

UFR

de mathématique
et d'informatique

Université de Strasbourg

PROJET SDA2

Aymen KOUSKOUESS

PROJET SDA2	1
I - Explication des fonctions	2
a) Fonctions pour l'image	2
1) Fonctions de lecture et écriture des données	2
Read	2
But:	2
Clarification:	2
Write	3
But:	3
Clarification:	3
Generate	3
But:	4
Clarification:	4
2) Fonctions coloriage , libération	4
Coloriage	4
But:	4
Clarification:	4
Libérer	5
But:	5
Clarification:	5
b) Implémentations pour la structure d'arbre	5
• Structure de données Pixel	5
Makeset	5
But:	5
Clarification:	5
Findset	5
But:	5
Clarification:	5
Union	6
But :	6
Clarification:	6
c) Implémentations pour la structure de liste	6
Structure de données Pixel	6
Structure de données Liste	6
Makeset	6
But:	6
Clarification:	7
Findset	7
But:	7
Clarification:	7
Union	7
But:	8
Clarification:	8

II- Réponses aux questions	8
Listes:	8
Arbres :	11
III)Utilisation	15
1) Pour l'implémentation de listes	15
2) Pour l'implémentation d'arbre	15
IV) Difficultées rencontrées	16

I - Explication des fonctions

a) Fonctions pour l'image

1) Fonctions de lecture et écriture des données

Read

```
pixel** Read(char* image);
```

But:

- **Lit** une image au format PBM ASCII et stocke ses pixels dans un tableau bidimensionnel.

Clarification:

- Cette fonction **vérifie l'existence de l'image** donnée en paramètre, **vérifie ses attributs** s'ils correspondent à celui d'une image PBM.
- Une image PBM est **codée ligne par ligne** en partant du haut et chaque ligne est codée de gauche à droite **ne dépassant jamais 70 caractères**.
- L'allure de **l'image PBM** d'un bitmap de la lettre "J" est comme suit :

```
P1
7 10
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 1 0 0 0 1 0
0 0 1 1 1 0 0
0 0 0 0 0 0 0
```

Source : Wikipedia contributors. (2022, October 2). *Portable pixmap*.

<https://fr.wikipedia.org/wiki/PortablePixmap>

- Les première et deuxième lignes correspondent respectivement au **nombre magique** et à la largeur puis la hauteur.
- De la ligne 4 à la fin correspondent à une **succession des valeurs** contenant la couleur de chaque pixel avec **0** pour une couleur **blanche** et **1** pour une couleur **noire**.
- La fonction **stocke** ensuite les données lues dans **un tableau bidimensionnel** et assigne à chaque pixel blanc une couleur RGB **aléatoire**.

Write

```
void Write(pixel** dpx, int hauteur, int largeur, char* image);
```

But:

Génère une image au format PPM.

Clarification:

- La fonction commence par écrire dans le fichier de l'image à générer les attributs correspondants. Puis place les valeurs de chaque pixel dans le fichier de l'image.

Generate

```
void Generate(int n,int m,char* image);
```

But:

La fonction **Generate** génère une image en **noir et blanc** de dimensions $n * m$.

Clarification:

- La fonction **crée** un fichier et met en première ligne le **nombre magique P1** (correspondant à un fichier PBM ASCII comme expliqué dans l'explication de la fonction **Read**) , en deuxième ligne **n et m** et met dans les lignes **3 à n** , une **séquence de 0 et 1 représentant le noir ou le blanc** choisis aléatoirement m fois et ne dépassant pas 70 caractères par ligne.

2) *Fonctions coloriage , libération*

Coloriage

```
void Coloriage(pixel** array);
```

But:

La fonction **coloriage** fait appel aux fonctions dont on a besoin pour colorier l'image.

