Extrait Centrale - Supelec
Mission d'exploration martienne

Mars Exploration Rovers (MER) est une mission de la NASA qui cherche à étudier le rôle joué par l'eau dans l'histoire de la planète Mars. Deux robots géologues se sont posés sur cette planète en janvier 2004. Leur mission est de rechercher et d'analyser différents types de roches et de sols qui peuvent contenir des indices sur la présence d'eau.

Chaque robot est équipé de plusieurs instruments d'analyse (caméra,microscope, spectromètres) et d'un bras qui permet d'amener les instruments au plus près des roches et sols dignes d'intérêt. À partir de photographies de la surface de la planète, prises à plusieurs longueurs d'ondes par différents satellites et par le robot lui-même, les scientifiques de la NASA définissent une liste d'emplacements (*points d'intérêt* ou PI) où effectuer des analyses. Cette liste est transmise au robot qui doit se rendre à chaque emplacement indiqué et y effectuer les analyses prévues. Chaque robot est capable d'effectuer un certain nombre de types d'analyses géologiques correspondant aux différents instruments dont il dispose. Une fois tous les points d'intérêts visités et les résultats des analyses transmis à la Terre, le robot reçoit une nouvelle liste de points d'intérêts et démarre une nouvelle *exploration*. Compte-tenu des contraintes de transmission entre la Terre et les robots (latence, périodes d'ombre, faible débit, etc.) il est prévu que les robots travaillent en autonomie pour planifier le parcours de chaque exploration. Ainsi, une fois la liste des points d'intérêt reçue, le robot analyse le terrain afin de détecter d'éventuels obstacles et détermine le meilleur chemin lui permettant de visiter l'ensemble de ces points en dépensant le moins d'énergie possible.

Après s'être intéressé à l'enregistrement des explorations, des points d'intérêts correspondants et des analyses à y mener, ce sujet aborde trois algorithmes qui peuvent être utilisés par le robot pour déterminer le meilleur parcours lui permettant de visiter chaque point d'intérêt une et une seule fois. Pour cela nous faisons quelques hypothèses simplificatrices :

- La zone d'exploration est dépourvue d'obstacle : le robot peut rejoindre directement en ligne droite n'importe quel point d'intérêt.
- Le sol est horizontal et de nature constante : l'énergie utilisé pour se déplacer entre deux points ne dépend que de leur distance, autrement dit le meilleur chemin est le plus court.
- La courbure de la planète est négligée compte tenu de la dimension réduite de la zone d'exploration : nous travaillerons en géométrie euclidienne et les points d'intérêts seront repérés par leurs coordonnées cartésiennes à l'intérieur de la zone d'exploration.

Les seuls langages de programmation autorisés dans cette épreuve sont `Python` et `SQL`. Toutes les questions sont indépendantes. Néanmoins, il est possible de faire appel à des fonctions ou procédures créées dans d'autres questions. Dans tout le sujet on suppose que la bibliothèque `random` a été importées grâce à l'instruction

```
import random
```

Si les candidats font appel à des fonctions d'autres bibliothèques ils doivent préciser les instructions d'importation correspondantes.

Ce sujet utilise la syntaxe suivante pour préciser dans l'en-tête d'une fonction les types des arguments et du résultat :

```
def maFonction(n:int, x:float, d:str) -> list:
```

Cela signifie que la fonction `maFonction` prend trois arguments, le premier est un entier, le deuxième un nombre à virgule flottante et le troisième une chaîne de caractères et qu'elle renvoie une liste.

Une liste de fonctions directement utilisables est donnée en annexe à la fin du sujet.

I Création d'une exploration et gestion des points d'intérêt

Une exploration est un ensemble de points d'intérêts à l'intérieur d'une zone géographique limitée, une série d'analyses étant associée à chaque point d'intérêt. Un point d'intérêt est repéré par deux entiers, positifs ou nuls, correspondant à ses coordonnées cartésiennes en millimètres à l'intérieur de la zone d'exploration. L'ensemble des n points d'intérêts d'une exploration est représenté par une liste de n listes `[`x, y`]` qui donnent l'abscisse et l'ordonnée du point d'intérêt. Par exemple :

```
[[345,635],[1076,415],[38,859],[121,582]]
```

I.1 Génération d'une exploration d'essai

I.1.1 Choix de points au hasard

Q1 Afin de disposer de données pour tester les différents algorithmes de calcul de chemin qui seront développés plus tard, écrire une fonction qui construit une exploration au hasard. Cette fonction d'en-tête :

```
def générer_PI(n:int, cmax:int) -> list:
```

prend en paramètres le nombre de points d'intérêts à générer et la largeur de la zone d'exploration (supposée carrée) et renvoie un objet de type `list` contenant les coordonnées de n points **deux à deux distincts** choisis au hasard dans la zone d'exploration.
Indication : on pourra penser à utiliser la fonction `in` décrite en annexe.

