BERGESE Jérôme
DARRIN Emilie
DULAURANS Thierry
UZAN Stephen
groupe 3B
Bloc 2

Algorithmes

Programme de 1ere – NSI

Algorithmique

Contenus	Capacités attendues	Commentaires
Tris par insertion, par sélection	**Écrire un algorithme de tri. Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection**	**La terminaison de ces algorithmes est à justifier. On montre que le leur coût est quadratique dans le pire cas.**

Prérequis :

images numérique
bibliothèques PIL
algorithmes (papier/crayon)

Activité découverte

Lorsque que l'on veut enregistrer une image sur une clé USB , transférer des images d'un appareil à un autre, ou encore prendre des photos lorsque les conditions météorologiques sont dégradées (pluie, nuit, etc.), cela peut amener à obtenir des images dites dégradées. Les images dégradées sont des images qui ont des défauts, par exemple une image floue est une image dégradée. Un exemple de dégradation vous est présenté ci-dessous, on peut voir l'image non dégradée sur la gauche, et celle dégradée sur la droite :

Cette dégradation vient du fait que lorsque l'image a été écrite sur la carte mémoire de l'appareil photo ou sur celle de votre Smartphone, les informations enregistrées étaient soit mauvaises, soit manquantes. Il faut savoir qu'une image numérique en niveau de gris est en fait un tableau à deux dimensions dans lequel on voit apparaître des nombres entre 0 et 255. Ils représentent les intensités de chaque pixel de l'image. Il suffit d'une petite modification de ces intensités pour que l'image perde en qualité. Voilà à quoi pourrait ressembler une image numérique en niveau de gris, une fois enregistrée sur votre Smartphone :

153	153	147	143	137	130	127	122	124
154	151	146	140	132	126	121	117	115
154	152	143	139	128	125	117	112	112
150	144	139	132	123	115	108	103	106
146	145	133	127	119	110	106	97	98
146	140	128	122	111	109	98	97	99
141	135	126	114	107	100	90	86	95
139	133	123	110	100	92	86	85	95
134	129	118	105	96	91	80	84	95
133	122	112	96	87	81	83	85	91
127	113	100	90	85	83	81	86	92
122	106	93	87	82	78	80	86	93
112	100	88	79	80	80	86	90	99
107	96	86	81	80	81	85	92	102
100	89	86	82	81	77	86	91	103
94	86	83	82	83	84	85	93	101
91	83	82	85	82	83	90	90	96
86	85	86	91	83	83	86	89	98
87	109	84	85	85	79	86	93	101
85	85	89	89	89	86	86	93	96
90	90	90	93	89	86	82	93	101

Ce tableau est en fait une partie de l'image Léna présentée en début de TP. Après avoir transféré l'image de Lena sur une clé USB, il s'avère qu'en l'ouvrant, on s'est aperçu que l'image était dégradée. La nouvelle image de Lena est la suivante :

On peut voir clairement que cette image est très dégradée. On peut observer des points sur l'image qui sont apparus sur la photo. On appelle ce genre de dégradation du bruit sur l'image et plus particulièrement du bruit aléatoire (les pixels aberrants ont des valeurs entre 0 et 255). Garder l'image en l'état ne paraît pas concevable...

But : Dans ce cours, nous allons donc essayer d'enlever ou d'atténuer le bruit de l'image afin d'améliorer sa qualité. On va essayer de *débruiter* l'image.

1) Le statisticien *J:W: Tuckey* qualifiait d'*aberrantes* les valeurs d'une série statistique se trouvant en dehors de l'intervalle :

$$\left[Q_1 - \frac{3}{2}(Q_3 - Q_1); Q_1 + \frac{3}{2}(Q_3 - Q_1)\right]$$

On s'intéresse au tableau de pixels ci-dessous :

100	102	99
101	25	103
103	100	103

a) Calculer le premier et troisième quartile de cette série.

Pour calculer les quartiles 1 et 3 de cette série statistique à 9 données, on commence par trier les éléments de la série dans l'ordre croissant. On obtient alors:
[25, 99, 100, 100, 101, 102, 103, 103, 103]

Une fois ceci fait, on calcule les positions de Q1 et Q3:
posQ1 = 9/4 = 2,25 -> 3ème position du tableau
posQ3 = 3*9/4 = 6,75 -> 7ème position du tableau

On a donc: Q1 = 100 et Q3 = 103

b) Le pixel central peut-il être considéré comme aberrant ? **(Justifier)**

Pour savoir si le pixel central est aberrant, on regarde s'il appartient à l'intervalle de Tuckey.
[Q1 - 3*(Q3 - Q1)/2; Q1 + 3*(Q3 - Q1)/2] = [-95,5; 104,5] ~ [95; 104] (car les pixels de l'image sont des valeurs entières)
25 n'appartient pas à l'intervalle [95; 104]. Il est donc considéré comme aberrant.

c) Si oui, on se propose de remplacer ce pixel par la moyenne des 9 nombres des pixels concernés. Calculer la. Cette valeur semble-t-elle plus convenable ?

On calcule la moyenne de cette série statistique:
m = (25 + 99 + 100 + 100 + 101 + 102 + 103 + 103 + 103)/9 ~ 93

Cette valeur se rapproche de l'intervalle de Tuckey mais est toujours aberrante.

