Calculabilité, Combinatoire et Complexité

Emmanuel Hebrard

Plan

Complexité des algorithmes

Algorithme A

Algorithme : Appartient(L,x)

Données : une liste L triée, un entier x
Résultat : `Vrai` si $x \in L$, `Faux` sinon
ℓ : entier;
début
  pour chaque $\ell \in L$ **faire**
    si $\ell = x$ **alors**
      retourner *Vrai*;
  retourner *Faux*;

Algorithme B

Algorithme : Appartient(L,x,a,b)

Données : une liste L triée, un entier x
Résultat : `Vrai` si $x \in L$, `Faux` sinon
p : entier;
début
  si *L est vide* **alors retourner** *Faux*;
  $p = \lfloor |L|/2 \rfloor$;
  si $T[p] = x$ **alors retourner** *Vrai*;
  si $T[p] > x$ **alors**
    retourner *Appartient($L[:p], x$)*;
  retourner *Appartient($L[p+1:], x$)*;

Quel algorithme est le plus **efficace** ?

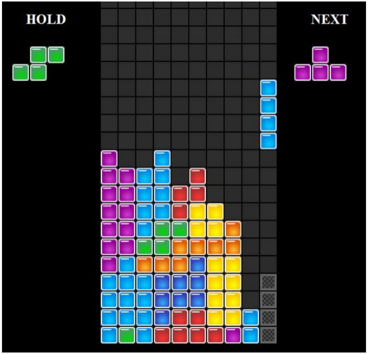
Calculabilité et complexité des problèmes

Les trois problèmes ci-dessous sont-ils faciles ? Difficiles ?

Les trois problèmes ci-dessous sont-ils faciles ? Difficiles ?

Problème 1 : (Tetris)

Etant donné une position de Tetris et la liste des pièces à venir, est-ce qu'il est possible de vider l'écran ?



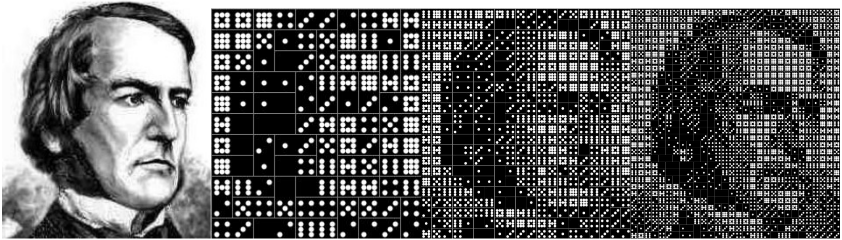
Calculabilité et complexité des problèmes

Les trois problèmes ci-dessous sont-ils faciles ? Difficiles ?

Problème 1 : (Tetris)

Problème 2 : (Portrait en dominos)

Etant donné *n* jeux de dominos et une image, trouver le pavage qui reproduit l'image au mieux



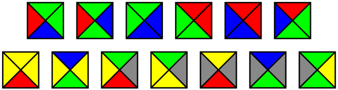
Les trois problèmes ci-dessous sont-ils faciles ? Difficiles ?

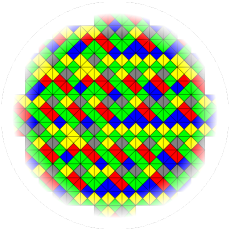
Problème 1 : (Tetris)

Problème 2 : (Portrait en dominos)

Problème 3 : (Dominos de Wang)

Etant donné un ensemble fini de dominos de Wang, est-ce qu'il existe un pavage du plan ?





Les trois problèmes ci-dessous sont-ils faciles ? Difficiles ?

Problème 1 : (Tetris)

Problème 2 : (Portrait en dominos)

Problème 3 : (Dominos de Wang)

Etant donné un ensemble fini de dominos de Wang, est-ce qu'il existe un pavage du plan ?

Qu'est-ce qu'être **facile** ? Être **difficile** ? Être **calculable** ? Comment le montrer ?

Vaincre la Combinatoire : Algo. ou Matériel ?

Problème du Voyageur de Commerce

- **donnée** : ensemble de villes
- **question** : quel est le plus court chemin passant par toutes les villes ?

