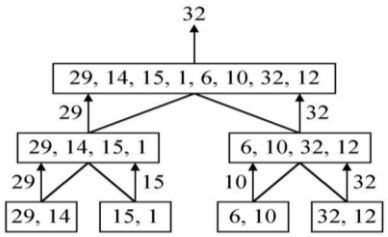
TNSI	Algorithmique Diviser pour régner	TP – « Diviser pour régner »

Objectifs :

- ⇨ Mettre en œuvre plusieurs algorithmes de type « diviser pour régner »
- ⇨ Voir l'intérêt en termes de complexité de ces algorithmes



I – Echauffement

On commence avec un algorithme qui n'est pas tout à fait du type « Diviser Pour Régner » (DPR), mais qui y ressemble : la recherche dichotomique.

Question 1 :

1) Ecrire une fonction `recherche_dichotomique(t, val)` qui prend en argument un tableau `t` et renvoi l'indice de l'élément du tableau ayant la valeur `val` (s'il y a plusieurs éléments valant `val`, n'importe lequel conviendra) ou `None` si aucun élément valant `val` n'est présent dans le tableau `t`. Testez votre fonction avec des assertions.
2) Si vous avez écrit une fonction récursive, écrivez une version itérative. Si vous avez écrit une version itérative, faire maintenant une version récursive. Dans ce dernier cas on pourra ajouter deux paramètres optionnels à la fonction pour indiquer l'indice de début et de fin de la partie du tableau considérée.

II – Minimaxi

On a déjà écrit des fonctions qui recherchent le minimum et le maximum dans un tableau, mais il s'agissait à chaque fois de déterminer uniquement le minimum ou uniquement le maximum. On cherche maintenant à écrire une fonction qui recherche à la fois le maximum et le minimum d'un tableau.

Question 2 :

1) Ecrire une fonction `minimaxi(t)` qui prend en argument un tableau `t` et renvoi un tuple contenant le minimum et le maximum des valeurs du tableau `t`.
   Par exemple `minimaxi([2, 5, 12, 9, 7, -1])` renverra le tuple `(-1, 12)`.
   On pourra par exemple chercher le minimum, puis le maximum.
2) Quel est la complexité de l'algorithme que vous avez écrit en nombre de comparaisons ?

On cherche maintenant à écrire un algorithme du type « diviser pour régner » pour diminuer le nombre de comparaisons à effectuer.

Question 3 :

1) Combien de comparaisons sont-elles nécessaires pour déterminer le minimum d'un tableau de n nombres ?
2) Combien de comparaisons sont-elles nécessaires pour déterminer le minimum, puis ensuite le maximum de n nombres ?
3) On cherche à déterminer le maximum et le minimum d'un tableau contenant deux nombres. Combien de comparaisons sont-elles nécessaires dans ce cas ?

On voit qu'on obtient un nombre de comparaisons meilleur dans le cas particulier de la comparaison de 2 nombres car on peut simultanément déterminer le maximum et le minimum puisque si l'un des deux nombres est le minimum alors l'autre est forcément le maximum.
On va chercher à utiliser cette propriété pour écrire un algorithme plus efficace.

Question 4 :

1) Ecrire le pseudo-code d'une fonction `minimax_DPR(t)`, basée sur la méthode « diviser pour régner » et qui détermine, comme la fonction `minimax` précédente, le minimum et le maximum du tableau `t`.
2) Ecrire la fonction python correspondante et la tester.
3) Combien cette fonction utilise-t-elle de comparaisons pour un tableau de 4 éléments ? De 8, 16 ou 32 éléments ?

Si on a $n = 2^k$ éléments dans le tableau, le nombre de comparaisons suit une suite arithmético-géométrique : $u_k = 2 \cdot u_{k-1} + 2$. On peut alors retrouver le nombre de comparaisons u_k en fonction de n en étudiant cette suite. Voir sur https://fr.wikipedia.org/wiki/Suite_arithm%C3%A9tico-g%C3%A9om%C3%A9trique pour déterminer u_k en fonction de n.

