HAI918I tp1

Houle Adrien

Septembre 2023

L'objectif de cette première partie est de chiffrer par permutation une image en niveau de gris (format pgm) à partir d'une clé secrète K de 8 bits et d'obtenir une image chiffrée en niveau de gris (format pgm) de la même taille que l'image originale. Le chiffrement par permutation permet d'introduire de la diffusion dans l'image chiffrée.

a)

Ecrire un programme permutation.cpp ayant comme arguments d'entrée, le nom de l'image à chiffrer, la valeur de la clé K ainsi que le nom souhaité pour l'image chiffrée. Il est fortement conseillé d'utiliser une carte des positions déjà utilisées dans l'image chiffrée afin de ne pas écraser des valeurs de pixels.

Dans le code suivant, j'effectue la permutation des pixels de manières aléatoire en utilisant, comme recommandé, une carte des positions utilisées. La clé utilisée est 128.

```cpp
std::list<int> unusedPositions; // Créez une liste pour stocker les positions non utilisées
for (int i = 0; i < nTaille; ++i)
    unusedPositions.push_back(i);

int *permutedPixels = new int[nTaille];

for (int i = 0; i < nTaille; ++i)
{
    // Choisir une position aléatoire parmi les non utilisées
    int randomPos = rand() % unusedPositions.size();

    auto it = std::next(unusedPositions.begin(), randomPos);
    int selectedPos = *it;

    permutedPixels[selectedPos] = ImgIn[i];

    unusedPositions.erase(it);
}

// Copiez les valeurs modifiées dans l'image de sortie
for (int i = 0; i < nTaille; i++)
{
    ImgOut[i] = permutedPixels[i];
}
```

Figure 1: Code de permutation

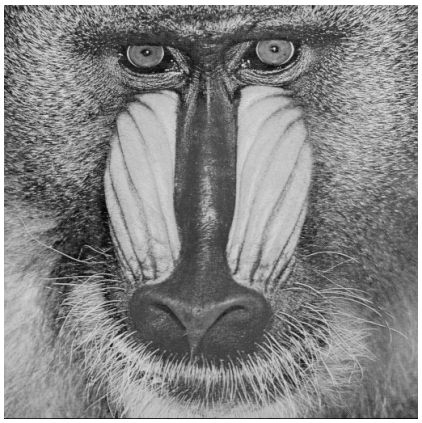

Figure 2: Image avant permutation

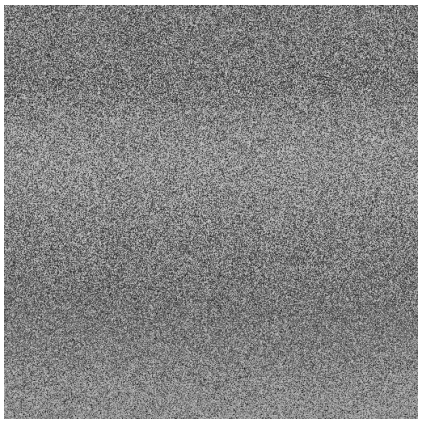

Figure 3: Image après permutation

b)

Tester ce programme avec plusieurs images et différentes valeurs de clé.

Avec la clé 29 et l'image de léna :

Figure 4: Image avant permutation

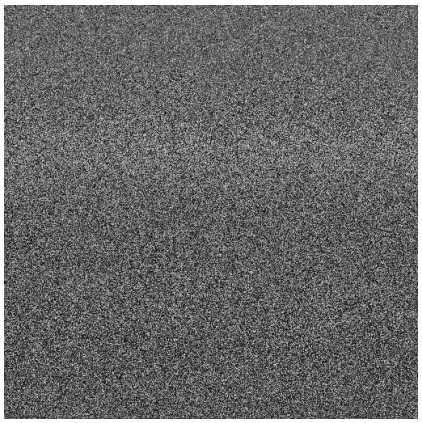

Figure 5: Image après permutation

Avec la clé 210 et l'image de la montagne :

Figure 6: Image avant permutation

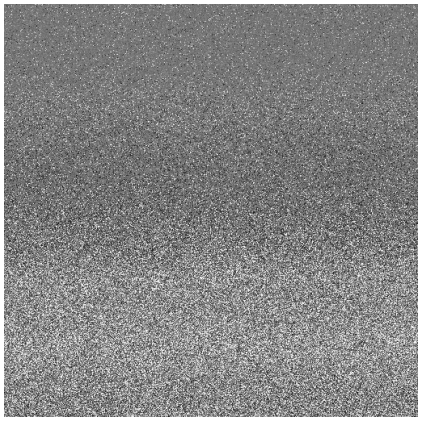

Figure 7: Image après permutation

c)

Choisir une de ces images et calculer :

Pour la suite, j'ai utilisé l'image du babouin et la clé 128.

