Pasquier Romain

Titre : Chiffrement et décryptage d'images.

Ancrage au thème de l'année : Le chiffrement permet d'envoyer en toute sécurité des documents aux destinataires qui possède la méthode de déchiffrement. Ainsi les messages envoyés par des sociétés à des employés ne peuvent pas être lu mais s'ils sont interceptés lors du transfert.

Motivation du choix de l'étude : Etant en spé info j'étais motivé par le fait de me lancer dans un réel « projet » informatique et de créer un programme utile et amusant. J'étais intéressé par le fait d'optimiser mon programme et de trouver les méthodes les plus rapides en relation avec la bibliothèque utilisée.

Positions thématiques : informatique, maths (c'est ça les thématiques ou bien : informatique, sécurité ?).

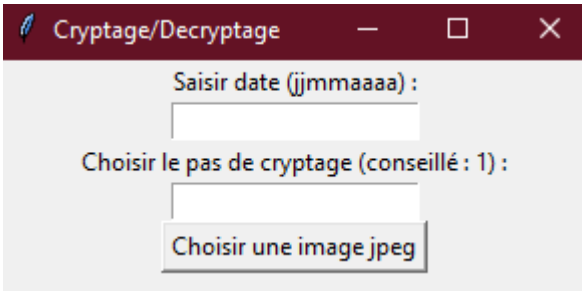
Mots clefs : Sécurité, Chiffrement, décryptage, Arithmétique, clef.

Problématique : Pour transmettre des documents entre plusieurs sociétés, j'utilise un chiffrement de données. Pour répondre au plus de contraintes, je crypte des images (scans ou photos) car elles peuvent contenir du texte, des plans, des schémas ou encore des images… Je peux crypter et décrypter en couleurs.

Comment transmettre en toute sécurité le plus de types de documents à des entreprises alliées ?

Objectif :

Il s'agit de créer une application (avec tkinter) qui permet de crypter et décrypter une image à l'aide d'une clef qui permettra de parcourir une matrice 10x10 donnée par l'envoyeur.



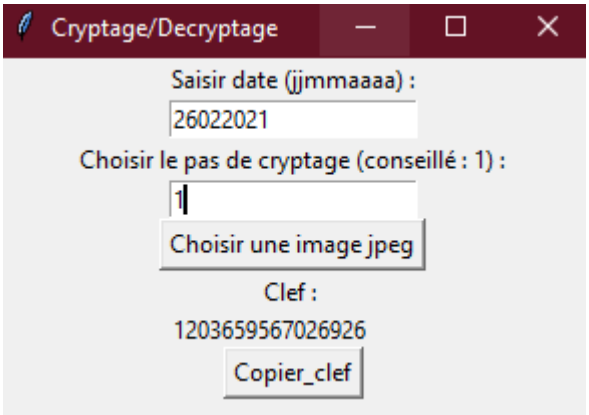


Image source

Image décryptée

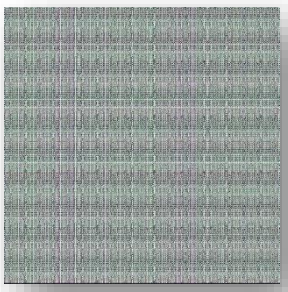


Image cryptée

Méthode 1

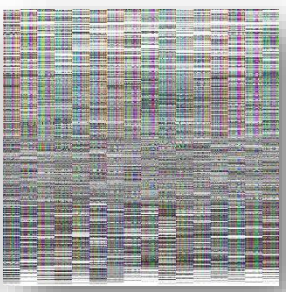


Image cryptée

Méthode 2

Explications du code :

Pour répondre au besoin des entreprises : l'objectif est de transférer en toute sécurité des documents (textes, code, image, schéma …).

Pour cela on cherche à chiffrer une image (scans, photos …) puis à la décrypter grâce à une clef et une matrice que l'envoyeur et le destinataire s'échangent.

On utilise pour cela le langage python en important la bibliothèque PIL.

Principe du code :

- On part d'une image de taille (X, Y) où X et Y sont des nombres premiers (Pour s'assurer de cela on recadre d'image). On créer deux images en parallèle : une image « fond_crypte » et « fond_decrypte » totalement noires aux mêmes dimensions sur laquelle on collera les pixels « déplacés ».

