SAE 1.03 - Image

Partie A

Question 0:

```
0000:0000 42 4D 99 73  0C 00 00 00  00 00 1A 00  00 00 0C 00
0000:0010 00 00 80 02  A9 01 01 00  18 00 FF FF  FF FF FF FF
0000:0020 FF FF FF FF  FF FF FF FF  FF FF FF FF  FF FF FF FF
```

Cette image peut s'obtenir à l'aide de l'application Okteta, qui permet d'obtenir le code de n'importe quelle image en binaire, octal, décimal ou hexadécimal.
On utiliseras ici durant la plus grande partie du rendu l'héxadécimal par le fait qu'il est plus facile de lire les valeurs associées aux adresses.

Expliquons désormais les valeurs que nous pouvons voir sur cette image:

1. L'en-tête du fichier

- On apelle un octet (soit 8 bits) un duo de charactères, et une adresse la position de l'octet dans le fichier (index à 0).
- A l'adresse 0x02, on peut retrouver la taille en bits du fichier défini sur 4 octets.
- A l'adresse 0x0A, on retrouve sur 4 octets l'adresse où on peut retrouver le début du codage des pixels, c'est à dire que allant à cette adresse, on trouveras le codage du premier pixel de l'image.

2. L'en-tête de la bitmap

- L'adresse 0xOE permet de donner la taille de ce second en-tête. Cette taille est défini sur 4 octets.
- La largeur et la auteur de l'image sont respectivement définis aux adresses 0x12 et 0x14 sur 2 octets chaqu'un.
- Enfin, à l'adresse 0x18 on peut retrouver le nombre de bits par pixel. Cette valeur est défini sur 4 octets.

Lorsqu'on essaye d'ouvrir l'image, on obtient l'erreur suivante:

```
dorian@dorian-ROG-Strix-G713RM-G713RM:~/Desktop/Sae-Image/Images$ display -sample 5000% ImageExemple.bmp
display-im6.q16: length and filesize do not match `ImageExemple.bmp' @ error/bmp.c/ReadBMPImage/854.
```

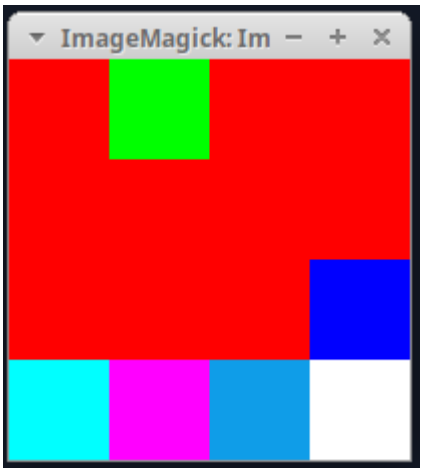
Celle-ci se trouve par le fait que la taille du fichier entrée à l'adresse 0x02 n'est pas égale à la taille réelle du fichier. En effet, on code 99 73 0C 00 alors que le fichier possède une taille de 9A 73 0C 00
width or height exceeds limit

Question 1:

En suivant la documentation donnée au début de la question, on peut facilement recoder une image de 4x4 facilement.

Question 2:

Pour efficacement arriver à l'image demandée dans cette question, on remarque que le fond est majoritairement rouge, ce qui nous permet alors de faciliter la construction en encodant tout les pixels avec 00 00 FF pour afficher une couleur rouge. Suite à cela, on ajoute alors les autres couleurs par dessus. Pour trouver facilement l'encodage des couleurs, on peut utiliser ctrl + F pour rechercher le nom de la couleur voulue, au lieu de la rechercher parmis les centaines de couleurs listées. Une fois avoir changer les 6 pixels qui ne sont pas rouges, on obtient alors l'image suivante:



Question 3:

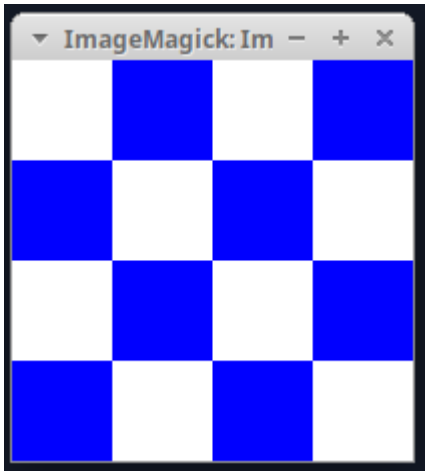
Pour répondre aux questions posées dans la question A.3, on utiliseras la documentation disponible via le lien suivant