1. Le PSNR en dB entre l'image originale et l'image chiffrée :

   Le PSNR que j'ai obtenu est de 11.654 dB. De ce que j'ai compris du PSNR, une valeur de 11.654 dB est considérée comme faible, ce qui suggère une grande différence entre l'image originale et l'image chiffrée, et donc un chiffrement efficace.

2. L'entropie de l'image originale ainsi que celle de l'image chiffrée :

Les entropies de l'image originale et l'image chiffrée sont les mêmes, 7.47443. Ce qui est normal car les pixels n'ont pas changé de place mais simplement de valeurs.

d)

Tracer sur la même courbe les histogrammes de l'image originale et de l'image chiffrée.

Que constatez-vous ?

Les histogrammes sont identiques; ils se superposent donc parfaitement. Ce qui est logique car nous n'avons pas modifié la valeur des pixels dans l'image.

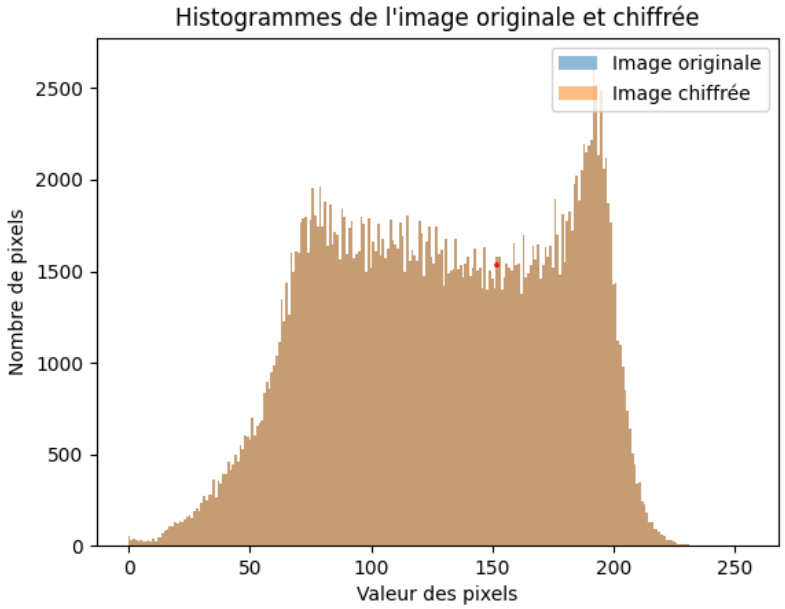

Figure 8: Histogramme image avant et après permutation

e)

Ecrire un programme permutinv.cpp permettant de déchiffrer une image chiffrée avec la même clé.

Explication rapide du code:

1. `positions` stocke les indices des pixels de 0 à `nTaille-1`.
2. `permutedPositions` stocke les indices après permutation.
3. Une boucle assigne aléatoirement une position inutilisée à chaque pixel dans `permutedPositions`.

4. Une autre boucle réaffecte les pixels à leurs positions d'origine en utilisant `permutedPositions`, dépermutant ainsi l'image.

```
std::list<int> positions; // Créez une liste pour stocker les positions originales des pixels après la permutation
for (int i = 0; i < nTaille; ++i)
    positions.push_back(i);

int *permutedPositions = new int[nTaille]; // Positions originales après la permutation

for (int i = 0; i < nTaille; ++i)
{
    // Choisir une position aléatoire parmi les non utilisées
    int randomPos = rand() % positions.size();

    auto it = std::next(positions.begin(), randomPos);
    int selectedPos = *it;

    permutedPositions[selectedPos] = i;

    positions.erase(it);
}

// Utilisez le mappage pour dé-permuter les pixels
for (int i = 0; i < nTaille; i++)
{
    ImgOut[permutedPositions[i]] = ImgIn[i];
}
```

Figure 9: Code d'inversion de permutation

Après avoir utilisé ce code, je reviens à mon image de babouin originale.

