Inf 431 – Cours 1

Parcours de graphes

jeanjacqueslevy.net

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX,
01 69 33 34 67

www.enseignement.polytechnique.fr/informatique

Objectifs de INF 431

- Principes de la Programmation (2ème partie)
- Modularité – Programmation par objets
- Graphes, Grammaires, Concurrence
- Initiation à l'informatique scientifique

Format du cours

- Début encadré (Amphi + Petite Classe + TD)
- Milieu (Amphi + Petite Classe + Devoir Maison)
- Fin (Amphi + Projet Informatique)

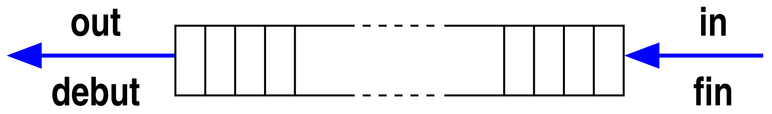
Notation

- CC 4h + PI ; 2 DMs
- $\boxed{\text{note\_module} = \max\{\text{CC}, (12 * \text{CC} + 6 * \text{PI})/18\}}$
  $\text{note\_litérale} = \text{note\_module} + \text{DMs}/6$
  $\text{DMs} = 6(A), 4(B), 2(C), 1(D), 0(E)$

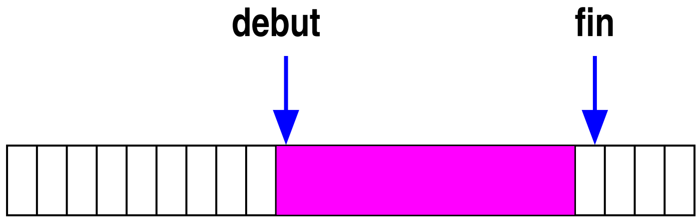
Plan

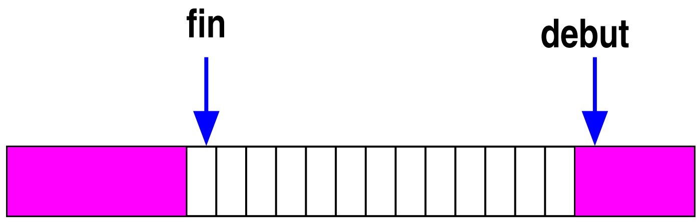
1. Files d'attente
2. Piles
3. Graphes
4. Représentation des graphes
5. Parcours en profondeur
6. Parcours en largeur
7. Arbres de recouvrement
8. Sortie de labyrinthe

File d'attente (1/4)

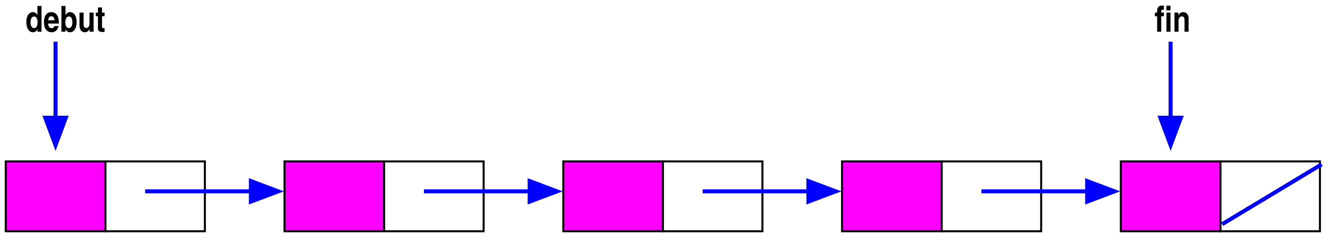


Deux représentations. Dans un tableau (tampon circulaire).





ou par une liste.



File d'attente (2/4)

Premier arrivé, premier servi (*First In, First Out*).

Structure de données très fréquente dans les programmes : par exemple dans les OS, le temps-réel, . . . et le *hardware*.

Une première méthode compacte consiste à gérer un tampon circulaire.

```
class FIFO {
  int debut, fin;
  boolean pleine, vide;
  int[ ] contenu;

  FIFO (int n) {
    debut = 0; fin = 0; pleine = n == 0; vide = true;
    contenu = new int[n];
  }
```

File d'attente (3/4)

```
static void ajouter (FIFO f, int x) {
  if (f.pleine)
      throw new Error ("File Pleine.");
  f.contenu[f.fin] = x;
  f.fin = (f.fin + 1) % f.contenu.length;
  f.vide = false; f.pleine = f.fin == f.debut;
}

static int supprimer (FIFO f) {
  if (f.vide)
      throw new Error ("File Vide.");
  int res = f.contenu[f.debut];
  f.debut = (f.debut + 1) % f.contenu.length;
  f.vide = f.fin == f.debut; f.pleine = false;
  return res;
}
```

Belle symétrie. Taille $\simeq n$.

Taille bornée (structure de donnée statique).

Complexité de `ajouter` et `supprimer` en $O(1)$.

Rappel de notions de Java

Un programme de test avec comme arguments :
la taille, les éléments à ajouter, les ordres de suppression

Exemple d'exécution :

```
% javac FIFO.java
% java FIFO 10 3 4 5 - - 7 8 - - 9
3
4
5
7
```

```
public static void main (String[ ] args) {
  int n = Integer.parseInt (args[0]);
  FIFO f = new FIFO (n);
  for (int i = 1; i < args.length; ++i)
    if (args[i].equals ("-") )
      System.out.println (supprimer(f));
    else {
      int x = Integer.parseInt (args[i]);
      ajouter (f, x);
    }
}
```

