INFORMATIQUE POUR TOUS
Devoir surveillé 1
Corrigé

1.
```
def générer_PI(n:int, cmax:int) -> np.ndarray:
    res = []
    while len(res) < n:
        x,y = random.randrange(0, cmax+1) , random.randrange(0, cmax+1)
        if [x,y] not in res:
            res.append([x,y])
    return np.array(res)
```

2. Il faut qu'il existe n points distincts dans la zone d'exploration, ie $n \leq (cmax+1)^2$.

3.
```
def calculer_distances(PI:np.ndarray) -> np.ndarray:
    n = len(PI)
    res = np.zeros([n+1, n+1])
    for i in range(n+1):
        if i == n:
            xi,yi = position_robot()
        else:
            xi,yi = PI[i]
        for j in range(n+1):
            if j == n:
                xj,yj = position_robot()
            else:
                xj,yj = PI[j]
            res[i][j] = math.sqrt((xi-xj)**2 + (yi-yj)**2)
    return res
```

4. Cette fonction compte le nombre d'occurrences de chaque valeur d'intensité parmi les points de la photo passée en argument.

5.
```
def sélectionner_PI(photo:np.ndarray, imin:int, imax:int) -> np.ndarray:
    res = []
    n,p = photo.shape
    for i in range(n):
        for j in range(p):
            if imin <= photo[i, j] <= imax:
                res.append([i, j])
    return np.array(res)
```

6. Remarque :`IS NULL` n'est pas au programme, on accordera donc les points pour `= NULL`, même si c'est en fait incorrect en SQL (`NULL = NULL` vaut `unknown` et non `true`).

```
SELECT EX_NUM
FROM EXPLO
WHERE EX_DEB IS NOT NULL AND EX_FIN IS NULL
```

7.
```
SELECT PI_NUM, PI_X, PI_Y
FROM PI
WHERE EX_NUM = n
```

où n est le numéro de l'exploration considérée.

8.
```
SELECT EX_NUM, 10**-6*(MAX(PI_X)-MIN(PI_X)) * (MAX(PI_Y)-MIN(PI_Y)) AS SURFACE
FROM EXPLO JOIN PI ON EXPLO.EX_NUM = PI.EX_NUM
```

```
WHERE EX_FIN IS NOT NULL
GROUP BY EX_NUM
```

9. Sous l'hypothèse que les entiers sont encodés en complément à deux sur 64 bits, l'entier le plus grand représentable est $2^{63} - 1 \approx 10^{19}$. La surface maximale stockable est alors de l'ordre de $10^{38} mm^2$, ie $10^{27} m^2$.

10.
```
SELECT IN_NUM, COUNT(*) AS NB_UTILISATIONS, SUM(IT_DUR) AS DUREE
FROM EXPLO
JOIN ANALY ON EXPLO.EX_NUM = ANALY.EX_NUM
JOIN INTYP ON ANALY.TY_NUM = INTYP.TY_NUM
WHERE EX_DEB IS NOT NULL AND EX_FIN IS NULL -- Exploration en cours uniquement
GROUP BY IN_NUM
```

11.
```
def longueur_chemin(chemin:list, d:np.ndarray) -> float:
    res = d[len(d)-1][chemin[0]]
    for k in range(len(chemin) - 1):
        i,j = chemin[k], chemin[k+1]
        res += d[i][j]
    return res
```

12.
```
def normaliser_chemin(chemin:list, n:int) -> list:
    res = []
    for point in chemin:
        if point not in res:
            res.append(point)
    for k in range(n):
        if k not in res:
            res.append(k)
    return res
```

13. Il y a n choix pour le premier point, puis $n-1$ pour le deuxième et ainsi de suite. Ainsi, il y a $n!$ chemins différents.

14. $20! \geq 10^{15}$. En considérant que 10^9 opérations sont processées par seconde, et en imaginant qu'une opération suffise à traiter un chemin (ce qui est plus qu'optimiste), il faudrait au moins 10^6 secondes, ce qui n'est pas un temps de calcul envisageable.

