Master Informatique - M1 - UE Complexité
Chapitre 2 : Rappels, algorithmique et complexité

Philippe Jégou

Laboratoire d'Informatique et Systèmes - LIS - UMR CNRS 7020

Équipe COALA - COntraintes, ALgorithmes et Applications

Campus de Saint-Jérôme

Département Informatique et Interactions

Faculté des Sciences

Université d'Aix-Marseille

philippe.jegou@univ-amu.fr

7 septembre 2020

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Objectif : savoir si un algorithme puis un programme l'implémentant, seront efficaces en termes de temps d'exécution

$\Rightarrow$ estimer les temps de calcul

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Objectif : savoir si un algorithme puis un programme l'implémentant, seront efficaces en termes de temps d'exécution

$\Rightarrow$ estimer les temps de calcul

Mais :

- implémenter peut coûter tres cher (hommes/mois)
- il faut estimer le temps avant d'implémenter !

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Objectif : savoir si un algorithme puis un programme l'implémentant, seront efficaces en termes de temps d'exécution

$\Rightarrow$ estimer les temps de calcul

Mais :

- implémenter peut coûter tres cher (hommes/mois)
- il faut estimer le temps avant d'implémenter !

Remarques :

- algorithme : c'est un objet abstrait donc temps de calcul "théorique"
- programme sur un environnement matériel/systeme donné :
  temps de calcul "physique" (temps au chronomètre)

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ? Pas impossible... mais sur quels jeux de données ?

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ? Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
  - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ?
  Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
  - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données
  - tri à bulles : efficacité qui évolue très mal avec la taille de la donnée
    - si 15 minutes sur un tableau d'une certaine taille
    - quel temps pour un tableau contenant 10 fois plus d'élements ?

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ? Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
  - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données
  - tri à bulles : efficacité qui évolue très mal avec la taille de la donnée
    - si 15 minutes sur un tableau d'une certaine taille
    - quel temps pour un tableau contenant 10 fois plus d'élements ?

    10 fois plus de temps (150 minutes = 2h30) ?

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

- Comment évaluer le temps de calcul "physique" ?
- Faire des expérimentations, des tests, sur la configuration prévue ? Pas impossible... mais sur quels jeux de données ?
- Cas des tris :
  - tri rapide (Quicksort) : très efficace en général, mais très mauvais sur certains jeux de données
  - tri à bulles : efficacité qui évolue très mal avec la taille de la donnée
    - si 15 minutes sur un tableau d'une certaine taille
    - quel temps pour un tableau contenant 10 fois plus d'élements ?

    10 fois plus de temps (150 minutes = 2h30) ?

    Non : 100 fois plus de temps ! (1500 minutes = 25 heures)

    Le temps de calcul du tri à bulles croît de façon quadratique (en n^2) en fonction de la taille n du tableau à trier (donc pas de façon linéaire)

Motivations : Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver a concevoir les algorithmes les plus rapides possibles avant d'implémenter

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver a concevoir les algorithmes les plus rapides possibles avant d'implémenter
- Donc, s'affranchir des aspects matériels dans un premier temps

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver a concevoir les algorithmes les plus rapides possibles avant d'implémenter
- Donc, s'affranchir des aspects matériels dans un premier temps
- Il faut s'appuyer sur un modéle theorique d'évaluation du temps d'exécution

Motivations :
Pourquoi l'Analyse de la Complexité des Algorithmes ?

Analyse de la Complexité des Algorithmes

- Évaluer le temps de calcul "théorique" d'un algorithme
- Si un algorithme est inefficace : ne pas perdre de temps à l'implémenter
- Arriver a concevoir les algorithmes les plus rapides possibles avant d'implémenter
- Donc, s'affranchir des aspects matériels dans un premier temps
- Il faut s'appuyer sur un modéle theorique d'évaluation du temps d'exécution

Remarque : on laisse de côté la question du coût en ressource mémoire (complexité en espace) au profit du coût en temps (complexité en temps) crucial dans cette UE.

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)
- être utilisable pour évaluer au mieux les temps d'exécution des programmes
  (un programme est un algorithme écrit dans un langage de programmation)
- envisager tous les cas de figures possibles en termes de données en entrée
- considérer l'accroissement de la taille de la donnée en entrée

Un modèle d'analyse fondé sur quelques principes

Objectif : introduire un modèle d'analyse du temps théorique d'exécution des algorithmes

Ce modèle doit :

- s'affranchir des contingences matérielles
  (i.e. être indépendant des configurations machines et systèmes)
- être utilisable pour évaluer au mieux les temps d'exécution des programmes
  (un programme est un algorithme écrit dans un langage de programmation)
- envisager tous les cas de figures possibles en termes de données en entrée
- considérer l'accroissement de la taille de la donnée en entrée

Ce modèle d'analyse est fondé sur quelques principes simples (la suite)

Principe 1 : Actions élémentaires en temps constant

Principe 1 : Actions élémentaires exécutables en temps constant

On identifie 4 types d''actions élémentaires :

- **Affectations :** par exemple **y = x;** dont le coût d'exécution est une constante $k_a \in \mathbb{R}^{*+}$ avec
  - $k_a \neq 0$: le temps peut être tres petit mais il ne peut être nul
  - $k_a > 0$: le temps ne peut être négatif

Principe 1 : Actions élémentaires en temps constant

Principe 1 : Actions élémentaires exécutables en temps constant

On identifie 4 types d''actions élémentaires :

- **Affectations :** par exemple **y = x;** dont le coût d'exécution est une constante $k_a \in \mathbb{R}^{*+}$ avec
  - $k_a \neq 0$: le temps peut être tres petit mais il ne peut être nul
  - $k_a > 0$: le temps ne peut être négatif
- **Opérations arithmétiques :** par exemple l'addition dans **x = a+n;** de coût $k_o \in \mathbb{R}^{*+}$
- **Tests :** par exemple $\mathbf{n} < \mathbf{2}$ de coût $k_t \in \mathbb{R}^{*+}$
- **Branchements :** dans le cadre d'une instruction conditionnelle ou d'une boucle, il est opéré un branchement vers l'alternative d'un **if** par exemple ou vers une sortie de boucle ; son coût est $k_b \in \mathbb{R}^{*+}$

On dit parfois opérations fondamentales à la place d'actions élémentaires

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1} :

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1} :

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1} :

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1} :

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o
- 1 test : coût k_t

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1} :

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o
- 1 test : coût k_t
- 1 branchement : coût k_b
  (1 branchement après **n** $<$ **2** ou après **x = 2015;**)

Principe 1 : Actions élémentaires en temps constant

Exemple : algorithme A_1 écrit en langage C

```
x = 2001; y = -25; m = 0;
if ( n < 2 ) x = 2015; else y = x;
x = a*n;
```

Analyser la complexité de A_1, c'est évaluer le temps de calcul T_{A_1} :

- 5 affectations : leur coût global est $5k_a$
  (le code en contient 6 affectations mais soit **x = 2015;** est exécutée, soit **y = x;**)
- 1 opération arithmétique : coût k_o
- 1 test : coût k_t
- 1 branchement : coût k_b
  (1 branchement après **n** $<$ **2** ou après **x = 2015;**)

Complexité de A_1 **:** $T_{A_1} = 5k_a + k_o + k_t + k_b$ soit un temps constant
(la somme de 8 constantes est une constante)

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

Attention : on verra plus loin que le modèle peut poser souci...

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

  Attention : on verra plus loin que le modèle peut poser souci...

- **L'analyse est totalement indépendante des machines utilisables** ramener le temps d'exécution d'une action élémentaire à une constante $k \in \mathbb{R}^{*+}$ permet de s'affranchir des questions de matériel

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

  Attention : on verra plus loin que le modèle peut poser souci...
- **L'analyse est totalement indépendante des machines utilisables** ramener le temps d'exécution d'une action élémentaire à une constante $k \in \mathbb{R}^{*+}$ permet de s'affranchir des questions de matériel
- k **sera précisé en pratique** selon que l'on utilise un "vieux" PC ou le supercalculateur Fugaku qui tourne à 418 péta FLOPS...

Principe 1 : Actions élémentaires en temps constant

Quelques remarques :

- **On simplifie le modèle** sans incidence sur la validité des évaluations
  - les branchements peuvent être ignorés
  - simplification en posant $k_a = k_o = k_t = k \in \mathbb{R}^{*+}$ (en fait $k_a, k_o, k_t \leq k$)
    Conséquence : l'analyse de la complexité d'un algorithme a pour objet de dénombrer les actions élémentaires exécutées par cet algorithme

  Attention : on verra plus loin que le modèle peut poser souci...
- **L'analyse est totalement indépendante des machines utilisables** ramener le temps d'exécution d'une action élémentaire à une constante $k \in \mathbb{R}^{*+}$ permet de s'affranchir des questions de matériel
- k **sera précisé en pratique** selon que l'on utilise un "vieux" PC ou le supercalculateur Fugaku qui tourne à 418 péta FLOPS...
- **Et l'algorithme est écrit en C car 76 fois moins énergivore que Python** ... merci pour la planète

Principe 2 : Analyse en fonction de la donnée en entrée.

L'analyse s'opère en fonction de la taille de la donnée en entrée

- **Évaluation du temps d'exécution :** en fonction de la taille de la donnée à traiter

  (Naturel : a priori, le temps de calcul augmente avec une donnée plus grande...)

Principe 2 : Analyse en fonction de la donnée en entrée.

L'analyse s'opère en fonction de la taille de la donnée en entrée

- **Évaluation du temps d'exécution :** en fonction de la taille de la donnée à traiter

  (Naturel : a priori, le temps de calcul augmente avec une donnée plus grande...)

- **Taille de la donnée en entrée ?**
  - test de primalité ($n \in \mathbb{N}$ premier ?) : taille du codage de l'entier
  - algorithme de tri : taille du tableau à trier
  - algorithme de graphe : selon la représentation listes ou matrices

Principe 2 : Analyse en fonction de la donnée en entrée.

L'analyse s'opère en fonction de la taille de la donnée en entrée

- **Évaluation du temps d'exécution :** en fonction de la taille de la donnée à traiter
  (Naturel : a priori, le temps de calcul augmente avec une donnée plus grande...)
- **Taille de la donnée en entrée ?**
  - test de primalité ($n \in \mathbb{N}$ premier ?) : taille du codage de l'entier
  - algorithme de tri : taille du tableau à trier
  - algorithme de graphe : selon la représentation listes ou matrices
- **Incidence formelle**
  - complexité d'un algorithme A : une fonction $T_A : \mathbb{N} \to \mathbb{R}^+$
  - avec une donnée de taille n, par exemple $T_A(n) = 7k.n^2 + 3kn + 24k$.

Principe 3 : Comportement asymptotique

L'analyse prend en compte le comportement asymptotique du temps

- **Un ordinateur est fait pour traiter des grands jeux de données**

Principe 3 : Comportement asymptotique

L'analyse prend en compte le comportement asymptotique du temps

- **Un ordinateur est fait pour traiter des grands jeux de données**
- **Comment évolue le temps quand la taille de la donnée s'accroît**

  Exploitation d'un programme sur de plus grands jeux de données que ceux utilisés pour sa mise au point

  (Rappel : le tri à bulles est 100 fois plus lent sur un tableau 10 fois plus grand...)

Principe 3 : Comportement asymptotique

L'analyse prend en compte le comportement asymptotique du temps

- **Un ordinateur est fait pour traiter des grands jeux de données**
- **Comment évolue le temps quand la taille de la donnée s'accroît**

  Exploitation d'un programme sur de plus grands jeux de données que ceux utilisés pour sa mise au point

  (Rappel : le tri à bulles est 100 fois plus lent sur un tableau 10 fois plus grand...)

- **Comportement asymptotique du temps**

  Étude de la fonction $T_A : \mathbb{N} \to \mathbb{R}^+$ quand n tend vers l'infini

  (même si aucun jeu de données ne sera de taille infinie en pratique... *a priori)*

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

En sortie **present** (variable de nature logique, vrai / faux) est affectée

- à vrai soit 1 si l'entier **x** est mémorisé dans le tableau **t** ;
- sinon à faux soit 0.

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$

Principe 3 : Comportement asymptotique

Exemple : A_2 vérifie si un entier **x** est présent dans un tableau **t**

```
typedef int TABLEAU[n]; /* n est supposee definie */
TABLEAU t;
int x,i, present;
...
present = 0;
for (i=0; i < n; i = i+1) if ( t[i] == x ) present = 1;
```

Analyse :

- initialisation de la variable **present** et de variable indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2kn$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$
- affectation de la variable **present** à 1 : peut coûter de 0 à $k.n$
  selon que **x** n'apparaît pas dans le tableau **t** ou figure dans chaque case

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement $(i = n)$
- test d'égalité **t[i] == x** : coûte $k.n$
- affectation de la variable **present** à 1 : peut coûter de 0 à $k.n$
  selon que **x** n'apparaît pas dans le tableau **t** ou figure dans chaque case

Principe 3 : Comportement asymptotique

Analyse : on fait la somme de tous les coûts

- initialisation de la variable **present** et de l'indice **i** : coûte $2k$
- incrémentation de l'indice **i** dans la boucle **for** : coûte $2k.n$
  car 1 affectation et 1 addition exécutées pour chacune des n incrémentations
- le test **i** $<$ **n** : coûte $k.n + k$
  car réalisé n fois positivement (de $i = 0$ à $n - 1$) et 1 fois négativement ($i = n$)
- test d'égalité **t[i] == x** : coûte $k.n$
- affectation de la variable **present** à 1 : peut coûter de 0 à $k.n$
  selon que **x** n'apparaît pas dans le tableau **t** ou figure dans chaque case

Complexité de A_2 :

$$2k + 2k.n + (k.n + k) + k.n = 3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$$

soit

$$3k + 4k.n \leq T_{A_2(n)} \leq 3k + 5k.n$$

(encadrement du fait de l'incertitude sur la présence de la valeur **x** dans **t**)

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 **:** $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2(n)}$ est **linéaire**

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2(n)}$ est **linéaire**
- $4k$ ou $5k$ sont juste des **constantes multiplicatives**
  car elles n'influent pas sur la forme de l'accroissement du temps
  (on dit parfois **constantes cachées** si on ne les évoque pas)

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2(n)}$ est **linéaire**
- $4k$ ou $5k$ sont juste des **constantes multiplicatives** car elles n'influent pas sur la forme de l'accroissement du temps (on dit parfois **constantes cachées** si on ne les évoque pas)
- Comportement linéraire de la fonction $T_{A_2}(n)$:
  - la complexité est en $O(n)$ pour parler de **majoration** du temps de calcul ce sera noté $T_{A_2(n)} \in O(n)$ (prononcer "*en grand o de n*")
  - la complexité est en $\Theta(n)$ pour parler **d'ordre exact** du temps de calcul ce sera noté $T_{A_2(n)} \in \Theta(n)$ (prononcer "*en thêta de n*")

Principe 3 : Comportement asymptotique

Remarques sur la complexité de A_2 : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$

- Expression imprécise mais tendance bien identifée
- Quand n est grand, $3k$ **devient négligeable** par rapport à $k.n$
- Que ce soit avec $4k.n$ ou $5k.n$, la croissance de $T_{A_2(n)}$ est **linéaire**
- $4k$ ou $5k$ sont juste des **constantes multiplicatives** car elles n'influent pas sur la forme de l'accroissement du temps (on dit parfois **constantes cachées** si on ne les évoque pas)
- Comportement linéraire de la fonction $T_{A_2}(n)$:
  - la complexité est en $\mathrm{O}(n)$ pour parler de **majoration** du temps de calcul ce sera noté $T_{A_2(n)} \in \mathrm{O}(n)$ (prononcer "*en grand o de n*")
  - la complexité est en $\Theta(n)$ pour parler **d'ordre exact** du temps de calcul ce sera noté $T_{A_2(n)} \in \Theta(n)$ (prononcer "*en thêta de n*")

Mais il y a 2 soucis :

- Encadrement du temps $\Rightarrow$ incertitude, imprécision
- Et qui écrirait un tel algorithme ?