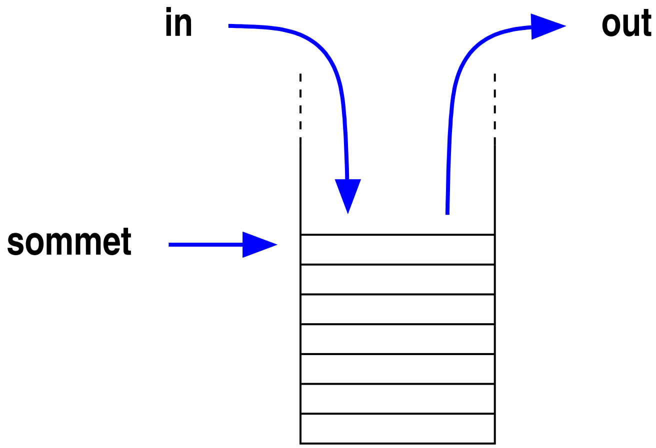
File d'attente (4/4)

```
class FIFO {
  Liste debut, fin;
  FIFO () { debut = null; fin = null; }

  static void ajouter (FIFO f, int x) {
    if (f.fin == null) f.debut = f.fin = new Liste (x);
    else {
      f.fin.suivant = new Liste (x);
      f.fin = f.fin.suivant;
  } }

  static int supprimer (FIFO f) {
    if (f.debut == null) throw new Error ("File Vide.");
    else {
      int res = f.debut.val;
      if (f.debut == f.fin) f.debut = f.fin = null;
      else f.debut = f.debut.suivant;
      return res;
  } }
```

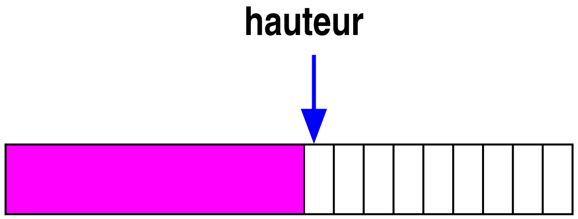
Taille non bornée (structure de donnée dynamique) en $2n$.

Complexité de `ajouter` et `supprimer` en $O(1)$.

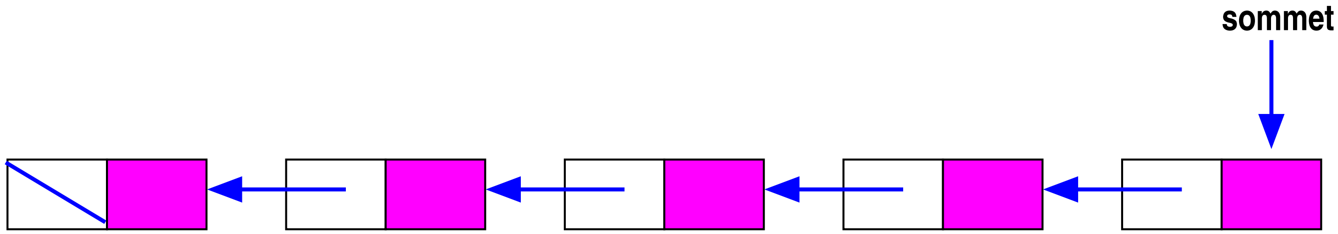
Pile (1/3)



Deux représentations. Dans un tableau



ou par une liste.



Pile (2/3)

Dernier arrivé, premier servi (Last In, First Out).
Utile en compilation, en informatique théorique.

```
class Pile {
  int hauteur ;
  int[ ] contenu;
  Pile (int n) {hauteur = 0; contenu = new int[n]; }

  static void empiler (int x, Pile p) {
    if (p.hauteur >= p.contenu.length)
      throw new Error ("Pile pleine.");
    p.contenu[p.hauteur++] = x;
  }

  static int depiler (Pile p) {
    if (p.hauteur <= 0)
      throw new Error ("Pile vide.");
    return p.contenu [--p.hauteur];
  }
```

Taille bornée (structure de donnée statique).

Complexité de `empiler` et `depiler` en $O(1)$.

Pile (3/3)

```
class Pile {
  Liste sommet;
  Pile () { sommet = null; }

  static void empiler (int x, Pile p) {
    p.sommet = new Liste (x, p.sommet);
  }

  static int depiler (Pile p) {
    if (p.sommet == null) throw new Error ("Pile vide.");
    int res = p.sommet.val;
    p.sommet = p.sommet.suivant;
    return res;
  }
```

Taille non bornée (structure de donnée dynamique).

Complexité de `empiler` et `depiler` en $O(1)$.

Loi [Randell et Russel, 1960]

Récursif = Itératif + Pile

⇒ premiers compilateurs

Graphe (1/3)

Un graphe $G = (V, E)$ a un ensemble de sommets V et d'arcs $E \subset V \times V$. Un arc $e = (v_1, v_2)$ a pour origine $org(e) = v_1$ et pour extrémité $ext(e) = v_2$.

Un graphe est une relation binaire sur ses sommets.

Exemples : les rues de Paris, le plan du métro.

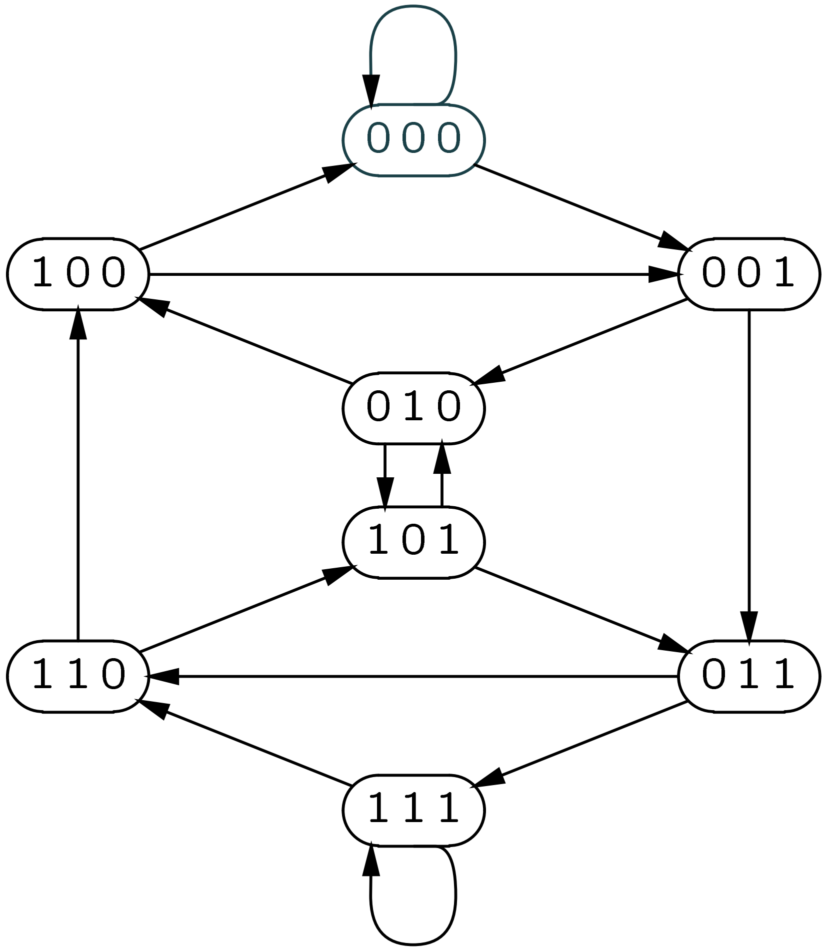
$G = (V, E)$ est un graphe non orienté ssi $(v_1, v_2) \in E$ implique $(v_2, v_1) \in E$. Par exemple, les couloirs de l'X.

