Corrigé de Marchons, marchons, marchons...

Remarques préliminaires

Les commentaires en italique donnent les intentions sous tendues par les questions posées. Ils sont purement informatifs, ne font pas partie des réponses et peuvent être ignorés.

Pour les requêtes SQL et les codes Python, il existe quantité de variations possibles qui sont fonctionnellement équivalentes, et donc acceptables, sous réserve que les méthodes employées n'induisent pas d'inefficacités algorithmiques majeures, qui elles devront être sanctionnées.

Les requêtes SQL fonctionnent avec SQLite et Postgres, sauf indication contraire.

Les codes Python sont demandés sous forme de fonctions, approche qu'il nous semble souhaitable d'encourager systématiquement. Plusieurs styles de corrections sont proposés quand des versions assez différentes peuvent être envisagées par les candidats.

Certaines questions impliquent des aspects pratiques triviaux : ajout des `import`, manipulations de fichiers, conversions d'unités, assertions pour vérifier des hypothèses... C'est volontaire et assumé : un candidat performant pour écrire des algorithmes mais moins pour produire des programmes qui marchent sera ainsi pénalisé.

Pour une même partie, il n'y a pas lieu de pénaliser un `import` oublié dans une question si il apparaît déjà dans une question précédente, les fonctions étant sensées être déclarées les unes à la suite des autres dans un même fichier.

Les codes Python fournis contiennent des lignes avec un commentaire `# IGNORER` qui ne font pas partie de la solution proposée mais qui sont utiles pour que le fichier montré soit réellement testable.

Le metteur au point présente par avance ses excuses en cas d'erreur ou d'approximation dans cette correction, assemblage de propositions du concepteur et des testeurs, dont il est in fine entièrement responsable.

Partie I. Randonnée

Requêtes SQL, importation et manipulation de données en Python.

❏ **Q1** – Compter le nombre de participants nés entre 1999 et 2003 inclus.

Une requête peu discriminante, une question offerte à tous, il faut bien commencer...

```sql
-- requête typique
SELECT COUNT(*) FROM Participant WHERE ne >= 1999 AND ne <= 2003;

-- opérateur between
SELECT COUNT(*) FROM Participant WHERE ne BETWEEN 1999 AND 2003;

-- et une version sans conjonction
SELECT COUNT(*) FROM Participant WHERE ABS(ne-2001) <= 2;
```

❏ **Q2** – Calculer la durée moyenne des randonnées pour chaque niveau de difficulté.

Idem, sauf si on s'interdit les regroupements avec un programme pas très cohérent...

```sql
-- version normale avec GROUP BY
SELECT diff, AVG(duree) FROM Rando GROUP BY 1;
```

```sql
-- autre version normale
SELECT diff, AVG(duree) FROM Rando GROUP BY diff;

-- version pédestre sans GROUP BY, si considéré comme hors programme
-- si une difficulté n'est pas utilisée, la moyenne sort NULL, pourquoi pas.
SELECT 1 AS diff, AVG(duree) FROM Rando WHERE diff = 1
UNION
SELECT 2, AVG(duree) FROM Rando WHERE diff = 2
UNION
SELECT 3, AVG(duree) FROM Rando WHERE diff = 3
UNION
SELECT 4, AVG(duree) FROM Rando WHERE diff = 4
UNION
SELECT 5, AVG(duree) FROM Rando WHERE diff = 5;
```

❏ **Q3** – Extraire le nom des participants pour lesquels la randonnée n°42 est trop difficile.

Jointure avec une condition un peu fine… ou produit cartésien, ou encore sous requête pas très jolie car hors d'un cadre relationnel strict.

```sql
-- version relationnelle au programme
SELECT pnom
FROM Participant
CROSS JOIN Rando
WHERE rid = 42 AND diff_max < diff;

-- version relationnelle avec jointure <
SELECT pnom
FROM Participant
JOIN Rando ON diff_max < diff
WHERE rid = 42;

-- version avec une sous-requête
SELECT pnom FROM Participant
WHERE diff_max < (SELECT diff FROM Rando WHERE rid = 42);
```

❏ **Q4** – Extraire (une seule fois chacune) les clés primaires des randonnées qui ont une ou des randonnées homonymes (nom identique et clé primaire distincte).

