Complexité des algorithmes

Abdelkamel, Ben Ali

Université d'El Oued

Printemps 2021

Master 1 d'informatique – SDIA

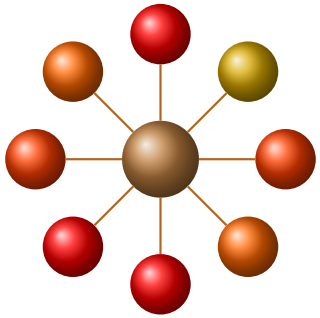
Plan du cours complet (1)

- **Chapitre 1 — Eléments de base**
  - Problème de décidabilité
  - Complexité et problèmes irrésolvables
  - Problèmes, algorithmes, complexité et schéma de codage
  - Problèmes polynomiaux et problèmes irrésolvables.
- **Chapitre 2 — Problèmes $\mathcal{NP}$-complets**
  - Machine de Turing déterministe
  - Problème SAT
  - Réduction polynomiale
  - Calcul non déterministe et la classe $\mathcal{NP}$
  - Classe des problèmes polynomiaux
  - Relation entre $\mathcal{P}$, $\mathcal{NP}$ et $\mathcal{NP}$-complet

Plan du cours complet (2)

- **Chapitre 3 — Techniques de démonstration de la $\mathcal{NP}$-complexité**
  - Les six problèmes "connus" comme étant $\mathcal{NP}$-complets
  - La méthode des restrictions
  - La méthode des remplacements locaux
  - Méthode de désignation de composantes
- **Chapitre 4 — Les autres classes de complexité**
  - $\mathcal{NP}$-difficiles
  - $\mathcal{NP}$-complets au sens fort
  - pseudo-polynomiales, ...
- **Chapitre 5 — Récurrence et complexité**

Extrait du cours introductif . . .



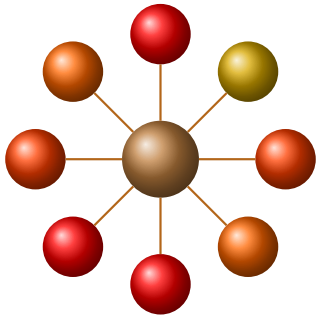
Existe-il un algorithme pour résoudre le probléme ?
Est-ce un problème classique ?

Comment concevoir un algorithme ? L'algorithme est-il correct ?

L'algorithme est-il efficace ?
Savoir analyser la Complexité des algorithmes

Peut-on trouver un algorithme plus efficace pour le probléme ?
Est-ce un probléme dur ? S'il est dur, comment l'appréhender ?

Extrait du cours introductif . . .



Existe-il un algorithme pour résoudre le probléme ?
Est-ce un problème classique ?

Comment concevoir un algorithme ? L'algorithme est-il correct ?

L'algorithme est-il efficace ?
Savoir analyser la Complexité des algorithmes

Peut-on trouver un algorithme plus efficace pour le probléme ?
Est-ce un probléme dur ? S'il est dur, comment l'appréhender ?

Première partie I

Éléments de base

Objectifs de cette partie du cours

L'objectif de cette partie "Complexité théorique et expérimentale" est :

- Présenter les notions et les principes liés à la complexité algorithmique.
- Donner une classification des problèmes du point de vue de leur complexité algorithmique.
- Etudier les outils mathématiques nécessaires à l'analyse des performances d'un algorithme.
- Montrer comment améliorer les performances des algorithmes faciles.

Plan de ce chapitre du cours

- Cette introduction
- Algorithmes et leur coût
- Indécidabilité
- Codage des instances et Modèles du calcul
- Principe de calcul
- Evaluations – Mesures de coût et Notations de Landau
- Exemples complets ...
- Ouverture vers le monde de la $\mathcal{NP}$-complétude

Motivation

- Un algorithme est une procédure finie et mécanique de résolution d'un problème.
  Exemples : les algorithmes d'Euclide, l'algorithme de Dijkstra ...
- Un algorithme doit se terminer sur toutes les données possibles du problème et doit fournir une solution correcte dans chaque cas.

Motivation

- Un algorithme est une procédure finie et mécanique de résolution d'un problème.
  Exemples : les algorithmes d'Euclide, l'algorithme de Dijkstra ...
- Un algorithme doit se terminer sur toutes les données possibles du problème et doit fournir une solution correcte dans chaque cas.
- Résoudre informatiquement un problème, c'est implanter un algorithme de résolution sur un ordinateur.
  Mais, il existe bien souvent plusieurs algorithmes pour le problème.
  Y a-t-il un intérêt à choisir ? et si oui comment choisir ?

Motivation

- Un algorithme est une procédure finie et mécanique de résolution d'un problème.
  Exemples : les algorithmes d'Euclide, l'algorithme de Dijkstra ...
- Un algorithme doit se terminer sur toutes les données possibles du problème et doit fournir une solution correcte dans chaque cas.
- Résoudre informatiquement un problème, c'est implanter un algorithme de résolution sur un ordinateur.
  Mais, il existe bien souvent plusieurs algorithmes pour le problème.
  Y a-t-il un intérêt à choisir ? et si oui comment choisir ?
- En pratique, il n'est même pas suffisant de détenir un algorithme.
  Il existe des problèmes pour lesquels on a des algorithmes, mais qui restent comme "informatiquement non résolus".
  - C'est parce que le temps d'exécution est vite exorbitant.
  - On cherche des (méta-)heuristiques pour abaisser ce temps de calcul.

Quelques problèmes

1. Savoir si un entier n est nombre premier ou non ? C'est un problème de décision : la réponse au problème est vrai ou faux. Il paraît simple de répondre à la question en essayant touts les diviseurs entre 2 et $\sqrt{n}$.

Quelques problèmes

1. Savoir si un entier n est nombre premier ou non ? C'est un problème de décision : la réponse au problème est vrai ou faux. Il paraît simple de répondre à la question en essayant touts les diviseurs entre 2 et $\sqrt{n}$.
2. L'existence d'un chemin entre deux sommets dans un graphe. C'est également un problème de décision. Il n'est pas difficile de trouver la réponse en visitant de proche en proche les sommets. Complexité est en $O(n+m)$.

Quelques problèmes

1. Savoir si un entier n est nombre premier ou non ? C'est un problème de décision : la réponse au problème est vrai ou faux. Il paraît simple de répondre à la question en essayant touts les diviseurs entre 2 et $\sqrt{n}$.
2. L'existence d'un chemin entre deux sommets dans un graphe. C'est également un problème de décision. Il n'est pas difficile de trouver la réponse en visitant de proche en proche les sommets. Complexité est en $O(n+m)$.
3. Un problème plus compliqué : déterminer un flot maximum dans un réseau. C'est un problème d'optimisation dont une résolution possible (Ford & Fulkerson) fait appel au problème précédent en saturant une série de chemins entre la source et le puits.

Quelques problèmes

4. La recherche d'un sous ensemble de sommets connectés deux à deux (une clique) de cardinalité maximale dans un graphe. C'est un problème difficile et on ne connait pas de solution satisfaisante, résolution autre que l'énumération de touts les $O(2^n)$ sous-graphes possibles.

Quelques problèmes

4. La recherche d'un sous ensemble de sommets connectés deux à deux (une clique) de cardinalité maximale dans un graphe. C'est un problème difficile et on ne connait pas de solution satisfaisante, résolution autre que l'énumération de touts les $O(2^n)$ sous-graphes possibles.
5. L'existence d'un pavage du plan par un ensemble de formes géométriques. Décider s'il est possible de paver entièrement le plan en n'utilisant que les formes fournies et sans avoir de recouvrements. Ce problème a été démontré indécidable.

Quelques définitions

- **Problème** : question comportant un ou plusieurs paramètres
  - Quel est le plus court chemin entre deux sommets donnés d'un graphe ?
- **Instance** : donnée du problème sur une valeur de ses paramètres.
  - $G = (S, A), x, y \in G$; quel est le plus court chemin entre x et y dans G ?

