- TP 1. La composante géante. -

Le but de ce TP est d'observer le phénomène suivant : si l'on tire au hasard un graphe G ayant n sommets et m arêtes, avec m proche de n, on s'aperçoit que les composantes de G sont de tailles très inégales. Plus précisement, une *composante géante* apparaît presque sûrement, alors que les autres composantes sont soit des points isolés, soit très petites.

Langage. Nous programmerons en C++. Votre programme pourra commencer de la manière suivante (le code est disponible sur le moodle du cours) :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int
main()
{
  int n;     // Nombre de sommets.
  int m;     // Nombre d'aretes.
  cout << "Entrer le nombre de sommets:";
  cin >> n;
  cout << "Entrer le nombre d'aretes:";
  cin >> m;
  int edge[m][2];    // Tableau des aretes.
  int comp[n];       // comp[i] est le numero de la composante contenant i.

  return EXIT_SUCCESS;
}
```

- Exercice 1 - Création d'un graphe aléatoire G à n sommets et m arêtes.

L'ensemble des sommets de G est codé par $\{0, \ldots, n-1\}$. L'ensemble des m arêtes de G est stocké dans un tableau **edge** de taille $m \times 2$ et dont les entrées appartiennent à $\{0, \ldots, n-1\}$. Ainsi, si xy est l'arête de G d'indice k, on aura **edge**$[k][0] = x$ et **edge**$[k][1] = y$. Par exemple, le graphe sur l'ensemble de sommets $\{0, 1, 2, 3\}$ ayant pour arêtes 01, 02, 03, 12, 23 est codé par le tableau **edge**[5][2]={{0,1},{0,2},{0,3},{1,2},{2,3}}. Écrire une fonction *void grapheRandom(int n, int m, int edge[][2])* qui engendre aléatoirement le tableau **edge** en tirant au hasard chacune de ses entrées. On permettra la création d'arêtes multiples ou de boucles. On pourra utiliser les appels :

- `srand(time(NULL))` // Initialise la graine (seed) de la fonction `rand` sur l'horloge.
- `rand()%k` // Retourne un entier entre 0 et $k-1$.

- Exercice 2 - Calcul des composantes connexes.

Implémenter l'algorithme du cours *void composantes(int n, int m, int edge[][2], int comp[])* qui calcule les entrées du tableau **comp** de telle sorte que **comp**$[i]$=**comp**$[j]$ si et seulement si i et j appartiennent à la même composante connexe de G.

- Exercice 3 - Retourner les tailles des composantes connexes.

Écrire un algorithme *void ecritureTailles(int n, int m, int comp[])* qui écrit :

- Le nombre de points isolés de G (i.e. les composantes de taille 1).
- Les nombres de composantes des autres tailles, dans l'ordre croissant.

Par exemple, le résultat sera de la forme :