4) Conclure sur la complexité de cet algorithme. L'algorithme DPR est-il plus efficace ?

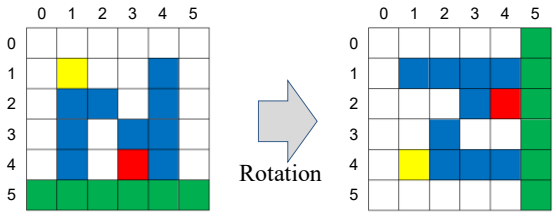
III - Rotation d'une image

Dans cette partie, nous allons nous intéresser à la rotation à 90° (vers la gauche) d'une image de dimensions carrées.

1) Méthode naïve

On peut montrer que lors d'une rotation de 90° vers la gauche d'une image de taille n×n pixels, un pixel initialement aux coordonnées (x, y) doit être déplacé aux coordonnées (y, n – 1 – x).

Image de taille 6×6 => n = 6



Le point **jaune** passe de la coordonnée (x=1, y=1) à (x'=1, y'=4)
On a bien x' = y et y' = n – 1 – x = 6 – 1 – x = 5 – x = 5 – 1 = 4

Le point **rouge** passe de (x=3, y=4) à (x'=4, y'=2)
On a bien x' = y et y' = n – 1 – x = 6 – 1 – x = 5 – x = 5 – 3 = 2

Pour tourner une image de 90°, il suffit donc de créer une image vide de n×n pixels, dans laquelle on recopie tous les pixels de l'images originale en les transposant au bon endroit à l'aide de la formule vue plus haut.

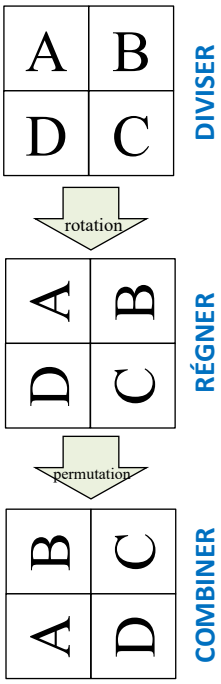
Afin de manipuler facilement les images en python, on va utiliser la bibliothèque PIL dont la documentation est disponible en ligne, mais pour laquelle le tableau suivant récapitule les principales fonctions à utiliser :

Syntaxe	Fonction
`from PIL import Image`	Permet d'importer la classe `Image` de PIL qui contient tout ce dont on aura besoin pour ce TP.
`img = Image.open(nomFichier)`	Ouvre le fichier dont on spécifie le nom (sous forme d'une chaîne de caractère) et renvoie un objet `Image` qui permet de manipuler l'image et ses pixels. PIL peut ouvrir de nombreux formats d'image. Nous utiliserons plus particulièrement des fichiers png.
`img.show()`	Affiche l'image `img` en utilisant le programme d'affichage d'image par défaut du système d'exploitation.
`largeur,hauteur = img.size`	`size` est un attribut de la classe `Image` contenant un tuple de deux entiers représentant respectivement la largeur et la hauteur de l'image en nombre de pixels.
`img2 = Image.new("L", (l, h))`	Crée une image vide de taille `l`×`h` pixels en niveaux de gris (`"L"`) ou en couleur (`"RGB"`).
`couleur = img.getpixel((x,y))`	Récupère la couleur du pixel aux coordonnées `x`,`y` dans l'image `img`.
`img.putpixel((x,y),couleur)`	Fixe la couleur du pixel de coordonnées `x`, `y` de l'image `img` à la valeur `couleur`.

Question 5 :

1) Ecrire un programme qui ouvre l'image « Turing.png[1] » et qui l'affiche.
2) Ecrire une fonction `tourne(img)` qui renvoie une version tournée à 90° vers la gauche de l'image `img` (sous forme d'objet `Image` de PIL) fournie en argument.

