Chapitre 1

THÉORIE DE LA COMPLEXITÉ ET MESURE DE PERFORMANCE

PARTIE 1. COMPLEXITÉ D'UN ALGORITHME

La théorie de la complexité à pour but de classifier des problèmes en fonction de la complexité du meilleur algorithme pour les résoudre.

- Certains problèmes sont durs**.**
  - Il peut ne pas exister d'algorithme efficace pour les réoudre,
- Certains problèmes sont faciles.
  - Connaît-on le meilleur algorithme?

Le domaine de l'analyse d'algorithmique permet de présenter un ensemble de concepts de bases pour mesurer la complexité et les performances des algorithmes.

1. Généralités

L'objectif de ce chapitre est de présenter les grands principes de la complexité algorithmique et les qualités peut-on demander à un algorithme ou à un programme. Ceci nous amène à donner les outils permettant de répondre aux questions suivantes :

- Le programme s'arrête-t-il de l'exécution?
- Est-ce qu'il réalise effectivement la tâche qu'on lui a demandé? C'est à dire, est-il correct?
- Il faut qu'il soit bien écrit, compréhensible en vue d'une maintenance ou d'une amélioration ce qui nous conduit à présenter le critère d'évaluation d'efficacité entre les algorithmes suivant :
- L'algorithme doit être performant, pour cela on distingue deux critères essentiels :
  - Le temps de calcul nécessaire, directement lié au nombre d'opérations élémentaires qu'effectué l'algorithme.
  - La quantité d'occupation mémoire nécessaire.

Le premier principe s'appelle la terminaison de l'algorithme, le deuxième sa correction et le dernier sa complexité**.** Nous donnons ces principes comme des critères de qualités d'un bon algorithme.

Pour décrire un algorithme, on lui donne en général un nom, on précise quels sont les paramètres qu'il peut prendre en entrée et le résultat qu'il est sensé renvoyer. On Utilise des spécification de l'algorithme (des modifications de la mémoire).

Si ces opérations s'exécutent en séquence, on parle d'algorithme séquentiel. Si les opérations s'exécutent sur plusieurs processeurs en parallèle, on parle d'algorithme parallèle. Un algorithme est donc un moyen de calculer une solution au problème posé, le calcul étant décomposé en un nombre fini d'instructions et devant être exécutable en un temps fini, quelle que soit la donnée en entrée.

On décrit un algorithme souvent à l'aide d'un pseudo-langage naturel très simplifié, à la fois lisible et formel. Les données, résultats, et variables intermédiaires sont clairement déclarés, chaque calcul est précisé de façon complète.

Définition 1. Un invariant de boucle est une propriété dépendant des variables de l'algorithme, qui est vérifiée à chaque passage dans la boucle.

En fin, l'algorithme est traduit sous forme d'un langage de programmation adapté. Il est clair que les ordinateurs ne savent faire que des opérations extrêmement élémentaires : accès mémoire et arithmétique essentiellement. C'est donc le travail de l'informaticien d'écrire une solution à un problème complexe sous forme d'une suite d'instructions élémentaires.

Terminaison

Pour montrer qu'un algorithme termine quel que soit le paramètre passé en entrée respectant la spécification, il faut montrer que chaque bloc élémentaire décrit ci-dessus termine, les boucles pour (for) et les instructions conditionnelles terminent forcément. Le seul souci pourrait venir d'une boucle tantque (while).

Un algorithme comporte :

- Une partie temporelle : séquence d'instructions en principe savamment organisée,
- Une partie spatiale : ensemble de données plus ou moins structurées (nombres, vecteurs, matrices, listes...).

Problème : donnée → *l'algorithme fournir* (nombre fini d'étapes)→ réponse
programme doit finir par s'arrêter

Formellement

Un problème abstrait = l'ensemble < E, A, S > où

E est une entrée (un des cas du problème),
A l'algorithme,
S la sortie (résultat).