- Méthode "Brute-force" : trois instructions par nano seconde
- Un ordinateur plus rapide : une instruction par *temps de Planck* ($5.39 \times 10^{-44} s$)

donnée	processeur 3 GHz	processeur de Planck
10 villes	1/100s	
15 villes	1 heure	
19 villes	1 an	
27 villes	8 × âge de l'univers	
35 villes	?	5/1000s
40 villes	?	12 heures
50 villes	?	4000 × âge de l'univers

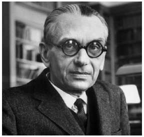
Analyse Asymptotique

Problème, Donnée & Algorithme

Définition : Problème $\simeq$ fonction sur les entiers

- Une question Q qui associe une donnée x à une réponse y
  - "Quel est le plus court chemin de x_1 vers x_2 par le réseau R ?"
  - "Quel est la valeur du carré de x ?"
- Toute donnée peut se coder par un *Nombre de Gödel*
- Q est une relation, pas toujours une fonction : plus court(s) chemin(s)
- On peut se restreindre aux fonctions

$$\begin{pmatrix} 1 & 1 \\ 2 & 4 \\ 3 & 9 \\ 4 & 16 \\ \dots & \dots \end{pmatrix}$$



- Entrée x & Sortie y ; Instance x & Solution y ; Argument x & Valeur y

Algorithme : méthode pour résoudre un problème

- Non-ambiguë, composée d'instructions primitives

Complexité Algorithmique : Problématique

On veut être capable de :

- Évaluer l'efficacité d'un algorithme ;
- Comparer deux algorithmes entre eux ;

... sans les implémenter ! ! ! !

Question

- Qu'est ce qu'un algorithme efficace ?
- Quels critères utiliser ?
  - espace mémoire ;
  - **temps d'exécution** ;
  - simplicité du code ;

Le temps d'exécution

Le temps d'exécution est le nombre de cycle machine (**secondes**) que prend le programme pour s'executer.

Mais le temps d'éxecution dépend :

- de la machine ;
- du langage ;
- de l'instance (des paramètres)...

On veut une méthode **indépendante de l'environnement**.

Nombre d'opérations élémentaires (I)

Opération élémentaire

Une opération élementaire est une opération qui prend un temps constant

- Même temps d'exécution quelque soit la donnée

Exemples d'opérations en temps constant

- Instructions assembleur
- Opérations arithmétiques $(+, \times, -)$, affectation, comparaisons sur les **types primitifs** (entiers, flottants, etc.)

Exemple

L'algorithme suivant calcule $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$ (avec $0! = 1$).

Algorithme : Factorielle(n)

Données : un entier n

Résultat : un entier valant $n!$

```
fact : entier;
début
    fact ← 1;
    pour i allant de 2 à n faire
        fact = fact ∗ i;
    retourner fact;
```

	nombre	coût
initialisation :	$1\times$	1 op.
itérations :	$n\times$	1 op.
mult. + affect. :	$(n-1)\times$	2 op.
retour fonction :	$1\times$	1 op.

Nombre total d'opérations :

$$1 + n + (n-1) * 2 + 1 = 3n$$

Nombre d'opérations élémentaires (II)

- Le nombre d'opérations dépend en général de la donnée du problème ;

  Exemple :

  (a) trier 10 entiers vs. trier 1000000 entiers

  (b) tri bulle de 1000 entiers déjà triés vs. tri bulle de 100 entiers triés dans le sens inverse

- (a) $\Rightarrow$ Donner la complexité en fonction de la **taille de la donnée** ;
  $\rightarrow$ nombre de bits de sa représentation en mémoire

- (b) $\Rightarrow$ Plusieurs types de complexités peuvent être calculées
  $\rightarrow$ pire/meilleur cas ou en moyenne.

Complexité en fonction de la taille de la donnée

Soit $\mathrm{Coût}_A(x)$ la complexité de l'algorithme A sur la donnée x de taille n.

Complexité dans le meilleur des cas

$$\mathrm{Inf}_A(n) = \min\{\mathrm{Coût}_A(x) \mid x \text{ de taille } n\}$$

Complexité dans le pire des cas

$$\mathrm{Sup}_A(n) = \max\{\mathrm{Coût}_A(x) \mid x \text{ de taille } n\}$$

Complexité en moyenne

Besoin d'une probabilité $P()$ pour toutes les données de tailles n

$$\mathrm{Moy}_A(n) = \sum_{x \text{ de taille } n} P(x) \cdot \mathrm{Coût}_A(x)$$

Exemple (I)

L'algo. suivant recherche l'élément e dans un tableau.

