Corrigé Épreuve informatique commune – Mines-Ponts 2022

Pour approfondir, regardez (au moins le début de) la conférence de Marie Théret sur la percolation. Elle commence justement par l'exemple des matériaux ferromagnétiques et une séquence filmée est particulièrement marquante. (minute 22 de https ://www.carmin.tv/c/309)

1.

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

2.

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

3.

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

4. À l'étape 0, la précision est $\frac{b-a}{2}$. À l'étape 1, la précision est $\frac{b-a}{2^2}$. À l'étape n, la précision est $\frac{b-a}{2^{n+1}}$.
Pour un arrêt en n étapes, on a

$$\frac{b-a}{2^n} \geqslant \varepsilon \quad \text{et} \quad \frac{b-a}{2^{n-1}} < \varepsilon$$

donc $\frac{\ln((b-a)/\varepsilon)}{\ln 2} < n+1$ et $\frac{\ln((b-a)/\varepsilon)}{\ln 2} \geqslant n$.
La complexité temporelle asymtotique de `dicho` est $O(\ln(\frac{b-a}{\varepsilon}))$.

5.

```
def construction_liste_m(t1, t2):
    liste_t = [t1 + k*(t2-t1)/499 for k in range(500)]
    liste_m = []
    for t in liste_t:
        x = dicho(f, t, 0.001, 1, 10**(-6))
        liste_m.append(x)
    return liste_m
```

6.

```
select nom from materiaux where t_curie < 500
```

7.

```
select nom_fournisseur, 4.5*prix_kg
from prix join fournisseurs on id_four = id_fournisseur
where id_mat = 8713
```

8.

```
select nom_fournisseur, 4.5*prix_kg
from prix join fournisseurs 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 prix join materiau on id_mat = id_materiau
group by id_materiau having moyenne < 50
```

10. Une réponse :

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

Une autre réponse :

```
def initialisation():
    return [1 for k in range(n)]
```

Une autre réponse :

```
def initialisation():
    L = []
    for k in range(n):
        L.append(1)
    return L
```

11. Notons qu'on pouvait ne considérer que le cas h pair (l'énoncé le précise).
Une réponse :

```python
def initialisation_anti():
    L = []
    s = 1
    for i in range(h):
        for j in range(h):
            L.append(s)
        s = -s
    return L
```

Une autre réponse :

```python
def initialisation_anti():
    reponse = ([1]*h + [-1]*h)*(h//2)
    if h%2 == 1:
        reponse = reponse + [1]*h
    return reponse
```

12. Une réponse :

```python
def repliement(s):
    spins = []
    for i in range(h):
        ligne = []
        for j in range(h):
            ligne.append(s[i*h+j])
        spins.append(ligne)
    return spins
```

Avec des tranches :

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

13.

```python
def liste_voisins(i):
    L = []
    # gauche
    if i%h == 0:
        # le voisin de gauche est tout à droite
        L.append(i+h-1)
    else:
        L.append(i-1)
    # droite
    if i%h == h-1:
        # le voisin de droite est tout à gauche
        L.append(i - (h-1))
    else:
        L.append(i+1)
    # dessous
    if i >= (h-1)*h :
        # le voisin du dessous est tout en haut
        L.append(i%h)
    else:
        L.append(i+h)
    # dessus
    if i < h:
        # le voisin du dessus est tout en bas
        L.append((h-1)*h + i)
    else:
        L.append(i-h)
    return L
```

14.

```python
def energie(s):
    E = 0
    for i in range(len(s)):
        voisins = liste_voisins(i)
        for j in voisins:
            E += s[i]*s[j]
    return -E/2
```

15.