Auto-jointure, c'est plus subtile ; éviter les doublons est tordu si on veut rester dans le programme, mais facile avec DISTINCT…

```sql
-- version auto équi-jointure & distinct
SELECT DISTINCT r1.rid
FROM Rando AS r1
JOIN Rando AS r2 ON r1.rnom = r2.rnom
WHERE r1.rid <> r2.rid;

-- version auto jointure & distinct
SELECT DISTINCT r1.rid
FROM Rando AS r1
```

```sql
JOIN Rando AS r2 ON r1.rnom = r2.rnom AND r1.rid <> r2.rid;

-- version auto produit cartésien et distinct
-- on joue sur les mots, il y a logiquement une condition entre les deux côtés,
-- c'est donc bien une jointure
SELECT DISTINCT r1.rid
FROM Rando AS r1
CROSS JOIN Rando AS r2
WHERE r1.rnom = r2.rnom AND r1.rid <> r2.rid;

-- version pédestre auto jointure orientée
-- on profite du fait que UNION élimine les doublons...
SELECT r1.rid
FROM Rando AS r1
JOIN Rando AS r2 ON r1.rnom = r2.rnom
WHERE r1.rid < r2.rid
UNION
SELECT r1.rid
FROM Rando AS r1
JOIN Rando AS r2 ON r1.rnom = r2.rnom
WHERE r1.rid > r2.rid;

-- autre variante avec la jointure plus complète
SELECT r1.rid
FROM Rando AS r1
JOIN Rando AS r2 ON r1.rnom = r2.rnom AND r1.rid < r2.rid
UNION
SELECT r1.rid
FROM Rando AS r1
JOIN Rando AS r2 ON r1.rnom = r2.rnom AND r1.rid > r2.rid;

-- version sous-requête group by & having (bof)
SELECT rid FROM Rando
WHERE rnom IN (SELECT rnom FROM Rando GROUP BY 1 HAVING COUNT(*) > 1);

-- version accumulative avec Postgres
SELECT ARRAY_AGG(rid) AS rids
FROM Rando
GROUP BY rnom
HAVING ARRAY_LENGTH(ARRAY_AGG(rid), 1) > 1;
```

❏ **Q5** – Implémenter la fonction `importe_rando`.

Compétences pratiques indispensables : Ouvrir un fichier, s'adapter à son format (sauter la première ligne, découper, convertir), le fermer proprement… Pénaliser un peu si le fichier n'est pas explicitement ou implicitement fermé ?

```python
# version typique attendue d'un candidat
def importe_rando_1(nom_fichier):
    coords = []
    fichier = open(nom_fichier, "r")
    fichier.readline()
```

```
    for ligne in fichier.readlines():
        coords.append( [ float(i) for i in ligne.split(",") ] )
    fichier.close()
    return coords

# version plus pythonesque
def importe_rando_2(nom_fichier):
    with open(nom_fichier, "r") as fichier:
        return [ [ float(i) for i in ligne.split(",") ]
                       for ligne in fichier.readlines()[1:] ]

# version peut-être encore plus pythonesque...
import csv
def importe_rando_3(nom_fichier):
    with open(nom_fichier, "r") as fichier:
        reader = csv.reader(fichier)
        next(reader) # saute la première ligne
        return [ [ float(i) for i in ligne ] for ligne in reader ]

# IGNORER ci-après, on teste une version au hasard...
import random as rnd
importe_rando = rnd.choice([ importe_rando_1, importe_rando_2, importe_rando_3 ])
```

❏ **Q6** – Implémenter la fonction `plus_haut`.

Fonction assez facile, on s'intéresse juste à l'index avant de retourner le résultat.

```
# version normale attendue d'un candidat
def plus_haut_1(coords):
    # on pourrait garder l'altitude maximale plutôt que de la rechercher
    i_altmax = 0
    for i in range(1, len(coords)):
        if coords[i][2] > coords[i_altmax][2]:
            i_altmax = i
    return [ coords[i_altmax][0], coords[i_altmax][1] ]

# version utilisant index et max...
def plus_haut_2(coords):
    alts = [ c[2] for c in coords ]
    i_altmax = alts.index(max(alts))
    return [ coords[i_altmax][0], coords[i_altmax][1] ]

# IGNORER ci-après, on teste une version au hasard...
import random as rnd
plus_haut = rnd.choice([ plus_haut_1, plus_haut_2 ])
```

❏ **Q7** – Implémenter la fonction `deniveles`.