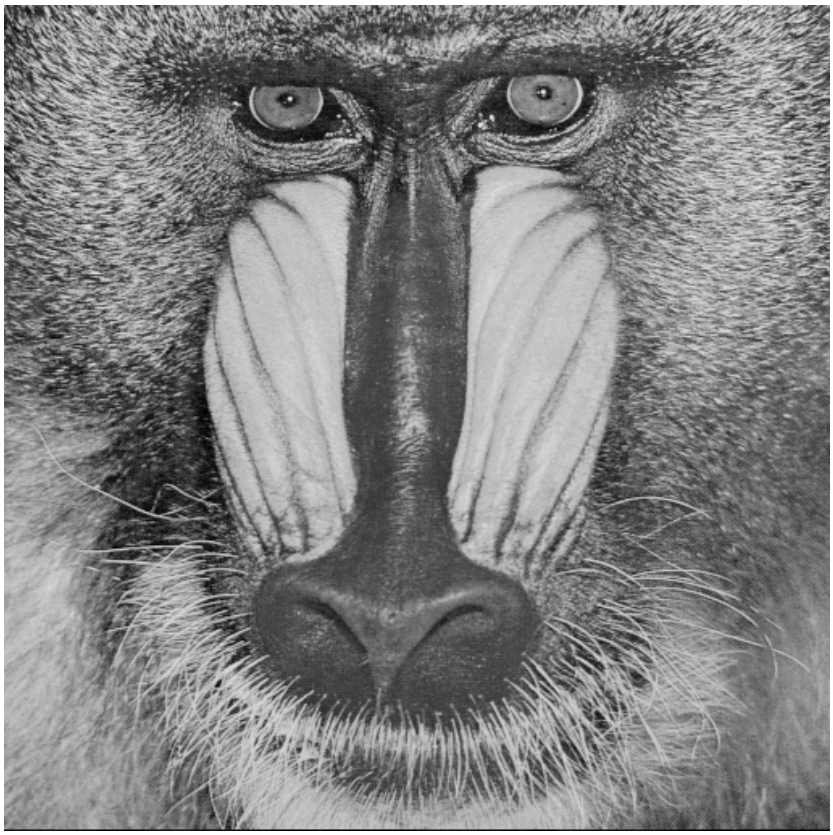

Figure 10: Image après inversion permutation

2. a)

Ecrire un programme subtitution.cpp ayant comme arguments d'entrée, le nom de l'image à chiffrer, la valeur de la clé K ainsi que le nom souhaité pour l'image chiffrée.

Dans le code suivant, j'effectue la substitution, en appliquant la formule expliquée en cours, sur les pixels en utilisant la clé 128.

```
for (int i = 0; i < nTaille; ++i)
{
    if (i == 0)
    {
        ImgIn[i] = ((rand() % 256) + ImgIn[i]) % 256;
    }
    else
    {
        ImgIn[i] = ((ImgIn[i - 1] + ImgIn[i]) + (rand() % 256)) % 256;
    }
}

    for (int i = 0; i < nTaille; ++i)
    {
        ImgOut[i] = ImgIn[i];
    }
```

Figure 11: Code de substitution

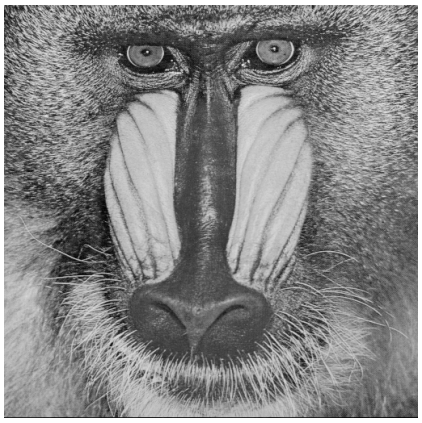

Figure 12: Image avant subtitution

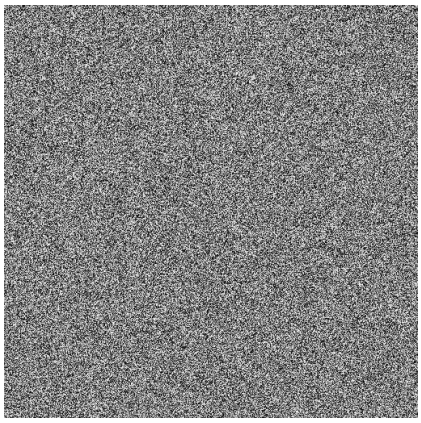

Figure 13: Image après subtitution

Le PSNR en dB entre l'image originale et l'image chiffrée :

Le PSNR que j'ai obtenu est de 9.24528 dB.
De ce que j'ai compris du PSNR une valeur de 9.24528 dB est considérée comme faible, ce qui suggère une grande différence entre l'image originale et l'image chiffrée, et donc un chiffrement efficace.