2) On considère ci-dessous un tableau de pixels quelconque.

a	b	c
d	e	f
g	h	i

pléter l'algorithme ci-dessous de façon à ce qu'il remplace le pixel central si nécessaire :

ALGO :

- Affecter la valeur du premier quartile à Q_1.
- Affecter la valeur du troisième quartile à Q_3.
- Si $e < \ldots$ OU $e > \ldots$ Alors
- Affecter $\ldots$ à e.
- FINSI

- Affecter la valeur du premier quartile à Q1
- Affecter la valeur du troisième quartile à Q3
- Si e < (Q1 - 3*(Q3 - Q1)/2) ou e > (Q1 + 3*(Q3 - Q1)/2)
- Affecter la moyenne de la série statistique à e
- FINSI

3)

a) Par quel autre indicateur aurait-on pu remplacer la valeur *e* ?

On aurait pu remplacer la valeur e par la médiane de la série statistique.

b) Faire le calcul dans l'exemple de la question 1) et expliquer en quoi cette méthode semble plus judicieuse.

Pour calculer la médiane, il faut calculer la position de cette dernière dans la série ordonnée.
posMe = 9/2 = 4,5 -> 5ème position
Me = 101

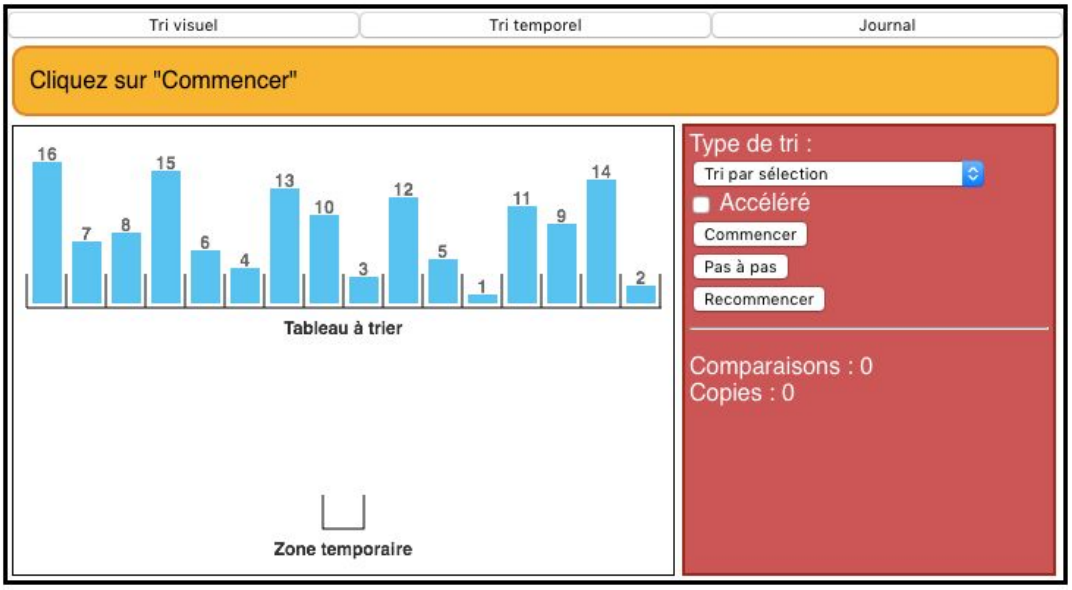
La valeur de la médiane appartient à l'intervalle de Tuckey, ce n'est donc plus un pixel aberrant. Cela peut s'expliquer par le fait que la moyenne est fortement impactée par les valeurs aberrantes (très petites ou très grandes) alors que la médiane n'en tient pas compte, car se base sur une position dans une série ordonnée.
Cette nouvelle façon de débruiter est appelé filtre médian.

1. Programmer en langage Python, les deux fonctions de débruitage *debruitageMoyenne* et *debruitageMediane* associé aux conditions de Tuckey et prenant en entrée une image dégradée (bruitée), sa taille, l'image débruité.

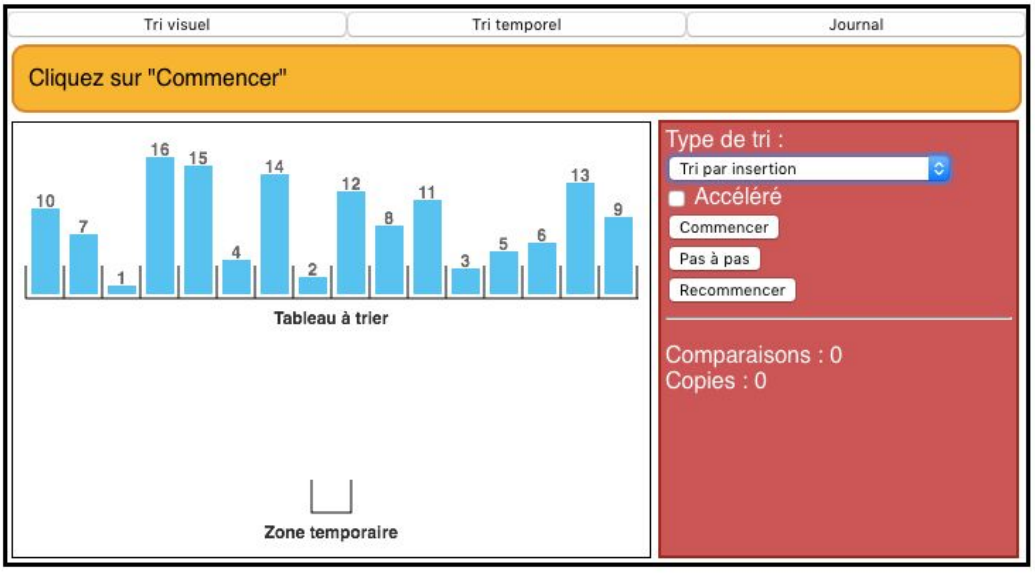
(*cf.* annexe code python)