Algorithme : RechercheElmt(T, e)
Données : un entier e et un tableau T contenant e
Résultat : l'indice i t.q. $T[i] = e$
i : entier;
début
  $i \leftarrow 0$;
  tant que $T[i] \neq e$ **faire**
    $i \leftarrow i + 1$;
  retourner i;

Le nombre de comparaison dépend de la donnée T :

- e est dans la case 1 $\rightarrow$ 1 comp.
- e est dans la case j $\rightarrow$ j comp.
- e est dans la case n $\rightarrow$ n comp. (n : taille de T)

meilleur : 1 comp.
pire : n comp.
moyenne : $\frac{n+1}{2}$ (voir slide suivant)

L'algo. suivant recherche l'élément e dans un tableau.

Algorithme : RechercheElmt(T,e)
Données : un entier e et un tableau T contenant e
Résultat : l'indice i t.q. $T[i] = e$
i : entier;
début
  $i \leftarrow 0$;
  tant que $T[i] \neq e$ **faire**
    $i \leftarrow i + 1$;
  retourner i;

moyenne : $\frac{n+1}{2}$

Hyp. :

- distribution uniforme
- $nbOcc(e) = 1$

$\Rightarrow P(T[i] = e) = 1/n.$

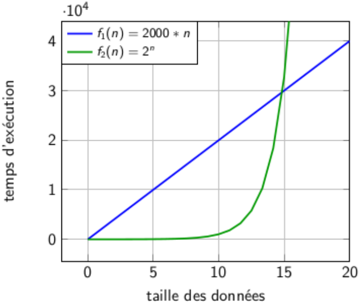
On applique la formule :

$$\mathrm{Moy}_A(n) = \sum_{x \text{ de taille } n} P(x) \cdot \mathrm{Coût}_A(x)$$

$$\mathrm{Moy}_A(n) = \sum_{i=1}^{n} P(T[i] = e) \times i$$

Complexité Algorithmique

- Vision pessimiste : la **complexité** d'un algortihme est souvent définie comme sa performance **asymptotique** dans le **pire cas**
- Que signifie **dans le pire des cas** ?
  - Parmi toutes les données x de taille n, on ne considère que celle qui maximise $\mathrm{Coût}_A(x)$
- Que signifie **asymptotique** ?
  - comportement de l'algorithme pour des données de taille *n arbitrairement grande*
  - pourquoi ?



- Soit deux algorithmes de complexités $f_1(n)$ et $f_2(n)$
- Quel algorithme préférez-vous ?
- La courbe verte semble correspondre à un algorithme plus efficace...
- ... mais seulement pour de très petites valeurs !

Ordre de grandeur : motivation

- Les calculs à effectuer pour évaluer le temps d'exécution d'un algorithme peuvent parfois être longs et pénibles ;
- De plus, le degré de précision qu'ils requièrent est souvent inutile ;
  - $n \log n + 5n \rightarrow 5n$ va devenir "négligeable" ($n >> 1000$)
  - différence entre un algorithme en $10n^3$ et $9n^3$: effacé par une accélération de $\frac{10}{9}$ de la machine
- On aura donc recours à une **approximation** de ce temps de calcul, représentée par les notations $\mathcal{O}, \Omega$ et Θ

Hypothèse simplificatrice

On ne s'intéresse qu'aux fonctions **asymptotiquement positives** (positives pour tout $n \geq n_0$)

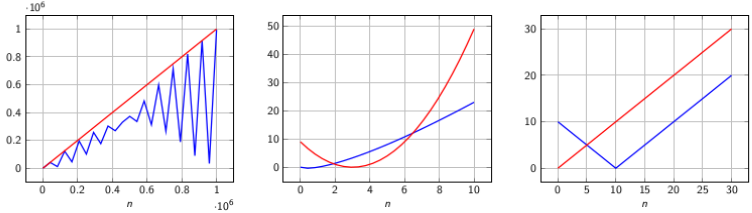
Notation $\mathcal{O}$: définition

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0 \quad : \qquad f(n) \leq c \times g(n)$$