- La clef est composée de 16 chiffres désignés par l'envoyeur (généré aléatoirement) (par exemple : 7579613548894536) et de 8 chiffres qui correspondent à la date d'envoi du message (par exemple : 18/06/2020. Soir la longueur de la liste est : L = 24.

Grâce aux 8 derniers chiffres, on se déplace dans la clef de la manière suivante (avec l'exemple précédent) :

ax = clef_a_la_position [1 mod (L - 8)] = 5 (car en python les positions vont de 0 à ...)

ay = clef_a_la_position [8 mod (L - 8)] = 4

bx = …

by = …

cx = …

cy = …

dx = …

dy = …

De telle sorte que [|ax … dy|] soient compris entre [0, 10].

- La matrice « X » est de taille (10, 10) et contient des valeurs aléatoires comprises entre 1 et le maximum de (X, Y) (pour ne pas sortir de l'image).

- Ainsi grâce à la clef et à la matrice X, on peut sortir 4 nombres :

a = X_a_la_position [ax][ay]

b = …

c = …

d = …

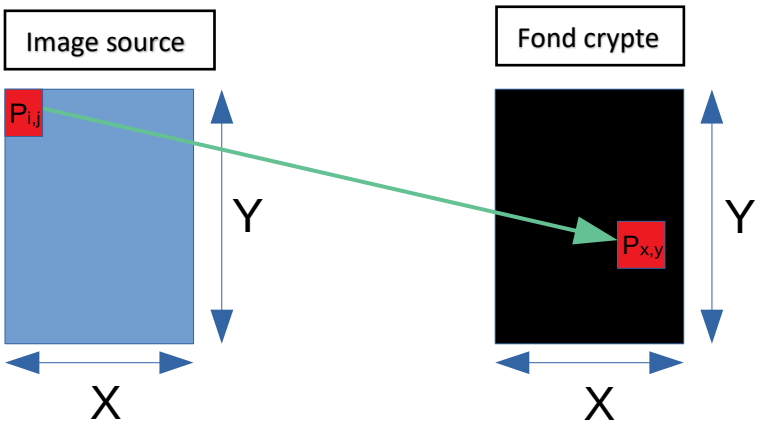
- <u>Le cryptage (affine) :</u>

On prend i ϵ]0, X] et j ϵ]0, Y] tel que :

$x = (a * i + b) \bmod(X)$

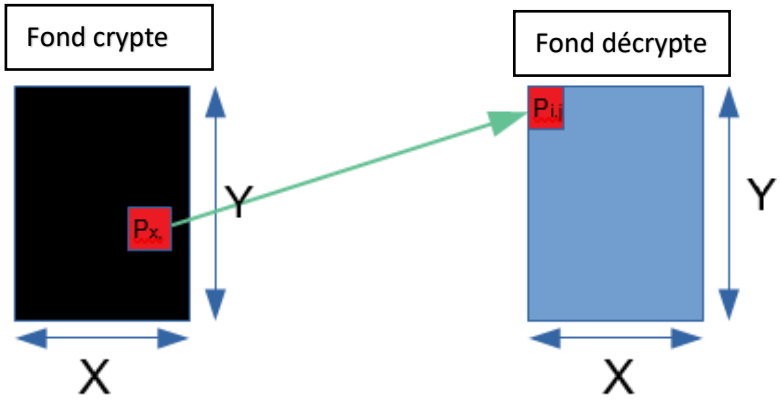
$y = (c * j + d) \bmod(Y)$

Ainsi avec P un carré de pixel de pas prédéfinie, on a :