Les deux programmes utilisés pour le débruitage de l'image font appel à une fonction de tri. Il en existe plusieurs sortes. Nous allons dès à présent, vous en présenter deux sortes: *tri par insertion* et *tri par sélection* dont vous trouverez le mode de fonctionnement à l'adresse suivante:
https://interstices.info/les-algorithmes-de-tri/

Vous allez obtenir, dans votre fenêtre de navigation, une fenêtre de test des algorithmes de tri, comme vu ci-dessous:



Vous allez donc pouvoir choisir le type de tri souhaité pour ranger dans l'ordre croissant, les valeurs des données présentes dans le tableau de droite.



-> Cours sur les tris.

1. A l'aide des explications données sur ce site, pour ranger les données, écrire sur votre cahier, les algorithmes de tri *insertion* et *sélection*.

(*cf.* annexe code python)

2. Une fois ceci fait, programmer ces deux algorithmes dans votre script Python, afin de pouvoir les utiliser dans votre fonction de débruitage d'image.

(*cf.* annexe code python)

3. A l'aide de la fonction *time* du package *time*, afficher les temps d'exécution de l'algorithme de débruitage d'image à l'aide des trois tris de données suivants:
   - sort de Python,
   - insertion,
   - sélection.

   Que constate-t-on?

(*cf.* annexe code python)

4. En changeant la taille des masques du filtre médian, ainsi que les fonctions tris, que remarque-t-on?

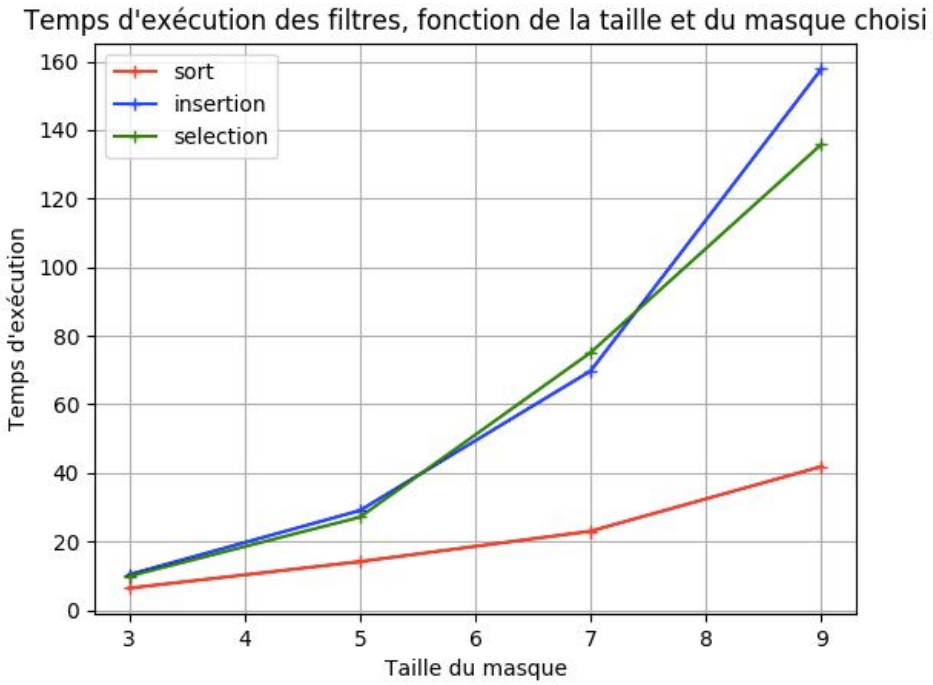
filtre médian 3x3:

filtre médian 5x5

filtre médian 7x7

filtre médian 9x9

On constate que plus la taille du masque augmente, moins il y a de bruit dans l'image, mais on perd également les informations importantes dans l'image tel que les contours, ce qui rend l'image de plus en plus floue.



On constate que les tris par insertion et sélection sont de l'ordre de n^2, alors que celui de Python semble être plus rapide avec une complexité de l'ordre de nlog(n).
Ces différences ont été vues en cours. On a vérifié que les tris insertion et sélection sont de l'ordre de n^2.

Cours

Tri par sélection (sélection du minimum)

Le principe du tri par sélection/échange (ou tri par extraction) est de parcourir le tableau, par exemple de gauche à droite, et d'aller chercher le plus petit élément du tableau pour le mettre en premier, puis de repartir du second élément et d'aller chercher le plus petit élément du tableau pour le mettre en second, etc…

En notant les éléments de 0 à n-1, au $i^{ème}$ passage tous les élément du tableau de T[0] à T[i-1] sont déjà triés. On sélectionne donc l'élément ayant la plus petite valeur parmi les éléments {T[i]…T[n-1]} et on l'échange avec T[i].
A la fin du $i^{ème}$ passage on est certain que les éléments sont triés jusqu'à T[i].