- L'image 1 possède alors un point de 102 bits.
- Pour trouver le nombre de bits par pixel, on lit l'adresse 0x1C sur 2 octets. Les deux images utilisent 18 biarts par pixels, c'est à dire que chaque couleur primaire du codage RVB peut être parmis 256 possibilitées.
- Pour l'image 1, l'adresse 0x22 nous informe que la taille des données pixels est de 48 bits, codés par l'hexadécimal 30. Cette réponse peut aussi se retrouver par le calcul hauteur X largeur X nombre d'octets par pixel, soit 4x4x3= 48. Cette donnée n'est par ailleurs par disponible sur l'image 0.
- Pour définir si une compression est utilisé, il faut lire l'adresse 0x1E et regarder si la valeur est différente de 0. Pour l'image 1, on retrouve effectivement que cette valeur est nulle, ce qui signifie qu'aucune compression est u Skip course categories Course categories Etudiants - PAONe la palette. Pour commencer, les couleurs de la palette sont définis après le bitmap info holder, et le nombre de

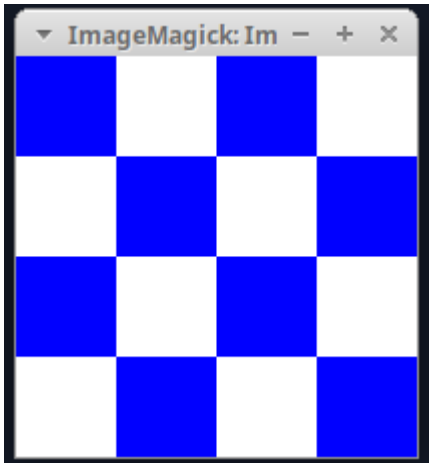
couleurs dans la palette peut se trouver à l'adresse 0x2E. Dans la zone de définition des couleurs, celle-ci sont codées en RVBA (Rouge Vert Bleu Alpha) avec A l'indice de transparence de la couleur, chaque couleur est donc codée sur 32 bits.
L'encodage des pixels seras détaillé ci-dessous

- Comme dit au-dessus, à l'aide de l'adresse 0x2E, on peut trouver que l'image 2 possède 2 couleurs qui sont le rouge et le blanc
- Pour ce qui est de l'encodage des pixels eux-mêmes, pour associer un pixel avec une couleur de la palette, il suffit d'appeler le numéro de palette sachant qu'elle à pour index 0. C'est à dire que pour la seconde couleur que nous avons codés, il suffit de mettre la valeur du bit à 1. Pour cette image, par le fait qu'elle ai seulement deux couleurs, chaque couleur est défini sur 1 bit, ce qui nous permet alors d'encoder les pixels via le binaire.
- Pour changer tout les pixels rouge en pixels bleu, il suffit de modifier le code de la couleur rouge pour qu'elle code du bleu. Pour cela, on modifie alors l'adresse 0x36 et on obtient le code et l'image ci-dessous:

```
0000:0030 00 00 02 00  00 00 FF 00  00 00 FF FF  FF 00
```



- Pour inverser les damiers de l'imag Skip course categories Course categories Etudiants - PAONest certainement la meilleure, surtout à grande échelle, par le fait qu'il faille uniquement changer 8 octets alors qu'il faudrais en modifier un nombre qui tend vers l'infceini plus l'image est grande.



- Comme dit précédemment, utilisé le binaire est bien plus intuitif pour codé des images avec une palette de deux couleurs, en voilà la raison:

```
11110000
```

Image 3
Il suffit alors de mettre un 0 pour un pixel rouge, et un 1 pour un pixel blanc!

- En lisant l'adresse 0x2E de l'image exemple index, on peut lire qu'elle possède 16 couleurs différentes. Chaque pixel va donc être codé sur un octet ($16 = 2^4$, donc il faut 4 prévoir 4 bits pour

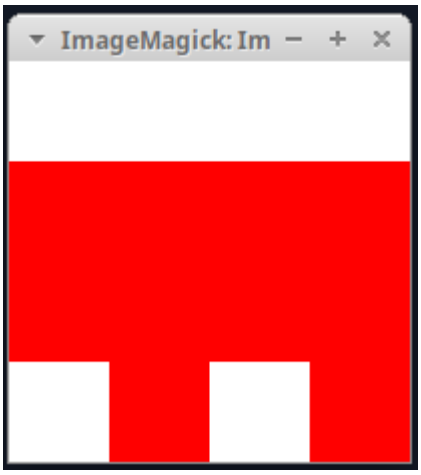
appeler les couleurs de la palette) Skip course categories Course categories Etudiants - PAONante dans l'image, elle se retrouve alors majoritairement dans le codage de celle-ci. On peut retrouver alors un grand nombre de C qui se repète. On peut alors déduire que ce C est la couleur blanche qui est majoritaire dans l'image. C étant égale à 12, la couleur blanche est la 13ème à être codée dans la palette de couleur. La couleur majoritaire est donc codée à l'adresse 0x66!