15.
```
def plus_proche(d, i, visités):
    dmin = np.inf
    for j in range(len(d)):
        if j not in visités and d[i, j] < dmin:
            dmin = d[i, j]
            proche = j
    return proche

def plus_proche_voisin(d:np.ndarray) -> list:
    p = len(d)
    res = [p]
    for k in range (len(d)-1):
        p = plus_proche(d, p, res)
        chemin.append(p)
    return res
```

16. Dans la fonction `calculer_distance`, l'appel à `np.zeros` est en $O(n^2)$. Les deux boucles `for` sont aussi en $O(n^2)$, la complexité totale est donc encore en $O(n^2)$.
    Dans la fonction `plus_proche`, le corps de la boucle `for` est en $O(n)$, en raison du test `j not in visités` où `visités` est une liste de taille au plus n. La complexité totale est donc en $O(n^2)$.

Dans la fonction `plus_proche_voisin`, on effectue n appels à `plus_proche`, d'où une complexité totale en $O(n^3)$.

La complexité totale de l'algorithme est donc $O(n^2) + O(n^3) = O(n^3)$.

17. Si le robot part de la position $(0, 4000)$, l'algorithme va sélectionner le chemin

$$(0, 4000), (0, 3000), (0, 0), (0, 7000)$$

de longueur 11000. Ce n'est pas le plus court chemin, puisque

$$(0, 4000), (0, 7000), (0, 3000), (0, 0)$$

est de longueur 10000.

18.
```
def créer_individu(d):
    chemin = list(range(len(d)-1))
    random.shuffle(chemin)
    long = longueur_chemin(chemin, d)
    return [long, chemin]

def créer_population(m:int, d:np.ndarray) -> list:
    res = []
    for k in range(m):
        L.append(créer_individu(d))
    return res
```

19.
```
def réduire(p:list) -> None:
    p.sort()
    del p[len(p)//2:]
```

20.
```
def muter_chemin(c:list) -> None:
    i = random.randint(0, len(c)-1)
    j = random.randint(0, len(c)-2)
    if j>=i:
        j +=1
    c[i], c[j] = c[j], c[i]
```

21.
```
def muter_individu(I, d):
muter_chemin(I[1])
I[0] = longueur_chemin(I[1], d)

def muter_population(p:list, proba:float, d:np.ndarray) -> None:
    for k in range(len(p)):
        if random.random()<proba:
            muter_individu(p[k], d)
```

22.
```
def croiser(c1:list, c2:list) -> list:
    n = len(c1)
    return normaliser(c1[:n//2] + c2[n//2:],n)
```

23.
```
def croiser_individus(i1, i2, d):
    c = croiser(i1[1], i2[1])
    return [longueur_chemin(c, d), c]

def nouvelle_génération(p:list, d:np.ndarray) -> None:
    n = len(p)
    for k in range(n):
        p.append(croiser_individus(p[k], p[(k+1)%n], d))
```

24.
```
def algo_génétique(PI:np.ndarray, m:int, proba:float, g:int) -> float, list:
    d = calculer_distances(PI)
    p = créer_population(m, d)
    for i in range(g):
        réduire(p)
        nouvelle_génération(p, d)
        muter_population(p, proba, d)
    p.sort()
    return p[0]
```

25. Un chemin peut clairement augmenter en distance en mutant, et il est possible lors d'une itération que le meilleur chemin mute. Il est donc possible que le résultat soit localement dégradé.

    Pour éliminer cette possibilité, il suffit d'empêcher que le meilleur chemin soit muté. À l'issue de la fonction `réduire`, le meilleur chemin est en tête de liste, il suffit donc de remplacer dans `muter_population` la ligne `for k in range(len(p)):` par `for k in range(1,len(p)):`.

26. On peut citer :
    - s'arrêter quand la distance du meilleur chemin passe sous un seuil donné en argument. On ne sait cependant pas comment ce seuil se compare à la solution optimale, et la terminaison du programme n'est pas garantie.
    - s'arrêter quand la meilleure solution ne progresse plus significativement. On prend alors le risque de s'arrêter sur une solution médiocre qui n'était la meilleure que localement.
    - s'arrêter quand la meilleure solution et la distance moyenne des chemins de la génération ne progressent plus significativement. La solution retenue aura tendance à être de meilleure qualité qu'avec la condition précédente, mais la distance moyenne n'est pas forcément décroissante et le temps d'exécution est d'autant plus long et difficile à estimer.

    Il est également possible de combiner plusieurs conditions d'arrêt pour obtenir des compromis entre ces approches.