Un algorithme = < C entraînent → I >,
C : clauses (règles)

I : instructions (traitements).
Ces règles forment une partition de l'univers**:**
- **U** des règles = tous les cas possibles.
- ∩ =∅

→ *L'algorithme ne doit pas oublier un cas, et on ne peut pas être dans deux cas à la fois.*

Exemple 1 : trouver le maximum d'un tableau T

A. Façon abstraite d'un algorithme

Une fois qu'on a compris ce qu'on vient de faire, on peut l'abstraire :
- On considère le premier élément du tableau comme maximum,
- Si on trouve un élément plus grand que lui, c'est notre nouveau maximum
- Une fois que nous avons tout parcouru, nous renvoyons notre dernier maximum : rien trouvé de plus grand

B. Les règles trouvées

- Max(T, n, i, max) = Max(T, |T|, 2, T[1]))}
- i ≤ n ^ T[i] > max → Max(T, n, i, max) = Max(T, n, i+1, T[i])
- Si je n'ai pas parcouru tout le tableau et que l'élément actuel est supérieur au max, c'est le nouveau max.
- i ≤ n ^ T[i] ≤ max → Max(T, n, i, max) = Max(T, n, i+1, max)
- Si mon élément actuel n'est pas supérieur au max, alors je continue simplement de parcourir.
- i > n → Max(T, n, i, max) = max Si j'ai parcouru tout le tableau, je renvoie le max.

C. Ces règles forment une partition de l'univers

- L'union de ces règles, donne tous les cas possible :

(i > n) ∪ (i ≤ n ^ T[i] > max) ∪ (i ≤ n ^ T[i] ≤ max) = {i, T[i]}

- L'intersection des règles deux à deux, donne l'ensemble vide :

(i ≤ n ^ T[i] > max) ∩ (i ≤ n ^ T[i] ≤ max) = {}, l'élément ne peut pas être à la fois plus petit et plus grand.

- (i > n) ∩ (i ≤ n ^ T[i] ≤ max) = {}, on ne peut pas être à la fois dans le tableau et hors du tableau.

Donc l'algorithme ne peut pas être dans deux cas à la fois.

1.1. Quelques problèmes algorithmiques classiques

– **Le problème de tri** d'un ensemble d'éléments selon une relation d'ordre (croissant ou décroissant), avec un milliard d'éléments.

– **Le problème de la Recherche** pertinente dans des grandes bases de données.

– **Réalisation d'une Optimisation multi objectifs,** trouver une combinaison des meilleurs critères qui minimisent une fonction coût (temps de calcul et espace mémoire).

1.2. Qualités d'un bon algorithme

- **Correct:** Il faut que le programme exécute correctement les tâches pour lesquelles il a été conçu.
- **Complet:** Il faut que le programme considère tous les cas possibles et donne un résultat dans chaque cas..
- **Efficace:** Il faut que le programme exécute sa tâche avec efficacité c'est à dire avec un coût minimal. Le coût pour un ordinateur se mesure en termes de temps de calcul et d'espace mémoire nécessaire.

Pour chaque problème, nous trouvons des différentes variantes définis le corps de l'algorithme. Ceci nous conduit à la nécessité d'estimer le coût d'un algorithme avant de l'écrire et l'implémenter.

2. Analyse et notions de complexité

L'analyse d'algorithmes permet de donner un outil d'aide à la comparaison des performances des algorithmes. Le mot complexité (latin : *complexus enlacé*, *embrassé* signifiant un objet constitué de plusieurs éléments ou plusieurs parties).

Cet outil permet de mesurer les ressources critiques (coûteuses) utilisées par les algorithmes.
Les ressources coûteuses :

- Le temps d'exécution
- Programmation
- La mémoire
- Les processeurs
- Les messages
- Les implémentations matérielles d'un algorithme
- La surface du circuit ou le nombre de portes logiques
- L'énergie consommée
- Les radiations émises.

La conception de certains ordinateurs mobiles permet de présenter le problème suivant :

Problème : l'énergie pour la conception de certains ordinateurs mobiles (la consommation électrique) induite par l'émission réception.

