I. Randonnée

1 On peut proposer par exemple

```
SELECT COUNT(*)
FROM Participant
WHERE ne >= 1993 AND ne <= 2003;
```

> Il aurait été acceptable de proposer, sous réserve de connaître le mot-clef `BETWEEN`,
>
> ```
> SELECT COUNT(*)
> FROM Participant
> WHERE ne BETWEEN 1993 AND 2003;
> ```

2 Cette deuxième requête demande de réunir les enregistrements de même difficulté afin de calculer par agrégation de leurs durées moyennes. On peut donc proposer

```
SELECT diff, AVG(duree)
FROM Rando
GROUP BY diff;
```

> Notons qu'il n'est pas clair dans le programme que `GROUP BY` soit autorisé. Par conséquent, en tirant profit du fait qu'il y a cinq difficultés possibles de randonnée, on peut également proposer la requête suivante, qui utilise l'union de plusieurs requêtes. C'est évidemment beaucoup moins élégant.
>
> ```
> SELECT diff, AVG(duree) FROM Rando WHERE diff = 1
> UNION
> SELECT diff, AVG(duree) FROM Rando WHERE diff = 2
> UNION
> SELECT diff, AVG(duree) FROM Rando WHERE diff = 3
> UNION
> SELECT diff, AVG(duree) FROM Rando WHERE diff = 4
> UNION
> SELECT diff, AVG(duree) FROM Rando WHERE diff = 5;
> ```

3 On peut proposer une approche consistant à récupérer l'information de la difficulté de la randonnée 42, afin d'utiliser celle-ci comme requête imbriquée dans une seconde requête ayant pour but d'identifier les bons randonneurs.

```
SELECT pnom
FROM Participant
WHERE diff_max < (SELECT diff FROM Rando WHERE rid = 42);
```

> Toutefois, on peut être tenté d'effectuer une jointure, qui semble moins naturelle car il n'y a aucun lien de jointure entre les deux tables. Par conséquent, et en l'absence de ce lien, cette jointure s'apparente au produit cartésien des deux tables. On retient alors les noms des randonneurs pour lesquels la difficulté maximale supportée est inférieure à celle de la randonnée 42.
>
> ```
> SELECT pnom
> FROM Participant JOIN Rando
> WHERE rid = 42 AND diff_max < diff;
> ```

> Il est possible toutefois de faire un lien de jointure qui soit directement une inégalité, solution très proche de la précédente.
>
> ```
> SELECT pnom
> FROM Participant JOIN Rando ON diff_max < diff
> WHERE rid = 42;
> ```

4 On veut étudier tous les couples de randonnées ayant le même nom, mais pas le même identifiant, autrement dit des clefs primaires différentes. On est donc obligé de réaliser une autojointure, c'est-à-dire une jointure de la table `Rando` avec elle-même. Une fois cette jointure réalisée, et afin de distinguer les randonnées venant de chaque couple, il faut procéder à un renommage de celles-ci avec le mot-clef `AS`. Enfin, le sujet demande de réunir ces informations en supprimant les doublons, ce que l'on peut faire avec le mot-clef `DISTINCT`. Finalement, on peut proposer la requête suivante.

```
SELECT DISTINCT r1.rid
FROM Rando AS r1 JOIN Rando AS r2 ON r1.rnom = r2.rnom
WHERE r1.rid <> r2.rid;
```

> Cette question est particulièrement difficile, car il n'est pas clair que le mot-clef `DISTINCT` soit autorisé et la notion d'auto-jointure est sans doute la plus subtile de celles que l'on peut rencontrer en base de données dans le programme d'informatique de classes préparatoires. Bien comprendre cette question est toutefois très bénéfique pour le SQL en général et il est intéressant de passer le temps nécessaire à bien la saisir.

