Devoir d'informatique n° 1 - Correction

Exercice 1. Pour s'échauffer

C'est exactement la même idée que dans le DM. On profite que la liste soit triée : il suffit de comparer chaque terme avec le suivant.

```
def tous_differents(valeurs: list) -> bool:
    for i in range(len(valeurs) - 1):
        if valeurs[i] == valeurs[i+1]:
            return False
    return True
```

Exercice 2. Autour du séquençage du génome (*d'après CCINP 2020*)

Partie I - Génération d'une séquence d'ADN

On considère la chaîne de caractère `seq = 'ATCGTACGTACG'`.

Q1. • La commande `seq[3]` renvoie le caractère d'indice 3 de `seq`, *i.e.* `'G'`.

• La commande `seq[2:6]` renvoie la chaîne de caractères constituée des éléments d'indices 2 (inclus) à 6 (exclu) de `seq`, *i.e.* `'CGTA'`.

Q2. En faisant attention au décalage d'indice (1 correspond à `'A'` qui est d'indice 0 dans la liste `nucleotides`) :

```
def generation(n: int) -> str:
    seq = ""
    nucleotides = ['A', 'C', 'G', 'T']
    for i in range(n):
        a = rd.randint(1, 5)
        seq = seq + nucleotides[a - 1]
    return seq
```

Q3. La fonction `mystere` prend en argument une chaîne de caractères représentant une séquence d'ADN et renvoie une liste composée des pourcentages d'apparition respectifs de chacune des bases `'A'`, `'C'`, `'G'` et `'T'`.

Q4. À chaque tour de la boucle `while`, la valeur de la variable `i` décroît strictement (de 1). Ainsi au bout d'un nombre fini d'étapes elle prendra une valeur négative et la boucle s'arrêtera. La fonction `mystere` termine donc.

Q5. La fonction `mystere` est de complexité $O(n)$ où n est la longueur de la chaîne `seq` donnée en argument car la boucle `while` est exécutée autant de fois qu'il y a de caractères dans `seq`.

Partie II - Recherche d'un motif

Q6. Il s'agit exactement de l'épreuve 3.07 du challenge.

On peut donner une version avec ou sans utiliser les tranches (la première est plus simple à écrire mais la complexité est plus explicite dans la seconde) :

```python
def recherche_v1(M: str, T: str):
    for i in range(len(T) - len(M) + 1):
        if T[i: i+len(M)] == M:    # mot trouvé
            return i
    return -1
```

```python
def recherche_v2(M: str, T: str):
    for i in range(len(T) - len(M) + 1):
        j = 0                  # compteur nb lettres correspondantes
        while j < len(M) and M[j] == T[i+j]:  # lettre corresp.
            j += 1             # On va à la lettre suivante.
        if j == len(M):  # Correspondance jusqu'à la fin du mot
            return i
    return -1
```

Q7. La complexité de cette fonction de recherche est donnée en $O(nm)$ avec ici $n = 3\times10^9$ et $m = 50$. On obtient ainsi $\boxed{1{,}5 \times 10^{11} \text{ opérations}}$ ce qui correspond à $\boxed{0{,}15 \text{ secondes}}$ si l'ordinateur réalise 10^{12} opérations par seconde.

Q8. Si on découpe la séquence de longueur $n = 3\times10^9$ en morceaux de taille 50, on obtient $\frac{n}{50} = 6\times10^7$ morceaux à tester. D'après la question précédente, chaque test demande 0,15 secondes ce qui donne en tout $0{,}15\times6\times10^7 = \boxed{9 \times 10^6 \text{ secondes}}$ ou encore 2 500 heures soit plus de 100 jours ! Il n'est donc pas envisageable d'utiliser l'algorithme de recherche naïf en pratique pour rechercher les similarités entre deux séquences d'ADN.