Deux parcours sommations assez simples. La définition donnée précise clairement que le dénivelé cumulé négatif est négatif, il n'a pas à être mis en valeur absolue.

```
# version boucle classique
def deniveles_1(coords):
```

```
    dpos, dneg = 0.0, 0.0
    for i in range(len(coords) - 1):
        deniv = coords[i+1][2] - coords[i][2]
        if deniv > 0:
            dpos += deniv
        else:
            dneg += deniv
    return [ dpos, dneg ]

# version compréhension
def deniveles_2(coords):
    deltas = [ coords[i+1][2] - coords[i][2] for i in range(len(coords)-1) ]
    dpos = sum(max(d, 0.0) for d in deltas)
    dneg = sum(min(d, 0.0) for d in deltas)
    return [ dpos, dneg ]

# version avec filter
def denivele_3(coords):
    deltas = [ coords[i+1][2] - coords[i][2] for i in range(len(coords)-1) ]
    dpos = sum(filter(lambda a: a >= 0, deltas))
    dneg = sum(filter(lambda a: a <= 0, deltas))
    return [ dpos, dneg ]

# IGNORER ci-après, on teste une version au hasard...
import random as rnd
deniveles = rnd.choice([ deniveles_1, deniveles_2, denivele_3 ])
```

❏ **Q8** – Implémenter la fonction `distance`.

Prétexte pour vérifier l'aptitude à traduire une grosse formule non triviale en Python : conversions d'unités (radians et km), trigonométrie, géométrie, arithmétique... Vérifier la prise en compte de l'altitude dans le calcul de la distance sur le grand cercle. Il faut pénaliser du code très moche : on a demandé du lisible ! Dans la vraie vie, le bruit sur les données GPS pourrait donner des résultats assez imprécis. La fonction de haversine n'apporte pas grand chose en terme de précision par rapport à un calcul en ligne droite, qui de toute façon serait assez complexe vu les projections nécessaires, en particulier si on utilise WGS84. Il y a probablement des choses à dire sur la précision numérique.

```
RT = 6371 # IGNORER

import math as m

def distance(coord1, coord2):
    lat1, lon1, alt1, _ = coord1
    lat2, lon2, alt2, _ = coord2
    # conversion en radians
    phi1, lam1 = m.radians(lat1), m.radians(lon1)
    phi2, lam2 = m.radians(lat2), m.radians(lon2)
    alt = 0.5 * (alt1 + alt2)
    # calcule du grand cercle, conversion en mètre du rayon de la Terre
    dgc = 2.0 * (1000.0 * RT + alt) * \
        m.asin( m.sqrt( m.sin(0.5 * (phi2 - phi1)) ** 2 +
                        m.cos(phi1) * m.cos(phi2) *
```

```
                        m.sin(0.5 * (lam2 - lam1)) ** 2 ) )
    # prise en compte pythagoricienne de la variation d'altitude
    dis = m.sqrt( dgc ** 2 + (alt2 - alt1) ** 2 )
    return dis
```

❏ **Q9** – Implémenter la fonction `distance_totale`.

Parcours sommation assez simple, pour intégrer la formule précédente, structure proche du calcul du dénivelé, en plus simple.

```
from rando_distance import distance # IGNORER

# version boucle
def distance_totale_1(coords):
    d = 0.0
    for i in range(len(coords) - 1):
        d += distance(coords[i], coords[i+1])
    return d

# version compréhension
def distance_totale_2(coords):
    return sum(distance(coords[i], coords[i+1]) for i in range(len(coords) - 1))

# IGNORER ci-après, on teste une version au hasard...
import random as rnd
distance_totale = rnd.choice([ distance_totale_1, distance_totale_2 ])
```

Partie II. Mouvement brownien d'une petite particule

Un peu d'analyse. Intégration numérique d'une équation différentielle qui ne peut être résolue autrement. Approche élégante avec un vecteur d'état.

❏ **Q10** – Implémenter la fonction `vma(v1, a, v2)`.

