```
# I.A.1)  concaténation : [1, 2, 3, 4, 5, 6]

# I.A.2)  multiplication : [1, 2, 3, 1, 2, 3]

# I.B.1)
def smul(n,liste):
    liste2=[]
    for i in liste:
       liste2.append(i*n)
    return liste2

# I.B.2)
def vsom(liste1,liste2):
    liste3=[]
    for i in range(len(liste1)):
       liste3.append(liste1[i]+liste2[i])
    return liste3

# I.B.3)
def vdif(liste1,liste2):
    liste3=vsom(liste1,smul(-1,liste2))
    return np.array(liste3)
```

```
## II.A1
def generer_PI(n, cmax):
    """n:int, cmax:int -> tableau des positions"""
    L=np.zeros((n,2))
    for i in range(n):
        x=random.randrange(0,cmax+1)
        y=random.randrange(0,cmax+1)
        if x!=L[i][0] or y!=L[i][1]: # 2 à 2 distincts
            L[i][0],L[i][1]=x,y
    return L

## A2
def calculer_distances(PI) :
    """PI:np.ndarray -> tableau de dim (n+1)(n+1) distance entre i et j"""
    posR = np.array([position_robot ()])
    piR = np.concatenate((PI, posR))
    n = len(piR)
    dist = np.zeros((n, n))
    for i in range(n):
          for j in range( i ): # Matrice symétrique
              dist [i,j] = sqrt((piR[i,0] - piR[j,0])**2 + (piR[i,1] - piR[j,1])**2)
              dist [ j , i ] = dist [ i , j ]
    return dist

## II.B1
def gris(photo):
    im1=photo.copy()
    for ligne in range(len(photo)):
        for col in range(len(photo[1])):
            moy=int((im1[ligne, col][0]+im1[ligne, col][1]+im1[ligne, col][2])/3)
            im1[ligne, col][0]=moy
            im1[ligne, col][1]=moy
            im1[ligne, col][2]=moy
    return im1

## II.B2
```

h a la taille du nombre de couleurs différentes que l'on peut voir dans la photo.

On compte combien de fois apparait l'intensité et **on renvoie un histogramme des couleurs de la photo**, c'est à dire le nombre de pixels de chaque intensité comprise entre l'intensité minimale `n` et maximale `b` de la photo considérée.

```
## B3
def selectionner_PI(photo, imin, imax):
    """ photo:np.ndarray, imin:int, imax:int-> tableau à deux dimensions contenant les
coordonnées des points dont l'intensité sur la photographie est comprise entre imin et
imax."""
    taille=photo.shape
    L=[]
    for i in range(taille[0]):
        for j in range(taille[1]):
            if imin<=photo[i][j]<=imax:
                 L.append([i,j])
    return np.array(L)
```

```
##II.C Bdd
#1
"""SELECT ex_num FROM explo WHERE ex_deb IS NOT NULL AND ex_fin IS NULL"""
#2
"""SELECT PI_NUM,PI_X, PI_Y FROM  PI WHERE EX_NUM='numéro connu' """

#3
"""SELECT EX_NUM, (MAX(pi_x) - MIN(pi_x)) * (MAX(pi_y) - MIN(pi_y))*1e-6 FROM explo
JOIN pi ON (ex_num) WHERE ex_fin IS NOT NULL GROUP BY ex_num """

#4
"""SELECT IN_NOM, COUNT(IN_DER), SUM(IT_DUR)  FROM  EXPLO JOIN ANALY using (EX_NUM)
JOIN INTYP using(TY_NUM) WHERE ex_deb IS NOT NULL AND ex_fin IS NULL GROUP BY IN_NOM
"""
```

```
## III.A .1
def longueur_chemin(chemin, d) :
    """ chemin:list, d:np.ndarray renvoie la distance que doit effectuer le robot"""
    longueur = d[-1, chemin[0]]
    for k in range(len(chemin) - 1):
        longueur += d[chemin[k], chemin[k+1]]
    return longueur

## III.A .2
def normaliser_chemin(chemin, n) :
    """ chemin:list, n:int renvoie  """
    L=[]
    for i in range (len(chemin)) :
        if chemin[i] < n :
            k = len(L)
            while k>0 and L[k-1]!= chemin[i] :
                k-=1
            if k==0:
                L.append(chemin[i])
         else :
            if chemin[i] not in T :
                T.append(chemin[i])
    T.sort()
    return L+T
#autre version … bcp plus courte !!
def normaliser_chemin(chemin, n):
    norme = []
    for p in chemin:
        if 0 <= p < n and p not in norme:
            norme.append(p)
    for i in range(n):
        if i not in norme:
            norme.append(i)
    return norme
```