Exercice 3. Modélisation numérique d'un matériau magnétique (*d'après CCMP 2022*)
Partie I : Transition paramagnétique/ferromagnétique sans champ magnétique extérieur

Q9. On utilise la syntaxe `from module import fonction` :

```
from math import exp, tanh
from random import randrange, random
```

Q10. L'équation (2) peut se récrire $m - \tanh(m/t) = 0$. Ainsi la fonction f telle que $f(x,t) = 0$ est donnée par $\boxed{f(x,t) = x - \tanh(x/t)}$. En Python, cela s'écrit simplement :

```
def f(x, t):
    return x - tanh(x/t)
```

Q11. On considère la valeur c du milieu de l'intervalle $[a\,;b]$ puis on teste si le changement de signe a lieu dans l'intervalle $[c\,;b]$. Si oui, on se place sur cet intervalle pour la prochaine itération, sinon on se place sur $[a\,;c]$.

Q12. Notons p le nombre d'itérations nécessaires de la boucle `while`. L'intervalle de départ est de longueur $b-a$ et à chaque tour cette longueur est divisée par 2. L'algorithme se termine lorsque la largeur est inférieure ou égale à `eps`. On a ainsi $\frac{b-a}{2^p} \leqslant$ `eps`, autrement dit p est l'entier juste supérieur ou égal à $\log_2\left(\frac{b-a}{\texttt{eps}}\right)$.

Finalement l'algorithme a $\boxed{\text{une complexité en } O\left(\log\left(\frac{b-a}{\texttt{eps}}\right)\right)}$.

Q13. On prend 500 valeurs équiréparties dans l'intervalle $[t_1\,;t_2]$. Pour chacune d'elle, on calcule la valeur de m correspondante : soit $m = 0$ si $t \geqslant 1$, soit via l'algorithme de dichotomie précédent si $t < 1$, avec l'intervalle et la précision donnés par l'énoncé.

```
def construction_liste_m(t1: float, t2: float) -> [float]:
    solutions = []
    pas = (t2 - t1) / 499
    for i in range(500):
        t = t1 + i*pas
        if t >= 1:
            m = 0
        else:
            m = dicho(f, t, 0.001, 1, 10**-6)
        solutions.append(m)
    return solutions
```

Partie II : Modèle microscopique d'un matériau magnétique

II.1. Premiers exemples

Le sujet donne les variables globales `h` et `n` que l'on utilisera régulièrement :

```
h = 100
n = h*h
```

Q14. Trois versions possibles : avec des `append` successifs, en utilisant l'opérateur `*` ou via les listes en compréhension.

```
def ligne(val: int) -> [int]:
    sortie = []
    for i in range(h):
        sortie.append(val)
    return sortie
```

```python
def ligne_v2(val: int) -> [int]:
    return [val] * h
```

```python
def ligne_v3(val: int) -> [int]:
    return [val for i in range(h)]
```

Q15. On crée une liste de h lignes de 1, lignes crées grâce à la fonction précédente.

```python
def initialisation() -> [[int]]:
    grille = []
    ligne_de_1 = ligne(1)
    for i in range(h):
        grille.append(ligne_de_1)
    return grille
```

Q16. On crée séparément des lignes de 1 et de -1 qu'on ajoute de façon alternée à la grille.

```python
def initialisation_anti() -> [[int]]:
    grille = []
    ligne_de_1 = ligne(1)
    ligne_de_moins1 = ligne(-1)
    for i in range(h):
        if i % 2 == 0:
            grille.append(ligne_de_1)
        else:
            grille.append(ligne_de_moins1)
    return grille
```

II.2. Énergie d'une configuration

Q17. Par exemple, pour le voisin de gauche de (i, j), on peut remarquer qu'il a toujours le même indice de ligne i et que son indice de colonne est en général $j-1$ sauf si $j = 0$ auquel cas c'est $h-1$. On peut regrouper ces deux cas en $(j-1)\%h$ car $-1\%h = h-1$.

```python
def liste_voisins(p: (int, int)) -> [(int, int)]:
    i, j = p  # ou si vous préférez i = p[0], j = p[1]
    gauche = (i, (j-1) % h)
    droite = (i, (j+1) % h)
    dessus = ((i-1) % h, j)
    dessous = ((i+1) % h, j)
    return [gauche, droite, dessus, dessous]
```