Opération qui fait appelle au fma (floating point multiply-add) optimisé sur les FPU. Aptitude à la programmation défensive suggérée par l'assertion.

```
# version compréhension
def vma_1(v1, a, v2):
    assert len(v1) == len(v2)
    return [ v1[i] + a * v2[i] for i in range(len(v1)) ]

# version boucle
def vma_2(v1, a, v2):
    assert len(v1) == len(v2)
    v = []
    for i in range(len(v1)):
        v.append(v1[i] + a * v2[i])
    return v

# IGNORER ci-après, on teste une version au hasard...
```

```
import random as rnd
vma = rnd.choice([ vma_1, vma_2 ])
```

❏ **Q11** – Implémenter la fonction `derive(E)`.

Implique de comprendre le vecteur d'état, et d'implémenter l'équation différentielle avec sa composante aléatoire à projeter sur les axes.

```
MU, SIGMA, M, ALPHA = 0.0, 1E-8, 1E-6, 1E-5 # IGNORER

import random as rnd
import math as m

# version directe
def derive_1(E):
    vx, vy = E[2:]
    angle = rnd.uniform(0.0, m.pi) # pi suffit car la norme est "signée"
    norme = rnd.gauss(MU, SIGMA) / M # ou / M sur MU et SIGMA, ou plus tard
    return [ vx,
             vy,
             # on devrait probablement précalculer - ALPHA / M
             - ALPHA / M * vx + m.cos(angle) * norme,
             - ALPHA / M * vy + m.sin(angle) * norme ]

from brownien import vma # IGNORER

# version avec vma
def derive_2(E):
    # angle sur 2 pi et norme avec valeur absolue
    angle = rnd.uniform(- m.pi, m.pi)
    norme = abs(rnd.gauss(MU, SIGMA)) / M
    a_B = [ norme * m.cos(angle), norme * m.sin(angle) ]
    return E[2:] + vma( a_B, - ALPHA / M, E[2:] )

# IGNORER ci-après, on teste une version au hasard...
derive = rnd.choice([ derive_1, derive_2 ])
```

❏ **Q12** – Implémenter la fonction `euler(E0, dt, n)`.

Presque question de cours, un peu plus subtile avec un vecteur d'état. On demande n pas de simulation, qui s'ajoutent au point initial. Pour être générique, il faudrait avoir la fonction dérivée en paramètre.

```
from brownien import vma, derive # IGNORER

def euler(E0, dt, n):
    Es = [ E0 ]
    for i in range(n):
        Es.append( vma(Es[-1], dt, derive(Es[-1])) )
    return Es
```

Partie III. Marche auto-évitante

Deux algorithmes pour résoudre un même problème de nature algébrique. Compréhension de code, interprétation d'une courbe, petit calcul, petite question de cours dans un coin, question un tout petit peu plus ouverte à la fin...

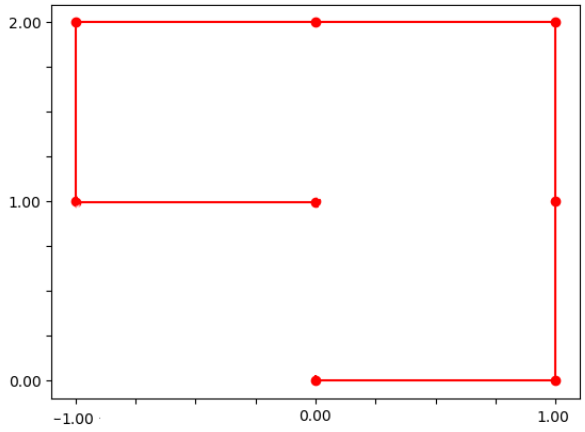
❏ **Q13** – Implémenter la fonction `positions_possibles(pos, atteints)`.

```
def positions_possibles(p, atteints):
    possibles = []
    x, y = p
    for q in [ [ x + 1, y ], [ x, y + 1 ], [ x - 1, y ], [ x, y - 1 ] ]:
        if not q in atteints:
            possibles.append(q)
    return possibles
```

❏ **Q14** – Mettre en évidence graphiquement un exemple de CAE le plus court pour lequel, à une étape donnée, la fonction `positions_possibles` va renvoyer une liste vide.