5 On utilise pour cette question ce qui est présenté dans le sujet. On commence par ouvrir le fichier, puis on lit la première ligne afin de s'en débarrasser, car elle ne sert pas dans la suite. On récupère alors les lignes les unes après les autres, en prenant soin à chaque étape d'utiliser la méthode `split` afin d'extraire les quatre données utiles, et enfin en n'oubliant pas de changer de type de variable grâce à la fonction `float`, qui convertit une chaîne de caractères en flottant. On n'oublie pas de fermer le fichier.

```
def importe_rando(nom_fichier):
    # Ouverture de nom_fichier en lecture
    fichier = open(nom_fichier, "r")
    # On lit la première ligne, qui ne sert pas
    fichier.readline()
    # On récupère les lignes suivantes
    lignes = fichier.readlines()
    fichier.close()  # Fermeture de l'objet fichier
    coords = [ ]
    for l in lignes:  # On parcourt chaque ligne du fichier
        # On découpe la ligne par rapport à la virgule
        interm = l.split(",")
        lat, lon, alt, t = interm  # On récupère chacun des 4 termes
        # Puis on les stocke en les convertissant en flottants
        coords.append([float(lat), float(lon), float(alt), float(t)])
    return coords
```

> La fermeture de fichier peut s'effectuer dès que l'ensemble des informations désirées est constitué, ce qui est le cas après la fin du `readlines`. Cette fermeture aurait tout aussi bien pu s'effectuer juste avant le `return`.

6 On programme ici une variante de la recherche du maximum d'une liste, dans une situation où ce que l'on veut retenir est une partie d'une structure : ici, une liste de quatre éléments, dont on veut retenir les deux premiers pour la liste ayant le plus grand troisième élément. Une solution possible consiste à repérer tout d'abord l'indice de la liste ayant l'altitude la plus élevée, afin d'extraire les éléments voulus en tout dernier lieu. On boucle alors sur l'indice des éléments de la liste d'entrée.

```
def plus_haut(coords):
    # Indice provisoire de la plus grande altitude rencontrée
    i_alt_max = 0
    # On balaie les points, sauf le premier
    for i in range(1, len(coords)):
        # On teste si c'est une nouvelle altitude max
        if coords[i][2] > coords[i_alt_max][2]:
            # On actualise alors l'indice de ce nouveau maximum
            i_alt_max = i
    return coords[i_alt_max][0:2]
```

> Une autre approche consiste à stocker d'une part les longitude et latitude du premier point, et d'autre part l'altitude de ce même premier point qui est défini comme maximum provisoire. On parcourt alors tous les autres points de la liste fabriquée à la question précédente, afin de comparer le maximum provisoire et l'altitude du point courant. Si cette dernière est supérieure au maximum provisoire, elle devient le nouveau maximum provisoire et on stocke les latitude et longitude du point.
>
> ```
> def plus_haut(coords):
>     # On stocke le premier point
>     ph = coords[0][0:2]
>     alt_max = coords[0][2]
>     # On balaie les autres points
>     for c in coords[1:]:
>         if c[2] > alt_max:  # Nouvelle altitude max trouvée !
>             alt_max = c[2]  # On actualise le nouveau maximum
>             ph = c[0:2]  # On stocke ses latitude et longitude
>     return ph
> ```

7 Il faut calculer la variation d'altitude entre deux points consécutifs. Si la différence est positive, on l'ajoute à l'encours du dénivelé positif, et sinon on l'ajoute à celui du dénivelé négatif. Chaque itération utilise les informations venant d'un point et du suivant, il faut donc s'arrêter à l'avant-dernier puisque l'on utilise alors le dernier pour calculer la dernière variation d'altitude.

```
def deniveles(coords):
    deniv_pos, deniv_neg = 0, 0
    for i in range(len(coords)-1):
        variation = coords[i+1][2]-coords[i][2]
        if variation > 0:
            deniv_pos = deniv_pos + variation
        else:
            deniv_neg = deniv_neg + variation
    return [deniv_pos, deniv_neg]
```