Clarification:

- Elle commence d'abord par **parcourir** le tableau **bidimensionnel** de la table des pixels. La fonction passe par plusieurs conditions pour ensuite effectuer un **Union**. Première condition est que déjà le pixel doit être de couleur différente de la couleur **Noir**, si la condition est valide, nous vérifions ensuite la couleur des voisins connexes **horizontal** et **vertical** qu'elle soit aussi différent de **Noir**. Enfin si les conditions sont valide une **Union** des pixel **connexes** est effectuée avec le pixel px.

Libérer

```
void liberer(pixel**bd);
```

But:

Libère l'espace occupé en mémoire du tableau **bidimensionnel** et des ensembles de pixels.

Clarification:

- Parcours le tableau **bidimensionnel** pour **libérer** chaque élément de la structure pixel.

b) *Implémentations pour la structure d'arbre*

• *Structure de données Pixel*

```
typedef struct pixel {  
    int couleur[3]; // premier param  
    int rang;      // deuxième param hauteur  
    struct pixel *father; // pixel pere  
}pixel;
```

Makeset

```
void makeset(pixel* px,int color)
```

But:

Initialise le pixel avec un rang qui vaut 0 et **pointant** sur lui-même.

Clarification:

- La fonction **initialise** le pixel donnée en paramètre avec un rang qui est égal à (0). et comme il ya un seul pixel dans l'arbre, le père de ce pixel et lui même. Donc il pointe vers lui-même.

Findset

```
pixel* Findset(pixel* px);
```

But:

Retourne la racine de l'arbre de px.

Clarification:

- La fonction **localise** le nœud avec px et **itère** à travers l'arbre où il appartient (le père de px , le père du père etc...) pour retrouver la racine de l'arbre (représentant de la l'ensemble).

Union

```
void Union(pixel* racpx1, pixel* racpx2);
```

But :

Fait l'union de deux arbres différents.

Clarification:

- La fonction appelle **Findset()** pour trouver la racine de chaque pixel donnée en paramètre puis relie le père de la racine du deuxième pixel a la racine du premier pixel.

c) *Implémentations pour la structure de liste*

Structure de données Pixel

```
typedef struct pixel {  
    int couleur[3];  
    struct pixel* next;  
    struct Liste* ensemble; // pointe vers l'ensemble qu'il appartient  
}pixel;
```

Structure de données Liste

```
typedef struct Liste  
{  
    pixel* head;  
    pixel* tail;  
  
    int size;  
}Liste;
```

Makeset

```
void makeset(pixel* px,int color);
```

But:

Crée un nouvel **ensemble** Sx, ne contenant que l'élément x et vérifie la couleur donnée en paramètre.

Clarification:

La fonction **crée** un nouvel **ensemble** dans l'ensemble **disjoint** où le seul élément est x et qui sera également le **représentant**.

Celle-ci **créera** et **allouera** d'abord **dynamiquement** une Liste dont la queue et la tête pointera vers **px**. (qui sera ensuite reliée avec d'autres pixel grâce à la fonction **union**).

L'élément suivant de px sera **NULL** et l'**ensemble** sera la liste créée.

On vérifie la couleur de la même manière que pour arbre.

Cas 1 :

Si la couleur est noire , on assigne aux pixels la couleur noire.

Cas 2 :

Si la couleur est blanche , on lui octroie une couleur au hasard (on utilise random dans la librairie time.h) . Enfin ,on incrémente la taille de la liste de 1.

Findset

```
pixel* Findset(pixel* px);
```

But:

Retourne le représentant de la liste donnée en paramètre.

Clarification:

La fonction fait un simple retour de la tête de l'ensemble ou le pixel px appartient.

Union

```
void Union(pixel* rpx1, pixel* rpx2);
```

But:

Unifie deux ensembles de pixels en un seul.

Clarification:

La fonction recherche l'ensemble contenant rpx1 et l'ensemble contenant rpx2 afin de les rejoindre ensemble. L'ensemble le plus petit entre les deux et celui qui va être traversé pour garder l'optimisation au mieux possible. Puis la fonction pointe tous les pixels de l'ensemble traverser vers l'autre ensemble. Enfin libère la mémoire de l'ensemble traverser et retourne le l'ensemble unifier .