- Pour trouver où commence le tableau de pixel, il suffit de lire l'adresse 0x0A où on peuttrouver comme valeur 76. Ce 76 signifie que l'adresse 0x76 est le début de la zone de définition des pixels.
- En changeant quelque pixels de l'adresse 0x76 à 0x7A de C à 0, on peut remarquer que le coin inférieur gauche de l'image a effectivement été affecté de manière suivante:

- En diminuant le nombre de couleurs dans l'image exemple index, on obtient l'image exemple index 2, qui possède alors un nombre de couleur plus faible. Cette seconde version possède toujours 16 couleurs selon l'adresse 0x2E, et c'est par le fait qu'on retrouve en effet 16 couleurs dans la palette, mais qu'une majorité soit désormais égale à 00 00 00 00.
  D'un point de vu viseul, on obtient alors l'image ci-contre:

Question 5:

Rappelant que la hauteur de l'image est défini à l'adresse 0x16, est que l'image 3 possède une hauteur de 00 00 00 04, pour trouver l'inverse on utilise l'encodage C2. FF FF FF FF représentant -1, FC FF FF FF est alors égal à -4! On obtient alors l'image suivante où l'ordre des lignes est inversé, c'est à dire que l'encodage ce fait du haut vers le bas:



De la même façon, on peut retourner le logo de l'iut pour obtenir l'image suivante

Question 6:

Suite à l'application de la conversion RLE, l'image 4 possède un poid de 1120 octets. Ce poid s'explique par le fait que la compression RLE créer une palette de 256 couleurs, même si l'image en utilise que deux.
La zone de définition des pixels se retrouve à l'adresse 0x0A où on peut trouver l'adresse 0x0436.
Explication de l'encodage des pixels:

- La compression RLE est intuitive et se base sur le codage par index de couleur, avec pour différence le fait qu'on ajoute le nombre de pixels conséquent qui auront cette couleur. C'est à dire qu'une ligne de pixel rouge-rouge-blanc-rouge seras codé 02 00 (2x rouge) 01 01 (1x blanc) 01 00 (1x rouge).
- De plus, on retrouve des 00 00 et 00 01 dans le code hexadécimal du codage des pixels. 00 00 signifie que c'est la fin de la ligne, et 00 01 la fin de l'image.

Question 7:

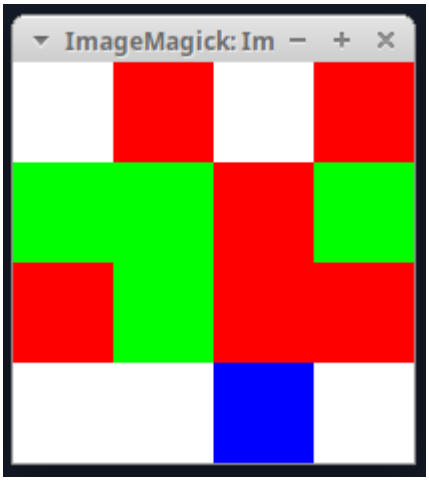
L'image 5 pèse 1102 octet, ce qui est plus petit que l'image 4. Ceci s'explique par le fait que l'image 5 possède des répétitions de pixels alors que l'image 4 n'en possède pas, ce qui rend la compression RLE plus efficace.
En différence au codage de l'image 4, on trouve le codage 04 00 (4 rouge) 04 01 (4 blanc) 04 00 (4 rouge) qui est là, source de la baisse du poid de l'image.

Question 8, 9 et 10:

Pour les question 8 à 10, on doit modifier le codage des pixels ainsi que la palette de couleur pour les question 9 et 10. Une fois avoir modifié les images, on obtient les images suivantes:

Pour réduire la taille de l'image 8, il suffit de supprimer les 252 couleurs non utilisés dans la palette de couleur et de redéfinir les valeurs de l'en-tête (poid de l'image, nombre de couleur dans la palette, ect)

Size: 106 bytes



Partie B

Pour les questions de la partie B, vous pouvez retrouvez un programme python correspondant (appart pour la B9) dans le dossier ./Scripts

Question 1 à 4:

Pour les quatres premières question, il suffit de prendre base sur le programme initial et d'y apporter de simple modifications:

- Pour le script 1, il faut inverser l'output (ligne,colonne) à (colonne,ligne) pour pouvoir obtenir la transposée de l'image.
- Pour le script 2, il suffit de modifier l'output à (-ligne, colonne) pour avoir l'impression de voir l'image comme dans un miroir.
- Pour le script 3, il suffit de modifier la valeur des pixels en ajoutant la ligne "gris = ((c[0]+c[1]+c[2])//3)", ce qui nous permet alors de définir chaque composante du pixel étant égale à gris
- Enfin, pour le script 4, il suffit d'ajouter un if/else, et de mettre un putpixel dépendamment de si le pixel valide la condition ou non.