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas : on considère un jeu de données qui maximise le temps de calcul de l'algorithme

- permet d'obtenir des garanties car ce temps ne sera jamais dépassé
- permet en général d'améliorer l'approche en évitant les pires cas

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas : on considère un jeu de données qui maximise le temps de calcul de l'algorithme

- permet d'obtenir des garanties car ce temps ne sera jamais dépassé
- permet en général d'améliorer l'approche en évitant les pires cas

Retour sur l'algorithme A_2 :

- on avait trouvé : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$
- on lève l'incertitude : $T_{A_2}(n) = 3k + 5k.n$

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas : on considère un jeu de données qui maximise le temps de calcul de l'algorithme

- permet d'obtenir des garanties car ce temps ne sera jamais dépassé
- permet en général d'améliorer l'approche en évitant les pires cas

Retour sur l'algorithme A_2 :

- on avait trouvé : $3k + 4k.n \leq T_{A_2}(n) \leq 3k + 5k.n$
- on lève l'incertitude : $T_{A_2}(n) = 3k + 5k.n$

Améliorer A_2 en évitant les pire cas ?

Idée : arrêter l'exécution dès que **x** est trouvé dans **t** !

On regarde cela, mais avant quelques remarques

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
  - mise en évidence d'un modèle probabiliste pour les jeux de données
    $\rightarrow$ pas toujours simple, et parfois impossible (cf. n'a aucun sens)
  - analyse souvent très complexe
  - pas pertinent dans cette UE...

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
  - mise en évidence d'un modèle probabiliste pour les jeux de données $\rightarrow$ pas toujours simple, et parfois impossible (cf. n'a aucun sens)
  - analyse souvent très complexe
  - pas pertinent dans cette UE...
- analyse **lisse** d'algorithme (*smoothed analysis*)
  - pour éviter certains soucis de l'analyse en moyenne
  - pas pertinent dans cette UE...

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
  - mise en évidence d'un modèle probabiliste pour les jeux de données $\rightarrow$ pas toujours simple, et parfois impossible (cf. n'a aucun sens)
  - analyse souvent très complexe
  - pas pertinent dans cette UE...
- analyse **lisse** d'algorithme (*smoothed analysis*)
  - pour éviter certains soucis de l'analyse en moyenne
  - pas pertinent dans cette UE...
- analyse de la complexié dans le **meilleur des cas**
  - pour les seuls optimiste ? Non, c'est parfois utile pour faciliter l'analyse
  - pas pertinent dans cette UE...

Principe 4 : Pire des cas (dans cette UE)

Analyse dans le pire des cas ? D'autres approches existent :

- analyse de la complexié en **moyenne**
  - mise en évidence d'un modèle probabiliste pour les jeux de données
    $\rightarrow$ pas toujours simple, et parfois impossible (cf. n'a aucun sens)
  - analyse souvent très complexe
  - pas pertinent dans cette UE...
- analyse **lisse** d'algorithme (*smoothed analysis*)
  - pour éviter certains soucis de l'analyse en moyenne
  - pas pertinent dans cette UE...
- analyse de la complexié dans le **meilleur des cas**
  - pour les seuls optimiste ? Non, c'est parfois utile pour faciliter l'analyse
  - pas pertinent dans cette UE...

Dans cette UE : focalisation sur l'analyse dans le pire des cas

Principe 4 : Pire des cas (dans cette UE)

Exemple : A_3 résout (plus efficacement ?) le même problème que A_2

```
present = 0; i = 0;
while( !present && (i < n) )
    if ( t[i] == x ) present = 1; else i = i+1;
```

Principe 4 : Pire des cas (dans cette UE)

Exemple : A_3 résout (plus efficacement ?) le même problème que A_2

```
present = 0; i = 0;
while( !present && (i < n) )
      if ( t[i] == x ) present = 1; else i = i+1;
```

Analyse : conduit à trouver $T_{A_3}(n) = 6k + 7k.n$

- identification du pire des cas : donnée maximisant le temps de calcul
  - meilleur des cas : $\mathbf{x = t[0]}$ car arrêt immédiat $\Rightarrow$ coûte $10k$
  - pire des cas : **x** n'est pas dans **t** $\Rightarrow$ tout le tableau est parcouru

Principe 4 : Pire des cas (dans cette UE)

Exemple : A_3 résout (plus efficacement ?) le même problème que A_2

```
present = 0; i = 0;
while( !present && (i < n) )
      if ( t[i] == x ) present = 1; else i = i+1;
```

Analyse : conduit à trouver $T_{A_3}(n) = 6k + 7k.n$

- identification du pire des cas : donnée maximisant le temps de calcul
  - meilleur des cas : **x = t[0]** car arrêt immédiat $\Rightarrow$ coûte $10k$
  - pire des cas : **x** n'est pas dans **t** $\Rightarrow$ tout le tableau est parcouru
- analyse dans le pire des cas (avec n passages dans la boucle)
  - n tests d'entée positifs et 1 test négatif (chacun coûte $4k$) (négation **!** + test **!present** + test **i<n** + conjonction **&&**)
  - n tests d'égalité **t[i] == x** négatifs (chacun coûte k)
  - n incrémentations de **i** (chacune coûte $2k$)

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

$$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

et c'est même pire !!!

(même si "en général" A_3 sera plus efficace que A_2)

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

$$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

  (même si "en général" A_3 sera plus efficace que A_2)

- analyse dans le pire des cas : offre une garantie :
  on ne fera jamais pire

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

  $$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

  (même si "en général" A_3 sera plus efficace que A_2)

- analyse dans le pire des cas : offre une garantie :
  on ne fera jamais pire
- cette analyse permet de lever les doutes précédents

Principe 4 : Pire des cas (dans cette UE)

Quelques remarques:

- A_3 pas plus efficace que A_2 dans le pire des cas :

  $$T_{A_2}(n) = 3k + 5kn \text{ VS } T_{A_3}(n) = 6k + 7k.n$$

  et c'est même pire !!!

  (même si "en général" A_3 sera plus efficace que A_2)

- analyse dans le pire des cas : offre une garantie :
  on ne fera jamais pire
- cette analyse permet de lever les doutes précédents

Dans cette UE, on ne s'intéressera qu'au pire des cas car :

- pour des problème difficiles on peut avoir des cas triviaux
  (si l'aiguille cherchée dans la botte de foin brille à l'entrée du tas...)
- un problème sera jugé difficile même s'il recèle "peu" de cas difficile

Plan

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Pire de cas ?

En fait, il n'y a ici qu'un seul cas !

On dit alors parfois : « dans tous les cas »

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

coûte $2k$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

coûte $2k$

coût : $k(n+1)$

(contrôle de la boucle)

n tests positifs : i=0,... i=n-1

1 test négatif : i=n

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

coûte $2k$

coût : $k(n+1)$

bloc exécuté n fois

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

coûte $2k$

coût : $k(n+1)$

bloc exécuté n fois

on évalue le coût d'une exécution de ce bloc

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

analyse d'une exécution de ce bloc

1 exécution du bloc ?

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

analyse d'une exécution de ce bloc

1 exécution du bloc :

- `j = 0;` coûte k
- `while (j < n)` coûte $k + k.n$
- `x = x + 1;` coûte $2k.n$
- `j = j + 1;` coûte $2k.n$
- `i = i + 1;` coûte $2k$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

analyse d'une exécution de ce bloc

1 exécution du bloc : $4k + 5k.n$

`j = 0;` coûte k

`while (j < n)` coûte $k + k.n$

`x = x + 1;` coûte $2k.n$

`j = j + 1;` coûte $2k.n$

`i = i + 1;` coûte $2k$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < n)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Coût global ?

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Coût global ?

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Algorithme A_4 : avec des boucles imbriquées

Coût global ?

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

`x = 0; i = 0;` : coûte $2k$

`while (i < n)` : coûte $k + k.n$

bloc exécuté n fois

1 exécution du bloc : $4k + 5k.n$
donc pour n exécutions :
$n \times (4k + 5k.n)$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Coût global ?

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

1 exécution du bloc : $4k + 5k.n$
donc pour n exécutions :
n x $(4k + 5k.n)$

Coût global :

$$2k + k + k.n + n \times (4k + 5k.n)$$
$$=$$
$$5k.n^2 + 5k.n + 3k$$

Algorithme A_4 : avec des boucles imbriquées

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < n)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Coût global :

$$T_{A_4}(n) = k(5.n^2 + 5n + 3)$$

Complexité :

A_4 est donc en $O(n^2)$
car $5k.n$ et $3k$ sont négligeables
par rapport à $5k.n^2$
(O : majoration)

et pour l'ordre exact

A_4 est donc en $\Theta(n^2)$

Algorithme A_5 : une variation sur A_4

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Juste une petite modification :

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Juste une petite modification :
le test
(j < n)
est remplacé par
(j < i)

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Juste une petite modification :
le test
(j < n)
est remplacé par
(j < i)

Incidence sur la complexité ?

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse :

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse :

Pire de cas :
un seul cas à considérer

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Analyse :

Pire de cas :
un seul cas à considérer

Et on peut déjà affirmer que
A_5 est aussi en $O(n^2)$
car moins d'actions exécutées
(à cause de j<i VS j<n)
mais ce n'est qu'une majoration

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
    j = 0;
    while (j < i)
    {
        x = x + 1;
        j = j + 1;
    }
    i = i + 1;
}
```

Analyse :

Pire de cas :
un seul cas à considérer

Et on peut déjà affirmer que
A_5 est aussi en $O(n^2)$
car moins d'actions exécutées
(à cause de j<i VS j<n)
mais ce n'est qu'une majoration

Quid de l'ordre exact ?
(cf. notation théta)

Algorithme A_5 : une variation sur A_4

Analyse à l'ordre exact

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

$x = 0;\ i = 0;$ coûte $2k$

while (i < n) coûte $k + k.n$

bloc exécuté n fois

Algorithme A_5 : une variation sur A_4

Analyse à l'ordre exact

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

`x = 0; i = 0;` ← coûte $2k$

`while (i < n)` ← coûte $k + k.n$

bloc ← bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

`x = 0; i = 0;` coûte $2k$

`while (i < n)` coûte $k + k.n$

bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

- si i=0 le coût est $4k$ (temps constant)

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

- si i=0 le coût est $4k$ (temps constant)
- si i=n-1 le coût est $4k + 5k.n$ (temps linéaire en n)

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Mais chaque exécution du bloc n'a pas le même coût :

- si i=0 le coût est $4k$ (temps constant)
- si i=n-1 le coût est $4k + 5k.n$ (temps linéaire en n)

L'analyse est plus subtile !

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

`x = 0; i = 0;` ← coûte $2k$

`while (i < n)` ← coûte $k + k.n$

bloc ← bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$
- si i=2 le coût est $4k + 2.5.k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$
- si i=2 le coût est $4k + 2.5.k$

et pour i avec $0 \leq i \leq n-1$

le coût est $4k + i.5k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

bloc exécuté n fois

Pour i=0, i=1, i=2,... et i=n-1

- si i=0 le coût est $4k$
- si i=1 le coût est $4k + 5k$
- si i=2 le coût est $4k + 2.5.k$

et pour i avec $0 \leq i \leq n\text{-}1$

le coût est $4k + i.5k$

Il faut donc additionner !

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

coût global du bloc

somme des $4k$ + i.$5k$
pour i allant de 0 à n-1

$$\sum_{i=0}^{i=n-1} (4k + i.5k)$$

qui est égale à

$$4k.n + \frac{5}{2} k.n(n\text{-}1)$$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

coûte $2k$

coûte $k + k.n$

coût global du bloc

$$4k.n + \frac{5}{2}k.n(n-1)$$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

- `x = 0; i = 0;` ← coûte $2k$
- `while (i < n)` ← coûte $k + k.n$
- bloc ← coût global du bloc $4k.n + \frac{5}{2}k.n(n\text{-}1)$

Le coût total est donc

$$2k + k + k.n + 4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
      j = 0;
      while (j < i)
      {
            x = x + 1;
            j = j + 1;
      }
      i = i + 1;
}
```

Analyse à l'ordre exact

`x = 0; i = 0;` ← coûte $2k$

`while (i < n)` ← coûte $k + k.n$

bloc ← coût global du bloc

$$4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Le coût total est donc

$$2k + k + k.n + 4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Soit $\frac{5}{2}k.n^2 + \frac{5}{2}k.n + 3k$

Algorithme A_5 : une variation sur A_4

```
int i,j,x;
x = 0; i = 0;
while (i < n)
{
        j = 0;
        while (j < i)
        {
                x = x + 1;
                j = j + 1;
        }
        i = i + 1;
}
```

Analyse à l'ordre exact

$\leftarrow$ coûte $2k$

$\leftarrow$ coûte $k + k.n$

$\leftarrow$ coût global du bloc

$$4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Le coût total est donc

$$2k + k + k.n + 4k.n + \frac{5}{2}k.n(n\text{-}1)$$

Soit $\frac{5}{2}k.n^2 + \frac{5}{2}k.n + 3k$

A_5 **est donc en** $\Theta(n^2)$

Plan

Majoration : notation "Grand O"

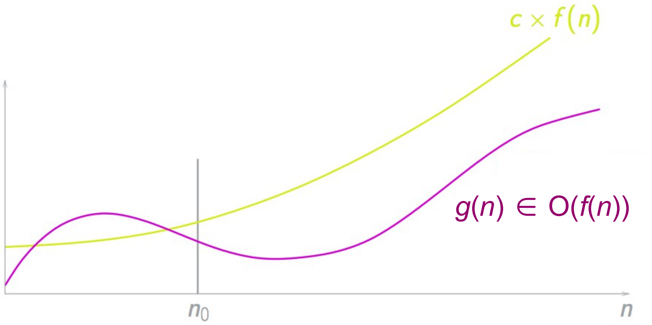
La complexité est toujours évaluée en ordre de grandeur asymptotique :

Majoration : notation "Grand O"

La complexité est toujours évaluée en ordre de grandeur asymptotique :

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \to \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$



(merci à Stéphane Grandcolas pour le tracé des courbes)

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$

Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$

Pourquoi peut-on écrire que $g(n) = 100n + 400 \in \mathrm{O}(n)$ (avec $f(n) = n$) ?

- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)
- parce que avec $c = 300$ et $n_0 = 2$, $\forall n \geq n_0$, on a $g(n) \leq c.f(n)$

  en effet $g(1) = 500$ et $c.f(1) = 300$,

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \to \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$

Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)
- parce que avec $c = 300$ et $n_0 = 2$, $\forall n \geq n_0$, on a $g(n) \leq c.f(n)$

  en effet $g(1) = 500$ et $c.f(1) = 300$, puis $g(2) = 600 = c.f(2)$,

Majoration : notation "Grand O"

Majoration (notation "Grand O", notation de Landau) : Etant donnée une fonction $f : \mathbb{N} \rightarrow \mathbb{R}^*$, l'ensemble des fonctions bornées supérieurement par un multiple réel de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$O(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \leq c.f(n)\}$$

Pourquoi peut-on écrire que $g(n) = 100n + 400 \in O(n)$ (avec $f(n) = n$) ?

