Projets utilisant la méthode « Diviser pour régner »

Projet 1 : Rotation d'une image d'un quart de tour

L'image *buffon.png* de dimensions 512 × 512 pixels a été tournée d'un quart de tour.

On s'intéresse ici à deux manières différentes d'afficher l'image dans son orientation originale.
On fournit pour cela, dans le fichier *rotation.py*, une classe Main qui affiche l'image dans une fenêtre pygame, et dont la méthode rotation est à compléter.
Dans la méthode rotation, le code est donné à titre d'exemple d'utilisation des méthodes get_at et set_at.
On consultera si nécessaire la documentation de la classe Surface de pygame pour obtenir des informations sur ces méthodes.

1.a) Modifier le code de la méthode rotation pour que l'image soit affichée dans son orientation originale.
Pour cela, on stockera dans un tableau l'ensemble des valeurs des pixels, qui seront ensuite affichés dans le bon ordre.

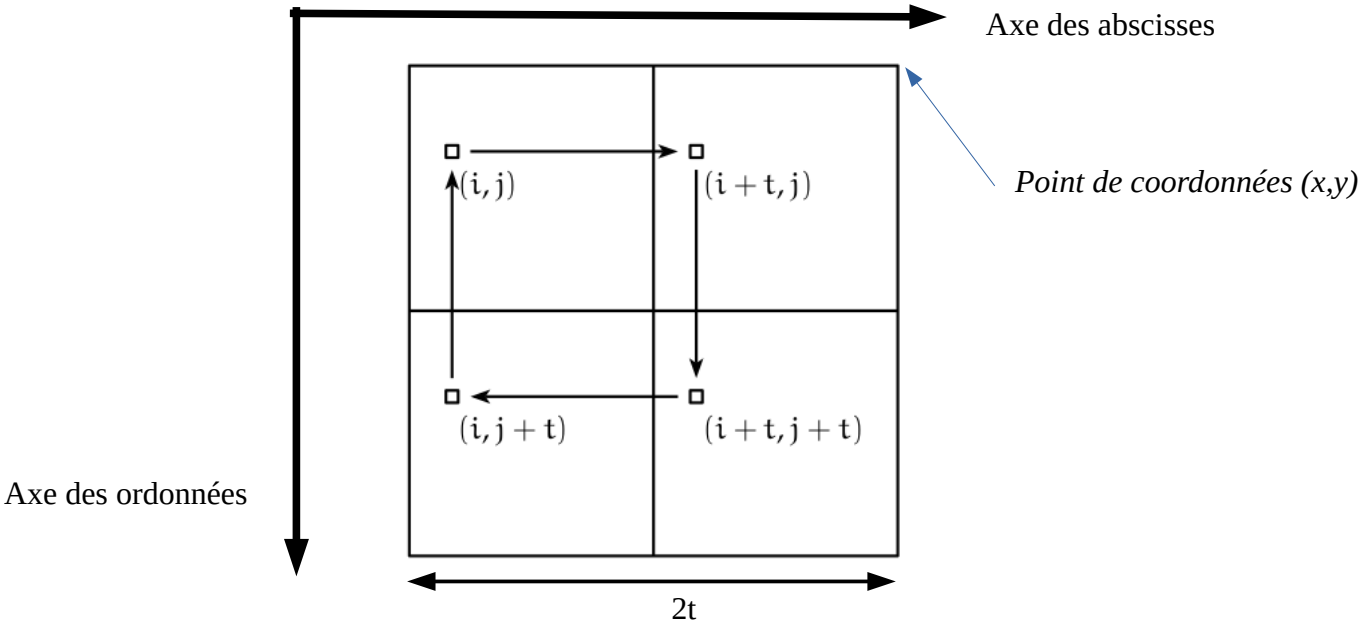
1.b) Sachant qu'un pixel est représenté par quatre nombres entiers compris entre 0 et 255 (rouge, vert, bleu et transparence), estimer le coût supplémentaire en mémoire (en Mo) de cette méthode rotation.