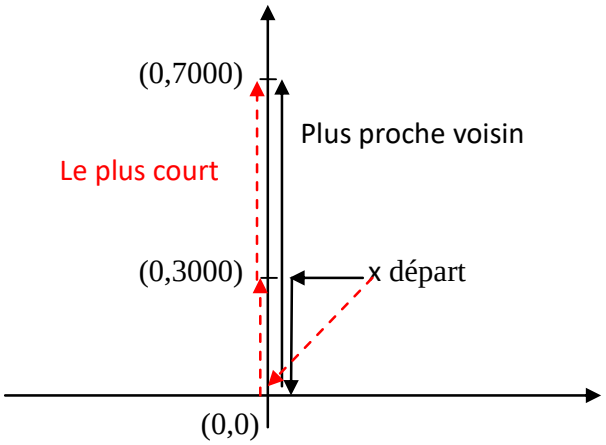
```
##III.B1
# n! :  Il y a n choix possibles pour le premier point du chemin, n - 1 pour le
deuxième, etc. Au total nous aurons donc n! possibilités.

#B2
#pour 20 points d'intérêts l'algorithme représenterait 2432902008176640000 coups  ! et
même si ces opérations ne prenaient que quelques nanosecondes (et ce n'est sans doute
pas le cas) il faudrait 2⁹ secondes, plus de 80 ans, on comprend bien sur que c'est
impossible à réaliser pour le robot.


##III.C1
def plus_proche_voisin(d):
    """d=tableau des distances -> chemin d'exploration (plus proche voisin)  """
    n = len(d) - 1
    dMax = d.max()
    res = [n]
    for p in range(n):
        dMin = dMax
        dist = d[res[-1]]
        for j in range(n):
            if j not in res:
                if dist[j] <= dMin:
                    dMin = dist[j]
                    jMin = j
        res.append(jMin)
    return res[1:]
```

**III.C2 Complexité du plus proche voisin :**

En appellant n la taille de d (donc le nombre de point) `calculer_distances` est en $O(n^2)$ 2 boucles imbriquées.
`plus_proche_voisin` est en $O(n^3)$ : boucle sur p (n) * boucle sur j (n) * le test `j not in res` en $O(n)$
Donc si on appelle succéssivement les deux fonctions la somme des 2 est en $O(n^3)$




**III.C3**

```
##IV A1
def creer_population(m, d) :
    L=[]
    n=len(d)
    for i in range (m):
        chemin = random.sample(range(n), n)
        L.append((longueur_chemin(chemin, d), chemin))
    return L


##IV B1
def reduire(p):
    p.sort()
```

```
    m = len(p)
    m = math.ceil(m/2)
    del p[m:] # On supprime de la liste p initiale les éléments les plus longs à partir de la moitié de la liste : on ne conserve que les individus correspondant aux chemins les plus courts.
La fonction réduire ne renvoie pas de résultat mais modifie la liste passée en paramètre.
```

```
##IV.C1
def muter_chemin(c):
    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, proba, d):
    n = len(p)
    for i in range(n):
         if random.random() <= proba:
              longueur, chemin = p[i]
              muter_chemin(chemin)
              p[i] = longueur_chemin(chemin, d), chemin

## III.D.1 Croisement
def croiser(c1, c2):
    n = len(c1)
    return normaliser_chemin(c1 [:n // 2] + c2[n // 2:], n)

##D.2. Nouvelle génération
def nouvelle_generation(p, d):
    m = len(p) # m individu
    for i in range(m):
        couple1 = p[i][1]
        couple2 = p[(i+1) % m][1] # Permet de récupérer le terme suivant i mais aussi de croiser le dernier et le premier élément lorsque i vaut la longueur de la liste-1
        c = croiser(couple1, couple2)
        p.append((longueur_chemin(c, d), c))

#III.E.1. Algorithme complet :
def algo_genetique(PI, m, proba, g):
    d = calculer_distances(PI)
    p = creer_population(m, d)
    for i in range(g):
        reduire(p)
        nouvelle_generation(p, d)
        muter_population(p, proba, d)
    return min(p)
```

**##III E2**

Il est possible que la fonction `muter_population` fasse muter le meilleur individu vers un chemin plus long. De ce fait, rien n'assure avec l'implantation proposée que la génération n+1 ne soit pas plus mauvaise que la génération n.

Pour éviter cela, il suffit de s'assurer de ne pas muter le meilleur individu. n pourrait alors comparer la population muter avec la population précédente. Si la longueur du chemin est plus grande alors on récupère la population précédente.