- i.e. $g(n) = 100n + 400$ dominée asymptotiquement par $f(n)) = n$ (à un facteur multiplicatif réel près)
- parce que avec $c = 300$ et $n_0 = 2$, $\forall n \geq n_0$, on a $g(n) \leq c.f(n)$

  en effet $g(1) = 500$ et $c.f(1) = 300$, puis $g(2) = 600 = c.f(2)$,

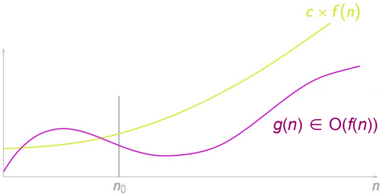
  puis $g(3) = 700$ et $c.f(3) = 900$, puis $g(4) = 800$ et $c.f(4) = 1200$, etc.

Majoration : notation "Grand O"

Commentaires :

- La fonction $g(n)$ exprime ici un temps d'exécution $T_A(n)$
- Permet de caractériser facilement des complexités :
  - temps constant : en $O(1)$
  - temps logarithmique : en $O(\log(n))$
  - temps linéaire : en $O(n)$
  - temps linéarithmique : en $O(n.\log(n))$
  - temps quadratique : en $O(n^2)$
  - temps cubique : en $O(n^3)$
  - temps exponentiel : en $O(2^n)$
  - etc.

Majoration : notation "Grand O"

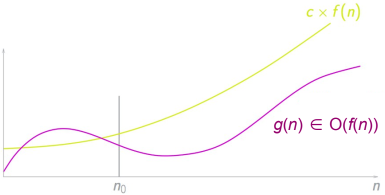


Commentaires :

- Permet de se concentrer sur l'essentiel : la nature du comportement
  - supprime les termes négligeables : n^2 est "oublié" face à n^3
  - élimine les facteurs multiplicatifs constants : $437/5.n^3$ s'écrira n^3

en effet : $g(n) = 2542.n^3 + 10^5.n^2 + 47/9.n \in O(n^3)$

Majoration : notation "Grand O"



Commentaires :

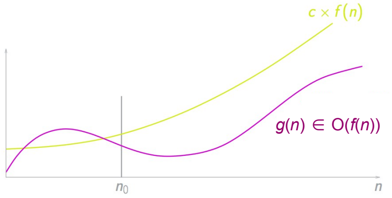
- Permet de se concentrer sur l'essentiel : la nature du comportement
  - supprime les termes négligeables : n^2 est "oublié" face à n^3
  - élimine les facteurs multiplicatifs constants : $437/5.n^3$ s'écrira n^3

  en effet : $g(n) = 2542.n^3 + 10^5.n^2 + 47/9.n \in O(n^3)$

- Parfois, il est écrit "$g(n) = n^2 + n = O(n^2)$"
  - mais selon D. Knuth (1976), O exprime des ensembles de fonctions...
  - il semble donc souhaitable d'écrire "$g(n) \in O(n^2)$"

  (on dit souvent "$g(n)$ **est en** $O(n^2)$" par traduction de "**is in**")

Majoration : notation "Grand O"



Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct !
  toute fonction linéaire est majorée asymptotiquement par une fonction quadratique

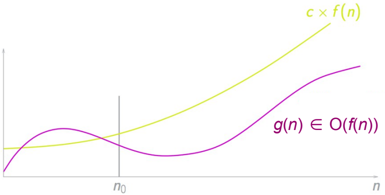
Majoration : notation "Grand O"

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct !
  toute fonction linéaire est majorée asymptotiquement par une fonction quadratique
- Utilisation de la notation "O" : c'est imprécis
  - l'ordre exact n'est pas exprimé
  - une forme d'aveu d'impuissance, d'incompétence

  O n'est à n'utiliser que si l'ordre exact est inconnu ou quand c'est suffisant
  (comme dans ce cours en général...)

Majoration : notation "Grand O"



Commentaires :

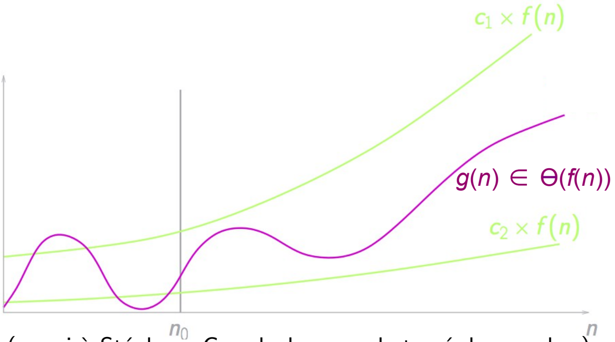
- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct !
  toute fonction linéaire est majorée asymptotiquement par une fonction quadratique
- Utilisation de la notation "O" : c'est imprécis
  - l'ordre exact n'est pas exprimé
  - une forme d'aveu d'impuissance, d'incompétence

  O n'est à n'utiliser que si l'ordre exact est inconnu ou quand c'est suffisant
  (comme dans ce cours en général...)
- À ne pas confondre avec la notation "petit o" : $g(n) \in o(f(n))$
  (si g est négligeable devant f asymptotiquement : $\forall c \in \mathbb{R}^+, \ldots g(n) \leq c.f(n)$)

Ordre exact : notation "Théta"

Ordre exact (notation "Θ", dire "Théta"): Etant donnée une fonction $f : \mathbb{N} \rightarrow R^*$, l'ensemble des fonctions bornées supérieurement et inférieurement par des multiples réels de $f(n)$, à partir d'un certain seuil n_0 est défini par :

$$\Theta(f(n)) = \{g : \mathbb{N} \rightarrow \mathbb{R}^* : \exists c_1, c_2 \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, c_2.f(n) \leq g(n) \leq c_1.f(n)\}$$



(merci à Stéphane Grandcolas pour le tracé des courbes)

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)
- Majoration ("Grand O") :
  - quand c'est suffisant (comme dans ce cours en général))
  - ou quand on ne sait pas faire mieux

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)
- Majoration ("Grand O") :
  - quand c'est suffisant (comme dans ce cours en général))
  - ou quand on ne sait pas faire mieux
- La minoration existe : "Grand Omega" avec la notation "Ω" :
  - $\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \geq c.f(n)\}$

Récapitulation

Commentaires :

- Écrire "$g(n) = 5.n + 3 \in O(n^2)$" est mathématiquement correct mais écrire "$g(n) = 5.n + 3 \in \Theta(n^2)$" **est faux** car $g(n) \in \Theta(n)$
- Ordre exact ("Θ") : idéal mais pas toujours nécessaire (cf. ce cours)
- Majoration ("Grand O") :
  - quand c'est suffisant (comme dans ce cours en général))
  - ou quand on ne sait pas faire mieux
- La minoration existe : "Grand Omega" avec la notation "Ω" :
  - $\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{R}^* : \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, g(n) \geq c.f(n)\}$
  - peut être utile avec la propriété : $\Theta((f(n)) = O(f(n)) \cap \Omega(f(n))$ si exprimer $\Theta()$ n'est pas immédiat et que O et Ω sont évidents

Plan

Incidence du codage

Complexité d'un algorithme : toujours fonction de la taille des entrées

$\Rightarrow$ il faut alors considérer des "**codages raisonnables**" :

Incidence du codage

Complexité d'un algorithme : toujours fonction de la taille des entrées

$\Rightarrow$ il faut alors considérer des "**codages raisonnables**" :

Exemple du traitement d'un entier n :

- Plusieurs codages sont *a priori* possibles dont :
  - codage unaire : n batonnets (souvenirs de l'école maternelle...)
  - codage binaire : $\lfloor log_2(n) \rfloor + 1$ bits

Incidence du codage

Complexité d'un algorithme : toujours fonction de la taille des entrées

$\Rightarrow$ il faut alors considérer des "**codages raisonnables**" :

Exemple du traitement d'un entier n :

- Plusieurs codages sont *a priori* possibles dont :
  - codage unaire : n batonnets (souvenirs de l'école maternelle...)
  - codage binaire : $\lfloor log_2(n) \rfloor + 1$ bits
- Le choix du codage a alors une incidence considérable !

  Un algorithme peut "devenir" polynomial avec un "mauvais codage".

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{\text{oui,non}\}$
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) **faire**
  si (i divise n) **alors** $prem \leftarrow$ non
  sinon $i \leftarrow i + 1$
fin tantque
Retourner $prem$.

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{\text{oui,non}\}$
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) **faire**
  si (i divise n) **alors** $prem \leftarrow$ non
  sinon $i \leftarrow i + 1$
fin tantque
Retourner $prem$.

Complexité :

- au plus $\sqrt{n} - 1$ passages dans la boucle **tantque**

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{\text{oui,non}\}$
  $prem \leftarrow$ oui
  $i \leftarrow 2$
  tantque $(prem \text{ AND } (i \leq \sqrt{n}))$ **faire**
    si (i divise n) **alors** $prem \leftarrow$ non
    sinon $i \leftarrow i + 1$
  fin tantque
  Retourner $prem$.

Complexité :

- au plus $\sqrt{n} - 1$ passages dans la boucle **tantque**
- supposition : traitement local à la boucle réalisable en temps constant

Incidence du codage

Un problème très connu en théorie de la complexité :

PREMIER (test de primalité) : $n \in \mathbb{N}^*$ est-il un nombre premier ?

Un algorithme très simple pour le résoudre :

Entrèes: $n \in \mathbb{N}^*$
Sorties: $prem \in \{\text{oui,non}\}$
$prem \leftarrow$ oui
$i \leftarrow 2$
tantque ($prem$ AND ($i \leq \sqrt{n}$)) **faire**
si (i divise n) **alors** $prem \leftarrow$ non
sinon $i \leftarrow i + 1$
fin tantque
Retourner $prem$.

Complexité :

- au plus $\sqrt{n} - 1$ passages dans la boucle **tantque**
- supposition : traitement local à la boucle réalisable en temps constant

La complexité est donc $\Theta(\sqrt{n}) = \Theta(n^{1/2})$, soit mieux que linéaire ?

Incidence du codage

Cet algorithme très simple

$prem \leftarrow$ oui
$i \leftarrow 2$
tantque $(prem$ AND $(i \leq \sqrt{n}))$ **faire**
  si (i divise n) **alors** $prem \leftarrow$ non
  sinon $i \leftarrow i + 1$
fin tantque

Incidence du codage

Cet algorithme très simple

$$
\begin{array}{l}
prem \leftarrow \text{oui} \\
i \leftarrow 2 \\
\textbf{tantque } (prem \text{ AND } (i \leq \sqrt{n})) \textbf{ faire} \\
\quad \textbf{si } (\ i \text{ divise } n\ ) \textbf{ alors } prem \leftarrow \text{non} \\
\quad \textbf{sinon } i \leftarrow i + 1 \\
\textbf{fin tantque}
\end{array}
$$

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

Incidence du codage

Cet algorithme très simple

$$
\begin{array}{l}
prem \leftarrow \text{oui} \\
i \leftarrow 2 \\
\textbf{tantque } (prem \text{ AND } (i \leq \sqrt{n})) \textbf{ faire} \\
\quad \textbf{si } (\ i \text{ divise } n\ ) \textbf{ alors } prem \leftarrow \text{non} \\
\quad \textbf{sinon } i \leftarrow i + 1 \\
\textbf{fin tantque}
\end{array}
$$

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire

Incidence du codage

Cet algorithme très simple

$prem \leftarrow$ oui
$i \leftarrow 2$
tantque $(prem$ AND $(i \leq \sqrt{n}))$ **faire**
  si (i divise n) **alors** $prem \leftarrow$ non
  sinon $i \leftarrow i + 1$
fin tantque

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire
- (sous-)**exponentielle pour le codage binaire !**

Incidence du codage

Cet algorithme très simple

$prem \leftarrow$ oui
$i \leftarrow 2$
tantque $(prem$ AND $(i \leq \sqrt{n}))$ **faire**
  si (i divise n) **alors** $prem \leftarrow$ non
  sinon $i \leftarrow i+1$
fin tantque

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire
- (sous-)**exponentielle pour le codage binaire !**
  - taille de la donnée : $Taille(n) = \lfloor log_2(n) \rfloor + 1$
  - d'où $n \in \Theta(2^{Taille(n)-1})$
  - complexité de l'algorithme : $\Theta(2^{[Taille(n)-1]/2})$
    qui est (sous-)exponentielle dans la taille de la donnée

Incidence du codage

Cet algorithme très simple

$$\begin{array}{l} prem \leftarrow \text{oui} \\ i \leftarrow 2 \\ \textbf{tantque } (prem \text{ AND } (i \leq \sqrt{n})) \textbf{ faire} \\ \quad \textbf{si } (\ i \text{ divise } n\ ) \textbf{ alors } prem \leftarrow \text{non} \\ \quad \textbf{sinon } i \leftarrow i+1 \\ \textbf{fin tantque} \end{array}$$

a une complexité $\Theta(\sqrt{n}) = \Theta(n^{1/2})$ qui est :

- "racinaire" : mieux que **polynomial** pour le codage unaire
- (sous-)**exponentielle pour le codage binaire !**
  - taille de la donnée : $Taille(n) = \lfloor log_2(n) \rfloor + 1$
  - d'où $n \in \Theta(2^{Taille(n)-1})$
  - complexité de l'algorithme : $\Theta(2^{[Taille(n)-1]/2})$

    qui est (sous-)exponentielle dans la taille de la donnée

On travaillera toujours avec des **codages raisonnables**

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Dans le cadre de ce cours, il le sera mais...

certaines hypothèses sont **fondamentalement fausses :**

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Dans le cadre de ce cours, il le sera mais...

certaines hypothèses sont **fondamentalement fausses :**

- coût d'une affectation d'entier n : pas du temps constant !
  les bits de l'entier sont tous recopiés $\Rightarrow$ en $\Theta(\mathit{Taille}(n)) = \Theta((\lfloor log_2(n) \rfloor + 1)$

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Dans le cadre de ce cours, il le sera mais...

certaines hypothèses sont **fondamentalement fausses :**

- coût d'une affectation d'entier n : pas du temps constant !
  les bits de l'entier sont tous recopiés $\Rightarrow$ en $\Theta(Taille(n)) = \Theta((\lfloor log_2(n) \rfloor + 1)$
- test de la forme $x < y$: pas du temps constant !
  linéaire en la taille du codage du plus petit entier : en $\Theta((\lfloor log_2(min(x, y)) \rfloor + 1)$

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Dans le cadre de ce cours, il le sera mais...

certaines hypothèses sont **fondamentalement fausses :**

- coût d'une affectation d'entier n : pas du temps constant !
  les bits de l'entier sont tous recopiés $\Rightarrow$ en $\Theta(Taille(n)) = \Theta((\lfloor log_2(n) \rfloor + 1)$
- test de la forme $x < y$: pas du temps constant !
  linéaire en la taille du codage du plus petit entier : en $\Theta((\lfloor log_2(min(x, y)) \rfloor + 1)$
- addition : pas du temps constant
  pensez à l'algorithme appris à l'école : $x + y$ en $O(log_2(x) + log_2(y))$

Remise en cause du modèle

Le modèle d'analyse de la complexité : est-il correct ?