- <u>Le décryptage (affine) :</u>

<u>Astuce :</u> Dans la théorie on aurait résolu grâce à Euclide une équation diophantienne, mais dans la pratique, je me suis rendu compte qu'on pouvait simplement tourner les choses de la manière suivante : (car i et j tournent en boucle for l'une dans l'autre).




```python
def cryptage(img):
    x_img, y_img = img.size

    taille_x = dimensions(y_img // pas) + 1
    taille_y = dimensions(x_img // pas) + 1

    grille_4, grille_2 = quadrillage(img)
    grille_crypte4, grille_crypte2 = zeros(taille_x, taille_y)

    for i in range(1, len(grille_2)):
        for j in range(1, len(grille_2[0])):
            x = (a * i + b) % (len(
                grille_2) - 1)  # car sinon ce n'est plus un no
            y = (c * j + d) % (len(grille_2[0]) - 1)  # idem

            grille_crypte4[x + 1][y + 1] = grille_4[i][j]
            grille_crypte2[x + 1][y + 1] = grille_2[i][j]

    return grille_crypte4, grille_crypte2
```

```python
def decryptage(img):
    x_img, y_img = img.size

    taille_x = dimensions(y_img // pas) + 1
    taille_y = dimensions(x_img // pas) + 1

    grille_crypte4, grille_crypte2 = cryptage(img)
    grille_decrypte4, grille_decrypte2 = zeros(taille_x, taille_y)

    for i in range(1, len(grille_crypte2)):
        for j in range(1, len(grille_crypte2[0])):
            x = (a * i + b) % (len(grille_crypte2) - 1)  # car sinon
            y = (c * j + d) % (len(grille_crypte2[0]) - 1)  # idem

            grille_decrypte2[i][j] = grille_crypte2[x + 1][y + 1]

    return grille_decrypte4, grille_decrypte2
```

<u>Comment coder tout ça :</u>

Pour commencer pour tout ce qui s'agit des clefs et de la matrice, pas besoin de rentrer dans les détails, il s'agit simplement de parcourir des listes et des matrices (ultra classique).

Par contre pour gérer l'image, à la place de coder les pixels directement (avec img.getpixel car sinon « i » et « j » peuvent valoir 0 et dans ce cas notre fonction n'est plus bijective et des pixels se superposent) on créer donc un tableau qui représentera la position de nos « carrés de pixels » par exemple un « carré de pixels » peut être un pixel (ce qu'on utilisera le plus fréquemment) ou alors un

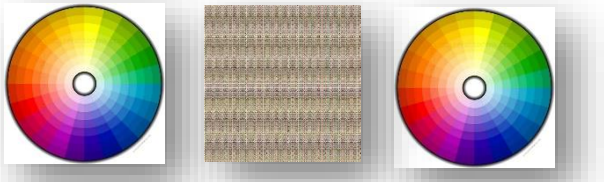
carré de 10x10 pixels. De ce tableau on bougera les couples de valeurs en parcourant cette matrice. Ainsi la première et colonne et première ligne doivent être rempli de 0 et transposé le reste pour avoir une fonction bijective en faisant commencer les indices i et j à 1 et non à 0 tel que pour une image carrée (11x11) et un pas = 1 on a :

```
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0), (8, 0), (9, 0), (10, 0)]
[(0, 0), (0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 1), (10, 1)]
[(0, 0), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 2), (10, 2)]
[(0, 0), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (10, 3)]
[(0, 0), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4), (7, 4), (8, 4), (9, 4), (10, 4)]
[(0, 0), (0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5), (7, 5), (8, 5), (9, 5), (10, 5)]
[(0, 0), (0, 6), (1, 6), (2, 6), (3, 6), (4, 6), (5, 6), (6, 6), (7, 6), (8, 6), (9, 6), (10, 6)]
[(0, 0), (0, 7), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7), (8, 7), (9, 7), (10, 7)]
[(0, 0), (0, 8), (1, 8), (2, 8), (3, 8), (4, 8), (5, 8), (6, 8), (7, 8), (8, 8), (9, 8), (10, 8)]
[(0, 0), (0, 9), (1, 9), (2, 9), (3, 9), (4, 9), (5, 9), (6, 9), (7, 9), (8, 9), (9, 9), (10, 9)]
[(0, 0), (0, 10), (1, 10), (2, 10), (3, 10), (4, 10), (5, 10), (6, 10), (7, 10), (8, 10), (9, 10), (10, 10)]
```

