Devoir d'informatique n° 1

vendredi 4 octobre 2024

Le sujet comporte 12 pages et est composé de trois exercices indépendants.

L'exercice 3 sera rédigée sur une copie distincte de celles des deux premiers exercices. Sur celle commençant par l'exercice 1, vous laisserez la première page vide.

*Les calculatrices **ne** sont **pas** autorisées.*

Le seul langage de programmation autorisé dans cette épreuve est Python. Les candidats peuvent à tout moment supposer qu'une fonction définie dans une question précédente est disponible, même s'ils n'ont pas traité la question correspondante.

Une attention particulière sera portée à la lisibilité et la simplicité du code proposé. En particulier, l'utilisation d'identifiants significatifs, l'emploi judicieux de commentaires et la description du principe de chaque programme seront appréciés.

Une liste de fonctions utiles est fournie à la fin du sujet.

Exercice 1. Pour s'échauffer

Écrire une fonction `tous_differents` qui prend en argument une liste `valeurs` que l'on **supposera triée** par ordre croissant et qui renvoie un booléen indiquant si oui ou non les éléments de la liste `valeurs` sont tous distincts. On fera en sorte que la fonction soit de complexité linéaire en la longueur de `valeurs`.

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

Dans ce sujet, on s'intéresse à la recherche d'un motif dans une molécule d'ADN.

Une molécule d'ADN est constituée de deux brins complémentaires, qui sont un long enchaînement de nucléotides de quatre type différents désignés par les lettres A, T, C et G. Les deux brins sont complémentaires : « en face » d'un 'A', il y a toujours un 'T' et « en face » d'un 'C', il y a toujours un 'G'. Pour simplifier le sujet, on va considérer qu'une molécule d'ADN est une chaîne de caractères sur l'alphabet {A, C, G, T} (on s'intéresse donc seulement à un des deux brins). On parlera de séquence d'ADN.

Partie I - Génération d'une séquence d'ADN

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

Q1. Que renvoie la commande `seq[3]` ? Que renvoie la commande `seq[2:6]` ?

Les fonctions que nous allons construire par la suite devront prendre en paramètre une chaîne de caractères ne contenant que des `'A'`, `'C'`, `'G'` et `'T'` (ceci correspond à une séquence d'ADN).

Pour générer aléatoirement une séquence d'ADN composée de n caractères, on utilisera le principe suivant.

① On commencer par créer une chaîne de caractères vide.

② Puis on tire aléatoirement n chiffres compris entre 1 et 4 et :

- si on obtient un 1, alors on ajoute un `'A'` à notre chaîne de caractères ;
- si on obtient un 2, alors on ajoute un `'C'` à notre chaîne de caractères ;
- si on obtient un 3, alors on ajoute un `'G'` à notre chaîne de caractères ;
- si on obtient un 4, alors on ajoute un `'T'` à notre chaîne de caractères.

③ On renvoie la chaîne de caractères ainsi construite.

Q2. Écrire une fonction `generation()` qui prend en paramètres un entier n et qui renvoie une chaîne de caractères aléatoires de longueur n ne contenant que des `'A'`, `'C'`, `'G'` et `'T'`.
On pourra utiliser les fonctions `random` ou `randint` détaillées en annexe.

Q3. Que fait la fonction suivante qui prend en argument une chaîne de caractère `seq` ne contenant que des `'A'`, `'C'`, `'G'` et `'T'` ?

```
def mystere(seq: str):
    a, b, c, d = 0, 0, 0, 0
    i = len(seq) - 1
    while i >= 0:
        if seq[i] == 'A':
            a = a + 1
        elif seq[i] == 'C':
            b = b + 1
        elif seq[i] == 'G':
            c = c + 1
        else:
            d = d + 1
        i = i - 1
    return [a*100/len(seq), b*100/len(seq), c*100/len(seq),
                                            d*100/len(seq)]
```

Q4. Justifier la terminaison de la fonction `mystere`.

Q5. Quelle est la complexité de la fonction `mystere` ?

Partie II - Recherche d'un motif

Soit une chaîne de caractères `S = 'ACTGGTCACT'`, on appelle sous-chaîne de caractères de `S` une suite de caractères incluse dans `S`. Par exemple, `'TCG'` est une sous-chaîne de `S` mais `'TAG'` n'est pas une sous-chaîne de `S`.