2) Méthode diviser pour régner

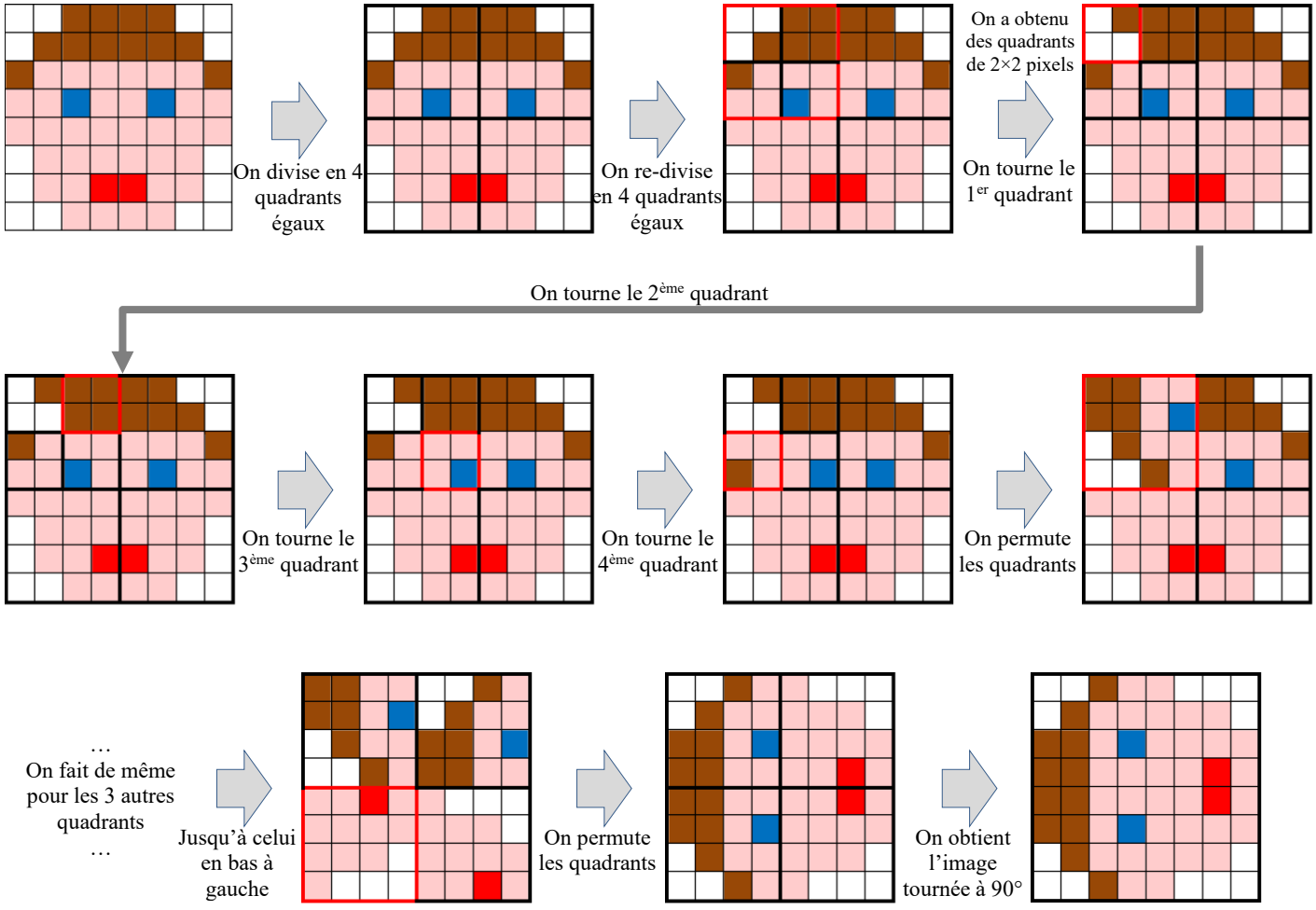


Il est possible d'appliquer la méthode « diviser pour régner » à la rotation de 90° d'une image carrée.
On procède alors par étapes :

1. on coupe l'image en quatre quadrants (A, B, C, D sur l'exemple ci-contre) ;
2. on effectue une rotation récursive de chacun des quadrants ;
3. on applique une permutation circulaire des quadrants.