8 La fonction que l'on doit programmer nécessite l'usage de fonctions mathématiques qu'il faut par conséquent penser à appeler. L'énoncé demandant de décomposer les formules pour gagner en lisibilité, on commence donc par récupérer les informations des deux points de passage considérés, puis on convertit les angles en radians, avant de programmer la formule de haversine. Rappelons que le rayon terrestre est défini comme variable globale par l'énoncé.

```
import math as m

def distance(c1, c2):
    # Récupération des informations de chaque point
    lat1, long1, alt1, t1 = c1
    lat2, long2, alt2, t2 = c2
    # Conversion en radians
    phi1, lamb1 = m.radians(lat1), m.radians(long1)
    phi2, lamb2 = m.radians(lat2), m.radians(long2)
    # Variable de l'altitude moyenne entre c1 et c2
    altmoy = (alt1 + alt2)/2
    # Calcul de r, attention RT est donné en km
    r = 1000*RT + altmoy
    # Formule proprement dite
    inter1 = m.sin((phi2 - phi1)/2)**2
    inter2 = m.cos(phi1)*m.cos(phi2)*m.sin((lamb2 - lamb1)/2)**2
    d = 2*r*m.asin(m.sqrt(inter1 + inter2))
    return d
```

> L'énoncé ne dit pas clairement comment sont structurés les points de passage et donc les deux arguments d'entrée `c1` et `c2`, mais il semble assez évident que c'est la même que pour un élément particulier de la liste `coords`.

Le sujet n'est pas très clair sur la fin de la question, il semble que la distance qui vient d'être calculée soit celle du segment droit perpendiculaire à la verticale évoqué dans le sujet. Comme la variation d'altitude correspond, selon la verticale, à une longueur `alt2 - alt1`, la distance réellement attendue vaudrait

$$\sqrt{(\texttt{alt2} - \texttt{alt1})^2 + \texttt{d}^2}$$

ce qui conduirait à changer la dernière ligne du code précédent en

```
return m.sqrt(d**2 + (alt2-alt1)**2)
```

9 Cette dernière fonction calcule entre deux points consécutifs la distance parcourue grâce à la fonction de la question précédente et cumule cette distance partielle avec l'encours de la distance parcourue depuis le début.

```
def distance_totale(coords):
    dtot = 0
    for i in range(0, len(coords)-1):
        dis = distance(coords[i], coords[i+1])
        dtot = dtot + dis
    return dtot
```

> On peut toujours appeler les fonctions introduites dans les questions précédentes. Ne pas avoir réussi à écrire la fonction de la question 8 n'empêchait donc pas de coder celle-ci.

II. Mouvement brownien d'une petite particule

10 On suit l'énoncé qui indique ce qu'il faut faire. On passe par un vecteur initialement vide afin de le remplir au fur et à mesure par les coordonnées désirées.

```
def vma(v1, a, v2):
    assert len(v1) == len(v2)
    n = len(v1)
    v = []
    for i in range(n):
        v.append(v1[i] + a*v2[i])
    return v
```

11 Il faut tout d'abord importer les deux bibliothèques utilisées plus bas :

```
import math as m
import random as rd
```

On observe ensuite que les deux premières composantes du vecteur dérivé $\dot{\mathrm{E}}$ sont en fait les deux dernières du vecteur E. La première ligne du programme correspond donc à la récupération de ces deux éléments. Il reste alors à déterminer les deux dérivées secondes selon x et y, c'est-à-dire le vecteur accélération. Celui-ci est directement calculé grâce à l'équation différentielle. Il est nécessaire pour cela de calculer au hasard une norme et un angle selon les indications de l'énoncé, pour établir l'expression du vecteur $\overrightarrow{f_{\mathrm{B}}}$, avant d'utiliser le programme de la question précédente pour calculer l'accélération, puis la dérivée du vecteur d'état.

```
def derive(E):
    V = [E[2], E[3]]  # Constitution du vecteur vitesse
    angle = rd.uniform(-m.pi, m.pi)  # Isotropie de l'angle
    norme = abs(rd.gauss(MU, SIGMA))/M  # Norme
    Fb = [m.cos(angle)*norme, m.sin(angle)*norme]
    A = vma(Fb, - ALPHA/M, V)
    return V + A  # Concaténation de vitesse et accélération
```

