Corrigé Centrale 2017

I.A.1)

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

I.A.2)

```
def calculer_distances(PI) :
    rep = []
    for i in range (len(PI)) :
        x,y = PI[i]
        sl = []
        for j in range (len(PI)) :
            z,t = PI[j]
            d=((x-z)**2+(y-t)**2)**0.5
            sl.append(d)
        z,t = position_robot()
        d = ((x-z)**2+(y-t)**2)**0.5
        sl.append(d)
        rep.append(sl)
    sl = []
    for i in range(len(PI)) :
        sl.append(rep[i][-1])
    sl.append(0)
    rep.append(sl)
    return rep
```

Pour fonctionner, il faut que $n \leq cmax^2$.

I.B.1) La fonction renvoie un tableau du nombre d'occurrences de chaque intensité de pixels possible.

I.B.2)

```
def sélectionner(photo,imin,imax) :
    rep = []
    for i in range (len(photo)) :
        for j in range (len(photo[i])) :
            var = photo[i][j]
            if var <= imax and var >= imin :
                rep.append((i,j))
    return rep
```

I.C.1)

```
SELECT EX_NUM
FROM EXPLO
WHERE EX_FIN = NULL
AND EX_DEB <> NULL
```

I.C.2)

```
SELECT PI_X, PI_Y
FROM PI
WHERE EX_NUM = n
```

I.C.3)

```
SELECT (MAX(PI_X) - MIN(PI_X)) * (MAX(PI_Y) - MIN(PI_Y))/(1000*1000)
FROM PI JOIN EXPLO ON EXPLO.EX_NUM = PI.EX_NUM
WHERE EX_FIN <> NULL
GROUP BY EX_NUM
```

I.C.4) Les entiers Pythons n'ayant pas de limite, il n'y a donc pas de limite à la surface pouvant être stocké.

I.C.5)

```
SELECT IN_NOM, COUNT(*), SUM(IT_DUR)
FROM EXPLO
JOIN ANALY ON EXPLO.EX_NUM = ANALY.EX_NUM
JOIN INTYP ON INTYP.TY_NUM = ANALY.TY_NUM
JOIN INSTR ON INSTR.IN_NUM = INTYP.IN_NUM
WHERE EX_FIN = NULL AND EX_DEB <> NULL
GROUP BY IN_NUM
```

II.A.1)

```
def longueur_chemin(chemin,d) :
    s = d[chemin[0]][-1]
    for i in range (len(chemin) - 1):
        p0 = chemin[i]
        p1 = chemin[i+1]
        s += d[p0][p1]
    return s
```

II.A.2)

```
def normaliser_chemin(chemin, n) :
    vu = [False for i in range (n)]
    supp = []
    for i in range (len(chemin)) :
        if chemin[i] < n :
            if vu[chemin[i]] :
                supp.append(i)
            else :
                vu[chemin[i]] = True
        else :
            supp.append(i)
    for j in range (len(supp)-1,-1,-1) :
        chemin.pop(supp[j])
    for i in range (n) :
        if not vu[i] :
            chemin.append(i)
    return chemin
```

II.B.1) On a n choix pour le premier point, puis (n-1) choix pour le deuxième, puis (n-3) choix, c'est à dire n! chemins différents.

II.B.2) 20! est de l'ordre de 10^{18} (la calculatrice était autorisée sur cette épreuve).
C'est beaucoup trop pour être calculé par un ordinateur qui a une fréquence de calcul de 10^9 opérations par seconde.

II.C.1)

```python
def plus_proche_voisin(d) :
    chemin = [(len(d)-1)]
    vu = [False for n in range (len(d)-1)]
    for j in range (len(d)-1) :
        pos = chemin[-1]
        min = float("inf")
        for i in range(len(d)-1) :
            if not vu[i] and d[pos][i] < min :
                min = d[pos][i]
                i_min = i
        chemin.append(i_min)
        vu[i_min] = True
    return chemin
```

II.C.2)
La complexité est en $O(n^2)$, car il y a deux boucles imbriquées en la taille du nombre de points à visiter et que les opérations à l'intérieur sont en O(1).

II.C.3) En partant de (0,2000), on part vers le point (0,3000), puis on fait demi-tour vers (0,0) avant de partir vers (0,7000) en appliquant l'algorithme des plus proche voisin.
Or, partir vers le point (0,0) aurait été plus été efficace, et une distance de -2000 aurait été parcouru.

III.A)

```python
def créer_population(m,d) :
    rep = []
    for i in range (m) :
        chemin = [len(d) - 1]
        point = [i for i in range (len(d)-1)]
        while point != []
            suiv = point.pop(random.randrange(0,len(point))
            chemin.append(suiv)
        dist = longueur_chemin(chemin,d)
        rep.append(dist,chemin)
    return rep
```

III.B)

```python
def réduire(p) :
# Il faut regarder les fonctions fournis en annexe
    p.sort()
    for i in range (len(p)//2) :
        p.pop()
```

III.C)

```python
def muter_chemin(c) :
    i = random.randrange(0,len(c))
    j = random.randrange(0,len(c) - 1)
    if j >= i :
        j += 1
    c[i], c[j] = c[j], c[i]
```

```python
def muter_population(p,proba,d) :
    for i in range (len(p)) :
        d,chem = p[i]
        if random.random() < proba :
            muter_chemin(chem)
            dist = longueur_chemin(chem,d)
            p[i] = (dist,chem)
```

III.D)

```python
def croiser(c1,c2) :
    c = c1[:len(c1//2)] + c2[len(c2)//2:]
    return normaliser_chemin(c,len(c))
```

```python
def nouvelle_generation(p,d) :
    # Boucle fixe malgré append
    # pour un for
    for i in range (len(p)) :
        # Le premier va nous donner
        # p[m-1] : pas encore d'ajout
        d1,c1 = p[-1+i]
        d2,c2 = p[i]
        c = croiser_chemin(c1,c2)
        dis = longueur_chemin(c,d)
        p.append(dis,c)
```

III.E)

```python
def algo_génétique(PI,m,proba,g) :) :
    d = calculer_distance(PI)
    init = créer_population(m,d)
    for i in range(2,g+1) :
        # Début boucle : génération i-1
        réduire(p)
        nouvelle_generation(p,d)
        muter_population(p,proba,d)
        # Fin boucle : génération i
    return min(p)
```

2) Une itération peut dégrader le résultat en mutant le meilleur chemin.
Dans `muter_population`, on change les bornes du range par (1,len(p)) (population triée à ce moment)

3) Arrêt lorsque plusieurs générations n'améliore pas le résultat -> plus rapide, mais risque d'extremum local qui bloque l'amélioration. Arrêt forcément avec la modification question 2).
Arrêt lorsque beaucoup de chemin partage la longueur minimale -> demande de vérifier si plusieurs chemin partage la même longueur (ne change pas la complexité asymptotique cependant).
Arrêt si le meilleur est mieux au moins que le chemin naïf.