L'objectif est de rechercher une sous-chaîne de caractères `M` de longueur m appelée motif dans une chaîne de caractères `S` de longueur n.

Il s'agit d'une problématique classique en informatique, qui répond aux besoins de nombreuses applications. On trouve plus de 100 algorithmes différents pour cette même tâche, les plus célèbres datant des années 1970, mais plus de la moitié ont moins de 10 ans.

On va s'intéresser ici à l'algorithme naïf et en voir ses limites.

Principe de l'algorithme naïf
On parcourt la chaîne. À chaque étape, on regarde si on a trouvé le bon motif. Si ce n'est pas le cas, on recommence avec l'élément suivant de la chaîne de caractères.

On admet que cet algorithme a une complexité en $O(nm)$ avec n la taille de la chaîne de caractères et m la taille du motif.

Q6. Écrire une fonction `recherche(M: str, S: str)` qui renvoie -1 si `M` n'est pas dans `S`, et la position de la première lettre de la chaîne de caractères `M` si `M` est présente dans `S`. Cet algorithme doit correspondre à l'algorithme naïf.

Q7. En supposant qu'une séquence d'ADN est composée de 3×10^9 caractères, combien faut-il d'opérations pour chercher un motif de 50 caractères dans une séquence d'ADN en utilisant l'algorithme naïf ?
En combien de temps un ordinateur réalisant 10^{12} opérations par seconde fait-il ce calcul ?

En génétique, on utilise des algorithmes de recherche pour identifier des similarités entre deux séquences d'ADN. Pour cela, on procède de la manière suivante :

- découper la première séquence d'ADN en morceaux de taille 50 ;
- rechercher chaque morceau dans la deuxième séquence d'ADN.

Q8. En utilisant les calculs précédents, combien de temps faut-il pour un ordinateur réalisant 10^{12} opérations par seconde pour comparer deux séquences d'ADN avec l'algorithme naïf ?
Vous semble-t-il intéressant d'utiliser l'algorithme de rechercher naïf ?

La suite du sujet original présentait l'algorithme de Knuth-Morris-Pratt, permettant de traiter ce problème avec une complexité $O(m+n)$. Pour les curieux, je mettrai en ligne le lien vers ce sujet.

Exercice 3. Modélisation numérique d'un matériau magnétique (*d'après CCMP 2022*)

Certains matériaux particuliers peuvent acquérir des états magnétiques qualifiés de paramagnétique et ferromagnétique. Le matériau est dit paramagnétique lorsqu'il ne possède pas d'aimantation spontanée, mais acquiert une aimantation sous l'effet d'un champ magnétique extérieur. Il est dit ferromagnétique lorsqu'il possède une aimantation même en l'absence de champ magnétique extérieur. Dans ces matériaux, la température T joue un rôle crucial : si T est supérieure à une température particulière T_C, nommée température de Curie, le matériau adopte un état paramagnétique. Dans le cas contraire ($T < T_C$), il adopte un état ferromagnétique. C'est par exemple le cas du fer, pour lequel la transition entre les deux états se produit à $T_C = 1\,043$ kelvin.

Dans un matériau magnétique, les divers éléments magnétiques (électrons, atomes) possédant un moment magnétique créent une aimantation moyenne à l'intérieur du matériau. Nous admettrons les principaux résultats de la théorie du paramagnétisme.

Cet exercice est constitué de 3 parties. Dans la première, on cherche à obtenir l'aimantation moyenne du matériau en fonction de la température à partir d'une formule théorique connue. Dans la seconde, on cherche à développer une modélisation microscopique d'un matériau magnétique à deux dimensions pour retrouver ce comportement (modèle d'Ising). Dans la troisième, on s'intéresse aux domaines magnétiques du matériau (nommés domaines de Weiss).

Une courte documentation de quelques fonctions utiles est disponible à la fin du sujet.

Partie I : Transition paramagnétique/ferromagnétique sans champ magnétique extérieur

La théorie des matériaux indique que, dans un matériau ferromagnétique, l'aimantation volumique moyenne du matériau est donnée par :

$$M = N\mu \tanh\left(\frac{\mu B}{k_B T}\right) \tag{1}$$

où N est le nombre d'atomes par unité de volume, B est la valeur du champ magnétique à l'intérieur du matériau, μ le moment magnétique des atomes ou des ions, k_B la constante de Boltzmann, T la température et $\tanh : x \mapsto \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$ la fonction tangente hyperbolique.