Dans le cadre de ce cours, il le sera mais...

certaines hypothèses sont **fondamentalement fausses :**

- coût d'une affectation d'entier n : pas du temps constant !
  les bits de l'entier sont tous recopiés $\Rightarrow$ en $\Theta(\mathit{Taille}(n)) = \Theta((\lfloor log_2(n) \rfloor + 1)$
- test de la forme $x < y$: pas du temps constant !
  linéaire en la taille du codage du plus petit entier : en $\Theta((\lfloor log_2(min(x, y)) \rfloor + 1)$
- addition : pas du temps constant
  pensez à l'algorithme appris à l'école : $x + y$ en $O(log_2(x) + log_2(y))$
- multiplication pire que l'addition : $x \times y$ en $O(log_2(x) \times log_2(y))$
- etc.

Précaution pour le modèle

Conséquences dans le cadre de ce cours :

Précaution pour le modèle

Conséquences dans le cadre de ce cours :

- il faudra s'assurer que les hypothèses ne posent pas de problème, i.e. qu'une erreur ne ferait pas passer de l'exponentiel à du polynomial

Précaution pour le modèle

Conséquences dans le cadre de ce cours :

- il faudra s'assurer que les hypothèses ne posent pas de problème, i.e. qu'une erreur ne ferait pas passer de l'exponentiel à du polynomial
- introduction d'un modele plus précis pour être plus rigoureux : il sera introduit au Chapitre 4 ("Cadre formel")

Précaution pour le modèle

Conséquences dans le cadre de ce cours :

- il faudra s'assurer que les hypothèses ne posent pas de problème, i.e. qu'une erreur ne ferait pas passer de l'exponentiel à du polynomial
- introduction d'un modele plus précis pour être plus rigoureux : il sera introduit au Chapitre 4 ("Cadre formel")
- travail avec des codages de données dits "raisonnables" :
  - entiers codés en binaire, ternaire, décimal, etc. mais jamais en unaire
  - codage unaire : un codage "déraisonnables"

  cela sera aussi introduit au Chapitre 4 ("Cadre formel")

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Mise en relation complexité et efficacité pratique

Objectif : traduire la complexité théorique en temps d'exécution machine

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

Mise en relation complexité et efficacité pratique

Objectif : traduire la complexité théorique en temps d'exécution machine

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$
$n = 10^2$	10^{-9} sec
$n = 10^3$	10^{-9} sec
$n = 10^4$	10^{-9} sec
$n = 10^6$	10^{-9} sec

Pour $\Theta(1)$, temps constant : pour toute valeur de n le temps est 10^{-9} seconde

Mise en relation complexité et efficacité pratique

Objectif : traduire la complexité théorique en temps d'exécution machine

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec
$n = 10^3$	10^{-9} sec	10^{-8} sec
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec

Pour $\Theta(log_2(n))$ (recherche dichotomique) : faible accroissement du temps machine

Mise en relation complexité et efficacité pratique

Objectif : traduire la complexité théorique en temps d'exécution machine

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec
$n = 10^4$	10^{-9} sec	$1{,}4 \times 10^{-8}$ sec	10^{-5} sec
$n = 10^6$	10^{-9} sec	2×10^{-8} sec	10^{-3} sec

Pour $\Theta(n)$ (recherche séquentielle) : accroissement linéaire du temps machine

Mise en relation complexité et efficacité pratique

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n(log_2(n)))$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec	$0,7 \times 10^{-6}$ sec
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec	10^{-5} sec
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec	10^{-5} sec	0,14 m sec
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec	10^{-3} sec	0,02 sec

Pour $\Theta(n.log_2(n))$ (tris par fusion) : accroissement "raisonnable" du temps machine

mais 0,02 seconde est presque perceptible pour un être humain

(ça décide d'une médaille d'Or en finale du 100 m aux Jeux Olympiques...)

Mise en relation complexité et efficacité pratique

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n(log_2(n)))$	$\Theta(n^2)$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec	$0,7 \times 10^{-6}$ sec	10^{-5} sec
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec	10^{-5} sec	10^{-3} sec
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec	10^{-5} sec	0,14 m sec	0,1 sec
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec	10^{-3} sec	0,02 sec	16 mn 40 sec

Pour $\Theta(n^2)$ (tri à bulles) : accroissement quadratique du temps machine

ne tient pas la comparaison avec le tri par fusion pour $n = 10^6$: 0,02 s VS 16 mn 40 s

(de l'intérêt d'investir du temps dans la recherche d'algorithmes performants)

Mise en relation complexité et efficacité pratique

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n(log_2(n)))$	$\Theta(n^2)$	$\Theta(n^3)$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec	$0,7 \times 10^{-6}$ sec	10^{-5} sec	10^{-3} sec
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec	10^{-5} sec	10^{-3} sec	1 sec
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec	10^{-5} sec	0,14 m sec	0,1 sec	16 mn 40 sec
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec	10^{-3} sec	0,02 sec	16 mn 40 sec	31 ans 8 mois

Pour $\Theta(n^3)$ (produit de matrices carrées $n \times n$: méthode de base)
des progrès ont été accomplis avec Strassen en 1969 et un algorithme en $\Theta(n^{log_2(7)})$
sachant que $log_2(7) \approx 2,8$ puis depuis... on en est à $n^{2,3728639}$ en 2014...

Attention : la taille de la donnée n'est pas n mais $n \times n$

Mise en relation complexité et efficacité pratique

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n(log_2(n)))$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec	$0,7 \times 10^{-6}$ sec	10^{-5} sec	10^{-3} sec	
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec	10^{-5} sec	10^{-3} sec	1 sec	
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec	10^{-5} sec	0,14 m sec	0,1 sec	16 mn 40 sec	
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec	10^{-3} sec	0,02 sec	16 mn 40 sec	31 ans 8 mois	

Pour $\Theta(2^n)$ (algorithme naïf pour SAT) : accroissement exponentiel du temps machine

Mise en relation complexité et efficacité pratique

Hypothèses :

- machine exécutant 10^9 actions élémentaires par seconde (10^{-9} seconde par action)
- n est la taille de la donnée en entrée (ou un paramètre de cette taille)
- aucune constante mutiplicative cachée : $\Theta(f(n))$ veut dire ici $f(n)$ actions

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n(log_2(n)))$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec	$0,7 \times 10^{-6}$ sec	10^{-5} sec	10^{-3} sec	
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec	10^{-5} sec	10^{-3} sec	1 sec	
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec	10^{-5} sec	0,14 m sec	0,1 sec	16 mn 40 sec	
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec	10^{-3} sec	0,02 sec	16 mn 40 sec	31 ans 8 mois	

Pour $\Theta(2^n)$ (algorithme naïf pour SAT) : accroissement exponentiel du temps machine

Quel est le temps de calcul ?

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

	$\Theta(1)$	$\Theta(log_2(n))$	$\Theta(n)$	$\Theta(n(log_2(n)))$	$\Theta(n^2)$	$\Theta(n^3)$	$\Theta(2^n)$
$n = 10^2$	10^{-9} sec	7×10^{-9} sec	10^{-7} sec	$0,7 \times 10^{-6}$ sec	10^{-5} sec	10^{-3} sec	4×10^{13} ans
$n = 10^3$	10^{-9} sec	10^{-8} sec	10^{-6} sec	10^{-5} sec	10^{-3} sec	1 sec	
$n = 10^4$	10^{-9} sec	1,4 $\times 10^{-8}$ sec	10^{-5} sec	0,14 m sec	0,1 sec	16 mn 40 sec	
$n = 10^6$	10^{-9} sec	2 $\times 10^{-8}$ sec	10^{-3} sec	0,02 sec	16 mn 40 sec	31 ans 8 mois	

2^n **correspond 40 000 milliards d'années !**

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

2^n **correspond 40 000 milliards d'années !**

Comment éviter cette explosion combinatoire
due à l'accroissement exponentiel du temps de calcul ?

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

2^n **correspond 40 000 milliards d'années !**

Comment éviter cette explosion combinatoire
due à l'accroissement exponentiel du temps de calcul ?

Utiliser une plus grosse machine qu'un PC ?

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

2^n **correspond 40 000 milliards d'années !**

Comment éviter cette explosion combinatoire
due à l'accroissement exponentiel du temps de calcul ?

Utiliser une plus grosse machine qu'un PC ?

Machine actuelle la plus rapide

Supercalculateur Fugaku (Fujitsu/ARM - Japon - 2020)

7 300 000 processeurs

vitesse = environ 418 péta FLOPS

(soit 418 × 1000 000 000 000 000 opérations à virgule flottante par seconde)

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

2^n **correspond 40 000 milliards d'années !**

Comment éviter cette explosion combinatoire
due à l'accroissement exponentiel du temps de calcul ?

Utiliser une plus grosse machine qu'un PC ?

Machine actuelle la plus rapide

Supercalculateur Fugaku (Fujitsu/ARM - Japon - 2020)

7 300 000 processeurs

vitesse = environ 418 péta FLOPS

(soit 418 $\times$ 1000 000 000 000 000 opérations à virgule flottante par seconde)

Pour $n = 100$ **: environ 100 000 ans de temps de calcul !**

Mise en relation complexité et efficacité pratique

Hypothèses : avec une machine exécutant 10^9 actions par seconde

2^n **correspond 40 000 milliards d'années !**

Comment éviter cette explosion combinatoire
due à l'accroissement exponentiel du temps de calcul ?

Utiliser une plus grosse machine qu'un PC ?

Machine actuelle la plus rapide

Supercalculateur Fugaku (Fujitsu/ARM - Japon - 2020)

7 300 000 processeurs

vitesse = environ 418 péta FLOPS

(soit 418 $\times$ 1000 000 000 000 000 opérations à virgule flottante par seconde)

Pour $n = 100$: environ 100 000 ans de temps de calcul !

Et dans cette UE :

étude des problèmes conjecturés de complexité exponentielle !

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Graphes et algorithme : Introduction

Dans quantité de domaines

- en informatique,
- en ingénierie,
- en sciences sociales,
- en intelligence artificielle,
- en chimie,
- etc.

de nombreux problèmes courants peuvent se représenter en termes de relations (binaires) entre objets

Graphes et algorithme : Introduction

Dans quantité de domaines

- en informatique,
- en ingénierie,
- en sciences sociales,
- en intelligence artificielle,
- en chimie,
- etc.

de nombreux problèmes courants peuvent se représenter en termes de

relations (binaires) entre objets

et donc de **graphes**.

Graphes et algorithme : Introduction

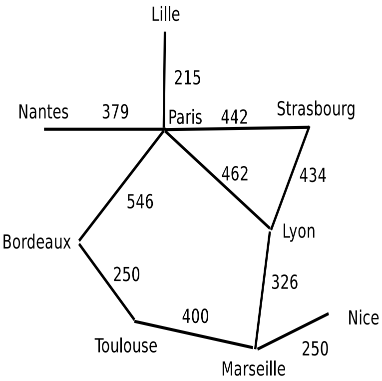
Des exemples :

- **Réseaux de communications ou de transports** : routiers, ferroviaires, machines en réseaux, circuits intégrés, électricité, eau...

Graphes et algorithme : Introduction

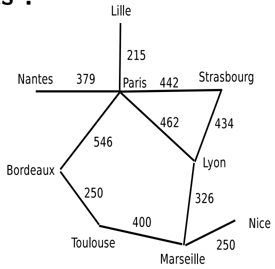
Des exemples :

- **Réseaux de communications ou de transports** : routiers, ferroviaires, machines en réseaux, circuits intégrés, électricité, eau...



Graphes et algorithme : Introduction

Un réseaux de transports :

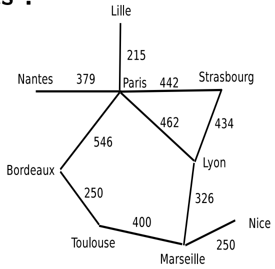


On distingue :

- les **sommets** ou **nœuds** ou **points** (villes, carrefours, gares, etc.)
- les **arcs** ou **arêtes** (communication entre sommets) :
    - mettent en relation les sommets.
    - peuvent en plus exprimer des distances, des coûts, des flux,...

Graphes et algorithme : Introduction

Un réseaux de transports :



On distingue :

- les **sommets** ou **nœuds** ou **points** (villes, carrefours, gares, etc.)
- les **arcs** ou **arêtes** (communication entre sommets) :
  - mettent en relation les sommets.
  - peuvent en plus exprimer des distances, des coûts, des flux,...

Permet d'exprimer des problèmes bien connus :

- itinéraire optimal entre deux sommets
- tournée optimale (problème classique du voyageur de commerce : visiter toutes les villes avec un cheminement optimal).

Graphes et algorithme : Introduction

Des exemples :

- Réseaux de communications ou de transports : routiers, ferroviaires, machines en réseaux, circuits intégrés, électricité, eau...

Graphes et algorithme : Introduction

Des exemples :

- Réseaux de communications ou de transports : routiers, ferroviaires, machines en réseaux, circuits intégrés, électricité, eau...
- **Relations sociales** : familiales (arbre généalogique), hiérarchiques, d'incompatibilité,...
  - sommets : les individus
  - arcs : relations entre individus.

Graphes et algorithme : Introduction

Des exemples :

- Réseaux de communications ou de transports : routiers, ferroviaires, machines en réseaux, circuits intégrés, électricité, eau...
- **Relations sociales** : familiales (arbre généalogique), hiérarchiques, d'incompatibilité,...
  - sommets : les individus
  - arcs : relations entre individus.
- **Ordonnancement de tâches** : ateliers, chantiers, centres de calcul...

  Avec plusieurs modélisations possibles :
  - représenter les instants de début et de fin d'activité par les sommets, et les activités par les arcs
  - représenter les tâches par les sommets et les relations de précédence (une fenêtre n'est posée qu'après la construction d'un mur) par les arcs.
  - etc.

Graphes et algorithme : Introduction

Des exemples :

- **Ordonnancement de tâches** : ateliers, chantiers, centres de calcul...
  Une modélisation possibles :
  - représenter les tâches par les sommets et les relations de précédence (une fenêtre n'est posée qu'après la construction d'un mur) par les arcs.

Graphes et algorithme : Introduction

Des exemples :

- **Ordonnancement de tâches** : ateliers, chantiers, centres de calcul...
  Une modélisation possibles :
  - représenter les tâches par les sommets et les relations de précédence (une fenêtre n'est posée qu'après la construction d'un mur) par les arcs.

Comment s'habiller en considérant 9 tâches numérotées de 1 à 9 ?

1 : calecon	4 : veste	7 : chemise
2 : pantalon	5 : cravate	8 : chaussures
3 : ceinture	6 : chaussettes	9 : montre

Graphes et algorithme : Introduction

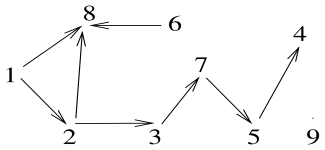
Des exemples :

- **Ordonnancement de tâches** : ateliers, chantiers, centres de calcul...
  Une modélisation possibles :
  - représenter les tâches par les sommets et les relations de précédence (une fenêtre n'est posée qu'après la construction d'un mur) par les arcs.

Comment s'habiller en considérant 9 tâches numérotées de 1 à 9 ?

