Stéganographie élémentaire

1 De l'art de dissimuler des messages

La confidentialité des échanges entre deux individus a été est restera toujours une problèmétique essentielle pour l'espèce humaine. Depuis que l'écriture existe, on a toujours cherché des méthodes pour pouvoir envoyer des messages à un interlocuteur sans qu'un indiscret ne puisse prendre connaissance du contenu. Et bien entendu, des indiscrets ont toujours cherché à récupérer le contenu d'échanges dont ils ne sont pas en théorie autorisés à disposer. On considère donc dans toutes les techniques de transmission sécurisée la situation suivante : une personne A (en général dénommée Alice) cherche à transmettre un message à une personne B (généralement prénommée Bob). On part alors du principe que le message est susceptible d'être intercepté avec probabilité non nulle (parfois même sans que ni Alice, ni Bob ne s'en rende compte) et on veut s'assurer que l'information contenue dans le message reste confidentielle. Au passage, le vilain curieux s'appelle souvent E ou (Eve) tandis que les lettres C et D sont attribuées à d'autres rôles pour des raisons qui m'échappent un peu.

La technique la plus utilisée reste celle de la cryptographie : on code le message suivant une technique qui fait en sorte que le message transmis est totalement inintelligible, et que la méthode pour reconstituer le message original ne peut être effectuée que par Bob lui-même. En pratique, les techniques de codage sont toutes universellement connues, et il est donc la plupart du temps possible de créer une méthode de décodage même si on ne s'appelle pas Bob. La subtilité réside alors dans le fait que Bob dispose en général d'une clé utilisée pour coder, laquelle permet de décoder rapidement, tandis qu'un indiscret ne disposant pas de cette clé va certes décoder le texte, mais en un temps infiniment plus long.

L'objectif de ce TP est de présenter une seconde méthode de transmission confidentielle d'information. Là où les méthodes de cryptographie partent du principe que le contenu transmis doit être complètement inintelligible pour Eve, la stéganographie est une technique dont l'objectif est de transmettre un message parfaitement intelligible, d'aspect totalement anodin, mais qui contient en fait (dissimulé bien entendu) le message secret. Là où un message codé attirerait forcément l'attention, l'objectif est d'espérer qu'un tel message passerai totalement inaperçu, et donc son contenu également. A titre d'exemple, on notera ce grand classique d'échange savoureux attribué comme une correspondance entre Georges Sand et Alfred de Musset :

> Je suis très émue de vous dire que j'ai bien compris l'autre soir que vous aviez toujours une envie folle de me faire danser. Je garde le souvenir de votre baiser et je voudrais bien que ce soit là une preuve que je puisse être aimée par vous. Je suis prête à vous montrer mon affection toute désintéressée et sans calcul, et si vous voulez me voir aussi vous dévoiler sans artifice mon âme toute nue, venez me faire une visite. Nous causerons en amis, franchement.
>
> Je vous prouverai que je suis la femme sincère, capable de vous offrir l'affection la plus profonde comme la plus étroite en amitié, en un mot la meilleure preuve dont vous puissiez rêver, puisque votre âme est libre. Pensez que la solitude où j'habite est bien longue, bien dure et souvent difficile. Ainsi en y songeant j'ai l'âme grosse. Accourrez donc vite et venez me la faire oublier par l'amour où je veux me mettre.

La présentation est bien entendue essentielle, et le texte révèle tous ses secrets en lisant une ligne sur deux ...