On considère une situation sans champ magnétique extérieur. Le champ magnétique local à l'intérieur du matériau ferromagnétique est donc celui crée par la matériau lui-même. On admet que ce champ magnétique est proportionnel à l'aimantation moyenne dans le matériau ($B = \lambda M$), et on obtient alors : $M = N\mu \tanh\left(\frac{\mu\lambda M}{k_B T}\right)$.

En introduisant l'aimantation réduite $m = \frac{M}{N\mu}$ et la température réduite $t = \frac{k_B T}{N\mu^2\lambda} = \frac{T}{T_C}$, l'équation (1) devient :

$$m = \tanh(m/t) \tag{2}$$

Cette équation d'inconnue m ne possède pas de solution analytique : si on veut connaître une approximation de l'aimantation moyenne dans le matériau, il est donc nécessaire de la résoudre numériquement par une méthode de recherche de zéro.

Q9. Écrire les instructions nécessaires pour importer exclusivement les fonctions exponentielle (`exp`) et tangente hyperbolique (`tanh`) du module `math`, ainsi que les fonctions `randrange` et `random` du module `random`. Ces fonctions seront ainsi utilisables dans tous les programmes que vous écrirez ultérieurement.

Q10. À partir de l'équation (2), indiquer une équation $f(x,t) = 0$, d'inconnue x que l'on doit résoudre et écrire en Python la définition de la fonction `f` correspondante (paramètres `x` et `t`).

Afin de résoudre numériquement cette équation, on propose la fonction suivante, reposant sur le principe de la dichotomie, qui calcule une valeur approchée à `eps` près du zéro d'une fonction `f(x, t)`, de variable `x` et de paramètre `t` fixé, sur un intervalle $[a\,;b]$.

On supposera pour simplifier que la fonction dont on recherche le zéro est continue, strictement monotone et s'annule une fois et une seule sur l'intervalle $[a\,;b]$.

```
1 def dicho(f, t, a, b, eps):
2     while b - a > eps:
3         c = (a + b) / 2
4         if f(c, t) * f(b, t) < 0:
5             a = c
6         else:
7             b = c
8     return (a + b) / 2
```

Q11. Expliquer brièvement les lignes 3 à 7 en illustrant la démarche sur le graphe d'une fonction vérifiant les hypothèses précédentes.

Q12. Établir l'expression de la complexité temporelle asymptotique de la fonction `dicho` en fonction de `a`, `b` et `eps`.

Une étude mathématique simple permet de prouver que l'équation $m = \text{th}(m/t)$ n'a de solution $m > 0$ que pour $0 < t < 1$, ce qui revient à dire que le matériau ne possède une aimantation non nulle que pour une température inférieure à la température de Curie. On admet ainsi que si $t \geqslant 1$ alors $m = 0$. De plus, pour $t < 1$, on admet que la solution $m = 0$ ne doit pas être prise en compte car elle correspond à une solution instable.

Q13. En utilisant la fonction `dicho`, écrire une fonction `construction_liste_m(t1, t2)` qui construit et retourne une liste de 500 solutions de l'équation (2), pour t variant linéairement de t_1 à t_2 (bornes incluses). On cherchera les valeurs de m à 10^{-6} près avec un intervalle de recherche initial $m \in [0{,}001\,;1]$.

En traçant l'aimantation m en fonction de la température t, on obtient le graphe de la figure 1 permettant de visualiser les domaines ferromagnétique ($t < 1$) et paramagnétique ($t \geqslant 1$).