Solution : évaluer les performances de l'algorithme en fonction de cette ressource (la taille de cette ressource utilisée par l'algorithme).

2.1. Critères d'évaluation de performance de l'algorithme

L'efficacité d'un algorithme est désignée sous le terme de « complexité ». On la mesure, sur une taille d'entrée n donnée, par deux fonctions : *le nombre d'opérations* (complexité en temps) et *la place mémoire* (complexité en espace). Déterminer ces fonctions est le domaine de « l'analyse de complexité ».

Définition 3. La complexité est la mesure de l'efficacité d'un programme pour un type de ressources :
— Complexité temporelle (temps de calcul CPU) : nombre d'opérations élémentaires réalisés par le processeur, ce qui est lié directement au temps de calcul.
— Complexité spatiale (mémoire nécessaire) : espace mémoire RAM ou sur disque dur requis par le calcul.

2.1.1. Calcul de nombre d'opérations

Plusieurs façons d'écrire la performance d'un algorithme, avec des méthodes plus ou moins efficaces :

unité de mesure = l'opération élémentaire, propre à chaque algorithme
En additionnant toutes les opérations élémentaires
→ une bonne estimation du coût (coût est faible).
→ algorithme meilleur

la performance de l'algorithme dépend de ses entrées.

2.1.1.1. Opérations élémentaires

L'exécution d'un algorithme est une séquence d'opérations nécessitant plus ou moins de temps. Pour mesurer ce temps, il faut préciser quelles sont les opérations élémentaires à prendre en compte.

Pour un algorithme numérique ce sont les opérations arithmétiques de base (addition, multiplication, soustraction, division...), pour un algorithme de tri, ce sont les comparaisons entre éléments à trier (opérations plus coûteuses). lire ou modifier un élément d'un tableau, ajouter un élément à la fin d'un tableau, affecter un entier ou un flottant. Les transferts de valeurs ou affectations sont souvent négligées.

2.1.2. Mesure d'espace mémoire

Il est particulièrement important de mesurer la mémoire utilisée, car c'est une ressource qui peut-être rare ou dont l'utilisation peut s'avérer très coûteuse du point de

vue énergétique (imaginons les satellites…). Pour un problème donné, on peut donc être amené à choisir entre un algorithme rapide mais utilisant beaucoup de mémoire, et un algorithme plus lent qui utilise la mémoire de façon modérée.

Il faut examiner la *complexité* des algorithmes, *la nature des données, et éventuellement la machine cible.*

2.2. Coût d'un algorithme

Définition 4.
L'algorithme a un coût qui est lié :
- au nombre d'opérations effectuées (opérations arithmétiques, logiques, transferts...),
- à l'espace mémoire occupé par les données.
Evaluer ce coût revient à mesurer ce qu'on appelle la complexité de l'algorithme, en temps comme en espace. Il s'agit donc de dénombrer des opérations et des octets.

Dans la procédure d'évaluation de complexité, la complexité temporelle est plus importante que la complexité spatiale. L'étude de la complexité d'une fonction consiste à estimer son coût en ressource en fonctions des entrées. Pour différencier deux entrées entre elles, on compare en général leur taille. Les entrées seront constituées d'entiers, de flottants ou de tableaux. Pour les tableaux, la donnée pertinente est la taille. Pour les entiers, cela dépend du contexte. Pour un entier n, on peut en effet exprimer la complexité d'une fonction dépendant de n en fonction :
— de l'entier n lui-même.
— ou de son nombre de chiffres (sa taille), correspondant à $\log_2(n)$.

Définition 5.
Coût de A sur n: l'exécution de l'algorithme A sur la donnée n requiert $C_A(n)$ opérations élémentaires.

Estimer le coût d'une fonction sur une entrée de taille donnée signifie estimer le nombre de ces opérations élémentaires effectuées par la fonction sur l'entrée. La complexité en mémoire consiste à estimer la mémoire nécessaire à une fonction pour son exécution.