Quelques définitions

- **Problème** : question comportant un ou plusieurs paramètres
  - Quel est le plus court chemin entre deux sommets donnés d'un graphe ?
- **Instance** : donnée du problème sur une valeur de ses paramètres.
  - $G = (S, A), x, y \in G$; quel est le plus court chemin entre x et y dans G ?
- **Problème de décision** : la solution du problème $\in \{oui, non\}$.
  - Existe-t-il un chemin de longueur $\leq k$ donnée entre deux sommets d'un graphe ?
- **Problème de calcul** : calculer la solution d'un problème.
  - Calculer le plus court chemin entre deux sommets donnés d'un graphe.

Indécidabilité

- Établir qu'un problème est indécidable est plus fort que dire simplement qu'on ne sait pas le résoudre.
- Par exemple : peut-on imaginer un algorithme capable de détecter les boucles infinies dans un programme ?
  C'est le "Halting problem" ou problème de l'arrêt !

  Preuve classique, qui repose sur un argument diagonal ...

Complexité

En informatique le mot complexité recouvre deux réalités :

Complexité des algorithmes

C'est l'étude de l'efficacité comparée des algorithmes. On mesure le temps et aussi l'espace mémoire nécessaire à un algorithme :

- Complexité **temporelle**
- Complexité spatiale

Cela peut se faire de façon expérimentale ou formelle.

Complexité

En informatique le mot complexité recouvre deux réalités :

Complexité des algorithmes

C'est l'étude de l'efficacité comparée des algorithmes. On mesure le temps et aussi l'espace mémoire nécessaire à un algorithme :

- Complexité **temporelle**
- Complexité spatiale

Cela peut se faire de façon expérimentale ou formelle.

Complexité des problèmes

La complexité des algorithmes a abouti à une classification des problèmes en fonction des performances des meilleurs algorithmes connus qui les résolvent.

- La progression de ordinateurs ne change rien à cette classification. Elle a été conçue indépendamment des caractéristiques techniques.

Temps d'exécution d'un programme

- On implante un algorithme dans un langage de haut niveau.
  Le temps d'exécution du programme dépend :
  - des données du problème pour cette exécution
  - de la qualité du code engendré par le compilateur
  - de la nature et de la rapidité des instructions offertes par l'ordinateur
  - de l'efficacité de l'algorithme
  - de l'encodage des données
  - . . .et aussi de la qualité de la programmation !

Temps d'exécution d'un programme

- On implante un algorithme dans un langage de haut niveau.
  Le temps d'exécution du programme dépend :
  - des données du problème pour cette exécution
  - de la qualité du code engendré par le compilateur
  - de la nature et de la rapidité des instructions offertes par l'ordinateur
  - de l'efficacité de l'algorithme
  - de l'encodage des données
  - . . .et aussi de la qualité de la programmation !
- A priori, on ne peut pas mesurer le temps de calcul sur toutes les entrées possibles. Il faut trouver une autre méthode d'évaluation.
- L'idée est de s'affranchir des considérations subjectives (programmeur, matériel, . . .).
  - On cherche une grandeur n pour "quantifier" les entrées.
  - On calcule les performances uniquement en fonction de n.

Complexité algorithmique

$$\mathbf{T_{ALG}(n) = ?}$$

Définition

La fonction de complexité d'un algorithme fait correspondre pour une taille donnée le nombre -maximum- d'instructions qui lui est nécessaire pour résoudre une instance quelconque de cette taille.

Modèles du calcul

- Les machines de Turing (TM) ou les Random Access Machines (RAM) sont des machines abstraites.
  Elles servent d'"étalon" à la mesure des complexités en temps et en espace des fonctions dites calculables.

  Thèse de Church [1]-Türing [2] (indépendamment mais les 2 en 1936)
  Toute procédure effectivement calculable (algorithme) l'est par une machine de Turing.

1. A.Church (1903-1995) : mathématicien et logicien américain
2. A.Türing (1912-1954) : mathématicien britannique

Modèles du calcul

- Les machines de Turing (TM) ou les Random Access Machines (RAM) sont des machines abstraites.
  Elles servent d'"étalon" à la mesure des complexités en temps et en espace des fonctions dites calculables.

  Thèse de Church [1]-Türing [2] (indépendamment mais les 2 en 1936)
  Toute procédure effectivement calculable (algorithme) l'est par une machine de Turing.
- Notre optique ici n'est évidemment pas de présenter la théorie de la calculabilité mais juste d'en appréhender les principes les plus simples.
- Notre approche pragmatique sert à comparer des algorithmes résolvant un même problème afin d'estimer rigoureusement lesquels sont les meilleurs.

1. A.Church (1903-1995) : mathématicien et logicien américain
2. A.Türing (1912-1954) : mathématicien britannique

Modèles du calcul – MT

- une TM est une unité de contrôle (automate) munie d'une tête de lecture/écriture positionnée sur un ruban (mémoire) qui comporte un nombre infini dénombrable de cases élémentaires
- l'unité de contrôle a un ensemble fini d'états possibles
- à chaque instant, la machine évolue en fonction de son état et de l'information lue sur la case courante : la tête écrit sur cette case en effaçant le contenu précédent, se déplace d'une case à gauche, à droite ou bien ne se déplace pas.
- Temps d'exécution d'un algorithme = nombre de transitions nécessaires pour aboutir un état accepteur
- Espace mémoire requis = nombre de cases utilisées

Modèles du calcul – RAM

- Machines RAM ou Machines à registres (1963), c'est un modèle qui se rapproche des ordinateurs de nos jours. Une RAM est composée :
  - d'une unité de contrôle
  - de n registres à accès direct + un compteur d'instructions AC et un accumulateur ACC
  - d'opérations élémentaires : les mouvements en mémoire (READ, WRITE), les accès aux registres (LAOD, STORE), les opérations de base (ADD), les branchements (JUMP), etc.
    Un programme = Suite finie d'instructions codée sur la mémoire

Modèles du calcul – RAM

- Machines RAM ou Machines à registres (1963), c'est un modèle qui se rapproche des ordinateurs de nos jours. Une RAM est composée :
  - d'une unité de contrôle
  - de n registres à accès direct + un compteur d'instructions AC et un accumulateur ACC
  - d'opérations élémentaires : les mouvements en mémoire (READ, WRITE), les accès aux registres (LAOD, STORE), les opérations de base (ADD), les branchements (JUMP), etc.
    Un programme = Suite finie d'instructions codée sur la mémoire
- Deux modèles de coût pour la mesure en temps :
  - Modèle uniforme : chaque instruction est de coût unitaire
  - Modèle logarithmique : le coût dépond logarithmiquement de la taille des opérandes
- L'espace mémoire : nombre de registres utilisées et leurs longueurs, indépendamment de leurs entrées-sorties

Tailles des entrées

- En premier lieu, il faut évaluer la taille des données nécessaires à l'algorithme. On les appelle les entrées ou les instances.
- La taille n va dépendre du codage de ces entrées
  Exemple en binaire, il faut $\lfloor \log_2(n) \rfloor + 1$ bits pour coder l'entier n.

Tailles des entrées

- En premier lieu, il faut évaluer la taille des données nécessaires à l'algorithme. On les appelle les entrées ou les instances.
- La taille n va dépendre du codage de ces entrées
  Exemple en binaire, il faut $\lfloor \log_2(n) \rfloor + 1$ bits pour coder l'entier n.
- En pratique, on choisit comme taille la ou les dimensions les plus significatives.
- Exemples, selon que le problème est modélisé par :
  - des nombres : ces nombres
  - des polynômes : le degré, le nombre de coefficients non nuls
  - des matrices $m \times n$: $\max(m, n)$, $m.n$, $m + n$
  - des graphes : ordre, taille, produit des deux
  - des arbres : comme les graphes et la hauteur, l'arité
  - des listes, tableaux, fichiers : nombre de cases, d'éléments.
  - des mots : leur longueur

Codage des instances

- Le choix de la représentation des instances (codage) n'est pas anodin quant à l'efficacité d'un algorithme.
  Illustration. Problème de Reachability : Existe-t-il un chemin de x à y dans un graphe $G(V, E)$.
  On peut choisir de représenter un graphe par exemple par :
  - C1 : sa matrice d'adjacence
  - C2 : une liste d'adjacence

  Les impacts sur la complexité de l'algorithme de résolution portent sur :
  - le temps d'exécution : $O(n^2)$ pour C1 et $O(n+m)$ pour C2
  - la taille du codage (bits) : $O(n^2)$ pour C1 et $O(m \log n)$ pour C2

Codage des instances

- Le choix de la représentation des instances (codage) n'est pas anodin quant à l'efficacité d'un algorithme.
  Illustration. Problème de Reachability : Existe-t-il un chemin de x à y dans un graphe $G(V, E)$.
  On peut choisir de représenter un graphe par exemple par :
  - C1 : sa matrice d'adjacence
  - C2 : une liste d'adjacence

  Les impacts sur la complexité de l'algorithme de résolution portent sur :
  - le temps d'exécution : $O(n^2)$ pour C1 et $O(n+m)$ pour C2
  - la taille du codage (bits) : $O(n^2)$ pour C1 et $O(m \log n)$ pour C2
- Comme on mesure la complexité -temporelle- en fonction de la taille de l'instance (espace mémoire), il faut la préciser sur le meilleur (raisonnable ou naturel) codage des instances.