12 Connaissant le vecteur d'état à l'instant t, on peut calculer celui à l'instant $t+\mathrm{d}t$ grâce à un développement limité à l'ordre 1 :

$$\mathrm{E}(t+\mathrm{d}t) = \mathrm{E}(t) + \mathrm{d}t \times \dot{\mathrm{E}}(t)$$

Ainsi, on part d'un vecteur d'état initial que l'on stocke dans une liste, puis on calcule les nouveaux vecteurs d'état de proche en proche grâce à cette relation.

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

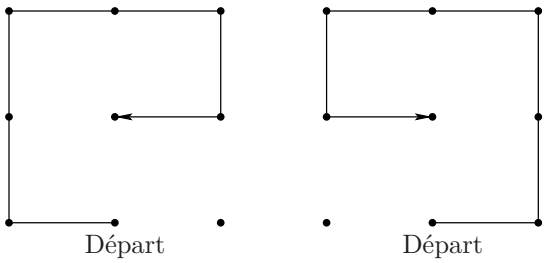
III. Marche auto-évitante

13 Chaque point possède quatre voisins possibles. Le programme proposé calcule les quatre et retient ceux qui ne sont pas déjà dans la liste `atteints`.

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

14 Un CAE conduisant à une évolution impossible consiste à cerner le plus rapidement possible les quatre voisins d'un point. On peut par exemple proposer les deux chemins suivants. Ainsi, aux symétries et aux sens d'enroulements près,

Il y a huit CAE d'une même longueur minimale de 7.



Le sujet ne demande pas de démonstration de cette minimalité.

15 Cette programmation nécessite de tirer des grandeurs aléatoirement, par conséquent il faut importer la bibliothèque `random`. L'idée du programme est d'utiliser la fonction de la question 13, en cherchant les positions atteignables possibles en partant du dernier point du chemin. On regarde alors si cette liste est vide, dans ce cas le chemin ne peut pas aller plus loin, et on le fait échouer, conformément au point 2 de l'algorithme décrit (l'algorithme ne cherche pas à revenir en arrière pour trouver un autre chemin par exemple). Chaque recherche d'un nouveau point se fait dans une boucle qui s'arrête lorsque `n` points ont été obtenus.

```
import random as rd

def genere_chemin_naif(n):
    chemin = [[0, 0]]
    for i in range(n):
        prochains = positions_possibles(chemin[-1], chemin)
        if prochains == []:
            return None
        # On choisit un point au hasard parmi les points possibles
        chemin.append(rd.choice(prochains))
    return chemin
```

16 Si la fonction précédente ne renvoie pas `None`, cela veut dire qu'on a parcouru entièrement la boucle. Il faut donc établir le coût d'une itération lorsque le chemin comporte `k` points. Dans une telle situation, le coût de l'itération est celui de l'appel à la fonction `positions_possibles` pour une liste de longueur `k`. Or celle-ci consiste à tester quatre positions en regardant pour chacune d'entre elles si elles sont contenues dans la liste des points déjà atteints, liste qui possède une longueur `k`. Dans le pire des cas, c'est-à-dire lorsque ce voisin n'a pas déjà été atteint ou alors s'il l'a été mais que cette position constitue la dernière de `atteints`, il faut un parcours séquentiel de `k` éléments, ainsi que le rappelle l'énoncé. Globalement, l'appel de cette fonction nécessite alors `4k` opérations. Si on revient sur la fonction de la question précédente, il faut par conséquent dans le pire des cas 4, puis 8, puis 12... opérations, ou encore

$$4 \times 1 + 4 \times 2 + \cdots + 4 \times k + \cdots + 4 \times n = 4 \times \frac{n(n+1)}{2} = \mathrm{O}\left(n^2\right)$$

Asymptotiquement, cet algorithme possède une complexité quadratique.