1 : calecon	4 : veste	7 : chemise
2 : pantalon	5 : cravate	8 : chaussures
3 : ceinture	6 : chaussettes	9 : montre

Le **graphe de précédence** (entre tâches):



Graphes et algorithme : Introduction

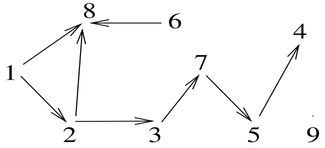
Des exemples :

- **Ordonnancement de tâches** : ateliers, chantiers, centres de calcul... Une modélisation possibles :
  - représenter les tâches par les sommets et les relations de précédence (une fenêtre n'est posée qu'après la construction d'un mur) par les arcs.

Comment s'habiller en considérant 9 tâches numérotées de 1 à 9 ?

1 : calecon	4 : veste	7 : chemise
2 : pantalon	5 : cravate	8 : chaussures
3 : ceinture	6 : chaussettes	9 : montre

Le **graphe de précédence** (entre tâches):



Se résout par un **tri topologique** : calcul d'un ordre des sommets compatible avec les arcs

Graphes et algorithme : Introduction

Des exemples :

- Réseaux de communications ou de transports
- Relations sociales
- Ordonnancement de tâches
- Et quantité d'autres objets, situations, problèmes :
  - **représentation d'algorithmes** (organigrammes)
  - **emplois du temps**
  - **automates à états finis**
  - **problèmes de coloration de cartes** (2 pays voisins doivent avoir 2 couleurs différentes)
  - **molécules en chimie**
  - **etc.**

Plan

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)

2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Graphes et algorithme : Définitions et terminologie

Définition

Un **graphe orienté** (fini) est un couple $G = (S, A)$ où :

- $S = \{x_1, x_2, \ldots x_n\}$ est un ensemble fini d'éléments appelés **sommets**
- $A \subset S \times S$ est un ensemble fini de couples de sommets appelés **arcs**

(s'il existe plusieurs arcs entre deux sommets (arcs multiples), on dit **multigraphe**)

Graphes et algorithme : Définitions et terminologie

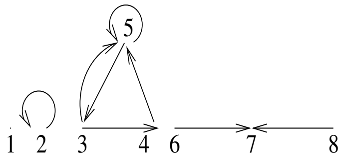
Définition

Un **graphe orienté** (fini) est un couple $G = (S, A)$ où :

- $S = \{x_1, x_2, \ldots x_n\}$ est un ensemble fini d'éléments appelés **sommets**
- $A \subset S \times S$ est un ensemble fini de couples de sommets appelés **arcs**

(s'il existe plusieurs arcs entre deux sommets (arcs multiples), on dit **multigraphe**)

Exemple : un graphe orienté $G = (S, A)$



avec

- $S = \{1, 2, \ldots 8\}$
- $A = \{(2,2), (3,5), (4,5), (3,4), (5,5), (5,3), (6,7), (8,7)\}$

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x
- **l'ensemble des successeurs** de x est $\Gamma^+(x) = \{y | (x, y) \in A\}$

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x
- **l'ensemble des successeurs** de x est $\Gamma^+(x) = \{y | (x, y) \in A\}$
- **l'ensemble des prédécesseurs** de x est $\Gamma^-(x) = \{y | (y, x) \in A\}$

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x
- **l'ensemble des successeurs** de x est $\Gamma^+(x) = \{y | (x, y) \in A\}$
- **l'ensemble des prédécesseurs** de x est $\Gamma^-(x) = \{y | (y, x) \in A\}$
- **l'ensemble des voisins** de x est $\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)$

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x
- **l'ensemble des successeurs** de x est $\Gamma^+(x) = \{y | (x, y) \in A\}$
- **l'ensemble des prédécesseurs** de x est $\Gamma^-(x) = \{y | (y, x) \in A\}$
- **l'ensemble des voisins** de x est $\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)$
- **le (demi) degré extérieur** de x est $d^+(x) = |\Gamma^+(x)|$

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x
- **l'ensemble des successeurs** de x est $\Gamma^+(x) = \{y | (x, y) \in A\}$
- **l'ensemble des prédécesseurs** de x est $\Gamma^-(x) = \{y | (y, x) \in A\}$
- **l'ensemble des voisins** de x est $\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)$
- **le (demi) degré extérieur** de x est $d^+(x) = |\Gamma^+(x)|$
- **le (demi) degré intérieur** de x est $d^-(x) = |\Gamma^-(x)|$

Graphes et algorithme : Définitions et terminologie

Définitions

Etant donné un graphe $G = (S, A)$ et $x \in S$

- si $(x, y) \in A$, alors y est un **successeur** de x
- si $(y, x) \in A$, alors y est un **prédécesseur** de x
- x et y sont **adjacents** ou **voisins** si y est un prédécesseur ou un successeur de x
- **l'ensemble des successeurs** de x est $\Gamma^+(x) = \{y | (x, y) \in A\}$
- **l'ensemble des prédécesseurs** de x est $\Gamma^-(x) = \{y | (y, x) \in A\}$
- **l'ensemble des voisins** de x est $\Gamma(x) = \Gamma^+(x) \cup \Gamma^-(x)$
- **le (demi) degré extérieur** de x est $d^+(x) = |\Gamma^+(x)|$
- **le (demi) degré intérieur** de x est $d^-(x) = |\Gamma^-(x)|$
- **le degré** de x est $d(x) = |\Gamma(x)|$
  (si G n'a pas de boucle, alors on a $d(x) = d^+(x) + d^-(x)$

Graphes et algorithme : Définitions et terminologie

Définition

Un **graphe non-orienté** (fini) est un couple $G = (S, A)$ où :

- $S = \{x_1, x_2, \ldots x_n\}$ est un ensemble fini d'éléments appelés **sommets**
- $A \subset S \times S$ est un ensemble fini de paires de sommets appelées **arêtes**
  Une arêtes entre deux sommets x et y est donc un ensemble $\{x, y\} = \{y, x\}$.

(pour les graphes non-orientés, on parle simplement de sommets voisins et de degré)

Graphes et algorithme : Définitions et terminologie

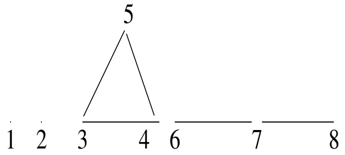
Définition

Un **graphe non-orienté** (fini) est un couple $G = (S, A)$ où :

- $S = \{x_1, x_2, \ldots x_n\}$ est un ensemble fini d'éléments appelés **sommets**
- $A \subset S \times S$ est un ensemble fini de paires de sommets appelées **arêtes** Une arêtes entre deux sommets x et y est donc un ensemble $\{x, y\} = \{y, x\}$.

(pour les graphes non-orientés, on parle simplement de sommets voisins et de degré)

Exemple : un graphe non-orienté $G = (S, A)$



avec

- $S = \{1, 2, \ldots 8\}$ et $A = \{\{3, 5\}, \{4, 5\}, \{3, 4\}, \{6, 7\}, \{8, 7\}\}$

Graphes et algorithme : Définitions et terminologie

Définition: **graphe complet**

- Un graphe orienté $G = (S, A)$ est dit **complet** si $A = S^2$ (tous les arcs possibles son présents)
- Un graphe non-orienté $G = (S, A)$ est dit **complet** si toute paire de sommets figure dans A

Graphes et algorithme : Définitions et terminologie

Définition: **graphe complet**

- Un graphe orienté $G = (S, A)$ est dit **complet** si $A = S^2$ (tous les arcs possibles son présents)
- Un graphe non-orienté $G = (S, A)$ est dit **complet** si toute paire de sommets figure dans A

Définition : **graphe réciproque**

Etant donné un graphe orienté $G = (S, A)$:

- le **graphe réciproque de** G est le graphe $G^{-1} = (S, A^{-1})$

  où $A^{-1} = \{(x, y) \in S \times S | (y, x) \in A\}$

Graphes et algorithme : Définitions et terminologie

Définition: **graphe complet**

- Un graphe orienté $G = (S, A)$ est dit **complet** si $A = S^2$ (tous les arcs possibles son présents)
- Un graphe non-orienté $G = (S, A)$ est dit **complet** si toute paire de sommets figure dans A

Définition : **graphe réciproque**

Etant donné un graphe orienté $G = (S, A)$:

- le **graphe réciproque de** G est le graphe $G^{-1} = (S, A^{-1})$

  où $A^{-1} = \{(x, y) \in S \times S | (y, x) \in A\}$

Définition : **graphe symétrisé**

Etant donné un graphe orienté $G = (S, A)$:

- le **graphe symétrisé** associé à G est le graphe $G_{Sym} = (S, A \cup A^{-1})$

Graphes et algorithme : Définitions et terminologie

Définition : **graphe valué**

Un **graphe valué** est un graphe orienté ou non $G = (S, A)$ auquel on associe une fonction de coût $c : A \to \mathbb{R}$. Un tel graphe est noté (S, A, c)

Graphes et algorithme : Définitions et terminologie

Définition : **graphe valué**

Un **graphe valué** est un graphe orienté ou non $G = (S, A)$ auquel on associe une fonction de coût $c : A \to \mathbb{R}$. Un tel graphe est noté (S, A, c)

Définition : **graphe étiqueté**

Un **graphe étiqueté aux sommets et aux arcs ou arêtes** est un graphe orienté ou non-orienté $G = (S, A)$ auquel on associe :

- une fonction d'étiquetage des sommets $e_S : S \to E_S$
- une fonction d'étiquetage des arcs ou des arêtes $e_A : A \to E_A$

Un tel graphe est noté (S, A, e_S, e_A)

Graphes et algorithme : Définitions et terminologie

Définition : graphe valué

Un **graphe valué** est un graphe orienté ou non $G = (S, A)$ auquel on associe une fonction de coût $c : A \to \mathbb{R}$. Un tel graphe est noté (S, A, c)

Définition : graphe étiqueté

Un **graphe étiqueté aux sommets et aux arcs ou arêtes** est un graphe orienté ou non-orienté $G = (S, A)$ auquel on associe :

- une fonction d'étiquetage des sommets $e_S : S \to E_S$
- une fonction d'étiquetage des arcs ou des arêtes $e_A : A \to E_A$

Un tel graphe est noté (S, A, e_S, e_A)

Exemple : un automate d'états fini $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ est un multigraphe avec :

- la fonction d'étiquetage des sommets $e_S : S \to Q$ associe un état à chaque sommet
- une fonction d'étiquetage des arcs $e_A : A \to \Sigma$ induite par δ

Graphes et algorithme : Définitions et terminologie

Définition : **graphe valué et étiqueté**

Un **graphe valué et étiqueté** est un graphe orienté ou non-orienté $G = (S, A)$ auquel on associe :

- une fonction de coût $c : A \rightarrow \mathbb{R}$
- une fonction d'étiquetage $e : S \rightarrow E$

Un tel graphe est noté (S, A, c, e)

Graphes et algorithme : Définitions et terminologie

Définition : **graphe valué et étiqueté**

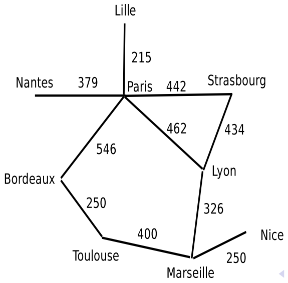
Un **graphe valué et étiqueté** est un graphe orienté ou non-orienté $G = (S, A)$ auquel on associe :

- une fonction de coût $c : A \to \mathbb{R}$
- une fonction d'étiquetage $e : S \to E$

Un tel graphe est noté (S, A, c, e)

Exemple : le graphe du réseau de transport déja vu :

- la fonction de coût $c : A \to \mathbb{R}$ correspond aux distances
- une fonction d'étiquetage $e : S \to E$ correspond au nom des villes



Graphes et algorithme : Définitions et terminologie

Un peu de dénombrement

(utile pour l'analyse de la complexité des algorithmes)

Notations

Pour $G = (S, A)$, on notera $|S| = card(S) = n$ et $|A| = card(A) = m$

Graphes et algorithme : Définitions et terminologie

Un peu de dénombrement

(utile pour l'analyse de la complexité des algorithmes)

Notations

Pour $G = (S, A)$, on notera $|S| = card(S) = n$ et $|A| = card(A) = m$

Rapports entre n et m (cas des graphes orientés)

Si $G = (S, A)$ n'est pas un multigraphe, on a $A \subseteq S^2$

- graphes complets avec boucles : $m = n^2$
- graphes complets sans boucle : $m = n(n-1)$

Graphes et algorithme : Définitions et terminologie

Un peu de dénombrement

(utile pour l'analyse de la complexité des algorithmes)

Notations

Pour $G = (S, A)$, on notera $|S| = card(S) = n$ et $|A| = card(A) = m$

Graphes et algorithme : Définitions et terminologie

Un peu de dénombrement
(utile pour l'analyse de la complexité des algorithmes)

Notations

Pour $G = (S, A)$, on notera $|S| = card(S) = n$ et $|A| = card(A) = m$

Relations entre d^+, d^-, d et m

Si $G = (S, A)$ n'est pas un multigraphe, on a $A \subseteq S^2$

- pour le cas des graphes orientés sans boucles :

$$\sum_{x \in S} d^+(x) = \sum_{x \in S} d^-(x) = \frac{1}{2} \sum_{x \in S} d(x) = m$$

Graphes et algorithme : Définitions et terminologie

Un peu de dénombrement
(utile pour l'analyse de la complexité des algorithmes)

Notations

Pour $G = (S, A)$, on notera $|S| = card(S) = n$ et $|A| = card(A) = m$

Relations entre d^+, d^-, d et m

Si $G = (S, A)$ n'est pas un multigraphe, on a $A \subseteq S^2$

- pour le cas des graphes orientés sans boucles :

$$\sum_{x \in S} d^+(x) = \sum_{x \in S} d^-(x) = \frac{1}{2} \sum_{x \in S} d(x) = m$$

- pour le cas des graphes non-orientés sans boucles :

$$\sum_{x \in S} d(x) = 2m$$

Graphes et algorithme : Définitions et terminologie

Définition : **sous-graphe**

Étant donné un graphe orienté $G = (S, A)$ et un sous-ensemble de sommets $X \subseteq S$, le **sous-graphe de** G **induit (ou engendré) par** X est $G(X) = (X, E)$ où $E = \{(x, y) \in A | x, y \in X\}$.

(on prend la partie du graphe réduite aux sommets de X)

La définition est similaire pour le cas des graphes non-orientés.

Graphes et algorithme : Définitions et terminologie

Définition : **sous-graphe**

Étant donné un graphe orienté $G = (S, A)$ et un sous-ensemble de sommets $X \subseteq S$, le **sous-graphe de** G **induit (ou engendré) par** X est $G(X) = (X, E)$ où $E = \{(x, y) \in A | x, y \in X\}$.

(on prend la partie du graphe réduite aux sommets de X)

La définition est similaire pour le cas des graphes non-orientés.

Définition : **graphe partiel**

Étant donné un graphe orienté ou non $G = (S, A)$ et un sous ensemble d'arcs ou d'arêtes $E \subseteq A$, le **graphe partiel de** G **induit par** E est le graphe $G(E) = (S, E)$.

(on prend tous les sommets du graphe en ne conservant que les arcs ou d'arêtes de E)

Graphes et algorithme : Définitions et terminologie

Définition : **sous-graphe**

Étant donné un graphe orienté $G = (S, A)$ et un sous-ensemble de sommets $X \subseteq S$, le **sous-graphe de** G **induit (ou engendré) par** X est $G(X) = (X, E)$ où
$E = \{(x, y) \in A | x, y \in X\}$.