II- Réponses aux questions

Listes:

Question 4 :

Afin de tester **Makeset()** et **Findset()** , nous procédons comme suit :

1. on crée 3 pixels px1,px2 et px3
2. On affiche leurs couleurs respectives
3. On relie px1 et px2 en mettant px1 comme tête de liste pour px2
4. On crée un pixel head qui aura pour valeur le retour de la fonction **Findset(px2)** (le head de px2)
5. On affiche et on voit bien que **Findset** retourne le head de px2 et que celui-ci reste inchangé.

```
int main(){
    srand(time(NULL));

    pixel* px1= malloc(sizeof(pixel));
    makeset(px1,0);
    pixel* px2= malloc(sizeof(pixel));
    makeset(px2,0);
    pixel* px3= malloc(sizeof(pixel));
    makeset(px3,0);

    printf("%d %d %d\n", px1->couleur[0],px1->couleur[1],px1->couleur[2]);
    printf("%d %d %d\n", px2->couleur[0],px2->couleur[1],px2->couleur[2]);
    printf("%d %d %d\n", px3->couleur[0],px3->couleur[1],px3->couleur[2]);

    px2->ensemble=px1->ensemble; // ligne 15
```

```

pixel* head = Findset(px2);
printf("%d %d %d\n", head->couleur[0],head->couleur[1],head->couleur[2]);
printf("%d %d %d\n", px2->couleur[0],px2->couleur[1],px2->couleur[2]);
}

```

Résultat sur la sortie standard :

```

39 222 21 // couleurs du premier pixel
70 119 223 // couleurs du deuxième pixel
50 224 251 // couleurs du troisième pixel
39 222 21 // couleurs de findset(px2)
70 119 223 // appel pour montrer que px2 est intact après la ligne 15

```

Question 5 :

Makeset :

- La fonction est constante ayant comme complexité en pire et meilleur cas **O(1)** alors **$\Theta(1)$** .
- **Coût en mémoire : O(1)** car aucun malloc a été utilisé donc on alloue pas de l'espace dans la mémoire.

Findset :

- **Coût en pire cas : $\Theta(1)$** car c'est une fonction constante .
- **Coût en mémoire : O(1)** prend pas de place en mémoire.

Question 7 :

- Notre fonction **Union()** va effectuer l'opération sous une complexité de **O(n)**.

Question 8 :

- Pour tester **Union**, on procède comme dans la question 4 avec le même programme mais cette fois-ci, au lieu d'écrire la ligne 15 , on met à la place l'appel de **Union()** comme suit :

```
Union(px1,px2);
```

- Et nous avons le même résultat en sortie standard :

```

75 172 142
117 14 236
59 113 57
75 172 142 // unifiée avec la ligne 15 , union
117 14 236

```

On teste maintenant le coloriage de celtique sans union qui nous donnera la **Figure 1**.

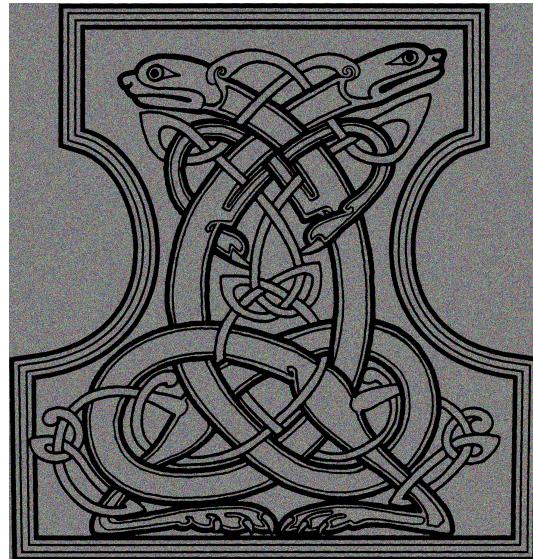


Figure 1 : Celtique sans utiliser d'union.