Une petite réflexion sur le fonctionnement de la méthode naïve, avant son implémentation. On ne demande pas de preuve de la minimalité du chemin exhibé. Si souhaité, celle-ci pourrait être obtenue par énumération exhaustive.

Voici un chemin bloquant, de longueur minimale 7 :



Il y a 8 symétries possibles qui donnent des chemins bloquants de même longueur : 4 rotations, plus miroir ou centrale.

❏ **Q15** – Implémenter la fonction `genere_chemin_naif(n)`.

Implémentation d'un algorithme fourni.

```
from chemin_naif import positions_possibles # IGNORER

import random as rnd

def genere_chemin_naif_1(n):
```

```
    chemin = [ [ 0, 0 ] ] # on part de l'origine
    for i in range(n):
        suivants = positions_possibles(chemin[-1], chemin)
        if len(suivants) == 0:
            return None
        chemin.append(rnd.choice(suivants))
    return chemin

# une version récursive
def genere_chemin_rec(chemin, n):
    if n <= 0:
        return chemin
    else:
        suivants = positions_possibles(chemin[-1], chemin)
        if len(suivants) == 0:
            return None
        return genere_chemin_rec(chemin + [ rnd.choice(suivants) ], n - 1)

def genere_chemin_naif_2(n):
    return genere_chemin_rec([ [ 0, 0 ] ], n)

# IGNORER ci-après, on teste une version au hasard...
genere_chemin_naif = rnd.choice([ genere_chemin_naif_1, genere_chemin_naif_2 ])
```

❏ **Q16** – Évaluer soigneusement la complexité de la fonction `genere_chemin_naif(n)` en fonction de `n` en supposant que la fonction ne renvoie pas `None`.

Python entretient la confusion tableau/liste, ce qui en fait un très mauvais outil pour discuter de complexité. Les différentes implémentations semblent utiliser des tableaux dont l'allocation est étendue quand nécessaire, mais les candidats peu curieux peuvent croire qu'ajouter un élément à une liste Python est au pire en temps constant, alors que c'est en réalité au mieux en temps constant, au pire en n et en moyenne en $\ln n$ (la politique d'extension étant proportionnelle à la longueur de la liste, au moins pour CPython).

La fonction `positions_possibles` fait une affectation en temps constant puis teste 4 fois l'appartenance d'un nouveau point au chemin en cours de construction (opérateur `in`), ce test étant supposé séquentiel, pour construire une liste de longueur maximale 4, donc d'ordre constant. Si ℓ est la longueur du chemin en cours de construction, la complexité asymptotique au pire est donc celle des tests d'appartenance, donc 4ℓ, donc de l'ordre de ℓ.

Si la fonction `genere_chemin_naif` ne renvoie pas `None`, alors la fonction `positions_possibles` renvoie toujours une liste avec au moins un élément, donc le chemin a été parcouru complètement au moins une fois pour vérifier que ce point n'en fait pas partie, c'est donc aussi la complexité dans le meilleur cas.

La fonction `genere_chemin_naif` appelle la fonction `position_possibles` pour des longueurs croissantes jusque n, puis opère un choix à coût constant sur le résultat supposé non vide et ajoute cet élément au chemin, opération dont la complexité dépend de l'implémentation sous-jacente, au pire la longueur pour une implémentation à base de tableau. Le test sur la longueur de la liste de choix est lui constant. Sa complexité en fonction de la longueur du chemin généré, si celui-ci n'est pas bloqué, est donc pour les tests d'appartenance et l'insertion de l'élément choisi :

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

donc de l'ordre de n^2. Autrement dit, la fonction `genere_chemin_naif` est de complexité quadratique en fonction de la longueur du chemin généré, quand il n'y a pas de blocage.

❏ **Q17** – Décrire et interpréter le graphique.

Cette question nous éclaire sur la probabilité de blocage, qui en réalité est assez vite très importante.

Le code évalue la probabilité de blocage (il teste si `genere_chemin_naif` a renvoyé `None`) pour toutes les longueurs de chemin de 1 à 350, en effectuant 10000 essais pour chaque longueur.

La figure affiche les longueurs testées en abscisse, et l'évaluation de la probabilité de blocage en ordonnée.