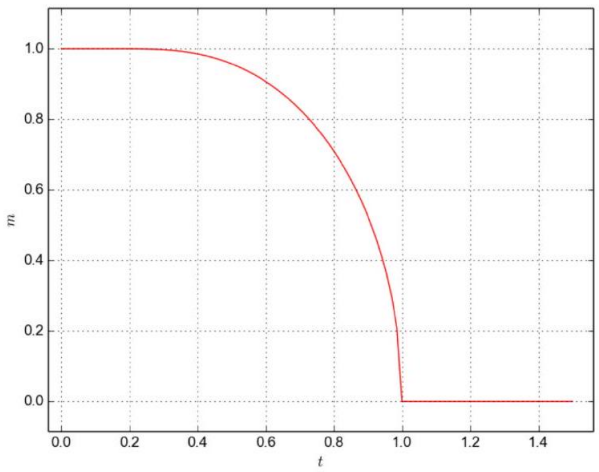

FIGURE 1 – Aimantation réduite m en fonction de la température réduite t

Partie II : Modèle microscopique d'un matériau magnétique

II.1. Premiers exemples

Pour étudier l'effet du champ magnétique sur un matériau magnétique, on adopte une modélisation microscopique. On modélise les atomes par des sites portant chacun une grandeur physique, nommée *spin*, dont il n'est pas nécessaire de connaître les propriétés.

L'échantillon modélisé est une zone carrée à deux dimensions possédant h *spins* régulièrement répartis dans chaque direction, donc formant une grille carrée de $n = h^2$ *spins*. Chaque *spin* ne possède que deux états down ou up, ce que l'on modélise par une variable $s_p \in \{-1, +1\}$.

Pour implémenter cette configuration de *spins* décrivant l'état microscopique du matériau, on choisit de travailler sur une liste `s`, contenant h listes de h entiers, chacun valant -1 ou 1. Dans la suite, une

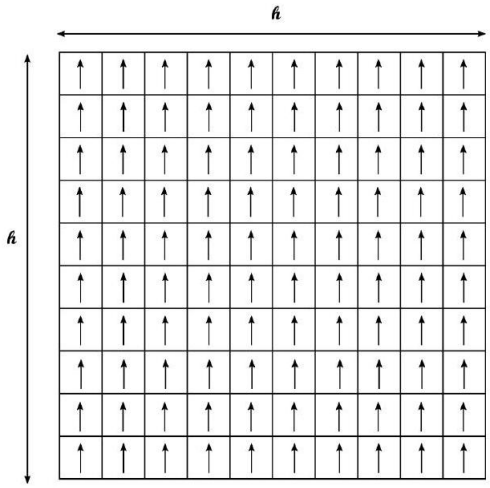

FIGURE 2 – Modèle des *spins* dans un matériau ferromagnétique

telle liste `s` sera appelée simplement « grille » (sous-entendu de taille h). On parlera de lignes et de colonnes en commençant l'indexation à 0. En particulier, la ligne du haut de la grille est la 0-ème ligne, et la dernière est la $(h-1)$-ème ligne (de même pour les colonnes).

Un domaine d'aimantation uniforme (cf. figure 2) sera donc représenté par une grille ne contenant que des 1.

Le début du programme, outre les imports de modules Python déjà réalisés à la question **Q9**, est défini par :

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

ce qui définit deux variables globales utilisables dans tout le programme.

Q14. Écrire une fonction `ligne(val: int)` qui renvoie une liste de h entiers tous égaux à `val`.

Q15. Écrire une fonction `initialisation()` renvoyant une grille contenant h^2 *spins* de valeur 1 comme sur la figure 2.

L'antiferromagnétisme est une propriété de certains milieux magnétiques. Contrairement aux matériaux ferromagnétiques, dans les matériaux antiferromagnétiques, l'interaction d'échange entre les atomes voisins conduit à un alignement antiparallèle des moments magnétiques atomiques (cf. figure 3). L'aimantation totale du matériau est alors nulle (on se limite au cas où h est pair).

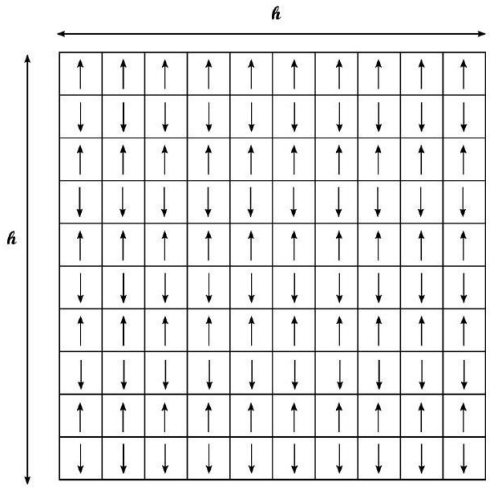

FIGURE 3 – Modèle des *spins* dans un matériau antiferromagnétique

Q16. Écrire une fonction `initialisation_anti()` renvoyant une grille alternant les 1 et les -1 comme sur la figure 3.