Un chemin est une suite d'arcs e_1, e_2, $\ldots e_n$, telle que $ext(e_i) = org(e_{i+1})$ pour $1 \leq i < n$, où $n \geq 0$. Un circuit (ou cycle) est un chemin où $ext(e_n) = org(e_1)$.

Les *dag* (*directed acyclic graphs*) sont des graphes orientés sans cycles. Exemple : le graphe des dépendances entre modules pour la création d'un projet informatique. (*Makefile*)

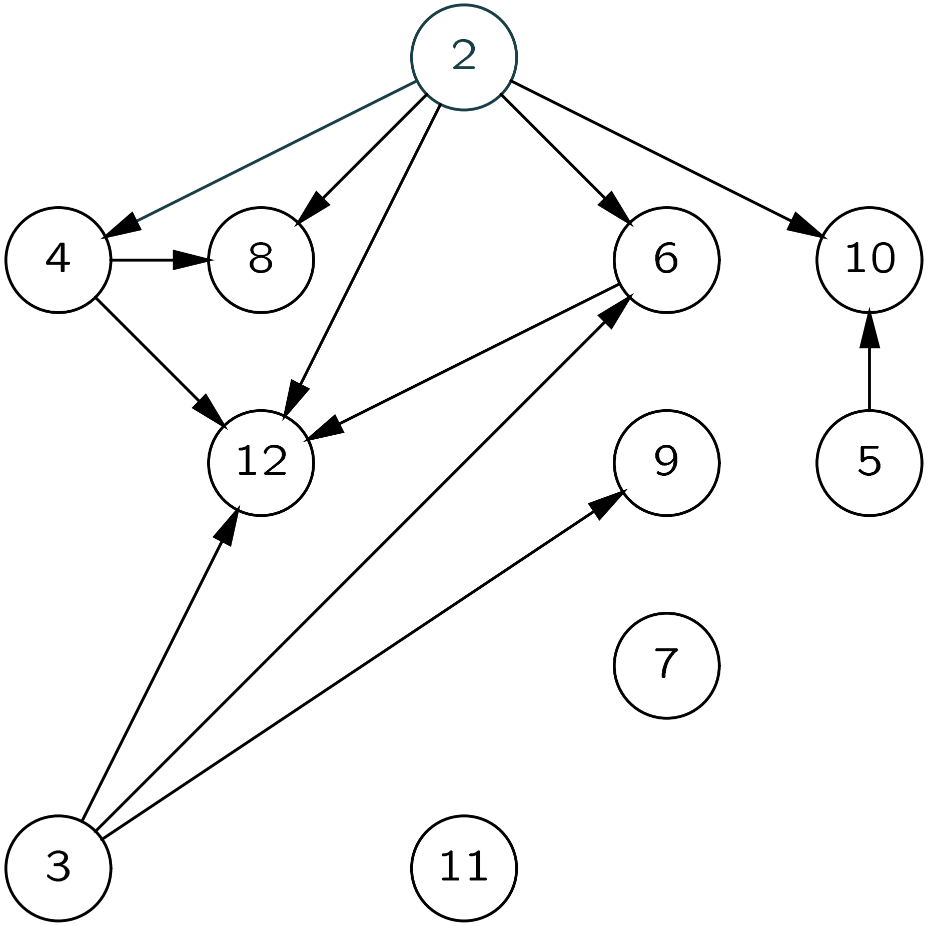
Un arbre est un *dag*. Une forêt (ensemble d'arbres) est un *dag*.

Graphe (2/3)