Si on ne pense pas à utiliser `%`, on peut procéder à des disjonctions de cas pour chaque voisin. Par exemple :

```python
if j > 0:  # pas 0-ème colonne
    gauche = (i, j-1)
else:  # 0-ème colonne donc voisin tout à droite de la grille
    gauche = (i, h-1)
```

Q18. Il suffit de concaténer les listes correspondant à chaque ligne.

```python
def deplier(grille: [[int]]) -> [int]:
    liste = []
    for ligne in grille:
        liste = liste + ligne
    return liste
```

Q19. Remarquons que la 0-ème ligne contient les éléments de `L` d'indices 0 à $h-1$, puis la 1-ère ligne ceux d'indices h à $2h-1$, etc.

```
def replier(L: [int]) -> [[int]]:
    assert len(L) == n
    grille = []
    for i in range(h):
        ligne = []
        for j in range(h):
            ligne.append(L[h*i + j])
        grille.append(ligne)
    return grille
```

Ou avec l'utilisation de tranches :

```
def replier_v2(L: [int]) -> [[int]]:
    assert len(L) == n
    grille = []
    for i in range(h):
        grille.append(L[h*i: h*(i+1)])
    return grille
```

Q20. Chaque ligne complète contient 10 *spins*. Il y a 6 lignes complètes avant, soit 60 *spins*. De plus il est dans la colonne d'indice 4 donc il y a 4 éléments avant.
Il se trouve donc à la position 64 dans `L`.

Q21. Il suffit de généraliser le calcul entrepris à la question précédente.

```
def conversion_coord(p: (int, int)) -> int:
    return p[0]*h + p[1]
```

Ce n'était pas demandé, mais voici la fonction permettant d'obtenir les indices des voisins dans la liste.

```
def liste_voisins_dans_liste(i: int):
    pos = i//h, i%h
    voisins_dans_grille = liste_voisins(pos)
    voisins_dans_liste = []
    for v in voisins_dans_grille:
        i = conversion_coord(v)
        voisins_dans_liste.append(i)
    return voisins_dans_liste
```

Q22. On utilise l'équation (3) en considérant pour chaque *spin* l'ensemble de ses voisins. On somme alors les produits de leurs valeurs.

```
def energie(s):
    L = deplier(s)
    E_totale = 0
    for i in range(n):
        E_i = 0
        voisins_i = liste_voisins_dans_liste(i)
        for v in voisins_i:
            E_i = E_i + L[i]*L[v]
        E_totale = E_totale + E_i
    return -1/2 * E_totale  # via équation (3) avec J = 1.
```

II.3. Recherche d'une configuration stable

Q23. On utilise la fonction `random` donnée en annexe.

```
def proba(p: float) -> bool:
    if random() < p:
        return True
    else:
        return False
```

ou de manière équivalente et plus condensée :

```
def proba_v2(p: float) -> bool:
    return random() < p
```

Q24. On suit l'énoncé en distinguant les cas suivant le signe du *spin*.

```
def test_boltzmann(delta_e: float, T: float) -> bool:
    if delta_e <= 0:
        return True
    else:
        p = exp(-delta_e / T)
        return proba(p)
```

Q25. La commande `L2 = L[:]` réalise une copie de la liste `L` et la stocke dans la variable `L2`.
En revanche la commande `L2 = L` ne fait que faire pointer la variable `L2` vers la valeur de `L`. Dans ce cas, toute modification de l'une modifie également l'autre, ce qui n'est pas le cas si on réalise une copie (les modifications ne sont faites que sur la copie, pas sur la liste originale).