Algorithme correspondant :

```
fonction tri_par_selection(tableau t, entier n)
    pour i de 0 à n - 2
        min ← i
        pour j de i + 1 à n - 1
            si t[j] < t[min], alors min ← j
        fin pour
    si min ≠ i, alors échanger t[i] et t[min]
    fin pour
fin fonction
```

Exercice d'application du tri par sélection (sélection du minimum)

1. Mettre en œuvre ce principe sur le tableau ci-dessous en écrivant le "résultat" de chaque étape :

- écrire en bleu le plus petit élément identifié au balayage du tableau ;
- écrire dans la mémoire le premier élément non trié
- écrire en vert les éléments échangés
- séparer la zone triée de la zone non triée par un trait vertical

Situation de départ

7	2	11	5	3	

Succession des opérations

1	7	2	11	5	3	
2		2	11	5	3	7
3	2		11	5	3	7
4	2	7	11	5	3	
5	2	7	11	5	3	
6	2		11	5	3	7
7	2	3	11	5		7
8	2	3	11	5	7	
9	2	3	11	5	7	
10	2	3		5	7	11
11	2	3	5		7	11
12	2	3	5	11	7	
13	2	3	5	11	7	
14	2	3	5		7	11
15	2	3	5	7		11
16	2	3	5	7	11	

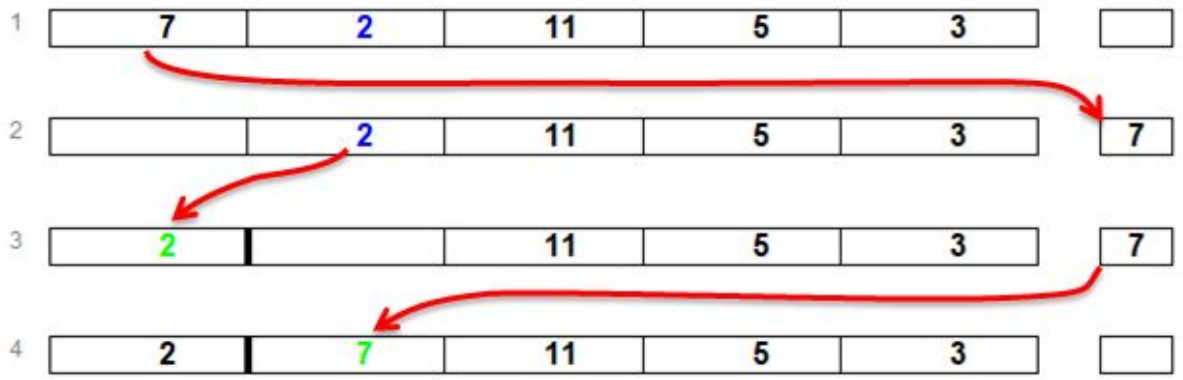
2. Quel est le nombre “d’étapes” de comparaisons ?

4 (ligne 1) + 3 (ligne 5) + 2 (ligne 9) +1 (ligne 13) = 10

Remarque : En notant *n* le nombre d'éléments on vérifie que ce nombre est $\frac{n\times(n-1)}{2}$

3. Quel est le nombre “d’étapes” de copies ?

12 (3 par échange entre “mise en mémoire” d’une des valeurs à échanger, écriture de l’autre valeur à échanger à la place de la première, écriture de la valeur de la mémoire à la place de la deuxième valeur à échanger).



4. Quel est le nombre minimal d’étapes de cet algorithme ?

Si tout est trié on a $n-1$ comparaisons, donc une complexité de l’ordre de n.

5. Quelle est la complexité “maximale” de cet algorithme ?

(Dans le “pire des cas”) on a $\frac{n\times(n-1)}{2}$ comparaisons et $3\,n$ copies.

On dit que l’on a une complexité de l’ordre de n^2.

Tri par insertion

Le principe du tri par insertion est de parcourir le tableau élément par élément, par exemple de gauche à droite, et de placer l'élément en cours de tri à la bonne place dans la partie déjà triée que l'on place par exemple à gauche.

En notant les éléments de 0 à n-1, au $i^{ème}$ passage, on traite donc l'élément T[i] et on l'insère à la bonne place entre T[0] et T[i-1] donc la partie du tableau déjà triée.
A la fin du ième passage on est certain que les éléments sont triés jusqu'à T[i].

Algorithme correspondant :

```
fonction tri_insertion(tableau T, entier n)
        pour i de 1 à n - 1
                # mémoriser T[i] dans x
                x ← T[i]
                # décaler vers la droite les éléments de T[0]..T[i-1] qui sont plus grands que x (en partant de T[i-1])
                j ← i
                tant que j > 0 et T[j - 1] > x
                    T[j] ← T[j - 1]
                    j ← j - 1
                fin tant que
                # placer x dans le "trou" laissé par le décalage
                T[j] ← x
        fin pour
fin fonction
```

Exercice d'application du tri par insertion

1. Mettre en œuvre ce principe sur le tableau ci-dessous en écrivant le "résultat" de chaque étape :

- écrire en bleu l'élément trié (mis temporairement en mémoire) ;
- écrire dans la mémoire le premier élément en cours de tri
- écrire en vert l'élément trié immédiatement supérieur à l'élément trié (s'il existe)
- séparer la zone triée de la zone non triée par un trait vertical

Situation de départ

7	2	11	5	3	

Succession des opérations

1	7	2	11	5	3	
2	7		11	5	3	2
3		7	11	5	3	2
4	2	7	11	5	3	
5	2	7		5	3	11
6	2	7	11	5	3	
7	2	7	11		3	5
8	2	7		11	3	5
9	2		7	11	3	5
10	2	5	7	11	3	
11	2	5	7	11		3
12	2	5	7	11		3
13	2	5	7		11	3
14	2	5		7	11	3
15	2		5	7	11	3
16	2	3	5	7	11	

2. Quel est le nombre d'éléments successivement mis en mémoire pour être triés ?

Il y en a 4 .