```
Il y a 464 points isoles.
Il y a 41 composantes de taille 2.
Il y a 12 composantes de taille 3.
Il y a 5 composantes de taille 4.
Il y a 1 composante de taille 4398.
```

Essayer d'avoir un algorithme linéaire en n pour la fonction *ecritureTailles*. Faire varier n et m pour constater l'apparition de la composante géante.

- Exercice 4 - Optimisation de l'algorithme.
Améliorer les performances de votre algorithme de telle sorte que :

1. Lors de la lecture d'une arête ij, si **comp**$[i]\neq$**comp**$[j]$, seuls les sommets k vérifiant **comp**$[k]=$**comp**$[j]$ sont relus et réaffectés en **comp**$[i]$.
2. Entre **comp**$[i]$ et **comp**$[j]$, choisir en priorité de réaffecter la composante de taille minimum.

On pourra à cet effet, notamment pour 1), utiliser la structure de donnée *vector* ou *stack*, voir ci-dessous. Essayer d'augmenter n et m le plus possible, avec la version non-optimisée et avec la version optimisée. La différence est-elle sensible ? Établir la complexité de chacune des versions.

- Exercice 5 - Pour aller plus loin.
Lorsque $n = 10000$ et $m = 5000$, quelle est environ la proportion de points isolés ? Faîtes plusieurs tests et relevez la moyenne obtenue. De même avec $n = 10000$ et $m = 10000$. Modélisez mathématiquement le problème et trouvez les moyennes théoriques.

Mémento vector (à garder pas loin pour les tps suivants)

- *vector<int> vect* // Déclare la variable *vect* comme vector d'entiers.
- *vect[i]* // Accède à la $i^{\text{ième}}$ entrée de *vect*.
- *vect.size()* // Renvoie la taille de *vect*.
- *vect.push_back(i)* // Empile la valeur i sur *vect*.
- *vect.pop_back()* // Dépile *vect*.
- *vect.back()* // Retourne la valeur en haut de *vect*.
- *vect.empty()* // Retourne VRAI lorsque *vect* est vide et FAUX sinon.
- *vect.erase(vect.begin())* //Supprime la première case de *vect*.

Mémento stack (à garder pas loin pour les tps suivants)

- *stack<int> s* // Déclare la variable s comme une pile d'entiers.
- *s.top()* // Accède à la valeur du haut de *s*.
- *s.push(i)* // Empile la valeur i sur *s*.
- *s.pop()* // Dépile *s*.
- *s.empty()* // Retourne VRAI lorsque *s* est vide et FAUX sinon.

Remarques :

— Vous pouvez utiliser les structures dédiées de la STL pour gérer les piles et les files (`stack` et `queue`).

— Pour des simplifications d'écriture (l'important dans ces tps, c'est l'algorithmique, un peu moins la programmation...), tous les tableaux de taille variable sont déclarés dans la pile, ce qui est permis depuis C99. Toutefois, on peut vite remplir l'espace autorisé. Si vous le souhaitez, faîtes des allocations dynamiques, c'est plus performant (en termes de la taille des graphes traitables).

- TP 2. Algorithme de Kruskal. -

Le but de ce TP est de calculer, pour un ensemble V de points du plan, un arbre couvrant $T = (V, A)$ qui vérifie que la somme des longueurs des arêtes de A est minimum. Le calcul de cet arbre s'effectue par l'algorithme de Kruskal.

Langage. Programme en C++. Votre programme pourra commencer de la manière suivante (le code est disponible sur le moodle du cours) :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <fstream>

using namespace std;
typedef struct coord{int abs; int ord;} coord;

int
main()
{
  int n;             //Le nombre de points.
  cout << "Entrer le nombre de points: ";
  cin >> n;
  int m=n*(n-1)/2;   // Le nombre de paires de points.
  coord point[n];    // Les coordonnees des points dans le plan.
  int edge[m][3];    // Les paires de points et le carre de leur longueur.
  int arbre[n-1][2]; // Les aretes de l'arbre de Kruskal.

  return EXIT_SUCCESS;
}
```

- Exercice 1 - Création d'un ensemble aléatoire V de n sommets dans le plan.
L'ensemble de sommets V est $\{0, \ldots, n-1\}$. L'ensemble des positions des éléments de V est stocké dans un tableau de coordonnées **point** de taille n vérifiant que **point**[i].abs est l'abscisse du sommet i, comprise entre 0 et 612, et **point**[i].ord est l'ordonnée du sommet i, comprise entre 0 et 792. Pour information, le format US-Letter, par défaut sur de nombreux afficheur postscript, a pour dimension 612 points par 792 points...

Écrire une fonction *void pointRandom(int n, coord point[])* qui engendre aléatoirement le tableau **point**.

- Exercice 2 - Création du tableau des distances.
Écrire une fonction *void distances(int n, int m, coord point[], int edge[][3])* qui engendre le tableau **edge** de taille $m \times 3$ de telle sorte que :

- Pour chaque paire $\{i, j\}$ avec $i < j$, il existe un k qui vérifie **edge**[k][0] $= i$ et **edge**[k][1] $= j$.
- L'entrée **edge**[k][2] est le carré de la distance euclidienne du point correspondant au sommet i au point correspondant au sommet j.

- Exercice 3 - Tri du tableau edge.
Écrire une fonction *void tri(int m, int edge[][3])* qui trie le tableau **edge**, selon l'ordre croissant des valeurs de **edge**[k][2]. Le but de ce TP n'étant pas le tri, on pourra se limiter à un simple tri à

bulles (tant qu'il existe deux entrées consécutives qui ne sont pas croissantes, on les inverse).

- Exercice 4 - Calcul de l'arbre couvrant de poids minimum.
Écrire une fonction *void kruskal(int n, int edge[][3], int arbre[][2])* qui applique l'algorithme KRUSKAL au tableau d'arêtes **edge** et construit le tableau **arbre** qui contient les $n-1$ arêtes de l'arbre couvrant de poids minimum. On pourra reprendre la fonction *composantes* du TP1, et y apporter des modifications mineures.