II.2. Énergie d'une configuration

Dans la modélisation adoptée (sans champ magnétique extérieur), l'énergie d'une configuration, définie par l'ensemble des valeurs de tous les *spins*, est donnée par :

$$E = -\frac{J}{2} \sum_{p} \sum_{v \in V_p} s_p s_v \tag{3}$$

avec V_p l'ensemble des voisins du *spin* situé à la position p.

On suppose que seuls les quatre *spins* situés juste au dessus, en dessous, à gauche et à droite de s_p sont capables d'interagir avec lui.

J est nommée intégrale d'échange et modélise l'interaction entre deux *spins* voisins. Pour simplifier, on considérera dans les programmes que $J = 1$.

Malgré le caractère fini de l'échantillon, on peut utiliser une modélisation très utile pour faire comme s'il était infini en utilisant les conditions aux limites périodiques. Lorsque l'on considère un *spin* dans la colonne située la plus à droite (resp. gauche), il ne possède pas de plus proche voisin à droite (resp. gauche) : on convient de lui en affecter un, qui sera situé sur la même ligne complètement à gauche (resp. droite) de l'échantillon. De même, le plus proche voisin manquant d'un *spin* situé sur la première (resp. dernière) ligne sera situé sur la dernière (resp. première) ligne de l'échantillon (voir la figure 4).

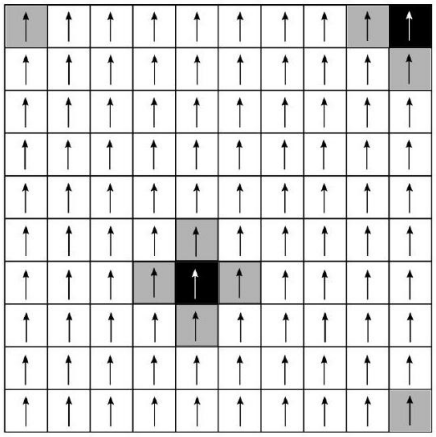

FIGURE 4 – Voisinage d'un *spin* (les voisins des *spins* sur les cases noires sont indiqués en gris)

Q17. Définir une fonction `liste_voisins(p: (int, int))` qui renvoie la liste des positions des plus proches voisins du *spin* s_p (dans l'ordre gauche, droite, dessus, dessous). Plus précisément, si $p = (i, j)$, la fonction doit renvoyer une liste de couples correspondants aux positions des voisins du *spin* situé à la i-ème ligne et j-ème colonne. On pourra utilement utiliser l'opération % de Python, qui renvoie le reste de la division euclidienne.

Afin de faciliter le calcul de l'énergie donnée par l'équation (3), il va être plus aisé de travailler avec une simple liste `L` qu'avec une grille `s`. On va donc transformer cette grille de taille $h \times h$ en une liste de longueur $n = h^2$.

Q18. Écrire une fonction `deplier(grille: [[int]])` qui renvoie une simple liste de $n = h^2$ entiers correspondants aux *spins* de la grille dans l'ordre suivant : première ligne puis deuxième ligne, etc.

Q19. Écrire une fonction `replier(L: [int])` qui prend en argument une liste de *spins* de longueur $n = h^2$ et qui renvoie une grille de taille $h \times h$ où les h premiers éléments de `L` forme la 0-ème ligne de la grille, les h suivants la 1-ère ligne, etc. On commencera par une assertion véridiant que la liste `L` est bien de longueur n.

Q20. À quelle position de la liste `L` définie par `L = deplier(s)` se trouve le *spin* correspondant à la case noire située à la position $(6, 4)$ de la grille `s` donnée par la figure 4 ?

Q21. Écrire une fonction `conversion_coord(p: (int, int))` qui renvoie l'indice correspondant dans la liste `L` au *spin* présent à la position `p` dans une grille `s` où `L = deplier(s)`.

Grâce à ces différentes fonctions, on suppose désormais, et jusqu'à la fin du sujet, que l'on dispose d'une fonction `liste_voisins_dans_liste(i: int)` qui renvoie la liste des indices des voisins du *spin* d'indice `i` dans la liste `L` où `L = deplier(s)`.