Remarque : En notant n le nombre d'éléments on vérifie que ce nombre est $n-1$.

3. Quel est le nombre “d’étapes” de copies ?

Il y en a 12

4. Quel est le nombre minimal d’étapes de cet algorithme ?

Si tout est trié on a $n-1$ comparaisons, donc une complexité de l’ordre de n .

5. Quel est le nombre maximal de comparaisons de cet algorithme ?

Dans le “pire des cas” on a i comparaisons pour chaque i variant de 2 à $n-1$.

La complexité est alors égale à la somme des $n-1$ termes suivants (i = 2, i = 3,.... i = n)

soit $2 + 3 + 4 + \ldots + n = \frac{n\times(n-1)}{2} = \frac{n^2-n}{2} - 1$ comparaisons au maximum.

C'est la somme des n premiers entiers moins 1.

On dit que l’on a une complexité de l’ordre de n^2 .

Remarque : on obtient aussi une complexité en n^2 si on prend comme critères le nombre maximal de transferts.

En effet, il y a autant de transferts dans la boucle "Tant que" qu'il y a de comparaisons il faut ajouter 2 transferts par la boucle "pour", soit au total dans le pire des cas :

$$\frac{n\times(n+1)}{2} + 2(n-1) = \frac{n^2+5n-4}{2}$$

Exercices

1- Sur papier : Dans le meilleur des cas, quelle est la complexité de chaque algorithme de tri ?

Dans le meilleur des cas les données sont triées dès le début. Il suffit alors de passer en revue le tableau, la complexité est de l'ordre de *n*.

2- Sur papier : À partir de l'exemple précédent, écrire les étapes du tri par sélection du maximum ?

Succession des opérations

1	7	2	11	5	3	
2	7	2	11	5		3
3	7	2		5	11	3
4	7	2	3	5	11	
5	7	2	3	5	11	
6	7	2	3		11	5
7		2	3	7	11	5
8	5	2	3	7	11	
9	5	2	3	7	11	
10	5	2		7	11	3
11		2	5	7	11	3
12	3	2	5	7	11	
13	3	2	5	7	11	
14	3		5	7	11	2
15		3	5	7	11	2
16	2	3	5	7	11	

3- Sur machine : Programmer le tri par sélection du minimum et le tester

4- Sur machine : Programmer le tri par insertion et le tester

5- Sur machine : Programmer le tri par sélection du maximum et le tester

6- Complexité d'autres algorithmes

- Déterminer les tailles des 4 images proposées dans le dossier "autre_algo".
- En déduire le nombre de pixel de chaque image. Que remarque-t-on pour le nombre de pixels quand on passe d'une image à la suivante ?
- Ouvrir le programme "programme_inconnu.py", identifier ce qu'il fait.
- Lancer le programme et observer le résultats.
- Conclure sur le lien entre la durée de traitement de chaque image et le nombre de pixel de chaque image. L'expliquer.

fichier	taille image (en pixels)	nombre de pixels
image1	375×250	93750
image2	750×500	375000
image3	1500×1000	1500000
image4	3000×2000	6000000

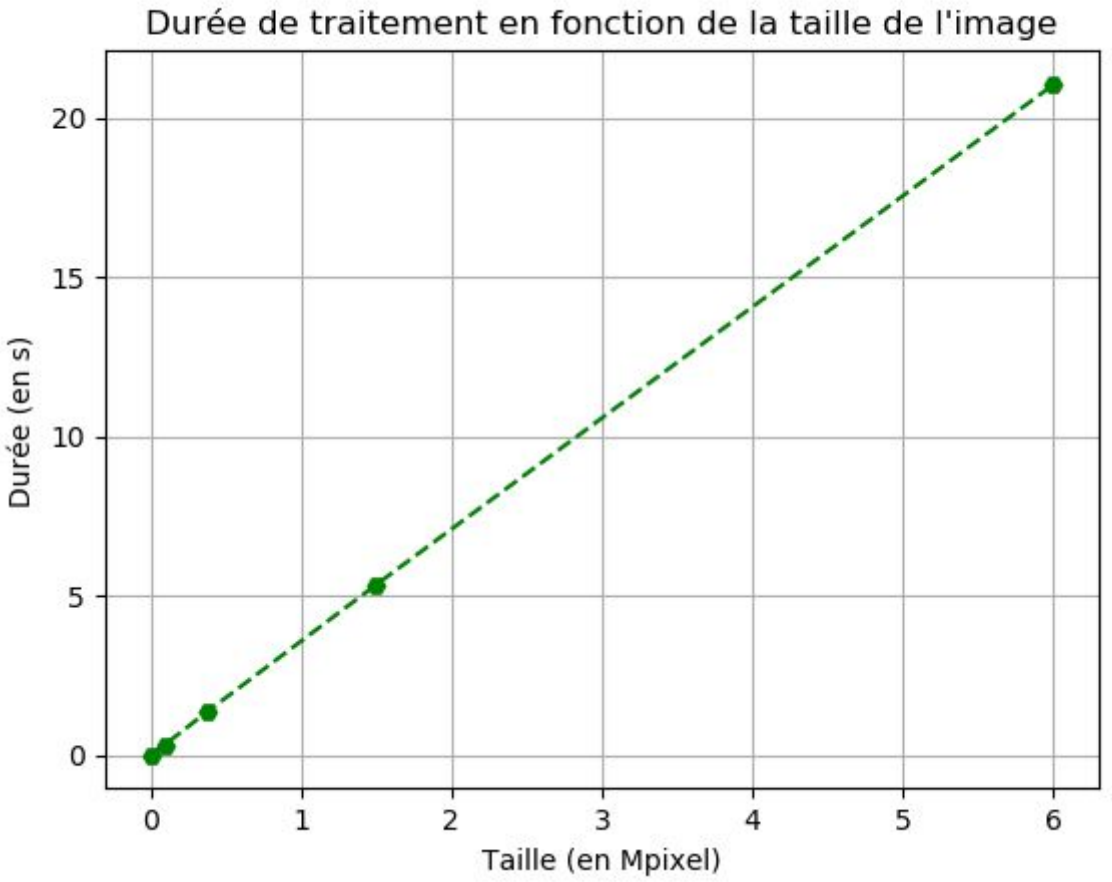
On peut remarquer que que le nombre de pixels est multiplié par 4 quand on passe d'une image à la suivante.