2.3. Complexité d'un algorithme

Définition 6. On appelle complexité de l'algorithme A pour la ressource R la fonction :
T(A,R, n) = max{ ressource R utilisée par A sur l'ensemble des données de taille n}

Définition 7. La taille d'une donnée est simplement la taille d'un bon codage en mémoire de cette donnée exprimée en nombres de bits.
La taille d'un codage d'un entier n sera :
[(log2(n + 1)], et non pas n.
∄ d'ambiguïté, T(A,R, n)= T(n).

⇨ la mesure du plus mauvais cas, car elle garantit que tout comportement de l'algorithme A utilisera au plus T(A,R, n) éléments de la ressource R.

Définitions 8.
On définit le Cas le pire et le cas moyen**.**
n désigne la taille de la donnée à traité.

- **Dans le pire des cas :** $C_A(n) := \max_{x|x|=n} C_A(x)$
- **En moyenne :** $C^{Moy}{}_A(n) := \sum_{x|x|=n} p_n(x)*C_A(x)$

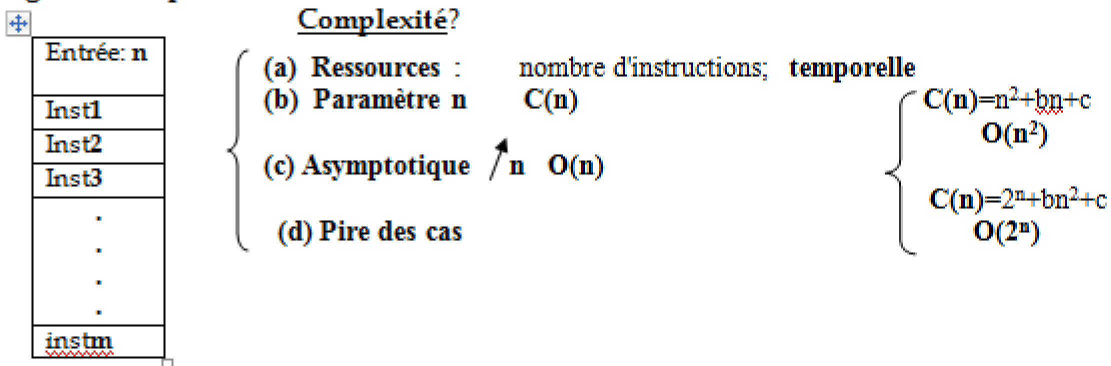
p_n : distribution de probabilité sur les données de taille n.

2.4. Complexité d’un problème

L’analyse d’un algorithme :
- Exprimer quelques chose sur le problème associé.
- La résolution d’un problème → tout algorithme n'a pas une complexité minimale.
≡ C'est la questions des *bornes inférieures* de complexité.

Problème : les calculs des fonctions de complexité sont difficiles à faire de manière exacte.
Solution : procède par approximations des notations (O, Ω et Θ).

2.5. Comment **calculer la complexité d'un algorithme?**



Algorithme: **Problème**

Complexité?

Entrée: n
Inst1
Inst2
Inst3
. . . .
instm

$$\begin{cases} C(n+1)=C(n) & O(1) \\ C(n+1)=C(n)+1 & O(n) \\ C(n+1)=C(n)+\varepsilon & O(\log_2(n)) \\ 2n \quad C(n)+1 & \\ C(n+1)=C(n)+n & O(n^2) \\ C(n+1)=2*C(n) & O(2^n) \end{cases}$$