17 Le code proposé tente de créer un chemin de façon naïve et de longueur `n`. Si la création du chemin échoue, une variable `nb` s'incrémente (lignes 6 et 10). En renouvelant cette tentative 10 000 fois, le programme permet d'estimer la probabilité d'échec. La liste `L` contient les valeurs de `n` évaluées, qui sont les entiers allant de 1 à 350, et la liste `P` contient, pour chaque valeur de `n`, le taux d'échec dans la recherche d'un CAE. Finalement,

Ce programme permet d'afficher le graphique du taux d'échec dans la fabrication d'un CAE avec la méthode naïve, en fonction de la taille du chemin espéré.

> Notons qu'on voit qu'il y a toujours réussite au début puisque le taux d'échec est nul, jusqu'à une valeur visiblement proche de 7, obtenue à la question précédente. On parle d'algorithme glouton, qualifiant ainsi le fait de procéder étape par étape en espérant atteindre une solution, quitte à recommencer plusieurs fois la tentative. Plus généralement, il s'agit d'une catégorie d'algorithmes cherchant à effectuer à chaque itération un choix d'optimum local en espérant que celui-ci conduise à un optimum global. Les deux exemples les plus célèbres de cette catégorie sont l'algorithme de Dijkstra de recherche d'un plus court chemin et le problème du rendu de monnaie, consistant à trouver comment obtenir une somme d'argent fixée avec le moins de pièces possible.

On constate que la probabilité d'échec est croissante, ce qui est logique : lorsque l'on fabrique un chemin de taille `n`, il faut d'abord en construire un de taille `n-1`, la probabilité d'échec ne pouvant donc que croître avec `n`. On remarque également qu'assez rapidement, il n'est plus raisonnablement possible d'espérer construire un chemin avec cette méthode naïve, le taux d'échec étant proche de 1. Cet algorithme n'est donc pas adapté lorsque l'on veut créer un CAE d'une taille supérieure à 200.

18 La meilleure complexité dans le pire cas que l'on puisse attendre d'un algorithme de tri par comparaison d'une liste de taille n est une complexité quasi linéaire, en $\mathrm{O}(n \log n)$. **C'est le cas du tri fusion.**

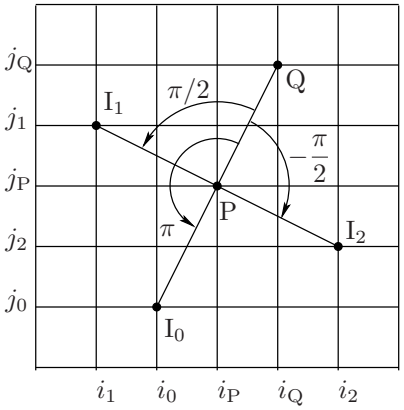
> Rappelons que les deux autres tris au programme, les tris insertion et rapide (Quicksort), sont de complexité quadratique dans le pire cas.

19 On commence par trier le chemin, ce qui fait que si deux éléments sont égaux (ce qui correspond à un chemin non auto-évitant), ils sont consécutifs. On parcourt donc la liste triée en vérifiant si deux éléments consécutifs sont égaux et dans ce cas on renvoie `False`. Si on ne rencontre pas cette situation, on renvoie `True` à la fin du balayage du chemin dans son intégralité.

```
def est_CAE(chemin):
    chemin_trie = sorted(chemin)
    for i in range(len(chemin)-1):
        if chemin_trie[i] == chemin_trie[i+1]:
            return False
    return True
```

Supposons que l'accès à un élément de la liste triée soit en temps constant, la boucle coûte alors une complexité linéaire donc cette fonction est bien de la même complexité que celle de `sorted`.