Question 5:

Le programme B5.py détaille l'ensemble des procédures faites pour répondre à cette question mais nous allons reprendre ici les procédures importantes.
Tout d'abord, on modifie l'image initiale en modifiant la composante rouge à l'entier pair inférieur. Faire ceci nous permet alors de créer un espace de travail au sein de cette image. L'image que nous allons coder sera alors codée au sein du dernier bit de la composante rouge des pixels.
Suite à cela, on prend les valeurs de l'image à cacher en définissant si les pixels sont noirs ou blanc, et selon le résultat, on ajoute alors 0 ou 1 dans la composante rouge de l'image qui serviras d'intermédiaire.

Pour ce qui est du décodage, on utilise le procédé inverse, c'est à dire que l'on obtient les valeurs de l'image intermédiaire en extractant les 0 et 1 du dernier bit de la composante rouge de chaque pixel, et on défini alors une image à partir des 0 et 1, un 0 reviens à un pixel noir et un 1 à un pixel blanc. L'ensemble des pixels se regroupe alors pour former une chaine, et à terme restitue l'image entière!

Question 6:

Encoder un texte dans une image est bien plus complexe que d'encoder une image dans une image (si l'image à cacher est en noire et blanc cela dit) par le fait qu'un seul charactère est définit sur 8 bits en ASCII. Il y a alors plusieurs possibilité, mais une seule est retenu par le fait qu'on ne peut pas modifier l'image initiale (du moins, ca ne doit pas être visible à l'oeil nu). On va alors utiliser la même méthode que le script B5.py mais un charactère va être encodé sur chaque composante RGB de 3 pixels! En effet, il y a 8 bits pour définir le charactère lui même, il reste alors un 9ème bit disponible. Ce neuvième bit sera utilisé pour définir si on doit continuer de décoder ou si l'entièreté du message à été décodé.
Enfin, il faut transformé chaque charactère du message en ASCII, puis en binaire avant de pouvoir l'intégrer des les pixels! Pour ce qui du décodage, on refais une nouvelle fois le procédé inverse.

Question 7:

Le chiffre de Vernam est incassable. En effet, il se base sur une clé qui est une suite de lettre, ce qui fait que pour un message de longueur x, il y a x^26 possibilités de clés! Pour un message de trois lettres, ca fait 17576 possiblités.
Seulement, il y a un problème de grande ampleur: "Comment transmettre cette clé ?". En effet cette clé est à usage unique, donc il faut réussir à la transmettre au destinataire à chaque fois sans que personne puisse l'intercepter, et c'est dans cette mesure qu'on peut dire que le chiffre de Vernam n'est pas incassable, par le simple fait qu'il faille réussir à transmettre cette clé au destinataire...
On peut retrouver un exemple d'utilisation du chiffre de Vernam dans le script B7.py

Question 8:

Pour réussir cette question, il suffit de fusionner les scripts B6 et B7 ensemble, pour pouvoir encoder et décoder un texte dans une image à l'aide d'une clé.
On voit que c'est effectivement assez simple de mettre en place, et que si on trouve un moyen de partager la clé, alors on peut facilement communiquer avec quelqu'un sans que personne puisse nous traquer.
On peut alors comprendre l'importance de la surveillance de masse, surtout sur le web par le fait qu'il est facilement possible de mettre en place un canal de communication privatisé, et que certaines personne puissent l'utiliser pour le mal.

Question 9:

La clé de Solitaire permet alors de répondre à la problématique posé par le chiffre de Vernam vu précédemment et cette clé peut être encodée de la même facon que le chiffre de Vernam dans une image (voir script B7.py et B8.py)

Conclusion générale:

Les images .bmp sont facile de compréhension par le fait qu'elles soient intuitivents. Toutes les données pevent s'y retrouver dans le bitmap info header, ce qui permet d'introduire facilement le codage des images. Pourtant le bmp n'est plus une extension d'image qu'on utilise régulièrement, au bénéfice du .jpg ou encore du .png. De plus, si on souhaite utiliser une extension d'image pour coder facilement, on aurais tendance à utiliser le .ppm face au .bmp.

SAE 1.03 - Image
Par Ludmann Dorian 1.3b