Opérations fondamentales

- Déterminer la complexité algorithmique c'est "compter" le nombre d'instructions (opérations) fondamentales qui seront effectuées ;
- C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme.
- Leur nombre intervient alors principalement dans l'étude de la complexité de l'algorithme.

Opérations fondamentales

- Déterminer la complexité algorithmique c'est "compter" le nombre d'instructions (opérations) fondamentales qui seront effectuées ;
- C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme.
- Leur nombre intervient alors principalement dans l'étude de la complexité de l'algorithme.
- Avec un peu d'habitude, on les repère :

Problème	Opérations fondamentales
Recherche d'un élément dans une liste, un tableau, un arbre ...	comparaisons
Tri d'une liste, d'un tableau, d'un fichier ...	comparaisons déplacements
Multiplication de polynômes, de matrices, de grands entiers	additions multiplications

Coût des opérations

- Dans un ordinateur (un vrai) comme dans une TM ou une RAM, toutes les opérations n'ont pas le même coût.

  Exemple. multiplication et addition

  C'est peu étonnant. La multiplication est bien une généralisation de l'addition. Pourquoi coûterait-elle pareil ?

  Seules les opérations élémentaires sont à égalité de coût (cout unitaire).

Coût des opérations

- Dans un ordinateur (un vrai) comme dans une TM ou une RAM, toutes les opérations n'ont pas le même coût.

  Exemple. multiplication et addition

  C'est peu étonnant. La multiplication est bien une généralisation de l'addition. Pourquoi coûterait-elle pareil ?

  Seules les opérations élémentaires sont à égalité de coût (cout unitaire).
- Pour simplifier, on peut faire l'hypothèse que toutes les opérations ont un coût uniforme (même si cela manque souvent de réalisme). Ce coût est alors constant car il ne dépend plus de rien.
  - Ainsi, deux expressions sont supposées équivalentes du point de vue du coût car proportionnelles.
    Exemples
    - $a \leftarrow 1$
    - $T[a] \leftarrow N[b]/\,W[z] \times sin(x)$
    - $(((a = 6)\ OU\ (n = B))\ ET\ (a = faux))$

Evaluation des coûts en séquentiel

- Dans un programme strictement séquentiel, les “boucles” sont disjointes ou emboîtées : il n’y a pas de récursivité.
- Les temps d’exécution s’additionnent :

Evaluation des coûts en séquentiel

- Dans un programme strictement séquentiel, les "boucles" sont disjointes ou emboîtées : il n'y a pas de récursivité.
- Les temps d'exécution s'additionnent :
- **Conditionnelle** `(si C alors J sinon K)`

$$T(n) = T_C(n) + \max\{T_J(n), T_K(n)\}$$

Evaluation des coûts en séquentiel

- Dans un programme strictement séquentiel, les "boucles" sont disjointes ou emboîtées : il n'y a pas de récursivité.
- Les temps d'exécution s'additionnent :
- **Conditionnelle** `(si C alors J sinon K)`

$$T(n) = T_C(n) + \max\{T_J(n), T_K(n)\}$$

- **Itération bornée** `(pour i de j à k faire B)`

$$T(n) = (k - j + 1) \cdot (T_{entete}(n) + T_B(n)) + T_{entete}(n)$$

entête est mis pour l'affectation de l'indice de boucle et le test de continuation.

Le cas des boucles imbriquées se déduit de cette formule.

Evaluation des coûts en séquentiel (2)

- Les temps d'exécution s'additionnent :
- **Itération non bornée**

(tant que C faire B)

$$T(n) = \#_{boucles} \cdot (T_C(n) + T_B(n)) + T_C(n)$$

(répéter B jusqu'à C)

$$T(n) = \#_{boucles} \cdot (T_B(n) + T_C(n))$$

Le nombre d'itérations $\#_{boucles}$ s'évalue inductivement.

Evaluation des coûts en séquentiel (2)

- Les temps d'exécution s'additionnent :
- **Itération non bornée**

(tant que C faire B)

$$T(n) = \#_{boucles} \cdot (T_C(n) + T_B(n)) + T_C(n)$$

(répéter B jusqu'à C)

$$T(n) = \#_{boucles} \cdot (T_B(n) + T_C(n))$$

Le nombre d'itérations $\#_{boucles}$ s'évalue inductivement.

- **Appel de procédures**
on peut ordonner les procédures de sorte que la ième ait sa complexité qui ne dépende que des procédures j, avec $j < i$.

Evaluation des coûts en séquentiel

Exercice interactif

Pour $i \leftarrow 1$ **à** n **Faire**
    instruction 1
    Si i est pair **Alors**
        instruction 2
    Sinon
        instruction 3
        instruction 4
        instruction 5
    FinSi
FinPour

Pour ce cas de figure, on pourrait compter $5 \times n$ instructions, mais d'autres informations sont disponibles sur la condition donc la complexité sera de $n + \frac{2+4}{2} n$.

Evaluation des coûts en séquentiel

Exercice accompagné

Appel : Algo1(n)
Données : un entier n
Résultat : un entier
Début
    i, j, p : **entiers**
    $p \leftarrow 1$
    pour $i \leftarrow 1$ **à** n **faire**
        si $i \ mod \ (2) = 0$ **alors**
            pour $j \leftarrow 1$ **à** i **faire**
                $p \leftarrow p + 1$
        sinon
            $p \leftarrow 2 \times p$
        finsi
    finpour
    retourner p
Fin

On considérera uniquement les opérations d'affectation.

1. Quel est le résultat de cet algorithme ?
2. Calculer son coût.

Evaluation des coûts en récursif

- On sait que les algorithmes du type "diviser pour régner" donnent lieu à des programmes récursifs.
- Comment évaluer leur coût ? Il faut trouver :
  - la relation de récurrence associée
  - la base de la récurrence, en examinant les cas d'arrêt de la récursion
  - et aussi . . . la solution de cette équation
- Techniques utilisées : résolution des équations de récurrence ou de partition ou par les séries génératrices – le sujet du cours 5.
- Exemple.
  - on va évaluer le temps $T(n)$ de la recherche dichotomique dans un tableau -trié- de n cases à

$$n \geq 1 : T(n) = 1 + \log_2(n)$$

Mesures du coût

- Le coût en temps d'un algorithme varie évidemment avec la taille n de la donnée x, mais peut aussi varier sur les différentes données de même taille n.

  Exemple. Recherche séquentielle

  - Si l'on compte le nombre de comparaison réalisées, il varie de 1 à n

Mesures du coût

- Le coût en temps d'un algorithme varie évidemment avec la taille n de la donnée x, mais peut aussi varier sur les différentes données de même taille n.