Q22. Définir la fonction `energie(s: [[int]])` qui calcule l'énergie d'une configuration `s` donnée (cf. équation (3)). On commencera par définir la liste `L` via `L = deplier(s)`.

II.3. Recherche d'une configuration stable

Jusqu'à la fin du sujet, on reprend la représentation des *spins* du matériau à l'aide d'une simple liste de $n = h^2$ entiers valant -1 ou 1 comme défini dans la partie précédente.

On rappelle que l'on dispose d'une fonction `liste_voisins_dans_liste(i)` qui renvoie les indices des voisins du *spin* d'indice `i` dans la liste `L` représentant la grille `s` où `L = deplier(s)`. On suppose également que la fonction `energie` définie en **Q22** a été modifiée et peut désormais indifféremment prendre pour argument soit une grille soit une liste représentant le matériau. Dans les deux cas, elle renvoie l'énergie de la configuration donnée en argument.

Pour trouver une configuration stable pour les *spins*, il faut faire évoluer la liste vers une situation d'équilibre conformément aux principes de la physique statistique. On adopte une méthode probabiliste connue sous le nom de méthode de Monte-Carlo, dont le principe de fonctionnement est le suivant. À chaque étape :

- on choisit un *spin* au hasard dans l'échantillon,
- on calcule la variation d'énergie ΔE qui résulterait d'un changement d'orientation de ce *spin*,
- si $\Delta E \leqslant 0$, ce *spin* change de signe,
- si $\Delta E > 0$, ce *spin* change de signe avec la probabilité donnée par la loi de Boltzmann :

$$p = \exp\left(\frac{-\Delta E}{k_B T}\right).$$

Dans la suite, ΔE et $k_B T$ seront désignées par les variables `delta_e` et `T` correspondantes dans le programme.

Q23. Écrire une fonction `proba(p: float)` qui renvoie `True` avec la probabilité `p` et `False` avec la probabilité `1-p`.

Q24. Définir une fonction `test_boltzmann(delta_e: float, T: float)` qui renvoie `True` si le *spin* change de signe, et `False` sinon, en suivant les règles définies ci-avant.

Juste après avoir sélectionné au hasard l'indice i d'un *spin* de la liste `L` à basculer éventuellement, pour évaluer l'écart d'énergie `delta_e` entre les deux configurations avant/après, on propose deux solutions sous forme des fonctions `calcul_delta_e1` et `calcul_delta_e2` :

```
1  def calcul_delta_e1(L: [int], i: int):
2      L2 = L[:]
3      L2[i] = -L[i]
4      delta_e = energie(L2) - energie(L)
5      return delta_e
6
7  def calcul_delta_e2(L: [int], i: int):
8      delta_e = 0
9      for j in liste_voisins_dans_liste(i):
10         delta_e = delta_e + 2*L[i]*L[j]
11     return delta_e
```

où `L[i]` est le *spin* choisi pour être éventuellement retourné.

Q25. Que fait la commande `L2 = L[:]` ? Quelle est la différence avec la commande `L2 = L` ?

Q26. Indiquer la solution qui vous paraît la plus efficace pour minimiser le temps de calcul en justifiant votre réponse.

Q27. En utilisant la fonction `test_boltzmann`, définir une fonction
`monte_carlo(L: [int], T: float, n_tests: int)`
qui applique la méthode de Monte-Carlo et qui modifie la liste `L` où l'on a choisi successivement `n_tests` *spins* au hasard, que l'on modifie éventuellement suivant les règles indiquées dans les explications.

Q28. Écrire la fonction `aimantation_moyenne(n_tests: int, T: float)` qui :

- initialise une liste des *spins* (avec la fonction `initialisation` par exemple),
- la fait évoluer en effectuant `n_tests` tests de Boltzmann et les inversions éventuelles qui en découlent,
- calcule et renvoie l'aimantation moyenne de la configuration à la température `T` (définie ici comme la somme des valeurs des *spins* divisée par le nombre total de *spins*).

On réalise plusieurs simulations en faisant varier la température T autour de la température de Curie (ici $T_C = 2\,269$ kelvin compte-tenu du choix des valeurs de J et k_B). On représente alors les *spins* orientés vers le haut par une case foncée et les *spins* orientés vers le bas par une case claire. On obtient les résultats de la figure 5.