La méthode utilisée dans ce TP va consister à utiliser des méthodes de dissimulations d'informations au sein d'image dont le format n'utilise pas de compression destructive (pour des raisons qui seront détaillées à la fin de cet énoncé). Les fichiers images que nous allons utiliser dans ce TP sont au format png (Portable Network Graphics), utilisant le système RBG qui permet le codage de 16 millions de couleurs à l'aide de trois entiers compris entre 0 et 255. A chaque pixel de l'image, on associe un triplet (r, g, b) donnant pour le premier la « teneur » en rouge, pour le second celle en vert et le dernier celle en bleu. Par conséquent, le triplet $(255, 0, 0)$ indique un rouge pétant, le triplet $(0, 255, 0)$ un vert plutôt flashy, et $(0, 0, 255)$ un bleu plutôt sombre. Le noir est codé par $(0, 0, 0)$ et le blanc par $(255, 255, 255)$. L'objet que l'on manipule en python s'apparente donc à une matrice de triplets d'entiers, dont on peut modifier chaque valeur.

2 Décomposition des entiers en base 2

On rappelle le résultat suivant : tout entier $n \in \mathbb{N}$ admet une décomposition unique sous la forme

$$n = \sum_{k=0}^{+\infty} \epsilon_k \, 2^k \qquad \text{avec} \qquad \forall k \in \mathbb{N}, \quad \epsilon_k \in \{0,1\} \quad \text{et} \quad (\epsilon_k)_{k\in\mathbb{N}} \text{ nulle a.p.c.r.}$$

En d'autre termes, n se décompose de manière unique comme somme de puissance de 2 distinctes. Cette décomposition s'appelle l'écriture en base 2 de l'entier 2. A titre d'exemple,

$$3 = 1 + 2 = 1 \cdot 2^0 + 1 \cdot 2^1 \qquad \text{et} \qquad 21 = 1 + 4 + 16 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4$$

Dans l'optique de ce TP, nous ne manipulerons que des entiers compris entre 0 et 255. Par conséquent, l'écriture en base 2 est toujours nulle à partir du rang 8. On se contente donc de calculer les 8 premiers termes. La méthode la plus efficace consiste à procéder par division euclidienne successive par 2 :

- On pose $q_0 = n$. Alors, ϵ_0 est le reste de la division euclidienne de q_0 par 2. Le quotient de cette division est noté q_1.
- Plus généralement, pour tout entier $i \in \mathbb{N}$, on récupère la valeur de ϵ_i par le reste de la division euclidienne de q_i par 2, avant de poser q_{i+1} comme le quotient de cette même division.

Question 1

Ecrire une fonction `int_to_8bits` qui prend en argument un entier n compris entre 0 et 255 et renvoie la liste $[\epsilon_0, \epsilon_1, \ldots, \epsilon_7]$ des 8 premiers bits de sa décomposition en base 2. Attention à l'ordre utilisé !

Question 2

Ecrire la fonction réciproque `bits_to_int` qui recalcule n à partir de la liste de ses bits.
Pour minimiser le nombre de multiplications effectuées, on pourra utiliser la méthode de Hörner qui consiste à remarquer que

$$\epsilon_0 + \epsilon_1 \cdot 2 + \cdots + \epsilon_p \cdot 2^p = \epsilon_0 + 2\left(\epsilon_1 + 2\left(\cdots\left(\epsilon_{p-1} + 2 \cdot \epsilon_p\right)\cdots\right)\right)$$

et à calculer donc successivement ϵ_p, puis $\epsilon_{p-1} + 2\epsilon_p$, puis $\epsilon_{p-2} + 2(\epsilon_{p-1} + 2\epsilon_p)$, etc ... ce qui ne nécessite que $p - 1$ multiplications en tout.

3 Stéganographie en deux dimensions

3.1 Dissumulation et extraction d'un entier

Le principe de dissimulation utilisé pour cacher une image dans une autre est basée sur le principe des bits de poids fort et des bits de poids faible. Considérons un entier n compris entre 0 et 255 comme une unité de couleur de l'image, dont la décomposition en base 2 est

$$[\epsilon_0, \epsilon_1, \ldots, \epsilon_7]$$

Si l'on modifie la valeur de ϵ_k, la valeur de n est modifiée de 2^k. Par conséquent, modifier les bits ϵ_0, ϵ_1 ou ϵ_2 (dit de poids faible) n'a que peu d'incidences sur la valeur de n, tandis qu'au contraire, modifier les derniers bits ϵ_6 ou ϵ_7 (dits de poids fort) a une incidence énorme sur la valeur de n. De manière similaire, on peut considérer que l'on peut se contenter de retenir les valeurs des 3 derniers bits pour avoir une valeur assez proche de n.

