I.A.1.*a* Une première version avec une boucle :

```
def generer_PI(n:int, cmax:int) -> np.ndarray:
    L = [] # liste des points déjà calculés
    while len(L) < n:
        x, y = random.randrange(0, cmax + 1), random.randrange(0, cmax + 1)
        if (x, y) not in L:
            L.append((x, y))
    return np.array(L)
```

Une autre version avec l'une des fonctions donnée en annexe :

```
def generer_PI_bis(n:int, cmax:int) -> np.ndarray:
    zone = [[(i, j) for i in range(cmax + 1)] for j in range(cmax + 1)]
    return random.sample(zone, n)
```

I.A.1.*b* On doit avoir $n \leqslant (1+cmax)^2$.

I.A.2 Il est pratique de commencer par définir une fonction donnant la distance entre dux points.

```
def distance(A:np.ndarray, B:np.ndarray) -> float:
    return math.sqrt((A[0]-B[0]) ** 2 + (A[1]-B[1]) ** 2)

def calculer_distances(PI:np.ndarray) -> np.ndarray:
    n = len(PI)
    d = np.zeros((n + 1, n + 1)) # tableau des distances
    P = position_robot()
    for i in range(n):
        for j in range(i): # tableau symétrique, nul sur la diagonale
            d[i, j] = d[j, i] = distance(PI[i], PI[j])
        d[n, i] = d[i, n] = distance(P, PI[i])
    return d
```

I.B.1 La fonction renvoie le tableau des intensités, de la plus faible rencontrée à la plus forte, avec les occurrences pour chaque intensité (donc éventuellement 0 pour une intensité non rencontrée mais entre les deux bornes).

```
def F1(photo:np.ndarray) -> np.ndarray:
    n = photo.min()                     # n et b sont les valeurs extrêmes.
    b = photo.max()                     # h est un tableau de 0 avec des cases
    h = np.zeros(b - n + 1, np.int64)   # pour représenter les entiers de n à b.
    for p in photo.flat:                # On itère sur les éléments de photo.
        h[p - n] += 1                   # On ajoute 1 à la case correspondant à l'intensité
            rencontrée.
    return h
```

I.B.2

```
def selectionner_PI(photo:np.ndarray, imin:int, imax:int) -> np.ndarray:
    longueur, hauteur = photo.shape
    L = []
    for x in range(longueur):
        for y in range(hauteur):
            if imin <= photo[x, y] <= imax:
                L.append((x, y))
    return np.array(L)
```

I.C.1

```
SELECT ex_num FROM explo WHERE ex_deb IS NOT NULL AND ex_fin IS NULL
```

I.C.2 Pour l'exploration numéro X :

```
SELECT pi_num, pi_x, pi_y FROM pi WHERE ex_num = X
```

I.C.3

```
SELECT (MAX(pi_x) - MIN(pi_x)) * (MAX(pi_y) - MIN(pi_y)) / 1000000
  FROM pi JOIN explo ON pi.ex_num = explo.ex_num
  WHERE pi.ex_fin IS NOT NULL
  GROUP BY pi.ex_num
```

I.C.4 On doit formuler une hypothèse quant à la façon dont sont représentés les entiers.
Si les entiers sont représentés par des entiers non signés sur 32 bits alors il valent au plus $2^{32}-1$ donc la surface maximale est de :

$$(2^{32}-1)^2.10^{-12}\ \text{km}^2$$

soit environ 18,5 millions de km^2.
La superficie de Mars étant d'environ 144,8 millions de km^2 (merci Google!), cela représenterait environ 13% de la surface.

I.C.5

```sql
SELECT in_num, COUNT(*), SUM(it_dur)
  FROM intyp
    JOIN analy ON intyp.ty_num = analy.ty_num
    JOIN explo ON explo.ex_num = analy.ex_num
  WHERE ex_deb IS NOT NULL AND ex_fin IS NULL
  GROUP BY in_num;
```

II.A.1

```python
def longueur_chemin(chemin:list, d:np.ndarray) -> float:
    longueur = 0
    point_precedent = len(d) - 1
    for point in chemin:
        longueur += d[point_precedent, point]
        point_precedent = point
    return longueur
```

II.A.2

```python
def normaliser_chemin(chemin:list, n:int) -> list:
    absent = [True] * n # tableau de booléen pour indiquer si un point a déjà été rencontré
    valide = []         # construction d'un chemin valide
    # d'abord les points sans doublon
    for point in chemin:
        if point < n and absent[point]: # évaluation paresseuse
            valide.append(point)
            absent[point] = False
    # on ajoute ensuite les points manquants
    for point in range(n):
        if absent[point]:
            valide.append(point)
    return valide
```

II.B.1 Il s'agit du nombre de permutations d'un ensemble à n éléments, il y en a n!

II.B.2 Tout d'abord : 20! vaut environ 2.10^{18}.
Sur un ordinateur avec un processeur à 2GHz, il y a au mieux 2.10^9 chemins traités par seconde (en fait BEAUCOUP moins) ce qui donne... 38 ans.
Les rapports de jury (en PC et PSI) demandent une estimation numérique même grossière pour avoir un ordre de grandeur.