L'entropie de l'image originale ainsi que celle de l'image chiffrée :

Entropie de l'image originale : 7.47443

Entropie de l'image chiffré : 7.99932

L'écart d'entropie montre bien que la valeur des pixels a été modifié, de plus l'entropie élévé montre que l'image est chiffré.

Tracer sur la même courbe les histogrammes de l'image originale et de l'image chiffrée :
Que constatez-vous ?

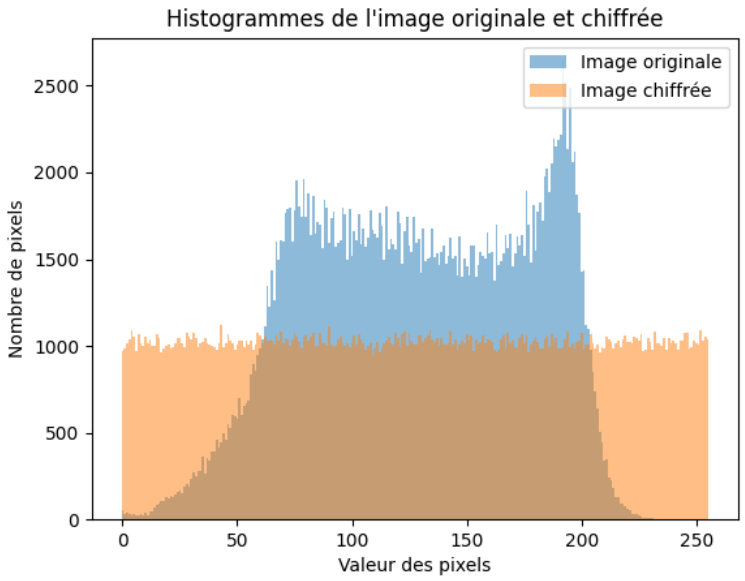

Figure 14: Histogramme image avant et après subtitution

Encore une fois graçe à l'histogramme on voit que on une plage de gris très réparti, qui ne ressemble pas du tout à l'image originiale. Je pense que cela montre que l'image chiffré est plus dur a déchiffré qu'avec la permutation car on n'a pas d'information de quels étaients les valeurs de pixels originaux sur l'image chiffré.

Ecrire un programme substitutioninv.cpp permettant de dechiffrer une image chiffrée sans connaitre la clé :

Pour cela Elias devait m'envoyer une image sur laquelle il avait fait sa subtituion mais malheureusement je n'ai pas réussi à la décripter car sa subtitution semble mal faites, j'ai demandé une image chiffré a Arthur ce qui m'a permis de tester mon code.

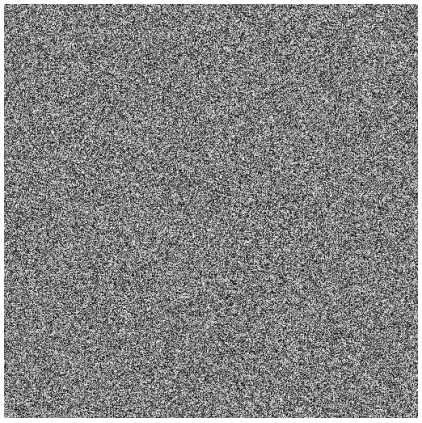

Figure 15: Image chiffré par Arthur

Figure 16: Image après déchiffrement

```
double minEntropy = 10;
int bestKey = 0;

for (int k = 1; k < 256; k++) {
    int currentKey = k; // Stockez la valeur actuelle de k

    OCTET *currentImage;
    allocation_tableau(currentImage, OCTET, nTaille);
    lire_image_pgm(cNomImgLue, currentImage, nH * nW);

    srand(currentKey); // Utilisez currentKey pour initialiser srand

        for(int i=0; i < nTaille ; i++ ){
            randarray[i] = rand()%256;
        }


        for(int i = nTaille-1; i >= 0; i--){

            currentImage[i] = (currentImage[i] - currentImage[i-1] - randarray[i])%256;

        }



        double entropy =  entropie.GetEntropieOfImage(currentImage,nTaille);

        free(currentImage);
        if (entropy < minEntropy) {
            minEntropy = entropy;
            bestKey = currentKey;
        }
}

srand(bestKey);

 for(int i=0; i < nTaille ; i++ ){
    randarray[i] = rand()%256;
}


 for(int i = nTaille-1; i >= 0; i--){

        ImgOut[i] = (ImgIn[i] - ImgIn[i-1] - randarray[i])%256;

    }
```

Figure 17: Code d'inversion de substitution

La clé que j'ai trouvé pour l'image d'Arthur était la 109, ce qui était la bonne clé mon inversion de subtitution est donction fonctionnelle.