Q1–
```
from math import sqrt

def distance(i, j):
    xi, yi = coords_x[i], coords_y[i]
    xj, yj = coords_x[j], coords_y[j]
    return sqrt((xj-xi)**2 + (yj-y_i)**2)
```

Q2–
```
def plus_proche():
    n = len(coords_x)
    imin, jmin = 0, 1
    for i in range(n):
        for j in range(i+1, n):
            if distance(i, j) < distance(imin, jmin):
                imin, jmin = i, j
    return imin, jmin
```

Q3– La fonction `distance` a une complexité en $O(1)$.

Le nombre d'itérations est $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = O(n^2)$ et chaque itération a donc un nombre d'opérations en $O(1)$.

Donc la complexité de `plus_proche` est $O(n^2)$.

Q4– Cette fonction n'a pas de `return` donc elle renvoie `None`.

Cette fonction trie la liste d'entiers donnée en argument par ordre croissant.

En effet, montrons par récurrence qu'à la fin du i-ème passage dans la boucle for, les $i+1$ premiers éléments sont triés par ordre croissant (et sont les mêmes éléments qu'initialement).

C'est vrai au départ (le premier élément de la liste est un élément trié).

En supposant que les i premiers éléments sont triés par ordre croissant à la fin de la $(i-1)$-ème itération, l'itération suivante considère l'élément en position i et l'intervertit avec ses voisins de gauche jusqu'à en rencontrer un plus petit ou arriver tout à gauche. Alors, par hypothèse de récurrence, tous les éléments à la gauche de l'élément placé seront inférieurs et leur ordre n'a pas changé, et tous les éléments à sa droite sont supérieurs et leur ordre relatif est inchangé. Ainsi, les $i+1$ premiers éléments sont désormais triés.

Q5– Chaque passage dans la boucle while réalise un nombre borné d'opérations, et on effectue au plus $i+1$ passages dans cette boucle (car `pos` décroit à chaque passage).

Comme la boucle for comporte n itérations et puisque $\sum_{i=0}^{n-1}(i+1) = O(n^2)$, la complexité est $O(n^2)$.

Q6– On remplace `liste[pos] < liste[pos-1]` par :
```
coords_x[liste[pos]] < coords_x[liste[pos-1]]
```

Q7– Le tri fusion a une complexité, dans le pire des cas, en $O(n\ln(n))$.

Q8–
```
def sous_cluster(cl, x_min, x_max):
    liste_x = []
    liste_y = []
    for p in cl[0]:
        if x_min <= coords_x[p] <= x_max:
            liste_x.append(p)
    for p in cl[1]:
        if x_min <= coords_x[p] <= x_max:
            liste_y.append(p)
    return [liste_x, liste_y]
```

Les corps de boucles ont une complexité en $O(1)$ donc cette fonction est bien de complexité linéaire en la taille du cluster.

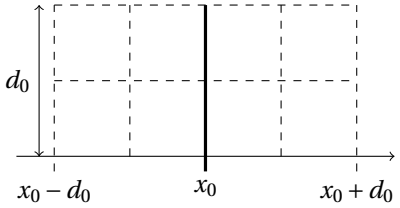
Q9–
```
def mediane(cl):
    n = len(cl[0])
    return coords_x[cl[0][n//2]]
```

Q10–
```
def gauche(cl):
    x_min = coords[cl[0][0]]
    x_max = mediane(cl)
    return sous_cluster(cl, x_min, x_max)
```

Q11– Notons $M_1(x_1, y_1)$ et $M_2(x_2, y_2)$.
Si M_1 est à gauche de la droite d'équation $x = x_0 - d_0$ alors $x_2 - x_1 > d_0$ et *a fortiori* $d(M_1, M_2) > d_0$.
Dans ce cas les points M_1 et M_2 ne sont pas les points les plus proches.
De même, si M_2 est à droite de la droite d'équation $x = x_0 + d_0$ alors $x_2 - x_1 > d_0$ et *a fortiori* $d(M_1, M_2) > d_0$.
Dans ce cas également les points M_1 et M_2 ne sont pas les points les plus proches.
Donc on peut se contenter de chercher les points M_1 et M_2 de l'étape 5 dans l'ensemble des points dont l'abscisse appartient à $I_0 = [x_0 - d_0, x_0 + d_0]$.

Q12–
```
def bande_centrale(cl, d0):
    x = mediane(cl)
    return sous_cluster(cl, x-d0, x+d0)
```

Q13– Considérons, comme sur le dessin ci-dessous, un rectangle situé entre les droites d'équations $x = x_0 - d_0$ et $x = x_0 + d_0$ et de hauteur d_0. On subdivise alors ce rectangle en 8 carrés de côté d_0. Ces carrés ont une diagonale de longueur $\sqrt{(d_0/2)^2 + (d_0/2)^2} = d_0/\sqrt{2}$ (donc inférieure à d_0) et sont situés d'un même côté de la droite d'équation $x = x_0$ donc aucun d'entre eux ne peut contenir deux points de l'ensemble (par définition de d_0).



Avec les notations de l'énoncé, M_1 et M_2 sont à une distance inférieure à d_0 donc, si l'on considère un rectangle comme celui qui précède avec M_1 sur la ligne du bas, il y a alors au plus 7 autres points (dont M_2) dans ce rectangle donc il y a au plus 6 points intercalés (en ordonnée) entre M_1 et M_2.

Q14–
```
def fusion(cl, d0):
    n = len(cl[0])
    dmin = d0
    for i in range(n):
        for j in range(i, i+8):
            if j < n and distance(cl[1][i], cl[1][j]) < dmin:
                dmin = distance(cl[1][i], cl[1][j])
    return dmin
```

La boucle for à l'intérieur a une complexité en $O(1)$.
La boucle for d'indice `i` a n itérations.
Donc la complexité est en $O(n)$ donc linéaire en la taille du cluster.

Q15–
```
def distance_minimale(cl):
    n = len(cl[0])
    if n == 2:
        return distance(cl[0][0], cl[0][1])
    if n == 3:
        d0 = distance(cl[0][0], cl[0][1])
        if distance(cl[0][0], cl[0][2]) < d0:
            d0 = distance(cl[0][0], cl[0][2])
        if distance(cl[0][1], cl[0][2]) < d0:
            d0 = distance(cl[0][1], cl[0][2])
        return d0
    d0 = distance_minimale(gauche(cl))
    d1 = distance_minimale(droite(cl))
    if d1 < d0:
        d0 = d1
    bande = bande_centrale(cl, d0)
    return fusion(bande, d0)
```

Q16– Si $n > 3$ alors il y a deux tests puis des appels à `gauche` et `droite` (en $O(n)$) et deux appels à `distance_minimale` sur des «moitiés de cluster» donc $2C(\lfloor n/2 \rfloor)$.
Ensuite on utilise `bande_centrale` et `fusion` qui ont une complexité en $O(n)$.
On a donc bien : $C(n) = 2C(\lfloor n/2 \rfloor) + O(n)$.