Corrigé proposé par Thibault PHILIPPE

Partie I

1.

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

2.

On pose $f(x,t) = x - tanh(x/t)$.

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

3.

Erreur *: On lira $f(x,t)$ au lieu de $f(m,t)$.*

Imprécision *: On supposera aussi que f change de signe sur son unique valeur d'annulation, sans quoi la dichotomie ne fonctionnera pas.*

```
def dicho(f, t, a, b, eps):
    while b - a > eps:
        m = (a + b) / 2
        if f(m, t) * f(b, t) < 0:
            a = m
        else:
            b = m
    return (a + b) / 2
```

4.

Posons p, le nombre de tours de boucle `while` nécessaires, avec à chaque tour une division par 2 de la largeur de l'intervalle. L'algorithme se termine lorsque l'intervalle atteint une largeur de `eps`, ainsi $(b-a)/2^p = eps$. On en déduit

$$p = \log_2\left(\frac{b-a}{eps}\right)$$

L'algorithme est donc en $O\left(log\left(\frac{b-a}{eps}\right)\right)$.

5.

```
def construction_liste_m(t1, t2):
    solutions = []
    pas = (t2 - t1) / 499  # Pour avoir les 2 bornes incluses
    for i in range(500):
        t = t1 + pas*i
        if t >= 1:
            m = 0
        else:
            m = dicho(f, t, 0.001, 1, 1e-6)
        solutions.append(m)
    return solutions
```

Partie II

6.

```
SELECT nom FROM materiaux WHERE t_curie > 500
```

7.

```
SELECT nom_fournisseur, prix_kg * 4.5 FROM fournisseurs
  JOIN prix ON id_four = id_fournisseur
  WHERE id_mat = 8713
```

8.

```
SELECT nom_fournisseur, prix_kg * 4.5 FROM fournisseurs
  JOIN prix ON id_four = id_fournisseur
  WHERE id_mat = 8713
    AND prix_kg =
      (SELECT MIN(prix_kg) FROM prix WHERE id_mat = 8713)
```

9.

```
SELECT nom, AVG(prix_kg) AS moyenne FROM materiaux
  JOIN prix ON id_mat = id_materiau
  GROUP BY id_mat HAVING moyenne < 50
```

Partie III

10.

```
def initialisation():
    return [1] * n
```

11.

```
def initialisation_anti():
    L = []
    for i in range(h):
        if i%2 == 0:
            L.extend([1] * h)
        else:
            L.extend([-1] * h)
    return L
```

12.

```
def repliement(s):
    L = []
    for i in range(h):
        L.append(s[h*i:h*(i+1)])
    return L
```

13.

Remarque : question qui me semble fastidieuse, mais peut-être ai-je raté un moyen plus simple...

```
def liste_voisins(i):
    v = []
    if i%h == 0:  # Voisin de gauche
        v.append(i + h-1)
    else:
        v.append(i-1)
    if i%h == h-1: # Voisin de droite
        v.append(i - (h-1))
    else:
        v.append(i+1)
    if i//h == h-1: # Voisin du dessous
        v.append(i%h)
    else:
        v.append(i + h)
    if i//h == 0: # Voisin du dessus
        v.append((h-1)*h + i%h)
    else:
```

```
            v.append(i - h)
        return v
```

14.

```
def energie(s):
    E = 0
    for i in range(n):
        E_i = 0
        L_voisins = liste_voisins(i)
        for iv in L_voisins:
            E_i += s[i] * s[iv]
        E += E_i
    return - 1/2 * E # En prenant J = 1 comme le demande l'énoncé
```

15.

```
def test_boltzmann(delta_e, T):
    if delta_e <= 0:
        return True
    p = exp(-delta_e / T)
    if random() < p:
        return True
    return False
```

16.

La solution 2, `calcul_delta_2e` est clairement la plus rapide, puisqu'elle part du principe que l'énergie étant la même presque partout, la variation à calculer ne nécessite que le calcul de l'énergie autour du spin à inverser. Elle se fait en temps constant, contrairement à la première qui se fait en temps linéaire.

17.

Remarque *: ici la fonction est une procédure, sans valeur de retour, qui modifie* `s` *par effet de bord.*

```
def monte_carlo(s, T, n_tests):
    for i_test in range(n_tests):
        i = randrange(0,n)
        if test_boltzmann(calcul_delta_e2(s, i), T):
            s[i] *= -1
```

18.

```
def aimantation_moyenne(n_tests, T):
    s = initialisation()
    monte_carlo(s, T, n_tests)

    # Calcul de l'aimantation moyenne
    m = 0
    for i in range(n):
        m += s[i]
    return m/n
```

19.

La fonction initialisation se fait en temps linéaire de n. `monte_carlo` est en $O(n_tests)$ grâce à `calcul_delta_e2` qui est en temps constant. Enfin, le calcul de l'aimantation moyenne est en temps linéaire de n.

Ainsi, la fonction `aimantation_moyenne` est en $O(n_tests + n)$ ou $O(n_tests + h^2)$.

20.

Dans le cas où l'on prendrait en compte toutes les interactions, le calcul de ΔE aurait nécessité $O(n)$ calculs. Cela aurait donc changé la complexité symptotique en $O(n_tests \times n)$.

21.

L'augmentation de la température a tendance à rendre aléatoire la répartition des domaines +1 et -1, et donc à supprimer l'aimantation du matériaux.

Partie IV

22.

```python
def explorer_voisinage(s, i, weiss, num):
    for iv in liste_voisins(i):
        if weiss[iv] == -1 and s[iv] == s[i]:
            weiss[iv] = num
            explorer_voisinage(s, iv, weiss, num)
```

23.

```python
def explorer_voisinage_pile(s, i, weiss, num, pile):
    pile.append(i)
    while len(pile) > 0:
        iv = pile.pop()
        weiss[iv] = num
        for ivv in liste_voisins(iv):
            if weiss[ivv] == -1 and s[ivv] == s[i]:
                pile.append(ivv)
```

24.

```python
def construire_domaines_weiss(s):
    num, i = 0, 0
    weiss = [-1] * n
    while i < n:
        # On marque tous les spins du domaine
        pile = []
        explorer_voisinage_pile(s, i, weiss, num, pile)
        # On cherche un nouveau point de départ
        i += 1
        while i < n and weiss[i] != -1:
            i += 1
        num += 1
    return weiss
```