  Exemple. Recherche séquentielle
  - Si l'on compte le nombre de comparaison réalisées, il varie de 1 à n
- But : évaluer le coût d'un algorithme, selon certains critères – et en fonction de la taille n des données.
  En particulier :
  - Coût dans le cas le plus défavorable
  - Coût moyen
  - . . .coût dans le meilleur des cas

Mesures du coût — Coût dans le cas le pire

- Le coût d'un algorithme dans le cas le plus défavorable ou dans le cas le pire est par définition le maximum de coûts, sur touts les entrées de taille n :

$$C(n) = \max_{|x|=n} C(n)$$

(on note $|x|$ la taille de x)

- Exemple
  Le cas pire pour le tri par insertion est quand le tableau est trié dans l'ordre inverse, ce coût est : $n(n+1)/2$ (nombre d'insertion effectuées)

Mesures du coût — Coût moyen

- Si l'on pense que le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme.
- On suppose que l'on connaisse une distribution de probabilités sur les données de taille n.
  Si $p(x)$ est la probabilité de la donnée x, le coût moyen $\gamma(n)$ d'un algorithme sur les données de taille n est par définition :

$$\gamma(n) = \sum_{|x|=n} p(x)c(x)$$

Mesures du coût — Coût moyen

- Si l'on pense que le cas le plus défavorable ne se présente que rarement, on est plutôt intéressé par le coût moyen de l'algorithme.
- On suppose que l'on connaisse une distribution de probabilités sur les données de taille n.
  Si $p(x)$ est la probabilité de la donnée x, le coût moyen $\gamma(n)$ d'un algorithme sur les données de taille n est par définition :

$$\gamma(n) = \sum_{|x|=n} p(x)c(x)$$

- Le plus souvent, on suppose que la distribution est uniforme, c-à-d que $p(x) = 1/T(n)$, ou $T(n)$ est le nombre de données de taille n.

$$\gamma(n) = \frac{1}{T(n)} \sum_{|x|=n} c(x)$$

- Exemple. Le coût moyen de l'alg. de tri par insertion, est $n(n+1)/4$

Mesures du coût – Exemple

Tri à bulles

Algorithme : Tri à bulle

Pour $i \leftarrow 1$ **à** $(N-1)$ **Faire**
  Pour $k \leftarrow N$ **à** $i+1$ **pas** (-1) **Faire**
    Si $(T[k] < T[k-1])$ **Alors**
      $x \leftarrow T[k]$
      $T[k] \leftarrow T[k-1]$
      $T[k-1] \leftarrow x$
    FinSi

Exercice acompagné

Quelle est la complexité (moyenne et dans le pire des cas) de cet algorithme en terme de comparaisons ?

Estimation asymptotique — Notations de Landau

- Evaluer la complexité d'un algorithme : donner l'ordre de grandeur du nombre d'opérations qu'il effectue lorsque la taille du problème qu'il résout augmente.
- . . . comparer le taux d'accroissement de différentes fonctions qui mesurent les performances d'algorithmes
  - On parle ainsi d'algorithme linéaire, quadratique, logarithmique, etc.
- Moyen commode : Notations de Landau
  - Notation O : donne une majoration de l'ordre de grandeur
  - Notation Ω : donne une minoration de l'ordre de grandeur
  - Notation Θ : donne deux bornes sur l'ordre de grandeur

Notations de Landau — Notation O

- On dit que que f -positive- est asymptotiquement majorée (ou dominée) par g et on écrit

$$f \in O(g) \quad \text{ou} \quad f = O(g)$$

quand il existe une constante $k > 0$, telle que pour un n assez grand, on a :

$$f(n) \leq k \cdot g(n)$$

Exemple.

$$n \in O(n^2), \frac{\ln n}{n} \in O(1), x + 1 \in O(x)$$

Notations de Landau – Notation O

- la notation $f = O(g)$ est scabreuse car elle ne dénote pas une égalité mais plutôt l'appartenance de f à la classe des fonctions en $O(g)$.

  Exemple : $4n^2 + n = O(n^2)$ et $n^2 - 3 = O(n^2)$

  sans que l'on ait $4n^2 + n = n^2 - 3$ à partir d'un n assez grand.

- Exercice accompagné :

  Soit $P(x) = a_0 x^k + a_1 x^{k-1} + ... + a_k$.

  Montrez que $P(x) \in O(x^k)$ au voisinage de l'infini.

Notations de Landau — Notation Ω

- Symétriquement, on dit que que f est asymptotiquement minorée par g et on écrit

$$f \in \Omega(g) \quad \text{ou} \quad f = \Omega(g)$$

quand il existe une constante $k > 0$, telle que pour un n assez grand, on a :

$$f(n) \geq k \cdot g(n)$$

En d'autres termes, on a

$$f \in \Omega(g) \Longleftrightarrow g \in O(f)$$

Exemple. Le nombre de comparaisons de tout algorithme de tri de suites de longueur n est $\Omega(n \ln n)$.

Notations de Landau — Notation Θ

- On dit que f est du même ordre de grandeur que g et on écrit

$$f \in \Theta(g) \quad \text{ou} \quad f = \Theta(g)$$

quand il existe deux constantes $k_1, k_2 > 0$, telle que pour un n assez grand, on a :

$$k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$$

En d'autres termes, on a

$$\Theta(g) = O(g) \cap \Omega(g)$$

Exemple. Pour tout $a, b > 0$ on a

$$\log_a n = \Theta(\log_b n)$$

puisque $\log_a n = \log_a b \log_b n$.

Notations de Landau — Propriétés

- Réflexivité
  - $g = O(g)$
  - $g = \Theta(g)$
- Symétrie
  - $f = \Theta(g)$ ssi $g = \Theta(f)$
- Transitivité
  - $f = O(g)$ et $g = O(h)$ alors $f = O(h)$
  - $f = \Theta(g)$ et $g = \Theta(h)$ alors $f = \Theta(h)$
- Produit par un scalaire
  - $c > 0, \quad cO(g) = O(g)$
- Somme et produit de fonctions
  - $O(f) + O(g) = O(\max(f, g))$
  - $O(f)O(g) = O(fg)$

Ordres de grandeurs — Comparaison asymptotique

$$\log(\mathbf{n}) \ll \sqrt{\mathbf{n}} \ll \mathbf{n} \ll \mathbf{n}\log(\mathbf{n}) \ll \mathbf{n^2} \ll \mathbf{n^3} \ll \mathbf{2^n} \ll \exp(\mathbf{n}) \ll \mathbf{n!} \ll \mathbf{n^n} \ll \mathbf{2^{2^n}}$$

$\mathbf{O(1)}$	Constant
$\mathbf{O(\log(n))}$	Logarithmique
$\mathbf{O(n)}$	Linéaire
$\mathbf{n\log(n)}$	$n\log(n)$
$\mathbf{O(n^2)}$	Quadratique
$\mathbf{O(n^3)}$	Cubique
$\mathbf{O(2^n)}$	Exponentiel

Algorithmes polynomiaux $\leftrightarrow$ exponentiels

- Complexité polynomiale $\Rightarrow$ souvent réalisable
  $\exists k > 0, \quad f(n) \in O(n^k)$.
- Complexité exponentielle $\Rightarrow$ en général irréalisable
  $\exists b > 1, \quad b^n \in O(f(n))$.
- Complexité doublement exponentielle
  par exemple : $f(n) = 2^{2^n}$.
- Complexité sous-exponentielle
  par exemple : $f(n) = 2^{\sqrt{n}}$.

$$\log(n) \ll \sqrt{n} \ll n \ll n\log(n) \ll n^2 \ll n^3 \ll 2^n \ll \exp(n) \ll n! \ll n^n \ll 2^{2^n}$$

Ordres de grandeurs — Remarque

	2	16	64	256
$\log\log \mathbf{n}$	0	2	2.58	3
$\log \mathbf{n}$	1	4	6	8
$\mathbf{n}$	2	16	64	256
$\mathbf{n}\log \mathbf{n}$	2	64	384	2048
$\mathbf{n^2}$	4	256	4096	65536
$\mathbf{2^n}$	4	65536	1.84467e+19	1.15792e+77
$\mathbf{n!}$	2	2.0923e+13	1.26887e+89	8.57539e+506
$\mathbf{n^n}$	4	1.84467e+19	3.9402e+115	3.2317e+616
$\mathbf{2^{n^2}}$	16	1.15792e+77	1.04438e+1233	2.00353e+19728