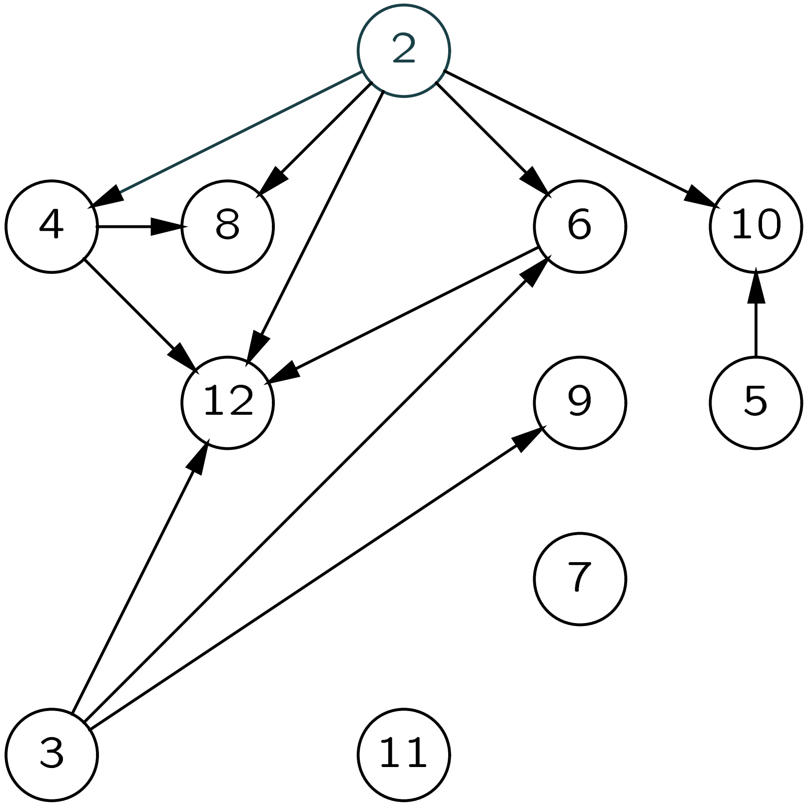
Graphe de de Bruijn

Graphe (3/3)



Graphe des diviseurs

Représentation d'un graphe (1/4)



Matrice d'adjacence

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
$$

Représentation d'un graphe (2/4)

Matrice d'adjacence $M = (m_{i,j})$
où $m_{i,j} = 1$ si (v_i, v_j) est un arc du graphe, sinon $m_{i,j} = 0$.

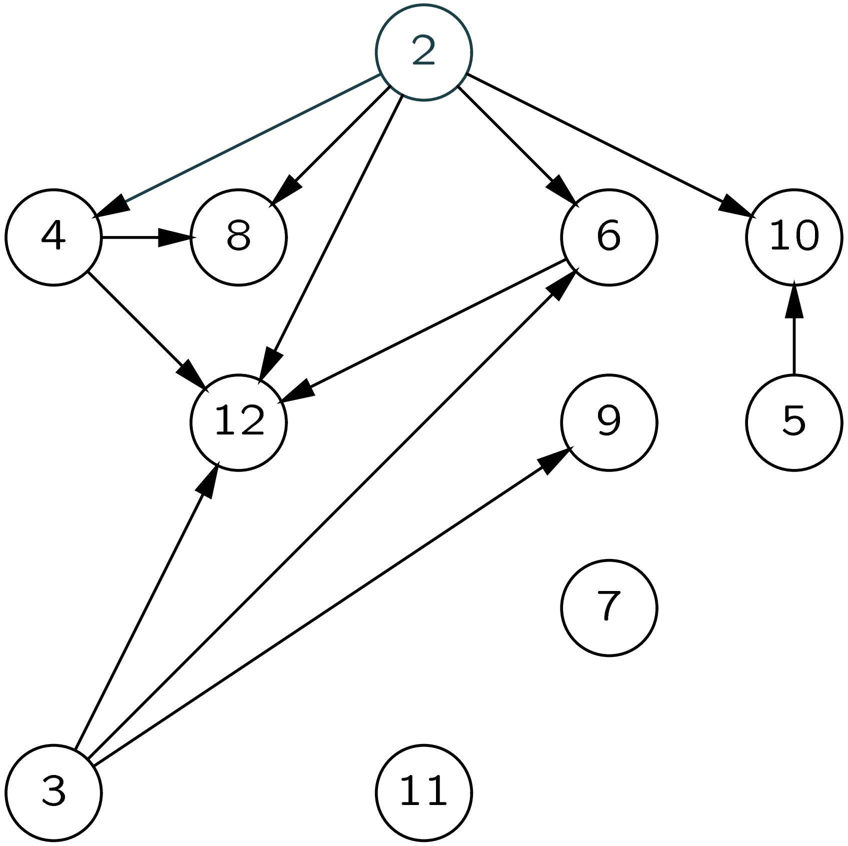
Matrice symétrique pour un graphe non orienté.

```
class Graphe {
  boolean[ ][ ] m;

  Graphe (int n) {
    m = new boolean[n][n];
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < n; ++j)
        m[x][y] = false;
  }

  static void ajouterArc (Graphe g, int x, int y) { g.m[x][y] = true; }
}
```

Place mémoire $O(V^2)$

(En fait l'initialisation de m est inutile, car c'est l'option par défaut en Java)

Représentation d'un graphe (3/4)




```
g.succ[0] = null;
g.succ[1] = null;
g.succ[2] = {4,6,8,12,10};
g.succ[3] = {6,9,12};
g.succ[4] = {8,12};
g.succ[5] = {10};
g.succ[6] = {12};
g.succ[7] = null;
g.succ[8] = null;
g.succ[9] = null;
g.succ[10] = null;
g.succ[11] = null;
g.succ[12] = null;
```

Remarque : `{4,6,8,10,12}` n'est malheureusement pas du Java légal, mais est ici une abréviation pour

`new Liste(4, new Liste(6, new Liste(8, new Liste(10, new Liste (12, null)))))`

Représentation d'un graphe (4/4)

Tableau de listes de successeurs
(Représentation creuse de la matrice d'adjacence)

```
class Graphe {
  Liste[ ] succ;

  Graphe (int n) { succ = new Liste[n]; }

  static void ajouterArc (Graphe g, int x, int y) {
    g.succ[x] = new Liste (y, g.succ[x]);
  }
}
```