2) On souhaite mettre en œuvre une solution s'appuyant sur la méthode « diviser pour régner » dont le coût supplémentaire en mémoire est négligeable. Pour une image carrée dont le côté est une puissance de 2, le principe est le suivant :

- on découpe l'image en quatre carrés de côté t (ce découpage est une vue de l'esprit, dans le sens où aucune structure de données n'est à créer en mémoire pour y enregistrer les pixels de ces quatre morceaux d'image)

- si t > 1, on applique récursivement la méthode rotation_dpr à chacun de ces quatre carrés (cf spécification ci-dessous ; cette méthode se substitue à la méthode rotation de la partie 1)

- on permute ensuite les valeurs des pixels de ces quatre carrés selon le schéma ci-dessous



En utilisant ce principe, écrire le code de la méthode rotation_dpr(self,x, y, t) pour qu'elle effectue la rotation d'un quart de tour du carré de côté 2t dont le coin supérieur droit a pour coordonnées x et y.

Pour aller plus loin : On proposera à l'utilisateur, depuis la console, d'effectuer une rotation d'un quart de tour du fichier image de son choix, en lui proposant de choisir le sens de rotation, en le prévenant que l'image sera rognée (en lui montrant de quelle manière) si elle n'est pas carrée de longueur une puissance de 2, etc …

Projet 2 : Détermination de la distance minimale entre deux points dans un ensemble de points du plan

Étant donnés n points du plan, on cherche à déterminer la plus petite distance entre deux de ces points.
On supposera dans la suite que les points sont donnés sous la forme d'un tableau de couples de coordonnées.

1) Soient *n* un entier naturel, *x_max* et *y_max* deux nombres réels strictement positifs donnés.
Générer un tableau de *n* points dont les coordonnées sont des nombres réels choisis aléatoirement dans les intervalles [*0,x_max*] pour l'abscisse et [*0,y_max*] pour l'ordonnée.

2) Ecrire une fonction distance(*p, q*) qui renvoie la distance entre deux points, les couples *p* et *q* représentant leurs coordonnées.

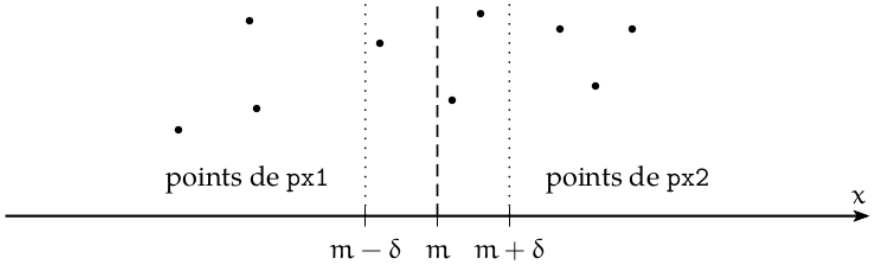
3.a) Écrire une fonction distance_minimale_force_brute(*pts*) qui renvoie la plus courte distance entre deux points du tableau *pts* en utilisant une approche par « force brute », c'est-à-dire en testant tous les segments que l'on pourrait tracer à partir des points de l'ensemble *pts* !

3.b) Estimer le nombre d'appels à la fonction distance en fonction de la taille *n* du tableau *pts*.

4) Ce problème se prête à une approche « diviser pour régner », que l'on présente dans cette partie.
Tout d'abord, la stratégie suppose que l'on dispose de deux tableaux *px* et *py* qui contiennent les coordonnées des points triés respectivement selon leur abscisse croissante et leur ordonnée croissante.
Le principe de l'algorithme est alors le suivant.

i) On coupe le tableau *px* en deux sous-tableaux *px1* et *px2* de « même taille ».
ii) On applique récursivement l'algorithme pour déterminer la plus petite distance d_1 entre deux des points de *px1*, et la plus petite distance d_2 entre deux des points de *px2*.
iii) La plus petite distance d_{min} entre deux des points de px est alors atteinte dans l'un des cas suivants :
- entre deux points de *px1*, auquel cas $d_{min} = d_1$
- entre deux points de *px2*, auquel cas $d_{min} = d_2$
- entre un point de *px1* et un point de *px2*.