Borne supérieure : $f(n) \in \mathcal{O}(g(n))$ **s'il existe une constante** c**, et un seuil à partir duquel** $f(n)$ est inférieure à $g(n)$, à un facteur c près ;

Exemple : $f(n) \in \mathcal{O}(g(n))$



$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0 \ : \qquad f(n) \leq c \times g(n)$$

Prouver que $f(n) \in \mathcal{O}(g(n))$: jeux contre un perfide adversaire $\forall$

Tour du joueur $\exists$ choisit c et n_0

Tour du joueur $\forall$ choisit $n \geq n_0$

Arbitre détermine le gagnant : $f(n) \leq cg(n)$

Notation $\mathcal{O}$: exemple

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0 \ : \qquad f(n) \leq c \times g(n)$$

Jeux : **prouver** que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $\mathcal{O}(n^2)$:

Tour du joueur $\exists$ choisit $c = 6$ et $n_0 = 0$

Tour du joueur $\forall$ choisit $n = 5$

Arbitre $6 \times 5^2 + 2 \times 5 - 8 > 6 \times 5^2$

LAAS CNRS

Notation $\mathcal{O}$: exemple

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R},\ \forall n \geq n_0 \ : \qquad f(n) \leq c \times g(n)$$

Jeux : **prouver** que la fonction $f_2(n) = 6n^2 + 2n - 8$ est en $\mathcal{O}(n^2)$:

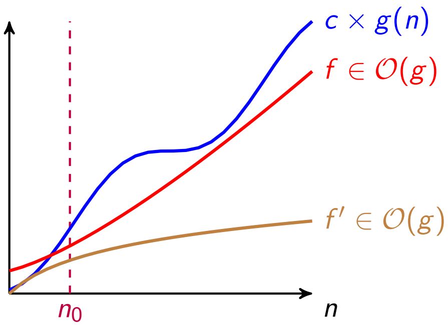
Tour du joueur $\exists$ choisit $c = 7$ et $n_0 = 0$

Tour du joueur $\forall$ choisit ?

Arbitre $n^2 - 2n + 8 = 0$ n'a pas de solution

$\mathcal{O}(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R}^{+*},\ \forall n \geq n_0 \quad : \qquad f(n) \leq c \times g(n)$$



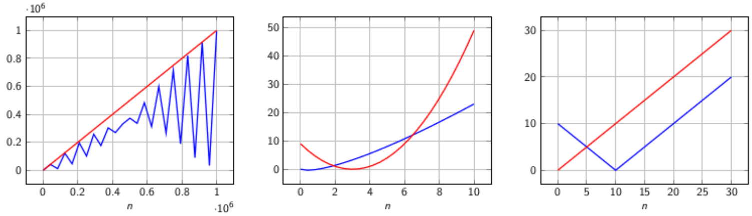
Exercice : $2n^2$ est-il en $\mathcal{O}(n^2)$? Pareil pour $2n$.

$\Omega(g(n))$ **est l'ensemble de fonctions** $f(n)$ **telles que :**

$$\exists n_0 \in \mathbb{N},\ \exists c \in \mathbb{R}^{+*},\ \forall n \geq n_0 \ : \qquad f(n) \geq c \times g(n)$$

Borne inférieure : $f(n) \in \Omega(g(n))$ s'il existe un seuil à partir duquel $f(n)$ est supérieure à $g(n)$, à une constante multiplicative près ;

Exemple : $g(n) \in \Omega(f(n))$



Notation Θ : définition

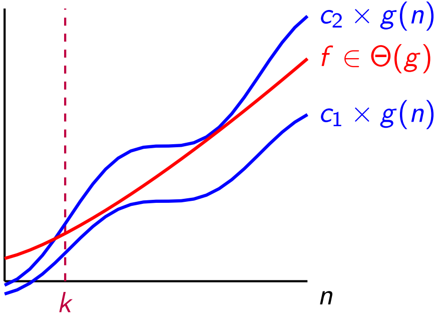
$\Theta(g(n))$ est l'ensemble de fonctions $f(n)$ telles que :

$$\exists c_1, c_2 \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n > n_0, c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$$

Borne supérieure et inférieure : $\Theta(g(n)) = \Omega(g(n)) \cap \mathcal{O}(g(n))$; $f(n)$ est en $\Theta(g(n))$ si elle est **prise en sandwich** entre $c_1 g(n)$ et $c_2 g(n)$;

$f(n)$ est en $\Theta(g(n))$ si :

$$\exists c_1, c_2 \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N}, \forall n > n_0, \qquad c_1 \times g(n) \leq f(n) \leq c_2 \times g(n)$$



Exercice : $2n^2$ est-il en $\Theta(n^2)$? Pareil pour $2n$.