Le programme "programme_inconnu.py" :

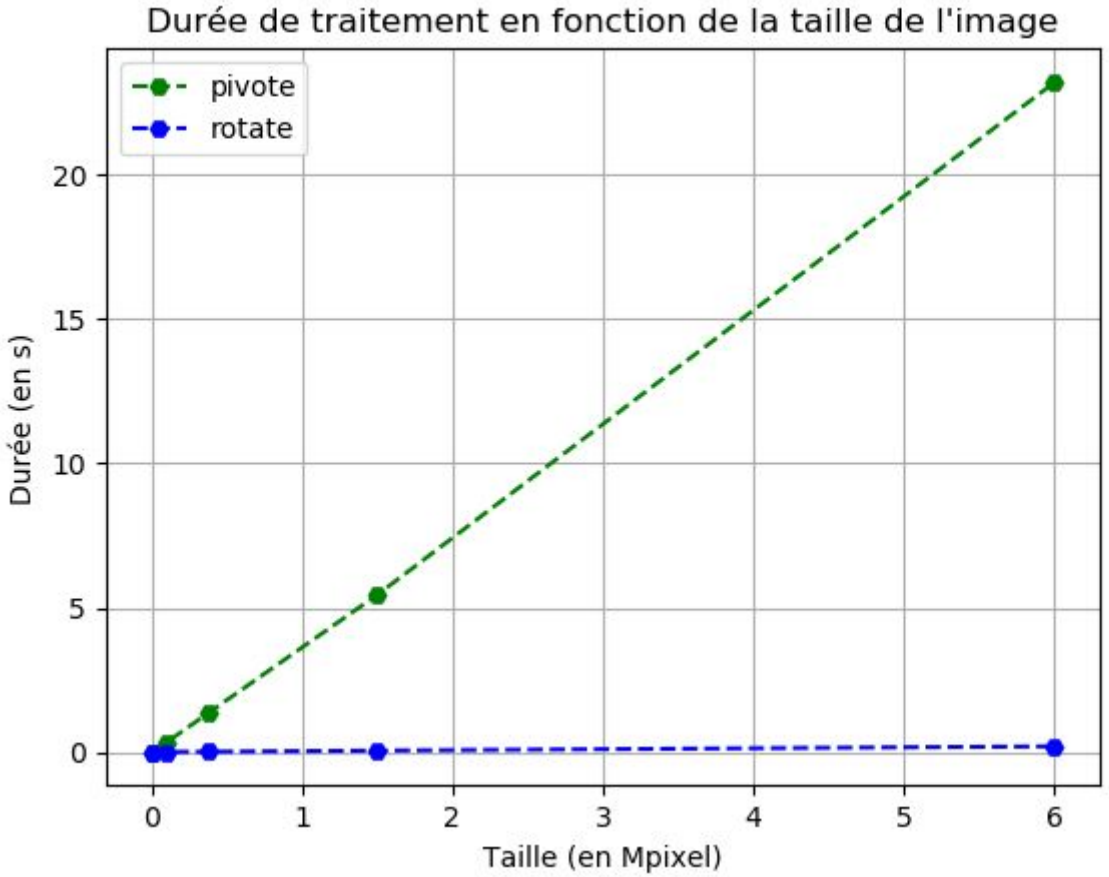
- ouvre les 4 images ;
- appelle une fonction "pivote" successivement sur ces 4 images ;
- cette fonction :
    - déclenche le chronométrage ;
    - détermine la taille de l'image ;
    - forme une chaîne de caractères constituée de la largeur et de la hauteur de l'image ;
    - calcule la taille (nombre de pixel en Mpixels) ;
    - crée une nouvelle image dans laquelle la largeur et la hauteur sont inversées ;
    - passe en revue tous les pixels, ligne après ligne ;
    - les colle dans la nouvelle image en basculant d'un quart de tour ;
    - calcule la durée depuis le début du chronométrage ;
    - sauve l'image obtenue sous le nom "imagexxx.jpg" où "xxx" est la chaîne construite plus haut ;
    - affiche la durée ;
    - renvoie la taille et la durée.
- La taille et la durée sont ajoutées tour à tour dans un tableau T ;

- On extrait un tableau Xp contenant les tailles ;
- extrait un tableau yp contenant les durées ;
- trace le nuage de points de la durée en fonction de la taille.