2.6. Type de complexité

2.6.1. Complexités temporelle et spaciale d'un algorithme

2.6.1.1. Complexité en temps d'un algorithme

Définition 9.
I : l'ensemble des données d'instances d'un problème abstrait Π.
In : les données de taille n (le coût dépend de la donnée)
c(i) : le coût de l'algorithme résolvant le problème Π pour une donnée $i \in In$.
On définit 3 types de complexité :
→ Le coût dans le pire des cas (l'algorithme est le moins performant) :

$$W_{algorithme}(n) = \max\{c(i) \mid i \in In\}$$

→ Le coût dans le meilleur des cas (l'algorithme est le plus performant) :

$$B_{algorithme}(n) = \min\{c(i) \mid i \in In\}$$

→ Le coût dans cas moyen :

$$A_{algorithme} = \sum_{i \in In} p(i) * c(i)$$

En général : $A_{algorithme} = \frac{1}{|In|} \sum_{i \in In} c(i)$

2.6.1.2. Complexité en espace d'un algorithme

Permet de définir l'espace mémoire requis par le calcul.

2.6.2. Complexités pratique et théorique

Définitions 10.
La complexité pratique est une mesure précise des complexités temporelles et spatiales pour un modèle de machine donné.

La complexité théorique est un ordre de grandeur de ces couts, exprimé de manière la plus indépendante possible des conditions pratiques d'exécution.

2.7. Notion d'optimalité

Définition 11.
Un algorithme est dit optimal (performant) si sa complexité est la complexité minimale (la borne inférieure de complexité) parmi les algorithmes de sa classe.

Exemple 2 :
On peut montrer que tout algorithme résolvant le problème du tri a une complexité dans le pire des cas en $\Omega(nlgn)$. Le tri par fusion est en $O(nlgn)$ dans le pire des cas : il est donc optimal.

PARTIE 2.

CLASSES ET COMPLEXITE D'UN PROBLEME

1. Complexité d'un problème, Présentation et définitions

L'étude de la complexité des problèmes -computational complexity- s'attache à déterminer la complexité nécessaire d'un problème et à classifier les problèmes selon celle-ci, donc à répondre à des questions comme:
- Quelle est la complexité minimale d'un algorithme résolvant tel problème?
- Comment peut-on dire qu'un algorithme est optimal (en complexité)?
- Existe-t-il un algorithme polynomial pour résoudre un problème donné?
- Qu'est-ce qu'un problème "dur"?
- Comment prouver qu'un problème est au moins aussi "dur" qu'un autre?

Définition.
La complexité d'un problème est la complexité minimale dans le pire des cas d'un algorithme qui le résout. C'est souvent la complexité en temps qu'on considère mais on peut s'intéresser à d'autres mesures comme par exemple la complexité en espace-.

Calculer la complexité d'un problème est une chose extremement ardue en général. On se contente souvent d'encadrer:
- Pour trouver une *borne supérieure*, il suffit de trouver **UN** algorithme.
- Pour trouver une "***bonne***" *borne inférieure*, les choses sont souvent plus dures... : par exemple, pour montrer par exemple qu'un problème est de complexité au moins exponentielle, il faut montrer que **TOUT** algorithme le résolvant est exponentiel.

selon Sophie Tison, Cerner exactement la complexité d'un problème est souvent fort difficile: quand les deux bornes coıncident, c'est l'idéal mais c'est assez rare!

2. Trois méthodes pour trouver une borne inférieure

2.1. Les méthodes dites d'oracle ou d'adversaire
On suppose qu'il existe un algorithme utilisant moins d'un certain nombre d'opérations d'un certain type; on construit alors une donnée qui met en défaut l'algorithme.

2.2. Les arbres de décision
Ils sont utilisés pour les algorithmes de type recherche ou tri "par comparaisons": on suppose que seules des comparaisons entre les éléments sont utilisées pour obtenir de l'information sur l'ordre ou l'égalité des éléments.

Un arbre de décision "représente" toutes les comparaisons exécutées par l'algorithme sur les données d'une certaine taille:

- Un noeud correspond à une comparaison,
- Ses fils aux différentes réponses possibles de la comparaison (donc si le test est à valeur booléenne, les noeuds sont binaires);
- Une exécution possible correspond donc à une branche
- Deux données correspondant à la même branche correspondront à la même suite d'instructions.