20 À partir d'un point P dont on récupère les coordonnées `ip` et `jp` et d'un second point Q de coordonnées `iq` et `jq`, il faut réfléchir à celles du point obtenu pour chacune des rotations proposées. Pour cela, prenons l'exemple du dessin suivant, le point I_0 correspondant à l'angle π, le point I_1 à l'angle $\pi/2$ et le point I_2 à l'angle $-\pi/2$, conformément à l'énoncé pour les valeurs de la variable d'entrée `a`. Prenons l'exemple de la rotation d'angle $\pi/2$: l'abscisse du point I_1 est obtenue en enlevant à celle du point P la différence des ordonnées entre ce même point et le point Q.



Sur le dessin, cette variation d'ordonnée vaut 2, qui correspond bien à la diminution de l'abscisse de P. De même, l'ordonnée de I_1 est obtenue en ajoutant cette fois-ci la variation d'abscisse entre P et Q. En raisonnant de la même façon pour les autres angles, on peut proposer le code suivant.

```
def rot(p, q, a) :
    ip, jp = p
    iq, jq = q
    di = iq - ip  # Variation d'abscisse
    dj = jq - jp  # Variation d'ordonnée
    if a == 0:  # Cas d'une rotation de pi
        return [ip - di, jp - dj]
    if a == 1:  # Cas d'une rotation de 90 degrés
        return [ip - dj, jp + di]
    else:  # Cas d'une rotation de -90 degrés
        return [ip + dj, jp - di]
```

> On peut aussi retrouver ces formules en interprétant matriciellement les rotations. En effet, la matrice d'une rotation d'angle θ est
>
> $$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

Dans le cas d'une rotation d'angle $\pi/2$, on peut alors écrire que

$$\overrightarrow{\mathrm{PI}_1} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \overrightarrow{\mathrm{PQ}} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} i_\mathrm{Q} - i_\mathrm{P} \\ j_\mathrm{Q} - j_\mathrm{P} \end{pmatrix} = \begin{pmatrix} j_\mathrm{P} - j_\mathrm{Q} \\ i_\mathrm{Q} - i_\mathrm{P} \end{pmatrix}$$

On retrouve donc bien que I_1 a pour coordonnées $(j_\mathrm{P} - j_\mathrm{Q} + i_\mathrm{P}, i_\mathrm{Q} - i_\mathrm{P} + j_\mathrm{P})$.

On suppose ici que l'entrée `a` vaut forcément une des valeurs autorisées, soit 0, 1 ou 2. On peut s'en assurer en effectuant un dernier test plutôt qu'un `else`, pour comparer `a` à 2, et sinon renvoyer `None`. Dans l'esprit du sujet, on peut aussi débuter la fonction par un test de validité de cette variable, en plaçant juste après la première ligne le code suivant :

```
assert a in [0, 1, 2]  # Ou encore a == 0 or a == 1 or a == 2
```

21 On procède conformément à l'énoncé : on part d'une liste vide dans laquelle on stocke tous les éléments jusqu'au pivot inclus, puis on applique à chacun des points suivants la rotation désirée grâce à la fonction précédente.

```
def rotation(chemin, i_piv, a):
    # Pour inclure le pivot, on va jusqu'à i_piv+1
    nouveau = chemin[0:i_piv+1]
    p = chemin[i_piv]
    for i in range(i_piv+1, len(chemin)):
        nouveau.append(rot(p, chemin[i], a)
    return nouveau
```

Le sujet insiste sur le fait que le pivot soit inclus dans la partie du chemin qui n'a pas bougé. Ce n'est pourtant pas fondamental, puisque modifier le pivot ou non ne change rien car c'est un point fixe de la rotation.