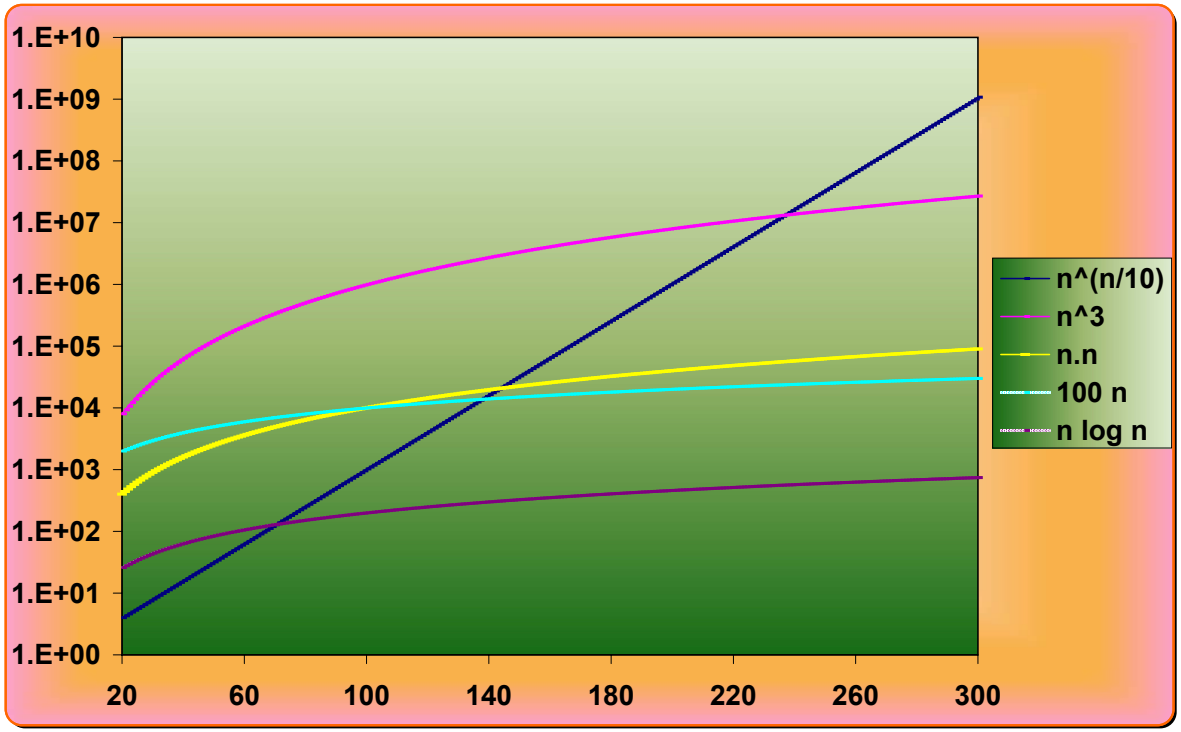
Méga $= 10^6(2^{20})$, Géga $= 10^9(2^{30})$, Tera $= 10^{12}(2^{40})$, Péta $= 10^{15}(2^{50})$

4 Ghz pendant 1 an $= 1,26 \times 10^{17}$

4 Ghz pendant 4 Milliards d'années $= 5 \times 10^{26}$

Ordres de grandeurs — Commentaires

- En pratique, un algorithme à complexité exponentielle est inutilisable.
- Pour n pas trop grand, les algorithmes polynomiaux sont encore efficaces.



Ouverture vers le monde de la $\mathcal{NP}$-complétude

- La distinction entre algorithmes polynomiaux et algorithmes exponentiels est essentiel à la notion de $\mathcal{NP}$-complétude.
- En effet, la manière dont les problèmes sont classés est directement liée à la possibilité de les résoudre à l'aide d'algorithmes de complexité polynomiale ou exponentielle.
- On distinguera ainsi les problèmes pour lesquels il existe des algorithmes efficaces permettant de les résoudre et des problèmes pour lesquels il est peu probable que de tels algorithmes existent.

Exemples complets — Un peu de mots

Voici le temps de calcul d'algorithmes récursifs classiques

- Recherche dichotomique dans un tableau de n cases

$$T(n) = 1 + \log_2(n)$$
$$\text{soit} \quad T(n) = O(\log_2(n))$$

- Problème des tours de Hanoï [montrer]

$$T(n) = 2^n - 1$$
$$\text{soit} \quad T(n) = O(2^n)$$

- Factorielle : temps linéaire.
- Evaluation de polynôme (Horner) : linéaire sur le degré.
- Multiplication de matrices : temps cubique.
- Tri rapide (quicksort) : $n \log n$. [c.f. cours #3]

Exemples complets – Graphe régulier

Soit le graphe $G = (S, A)$, on veut vérifier que ce graphe est régulier, c'est-à-dire que tous les sommets ont le même degré.

Soit S l'ensemble des sommets
régulier $\leftarrow$ vrai
$s \leftarrow$ retirer_sommet(S) /* s un sommet quelconque du graphe */
d $\leftarrow$ degré(s)
TantQue $S \neq \phi$ ET régulier **Faire**
    $s \leftarrow$ retirer_sommet(S)
    Si degré(s) $\neq d$ **Alors**
        régulier $\leftarrow$ faux
    FinSi
FinTantQue
Si régulier **Alors**
    Le graphe est régulier
Sinon
    Le graphe n'est pas régulier
FinSi

Graphe régulier : complexité

- 3 instructions + complexité de la boucle **TantQue** + complexité de la conditionnelle
  $\Rightarrow$ 3 + C(boucle) + 1 + 1 = 5 + C(boucle).

  Analyse de la boucle :

  TantQue $S \neq \phi$ ET régulier **Faire**
  $s \leftarrow$ retirer_sommet(S)
  Si degré(s) $\neq d$ **Alors**
  régulier $\leftarrow$ faux
  FinSi
  FinTantQue

Graphe régulier : complexité

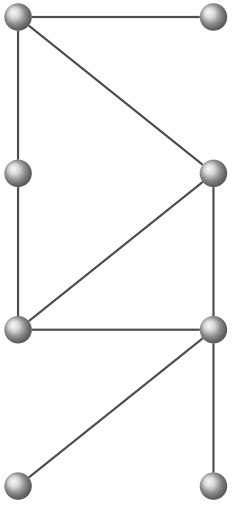
- Chaque sommet n'est examiné qu'une seule fois, donc la boucle sera parcourue, dans le pire des cas que $n-1$ fois (un premier sommet a été retiré au départ).
- Donc, la condition ($S \neq \phi$ ET régulier) est évaluée n fois
  $\Rightarrow n$ évaluations de conditions.
- Dans le pire des cas (en termes de complexité), le graphe est régulier pour les $n-2$ premiers sommets examinés et le dernier sommet présente un degré différent.
  $\Rightarrow 2 \times (n-2)$ pour le retrait des sommets de S et pour l'évaluation de la condition *degré*$(s) \neq d$.
- Enfin, pour le dernier sommet, il faut rajouter une instruction $\Rightarrow 3$.
  $\Rightarrow$ au final $C(algo) = 5 + n + 2(n-2) + 3 = 4 + 3n = O(n)$

Exemples complets — Connexité

- **Intitulé du problème de décision** : Détermination de la connexité d'un graphe.
- **Description des paramètres** : un graphe $G = (S, A)$, $(|S| = n, |A| = m)$.
- **Question** : étant donné un sommet $s \in S$, alors, $\forall s' \in S, s \neq s'$, existe-t-il un chemin de s à s' ?

Connexité : principe de l'algorithme

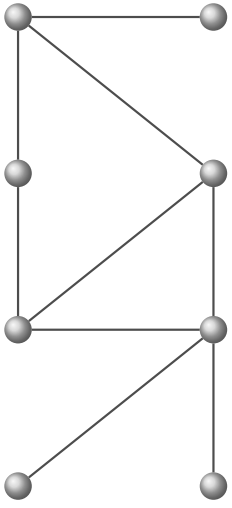
Sommet non atteint Sommet atteint et non traité Sommet traité



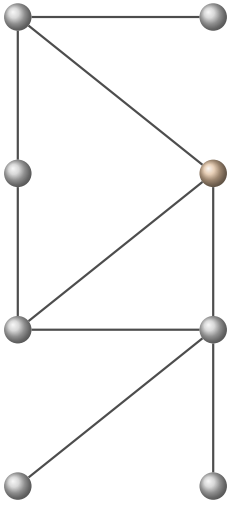
connexe ?