Pour « dissimuler » un entier $x \in [\![0; 255]\!]$ dans un entier $y \in [\![0; 255]\!]$, une méthode simple consiste donc à remplacer quelques bits de poids faible de y par quelques bits de poids fort de x. On modifie donc légèrement la valeur de y, et on ne peut récupèrer qu'une valeur approchée de x, mais cela suffit largement pour la suite. Le nombre de bits à utiliser ne doit être ni trop grand (pour ne pas trop modifier y), ni trop petit (pour ne pas récupérer une valeur trop éloignée de x). J'ai personnellement utilisé la valeur $p = 3$ pour mes fichiers (voir partie 4).

Question 3

Ecrire une fonction `cache_dans_int` qui prend en argument deux entiers x et y compris entre 0 et 255, et un entier $p \in [\![1; 8]\!]$. Si l'on note $[\epsilon_0, \epsilon_1, \ldots \epsilon_7]$ la liste des 8 premiers bits de x et $[\mu_0, \ldots, \mu_7]$ celle de y, la fonction renvoie l'entier z dont l'écriture en base 2 est

$$[\epsilon_{8-p}, \ldots, \epsilon_7, \mu_p, \ldots \mu_7]$$

En d'autres termes, elle remplace les p premiers bits de l'écriture de y par les p derniers bits de x et renvoie **l'entier** représenté par le résultat.

Pour récupérer la valeur de x, on ne peut donc récupérer que les p bits de poids fort. Puisque l'on ne peut connaître la valeur des bits de poids faible, on les remplace par des 0.

Question 4

Ecrire une fonction `extrait` qui prend en argument un entier $z \in [\![0;255]\!]$ et un entier $p \in [\![1;8]\!]$. Si z admet pour écriture $[z_0, \dots, z_7]$, alors la fonction renvoie l'entier $\widetilde{x}$ dont l'écriture en base 2 est

$$[0, \dots, 0, z_0, \dots z_{p-1}]$$

En d'autres termes, elle récupère les p premiers bits de l'écriture de z, ajoute des 0 en tête de liste jusqu'à obtenir une liste de longueur 8, et renvoie l'entier représenté par le résultat.

3.2 Manipulation rudimentaire d'image sous python

Dans un premier temps, il convient d'importer une partie du module PIL (Python Imaging Library).

```
from PIL import Image
```

Les méthodes élémentaires d'importation/exportation et de manipulations d'images sont les suivantes :

- `Image.open` prend en argument un nom de fichier (avec adresse éventuelle) et renvoie un objet du type Image. Pour un fichier en png, cela s'apparente à une matrice de triplet (ou de quadruplet si l'image utilise la notion de transparence, ce qui n'est pas le cas de `originale.png`). Si le fichier est à la racine du compte, il suffit donc d'écrire

  ```
  image1 = Image.open("originale.png")
  ```

  S'il est sur le bureau (c'est-à-dire en pratique dans le répertoire `Bureau` situé à la racine), on écrira

  ```
  image1 = Image.open("Bureau/originale.png")
  ```

  Si vous avez une erreur du type « `could not find` », essayez de remplacer `Bureau` par son nom anglais `Desktop` ou déplacer le fichier à la racine de votre compte ou du compte Joffre. Notez qu'on peut créer plusieurs objets de type Image à partir du même fichier png :

  ```
  image2 = Image.open("Bureau/originale.png")
  ```

  et modifier ensuite `image1` et `image2` séparemment, ce qui n'aura aucune incidence sur le fichier `originale.png`.
- La commande `show` permet d'obtenir un aperçu par Python de l'objet de type Image défini en python. Elle ouvre donc une fenêtre en affichant l'image. Si votre fichier s'appelle `image1`, il suffit donc d'exécuter

  ```
  image1.show()
  ```
- La taille de l'image (nombre de lignes/colonnes) s'obtient par la commande `size`.

  ```
  l,h = image1.size
  ```

