

c) Compléter la fonction `closest_code_word`, qui accepte deux paramètres: (1) `code_words`, qui représente M sous forme de `Set[str]`, et (2) $\mathbf{c}$ sous forme de string; et qui retourne le mot de code $\mathbf{c}_j$ qui est le plus proche de $\mathbf{c}$.

Votre réponse:

```
def closest_code_word(code_words: Set[str], c: str) -> ______:
```

BONUS d) Si le nombre M de mots de code est exponentiel en n, quel est l'ordre de complexité (en fonction de n) de votre programme ci-dessus? (Utiliser la notation de Landau $\mathcal{O}(\cdot)$)

Votre réponse:

Résultat:

a) (2 pts)	**b)** (4 pts)	**c)** (6 pts)	**d)** (+ 2 pts)	**Total** (12 pts)

(Laisser en blanc)