DM 1 – Nuage de points

Amar AHMANE
MP2I.

1 – Approche exhaustive naïve

Question 1 Définition de la fonction `distance`.

```c
/*
Input:
        p1: parameter of type struct point containing the coordinates of a point in an eucledian sapce
        p2: parameter of type struct point containing the coordinates of a point in an eucledian sapce
Output: eucledian distance between these two points.
*/
double distance(struct point p1, struct point p2){
        return sqrt((p2.x-p1.x)*(p2.x-p1.x)+(p2.y-p1.y)*(p2.y-p1.y));
}
```

Question 2 Définition de la fonction `plus_proches`.

```c
/*
        Input:
        Output: Void. Changes the values of p and q setting them to the indexes of the two closest points in the cloud.
*/
void plus_proches(struct nuage* n, int* p, int* q){
        if(n == NULL || p == NULL || q == NULL){
                return;
        }
        assert(n->taille > 1);
        double ref = distance(n->P[0], n->P[1]);
        *p = 0, *q = 0;
        for(int i = 0; i < n->taille; i++){
                for(int j = 0; j < n->taille; j++){
                        if(distance(n->P[i], n->P[j]) < ref){
                                *p = i, *q = j;
                        }
                }
        }
}
```

Question 3 En supposant que les opérations s'effectuant dans la deuxième boucle sont faites en un temps $\Theta(1)$, alors la complexité est en $\sum_{i=1}^{n}\sum_{j=1}^{n}\Theta(1) = \Theta(n^2)$.

2 – Méthode sophistiquée

Question 4 Cette fonction trie le tableau `tab`. En effet, si l'on choisit la phrase suivante comme invariant, pour une itération i donnée :

« le tableau `tab[:i]` est trié »

On utilise ici la notation python pour le saucissonnage de `tab`.

i) Il est clair que lorsque $i = 0$, `tab[:0]` est trié puisqu'il est vide.

ii) On suppose que pour un i donné (une itération donnée), `tab[:i]` est trié. Ainsi, à l'itération suivante, le $i$ème élément sera placé dans `tab[:i+1]` de sorte que tous les élements à droite soient plus grands et que tous les éléments à gauche soient plus petits : en effet, ceci est contrôlé par la condition `tab[pos]<tab[pos - 1]`. Le tableau `tab[:i]` étant déjà trié par hypothèse de récurrence, le tableau `tab[:i + 1]` l'est aussi.

iii) par le principe de récurrence, la phrase déclarée plus haut est bien un invariant de boucle. Ceci prouve la correction de l'algorithme.

Question 5 En supposant que les opérations faites à l'intérieur de la boucle `while` soient effectuées en temps constant, on a alors une complexité en $\sum_{i=0}^{n-1}\sum_{j=1}^{i}\Theta(1) = \sum_{i=1}^{n-1}\Theta(j) = \Theta\left(\frac{n(n-1)}{2}\right) = \Theta(n^2)$.

Question 6 On programme la fonction `tri_cluster` qui comporte les modifications voulues.

```
1                       void tri_cluster(struct nuage* n){
2                               int pos;
3                               for(int i = 0; i < n->taille; i++){
4                                       pos = i;
5                                       while(pos > 0 && n->P[pos].x < n
                                            ->P[pos - 1].x){
6                                               struct point tmp = n->P[
                                                    pos];
7                                               n->P[pos] = n->P[pos -
                                                    1];
8                                               n->P[pos - 1] = tmp;
9                                               pos = pos - 1;
10                                      }
11                              }
12                      }
```

Question 7 L'algorithme de tri par tas est un algorithme de tri de complexité $\mathcal{O}(n \log n)$ au pire des cas.