22 On suit une fois de plus l'algorithme décrit par l'énoncé. On génère tout d'abord un chemin de longueur `n` allant tout droit, puis on effectue une boucle de `n_rot` itérations qui doit permettre d'effectuer autant de rotations. Ainsi formulé par l'énoncé, les rotations que l'on retient doivent être les rotations réussies. Dans ce but, on tire au sort deux nombres, le premier pour avoir un point au hasard du chemin autre que les extrémités, le second pour avoir un angle au hasard. On calcule alors le nouveau chemin, et on teste s'il est auto-évitant. Si c'est le cas, on ajoute une rotation réussie, sinon on tire une nouvelle paire de nombres.

```
def genere_chemin_pivot(n, n_rot):
    # Création du chemin droit initial
    chemin = []
    for i in range(n+1):
        chemin.append([i,0])
    # Boucle des rotations
    n_ok = 0
    while n_ok < n_rot:
        i_piv = rd.randrange(1, n)
        a = rd.randrange(0, 3)
        essai = rotation(chemin, i_piv, a)
        if est_CAE(essai):
            chemin = essai
            n_ok = n_ok + 1
    return chemin
```

23 Par construction, deux points consécutifs sont toujours espacés d'une même distance, propriété qui reste invariante par rotation. En particulier, les points précédent et suivant le pivot restent donc toujours équidistants, après n'importe quelle rotation. Par conséquent, les angles des rotations étant ceux proposés, il y a une chance sur trois que la rotation tirée au sort renvoie le point après le pivot sur le point avant. En quelque sorte, on doit éviter que la rotation choisie ne fasse revenir à la position précédente. En déterminant, après le tirage d'un nouveau pivot, l'angle de la rotation entre le point précédent et le point suivant, on peut interdire une valeur de rotation dans le tirage au sort afin de ne pas effectuer des tirages pour rien.

> À moins d'avoir beaucoup de temps, le sujet ne demande pas un code pour cette dernière question, mais plutôt des idées d'améliorations. Il ne faut donc pas perdre trop de temps à fournir un code superflu, mais plutôt relire son sujet si on a fini celui-ci et vraiment à la toute fin, le cas échéant, se permettre de prendre le temps de faire un code. Cette dernière question est volontairement ouverte pour voir comment se comportent les étudiants. Les idées, plus que le code en lui-même, sont donc valorisées. Toutefois, on propose ci-dessous un code qui permet d'utiliser la remarque précédente. L'idée est de tester les rotations les unes après les autres et de stocker les valeurs qui ne renvoient pas le point suivant sur le précédent. On obtient alors une liste dans laquelle l'utilisation de la fonction `choice` permet de tirer au sort une rotation qui exclut le retour en arrière (ce qui ne veut pas dire que le chemin est alors auto-évitant, mais au moins l'est-il « juste autour » du pivot). Concrètement, on remplace la ligne du code précédent qui permet de tirer au sort `a` par les huit lignes de code suivantes :
>
> ```
> a_ok = []
> piv = chemin[i_piv]
> avant = chemin[i_piv-1]
> apres = chemin[i_piv+1]
> for i in range(0, 3):
>    if rot(piv, apres, i) != avant:
>       a_ok.append(i)
> a = rd.choice(a_ok)
> ```
>
> Le sujet pose une question ouverte en orientant la méthode utilisée. On pouvait imaginer par ailleurs par exemple une détection précoce du caractère non auto-évitant, en exploitant le fait que si intersection il y a, elle a plus de chances de se produire autour du pivot que très loin de lui. Ou encore, envisager l'application de la rotation sur la partie la plus courte du chemin, qu'elle se situe avant ou après le pivot, pour avoir deux fois moins de rotations à effectuer lors de la mise à jour. Cette catégorie d'algorithmes, mettant en œuvre des évolutions aléatoires, est connue sous le nom de méthodes de Monte-Carlo.