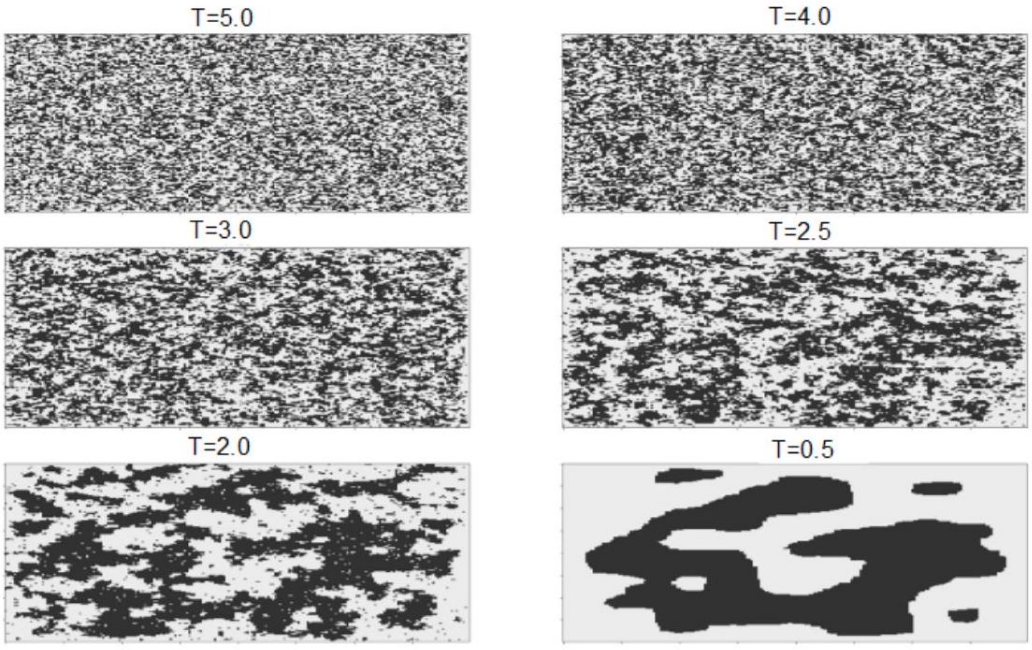

FIGURE 5 – Évolution du domaine en fonction de la température T

Q29. Indiquer l'influence de l'augmentation de la température sur le comportement du matériau ferromagnétique.

Partie III : Exploration des domaines de Weiss

Une observation microscopique des matériaux magnétiques nous apprend que les zones magnétiques du matériau sont organisées en domaines, nommés domaines de Weiss. Dans un domaine de Weiss donné, tous les *spins* ont la même valeur.

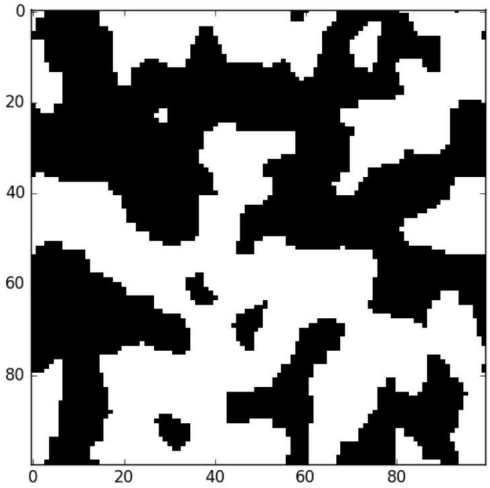

FIGURE 6 – Représentation des domaines de Weiss

On souhaite dans la suite décrire les différents domaines de Weiss afin, par exemple, de les colorier d'une manière différente. Pour cela, on va construire une liste, nommée `weiss`, possédant exactement la même taille que `L` (soit n) contenant initialement des -1 (cette valeur signifiant que le *spin* correspondant n'a encore été affecté à aucun domaine de Weiss).