  Les fonctions `getpixel` et `putpixel` permettent respectivement de récupérer et modifier le triplet d'une composante RGB du pixel d'indices $(i,j) \in [\![0;l-1]\!] \times [\![0;h-1]\!]$. Si l'objet image Python s'appelle `image1`, il suffit d'écrire

  ```
  image1.getpixel((0,0))
  ```

  pour récupérer les trois valeurs du pixel d'indice $(0,0)$, et pour les modifier, d'écrire par exemple

  ```
  image1.putpixel((0,0),(0,255,0))
  ```
- `save` crée un fichier image à partir de l'objet du type Image qu'on lui donne en argument. Le fichier est placé à la racine du compte, faute de préciser un chemin où le déposer. On écrira par exemple

  ```
  image1.save("Images/modifiee.png")
  ```

Question 5

Importer l'image `originale.png`, puis modifier le fichier de type `Image` ainsi crée pour rajouter un carré de vert et un rectangle de rouge n'importe où dans l'image (suffisamment gros pour être vu à l'oeil toutefois). Admirez le résultat avec la commande `show` puis tester la sauvegarde de l'image ainsi modifiée.

3.3 Dissimulation et extraction d'images

Le principe pour dissimuler une image `secret` dans une image `originale` est alors extrêmement simple : pour chaque pixel d'indice (i, j) de `originale`, dont la couleur est donnée par le triplet (r_o, g_o, b_o), on récupère le triplet de couleur (r_s, g_s, b_s) du pixel d'indice (i, j) de `secret` et on « cache » r_s dans r_o, g_s dans g_o et b_s dans b_o. Les modifications sont alors en général invisibles à l'oeil nu (du moins lorsque p n'est pas trop grand). On récupère l'image secrète de manière similaire à l'aide de la fonction d'extraction. L'image obtenue n'est pas une copie exacte de l'image `secret` initiale, mais les différences sont encore une fois relativement invisibles à l'oeil nu.

Question 6

Ecrire une fonction `extrait_image` qui prend en argument deux fichiers `sortie` et `originale` et un entier p. Pour chaque pixel `originale[i,j]`, la fonction récupère l'entier caché dans les p premiers bits de chaque composante du triplet, et recopie le résultat dans le fichier `sortie`.
Utilisez cette fonction pour découvrir l'image cachée dans le fichier `originale_et_secret_2025_2026.png`.

Question 7

Ecrire une fonction `cache_dans_image` prenant en argument deux fichiers images `originale` et `secret` et un entier $p \in [\![1; 8]\!]$. La fonction modifie chaque pixel de l'image `originale` en cachant les p premiers bits de chaque composante du triplet `secret[i,j]` dans la composante correspondante du triplet `originale[i,j]`.

Note : A priori, il est rare de tomber sur des images ayant exactement la même taille. Ajustez vos fonctions en conséquence, quitte à n'insérer qu'une partie de `secret`.

4 Quelques remarques pour finir : compression destructive et stérilisation

A partir du moment où le fichier qui contient l'image dissimule son contenu secret au niveau des bits de poids faible, il est clair que toute modification de ceux-ci fait disparaître le message dissimulé. Or, la plupart des formats d'image utilisé à l'heure actuelle sont munis d'un système de compression. En effet, un entier compris entre 0 et 255 est codé en base 2 par 8 bits, soit un octet. Un triplet est donc codé par 3 octets et une image brute de taille 800×600 par 1 440 000 octets soit environ 1.37 Mo. C'est bien trop gros pour une simple image, d'où la nécessité de compresser le fichier.