- Exercice 5 - Affichage.
Utiliser la fonction *affichageGraphique* (disponible sur le moodle du cours) afin d'afficher le résultat dans le fichier `Exemple.ps`. L'appel se fera par *affichageGraphique(n,point,arbre) ;*

- Exercice 6 - Pour aller plus loin.
Apporter les améliorations ou modifications suivantes :

- Si ce n'est pas déjà le cas, utiliser la version optimisée de l'agorithme COMPOSANTE pour implémenter l'algorithme KRUSKAL.
- Améliorer les performances de votre algorithme en utilisant un tri plus efficace, par exemple tri fusion.
- Montrer que les arêtes de l'arbre obtenu ne peuvent pas se croiser.
- Utiliser d'autres distances (Manhattan, sup,...) pour créer votre arbre.

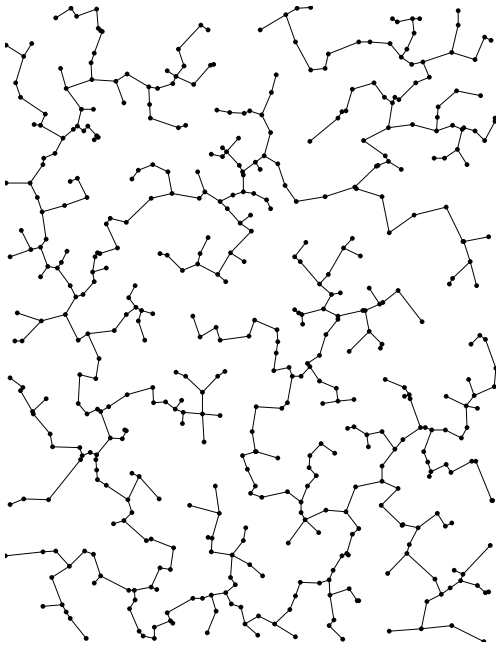

FIGURE 1 – Un exemple d'arbre de Kruskal.

- TP 3. Arbre en largeur et en profondeur. -

Le but de ce TP est de calculer un arbre en largeur et un arbre en profondeur sur un graphe G. L'ensemble des sommets de G est $\{0, \ldots, n-1\}$; les arêtes sont codées par listes de voisinages. Ainsi, à chaque sommet i est associée la pile `voisins[i]` des voisins de i. Votre programme pourra commencer par (le source est disponible sur le moodle) :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <fstream>
using namespace std;

int
main()
{
  int n;                                  // Le nombre de sommets.
  int m;                                  // Le nombre d'aretes.
  cout << "Entrer le nombre de sommets: ";
  cin >> n;
  cout << "Entrer le nombre d'aretes: ";
  cin >> m;
  vector<int> voisins[n];                 // Les listes des voisins.
  int pere[n];                            // L'arbre en largeur.
  int ordre[n];                           // L'ordre de parcours.
  int niveau[n];                          // Le niveau du sommet.

  return EXIT_SUCCESS;
}
```

- Exercice 1 - Création d'un graphe aléatoire.
Écrire une fonction `void voisinsRandom(int n, int m, vector<int>voisins[])` qui engendre aléatoirement les listes de voisins d'un graphe aléatoire de n sommets et m arêtes. On prendra garde à :

- la symétrie: si x est voisin de y, alors y est voisin de x ;
- ne pas créer de boucle ;
- ne pas créer d'arête multiple.

- Exercice 2 - Parcours en largeur.
Implémenter l'algorithme du cours :
`void parcoursLargeur(int n,vector<int> voisins[], int niveau[], int ordre[], int pere[])`
qui effectue un parcours en largeur de racine 0 et calcule pour tout i :

- **pere**$[i]$, représentant le père de i dans l'arbre en largeur, lorsque $i \neq 0$.
- **ordre**$[i]$, représentant la date à laquelle i a été lu en premier.
- **niveau**$[i]$, représentant le niveau de i dans l'arbre.

- Exercice 3 - Écriture des niveaux.
Écrire une fonction `void ecritureNiveaux(int n, int niveau[])` qui écrit le nombre de sommets dans chaque niveau et le nombre de sommets qui ne sont pas joignables à partir de 0. Votre résultat sera de la forme (ici pour $n = 40$ et $m = 80$):