On « crypte » alors notre tableau a l'aide de la méthode vu précédemment pour obtenir avec (a = 4, b = 8, c = 9, d = 1) :

```
[(0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0)]
[(0, 0), (5, 8), (10, 8), (4, 8), (9, 8), (3, 8), (8, 8), (2, 8), (7, 8), (1, 8), (6, 8), (0, 8)]
[(0, 0), (5, 0), (10, 0), (4, 0), (9, 0), (3, 0), (8, 0), (2, 0), (7, 0), (1, 0), (6, 0), (0, 0)]
[(0, 0), (5, 3), (10, 3), (4, 3), (9, 3), (3, 3), (8, 3), (2, 3), (7, 3), (1, 3), (6, 3), (0, 3)]
[(0, 0), (5, 6), (10, 6), (4, 6), (9, 6), (3, 6), (8, 6), (2, 6), (7, 6), (1, 6), (6, 6), (0, 6)]
[(0, 0), (5, 9), (10, 9), (4, 9), (9, 9), (3, 9), (8, 9), (2, 9), (7, 9), (1, 9), (6, 9), (0, 9)]
[(0, 0), (5, 1), (10, 1), (4, 1), (9, 1), (3, 1), (8, 1), (2, 1), (7, 1), (1, 1), (6, 1), (0, 1)]
[(0, 0), (5, 4), (10, 4), (4, 4), (9, 4), (3, 4), (8, 4), (2, 4), (7, 4), (1, 4), (6, 4), (0, 4)]
[(0, 0), (5, 7), (10, 7), (4, 7), (9, 7), (3, 7), (8, 7), (2, 7), (7, 7), (1, 7), (6, 7), (0, 7)]
[(0, 0), (5, 10), (10, 10), (4, 10), (9, 10), (3, 10), (8, 10), (2, 10), (7, 10), (1, 10), (6, 10), (0, 10)]
[(0, 0), (5, 2), (10, 2), (4, 2), (9, 2), (3, 2), (8, 2), (2, 2), (7, 2), (1, 2), (6, 2), (0, 2)]
[(0, 0), (5, 5), (10, 5), (4, 5), (9, 5), (3, 5), (8, 5), (2, 5), (7, 5), (1, 5), (6, 5), (0, 5)]
```

Enfin on découpe nos carrés de pixels grâce à ces tableaux et à la fonction img.crop pour les coller sur les « fonds ».

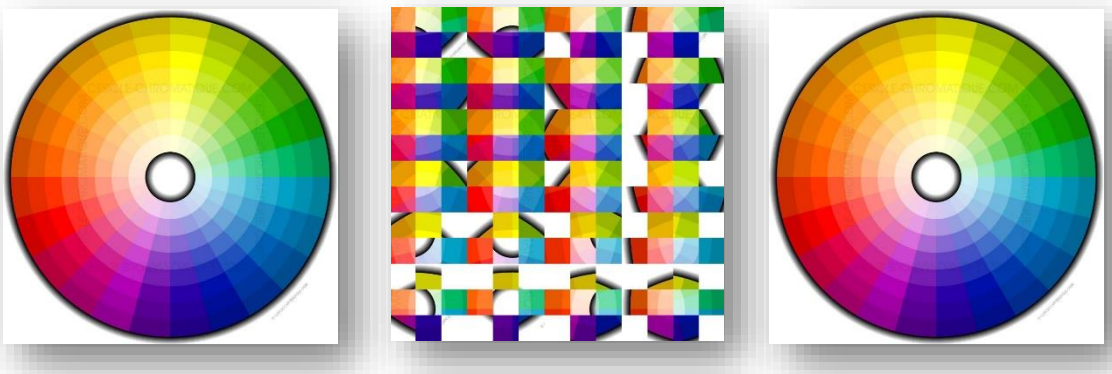


Complexité :

Complexité quadratique car on se déplace dans des tableaux. Il ne me semble pas possible avec cette méthode de faire mieux à ma connaissance.

Points forts du code :

- Assez rapide dans l'exécution, et quasiment sans perte d'information (dut à la forme des p).
- La méthode n'est pas si difficile à comprendre et à mettre en place et donne un rendu plus que correct et très visuel. En effet en augmentant le pas sur la première image on remarque que en réalité on peut résoudre à la main ce « puzzle » :



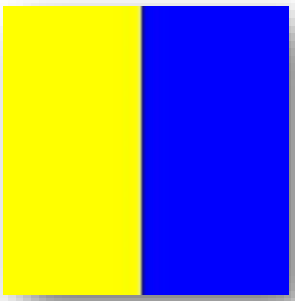
- Impossible à craquer par force brut :
  En effet si on fait un rapide calcul, on a :
  24 caractères dans la clef et on extrait 8 chiffres de cette clef puis 4 chiffres de la matrice W (10x10) et enfin la matrice W contient des valeurs aléatoires comprises entre 1 et 1920 (min d'une image en full HD 1920x1080) :

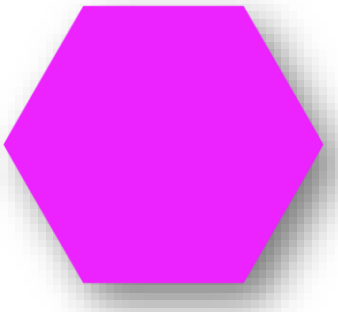
  Nombre d'opérations à effectuer = $10^{24} * \binom{24}{8} * \binom{100}{4} * 10^{1080} = 3.10^{1956}$
  Or l'ordinateur le plus puissant est le Fugaku 415-PFLOPS qui effectue en théorie 513 quadrillons de calculs par seconde.