2.3. Les Réductions
Le principe est simple. On essaie de ramener un problème à un autre (dont par exemple on connaıt la complexité).

Exemple. Si il existait un algorithme en O(nlogn) pour calculer le carré d'un entier de n chiffres, il existerait un algorithme en O(nlogn) pour calculer le produit de deux entiers de n chiffres,

3. Classes de complexité d'un problème

On s'intéresse aux ressources nécessaires pour la résolution de problèmes. Les ressources essentielles sont le temps et l'espace. L'idée est de classifier les problèmes selon l'ordre de grandeur de leur complexité.

3.1. Classes de complexité en temps

On ne considère ici que des problèmes décidables. On suppose que toutes les machines de Turing considérées s'arrêtent toujours. On définit le temps et espace de calcul relativement aux machines de Turing.

- **TIME(t(n))** = { L | L peut être décidé en temps t(n) par une machine de Turing déterministe }.
- **NTIME(t(n))** = { L | L peut être décidé en temps t(n) par une machine de Turing non déterministe }.
- Classes importantes : celles des problèmes qui peuvent être résolus en temps polynomial par une machine de Turing.

3.1.1. La classe P

- **Définition 1.** La classe des problèmes qui peuvent être résolus en temps polynomial par une machine déterministe.

  $\mathbf{P = \cup_{k\geq 0} TIME(n^k)}$

- **Définition 2.** **La classe P (ou PTIME)** est la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en temps.

Cette définition est (quasiment-) indépendante du modèle d'algorithme choisi: un algorithme polynomial en C sera polynomial si il est traduit en termes de machine de Turing et les modèles classiques de calcul (excepté les ordinateurs quantiques) sont polynomialement équivalents.

- **Définition 3**. P est la classe des problèmes praticables

Un problème de décision est dit praticable si il est dans P, impraticable sinon;

"existe-t-il un algorithme praticable pour ce problème?" se ramène à "ce problème est-il dans P?

3.1.2. La classe NP

- **Définition 1**. Classe des problèmes qui peuvent être résolus en temps polynomial par une machine non déterministe.

  $\mathbf{NP = \cup_{k\geq 0} NTIME(n^k)}$

- **Définition 2**. La classe **NP** contient énormément de propriétés associées à des problèmes courants qui apparaissent dans de multiples domaines: problèmes de tournées, d'emploi du temps, placement de taches, affectation de fréquences, rotation d'équipages, découpage, ...

3.1.3. La classe EXP

- **Définition 1**. Classe des problèmes qui peuvent être résolus en temps exponentiel par une machine déterministe.

  $\mathbf{EXP = \cup_{k\geq 0} TIME(2^{nk})}$

- **<u>Définition 2</u>**. **EXPTIME,** la classe des problèmes de décision pour lesquels il existe un algorithme de résolution exponentiel en temps.

PSPACE: la classe des problèmes de décision pour lesquels il existe un algorithme de résolution polynomial en espace. Bien sur PTIME est inclus dans PSPACE: un algo polynmial en temps "consomme au plus un espace polynomial".

La classe **NP** contient **P** et est contenue dans **ExpTime** (et même dans **Pspace**). Pour beaucoup de propriétés NP, on n'a pas trouvé d'algorithme polynomial, mais pour aucune d'entre elles, on n'a prouvé qu'elle ne pouvait pas être décidée en temps polynomial.

- **P ⊆ NP ⊆ EXP**

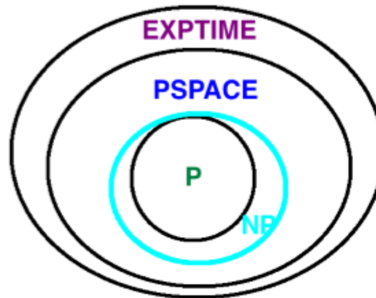


Figure 15. Classes de complexité d'un problème

3.2. Classes de complexité en espace

- **SPACE (f(n)) :** classe des problèmes de décision solvables par une TM (à K+2 bandes) en espace f(n)

- **NSPACE (f(n)) :** classe des problèmes de décision solvables par une NTM (à K+2 bandes) en espace f(n)

- **PSPACE:** classe de tous les problèmes de décision solvables par une TM utilisant un espace de travail limité par le polynôme de la taille de l'entrée.