```
Il y a 1 sommets au niveau 0.
Il y a 4 sommets au niveau 1.
```

```
Il y a 12 sommets au niveau 2.
Il y a 17 sommets au niveau 3.
Il y a 4 sommets au niveau 4.
Il y a 2 sommets qui ne sont pas dans la composante de 0.
```

- Exercice 4 - Parcours en profondeur.
Modifier votre algorithme afin de le transformer en parcours en profondeur. Énumérer de même le nombre de sommets sur chaque niveau de l'arbre en profondeur.

- Exercice 5 - Pour aller plus loin.

• Lorsque $m = 2n$, c'est à dire lorsque le degré moyen des sommets du graphe est égal à 4, quel est à votre avis le nombre de niveaux, en fonction de n, de l'arbre en largeur et de l'arbre en profondeur. Faire pour cela des essais avec n de plus en plus grand.

• Au lieu d'un graphe aléatoire, tirer les sommets du graphe au hasard parmi les points de la grille 612x792 et ne garder que les arêtes de longueur inférieure à un certain seuil. Afficher ce graphe en utilisant la fonction d'affichage du TP2. Calculer ensuite un arbre en largeur dans ce graphe et afficher celui-ci. Voir les figures ci-dessous.

• Implémenter le calcul des arêtes séparatrices vu en TD, et tester sur un graphe avec $\frac{3}{2}n$ arêtes.

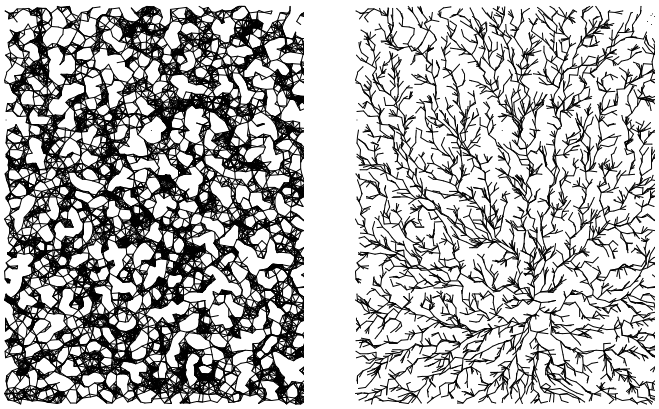

Figure 1: Un bien bel exemple d'arbre en largeur.

Mémento queue (à garder pas loin pour les tps suivants)

- *queue<int> q* // Déclare la variable q comme une file d'entiers.
- *q.front()* // Accède au premier élément de q.
- *q.push(i)* // Enfile i.
- *q.pop()* // Défile q.
- *q.empty()* // Retourne VRAI lorsque q est vide et FAUX sinon.

- TP 4. Plus courts chemins. Algorithme de Dijkstra. -

Le but de ce TP est de calculer un arbre des plus courts chemins (en terme de distance euclidienne) issu d'un sommet dans un graphe G dont les sommets sont des points du plan et les arêtes xy sont toutes les paires $\{x, y\}$ de sommets dont la distance est inférieure à une valeur fixée d_{max}.

Langage. Programme en C++. Votre programme pourra contenir (disponible sur le moodle du cours) :

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <fstream>
#include <cmath>
using namespace std;
typedef struct coord{int abs; int ord;} coord;

const int N=1400;
const int M=(N*(N-1))/2;

void pointRandom(int n,coord point[]);
float distance(coord p1,coord p2);
void voisins(int n,int dmax,coord point[],vector<int> voisin[],int &m);
void voisins2arete(int n,vector<int>voisins[],int arete[][2]);
void affichageGraphique(int n,int m,coord point[],int arete[][2],string name);
bool existe(int n,float dis[],bool traite[],int &x);
void dijkstra(int n,vector<int> voisin[],coord point[],int pere[]);
int construireArbre(int n,int arbre[][2],int pere[]);

int
main()
{
  int n;                            // Le nombre de points.
  cout << "Entrer le nombre de points: ";
  cin >> n;
  int dmax=50;                      // La distance jusqu'a laquelle on relie deux points.
  coord point[N];                   // Les coordonnees des points.
  vector<int> voisin[N];            // Les listes de voisins.
  int arbre[N-1][2];                // Les aretes de l'arbre de Dijkstra.
  int pere[N];                      // La relation de filiation de l'arbre de Dijkstra.
  int m;                            // Le nombre d'aretes
  int arete[M][2];                  // Les aretes du graphe
  return EXIT_SUCCESS;
}
```

- Exercice 1 - Création du graphe.
Reprendre la fonction `void pointRandom(int n,coord point[])` du TP2 qui engendre aléatoirement le tableau **point** représentant un ensemble aléatoire de n points dans le plan. Rappelons que **point** est de taille n, l'abscisse du point i (entre 0 et 612) est stockée dans **point**$[i]$.abs et l'ordonnée (entre 0 et 792) est stockée dans **point**$[i]$.ord.