Q26. D'une part, la fonction `calcul_delta_e1` nécessite de calculer l'énergie de la configuration de départ et de celle d'arrivée : on doit donc parcourir deux fois toute la grille, c'est de complexité $O(n)$.
D'autre part, la fonction `calcul_delta_e2` ne prend en compte que les voisins du *spin* modifié. En effet, tout le reste de la grille est inchangé donc il n'est pas nécessaire de tout recalculer. Cette seconde fonction est de complexité constante, elle est donc plus efficace.

Q27. On traduit l'énoncé : choix aléatoire du *spin* à modifier puis calcul de la variation d'énergie et enfin modification éventuelle de ce *spin* suivant le résultat du test de Boltzmann.

```
def monte_carlo(L: [int], T: float, n_tests: int) -> None:
    for k in range(n_tests):
        i_spin = randrange(n)   # rappel : len(L) = n
        delta_e = calcul_delta_e2(L, i_spin)
        if test_boltzmann(delta_e, T):
            L[i_spin] = -1 * L[i_spin]
```

Q28. À nouveau, on traduit l'énoncé pas à pas.

```
def aimantation_moyenne(n_tests: int, T: float) -> float:
    # Initialisation du matériau
    s = initialisation()
    L = deplier(s)
    # Évolution
    monte_carlo(L, T, n_tests)  # Action par effet de bord
    # Calcul aimantation moyenne
    valeurs_spins = 0
    for i in range(n):
        valeurs_spins = valeurs_spins + L[i]
    return valeurs_spins / n
```

Q29. L'augmentation de la température semble rendre la répartition des *spins* plus aléatoires, il n'y a plus de zones dans lesquelles les *spins* sont tous identiques. Physiquement cela signifie que l'aimantation du matériau tendra à disparaître avec l'augmentation de température (cela rejoint le début de l'énoncé à propos de la température de Curie).

Partie III : Exploration des domaines de Weiss

Q30. Encore une fois, on traduit pas à pas la démarche décrite dans l'énoncé.

```
def explorer_voisinage(L: [int], i: int, weiss: [int],
                                              num: int) -> None:
    for i_voisin in liste_voisins_dans_liste(L):
        if L[i_voisin] == L[i] and weiss[i_voisin] == -1:
            weiss[i_voison] = num
            explorer_voisinage(L, i_voisin, weiss, num)
```

Q31. Le nombre d'appels récursifs peut devenir très grand : pour chaque *spin*, on peut considérer jusqu'à 4 de ses voisins et ainsi de suite, ce qui requiert une grande utilisation de la mémoire afin de stocker ces appels.

Q32. On commence par explorer le *spin* `i` et on regarde ses voisins. Dans la pile, on ajoute ceux non déjà marqués et on réitère le processus.

```
def explorer_voisinage_pile(L: [int], i: int, weiss: [int],
                                num: int, pile: [int]) -> None:
    pile = [i]
    while len(pile) > 0:  # ou juste while pile:
        i_explo = pile.pop()
        weiss[i_explo] = num
        for i_voisin in liste_voisins_dans_liste(i_explo):
            if L[i_voisin]==L[i_explo] and weiss[i_voisin]==-1:
                pile.append(i_voisin)
```

Q33. On commence par le *spin* d'indice 0 et on marque tous ceux situés dans le même domaine (numéro 0). On cherche alors le prochain *spin* non marqué et on réitère le processus avec le numéro de domaine suivant. On répète cela jusqu'à avoir marqué tous les *spins*.

```
def construire_domaines_weiss(L: [int]) -> [int]:
    num_zone = 0
    i = 0
    weiss = [-1] * n
    while i < n:
        # Marquage de tous les spins du même domaine
        pile = []
        explorer_voisinage_pile(L, i, weiss, num_zone, pile)
        # Recherche du prochain spin non marqué
        while i < n and weiss[i] != -1:
            i = i + 1
        # Incrémentation du numéro de zone pour la prochaine
        num_zone = num_zone + 1
    return weiss
```