Connexité : principe de l'algorithme

Sommet non atteint Sommet atteint et non traité Sommet traité



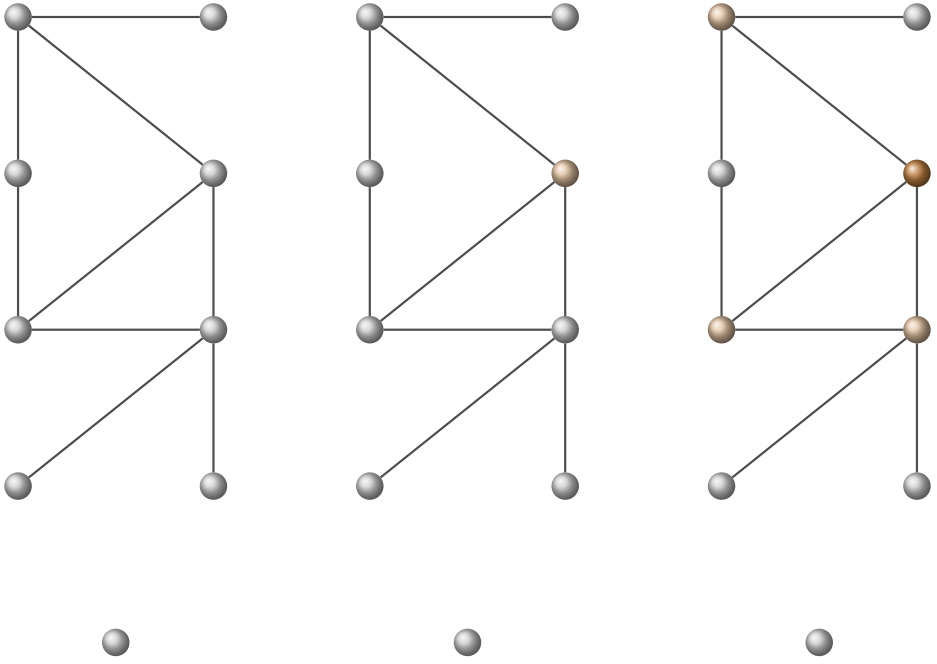
connexe ?



étape 1

Connexité : principe de l'algorithme

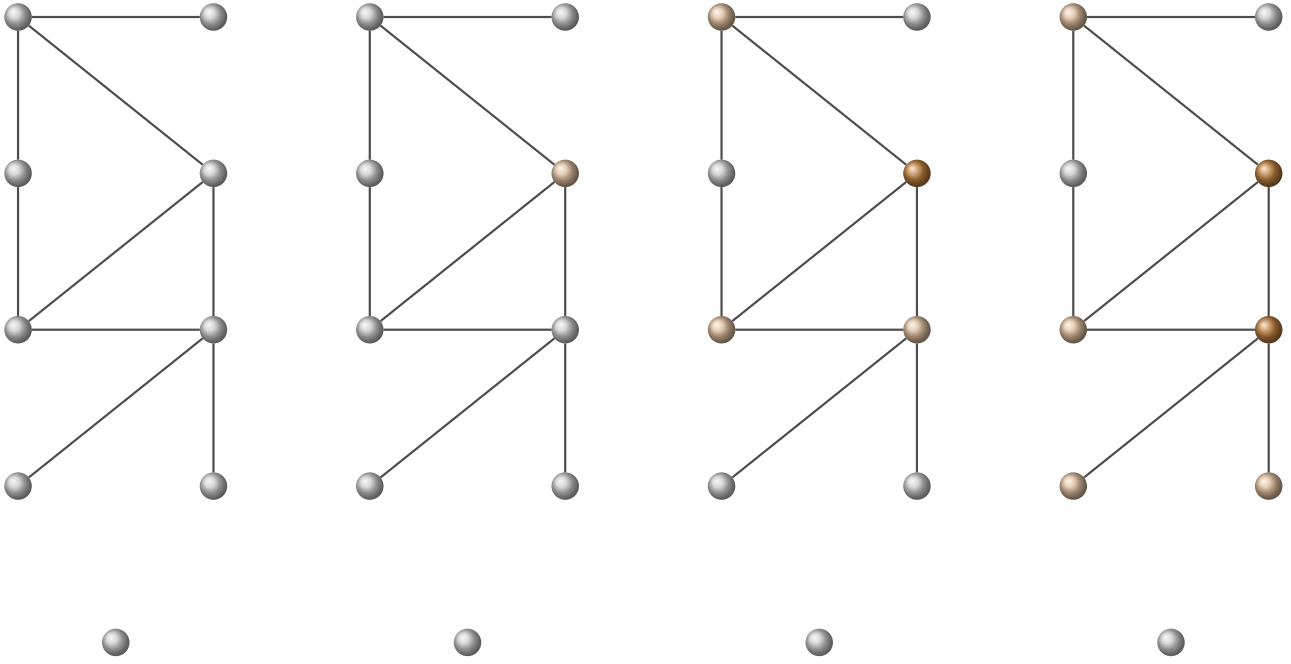
Sommet non atteint Sommet atteint et non traité Sommet traité



connexe ? étape 1 étape 2

Connexité : principe de l'algorithme

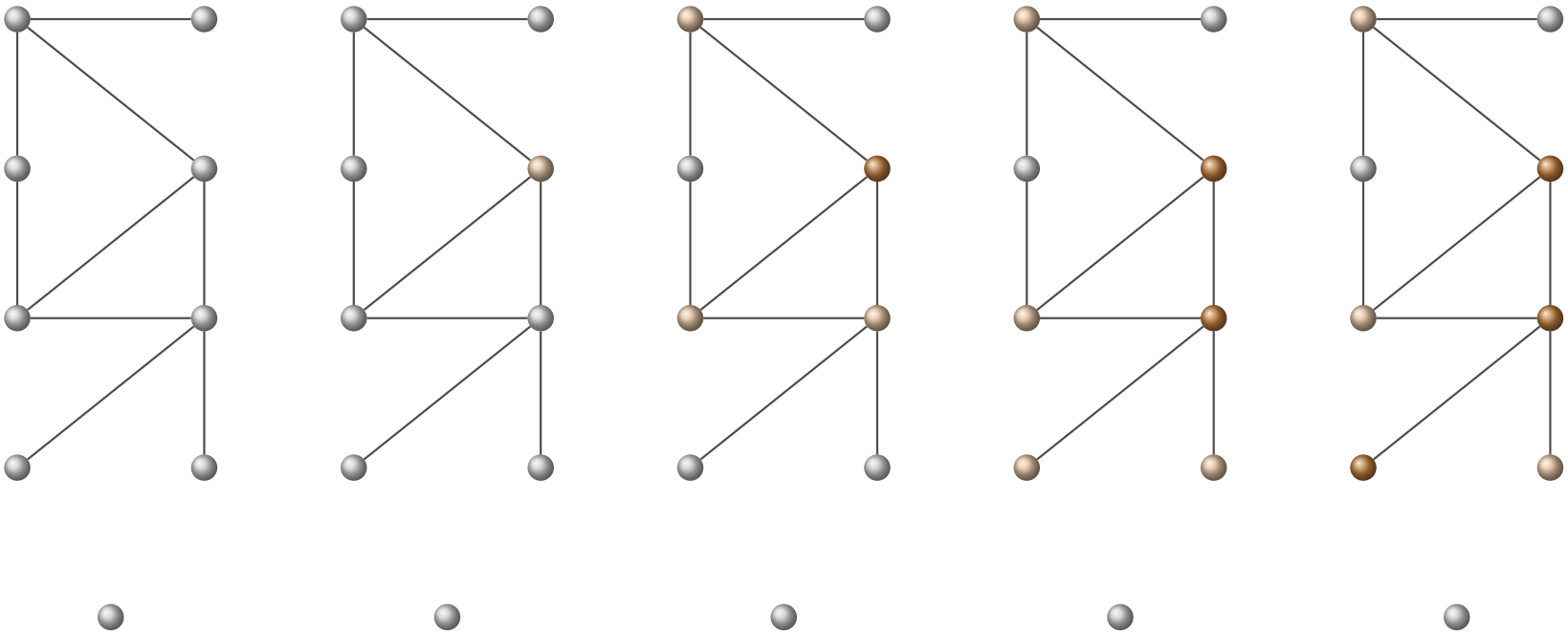
Sommet non atteint Sommet atteint et non traité Sommet traité



connexe ? étape 1 étape 2 étape 3

Connexité : principe de l'algorithme

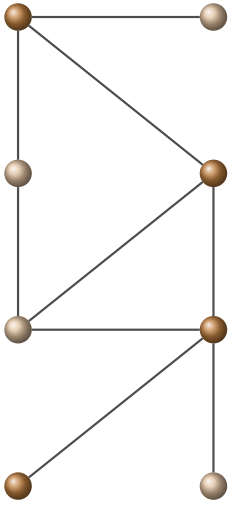
Sommet non atteint Sommet atteint et non traité Sommet traité



connexe ? étape 1 étape 2 étape 3 étape 4

Connexité : principe de l'algorithme

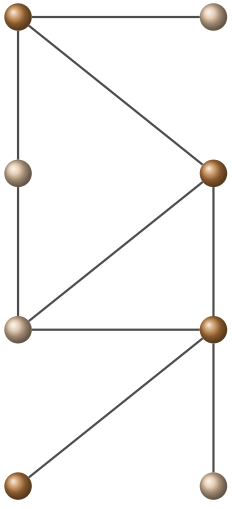
Sommet non atteint Sommet atteint et non traité Sommet traité