Q2 Quelles contraintes doivent vérifier les arguments de la fonction `generer_PI` pour qu'il soit possible d'obtenir un résultat ?

I.1.2 Calcul des distances

On dispose de la fonction d'en-tête :

```
def position_robot() -> list:
```

qui renvoie un couple donnant les coordonnées actuelles du robot dans le système de coordonnées de l'exploration à planifier. Ainsi l'instruction `[x, y] = position_robot()` permet de récupérer les coordonnées courantes du robot.

Q3 Afin de faciliter l'application des différents algorithmes de recherche de chemin, on souhaite construire un tableau des distances entre les différents points d'intérêt d'une exploration et entre ceux-ci et la position courante du robot au moment du calcul. Écrire une fonction d'en-tête :

```
def calculer_distances(PI:list) -> list:
```

qui prend en paramètre un tableau de n points d'intérêt tel que décrit précédemment et renvoie une liste `L` de $n+1$ listes formée chacune de $n+1$ nombres flottants (i.e. de type `float`) telle que `L[i][j]` fournit la distance entre les points d'intérêt i et j, l'indice `n` désignant le point de départ du robot (coordonnées actuelles).

I.2 Traitement d'image

On dispose de photographies d'une zone d'exploration effectuées à différentes longueur d'onde. Chaque photographie a été mises à l'échelle de la zone d'exploration puis stockée dans un tableau `photo` à deux dimensions (représenté par une liste de listes) d'entiers compris entre 0 et 255. Pour tout point de coordonnées géographiques (x, y), la valeur `photo[`x`][`y`]` est un entier entre 0 et 255 qui donne l'intensité lumineuse du point considéré sur la photographie. À partir de ces photographies, les géologues déterminent les endroits à analyser en filtrant ceux qui ont un profil d'émission caractéristique de certaines roches intéressantes.

I.2.1 Analyse d'une image

Q4 La fonction `F1` ci-dessous prend en paramètre une photographie représentée comme décrit plus haut. Expliquer ce que fait cette fonction et décrire son résultat.

```
def F1(photo:list) -> list:
    L = []
    for ligne in photo:
        L = L + ligne
    n = min(L)
    b = max(L)
    H = [0 for i in range(b - n + 1)]
    for p in L:
        H[p - n] += 1
    return H
```

I.2.2 Sélection de points d'intérêts

Q5 Écrire une fonction d'en-tête :

```
def selectionner_PI(photo:list, imin:int, imax:int) -> list:
```

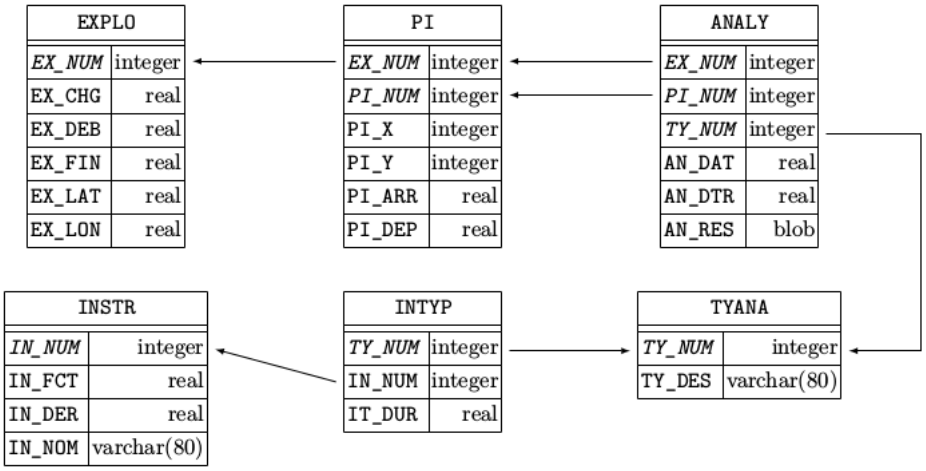
où `photo` est un tableau représentant une photographie.
Le résultat de la fonction `selectionner_PI` est la liste des coordonnées de points dont l'intensité sur la photographie est comprise (au sens large) entre `imin` et `imax`.