II.C.1 On commence par définir une fonction auxiliaire.

```python
def indice_mini(position:int, d:np.ndarray, pas_encore_visite:list) -> int:
    n = len(d) - 1
    mini = -1
    for i in range(n):
        if pas_encore_visite[i] and (mini < 0 or d[position, i] < d[position, mini]):
            mini = i
    return mini

def plus_proche_voisin(d:np.ndarray) -> list:
    n = len(d) - 1
    chemin = []
    pas_encore_visite = [True] * n + [False]
    position =  n
    while len(chemin) < n:
        position = indice_mini(position, d, pas_encore_visite)
        chemin.append(position)
        pas_encore_visite[position] = False
    return chemin
```

II.C.2 La fonction `indice_min` a une complexité en $O(n)$.
La fonction `plus_proche_voisin` a une complexité en $O(n^2)$ (car initialisation en $O(n)$ puis $O(n^2)$).
La fonction `calculer_distances` a une complexité $O(n^2)$ avec la méthode choisie mais si l'on avait utilisé des *not in* alors la complexité aurait été en $O(n^3)$.
Le rapport du jury MP attire l'attention sur la complexité du `not in`.

II.C.3 Il suffit par exemple de partir de $(0,2000)$.
Avec l'algorithme du plus proche voisin, on décrit successivement les points : $(0,2000)$, $(0,3000)$, $(0,0)$ et $(0,7000)$. Le déplacement total mesure donc 11m.
On aurait pu aussi décrire successivement : $(0,2000)$, $(0,0)$, $(0,3000)$ et $(0,7000)$ avec un déplacement total de 9m.
Le rapport PC recommande de faire un dessin pour illustrer la situation.

III.A

```
def creer_population(m:int, d:np.ndarray) -> list:
    '''- paramètres: nombre d'individus à engendrer et tableau des distances (et position du
        robot)
       - renvoie une liste de couples (longueur, chemin)'''
    population = []
    n = len(d) - 1
    points = list(range(n))
    for k in range(m):
        chemin = random.sample(points, n)
        longueur = longueur_chemin(chemin, d)
        population.append((longueur, chemin))
    return population
```

III.B

```
def reduire(p:list) -> None:
    ''' On modifie p en ne gardant que la moitié des plus petites longueurs '''
    m = len(p)
    p.sort() # le tri est effectué avec l'ordre lexicographique (cf. fin du sujet)
    del p[m // 2:] # on supprime la deuxième moitié
```

III.C.1

```
def muter_chemin(c:list) -> None:
    ''' On transforme le chemin c en permutant deux indices aléatoires. '''
    n = len(c)
    i, j = random.sample(range(n), 2)
    c[i], c[j] = c[j], c[i]
```

III.C.2

```
def muter_population(p:list, proba:float, d:np.ndarray) -> None:
    m = len(p)
    for i in range(m):
        if random.random() <= proba:
            chemin = p[i][1]
            muter_chemin(chemin)
            longueur = longueur_chemin(chemin, d)
            p[i] = (longueur, chemin)
```

III.D.1

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

III.D.2

```
def nouvelle_generation(p:list, d:np.ndarray) -> None:
    m = len(p)
    for i in range(m):
        c1, c2 = p[i][1], p[(i + 1)%m][1] # le modulo % sert à gérer le cas où i=m-1
        chemin = croiser(c1, c2)
        longueur = longueur_chemin(chemin, d)
        p.append((longueur, chemin))
```

III.E.1

```python
def algo_genetique(PI:np.ndarray, m:int, proba:float, g:int) -> (float, list):
    ''' - paramètres: tableau de points d'intérêts, taille de la population,
                      probabilité de mutation, nombre de générations
        - renvoie la longueur du plus court chemin et le chemin lui-même au bout de g
            générations '''

    # initialisation et évaluation
    d = calculer_distances(PI)
    p = creer_population(m, d)
    for k in range(g):
        reduire(p) # sélection
        nouvelle_generation(p, d) # croisement
        muter_population(p, proba, d) # mutation

    # recherche du chemin le plus court
    indice_min = 0
    for i in range(len(p)):
        if p[i][0] < p[indice_min][0]:
            indice_min = i

    return p[indice_min] # on pouvait aussi utiliser sort et renvoyer p[0]
```

III.E.2 Il se peut que l'on ait muté un chemin optimal.
On peut palier ce problème en ne mutant un individu que si le chemin obtenu est meilleur.

III.E.3 Des idées pour décider quand s'arrêter.

- Si l'on trouve une longueur meilleure que celle donnée par l'algorithme glouton du plus proche voisin.
- Au bout d'un certain temps écoulé... mais l'on n'a aucune idée de la précision du résultat.
- S'il y a stagnation pendant un certain nombre de générations mais il faut calculer le minimum à chaque étape et cela peut durer longtemps.
- Si la population évolue peu mais il faut vérifier à chaque étape.
- Si le meilleur chemin est peu «éloigné» du pire (*i.e.* la population est relativement homogène).