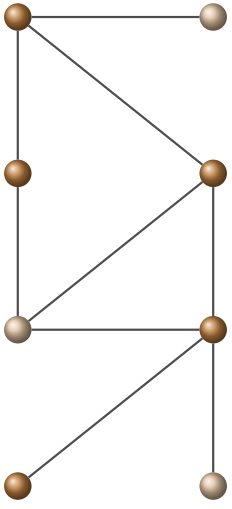
étape 5

Connexité : principe de l'algorithme

Sommet non atteint Sommet atteint et non traité Sommet traité



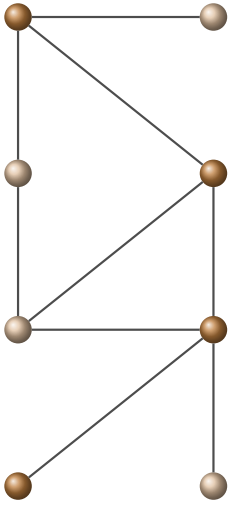
étape 5



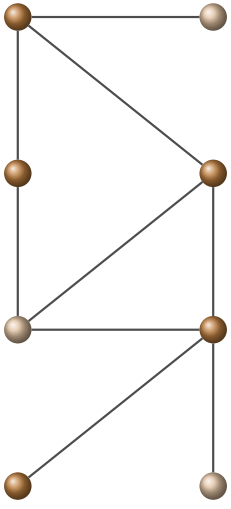
étape 6

Connexité : principe de l'algorithme

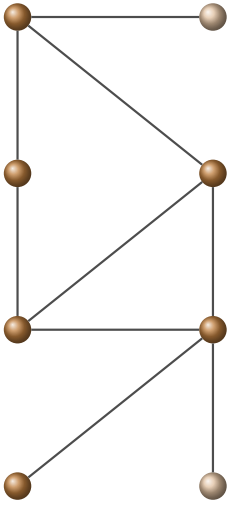
Sommet non atteint Sommet atteint et non traité Sommet traité



étape 5



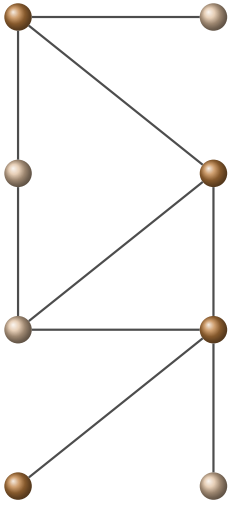
étape 6



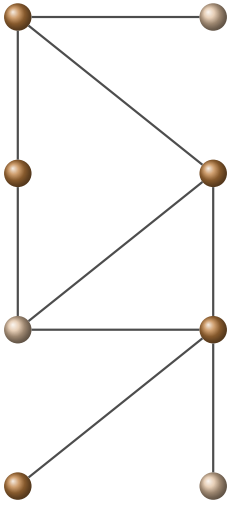
étape 7

Connexité : principe de l'algorithme

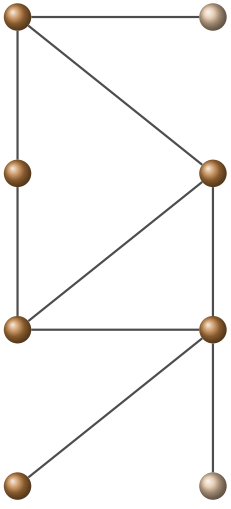
Sommet non atteint Sommet atteint et non traité Sommet traité



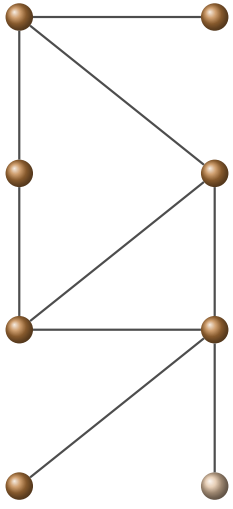
étape 5



étape 6



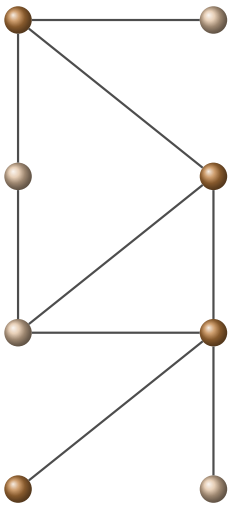
étape 7



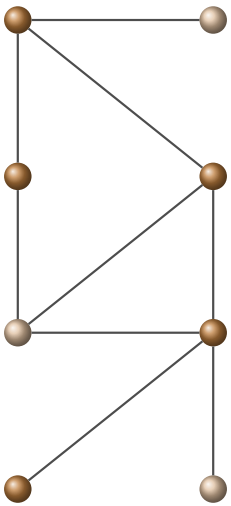
étape 8

Connexité : principe de l'algorithme

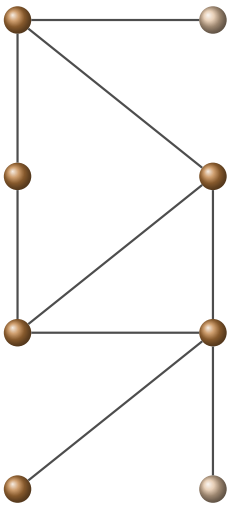
Sommet non atteint Sommet atteint et non traité Sommet traité



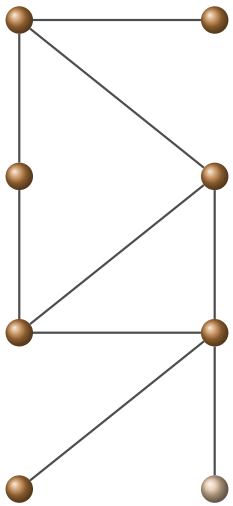
étape 5



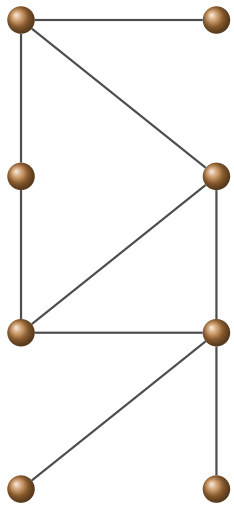
étape 6



étape 7



étape 8



non connexe

Connexité : algorithme

Soit L ensemble des sommets déjà atteints mais non traités (non visités)
Soit V ensemble des sommets visités
$L \leftarrow s$ /* s un sommet quelconque du graphe */
V $\leftarrow \phi$
TantQue $L \neq \phi$ **Faire**
    $s \leftarrow$ tirage arbitraire d'un élément de L
    $V \leftarrow V \cup \{s\}$
    $N \leftarrow$ voisins(s)
    Pour chaque élément e de N **Faire**
        Si $e \notin V$ ET $e \notin L$ **Alors**
            $L \leftarrow L \cup \{e\}$
        FinSi
    FinPour
FinTantQue
Si $|V| = |S|$ **Alors**
    Le graphe est connexe
Sinon
    Le graphe n'est pas connexe
FinSi

Connexité : analyse de la complexité

- En dehors de structures itératives : 3 instructions + évaluation d'une condition.
- C(algo) = 4 + C(TantQue)

TantQue $L \neq \phi$ **Faire**
    $s \leftarrow$ tirage arbitraire d'un élément de L
    $V \leftarrow V \cup \{s\}$
    $N \leftarrow$ voisins(s)
    Pour chaque élément e de N **Faire**
        Si $e \notin V$ ET $e \notin L$ **Alors**
            $L \leftarrow L \cup \{e\}$
        FinSi
    FinPour
FinTantQue

Connexité : analyse de la complexité

- Difficulté du comptage du nombre d'instructions.
- Une boucle **TantQue** se termine lorsque la condition n'est plus vérifiée.
- La condition d'arrêt est $L \neq \phi$, or à chaque tour de boucle, cette liste est modifiée à deux endroits :
  - $s \leftarrow$ tirage arbitraire d'un élément de L. Un élément est systématiquement retiré de la liste L et ajouté à une autre liste (V).
  - $L \leftarrow L \cup \{e\}$: on ajoute un élément à L ! Mais cet élément est ajouté seulement s'il n'est pas déjà présent dans V.
    $\Rightarrow$ chaque sommet passe dans L une et une seule fois.