Question 8 On programme la fonction `sous_cluster`.

```
1                       /*
2                               Input:
3                                       c: parametre de type pointeur sur
                                             struct cluster; cluster dont
                                            on veut former un sous-cluster
4                                       min: parametre de type double;
                                            abscisse minimale
5                                       max: parametre de type double;
                                            abscisse maximale
6                               Output: Sous-cluster du cluster c dont
                                    les abscisses sont comprises entre min
                                     et max.
7                       **/
8                       struct cluster* sous_cluster(struct cluster* c,
                            double min, double max){
9                               assert(!(c == NULL))
10                              struct cluster* result;
11                              struct nuage* N;
12                              int a, b;
13                              for(int i = 1; i < c->taille; i++){
14                                      if(c->N->P[c->abs[i]] >= min){
15                                              a = i;
16                                      }
17                              }
18                              for(int i = c->taille - 1; i >= 0; i--){
19                                      if(c->N->P[c->abs[i]] <= max){
20                                              b = i;
21                                      }
22                              }
23                              result = (struct cluster*) malloc((b - a
                                    + 1)*sizeof(struct cluster));
```

```
24                      N = (struct nuage*) malloc((b - a + 1)*
                            sizeof(struct nuage));
25                      result->abs = (int*) malloc((b - a +1)*
                            sizeof(int));
26                      result->ord = (int*) malloc((b - a +1)*
                            sizeof(int));
27                      N->taille = b - a + 1;
28                      result->taille = N->taille;
29                      for(int i = 0; i + a <= b; i++){
30                              result->abs[i] = c->abs[i + a];
31                              result->ord[i] = c->abs[i + a];
32                              N->P[i] = c->N->P[result->abs[i
                                    ]];
33                      }
34                      result->N = N;
35                      return result;
36              }
```

Question 9 On programme la fonction `mediane`.

```
1               /*
2                       Input:
3                               c: parametre de type pointeur sur
                                     struct cluster; cluster dont
                                    on veut recuperer la mediane
4                       Output: mediane du cluster fourni en
                            parametre
5               **/
6               double mediane(struct cluster* c){
7                       assert(!(c == NULL));
8                       return c->N->P[c->abs[c->taille/2]].x;
9               }
```

Question 10 On programme la fonction `gauche`.

```
1               /*
2                       Input:
3                               c: parametre de type pointeur sur
                                     struct cluster; cluster dont
                                    on veut recuperer la partie
                                    gauche
4                       Output: Partie gauche du cluster fourni
                            en parametre
5               **/
6               struct cluster* gauche(struct cluster *c){
7                       return sous_cluster(c, c->N->P[c->abs
                            [0]].x, mediane(c));
8               }
```

Question 11 Si l'on choisit un premier point en dehors de l'intervalle $[x_0 - \delta, x_0 + \delta]$, la distance entre celui-ci et un point quelconque de l'autre moitié du cluster sera forcément strictement supérieure à δ ; on se rend alors compte que si on choisit deux points de deux clusters à une distance δ l'un de l'autre, on les trouvera forcément dans $[x_0 - \delta, x_0 + \delta]$.

Question 12 On programme la fonction `bande_centrale`.

```
1               /*
2                       Input:
3                               c: parametre de type pointeur sur
                                     struct cluster; cluster dont
                                    on veut recuperer un sous-
                                    clusters dont les points sont
                                    a une distance au plus d0 de
                                    la mediane.
4                               d0: parametre de type double;
                                    distance maximale de la
                                    mediane.
```

```
5                                   Output: Bande centrale de largeur 2*d0.
6                           **/
7                           struct cluster* bande_centrale(struct cluster* c,
                                double d0){
8                               return sous_cluster(c, mediane(c) - d0,
                                    mediane(c) + d0);
9                           }
```

Question 13 Voir le fichier `part2.c`.

Question 14 Voir le fichier `part2.c`.

Question 15 Lors du premier appel de la fonction, on effectue plusieurs opérations :

i) l'appel récursif de cette même fonction, deux fois pour un cluster de taille deux fois moindre (donc $n/2$).

ii) un appel à la fonction fusion avec une entrée de taille n.

En notant $C(n)$ le nombre d'opérations effectuées pour une entrée de taille n, on a bien $C(n) = 2C(n/2) + \mathcal{O}(n)$ (`fusion` ayant une complexité linéaire).

Question 16 On suppose qu'il existe $k \in \mathbb{N}$ tel que $n = 2^k$, on pourra ainsi note $C_k = C(2^k)$. D'où

$$C_k = 2C_{k-1} + \mathcal{O}(n)$$

Par récurrence, on obtient que

$$\begin{aligned} C_k &= 2^k C_0 + \sum_{i=0}^{k-1} 2^i \mathcal{O}(2^{k-i}) \\ &= 2^k C_0 + \sum_{i=0}^{k-1} \mathcal{O}(2^k) \\ &= 2^k C_0 + \mathcal{O}(k2^k) \end{aligned}$$

Finalement

$$C(n) = nC_0 + \mathcal{O}(n \log_2(n)$$

Or, C_0 est une constante, donc $C(n) = \mathcal{O}(n \log n)$.