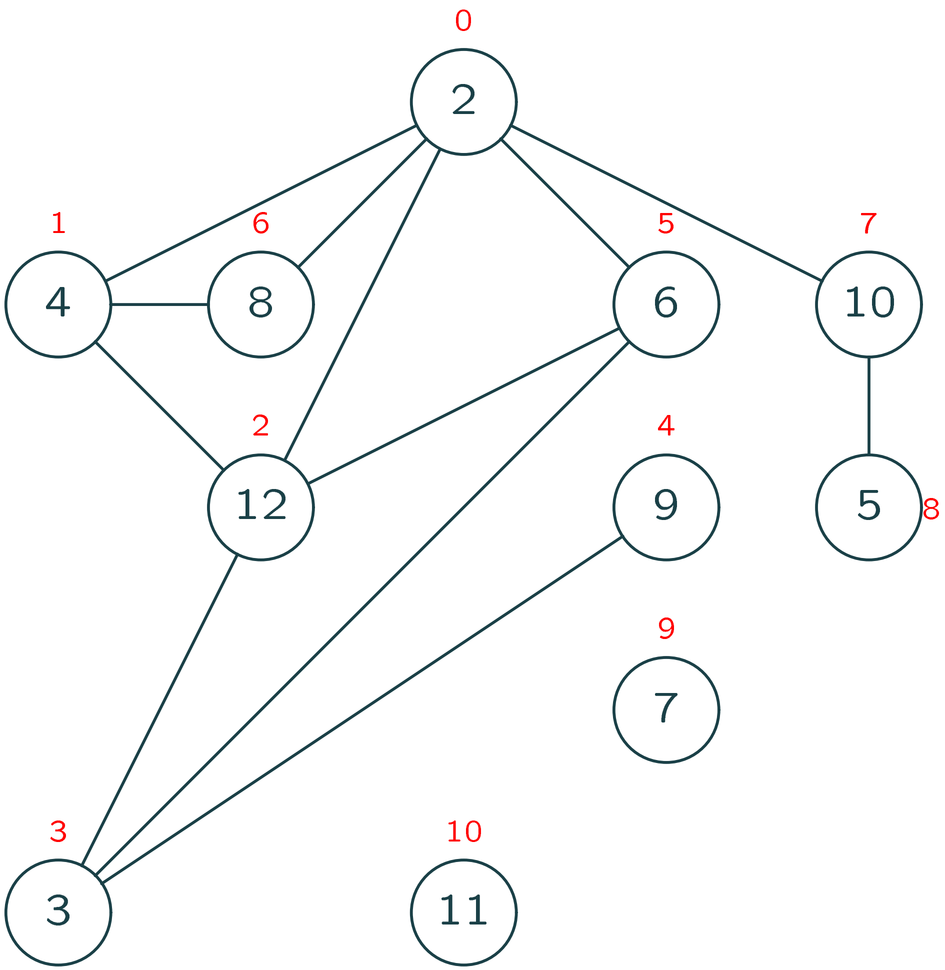
Place mémoire en $O(V+E)$

Entrée textuelle d'un graphe

1ère ligne : $n = card(V)$; lignes suivantes : arcs $x_i \leftrightarrow y_i$

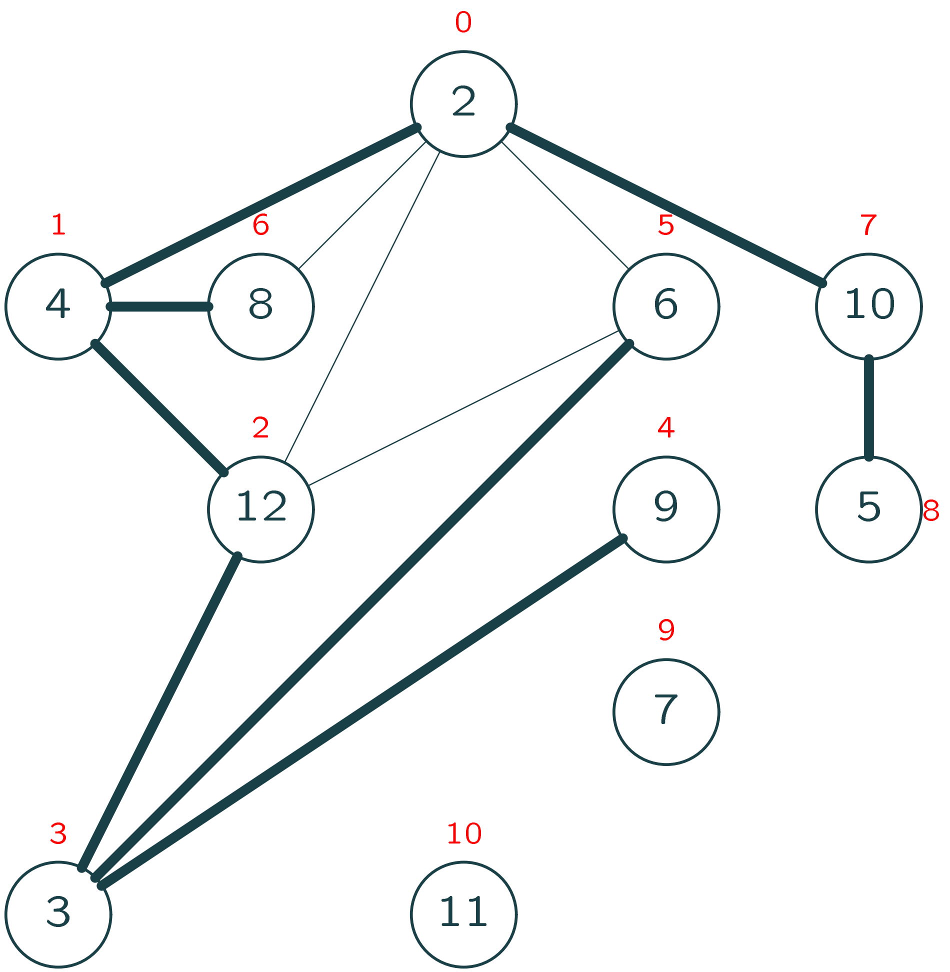
```
static Graphe lireGraphe () {
  BufferedReader in = // idiosyncratie Java !
     new BufferedReader(new InputStreamReader(System.in));
  try {
    String s = in.readLine(); int n = Integer.parseInt(s);
    Graphe g = new Graphe (n);
    while ((s = in.readLine()) != null) {
      StringTokenizer st = new StringTokenizer(s);
      int x = Integer.parseInt(st.nextToken());
      int y = Integer.parseInt(st.nextToken());
      if (0 <= x && x < n && 0 <= y && y < n) {
        ajouterArc(g, x, y);
        ajouterArc(g, y, x);
      }
    }
    return g;
  } catch (IOException e) {
    System.err.println(e); System.exit(1); return null;
} }
```