(on prend la partie du graphe réduite aux sommets de X)

La définition est similaire pour le cas des graphes non-orientés.

Définition : **graphe partiel**

Étant donné un graphe orienté ou non $G = (S, A)$ et un sous ensemble d'arcs ou d'arêtes $E \subseteq A$, le **graphe partiel de** G **induit par** E est le graphe $G(E) = (S, E)$.

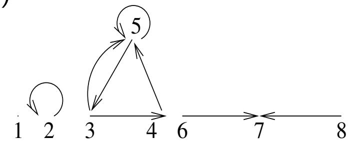
(on prend tous les sommets du graphe en ne conservant que les arcs ou d'arêtes de E)

Définition : **sous-graphe partiel**

Même principe que pour les sous-graphes et les graphes partiels mais la restriction porte à la fois sur les sommets et les arcs ou arêtes.

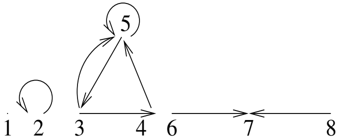
Graphes et algorithme : Définitions et terminologie

- Graphe $G = (S, A)$

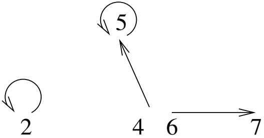


Graphes et algorithme : Définitions et terminologie

- Graphe $G = (S, A)$

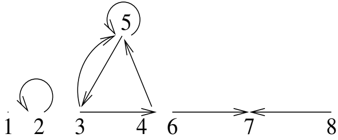


- $G(X)$: sous-graphe induit par $X = \{2, 4, 5, 6, 7\}$

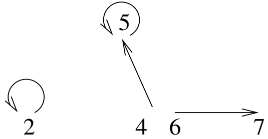


Graphes et algorithme : Définitions et terminologie

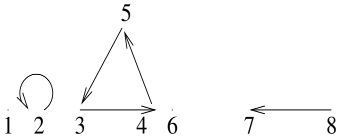
- Graphe $G = (S, A)$



- $G(X)$: sous-graphe induit par $X = \{2, 4, 5, 6, 7\}$



- $G(E)$: graphe partiel induit par $E = \{(2, 2), (3, 4), (4, 5), (5, 3), (8, 7)\}$



Graphes et algorithme : Définitions et terminologie

Graphe quotient : une forme d'abstraction (ou de vue synthétique)

Partition d'un ensemble ? Rappel :

$P = \{S_1, S_2, \dots S_p\}$ est une **partition** de S ssi

- $\bigcup_{1 \leq i \leq p} S_i = S$ (tous les sommets figurent dans une partie) et
- $\forall i \neq j,\ S_i \cap S_j = \emptyset$ (un sommet ne figure que dans une partie)

Graphes et algorithme : Définitions et terminologie

Graphe quotient : une forme d'abstraction (ou de vue synthétique)

Partition d'un ensemble ? Rappel :

$P = \{S_1, S_2, \dots S_p\}$ est une **partition** de S ssi

- $\bigcup_{1 \leq i \leq p} S_i = S$ (tous les sommets figurent dans une partie) et
- $\forall i \neq j$, $S_i \cap S_j = \emptyset$ (un sommet ne figure que dans une partie)

Définition (dans le cadre des graphes orientés)

Étant donnés $G = (S, A)$ et une partition $P = \{S_1, S_2, \dots S_p\}$ de S,
le graphe quotient de G **induit par** P est le graphe $G/_P = (V, E)$ où :

Graphes et algorithme : Définitions et terminologie

Graphe quotient : une forme d'abstraction (ou de vue synthétique)

Partition d'un ensemble ? Rappel :

$P = \{S_1, S_2, \ldots S_p\}$ est une **partition** de S ssi

- $\bigcup_{1 \leq i \leq p} S_i = S$ (tous les sommets figurent dans une partie) et
- $\forall i \neq j,\ S_i \cap S_j = \emptyset$ (un sommet ne figure que dans une partie)

Définition (dans le cadre des graphes orientés)

Étant donnés $G = (S, A)$ et une partition $P = \{S_1, S_2, \ldots S_p\}$ de S,
le graphe quotient de G **induit par** P est le graphe $G/_P = (V, E)$ où :

- $V = \{s_1, s_2, \ldots s_p\}$
  (à chaque partie S_i de P, on associe un sommet s_i)

Graphes et algorithme : Définitions et terminologie

Graphe quotient : une forme d'abstraction (ou de vue synthétique)

Partition d'un ensemble ? Rappel :

$P = \{S_1, S_2, \ldots S_p\}$ est une **partition** de S ssi

- $\bigcup\limits_{1 \leq i \leq p} S_i = S$ (tous les sommets figurent dans une partie) et
- $\forall i \neq j,\ S_i \cap S_j = \emptyset$ (un sommet ne figure que dans une partie)

Définition (dans le cadre des graphes orientés)

Étant donnés $G = (S, A)$ et une partition $P = \{S_1, S_2, \ldots S_p\}$ de S, **le graphe quotient de** G **induit par** P est le graphe $G/_P = (V, E)$ où :

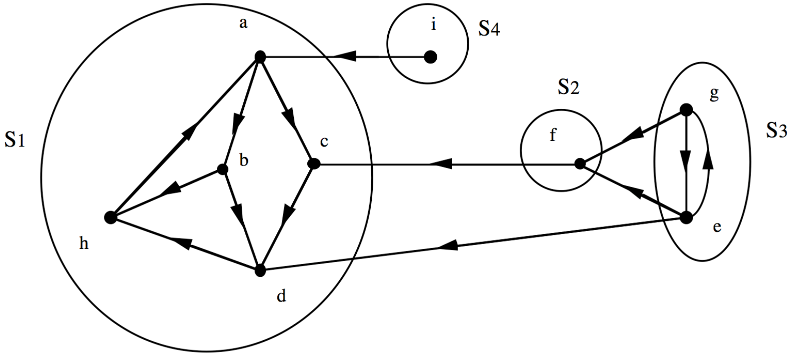
- $V = \{s_1, s_2, \ldots s_p\}$
  (à chaque partie S_i de P, on associe un sommet s_i)
- $E = \{(s_i, s_j) \mid \exists (x, y) \in A \text{ avec } x \in S_i \text{ et } y \in S_j\}$ (deux parties sont connectées s'il existe au moins un arc entre elles)

Graphes et algorithme : Définitions et terminologie

Graphe quotient : Vite, un exemple !

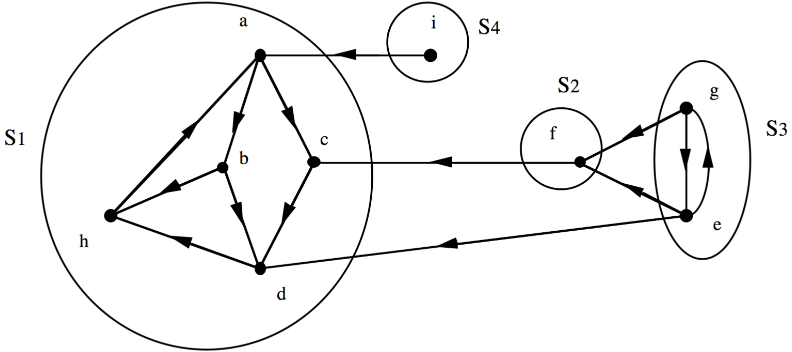
Graphes et algorithme : Définitions et terminologie

Graphe quotient : Vite, un exemple ! Un graphe $G = (S, A)$ avec une partition $P = \{S_1 = \{a, b, c, d, h\}, S_2 = \{f\}, S_3 = \{e, g\}, S_4 = \{i\}\}$ de S

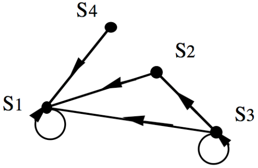


Graphes et algorithme : Définitions et terminologie

Graphe quotient : Vite, un exemple ! Un graphe $G = (S, A)$ avec une partition $P = \{S_1 = \{a, b, c, d, h\}, S_2 = \{f\}, S_3 = \{e, g\}, S_4 = \{i\}\}$ de S



Le graphe quotient $G/_P = (V, E)$ du graphe G induit par P :



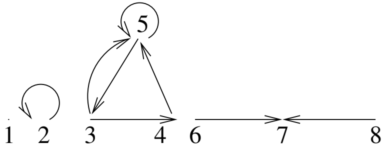
1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)

2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Représentation machine

Les graphes comme une structure de données

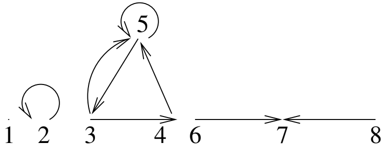
(1) Par matrices d'adjacence :



Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :



Représentation par tableau à deux dimensions indexé sur l'ensemble des sommets

S	1	2	3	4	5	6	7	8
1	F	F	F	F	F	F	F	F
2	F	V	F	F	F	F	F	F
3	F	F	F	V	V	F	F	F
4	F	F	F	F	V	F	F	F
5	F	F	V	F	V	F	F	F
6	F	F	F	F	F	F	V	F
7	F	F	F	F	F	F	F	F
8	F	F	F	F	F	F	V	F

Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :

Éléments du tableau : de différents types selon le graphe considéré

- **Booléen** : cas standard
  - graphe orienté : un élément [i,j] est vrai si l'arc (i, j) est présent
  - graphe non-orienté : un élément [i,j] est vrai $\Rightarrow$ l'élément [j,i] est vrai

Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :

Éléments du tableau : de différents types selon le graphe considéré

- **Booléen** : cas standard
  - graphe orienté : un élément [i,j] est vrai si l'arc (i, j) est présent
  - graphe non-orienté : un élément [i,j] est vrai $\Rightarrow$ l'élément [j,i] est vrai
- **Réel** : cas des graphes valués : représentation de la fonction de coût
  - valeur $c(i, j)$ si l'arc (i, j) figure dans le graphe
  - une valeur conventionnelle si l'arc (i, j) ne figure pas dans le graphe

Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :

Éléments du tableau : de différents types selon le graphe considéré

- **Booléen** : cas standard
  - graphe orienté : un élément [i,j] est vrai si l'arc (i, j) est présent
  - graphe non-orienté : un élément [i,j] est vrai $\Rightarrow$ l'élément [j,i] est vrai
- **Réel** : cas des graphes valués : représentation de la fonction de coût
  - valeur $c(i, j)$ si l'arc (i, j) figure dans le graphe
  - une valeur conventionnelle si l'arc (i, j) ne figure pas dans le graphe
- **Entier** : cas des multigraphes :

  [i,j] aura pour valeur le nombre d'occurrence de l'arc (i, j) dans le graphe.

Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :

Implémentation en langage C

```
#define NMAX .../* nombre maximum de sommets */

typedef int SOMMET; /* indice des sommets */

/* representation par matrices d'adjacence */
typedef struct {    int A[NMAX][NMAX]; /* matrice carree 0/1 */;
                    int n; /* valeur comprise entre 0 et NMAX */
} GRAPHEMAT;
```

Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :

- **Avantages**
  - Accès direct aux arcs : test d'existence d'un arc, ajout, ou suppression réalisables en temps constant : $\Theta(1)$
  - Accès aux prédécesseurs d'un sommet facilité : $\Theta(n)$
  - Écriture des algorithmes généralement simplifiée

Représentation machine

Les graphes comme une structure de données

(1) Par matrices d'adjacence :

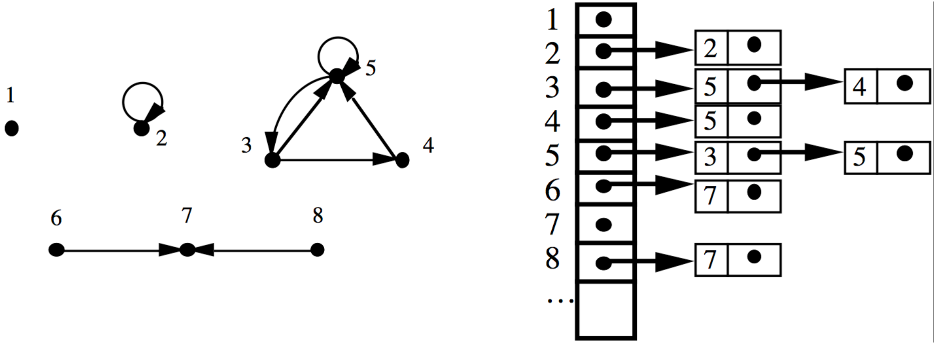
- **Avantages**
  - Accès direct aux arcs : test d'existence d'un arc, ajout, ou suppression réalisables en temps constant : $\Theta(1)$
  - Accès aux prédécesseurs d'un sommet facilité : $\Theta(n)$
  - Écriture des algorithmes généralement simplifiée
- **Inconvénients**
  - Encombrement maximal de la mémoire :
    $\forall$ le nombre d'arcs dans le graphe : espace mémoire en $\Theta(\text{n\_max}^2)$
    cas très génant : $m \ll n^2$
  - Coût de l'initialisation : $\Theta(n^2)$
  - Coût de la recherche des successeurs d'un sommet : $\Theta(\text{n})$
    (même s'il y'a pas de successeurs)

Représentation machine

Les graphes comme une structure de données

(2) Par listes d'adjacence :

Représentation par un tableau de listes indexée sur l'ensemble des sommets



Représentation standard car algos. généralement les plus performants

Représentation machine

Les graphes comme une structure de données

(2) Par listes d'adjacence :

Implémentation en langage C

```
/* representation par listes d'adjacence */
/* avec en premier definition des listes */
typedef struct maillon {  SOMMET st;
                          struct maillon *suivant;
} CHAINON;
typedef CHAINON *PTR_CHAINON;

typedef struct {    PTR_CHAINON A[NMAX]; /* tableau de listes */;
                    int n; /* valeur comprise entre 0 et NMAX */
} GRAPHELIST;
```

Représentation machine

Les graphes comme une structure de données

(2) Par listes d'adjacence :

- **Avantages**
  - Encombrement minimal de la mémoire : $\Theta(n\_max + m)$
  - Accès aux successeurs d'un sommet x linéaire en $\Theta(d^+(x))$

Représentation machine

Les graphes comme une structure de données

(2) Par listes d'adjacence :

- **Avantages**
  - Encombrement minimal de la mémoire : $\Theta(n\_max + m)$
  - Accès aux successeurs d'un sommet x linéaire en $\Theta(d^+(x))$
- **Inconvénients**
  - Test d'existence d'un arc (x, y) : en $\Theta(d^+(x))$
  - Accès aux prédécesseurs d'un sommet $x \Rightarrow$
    parcours de toutes les listes d'adjacence : $\Theta(n + m)$

1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)

2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes orientés)

- (Définition par séquence d'arcs) Un **chemin** de $G = (S, A)$ est une séquence d'arcs $(a_1, a_2, \ldots a_{k-1})$ t.q.
  $\forall i, 1 \leq i \leq k-1, a_i = (x_i, x_{i+1}) \in A$.