TSVP

I.3 Base de données

Afin d'assurer son autonomie opérationnelle, le robot dispose localement des informations nécessaires à ses déplacements quotidiens. Il enregistre notamment la prochaine exploration, c'est-à-dire les différents points d'intérêts qu'il doit visiter. Ces résultats ne sont effacés qu'après confirmation de leur bonne transmission sur Terre.

Ces différentes informations sont stockées dans une base de données relationnelle dont le modèle physique peut être schématisé de la manière suivante :



Cette base comporte les deux tables suivantes :

- la table `EXPLO` des explorations, avec les colonnes
  - `EX_NUM` numéro (entier) de l'exploration (clef primaire)
  - `EX_CHG` date de transmission des points d'intérêts de cette exploration
  - `EX_DEB` date de début de l'exploration (`NULL` si l'exploration n'est pas encore commencée)
  - `EX_FIN` date de fin de l'exploration (`NULL` si l'exploration n'est pas encore terminée)
  - `EX_LAT` latitude (en degrés décimaux) du point de coordonnées $(0, 0)$ de la zone d'exploration
  - `EX_LON` longitude (en degrés décimaux) du point de coordonnées $(0, 0)$ de la zone d'exploration
- la table `PI` des points d'intérêts, de clef primaire `(EX_NUM, PI_NUM)`, avec les colonnes
  - `EX_NUM` numéro de l'exploration à laquelle appartient le point d'intérêt
  - `PI_NUM` numéro du point d'intérêt dans l'exploration (au sein d'une exploration les PI sont numérotés en séquence en commençant à 0, ce numéro n'a pas de rapport avec l'ordre dans lequel les PI sont explorés par le robot)
  - `PI_X` l'abscisse du point d'intérêt dans la zone d'exploration (entier positif en millimètres)
  - `PI_Y` l'ordonnée du point d'intérêt dans la zone d'exploration (entier positif en millimètres)
  - `PI_ARR` date d'arrivée du robot au point d'intérêt (`NULL` si ce point n'a pas encore été visité)
  - `PI_DEP` date à laquelle le robot a quitté le point d'intérêt (`NULL` si ce point n'a pas encore été exploré ou si la visite est en cours)

Toutes les dates sont stockées sous forme d'un nombre décimal correspondant au nombre de jours martiens depuis l'arrivée du robot sur la planète. On utilisera `IS NULL` ou `IS NOT NULL` pour tester si un champ vaut `NULL` ou non.

Q6 Écrire une requête SQL qui donne le numéro de l'exploration en cours, s'il y en a une.

Q7 Écrire une requête SQL qui donne, pour une exploration dont on connaît le numéro (par exemple 42), la liste des points d'intérêts de cette exploration avec leurs coordonnées.

Q8 Écrire une requête SQL qui donne la surface, en mètres carrés, de chaque zone déjà explorée par le robot. La zone d'exploration est définie comme le plus petit rectangle qui englobe l'ensemble des points d'intérêts de l'exploration et dont les bords sont parallèles aux axes de référence (axes des abscisses et des ordonnées).

II Planification d'une exploration : première approche

Avant de démarrer une nouvelle exploration, le robot doit déterminer un chemin qui lui permet de passer par tous les points d'intérêts une et une seule fois. L'enjeu de l'opération est de trouver le chemin le plus court possible afin de limiter la dépense d'énergie et de limiter l'usure du robot.

Chaque point d'intérêt sera repéré par un entier positif ou nul correspondant à son indice dans le tableau des points d'intérêt. Un chemin d'exploration sera représenté par un objet de type `list` donnant les indices des points d'intérêt dans l'ordre de leur parcours.