Arbre de recouvrement (1/4)



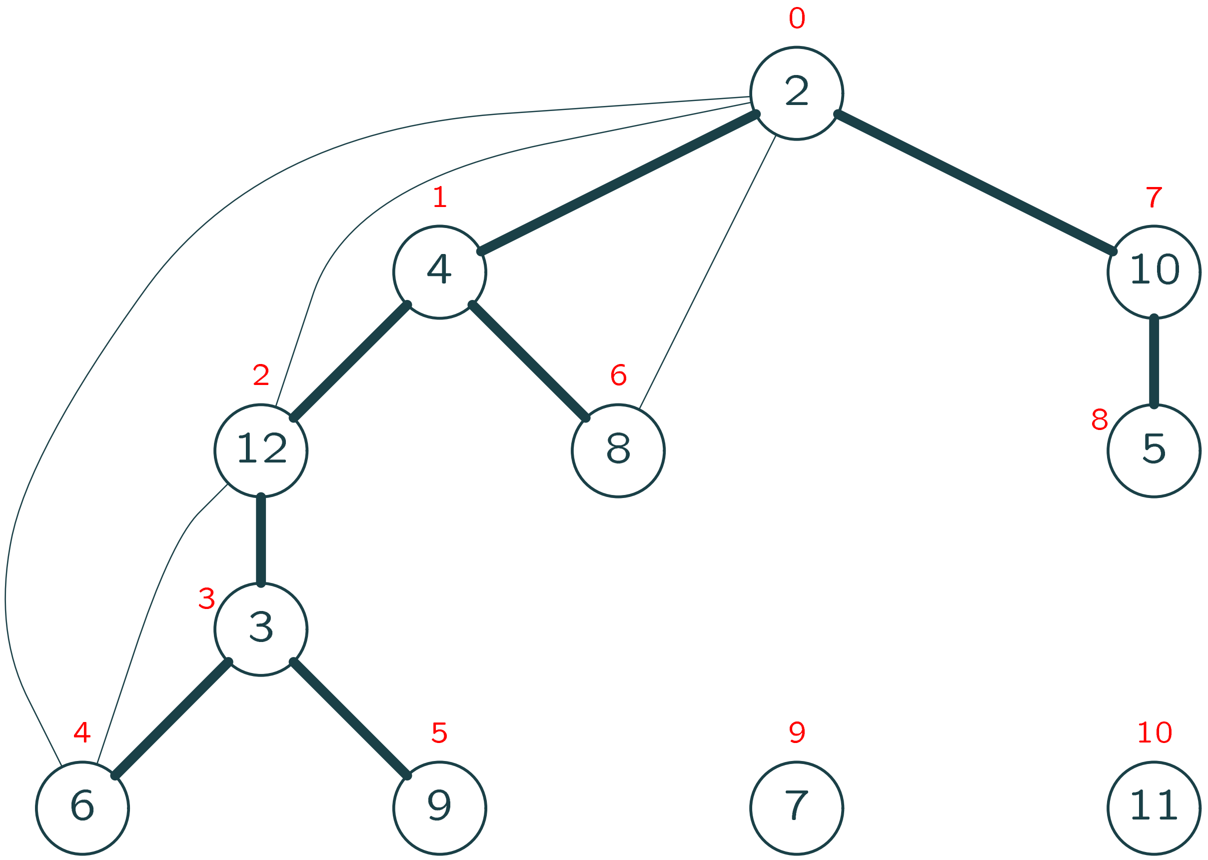
Comment visiter tous les sommets sans boucler ?

Arbre de recouvrement (2/4)



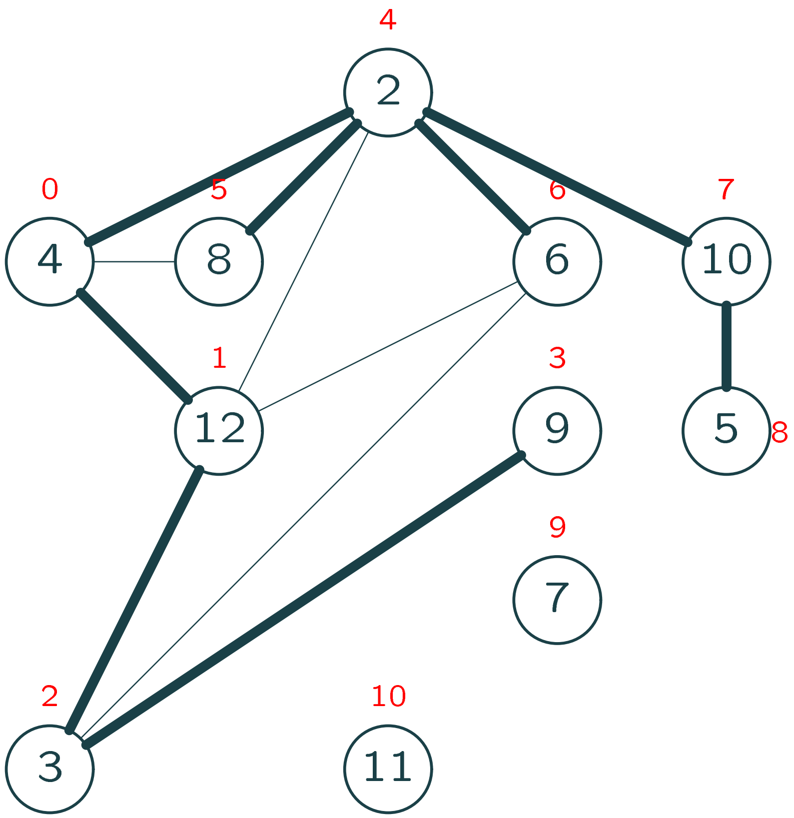
⇒ Trouver une forêt qui recouvre tous les sommets du graphe.