  Donc il effectue 10^{15} calculs par seconde. Soit il craquera notre code en 3.10^{1941}secondes qui équivaut à : 9.10^{1933} années donc 9.10^{1924} $milliards\ d'années$ pour comparer l'âge de l'univers est à peu près de 15 milliards d'années.

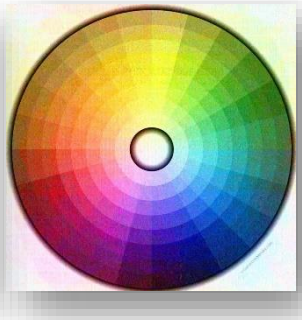
Limites du code :

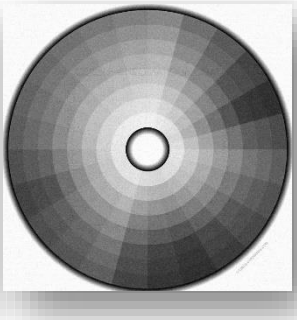
- Limite de la bibliothèque (fixé) : dans mon code je sauvegarde les images cryptées et décrypter donc je sauvegarde les fonds sur lequel les pixels ont été coller. Mais je me suis rendu compte que la sauvegarde et le copier-coller de PIL n'était pas parfait. En effet en créant une image à partir de la bibliothèque et en collant sur toute l'image des pixels de jaune pure puis en faisant de même avec du bleu et en les collant l'une à côté que en sauvegardant l'image il y avait des imperfections. Cela est dut à la forme d'un pixel qui est hexagoanl. En effet pour sauvegarder une image pour ce type de programme, il faut utiliser le format .bmp : c'est un fichier bitmap, c'est-à-dire un fichier d'image graphique stockant les pixels sous forme de tableau de points et gérant les couleurs soit en couleur vraie soit grâce à une palette indexée.



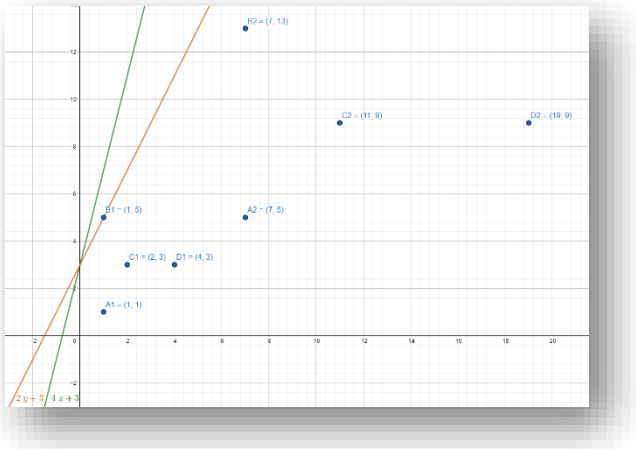


Donc si sur mon code je cryptais l'image sauvegardé et non le fond crypté directement j'obtiendrai un étalement des couleurs (qui ne se verrait pas en niveaux de gris, car au niveaux de gris on fait une moyenne) :





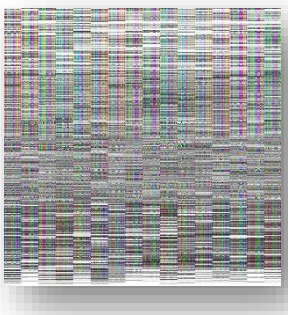
- Enfin on remarque que la méthode « affine » est en fait simple à craquer par l'intelligence. En effet on remarque que notre méthode change la ligne et la colonne des pixels, mais les pixels originalement sur la même ligne et même colonne se retrouve encore sur la même colonne à une ligne différente et inversement pour la ligne. Cela est dut à :



On observe donc des motifs qui se répètent sur l'image cryptée.