Écrire une fonction `void voisins(int n,int dmax,coord point[],vector<int> voisin[],int &m)` qui, pour tout sommet i, construit la liste **voisin**$[i]$ vérifiant qu'un point $j \neq i$ apparait dans

voisin[i] si et seulement si la distance euclidienne du point i au point j est au plus égale à d_{max}.

- Exercice 2 - Affichage du graphe.
Écrire `void affichageGraphique(int n,int m,coord point[],int arete[][2],string name)`, inspirée de la fonction d'affichage des TP2 et TP3, qui permet d'afficher le graphe créé dans l'Exercice 1 à l'aide d'un fichier *Graphe.ps*. Tester sur au moins 300 points.

- Exercice 3 - Arbre de Dijkstra.
Écrire une fonction `void dijkstra(int n,vector<int> voisin[],coord point[],int pere[])` sur le modèle de l'algorithme vu en cours. La racine de l'arbre des plus courts chemins est le sommet 0. En sortie, le tableau **pere** représente l'arbre des plus courts chemins. Ainsi, tout sommet i distinct de la racine et accessible depuis celle-ci vérifie que **pere**[i] est différent de -1, valeur donnée à l'initialisation.

- Exercice 4 - Affichage de l'arbre.
Écrire une fonction `int construireArbre(int n,int arbre[][2],int pere[])` qui remplit le tableau **arbre** avec toutes les arêtes i**pere**[i] et retourne le nombre k de ces arêtes (i.e. le nombre de points accessibles depuis la racine moins un).
Utiliser la fonction `void affichageGraphique(int n,int m,coord point[],int arete[][2],string name)` pour créer un fichier *Arbre.ps* qui représente l'arbre.

- Exercice 5 - Pour aller plus loin.
Répondre ou améliorer les points suivants :

- Lorsque d_{max} est très grand, que constatez-vous ?
- Les arêtes de l'arbre de Dijkstra peuvent-elles se couper ?
- Essayer des métriques différentes (sup, manhattan).

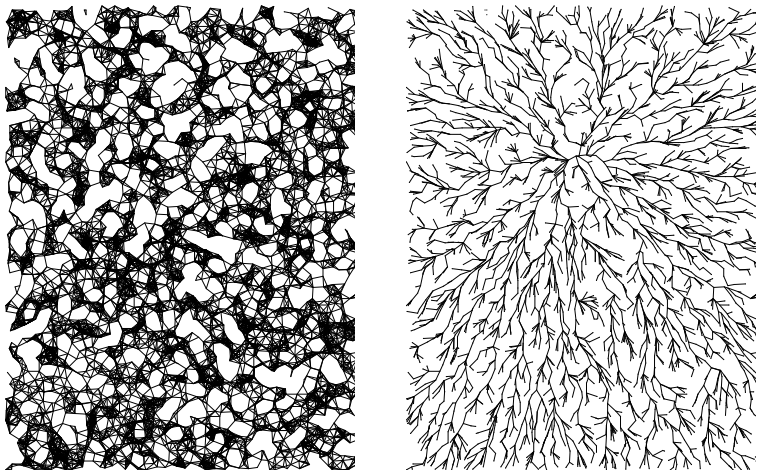

FIGURE 1 – Un exemple d'arbre de plus courts chemins.

- TP 5. Plus courts chemins entre tous couples. Algorithme de Floyd-Warshall -

Le but de ce TP est de calculer l'ensemble des plus courts chemins entre tous les couples de sommets d'un graphe orienté $D = (V, A)$, puis d'appliquer ce calcul à la recherche de la fermeture transitive d'un graphe orienté.