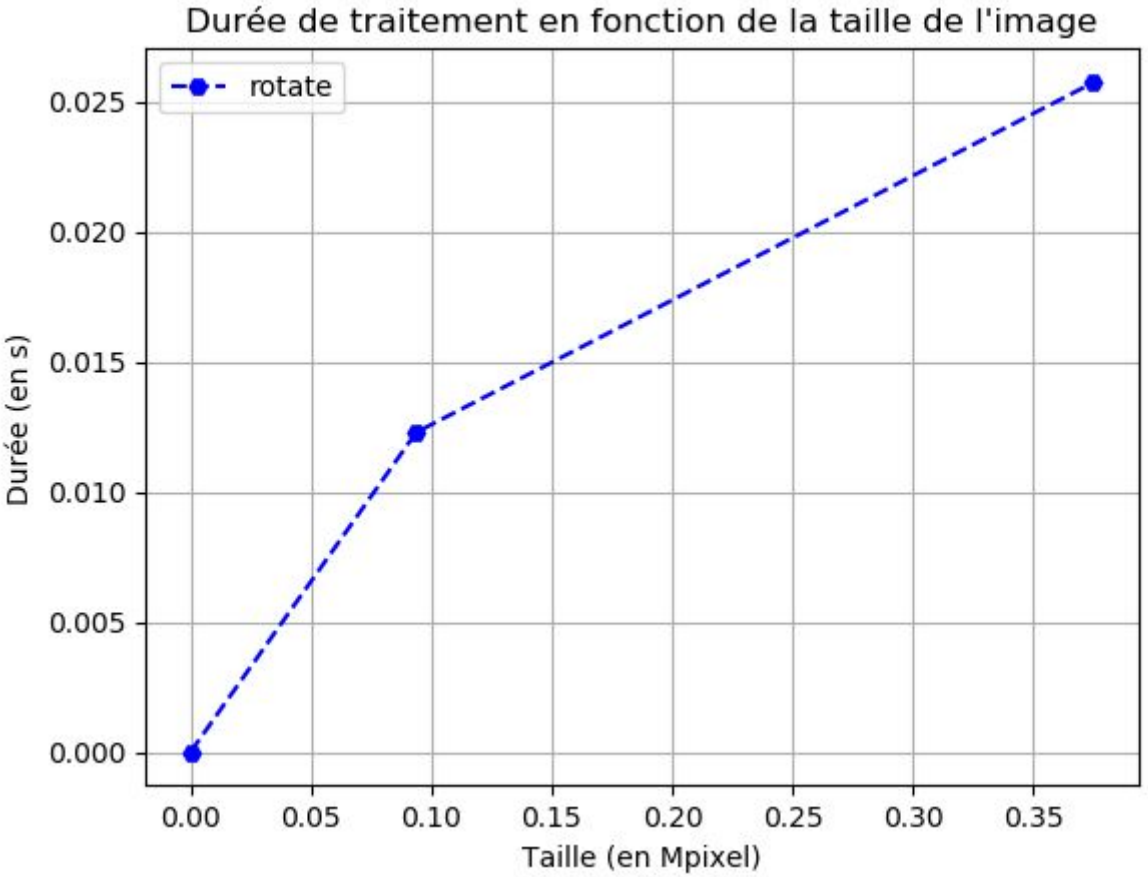
On observe une progression linéaire : la durée croît comme le nombre de pixels. C'est logique car le nombre d'opérations à effectuer sur les pixels double quand le nombre de pixels double.