Notation asymptotique d'une fonction

- Quelle est la borne asymptotique de $f(n)$?

Notation asymptotique (de l'expression fermée) d'une fonction

Les mêmes simplifications pour $\mathcal{O}, \Omega$ et Θ :

- **on ne retient que les termes dominants**
- **on supprime les constantes multiplicatives**

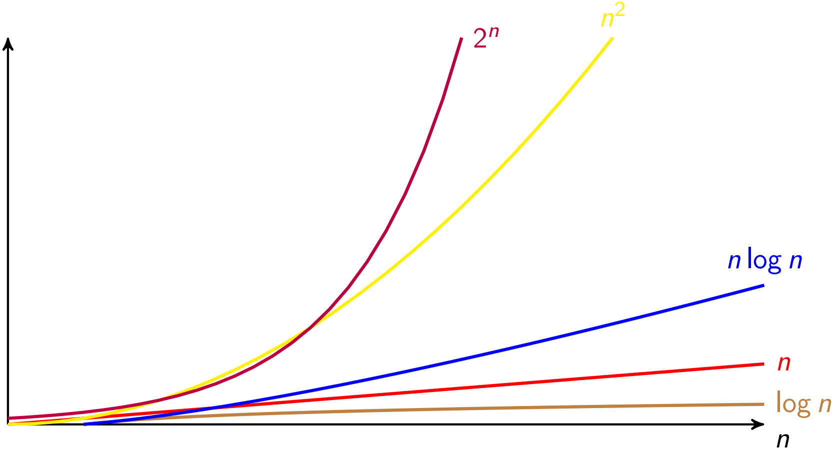
Exemple

Soit $g(n) = 4n^3 - 5n^2 + 2n + 3$;

1. on ne retient que le terme de plus haut degré : $4n^3$ (pour n assez grand le terme en n^3 "domine" les autres, en choisissant bien c_1, c_2, on peut avoir $c_1 n^3 \leq g(n) \leq c_2 n^3$)
2. on supprime les constantes multiplicatives : n^3 (on peut la choisir !)

et on a donc $g(n) \in \Theta(n^3)$

Relation des principaux ordres de grandeur



Indépendant de la taille de la donnée : $\mathcal{O}(1)/\Theta(1)$

Ordres de grandeurs : exemples

Une instruction de base est de l'ordre de la μs.

T./C.	$\log n$	n	$n \log n$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14} siècles
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	astronomique
10000	$14\mu s$	$1/100s$	$1/7s$	$1.7mn$	astronomique
100000	$17\mu s$	$1/10s$	$2s$	$2.8h$	astronomique

Algorithmes Récursifs

Règles de calculs : combinaisons des complexités

- Les instructions de base prennent un temps constant, noté $\mathcal{O}(1)$;
- On additionne les complexités d'opérations en séquence :

$$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$$

- Même chose pour les branchements conditionnels :

$$\max(\mathcal{O}(f(n)), \mathcal{O}(g(n))) = \mathcal{O}(f(n)) + \mathcal{O}(g(n))$$

Exemple

si *<condition>* **alors** — $\mathcal{O}(g(n))$
 #instructions (1); — $\mathcal{O}(f_1(n))$
sinon
 #instructions (2); — $\mathcal{O}(f_2(n))$

$$\left.\begin{array}{l}\mathcal{O}(g(n))\\ \mathcal{O}(f_1(n))\\ \mathcal{O}(f_2(n))\end{array}\right\} = \mathcal{O}(g(n) + f_1(n) + f_2(n))$$

Règles de calculs : combinaison des complexité

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;

Règles de calculs : combinaison des complexité

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;

- Calcul de la complexité d'une boucle `while` :