La division s'effectue récursivement jusqu'à une image de 2×2 pixels où il suffit de permuter les pixels pour obtenir la rotation de l'image 2×2.

Voyons ce que cela donne avec une image élémentaire de 8×8 pixels :



[1] Le temps de traitement de vos algorithmes risque d'être long par la suite avec cette image de 512×512 pixels. Pour les tests, on pourra utiliser les images « Turing_256.png » ou « Turing_128.png » qui sont réduites et permettent des temps de traitement plus acceptables.

Question 6 :

1) Ecrire une fonction `echange_pixels(img,pixelA,pixelB)` qui échange dans l'image `img` les couleurs des pixels A et B dont on donne les coordonnées x et y sous forme de tuples. Tester la fonction avec l'image du fichier « tete.png ».
2) Quelles sont les 3 permutations de quadrants successives qu'il faut effectuer pour passer de la position ABCD à la position BCDA comme sur la figure précédente et ainsi réaliser une rotation des sous-images ?
3) En s'appuyant sur la fonction `echange_pixels` et la réponse à la question précédente, écrire une fonction `rotation2x2(img, coinHG)` qui effectue la rotation de la sous-image de 2×2 pixels dans l'image `img` dont les coordonnées x,y du coin supérieur gauche sont données en argument sous forme du tuple `coinHG`. Tester la fonction avec l'image du fichier « tete.png ».
4) Ecrire une fonction `echangeQuadrant(image,coinHG_A,coinHG_B,n)` qui échange les quadrants A et B dont on donne la taille `n` en pixels et les coins supérieurs gauche sous forme de tuples x,y. Tester la fonction avec l'image du fichier « tete.png » puis du fichier « Turing.png » pour des valeurs de `n` plus grandes.
5) On veut écrire une fonction `tourneRecursif(image,coinHG,taille)` qui tourne récursivement la partie de `taille`×`taille` pixels de l'image `img` commençant dans le coin en haut à gauche de coordonnées x,y fournies dans le tuple `coinHG`.
   a) Quel est le cas de base de la récursion ? Quelle fonction déjà écrite peut-on appeler dans ce cas ?
   b) Dans le cas général, quel doit être la taille des 4 sous-quadrants dans lesquels on peut décomposer le quadrant pour lequel la fonction est appelée ?
   c) Quelles doivent être les coordonnées des coins en haut à gauche des 4 sous-quadrants ? On attend pour chacun une formule de calcul en fonction de `coinHG` et de `taille`.
   d) Ecrire la fonction en séparant bien les parties DIVISER, REGNER et COMBINER de l'algorithme.
6) Ecrire une fonction `tourne_DPR(img)` qui s'appuie sur la précédente pour tourner l'image `img`.

Question 7 :

1) Quelle est la complexité de la fonction `tourne` de l'algorithme naïf (question 5) ?
2) Même question pour `tourne_DPR` programmée à la question précédente. Les complexités sont-elles comparables ?
3) Exécuter les deux fonctions sur l'image « Turing_128.png », puis « Turing_256.png » et comparer leur temps d'exécution (les temps de calculs sont suffisamment longs pour être chronométrés directement « à la main »).
4) Quelle est la complexité spatiale (nombre d'élément de mémoire utilisés en fonction de la taille n des données) pour chaque algorithme (naïf et DPR) ?
5) Finalement quel est l'intérêt de la méthode « diviser pour régner » dans notre cas ?