II.1 Quelques fonctions utilitaires

II.1.1 Longueur d'un chemin

Q9 Écrire une fonction d'en-tête :

```
def longueur_chemin(chemin:list, d:list) -> float:
```

qui prend en paramètre un chemin à parcourir et la matrice des distances entre points d'intérêt (telle que renvoyée par la fonction `calculer_distances`) et qui renvoie la distance totale que doit effectuer le robot pour suivre ce chemin en partant de sa position courante (correspondant à la dernière ligne/colonne du tableau `d`) et en visitant tous les points d'intérêt dans l'ordre indiqué.

II.1.2 Normalisation d'un chemin

Q10 Écrire une fonction d'en-tête :

```
def normaliser_chemin(chemin:list, n:int) -> list:
```

qui prend en paramètre une liste `chemin` d'entiers positifs ou nuls et qui renvoie une liste contenant une seule fois tous les entiers entre `0` (inclus) et `n` (exclu). Pour cela cette fonction commence par supprimer dans `chemin` les éventuels doublons (en ne conservant que la première occurrence) ainsi que les valeurs supérieures ou égales à `n`, sans modifier l'ordre relatif des éléments conservés ; puis elle ajoute à la fin les éventuels éléments manquants, dans l'ordre croissant de numéros. Par exemple :

```
normaliser_chemin([1, 0, 1, 0, 5, 6, 6, 0, 2, 2, 1], 6)->[1, 0, 5, 2, 3, 4]
```

II.2 Force brute

Pour rechercher le plus court chemin, on peut imaginer de considérer tous les chemins possibles et de calculer leur longueur. On obtiendra ainsi à coup sûr le chemin le plus court.

Q11 Déterminer en fonction du nombre n de points à visiter, le nombre de chemins possibles passant exactement une fois par chacun des points.

Q12 Cet algorithme est-il utilisable pour une zone d'exploration contenant 20 points d'intérêts ? Justifier.

II.3 Algorithme du plus proche voisin

Une idée simple pour obtenir un algorithme utilisable est de construire un chemin en choisissant systématiquement le point, non encore visité, le plus proche de la position courante.

Q13 Écrire une fonction d'en-tête :

```
def plus_proche_voisin(d) -> list:
```

qui prend en paramètre le tableau des distances résultat de la fonction `calculer_distances` et fournit un chemin d'exploration en appliquant l'algorithme du plus proche voisin.

Q14 Quelle est la complexité temporelle de l'algorithme du plus proche voisin en considérant que cet algorithme est constitué des deux fonctions `calculer_distances` et `plus_proche_voisin` ?

Q15 En considérant les trois points de coordonnées `[0, 0]`, `[0, 3000]`, `[0, 7000]` et en choisissant un point de départ adéquat pour le robot, montrer que l'algorithme du plus proche voisin ne fournit pas nécessairement le plus court chemin.

Dans la pratique, on constate que, dès que le nombre de points d'intérêt devient important, l'algorithme du plus proche voisin fournit un chemin qui peut être 50% plus long que le plus court chemin.

III Deuxième approche : algorithme génétique

Les algorithmes génétiques s'inspirent de la théorie de l'évolution en simulant l'évolution d'une population. Ils font intervenir cinq traitements.

1. Initialisation
   Il s'agit de créer une population d'origine composée de m individus (ici des chemins possibles pour l'exploration à planifier de n points d'intérêt donnés). Généralement la population de départ est produite aléatoirement.
2. Évaluation
   Cette étape consiste à attribuer à chaque individu de la population courante une note correspondant à sa capacité à répondre au problème posé. Ici la note sera simplement la longueur du chemin.
3. Sélection
   Une fois tous les individus évalués, l'algorithme ne conserve que les « meilleurs » individus. Plusieurs méthodes de sélection sont possibles : choix aléatoire, ceux qui ont obtenu la meilleure note, élimination par tournoi, etc.
4. Croisement
   Les individus sélectionnés sont croisés deux à deux pour produire de nouveaux individus et donc une nouvelle population. La fonction de croisement (ou reproduction) dépend de la nature des individus.

5. Mutation
   Une proportion d'individus est choisie (généralement aléatoirement) pour subir une mutation, c'est-à-dire une transformation aléatoire. Cette étape permet d'éviter à l'algorithme de rester bloqué sur un optimum local.

En répétant les étapes de sélection, croisement et mutation, l'algorithme fait ainsi évoluer la population, jusqu'à trouver un individu qui réponde au problème initial. Cependant dans les cas pratiques d'utilisation des algorithmes génétiques, il n'est pas possible de savoir simplement si le problème est résolu (le plus court chemin figure-t-il dans ma population ?). On utilise donc des conditions d'arrêt heuristiques basées sur un critère arbitraire.