On rajoute maintenant Union à notre algorithme de coloriage et nous obtenons la **Figure 2**.



Figure 2 : Celtique avec coloriage

Question 9 : la complexité asymptotique en pire cas et le coût en mémoire de l'algorithme.

Coloriage:

- $O(n * m * t)$ où m et n sont les dimensions de l'image (**$n*m$ est le nombre de pixel**) et t est la taille moyenne de l'ensemble du pixel.
- S'il y a une limite dans la taille de l'ensemble , par exemple si elle est limitée par le nombre de pixels qu'il y a , alors le pire cas de la complexité asymptotique serait $O(m^2 * n^2)$.

- Ceci est une conséquence des trois boucles l'une dans l'autre , la première est de taille n , la deuxième est de taille m et la dernière est de taille ($m*n$). La complexité serait donc de $O(n*m*(m*n))$.
- **Coût en mémoire : $O(1)$**

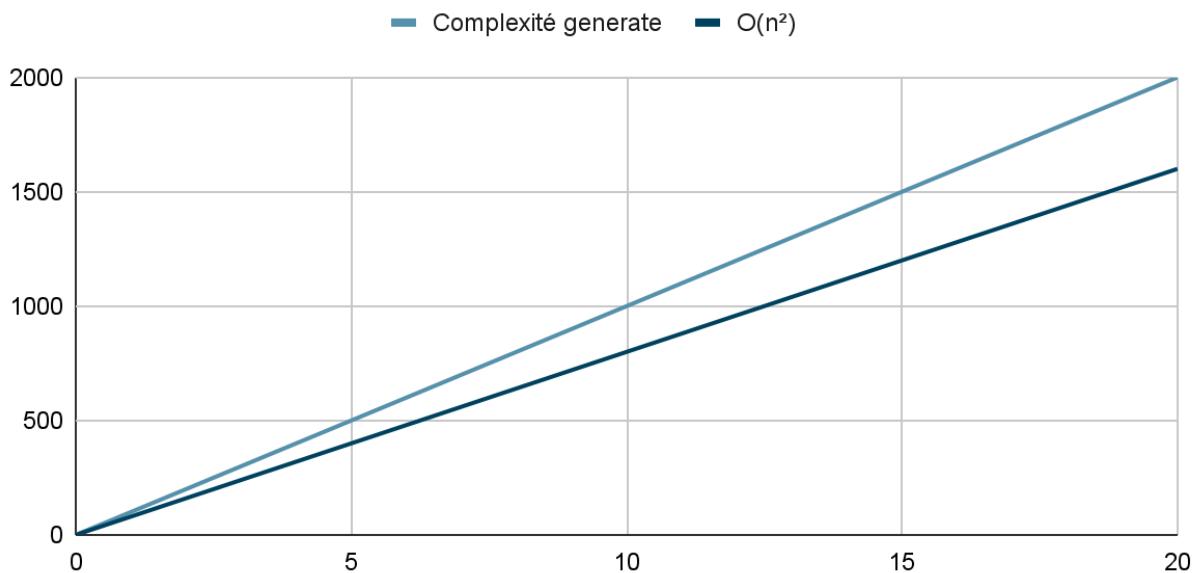
Question 10 :

Nous cherchons à comparer la complexité de generate à celle de $O(n^2)$

- La fonction generate possède deux boucles imbriquées et sa complexité au pire cas est donc de $O(n*m)$ avec n correspondant à la largeur de l'image et m correspondant à la longueur de l'image.
- Nous avons tracé un tableau avec les tests qu'on a effectués pour le temps de génération suivant la taille du tableau avec un pas de 500.
- Nous avons ensuite colorié les images et avons fait un timer pour chacune des images.