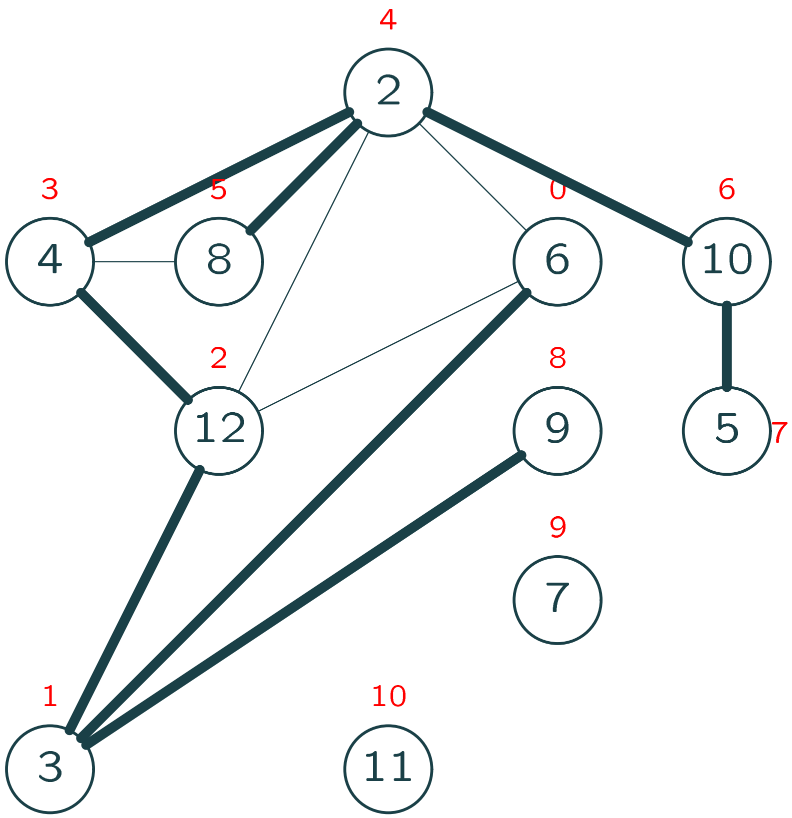
Arbre de recouvrement (3/4)



Vue de l'arbre de recouvrement avec des numéros correspondant à l'ordre préfixe sur l'arbre de recouvrement.

Arbre de recouvrement (4/4)





Un graphe peut avoir plusieurs arbres de recouvrement.

Parcours en profondeur (1/6)

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;
static int[ ] couleur;

static void visiter (Graphe g) {
  int n = g.succ.length; couleur = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC)
      dfs(g, x);
}

static void dfs (Graphe g, int x) {
  couleur[x] = GRIS;
  Pour tout y successeur de x dans G
  faire {
    if (couleur[y] == BLANC)
      dfs(g, y);
  }
  couleur[x] = NOIR;
}
```

BLANC = non traité, NOIR = traité,

GRIS = en cours de traitement. 1 2 3 4

Parcours en profondeur (2/6)

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;
static int[ ] couleur;

static void visiter (Graphe g) {
  int n = g.succ.length; couleur = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC)
      dfs(g, x);
}

static void dfs (Graphe g, int x) {
  couleur[x] = GRIS;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (couleur[y] == BLANC)
      dfs(g, y);
  }
  couleur[x] = NOIR;
}
```

BLANC = non traité, NOIR = traité,

GRIS = en cours de traitement. 1 2 3 4

Parcours en profondeur (3/6)

```
static int numOrdre;
static int[ ] num;

static void visiter (Graphe g) {
  int n = g.succ.length; num = new int[n]; numOrdre = -1;
  for (int x = 0; x < n; ++x) num[x] = -1;
  for (int x = 0; x < n; ++x)
    if (num[x] == -1)
      dfs(g, x);
}

static void dfs (Graphe g, int x) {
  num[x] = ++numOrdre;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (num[y] == -1)
      dfs(g, y);
  }
}
```

Son temps est en $O(V + E)$.

[Tarjan (1972)] a démontré l'utilité de cette méthode. 1 2 3 4

Parcours en profondeur (4/6)

```
static int numOrdre;


static void visiter (Graphe g) {
  int n = g.succ.length; int[ ] num = new int[n]; numOrdre = -1;
  for (int x = 0; x < n; ++x) num[x] = -1;
  for (int x = 0; x < n; ++x)
    if (num[x] == -1)
      dfs(g, x, num);
}

static void dfs (Graphe g, int x, int[ ] num) {
  num[x] = ++numOrdre;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (num[y] == -1)
      dfs(g, y, num);
  }
}
```

Mieux car moins de variables globales.

Exercice 1 Supprimer la variable globale *numOrdre*.

Parcours en profondeur (5/6)

```
final static int BLANC = 0, GRIS = 1, NOIR = 2;


static void visiter (Graphe g) {
  int n = g.succ.length; int[ ] couleur = new int[n];
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC)
      dfs(g, x, couleur);
}

static void dfs (Graphe g, int x, int[ ] couleur) {
  couleur[x] = GRIS;
  for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
    int y = ls.val;
    if (couleur[y] == BLANC)
      dfs(g, y, couleur);
  }
  couleur[x] = NOIR;
}
```