Le but de cette partie est de construire un algorithme génétique pour rechercher un meilleur chemin d'exploration que celui obtenu par l'algorithme du plus proche voisin.

III.1 Initialisation et évaluation

Une population est représentée par une liste d'individus, chaque individu étant représenté par une liste `[longueur, chemin]` dans lequel

- `chemin` désigne un chemin représenté comme précédemment par une liste d'entiers correspondant aux indices des points d'intérêt dans le tableau des distances produit par la fonction `calculer_distances` ;
- `longueur` est de type `float` et donne la longueur du chemin, en tenant compte de la position de départ du robot.

Q16 Écrire une fonction d'en-tête :

```
def créer_population(m:int, d:list) -> list:
```

qui, étant donnée `d` le tableau des distances entre les points d'intérêt (et la position courante du robot) tel que produit par la fonction `calculer_distances`, crée une population de m individus aléatoires. Elle renvoie une liste d'individus `[longueur, chemin]`.

III.2 Sélection

Q17 Écrire une procédure d'en-tête :

```
def reduire(p:list) -> None:
```

qui réduit une population de moitié en ne conservant que les individus correspondant aux chemins les plus courts. On rappelle que `p` est une liste de couples `[longueur, chemin]`. La procédure `reduire` ne renvoie pas de résultat mais modifie la liste passée en paramètre. *Indication :* on pourra utiliser les fonctions `sort` ou `sorted` décrites en annexe.

III.3 Mutation

Q18 Écrire une procédure d'en-tête :

```
def muter_chemin(c:list) -> None:
```

qui prend en paramètre un chemin et le transforme en inversant aléatoirement deux de ses éléments.

Q19 Écrire une procédure d'en-tête :

```
def muter_population(p:list, proba:float, d:list) -> None:
```

qui prend en paramètre une population dont elle fait muter un certain nombre d'individus. Le paramètre `proba` (compris entre 0 et 1) désigne la probabilité de mutation d'un individu. Le paramètre `d` est la matrice des distances entre points d'intérêt.

III.4 Croisement

Q20 Écrire une fonction d'en-tête :

```
def croiser(c1:list, c2:list) -> list:
```

qui crée un nouveau chemin à partir de deux chemins passés en paramètre. Ce nouveau chemin sera produit en prenant la première moitié du premier chemin suivi de la deuxième moitié du deuxième puis en « normalisant » le chemin ainsi obtenu.

Q21 Écrire une procédure d'en-tête :

```
def nouvelle_génération(p:list, d:list) -> None:
```

qui fait grossir une population en croisant ses membres pour en doubler l'effectif. Pour cela, la procédure fait se reproduire tous les couples d'individus qui se suivent dans la population `(p[i],p[i+1])` et `(p[m-1], p[0])` de façon à produire m nouveaux individus qui s'ajoutent aux m individus de la population de départ.

III.5 Algorithme complet

Q22 Écrire une fonction d'en-tête :

```
def algo_génétique(PI:list, m:int, proba:float, g:int) -> [float, list]:
```

qui prend en paramètre un tableau de points d'intérêts, la taille `m` de la population, la probabilité de mutation `proba` et le nombre de générations `g`. Cette fonction implante un algorithme génétique à l'aide des différentes fonctions écrites jusqu'à présent et renvoie la longueur du plus court chemin d'exploration et le chemin lui-même obtenus au bout de `g` générations.

Q23 Est-il possible avec l'implantation réalisée, qu'une itération de l'algorithme dégrade le résultat : le meilleur chemin obtenu à la génération $n+1$ est plus long que celui de la génération n ?
Dans l'affirmative, comment modifier le programme pour que cette situation ne puisse plus arriver ?

Q24 Quelles autres conditions d'arrêt peut-on imaginer ? Établir un comparatif présentant les avantages et inconvénients de chaque condition d'arrêt envisagée.