- **NPSPACE :** classe de tous les problèmes de décision solvables par une NTM en espace polynomial de la taille de l'entrée.

4. NP-Complétude

Définition.
Un problème est simple quand le nombre d'étapes nécessaires à sa résolution peut-être borné par un polynôme ; c'est un problème **polynomial**, que l'on notera avec P.
Sinon, il est **difficile** : il s'agit d'un problème non-polynomial, noté **NP**.

De petites variations d'un problème peuvent être suffisantes pour passer d'un problème P à un problème NP :

- On veut partitionner un ensemble en deux sous-ensembles de même cardinalité, telle que la différence des sommes de ces sous-ensembles soit maximale.
- On veut partitionner un ensemble en deux sous-ensembles de même cardinalité, telle que la différence des sommes de ces sous-ensembles soit minimale.

Il est facile de couper un ensemble en deux de façon à avoir cette différence maximale.

Exemple :
On peut trier l'ensemble, et prendre comme parties les éléments de la fin et les éléments du début.
→ Obtenir une *différence minimale est un problème de combinatoire*.

Il existe de nombreux problèmes NP, souvent en combinatoire (l'algorithmique difficile) :
- Problème du voyageur de commerce (Travelling Salesman). Y a-t-il un cycle qui visite tous les noeuds d'un graphe, et dont le poids soit au plus K ? Minimiser le trajet pour voir les clients
- Factorisation. On veut décomposer un grand nombre en nombres premiers.
- Coloration de graphes. On veut attribuer à chaque sommet une couleur sans que deux sommets reliés soient de la même couleur ; le nombre minimal de couleurs requises est dit 'nombre chromatique' et noté $\gamma(G)$. Déterminer $\gamma(G)$ est NP-Complet dans le cas général.

Le problème qui se pose maintenant est de voir lorsque deux problèmes sont du même niveau de difficulté. L'outil de base pour établir des relations entre les complexités de différents problèmes est la *Réduction*.

5. Réduction d'un problème Q à un problème Π

Soit X une donnée d'entrée pour le problème Q, dont on veut une réponse 'oui' ou 'non'.
On transforme X en une donnée d'entrée Y pour le problème Π avec un algorithme A.
La réponse donnée au problème Π avec la donnée Y doit être la même que celle donnée au problème Q avec la donnée Y.

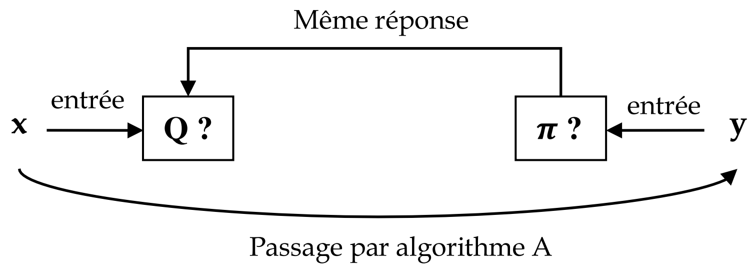


Figure 16. Réduction d'un problème Q à un problème Π

Si la donnée X peut être transformée en donnée Y par un algorithme polynomial, alors il n'est pas plus 'difficile' de résoudre Π que Q. Avec un polynôme, on passe d'un problème à l'autre.
Lorsque l'algorithme A est polynomial, on note Q → pΠ.
Si Q → pΠ et Π → pQ alors Q = pΠ ; on dit que Q et Π sont des problèmes équivalents en temps polynomial.

La plupart des problèmes d'optimisation sont NP-difficiles, en raison de la croissance de la complexité pour explorer un espace de recherche très grand, c'est-à-dire traiter des données de taille énormément pour atteindre la solution de problème.