Connexité : analyse de la complexité

- Ainsi, dans le pire des cas, la boucle **TantQue** sera parcourue n fois.
- A chaque tour de boucle **TantQue**, 3 instructions seront exécutées $\rightarrow 3 \times n$.
- Quoi de la boucle **Pour** ?
- Si le graphe est k-régulier, le nombre total de tour de boucle **Pour** sera de $n \times k$.
- Si le degré maximum est borné par une valeur indépendante de $n \Rightarrow$ complexité
  $C(algo) = 4 + 3n + kn = 3 + (3 + k)n = O(n)$

Connexité : analyse de la complexité

- Mais dans le cas général, le graphe n'est pas régulier. Donc, il faut identifier ce que représente le $n \times k$.
- Cette valeur représente la somme du nombre de voisins de tous les sommets
  $\Rightarrow$ la somme des degrés des sommets
  $\Rightarrow$ le double du nombre d'arêtes : $2 \times m$
- Ainsi, la complexité algorithmique de cet algorithme est de :
  $C(algo) = 4 + 3n + 2m = 4 + n + 2(n + m) = O(n + m)$

Exemples complets – Énumeration de mots

Soit un ensemble de symboles S de cardinalité k, on souhaite énumérer l'ensemble des mots de longueur L formés à partir des symboles de S.

Énumeration de mots : algorithme

programme principal
T : tableau de longueur L qui contient le mot courant
S : liste de k symboles
position $\leftarrow$ 1
EnumerationDesMots(position)

procédure EnumerationDesMots(position)
Début
  Si position $\leq L$ **Alors**
    symbole $\leftarrow$ S.premierSymbole()
    TantQue il reste des symboles non considérés pour cette position **Faire**
      T[position] $\leftarrow$ symbole
      EnumerationDesMots(position $+1$) /*appel récursif */
      symbole $\leftarrow$ S.suivant()
    FinTantQue
  Sinon
    afficher le tableau T
  FinSi
Fin

Énumeration de mots : complexité

Exercice interactif

Montrez que la complexité de l'algorithme est en $O(k^L)$ [DEMO]

Exemples complet — Fusion de listes

1. **Fonction** fusion(L, K : liste) : liste
2. **Si** L vide **Alors**
3. résultat $\leftarrow K$
4. **Sinon** si K vide **Alors**
5. résultat $\leftarrow L$
6. **Sinon Si** elmCourant(L) $<$ elmCourant(K) **Alors**
7. résultat $\leftarrow$ concatène(elmCourant(L), fusion(suivant(L), K))
8. **Sinon**
9. résultat $\leftarrow$ concatène(elmCourant(K), fusion(L, suivant(K))

Exemples complet — Fusion de listes

1.	**Fonction** fusion(L, K : liste) : liste	$\rightarrow O(1)$
2.	**Si** L vide **Alors**	$\rightarrow O(1)$
3.	résultat $\leftarrow K$	$\rightarrow O(1)$
4.	**Sinon** si K vide **Alors**	$\rightarrow O(1)$
5.	résultat $\leftarrow L$	$\rightarrow O(1)$
6.	**Sinon Si** elmCourant(L) $<$ elmCourant(K) **Alors**	$\rightarrow O(1)$
7.	résultat $\leftarrow$ concatène(elmCourant(L), fusion(suivant(L), K))	$\rightarrow O(1)+O(T(n-1))$
8.	**Sinon**	
9.	résultat $\leftarrow$ concatène(elmCourant(K), fusion(L, suivant(K))	$\rightarrow O(1)+O(T(n-1))$

Exemples complet — Fusion de listes

1.	**Fonction** fusion(L, K : liste) : liste	$\rightarrow O(1)$
2.	**Si** L vide **Alors**	$\rightarrow O(1)$
3.	résultat $\leftarrow K$	$\rightarrow O(1)$
4.	**Sinon** si K vide **Alors**	$\rightarrow O(1)$
5.	résultat $\leftarrow L$	$\rightarrow O(1)$
6.	**Sinon Si** elmCourant(L) $<$ elmCourant(K) **Alors**	$\rightarrow O(1)$
7.	résultat $\leftarrow$ concatène(elmCourant(L), fusion(suivant(L), K))	$\rightarrow O(1)+O(T(n-1))$
8.	**Sinon**	
9.	résultat $\leftarrow$ concatène(elmCourant(K), fusion(L, suivant(K))	$\rightarrow O(1)+O(T(n-1))$

- Si $n = 0$, on exécute les lignes 1, 2, 3 : temps $O(1)$
- Si $n = 1$, on exécute les lignes 1, 2, 4, 5 : temps $O(1)$
- Si $n > 1$, on exécute 1, 2, 4, 6, 7 ou 1, 2, 4, 6, 8, 9 : temps $O(1) + T(n-1)$
- Soit à résoudre l'équation $T(0) = a$ et $T(n) = b + T(n-1)$.
  $\Longrightarrow$ ... la résolution donne un coût linéaire : $T(n) = n.b + a = O(n)$

Exemples complet — Division de listes

1. **Fonction** split(L : liste) : un couple de listes $\rightarrow O(1)$
2. $J, K \leftarrow$ listeVide $\rightarrow O(1)$
3. **Si** vide(L) **Alors** $\rightarrow O(1)$
4. résultat $\leftarrow (L, L)$ $\rightarrow O(1)$
5. **Sinon Si** vide(suivant(L)) **Alors** $\rightarrow O(1)$
6. résultat $\leftarrow$ (L, listeVide) $\rightarrow O(1)$
7. **Sinon**
8. $i \leftarrow 1$ $\rightarrow O(1)$
9. **Tant que** non listeVide(L) **Faire** $\rightarrow O(1)$
10. $e =$ retire(L) $\rightarrow O(1)$
11. **Si** impair(i) **Alors** $\rightarrow O(1)$
12. ajoute (e, J) $\rightarrow O(1)$
13. **sinon**
14. ajoute (e, K) $\rightarrow O(1)$
15. **FinSi**
16. $i \leftarrow i + 1$ $\rightarrow O(1)$
17. **FinTantQue**
18. résultat $\leftarrow (J, K)$ $\rightarrow O(1)$
19. **FinSi**

- Si $n = 0$ ou 1 : temps $O(1)$
- Si $n > 1$, on exécute la boucle **tant que** n fois : temps $O(n)$
  $\Longrightarrow$ la fonction `split` a donc un coût linéaire.

Ce qu'il faut retenir de ce cours

- La complexité mesure l'efficacité d'un algorithme :
  - indépendante de l'implémentation,
  - permet de prédire le temps que va mettre un calcul.
- Il existe beaucoup d'approches pour résoudre un même problème :
  - trouver la meilleure approche peut être compliqué.
  - la complexité n'est pas le seul critère
    $\Rightarrow$ les contraintes d'implémentation comptent aussi.
- Il existe une théorie de la complexité :
  - certains problèmes sont durs
    $\Rightarrow$ il peut ne pas exister d'algorithme efficace pour les résoudre,
  - certains problèmes sont faciles
    $\Rightarrow$ connaît-on le meilleur algorithme ?