Pour des chemins très courts, la probabilité de blocage est nulle car il n'y a aucune possibilité de blocage, comme vu à une question précédente, la courbe est donc complètement plate.

Comme l'algorithme est glouton, la probabilité d'échec ne peut qu'augmenter en fonction de la longueur : en effet, il faut déjà avoir réussi à construire les longueurs inférieures avant que d'ajouter un point, les coupures pour les chemins plus court empêchent donc la construction des longueurs subséquentes. Ceci est bien observé expérimentalement.

On constate que cette probabilité tend assez vite vers 1 : la complexité quadratique décrite à la question précédente est donc sous une hypothèse optimiste (pas de blocage), la réalité est que les blocages sont fréquents, voir très fréquents, pour des chemins de quelques centaines points.

La méthode naïve va donc être très inefficace pour générer des CAE de quelques centaines de points.

❏ **Q18** – Donner, sans la justifier, la complexité temporelle asymptotique dans le pire des cas attendue de `sorted` en fonction de la longueur de la liste à trier. Donner le nom d'un algorithme possible pour son implémentation.

On pourrait chipoter un peu entre liste et tableau, mais transformer une liste en tableau ou un tableau en liste coûte n. Attention, quicksort *est quadratique au pire.*

La complexité temporelle asymptotique dans le pire des cas du meilleur tri par comparaison possible sur une liste de taille n est $n \ln n$. Les algorithmes de tri fusion (*merge sort*), de tri par tas (*heapsort*) ou la variante du tri rapide *introsort* peuvent par exemple être utilisés.

❏ **Q19** – Implémenter la fonction `est_CAE(chemin)` et discuter sa complexité.

Une fois la liste triée, les éléments égaux sont consécutifs, ce qui facilite leur détection avec un simple parcourt séquentiel.

```
# version qui suppose que trie[i] est en O(1)
def est_CAE_1(chemin):
    trie = sorted(chemin)
    for i in range(len(trie)-1):
        if trie[i] == trie[i+1]:
            return False
    return True

# version qui ne fait pas cette hypothèse
def est_CAE_2(chemin):
    trie, courant = sorted(chemin), None
    for suivant in trie:
        if courant is not None and suivant == courant:
            return False
        courant = suivant
    return True

# version similaire mais sans None
def est_CAE_3(chemin):
```

```
    trie = sorted(chemin)
    courant = trie[0]
    for suivant in trie[1:]:
        if suivant == courant:
            return False
        courant = suivant
    return True

# IGNORER ci-après, on teste une version au hasard...
import random as rnd
est_CAE = rnd.choice([ est_CAE_1, est_CAE_2, est_CAE_3 ])
```

La complexité du tri de la liste de taille n est de l'ordre de $n \ln n$. Le parcours séquentiel pour trouver les éléments consécutifs égaux, est lui de l'ordre de n. Les autres opérations (affectations, accès, comparaisons) sont en temps constant. La complexité globale de la fonction est donc celle du tri.

❏ **Q20** – Implémenter la fonction `rot(p, q, a)`.

Faire simple, pas la peine de s'embêter avec des matrices de rotation.

```
def rot(p, q, a):
    assert a in [ 0, 1, 2 ] # IGNORER
    x, y = p
    dx, dy = q[0] - x, q[1] - y
    if a == 0:
        return [ x - dx, y - dy ]
    elif a == 1:
        return [ x - dy, y + dx ]
    else: # a == 2
        return [ x + dy, y - dx ]
```

❏ **Q21** – Implémenter la fonction `rotation(chemin, i_piv, a)`.

On a insisté pour dire que ça tourne après le pivot, donc pénaliser les solutions qui incluent le pivot, même si la rotation du pivot sur lui-même ne le modifie pas.

```
from chemin_pivot import rot # IGNORER

# version compréhension
def rotation_1(chemin, i_pivot, a):
    return chemin[:i_pivot+1] + \
           [ rot(chemin[i_pivot], p, a) for p in chemin[i_pivot+1:] ]

# version avec des boucles bien explicites
def rotation_2(chemin, i_pivot, a):
    nouveau = []
    for i in range(i_pivot+1):
        nouveau.append(chemin[i])
    for i in range(i_pivot+1, len(chemin)):
        nouveau.append(rot(chemin[i_pivot], chemin[i], a))
    return nouveau

# IGNORER ci-après, on teste une version au hasard...
```

```
import random as rnd
rotation = rnd.choice([ rotation_1, rotation_2 ])
```

❏ **Q22** – Implémenter la fonction `genere_chemin_pivot(n, n_rot)`.