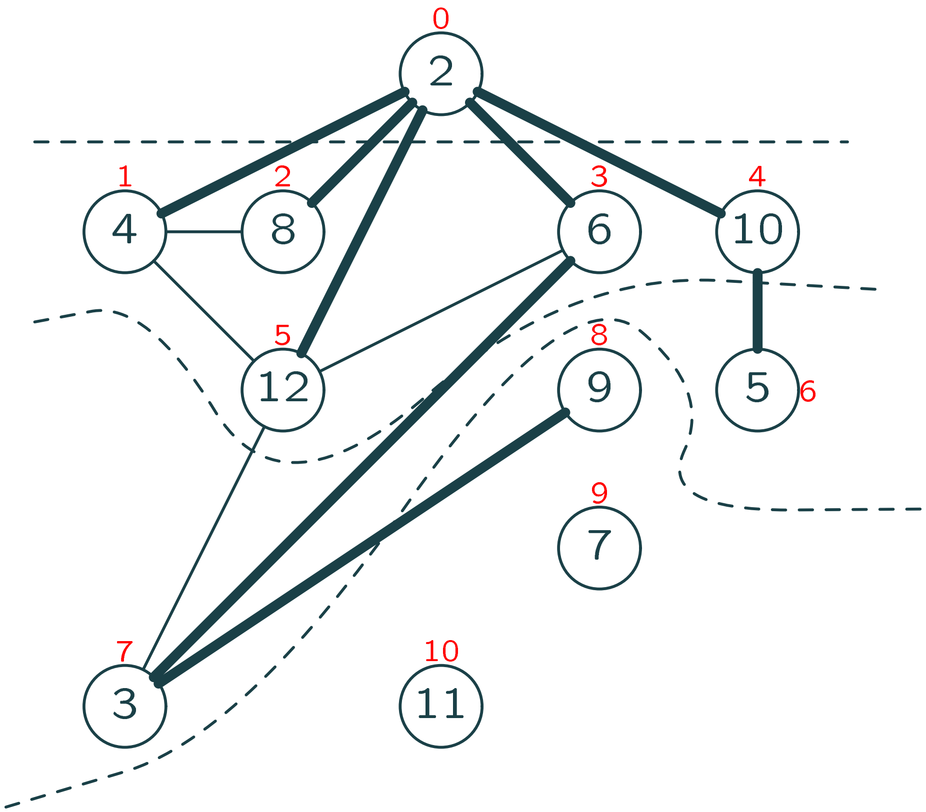
Pas de variables globales $\Rightarrow$ modularité.

Parcours en profondeur (6/6)

```
static void visiter (Graphe g) {
  int n = g.succ.length; couleur = new int[n];
  Pile p = new Pile (n);
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC) {
      Pile.empiler(p, x); couleur[x] = GRIS;
      dfs (g, p);
} }

static void dfs (Graphe g, Pile p) {
  while ( !p.vide() ) {
    int x = Pile.depiler (p);
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if (couleur[y] == BLANC) {
        Pile.empiler(f, y); couleur[y] = GRIS;
      }
    couleur[x] = NOIR;
} } }
```

Itératif ⇒ plus compliqué que récursif.

Parcours en largeur (1/2)



Parcours selon la distance à un sommet de départ.

Parcours en largeur (2/2)

```
static void visiter (Graphe g) {
  int n = g.succ.length; couleur = new int[n];
  FIFO f = new FIFO (n);
  for (int x = 0; x < n; ++x) couleur[x] = BLANC;
  for (int x = 0; x < n; ++x)
    if (couleur[x] == BLANC) {
      FIFO.ajouter(f, x); couleur[x] = GRIS;
      bfs (g, f);
} }

static void bfs (Graphe g, FIFO f) {
  while ( !f.vide() ) {
    int x = FIFO.supprimer (f);
    for (Liste ls = g.succ[x]; ls != null; ls = ls.suivant) {
      int y = ls.val;
      if (couleur[y] == BLANC) {
        FIFO.ajouter(f, y); couleur[y] = GRIS;
      }
      couleur[x] = NOIR;
} } }
```

Mêmes programmes itératifs : DFS $\equiv$ pile   BFS $\equiv$ file.   1 2 3 4

Sortie de labyrinthe

On cherche un chemin de d à s. Exécution

```
static Liste chemin (Graphe g, int d, int s, int[ ] couleur) {
  couleur[d] = GRIS;
  if (d == s)
    return new Liste (d, null);
  for (Liste ls = g.succ[d]; ls != null; ls = ls.suivant) {
    int x = ls.val;
    if (num[x] == BLANC) {
      Liste r = chemin (g, x, s);
      if (r != null)
        return new Liste (d, r);
    }
  }
  return null;
}
```

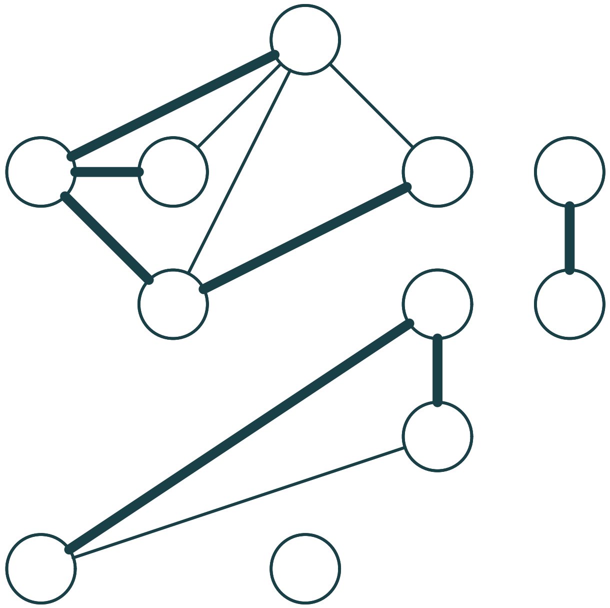
Complexité en temps en $O(V + E)$

Exercice 2 Déterminer le chemin le plus court vers la sortie.

Exercice 3 Imprimer tous les chemins vers la sortie.

Connexité

Dans un graphe non-orienté, une composante connexe est un ensemble maximal de sommets reliés entre eux.



Exercice 4 Quelles sont les composantes connexes dans le graphe des diviseurs (non-orienté).

Exercice 5 Ecrire le programme qui imprime les composantes connexes dans un graphe non-orienté.

Exercices

Exercice 6 Ecrire `dfs` pour un graphe représenté par sa matrice d'adjacence.

Exercice 7 Montrer que la signification des couleurs n'est pas tout à fait la même entre le parcours en profondeur récursif et celui itératif avec pile. Montrer aussi que le parcours n'est pas le même non plus.

Exercice 8 Corriger le parcours itératif pour obtenir le même résultat que le parcours récursif. Faire de même avec les parcours produisant des numérotations des sommets.