Annexe : opérations et fonctions Python disponibles

Fonctions

- `[i for i in range(n)]` renvoie une liste contenant les `n` premiers entiers dans l'ordre croissant : `[i for i in range(5)]` → `[0, 1, 2, 3, 4]` ;
- `random.randrange(a, b)` renvoie un entier aléatoire compris entre `a` et `b-1` inclus (`a` et `b` entiers) ;
- `random.random()` renvoie un nombre flottant tiré aléatoirement dans $[0, 1[$;
- `random.shuffle(u)` permute aléatoirement les éléments de la liste `u` (modifie `u`) ;
- `random.sample(u, n)` renvoie une liste de `n` éléments distincts de la liste `u` choisis aléatoirement, si `n > len(u)`, déclenche l'exception `ValueError` ;
- `x ** .5` calcule la racine carrée du nombre x ;

Opérations sur les listes

- `len(u)` donne le nombre d'éléments de la liste `u` :
  `len([1, 2, 3])` → 3 ; `len([[1,2], [3,4]])` → 2 ;
- `u + v` construit une liste constituée de la concaténation des listes `u` et `v` :
  `[1, 2] + [3, 4, 5]` → `[1, 2, 3, 4, 5]` ;
- `n * u` construit une liste constitué de la liste `u` concaténée `n` fois avec elle-même :
  `3 * [1, 2]` → `[1, 2, 1, 2, 1, 2]` ;
- `e in u` et `e not in u` déterminent si l'objet `e` figure ou pas dans la liste `u`, cette opération a une complexité temporelle en $O($`len(u)`$)$:
  `2 in [1, 2, 3]` → `True` ; `2 not in [1, 2, 3]` → `False` ;
- `u.append(e)` ajoute l'élément `e` à la fin de la liste `u` (similaire à `u = u + [e]`) ;
- `del u[i]` supprime de la liste `u` son élément d'indice `i` ;
- `del u[i:j]` supprime de la liste `u` tous ses éléments dont les indices sont compris dans l'intervalle $[$`i`$,$ `j`$[$;
- `u.remove(e)` supprime de la liste `u` le premier élément qui a pour valeur `e`, déclenche l'exception `ValueError` si `e` ne figure pas dans `u`, cette opération a une complexité temporelle en $\mathcal{O}($`len(u)`$)$;
- `u.insert(i, e)` insère l'élément `e` à la position d'indice `i` dans la liste `u` (en décalant les éléments suivants) ; si `i >= len(u)`, `e` est ajouté en fin de liste ;
- `u[i], u[j] = u[j], u[i]` permute les éléments d'indice `i` et `j` dans la liste `u` ;
- `sorted(u)` renvoie une nouvelle liste contenant les éléments de la liste `u` triés dans l'ordre « naturel » de ses éléments (si les éléments de `u` sont des listes , l'ordre utilisé est l'ordre lexicographique).
- `u.sort()` trie la liste `u` en place, dans l'ordre « naturel » de ses éléments (si les éléments de `u` sont des listes, l'ordre utilisé est l'ordre lexicographique).
- `min(L)` (resp. `max(L)`) renvoie le plus petit (resp. grands) des élements de la liste `L`.

FIN DU SUJET

Corrigé

Q1

```
def generer_PI(n:int, cmax:int):
    L = []
    while len(L) < n:
        x = random.randrange(0, cmax + 1)
        y = random.randrange(0, cmax + 1)
        if [x, y] not in L:
            L.append([x, y])
    return L

#ou avec un compteur
def generer_PI(n:int, cmax:int):
    L = []
    nombre = 0
    while nombre < n:
        x = random.randrange(0, cmax + 1)
        y = random.randrange(0, cmax + 1)
        if [x, y] not in L:
            L.append([x, y])
            nombre +=1
    return L
```

Q2 Il faut avoir les pré-conditions minimales suivantes :

(a) `n`$\geq$`1` et `cmax`$\geq$`1`

(b) Comme la largeur de la zone d'exploration vaut , il y a `cmax+1` absicisses possibles, et autant d'ordonnées possibles (car la zone est carrée). Il y a donc `(cmax+1)**2` points possibles ; il faut donc que `n<=(cmax+1)**2.`