Attention, les pivotages comptent, pas les tentatives qui échouent.

```
from chemin_pivot import est_CAE, rotation # IGNORER

import random as rnd

# version while simple
def genere_chemin_pivot_1(n, n_rot):
    chemin = [ [ i, 0 ] for i in range(n+1) ]
    while n_rot > 0:
        i_pivot, a = rnd.randrange(1, n), rnd.randrange(3)
        essai = rotation(chemin, i_pivot, a)
        if est_CAE(essai):
            chemin = essai
            n_rot -= 1
    return chemin

# version for/while
def genere_chemin_pivot_2(n, n_rot):
    chemin = [ [ i, 0 ] for i in range(n+1) ]
    for i in range(n_rot):
        fait = False
        while not fait:
            i_pivot, a = rnd.randrange(1, n), rnd.randrange(3)
            essai = rotation(chemin, i_pivot, a)
            if est_CAE(essai):
                chemin, fait = essai, True
    return chemin

# IGNORER ci-après, on teste une version au hasard...
import random as rnd
genere_chemin_pivot = rnd.choice([ genere_chemin_pivot_1, genere_chemin_pivot_2 ])
```

❏ **Q23** – On considère un pivot, son point précédent, son point suivant. Quel est l'impact prévisible sur les rotations admissibles ? Suggérer un moyen de prendre en compte cette propriété pour améliorer l'algorithme.

Question sous forme un peu ouverte, qui aurait pû être intégrée dans le flot de fonctions ci-dessus, mais vue la longueur du sujet cela permet de laisser réfléchir un peu plus l'élève débrouillard arrivé jusque là... Valoriser l'idée du code autant qu'un code opérationnel ?

Version courte : La rotation doit éviter de repartir d'où l'on vient, *i.e.* de mettre le point suivant sur le point précédent.

Version trop longue : Comme les points précédent et suivant sont équi-distants du pivot et chacun sur une des 4 directions cardinales distinctes, l'une des trois rotations rammène nécessairement le suivant sur le précédent, et donc aboutira à un échec assuré de la tentative de rotation.

On peut profiter de cette propriété pour éviter facilement une tentative coûteuse, la complexité de `est_CAE` étant en $n \ln n$. Une approche simple pour cela est de tester les 3 rotations pour détecter laquelle est inutile, avant le choix aléatoire parmi les 2 restantes, comme dans la version ci-après :

```python
from chemin_pivot import est_CAE, rot, rotation # IGNORER

import random as rnd

# variante déterministe
def genere_chemin_pivot_mieux_1(n, n_rot):
    chemin = [ [ i, 0 ] for i in range(n+1) ]
    while n_rot > 0:
        i_pivot = rnd.randrange(1, n)
        # on saute la rotation qui rammènent le suivant sur le précédent
        la = []
        for a in range(3):
            if rot(chemin[i_pivot], chemin[i_pivot+1], a) != chemin[i_pivot-1]:
                la.append(a)
        # avant de choisir dans celles qui restent
        a = rnd.choice(la)
        essai = rotation(chemin, i_pivot, a)
        if est_CAE(essai):
            chemin = essai
            n_rot -= 1
    return chemin

# variante probabiliste
def genere_chemin_pivot_mieux_2(n, n_rot):
    chemin = [ [ i, 0 ] for i in range(n+1) ]
    while n_rot > 0:
        i_pivot = rnd.randrange(1, n)
        # Python manque d'un do/while
        a = rnd.randrange(3)
        while rot(chemin[i_pivot], chemin[i_pivot+1], a) == chemin[i_pivot-1]:
            a = rnd.randrange(3)
        essai = rotation(chemin, i_pivot, a)
        if est_CAE(essai):
            chemin = essai
            n_rot -= 1
    return chemin

# IGNORER ci-après, on teste une version au hasard
genere_chemin_pivot_mieux = \
    rnd.choice([ genere_chemin_pivot_mieux_1, genere_chemin_pivot_mieux_2])
```