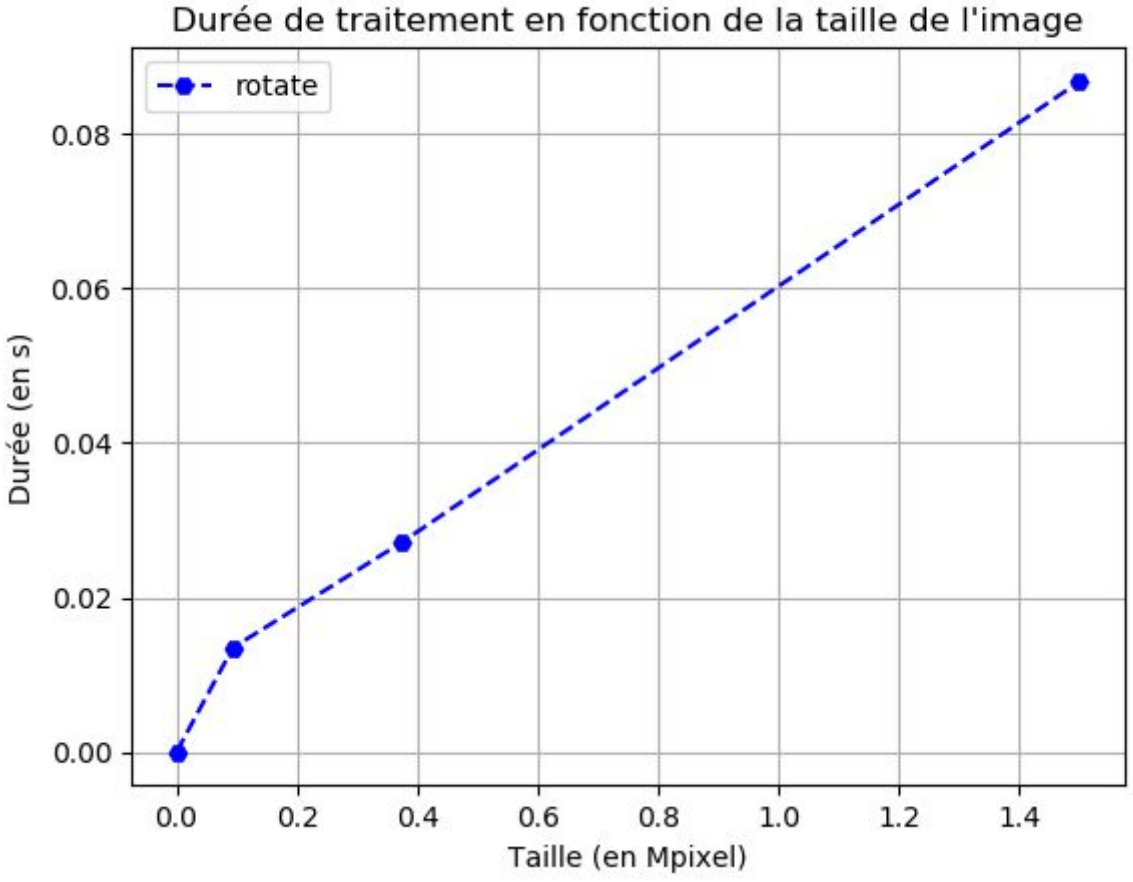


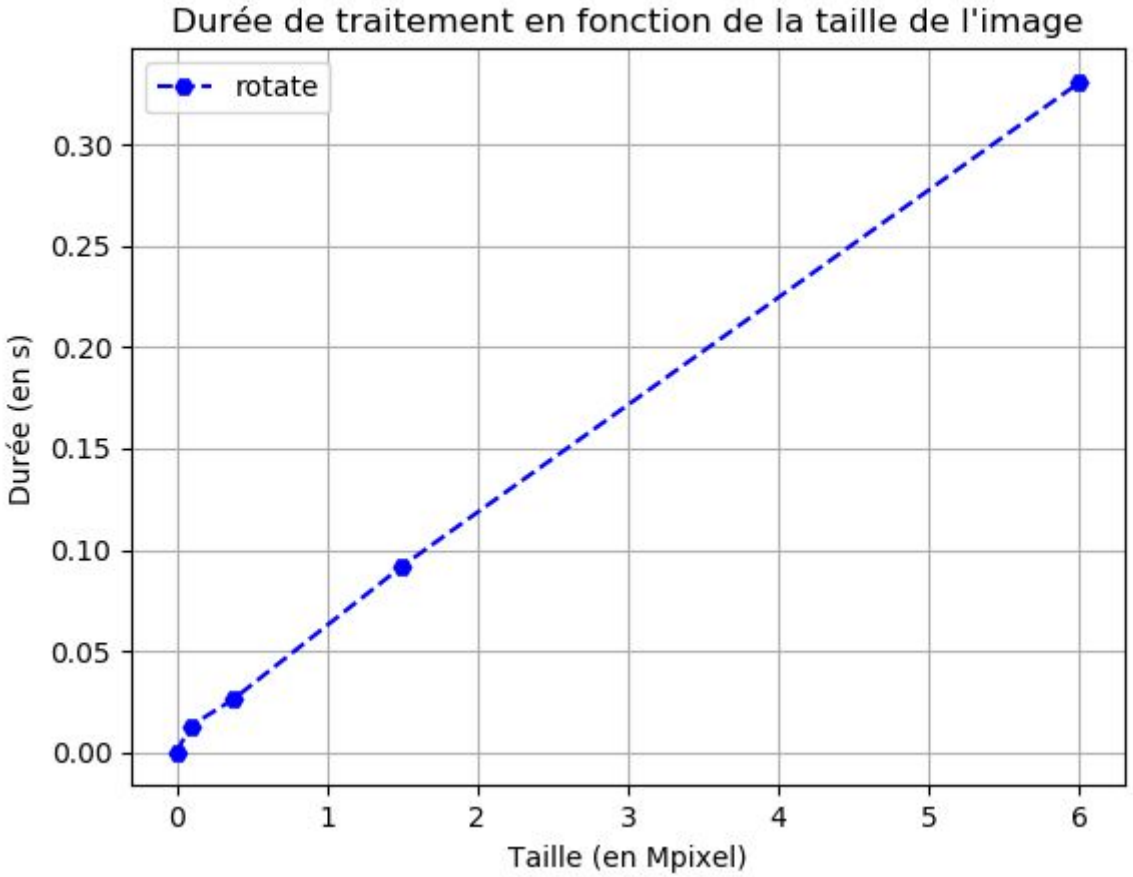
Pour les plus rapide : ouvrir le fichier "autre_programme_inconnu.py" et comparer l'agorithme précédent avec la fonction "rotate" préprogrammée dans python.



La fonction rotate est bien plus rapide… On peut cependant observer qu'elle n'est pas linéaire pour les "petites" images mais que cela devient négligeable quand la taille devient grande…


Durée de traitement en fonction de la taille de l'image
rotate
Durée (en s)
0.025
0.020
0.015
0.010
0.005
0.000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
Taille (en Mpixel)


Durée de traitement en fonction de la taille de l'image
rotate
Durée (en s)
0.08
0.06
0.04
0.02
0.00
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Taille (en Mpixel)


Durée de traitement en fonction de la taille de l'image
rotate
Durée (en s)
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0
1
2
3
4
5
6
Taille (en Mpixel)