Q3

```
#pour les tests
def position_robot():
    return[12,20]

def distance(P1:list, P2:list) -> float:
    # fonction auxiliaire qui renvoie la distance entre les points P1 et P2
    [x1, y1] = P1
    [x2, y2] = P2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** .5

def ligne(PI,i):
    ligne = []
    for j in range(len(PI)):
        ligne.append(distance(PI[i],PI[j]))
    return ligne

def calculer_distances(PI:list) -> list:
    PI.append(position_robot()) #liste de tous les points d'intérêt y compris le point de départ
    L = []
    for i in range(len(PI)):
        L.append(ligne(PI,i))
    return L

# remarque : on pourrait optimiser le calcul en utilisant le fait que le résultat est une matrice symétrique avec des zéros sur la diagonale, (mais pas très facile avec des listes de listes).
```

Q4 Cette fonction crée une liste `L` des intensités lumineuses de tous les pixels de la photo, puis parcourt cette liste pour alimenter l'histogramme des effectifs des intensités rencontrés, en prenant pour base d'intensité, l'intensité minimale.
Ainsi H contient le nombre de pixels de chaque intensité rencontrée.

Q5

```
70 def selectionner_PI(photo, imin:int, imax:int) -> list:
71     longueur, hauteur = len(photo[0]), len(photo)
72     L = []
73     for y in range(hauteur):
74         for x in range(longueur):
75             if imin <= photo[x, y] and photo[x, y] <= imax:
76                 L.append([x, y])
77     return L
```

Q6 La requête demandée est :
SELECT ex_num FROM explo WHERE ex_deb IS NOT NULL AND ex_fin IS NULL

Q7 Pour l'exploration de "Numero" connu, par exemple, Numero = 42, la requête est :
SELECT pi_num, pi_x, pi_y FROM pi WHERE ex_num = Numero

Q8 La requête demandée est :
SELECT (MAX(pi_x) - MIN(pi_x)) * (MAX(pi_y) - MIN(pi_y)) * 1e-6 FROM pi JOIN explo ON pi.ex_num = explo.ex_num
WHERE ex_fin IS NOT NULL (exploration terminée)
OR ex_deb IS NOT NULL AND ex_fin IS NULL AND pi_arr IS NOT NULL (exploration en cours et PI visité ou en cours de visite)
GROUP BY pi.ex_num

Q9

```
86 def longueur_chemin(chemin:list, d:list) -> float:
87     longueur = d[-1][chemin[0]] #intialisation du calcul
88     for i in range(len(chemin)-1):
89         longueur = longueur + d[chemin[i]][chemin[i+1]]
90     return longueur
91
92 #ou
93 def longueur_cheminB(chemin:list, d:list) -> float:
94     longueur = 0 #intialisation du calcul
95     point_precedent = len(d)-1 #numero du point de depart
96     for point in chemin:
97         longueur = longueur + d[point_precedent][point]
98         point_precedent = point
99     return longueur
```

Q10

```
102 def normaliser_chemin(chemin:list, n:int) -> list:
103     L=[]
104     # on crée la liste des tous les entiers du chemin sans doublon
105     for point in chemin:
106         if point not in L and point < n:
107             L.append(point)
108     for i in range(n): # on ajoute les points manquants
109         if i not in L:
110             L.append(i)
111     return L
```

Q11 Un chemin est une manière d'ordonner les n points, c'est à dire une permutaions de l'ensmble des entiers $[\![0, n-1]\!]$; il y en a donc $n!$.

Q12 Comme $20! \approx 2 \cdot 10^{18}$, en estimant qu'un test de chemin prenne (minoration très grossière !) 10^{-5} seconde, il faudrait environ 771 500 ans pour trouver le bon chemin. Pas très raisonnable, donc. *Remarque :* le seul nombre de chemins à examiner ne suffit pas à répondre à la question ; il faut aussi tenir compte du temps de traitement de chaque chemin, ainsi que du temps de construction de tous les chemins à traiter.

Q13

```
118 def plus_proche_point(chemin,L):
119     '''détermine le point le plus proche à visiter d'un point donné'''
120     indice = 0
121     mini = L[0]
122     for i in range(len(L)):
123         if L[i] < mini and i not in chemin : # point non visité
124             mini = L[i]
125             indice = i
126     return indice
127 
128 def plus_proche_voisin(d:list) -> list:
129     r = len(d) - 1
130     chemin = [r]
131     for _ in range(len(d)-1):
132         r = plus_proche_point(chemin,d[r])
133         chemin. append(r)
134     return chemin
```

Q14 (a) Dans la fonction `calculer_distances`, la boucle `for i` est exécutée `n+1` fois et la boucle de la fonction `ligne` est aussi exécutée `n+1` fois, d'où une complexité temporelle en $O(n^2)$.