Il faut donc déterminer la plus petite distance entre les points de *px1* et les points de *px2*.
Pour cela, on note *m* la moyenne des abscisses du dernier élément de *px1* et du premier élément de *px2* : le tableau *px* étant trié, tous les points de *px1* ont une abscisse inférieure à *m*, tandis que tous les points de *px2* ont une abscisse supérieure à *m*. En notant $\delta = \min(d_1, d_2)$, la recherche de la plus courte distance peut alors être limitée aux points dont les abscisses sont comprises entre $m - \delta$ et $m + \delta$.



En sélectionnant les points du tableau *py* dont les abscisses sont comprises entre $m - \delta$ et $m + \delta$, il ne reste plus qu'à faire la remarque cruciale suivante : la différence entre les ordonnées des éléments d'indice i + 7 et d'indice i est supérieure à δ.

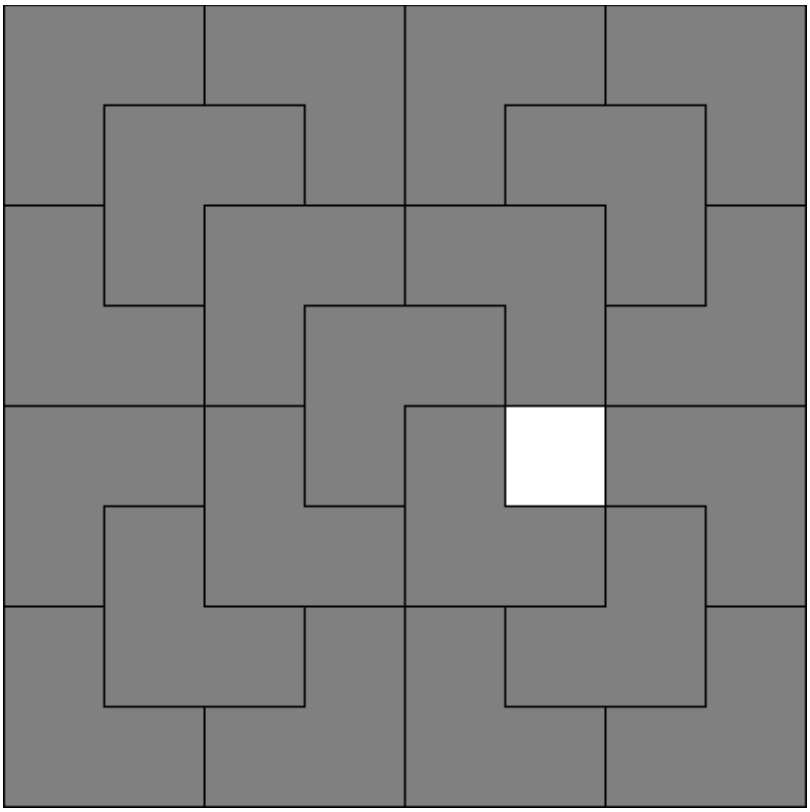
4.a) Écrire une fonction trier(pts) qui renvoie les tableaux triés px et py.
4.b) Écrire une fonction récursive distance_minimale_dpr qui prend en argument *px* et *py*, et met en œuvre la méthode proposée ci-dessus.
4.c) La fonction obtenue est-elle plus performante que la fonction distance_minimale_force_brute ? Optimiser l'implémentation pour que ce soit bien le cas.

Nota Bene : on prêtera une attention toute particulière aux tests menés pour vérifier le bon fonctionnement de chaque partie de l'algorithme Ces tests devront envisager les différents cas possibles …

Projet 3 : La grille avec une case manquante

Sur une grille carrée comportant $2^n \times 2^n$ cases, on souhaite disposer les motifs ci-dessous pour qu'ils recouvrent la grille, à l'exception d'une seule case (il n'est pas possible de recouvrir parfaitement la grille avec ces motifs).

Exemple de recouvrement avec n = 3.



Écrire un algorithme récursif s'appuyant sur une stratégie « diviser pour régner » qui permette, étant donnée une case repérée par ses coordonnées (i, j), de recouvrir un échiquier carré de dimensions $2^k \times 2^k$ par les motifs ci-dessus.

L'utilisateur devra pouvoir choisir facilement, par exemple depuis la console, la valeur de n, ainsi que l'emplacement de la case vide.

La partie graphique sera implémentée en utilisant le module turtle de python.