Langage. Programme en C++. Votre programme pourra contenir (disponible sur le moodle du cours) :

```
#include <iostream>
#include <vector>
using namespace std;

const int N=5;
const int INF=9999;                      //La valeur infinie.

void floydWarshall(int longueur[][N],int dist[][N],int chemin[][N]);
void affichage(int dist[][N],int chemin[][N]);
void itineraire(int i,int j,int chemin[][N]);

int
main()
{

  int longueur[N][N]={{0,2,INF,4,INF},   //Les longueurs des arcs.
                      {INF,0,2,INF,INF}, //longueur[i][j]=INF si l'arc ij n'existe pas
                      {INF,INF,0,2,INF},
                      {INF,-3,INF,0,2},
                      {2,INF,INF,INF,0}};
  int dist[N][N];                        //Le tableau des distances.
  int chemin[N][N];                      //Le tableau de la premiere etape du chemin de i a j.
  floydWarshall(longueur,dist,chemin);
  affichage(dist,chemin);
  return EXIT_SUCCESS;
}
```

- Exercice 1 - À quoi ça sert les cours de graphes ?
On trouve à cette adresse : `http://about-france.com/france-rail-map.htm` un plan des principales lignes de train en France. Essayer de trouver un itinéraire nécessitant strictement plus de deux changements et trouver le trajet correspondant sur le site `https://www.oui.sncf/`. À titre d'exemple, on pourra essayer *Dinan-Mende* ou *Lons le Saunier-Guéret*. Faites la même requête sur le site `https://www.bahn.com/fr/view/index.shtml` ...

- Exercice 2 - Floyd-Warshall.
Initialiser le tableau **dist** à **longueur**.
Écrire une fonction `void floydWarshall(int longueur[][N], int dist[][N])` qui construit le tableau **dist** dont chaque entrée **dist**$[i][j]$ est la longueur minimale d'un chemin de i à j, et vaut *INF* si un tel chemin n'existe pas. Afficher ensuite le tableau **dist**, et vérifier la solution obtenue (à la main...).

- Exercice 3 - Calcul des chemins.
Initialiser le tableau **chemin** de telle sorte que :

— **chemin**$[i][j] = j$ lorsque ij est un arc.
— **chemin**$[i][j] = -1$ dans les autres cas.

Modifier ensuite la fonction `floydWarshall` de sorte que, lors du calcul du tableau **dist**, le tableau **chemin** soit aussi calculé, **chemin**$[i][j]$ devant contenir le voisin sortant de i le long d'un plus court chemin de i à j, ou -1 si un tel chemin n'existe pas. Afficher ensuite le tableau **chemin** et vérifier qu'il soit correct (à la main...).

- Exercice 4 - Calcul d'un itinéraire.

Écrire une fonction `void itineraire(int i, int j, int chemin[][N])` qui, prenant en entrée deux sommets du graphe, affiche un plus court chemin de i à j. L'appel à cette fonction affichera :

```
Entrer le depart : 2
Entrer la destination : 4
L'itineraire est : 2 3 4
```

Faites de même avec le réseau routier proposé sur le moodle du cours.

- Exercice 5 - Fermeture transitive.

La *fermeture transitive* d'un graphe orienté D est une matrice **fermeture** vérifiant **fermeture**$[i][j] = 1$ s'il existe un chemin orienté de i à j dans D, et **fermeture**$[i][j] = 0$ sinon.

En vous inspirant de la fonction `floydWarshall`, écrire une fonction `void fermetureTransitive(int arc[][N], int fermeture[][N])` qui calcule le tableau **fermeture**, le tableau **arc** étant la matrice d'adjacences d'un graphe orienté D. Tester votre fonction sur l'exemple suivant :

```
const int N=6;
int arc[N][N]={{0,0,0,1,0,1},//La matrice d'adjacences du graphe oriente D.
               {1,0,1,1,0,0},
               {0,0,0,1,0,0},
               {0,0,0,0,1,1},
               {0,0,1,0,0,1},
               {0,0,1,0,0,0}};
int fermeture[N][N];         // La matrice de la fermeture transitive de D.
```

- Exercice 6 - Composantes fortement connexes.

Écrire une fonction `void compFortConnexe(int n, int fermeture[][N])` qui affiche les composantes fortement connexes du graphe D. Le résultat pourra être de la forme :
`Les composantes fortement connexes sont : {1,4}, {2}, {3,5,6}, {7}`

- Exercice 7 - Pour aller plus loin.

Concernant le calcul de la fermeture transitive :

- Afficher les composantes fortement connexes de telle sorte que si une composante C apparaît avant C', il n'existe pas d'arc de C' vers C.