On souhaite alors écrire une fonction récursive
`explorer_voisinage(L: [int], i: int, weiss: [int], num: int)`
qui, à partir d'une configuration donnée `L`, d'un indice de départ `i` (repérant le *spin* dans `L`), ainsi qu'un entier `num` qui précise le numéro du domaine auquel appartient L_i, construit récursivement la liste `weiss` par effet de bord. Cette fonction doit réaliser les opérations suivantes :

— Pour chaque *spin* voisin du *spin* L_i, elle doit vérifier si les *spins* sont identiques, et s'il n'a pas déjà été affecté à un domaine de Weiss précédemment.

— Dès qu'un tel *spin* est ajouté, on inscrit son numéro de domaine dans la liste `weiss`, et on explore récursivement son voisinage.

Q30. Écrire le code de la fonction récursive
`explorer_voisinage(L: [int], i: int, weiss: [int], num: int)`
conforme à la description ci-dessus.

Q31. Quel est l'inconvénient de cette méthode ?

Afin de mieux contrôler ce parcours, on choisit de l'effectuer avec une structure de pile explicite. Cette pile sera représentée par une liste nommée `pile` sur laquelle on peut ajouter un élément (avec `append`) ou récupérer l'élément du dessus (via `pile.pop()` qui renvoie l'élément sur le dessus de la pile et le retire de la pile). Il s'agit donc, tant qu'il reste des *spins* à explorer, de :

- récupérer l'indice d'un *spin* à explorer dans la pile et le marquer dans la liste `weiss`,
- regarder dans son voisinage si des *spins* possèdent la même valeur et n'ont pas encore été affectés à un domaine, puis ajouter leurs indices dans la pile si c'est le cas.

Q32. Écrire le code de la fonction itérative
`explorer_voisinage_pile(L: [int], i: int, weiss: [int], num: int, pile: [int])`
conforme à la description ci-dessus.

Enfin, la fonction précédente va permettre de construire la liste `weiss` contenant les numéros des domaines auxquels appartiennent tous les *spins* (le numéro du domaine auquel appartient le *spin* d'indice 0 sera pris à 0, le domaine suivant à 1, etc.).

Q33. Écrire le code de la fonction `construire_domaines_weiss(L)` qui construit et renvoie une liste contenant le numéro des domaines de Weiss de chaque *spin* du domaine. De cette façon, on pourra créer la liste voulue `weiss` grâce à l'appel

```
weiss = construire_domaines_weiss(L)
```

L'intérêt de la construction précédente est de disposer d'un marqueur différent pour chaque domaine de Weiss, permettant par exemple de les visualiser dans deux dimensions avec des couleurs différentes, comme dans l'image de la figure 7 :

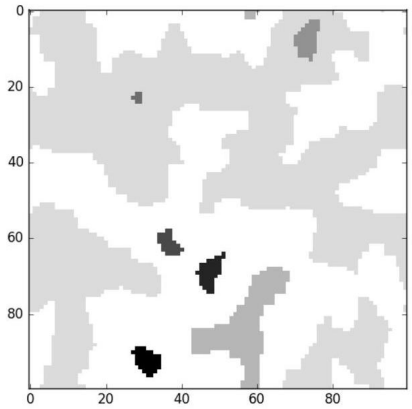

FIGURE 7 – Domaine de Weiss après marquage en nuances de gris

Annexe : Quelques fonctions utiles

Les fonctions présentées ci-dessous pourront être utilisées sous réserve de l'import du module Python auquel elles appartiennent.

Opérateurs mathématiques

- Dans le module `math`, on dispose des fonctions `exp` et `tanh` permettant de calculer l'exponentielle et la tangente hyperbolique d'un entier ou d'un flottant.
- L'opération `%` renvoie le reste de la division euclidienne. En particulier l'opération `n % p` (avec `n` et `p` entiers positifs) renvoie un entier compris entre 0 et `p-1` (les deux inclus). Par exemple `14 % 3` renvoie `2` et `12 % 3` renvoie `0`.

Aléatoire

Dans le module `random` :

- `randrange(n)` permet de renvoyer un entier aléatoirement choisi parmi 0, 1, 2, ..., $n-1$.
- `random()` permet de renvoyer un flottant aléatoire entre 0 et 1 (probabilité uniforme).
- `randint(a, b)` renvoie un entier aléatoirement choisi entre `a` (inclus) et `b` (exclu).

On admet que ces trois fonctions sont de complexité constante.