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

16. La première fonction a un fonctionnement plus facile à comprendre et je vous laisse voir. La complexité est linéaire en la taille de la liste $O(n)$. La deuxième fonction a une complexité en $O(1)$ (« temps constant »). Le calcul sur lequel elle

repose est le suivant :

$$\begin{aligned} E_{\text{avant}} &= -\frac{1}{2}\sum_{k}\sum_{j\in V_k} s_k s_j \\ &= -\frac{1}{2}\sum_{(k,j)/j\in V_k \text{ et } k\neq i,\, j\neq i} s_k s_j - \frac{1}{2}\sum_{(k,j)/j\in V_k \text{ et } k \text{ ou } j=i} s_k s_j \end{aligned}$$

le spin i est changé en son opposé

$$\begin{aligned} E_{\text{après}} &= -\frac{1}{2}\sum_{(k,j)/j\in V_k \text{ et } k\neq i,\, j\neq i} s_k s_j - \frac{1}{2}\sum_{(k,j)/j\in V_k \text{ et } k \text{ ou } j=i} -s_k s_j \\ \Delta E &= \sum_{(k,j)/j\in V_k \text{ et } k \text{ ou } j=i} s_k s_j \end{aligned}$$

Ainsi

$$\begin{aligned} \Delta E &= \sum_{(i,j)/j\in V_i} s_i s_j + \sum_{(k,i)/i\in V_k} s_k s_i \\ &= 2\sum_{\ell}\sum_{r\in V_\ell} s_r s_\ell \end{aligned}$$

et on reconnaît le calcul donné dans `calcul_delta_e2(s, i)`.

17.

```
def monte_carlo(s, T, n_tests):
    for k in range(n_tests):
        numero = randrange(n) # un spin au hasard
        delta_e = calcul_delta_e2(s, numero)
        if test_boltzmann(delta_e, T):
            s[numero] = - s[numero]
                                # passe de 1 à -1 et réciproquement
    # (modification de s par effet de bord)
```

18.

```
def aimantation_moyenne(n_tests, T):
    s = initialisation()
    monte_carlo(s, T, n_tests)
    somme = 0
    for valeur in s:
        somme += valeur
    return somme/len(s)
```

19. La fonction `initialisation` se fait en temps linéaire en n. `monte_carlo` est en $O(ntests)$ et le calcul de l'aimantation moyenne est en $O(n)$.
Ainsi, la fonction `aimantation_moyenne` est en $O(n + ntests)$.

20. Si on avait pris en compte toutes les interactions, le calcul de ΔE aurait nécessité $O(n)$ calculs. Cela aurait donc changé la complexité en $O(n \times ntests)$.

21. Plus la température augmente, plus la répartition des 1 et -1 semble aléatoire, et donc moins le matériau est ferromagnétique.

22.

```
def explorer_voisinage(s, i, weiss, num):
    '''
␣␣␣␣s␣:␣configuration␣(n-liste)
␣␣␣␣i␣:␣indice␣de␣départ␣repérant␣le␣spin␣dans␣s
␣␣␣␣num␣:␣numéro␣du␣domaine␣auquel␣appartient␣s[i]
␣␣␣␣but:␣obtenir␣le␣domaine␣de␣Weiss␣du␣spin␣s[i]
␣␣␣␣'''
    voisins = liste_voisins(i)
    for j in voisins:
        if s[j] == s[i] and weiss[j] == -1:
                            # spins identiques et pas encore affecté
            weiss[j] = num
                            # on lui affecte le meme numéro que le spin s[i]
            explorer_voisinage(s, j, weiss, num)
```

23.

```
def explorer_voisinage_pile(s, i, weiss, num, pile):
    while pile != []:
        indice = pile.pop()
        weiss[indice] = num
        voisins = liste_voisins(indice)
        for j in voisins:
            if s[j] == s[indice] and weiss[j] == -1:
        # spins identiques et pas encore affecté
                pile.append(j)
```

24.

```
def construire_domaines_weiss(s):
    weiss = [-1] * n
    num = 0
    indice_courant = 0
    while -1 in weiss:
        pile = [indice_courant]
        explorer_voisinage_pile(s, indice_courant, weiss, num, pile)
        indice_suivant = indice_courant + 1
        while indice_suivant < len(s)
                                and weiss[indice_suivant] != -1:
            # Recherche de l'indice du spin du prochain domaine
            indice_suivant += 1
        indice_courant = indice_suivant
        num += 1
    return weiss
```