(b) Dans la fonction `plus_proche_voisin`, la boucle `for` est exécutée `n` fois.
Dans la sous fonction `plus_proche_point`, la boucle `for i` est exécutée `n+1` fois et la condition `i not in chemin` multiplie le nombre d'étapes par `n`, d'où une complexité temporelle de cette sous-fonction en $O(n^2)$.
Ainsi, la fonction `plus_proche_voisin` a une complexité temporelle en $O(n^3)$.
Au final, l'algorithme des plus proches voisins a une complexité temporelle en $O(n^3)$.

Q15 Avec les points de coordonnées $A = (0,0)$, $B = (0,3000)$ et $C = (0,7000)$, si le robot se trouve initialement en $P = (0,2000)$, il va aller en d'abord en B puis en A puis en C et parcourir $1+3+7 = 11$ mètres alors que le chemin $PABC$ de longueur $2+3+4 = 9$ mètres est plus court.

Q16

```
141 def creer_individu(d):
142     n = len(d) - 1
143     points = [i for i in range(n)] # liste de tous les indices de points
        possibles
144     chemin = random.sample(points, n)
145     longueur = longueur_chemin(chemin, d)
146     return [longueur, chemin]
147 
148 def creer_population(m:int, d:list) -> list:
149     population = []
150     for _ in range(m):
151         individu = creer_individu(d)
152         population.append(individu)
153     return population
```

Q17

```
156 def reduire(p:list):
157     m = len(p)
158     p.sort()
159     del p[m // 2:] #ou (m+1)//2
```

Q18

```
def muter_chemin(c:list):
    n = len(c)
    i, j = random.sample(range(n), 2)
    c[i], c[j] = c[j], c[i]
```

Q19

```
def muter_population(p:list, proba:float, d:list):
    m = len(p)
    for i in range(m):
        if random.random() <= proba:
            l,chemin = p[i]
            muter_chemin(chemin)
            longueur = longueur_chemin(chemin, d)
            p[i] = [longueur, chemin]
```

Q20

```
def croiser(c1:list, c2:list) -> list:
    n=len(c1)
    return normaliser_chemin(c1[:n // 2] + c2[n // 2:], n)
```

Q21

```
def nouvelle_generation(p:list, d:list):
    m = len(p)
    for i in range(m):
        c1, c2 = p[i][1], p[(i + 1) % m][1] # convient aussi pour i = m
    - 1
        chemin = croiser(c1, c2)
        longueur = longueur_chemin(chemin, d)
        p.append([longueur, chemin])
```

Q22

```
def algo_genetique(PI:list, m:int, proba:float, g:int) -> [float, list]:
    # 1. Initialisation & 2. Evaluation
    d = calculer_distances(PI)
    p = creer_population(m, d)
    for _ in range(g):
        # 3. Selection
        reduire(p)
        # 4. Croisement
        nouvelle_generation(p, d)
        # 5. Mutation
        muter_population(p, proba, d)
    print(p)
    # Recherche du chemin le plus court
    return min(p)

    # ou aussi
    #p.sort()
    #return p[0]
```

Q23 Le résultat peut se dégrader car on peut muter un individu réalisant le minimum à un instant donné (cf. Q15 avec PABC qui deviendrait PBAC). Pour éviter ce problème, on peut décider de ne pas muter un individu réalisant le minimum (ce qui oblige à le calculer à chaque itération), ou bien de ne muter un individu que si le mutant est meilleur que l'individu lui-même.

Q24 On peut décider de s'arrêter lorsque :

- On a trouvé la longueur minimale.
  Avantage : on a résolu le problème.
  Inconvénient : il fallait connaître la solution, pas applicable. On peut cependant comparer à la valeur renvoyée par `plus_proche_voisin`.
- Un certain temps s'est écoulé.
  Avantage : il suffit d'un chronomètre.
  Inconvénient : aucune idée sur la précision du résultat.
- Le meilleur chemin stagne sur plusieurs générations.
  Avantage : facile à écrire.
  Inconvénient : on peut attendre longtemps, possibilité de minimum local. Il faut calculer le minimum à chaque étape.
- La population évolue peu.
  Inconvénient : coûteux à vérifier à chaque étape. Pertinence du critère ?