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes orientés)

- (Définition par séquence d'arcs) Un **chemin** de $G = (S, A)$ est une séquence d'arcs $(a_1, a_2, \ldots a_{k-1})$ t.q. $\forall i, 1 \leq i \leq k-1, a_i = (x_i, x_{i+1}) \in A$.
- (Définition par séquence de sommets) Un **chemin** de $G = (S, A)$ peut se définir par la séquence de sommets $(x_1, x_2, \ldots x_{k-1}, x_k)$ qui le composent.
  On le note par $[x_1, x_2, \ldots x_k]$, voire parfois par $[x_1, x_k]$

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes orientés)

- (Définition par séquence d'arcs) Un **chemin** de $G = (S, A)$ est une séquence d'arcs $(a_1, a_2, \ldots a_{k-1})$ t.q. $\forall i, 1 \leq i \leq k-1, a_i = (x_i, x_{i+1}) \in A$.
- (Définition par séquence de sommets) Un **chemin** de $G = (S, A)$ peut se définir par la séquence de sommets $(x_1, x_2, \ldots x_{k-1}, x_k)$ qui le composent.
  On le note par $[x_1, x_2, \ldots x_k]$, voire parfois par $[x_1, x_k]$
- Un chemin $[x_1, x_2, \ldots x_k]$ est dit **élémentaire** s'il ne contient pas deux fois le même sommet, i.e. $\forall i, j$ avec $1 \leq i < j \leq k$, $x_i \neq x_j$

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes orientés)

- (Définition par séquence d'arcs) Un **chemin** de $G = (S, A)$ est une séquence d'arcs $(a_1, a_2, \dots a_{k-1})$ t.q.
  $\forall i, 1 \leq i \leq k-1, a_i = (x_i, x_{i+1}) \in A$.
- (Définition par séquence de sommets) Un **chemin** de $G = (S, A)$ peut se définir par la séquence de sommets $(x_1, x_2, \dots x_{k-1}, x_k)$ qui le composent.
  On le note par $[x_1, x_2, \dots x_k]$, voire parfois par $[x_1, x_k]$
- Un chemin $[x_1, x_2, \dots x_k]$ est dit **élémentaire** s'il ne contient pas deux fois le même sommet, i.e. $\forall i, j$ avec $1 \leq i < j \leq k$, $x_i \neq x_j$
- La **longueur d'un chemin** est égale à son nombre arcs (nombre de sommets moins un)
  Un chemin d'un seul sommet x_1 noté $[x_1]$ est de **longueur nulle**

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes non-orientés)

- (Définition par séquence d'arêtes) Une **chaîne** de $G = (S, A)$ est une séquence d'arêtes $(a_1, \ldots a_{k-1})$ t.q. $\forall i, 1 \leq i < k, a_i = \{x_i, x_{i+1}\} \in A$

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes non-orientés)

- (Définition par séquence d'arêtes) Une **chaîne** de $G = (S, A)$ est une séquence d'arêtes $(a_1, \ldots a_{k-1})$ t.q. $\forall i, 1 \leq i < k, a_i = \{x_i, x_{i+1}\} \in A$
- (Définition par séquence de sommets) Une **chaîne** de $G = (S, A)$ peut se définir par la séquence de sommets $(x_1, x_2, \ldots x_{k-1}, x_k)$ qui la composent. On la note par $(x_1, x_2, \ldots x_k)$

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes non-orientés)

- (Définition par séquence d'arêtes) Une **chaîne** de $G = (S, A)$ est une séquence d'arêtes $(a_1, \dots a_{k-1})$ t.q. $\forall i, 1 \leq i < k, a_i = \{x_i, x_{i+1}\} \in A$
- (Définition par séquence de sommets) Une **chaîne** de $G = (S, A)$ peut se définir par la séquence de sommets $(x_1, x_2, \dots x_{k-1}, x_k)$ qui la composent. On la note par $(x_1, x_2, \dots x_k)$
- Une chaîne $(x_1, x_2, \dots x_k)$ est dite **élémentaire** s'il ne contient pas deux fois le même sommet, i.e. $\forall i, j$ avec $1 \leq i < j \leq k$, $x_i \neq x_j$

Parcours, numérotations et connexité : Introduction

Définition (Cheminement dans les graphes non-orientés)

- (Définition par séquence d'arêtes) Une **chaîne** de $G = (S, A)$ est une séquence d'arêtes $(a_1, \ldots a_{k-1})$ t.q. $\forall i, 1 \leq i < k, a_i = \{x_i, x_{i+1}\} \in A$
- (Définition par séquence de sommets) Une **chaîne** de $G = (S, A)$ peut se définir par la séquence de sommets $(x_1, x_2, \ldots x_{k-1}, x_k)$ qui la composent. On la note par $(x_1, x_2, \ldots x_k)$
- Une chaîne $(x_1, x_2, \ldots x_k)$ est dite **élémentaire** s'il ne contient pas deux fois le même sommet, i.e. $\forall i, j$ avec $1 \leq i < j \leq k$, $x_i \neq x_j$
- La **longueur d'une chaîne** est égale à son nombre d'arêtes (nombre de sommets moins un)
  Une chaîne d'un seul sommet x_1 notée (x_1) est de **longueur nulle**

NB. On peut parler de chaîne dans les graphes non-orientés en ne considérant pas l'orientation des arcs (un arc est assimilé à une arête).

Parcours, numérotations et connexité : Introduction

Définition

- un **circuit de** $G = (S, A)$ est un chemin $[x_1, x_2, \ldots x_k]$ de G tel que $k \geq 2$ et $x_1 = x_k$.
- un **cycle de** $G = (S, A)$ est une chaine $(x_1, x_2, \ldots x_k)$ de G telle que $k \geq 2$ et $x_1 = x_k$.

Parcours, numérotations et connexité : Introduction

Définition

- un **circuit de** $G = (S, A)$ est un chemin $[x_1, x_2, \dots x_k]$ de G tel que $k \geq 2$ et $x_1 = x_k$.
- un **cycle de** $G = (S, A)$ est une chaine $(x_1, x_2, \dots x_k)$ de G telle que $k \geq 2$ et $x_1 = x_k$.

Définition (Cas des graphes orientés)

Étant donné un graphe $G = (S, A)$ et $x \in S$, on définit les ensembles de **descendants** et **ascendants** de x par :

- $Desc_G(x) = \{y \in S \mid \exists [x = x_1, x_2, \dots x_k = y] \text{ chemin de } G\}$
- $Asc_G(x) = \{y \in S \mid \exists [y = x_1, x_2, \dots x_k = x] \text{ chemin de } G\}$
- Une **racine** de $G = (S, A)$ est un sommet $r \in S$ tel que $S = Desc_G(r)$

Parcours et numérotations

Parcours d'un graphe : visite de ses sommets en empruntant ses arcs (ou ses arêtes) et donc ses chemins (ou ses chaînes). Deux stratégies existent :

Parcours et numérotations

Parcours d'un graphe : visite de ses sommets en empruntant ses arcs (ou ses arêtes) et donc ses chemins (ou ses chaînes). Deux stratégies existent :

- **Parcours en profondeur** (depth-first search)
  - Visite la descendance en suivant un chemin le plus loin possible.
  - En cas d'impasse : remontée (backtrack).
  - Approche naturellement récursive.

Parcours et numérotations

Parcours d'un graphe : visite de ses sommets en empruntant ses arcs (ou ses arêtes) et donc ses chemins (ou ses chaînes). Deux stratégies existent :

- **Parcours en profondeur** (depth-first search)
  - Visite la descendance en suivant un chemin le plus loin possible.
  - En cas d'impasse : remontée (backtrack).
  - Approche naturellement récursive.
- **Parcours en largeur** (breadth-first search)
  - Visite la descendance en accédant aux sommets niveau par niveau (éloignement progressif du sommet de départ).
  - Approche naturellement fondée sur les files d'attentes.

Parcours et numérotations

Parcours d'un graphe : visite de ses sommets en empruntant ses arcs (ou ses arêtes) et donc ses chemins (ou ses chaînes). Deux stratégies existent :

- **Parcours en profondeur** (depth-first search)
  - Visite la descendance en suivant un chemin le plus loin possible.
  - En cas d'impasse : remontée (backtrack).
  - Approche naturellement récursive.
- **Parcours en largeur** (breadth-first search)
  - Visite la descendance en accédant aux sommets niveau par niveau (éloignement progressif du sommet de départ).
  - Approche naturellement fondée sur les files d'attentes.

Dans chaque cas :

- Démarre d'un premier sommet
- Pour éviter de boucler dans les circuits : marquage des sommets déjà visités
- Si aucun nouveau sommet accessible : redémarrage d'un sommet pas encore visité.
- En $\Theta(n+m)$ pour les listes et $\Theta(n^2)$ pour les matrices

Parcours et numérotations

Numérotation d'un graphe : numérotation des sommets en fonction d'un parcours et d'un ordre.

Parcours et numérotations

Numérotation d'un graphe : numérotation des sommets en fonction d'un parcours et d'un ordre.

- **En pré-ordre :** Des qu'un sommet est atteint, il est numéroté.

Parcours et numérotations

Numérotation d'un graphe : numérotation des sommets en fonction d'un parcours et d'un ordre.

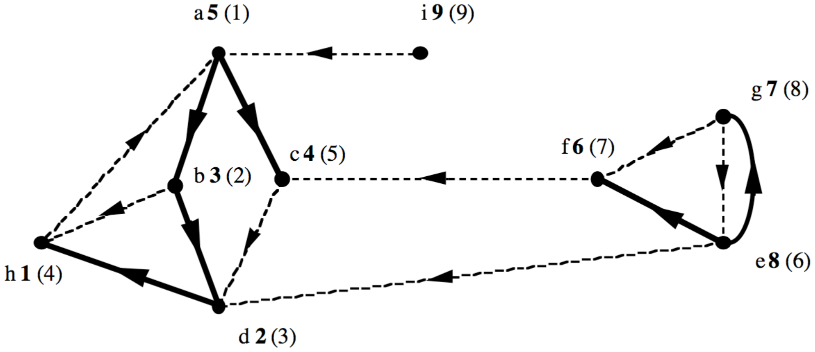
- **En pré-ordre :** Des qu'un sommet est atteint, il est numéroté.
- **En post-ordre :** Quand toute la descendance d'un sommet est numérotée, il est alors numéroté.

Parcours et numérotations

Numérotation d'un graphe : numérotation des sommets en fonction d'un parcours et d'un ordre.

- **En pré-ordre :** Des qu'un sommet est atteint, il est numéroté.
- **En post-ordre :** Quand toute la descendance d'un sommet est numérotée, il est alors numéroté.

Exemple : numérotation en pré-ordre (entre parenthèses) et en post-ordre (en gras) (les sommets sont $a, b, c, \ldots i$)



1. Analyse de la Complexité des Algorithmes
   - Motivations
   - Un modèle d'analyse fondé sur quelques principes
   - Quelques exemples moins triviaux (mais du niveau L2...)
   - Notations Θ, O, et Ω : définitions formelles
   - Incidence du codage et précautions pour le modèle d'analyse
   - De la théorie (complexité) à la pratique (efficacité pratique)
2. Algorithmique des graphes : quelques rappels
   - Graphes et algorithmes : présentation
     - Introduction
     - Définitions et terminologie
     - Représentation machine
   - Graphes et algorithmes : Parcours, numérotations et connexité
     - Introduction
     - Parcours et numérotations
     - ConnexitéS

ConnexitéS

- **Problèmes de connexité :**
  - D'un intérêt pratique majeur
  - De nombreuses applications liées à toutes sortes de réseaux

    pour savoir quels sont les sommets qui sont reliés sans chercher à savoir explicitement comment

ConnexitéS

- **Problèmes de connexité :**
  - D'un intérêt pratique majeur
  - De nombreuses applications liées à toutes sortes de réseaux

    pour savoir quels sont les sommets qui sont reliés sans chercher à savoir explicitement comment

- **Deux types de connexités :**
  - Celle liée à l'existence de chaînes : notion de **connexité**

    On parle de **composantes connexes**
  - Celle liée à l'existence de chemins : notion de **forte connexité**

    On parle de **composantes fortement connexes**

Relation de Connexité

Définition

La **relation de connexité** sur S pour un graphe $G = (S, A)$, notée R_c est définie par : $\forall x, y \in S$,

$$xR_cy \Leftrightarrow \exists \text{ une chaîne } (x_1 = x, x_2, \dots x_k = y) \text{ dans } G$$

Relation de Connexité

Définition

La **relation de connexité** sur S pour un graphe $G = (S, A)$, notée R_c est définie par : $\forall x, y \in S$,

$$xR_cy \Leftrightarrow \exists \text{ une chaîne } (x_1 = x, x_2, \ldots x_k = y) \text{ dans } G$$

Propriété

R_c est une relation d'équivalence

Relation de Connexité

Définition

La **relation de connexité** sur S pour un graphe $G = (S, A)$, notée R_c est définie par : $\forall x, y \in S$,

$$xR_cy \Leftrightarrow \exists \text{ une chaîne } (x_1 = x, x_2, \ldots x_k = y) \text{ dans } G$$

Propriété

R_c est une relation d'équivalence

Preuve : par définition des chaines, R_c est bien une relation qui est :

- réflexive : $\forall x \in S, xR_cx$
  (cf. chaînes de longueur nulle)

Relation de Connexité

Définition

La **relation de connexité** sur S pour un graphe $G = (S, A)$, notée R_c est définie par : $\forall x, y \in S$,

$$xR_cy \Leftrightarrow \exists \text{ une chaîne } (x_1 = x, x_2, \ldots x_k = y) \text{ dans } G$$

Propriété

R_c est une relation d'équivalence

Preuve : par définition des chaines, R_c est bien une relation qui est :

- réflexive : $\forall x \in S, xR_cx$
  (cf. chaînes de longueur nulle)
- symétrique : $\forall x, y \in S, xR_cy \Leftrightarrow yR_cx$
  (la notion de chaînes n'est pas orientée)

Relation de Connexité

Définition

La **relation de connexité** sur S pour un graphe $G = (S, A)$, notée R_c est définie par : $\forall x, y \in S$,

$$xR_cy \Leftrightarrow \exists \text{ une chaîne } (x_1 = x, x_2, \ldots x_k = y) \text{ dans } G$$

Propriété

R_c est une relation d'équivalence

Preuve : par définition des chaines, R_c est bien une relation qui est :

- réflexive : $\forall x \in S, xR_cx$
  (cf. chaînes de longueur nulle)
- symétrique : $\forall x, y \in S, xR_cy \Leftrightarrow yR_cx$
  (la notion de chaînes n'est pas orientée)
- transitive : $\forall x, y, z \in S$, si xR_cy et $yR_cz \Rightarrow xR_cz$
  (par composition de chaînes)

Notion de Composantes Connexe

Définition

Soit $G = (S, A)$ un graphe. Une classe d'équivalence de S modulo R_c est appellée **composante connexe** de G.

Un graphe est dit **connexe** s'il n'a qu'une seule composante connexe.

(petit rappel : classe d'équivalence de S modulo R_c = sommets en relations)

Notion de Composantes Connexe

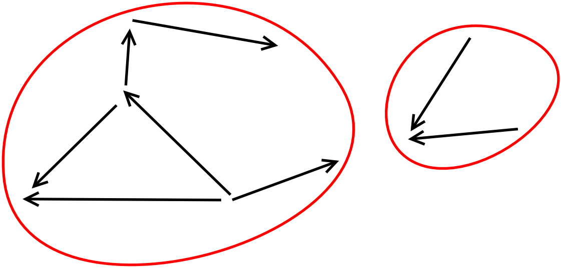
Définition

Soit $G = (S, A)$ un graphe. Une classe d'équivalence de S modulo R_c est appellée **composante connexe** de G.