Taille tableau	Temps de génération en millisecondes
0*0	0.000014
500*500	1.4670
1000*1000	5.7642
1500*1500	10.6758
2000*2000	20.7764

Graphique du temps écoulé en ms en fonction de la taille du tableau



Arbres :

Question 11 :

- Nous procédons encore de la même manière que dans la **question 4** , en remplaçant cette fois-ci la ligne 15 par :

```
px2->father=px1;
```

- Et nous avons le même résultat dans la sortie standard qui montre que notre **Makeset** et **Findset** fonctionnent :

```
155 163 182
```

```
39 62 37
```

```
177 90 40
155 163 182 // obtenue avec la ligne 15 décrite au début de la réponse q)11
39 62 37
```

Question 12 :

Makeset :

- La fonction est constante ayant comme complexité asymptotique en pire et meilleur cas **O(1)** alors $\Theta(1)$.
- **Coût en mémoire : O(1)** la mémoire a été alloué une fois.

Findset :

- **Coût en pire :** complexité de **O(n)**.
- **Coût en mémoire : O(1)**.

Question 14 :

La complexité asymptotique en pire cas d'une séquence de n créations d'ensembles S_1, \dots, S_n suivie des unions successives de ces ensembles en un seul ensemble final

- l'union de ces ensembles va s'effectuer sous un temps linéaire de **O(n)** avec notre fonction union.
T(n,m) = n + m où **n** et la hauteur du premier arbre et **m** la hauteur du deuxième arbre.

Question 15 :

- Nous procédons encore de la même manière que dans la **question 4**, en remplaçant cette fois-ci la ligne 15 par :

```
Union(px1,px2);
```

- Nous obtenons les mêmes résultats que dans la question 8 (**Figure 1**) avant de faire l'**union**, c'est-à-dire, des pixels colorés aléatoirement.
- Après avoir fait l'**union**, nous obtenons une image colorée aléatoire comme dans la **Figure 2** de la question 8.

Question 16 :

Coloriage :

- $O(n * m * t)$ où m et n sont les dimensions de l'image ($n * m$ est le nombre de pixel) et t est la taille moyenne de la hauteur de l'arbre.

- **Coût en mémoire : $O(1)$**

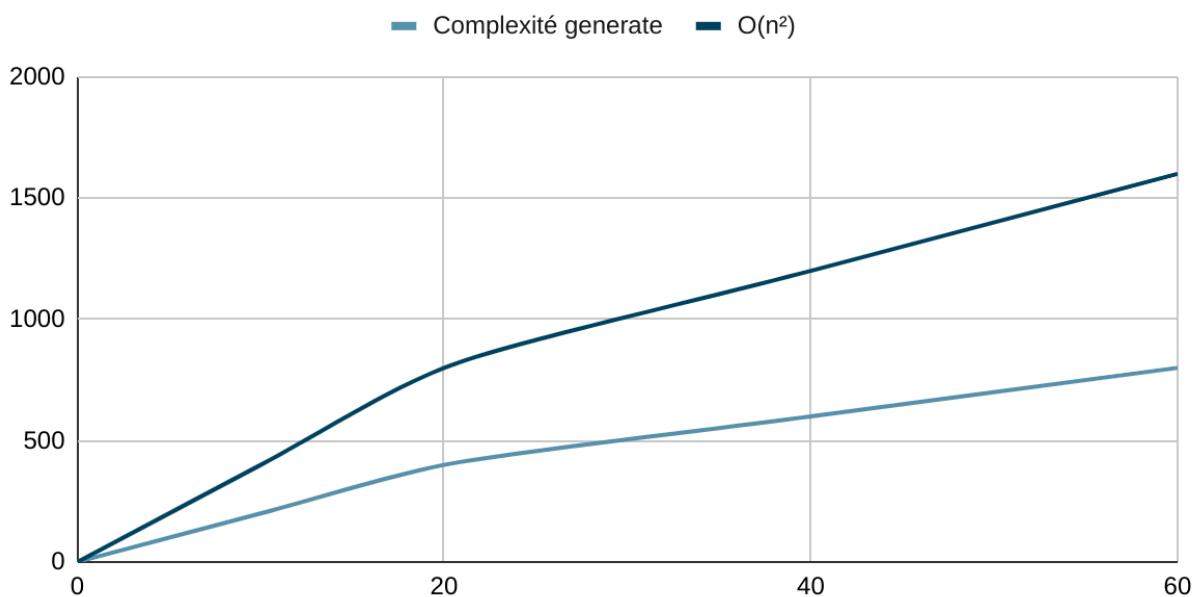
Question 17 :