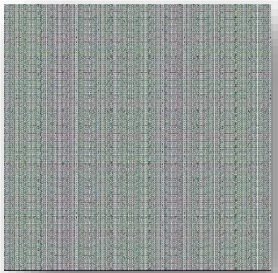
Solutions pour rendre plus délicat le décryptage intelligent :

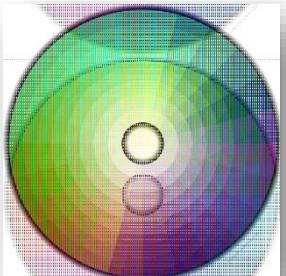
- Première idée ne pas utiliser le codage affine et coder par ligne récursivement en faisant des paquets de 16 pixels et en les permutants. Méthode efficace mais la clef n'est pas dure à trouver (elle ne contient que le nombre de fois ou l'image est coupé en rectangles puis dans chaque rectangle on permute en rectangle deux fois plus petit…) donc cassable par la force brut (d'autant plus que les actions récursives se voient sur chaque ligne). Ce qui donne avec l'idée suivante implémentée :



- Deuxième idée : avec les chiffres non utilisés de la clef originale, changer les coefficients des couleurs RGB en faisant des rotations de R, G et B suivant leurs positions sur la ligne dans l'image tel qu'avec mon code affine on a (ce qui augmente encore le nombre de calculs on remplace $\binom{24}{8}$ par : $\binom{24}{12}$) :

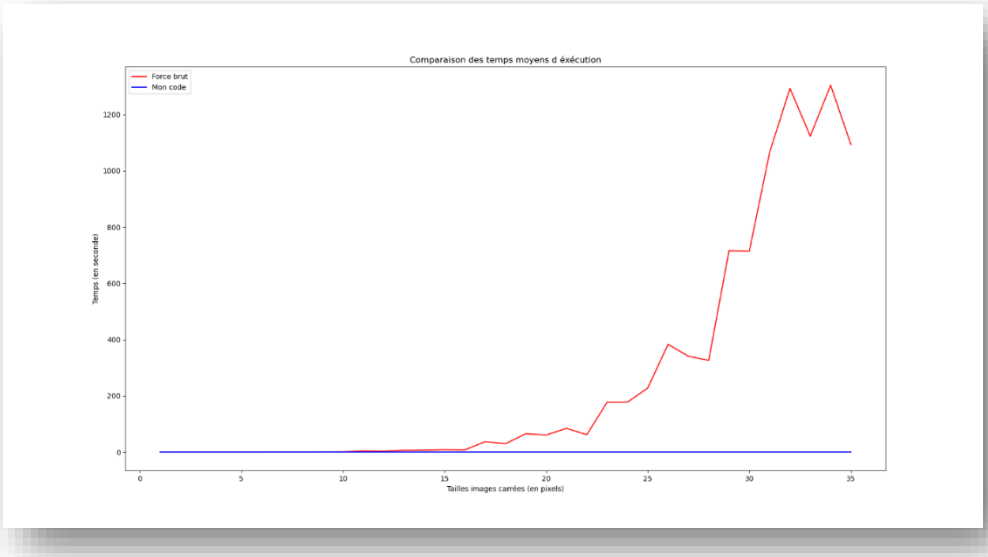
Si on ne décrypte pas RGB





Comparaison des temps moyens :

- En créant un code force brut qui créer toutes les possibilités des combinaisons des coefficients (a, b, c, d) (par une méthode de coefficients aléatoire ça aurait été trop long et les graphes observés actuellement sont quand même cohérents) et qui vérifie pour chaque combinaison si l'image en entier et la même que l'originale, on obtient des temps déjà très important pour des images 35x35 (au-delà faire tourner le code est très long d'autant plus que pour chaque taille, on a effectué 100 moyennes. Les mesures non cohérentes sont dues au fait que l'on parcourt une liste contenant toutes les possibilités, au plus l'image est grande au plus la liste sera grande. Sur des images 10x10 ce n'est pas un problème, mais au-delà d'un seuil la liste devient plus grande d'ordre $10^{\cdots}$).



- Enfin pour vraiment comparer, regardons sur des images 500x500 avec mon code, le temps d'exécution (moyenne de 500) (échelle log en y) :