Un graphe est dit **connexe** s'il n'a qu'une seule composante connexe.

(petit rappel : classe d'équivalence de S modulo R_c = sommets en relations)

Exemple : un graphe avec deux composantes connexes



Calcul des Composantes Connexe

Deux propriétés utiles :

Calcul des Composantes Connexe

Deux propriétés utiles :

Propriété

Les composantes connexes d'un graphe G sont les mêmes que celles de son graphe symétrisé G_{Sym}

Preuve : triviale !

$\exists$ une chaîne $(x_1 = x, x_2, \ldots, x_k = y)$ dans $G \Leftrightarrow$

$\exists$ une chaîne $(x_1 = x, x_2, \ldots, x_k = y)$ dans G_{Sym}.

Calcul des Composantes Connexe

Deux propriétés utiles :

Propriété

Les composantes connexes d'un graphe G sont les mêmes que celles de son graphe symétrisé G_{Sym}

Preuve : triviale !

$\exists$ une chaîne $(x_1 = x, x_2, \ldots, x_k = y)$ dans $G \Leftrightarrow$

$\exists$ une chaîne $(x_1 = x, x_2, \ldots, x_k = y)$ dans G_{Sym}.

Propriété

Etant donné un graphe $G = (S, A)$

$\forall x \in S, Desc_{G_{Sym}}(x) =$ Composante connexe contenant x dans G_{Sym}

Preuve : triviale en montrant la double inclusion !

Calcul des Composantes Connexe

Deux propriétés utiles :

Propriété

Les composantes connexes d'un graphe G sont les mêmes que celles de son graphe symétrisé G_{Sym}

Preuve : triviale !
$\exists$ une chaîne $(x_1 = x, x_2, \ldots, x_k = y)$ dans $G \Leftrightarrow$
$\exists$ une chaîne $(x_1 = x, x_2, \ldots, x_k = y)$ dans G_{Sym}.

Propriété

Etant donné un graphe $G = (S, A)$
$\forall x \in S, Desc_{G_{Sym}}(x) =$ Composante connexe contenant x dans G_{Sym}

Preuve : triviale en montrant la double inclusion !

Ces deux propriétés permettent de travailler sur G_{Sym} pour chercher les composantes connexes d'un graphe G

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

calculer $Desc_{G_{Sym}}(x)$ /* il s'agit d'une nouvelle composante connexe */

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

calculer $Desc_{G_{Sym}}(x)$ /* il s'agit d'une nouvelle composante connexe */

pour tout sommet y de $Desc_{G_{Sym}}(x)$ **faire** marquer y à vrai

fin tant que

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

calculer $Desc_{G_{Sym}}(x)$ /* il s'agit d'une nouvelle composante connexe */

pour tout sommet y de $Desc_{G_{Sym}}(x)$ **faire** marquer y à vrai

fin tant que

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n + m)$

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

calculer $Desc_{G_{Sym}}(x)$ /* il s'agit d'une nouvelle composante connexe */

pour tout sommet y de $Desc_{G_{Sym}}(x)$ **faire** marquer y à vrai

fin tant que

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

- Étape 1 : coût de la symétrisation du graphe G en $\Theta(n+m)$

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

calculer $Desc_{G_{Sym}}(x)$ /* il s'agit d'une nouvelle composante connexe */

pour tout sommet y de $Desc_{G_{Sym}}(x)$ **faire** marquer y à vrai

fin tant que

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

- Étape 1 : coût de la symétrisation du graphe G en $\Theta(n+m)$
- Étape 2 : linéaire en le nombre de sommets du graphe : $\Theta(n)$

Calcul des Composantes Connexe

Schéma d'algorithme découlant des propriétés

(1) calcul de G_{Sym}

(2) **pour** tout sommet x de G_{Sym} **faire** marquer x à faux **fin pour**

(3) **tant que** $\exists x \in S$ tel que le marquage de x est faux **faire**

calculer $Desc_{G_{Sym}}(x)$ /* il s'agit d'une nouvelle composante connexe */

pour tout sommet y de $Desc_{G_{Sym}}(x)$ **faire** marquer y à vrai

fin tant que

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

- Étape 1 : coût de la symétrisation du graphe G en $\Theta(n+m)$
- Étape 2 : linéaire en le nombre de sommets du graphe : $\Theta(n)$
- Étape 3 : en $\Theta(n+m)$ (CC étant l'ensemble des composantes connexes)

  n_k et m_k nombre de sommets et d'arcs dans la k^{eme} composante connexe calculée

  Le calcule du k^{eme} en semble $Desc_{G_{Sym}}(x)$ est en $\Theta(n_k + m_k)$

  Le coût global est donc $\Theta(\sum_{1 \leq k \leq |CC|}(n_k + m_k) = \Theta(n+m)$

Relation de Forte Connexité

Définition

La **relation de forte connexité** sur S pour un graphe $G = (S, A)$, notée R_{Fc} est définie par : $\forall x, y \in S$,

$$xR_{Fc}y$$
$$\Leftrightarrow$$
$$\exists \text{ des chemins } [x_1 = x, x_2, \ldots x_k = y] \text{ et } [x_k = y, x'_2, \ldots x_1 = x] \text{ dans } G$$

Relation de Forte Connexité

Définition

La **relation de forte connexité** sur S pour un graphe $G = (S, A)$, notée R_{Fc} est définie par : $\forall x, y \in S$,

$$xR_{Fc}y$$
$$\Leftrightarrow$$
$$\exists \text{ des chemins } [x_1 = x, x_2, \ldots x_k = y] \text{ et } [x_k = y, x_2', \ldots x_1 = x] \text{ dans } G$$

Propriété

R_{Fc} est une relation d'équivalence

Relation de Forte Connexité

Définition

La **relation de forte connexité** sur S pour un graphe $G = (S, A)$, notée R_{Fc} est définie par : $\forall x, y \in S$,

$$xR_{Fc}y$$
$$\Leftrightarrow$$
$$\exists \text{ des chemins } [x_1 = x, x_2, \dots x_k = y] \text{ et } [x_k = y, x_2', \dots x_1 = x] \text{ dans } G$$

Propriété

R_{Fc} est une relation d'équivalence

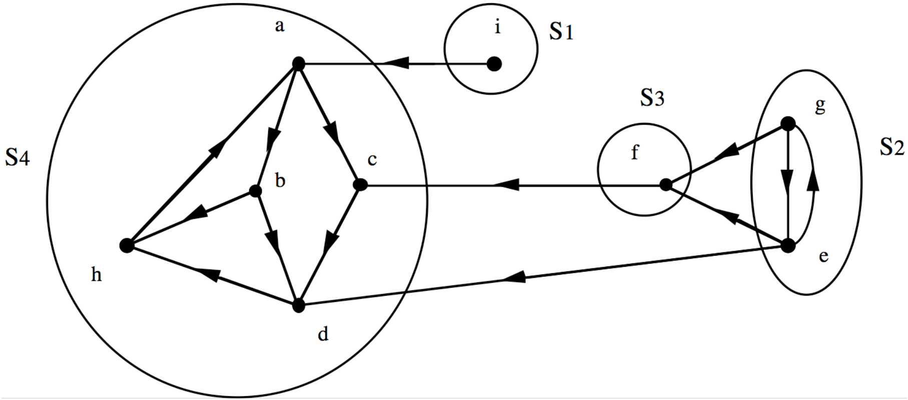
Définition

Soit $G = (S, A)$ un graphe. Une classe d'équivalence de S modulo R_{Fc} est appellée **composante fortement connexe** de G.

Un graphe est dit **fortement connexe** s'il n'a qu'une seule composante fortement connexe.

Composantes Fortement Connexes

Exemple : un graphe avec 4 composantes fortement connexes

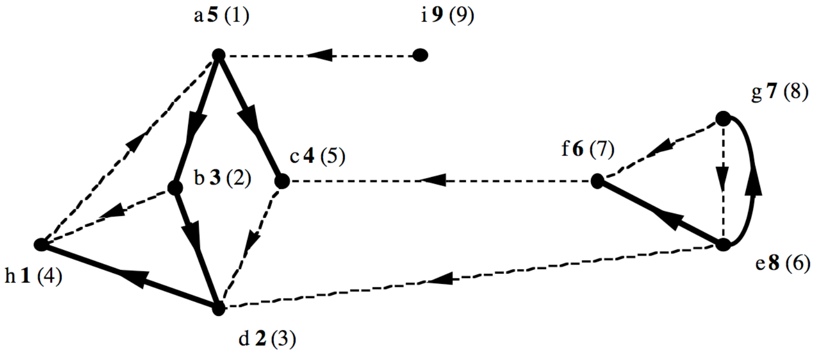


Calcul des Composantes Fortement Connexes

Sur un exemple, un algorithme linéaire "magique" !

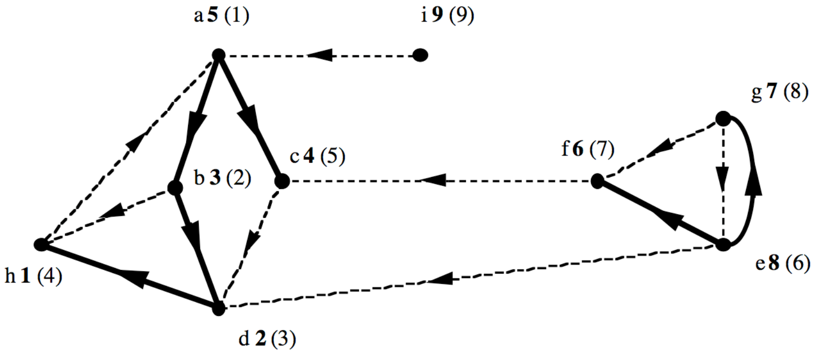
Calcul des Composantes Fortement Connexes

Sur un exemple, un algorithme linéaire "magique" !



Calcul des Composantes Fortement Connexes

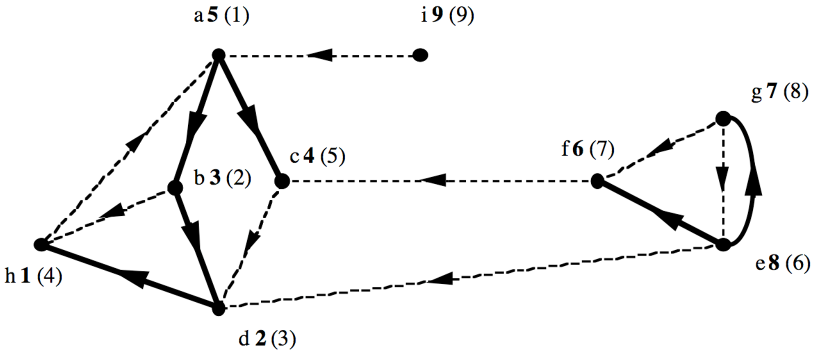
Sur un exemple, un algorithme linéaire "magique" !



(1) parcours de G en profondeur avec numérotation post-ordre (numéros en gras)

Calcul des Composantes Fortement Connexes

Sur un exemple, un algorithme linéaire "magique" !

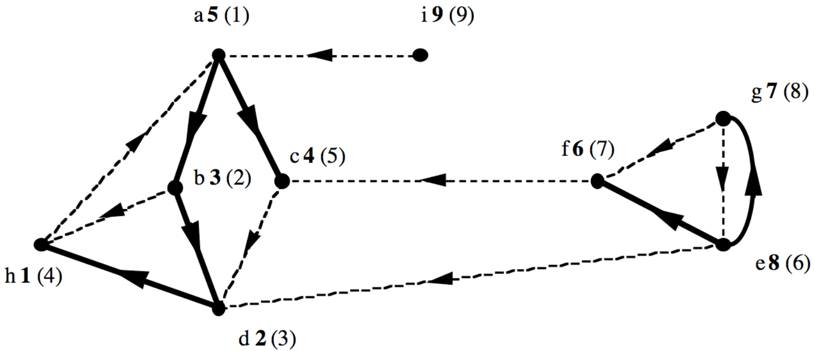


(1) parcours de G en profondeur avec numérotation post-ordre (numéros en gras)

(2) calcul du graphe réciproque de G que l'on note G^{-1}

Calcul des Composantes Fortement Connexes

Sur un exemple, un algorithme linéaire "magique" !



(1) parcours de G en profondeur avec numérotation post-ordre (numéros en gras)

(2) calcul du graphe réciproque de G que l'on note G^{-1}

(3) parcours de G^{-1} en profondeur avec
- apres chaque recherche, choix du sommet de numéro maximum non encore atteint
- chaque nouvelle recherche calcule une nouvelle Composantes Fortement Connexe de G

Calcul des Composantes Fortement Connexes

Un algorithme linéaire "magique" !

(1) parcours de G en profondeur avec numérotation post-ordre

(2) calcul du graphe réciproque de G que l'on note G^{-1}

(3) parcours de G^{-1} en profondeur avec
- apres chaque recherche, choix du sommet de numéro maximum non encore atteint
- chaque nouvelle recherche calcule une nouvelle Composantes Fortement Connexe de G

Calcul des Composantes Fortement Connexes

Un algorithme linéaire "magique" !

(1) parcours de G en profondeur avec numérotation post-ordre

(2) calcul du graphe réciproque de G que l'on note G^{-1}

(3) parcours de G^{-1} en profondeur avec
- apres chaque recherche, choix du sommet de numéro maximum non encore atteint
- chaque nouvelle recherche calcule une nouvelle Composantes Fortement Connexe de G

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

Calcul des Composantes Fortement Connexes

Un algorithme linéaire "magique" !

(1) parcours de G en profondeur avec numérotation post-ordre

(2) calcul du graphe réciproque de G que l'on note G^{-1}

(3) parcours de G^{-1} en profondeur avec
- apres chaque recherche, choix du sommet de numéro maximum non encore atteint
- chaque nouvelle recherche calcule une nouvelle Composantes Fortement Connexe de G

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

- Étape 1 : Parcours de G en profondeur avec numérotation post-ordre en $\Theta(n+m)$

Calcul des Composantes Fortement Connexes

Un algorithme linéaire "magique" !

(1) parcours de G en profondeur avec numérotation post-ordre

(2) calcul du graphe réciproque de G que l'on note G^{-1}

(3) parcours de G^{-1} en profondeur avec
- apres chaque recherche, choix du sommet de numéro maximum non encore atteint
- chaque nouvelle recherche calcule une nouvelle Composantes Fortement Connexe de G

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

- Étape 1 : Parcours de G en profondeur avec numérotation post-ordre en $\Theta(n+m)$
- Étape 2 : Calcul de G^{-1} en $\Theta(n+m)$

Calcul des Composantes Fortement Connexes

Un algorithme linéaire "magique" !

(1) parcours de G en profondeur avec numérotation post-ordre

(2) calcul du graphe réciproque de G que l'on note G^{-1}

(3) parcours de G^{-1} en profondeur avec
- apres chaque recherche, choix du sommet de numéro maximum non encore atteint
- chaque nouvelle recherche calcule une nouvelle Composantes Fortement Connexe de G

Complexité (pour les listes d'adjacence) : linéaire en en $\Theta(n+m)$

- Étape 1 : Parcours de G en profondeur avec numérotation post-ordre en $\Theta(n+m)$
- Étape 2 : Calcul de G^{-1} en $\Theta(n+m)$
- Étape 3 : Parcours de G^{-1} en profondeur en $\Theta(n+m)$