Nous cherchons à comparer la complexité de generate à celle de $O(n^2)$

- La fonction generate possède deux boucles imbriquées et sa complexité au pire cas est donc de **$O(n*m)$** avec **n** correspondant à la largeur de l'image et m correspondant à la longueur de l'image.
- Nous avons tracé un tableau avec les tests qu'on a effectués pour le temps de génération suivant la taille du tableau avec un pas de 500.
- Nous avons ensuite colorié les images et avons fait un timer pour chacune des images.

Taille tableau	Temps de génération en millisecondes
0*0	0.000014
500*500	2.4584
1000*1000	8.14585
1500*1500	14.1447
2000*2000	18.2146

Graphique du temps écoulé en ms en fonction de la taille du tableau



Question 18

Nous rajoutons dans la fonction main un timer avant l'appel du **Read** avec

```
clock_t start =clock();
```

et après la libération de mémoire , on met fin au timer et on affiche le temps écoulé avec

```
clock_t end =clock();
printf("\nTemps écoulé :%f secondes \n", (double)(end-start)/CLOCKS_PER_SEC);
```

Source : *CLOCKS_PER_SEC in C language found the time.h library.* (2018, July 24). Stack Overflow.

<https://stackoverflow.com/questions/51501410/clocks-per-sec-in-c-language-found-the-time-h-library>

Temps pour la structure de données de liste pour l'image de **carte du monde** :

```
Temps écoulé :2.536294 secondes
```

Temps pour la structure de données d'arbre pour l'image de **carte du monde** :

```
Temps écoulé :1203.238209 secondes
```

III) Utilisation

Nous avons séparé les deux implémentations en deux répertoires différents : l'un pour liste , l'autre pour arbre.

1) Pour l'implémentation de listes

Pour colorier une image en utilisant les fonctions implementant une liste chaînée améliorée , il faut aller dans le répertoire liste puis saisir :

```
make
./liste ImagesTests\ /carte_france.pbm CarteFranceListe
```

avec “**carte_france.pbm**” l'image PBM de test et “**CarteFranceListe**” , le nom de l'image PPM qui sera créée

2) Pour l'implémentation d'arbre

Pour colorier une image avec une implémentation d'arbre , il faut aller dans le répertoire arbre puis saisir :

```
make  
./arbre ImagesTests\ /carte_france.pbm CarteFranceArbre
```

avec “**carte_france.pbm**” l'image PBM de test et “**CarteFranceArbre**” , le nom de l'image PPM qui sera créée.

IV) Difficultées rencontrées

Nous avons passé beaucoup de temps à tenter de résoudre un souci d'inclusion de fichiers.

Après la fin de l'implémentation de la structure avec une liste, le programme prenait plusieurs minutes à colorier l'image de la carte du monde. Nous avons réglé le problème avec plus de conditions sur les ensembles et comme ca le programme tourne même 10x plus rapide, a la place de 10 min pour générer l'image il est maintenant inférieur à 5s . Sauf que pour l'implémentation en arbres , les différences ont fait que l'optimisation était déjà proche du maximum. Le coloriage des images de tests pour arbre est rapide mais l'implémentation en liste chaînée est plus puissante.

Nous avions rencontré des difficultés par rapport à la complexité asymptotique de coloriage qui était difficile à faire et nous avons dû relire les cours sur la complexité que nous avions.