Il existe de nombreux algorithmes de compressions de données. Sans rentrer dans les détails, étant donné un objet `O1` (texte, son, image), on produit un objet `O2` et l'algorithme doit être conçu pour que `O2` soit de taille inférieure à `O1`. On peut les décomposer en deux grandes catégories :

- la compression non destructive (ou sans perte) : l'application `O1 -> O2` est injective, et si l'on dispose de l'algorithme implémentant la réciproque `O2 -> O1`.
- la compression destructive (ou avec perte) : l'application `O1 -> O2` n'est pas injective. On dispose cependant d'une pseudo-réciproque qui à partir de `O2` reproduit un objet `O3` relativement proche de `O1` (du moins à l'oeil ou à l'oreille humaine). Cet outil est fréquemment utilisé pour la transmission de données (MP3 pour le son, JPEG pour l'image) lorsque l'on accepte de rogner légèrement la qualité pour gagner considérablement en efficacité sur la compression. Le format JPEG permet des taux de compression pouvant aller jusqu'à une taille divisée par 100.

Le problème principal d'un format d'image utilisant une compression destructive est donc qu'il modifie le fichier dans lequel on cache notre information, qui est donc en général détruite et irrécupérable (on parle alors de stérilisation du fichier). C'est la raison pour laquelle certaines entreprises imposent ce format d'image dans les échanges de leurs employés pour éviter la transmission d'informations confidentielle.

5 Au cas où PIL ne fonctionne pas (machines persos) ...

Le module que j'ai utilisé pour manipuler les images sur mon ancien ordinateur personnel s'appelle `imageio`. Aux dernières nouvelles, il n'est pas installé sur les machines du lycée, contrairement à PIL, qui à l'époque ne fonctionnait pas chez moi. Le paragraphe ci-dessous est destinés à ceux qui, comme moi, n'arrivent pas à utiliser PIL sur leur machine personnelle (en espérant que `imageio` fonctionne lui). L'importation se fait via la commande

```
import imageio
```

Les méthodes élémentaires d'importation/exportation et de manipulations d'images sont cette fois les suivantes :

- `imageio.imread` pour ouvrir les fichiers d'image. Si le fichier est à la racine du compte, il suffit donc d'écrire

  ```
  image1 = imageio.imread("originale.png")
  ```

  S'il est dans le répertoire `Bureau` (ou `Desktop`) situé à la racine, on écrira

  ```
  image1 = imageio.imread("Bureau/originale.png")
  ```

- `imageio.imsave` réalise l'opération inverse : elle crée un fichier image à partir de l'objet du type Image qu'on lui donne en argument. Le fichier est placé à la racine du compte, faute de préciser un chemin où le déposer. On écrira par exemple

  ```
  imageio.imsave("Bureau/modifiee.png",image1)
  ```

  si on veut que la nouvelle image soit déposée dans le même répertoire que la première.

- La commande `imshow` du module `matplotlib` permet d'obtenir un aperçu par Python de l'objet de type Image défini en python. Elle ouvre donc une fenêtre en affichant l'image.

  ```
  import matplotlib.pyplot as plt
  plt.imshow(image1)
  ```

- La taille de l'image s'obtient par le champ `shape`. Si le fichier s'appelle `image1`, on pourra donc écrire

  ```
  (l,h,c) = image1.shape
  ```

  Alors, `l` est égal au nombre de ligne de l'image, `h` au nombre de colonnes, et `c` au nombre de composantes (3 pour du RGB, 4 s'il y a en plus une composante de transparence (dont l'utilité m'échappe un peu).

- Pour finir, on peut modifier directement la valeur d'un pixel ou d'une composante RGB d'un pixel. Si l'objet image Python s'appelle `image1`, il suffit à ce moment là d'écrire

  ```
  image1[0,0]=(255,0,0)
  ```

  pour remplacer la valeur du triplet par $(255, 0, 0)$, ou si on veut travailler composante par composante

  ```
  image1[0,0,0]=255; image1[0,0,1]=0; image1[0,0,2]=0
  ```