Exemple

en supposant qu'on a $\mathcal{O}(h(n))$ itérations

tant que *<condition>*
faire
#instructions ;

$$\left.\begin{array}{l}\mathcal{O}(g(n))\\ \mathcal{O}(f(n))\end{array}\right\} = \mathcal{O}(h(n) \times (g(n) + f(n)))$$

Règles de calculs : combinaison des complexité

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;
- Calcul de la complexité d'une boucle `for` :

Exemple

pour *i allant de a à b* **faire**
 #instructions ;

$$\left.\begin{array}{l} \\ \mathcal{O}(f(n)) \end{array}\right\} = \mathcal{O}((b-a+1)\times f(n))$$

Calcul de la complexité asymptotique d'un algorithme

- Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque "partie" de l'algorithme ;
2. on combine ces complexités conformément aux règles qu'on vient de voir ;
3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues ;
   - élimination des constantes, et
   - conservation du (des) termes dominants

Exemple : calcul de la factorielle de $n \in \mathbb{N}$

- Reprenons le calcul de la factorielle, qui nécessitait $3n$ opérations :

Algorithme : Factorielle(n)
Données : un entier n
Résultat : un entier valant $n!$
fact, *i* : entier;
début
  $fact \leftarrow 2$;
  pour *i allant de* 3 *à n* **faire**
    $fact \leftarrow fact * i$;
  retourner *fact*;

	nombre	coût
initialisation :	$\Theta(1)\times$	$\Theta(1)$
itérations :	$\Theta(n)\times$	$\Theta(1)$
mult. + affect. :	$\Theta(n)\times$	$\Theta(1)$
retour fonction :	$\Theta(1)\times$	$\Theta(1)$

Nombre total d'opérations :

$$\Theta(1) + \Theta(n) * \Theta(1) + \Theta(n) * \Theta(1) + \Theta(1) = \Theta(n)$$

Exemple : Tri à bulles

- La méthode de "comptage" n'est pas toujours applicable directement

Algorithme : TriBulles(T)
Données : un tableau T
Résultat : le tableau T trié
début

```
pour i allant de 2 à |T| faire
    pour j allant de 1 à i − 1 faire
        si T[j + 1] < T[j] alors
            échanger(T[j + 1], T[j]);
```

	nombre	coût
itérations :	$\Theta(\lvert T\rvert)\times$	$\Theta(1)$
itération :	$?\times$	$\Theta(1)$
1ère itération :	1 comparaison	
2ème itération :	2 comparaisons	
...		

Séries arithmétiques

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{1}{2}n(n+1) \in \Theta(n^2)$$

Nombre total d'opérations : $\Theta(|T|^2)$

Récurrences

- Stratégie "diviser pour régner"

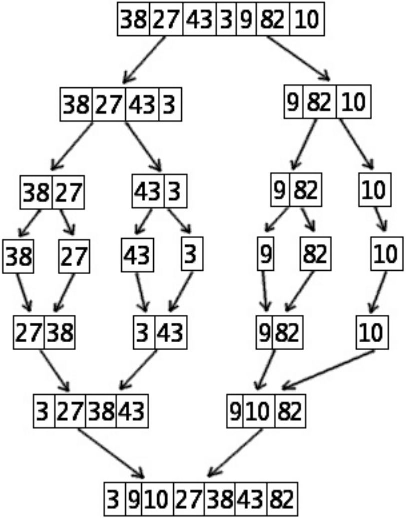
Algorithme : TriFusion(T)

Données : un tableau T
Résultat : le tableau T trié
mil : entier;
début
  si $|T| \leq 1$ **alors**
    retourner T;
  sinon
    $milieu \leftarrow \lfloor \frac{|T|+1}{2} \rfloor$;
    retourner
     $Fusion(T[:mil]\ ,\ T[mil:])$;

Algorithme : Fusion(T1,T2)

Données : deux tableaux $T1$ et $T2$
Résultat : un tableau T trié contenant les éléments de $T1$ et de $T2$
T : tableau de taille $T1 + T2$;
début
  si $|T1| = 0$ **alors**
    $T \leftarrow T2$;
  sinon si $|T2| = 0$ **alors**
    $T \leftarrow T1$;
  sinon si $T1[1] < T2[1]$ **alors**
    $T[1] \leftarrow T1[1]$;
    $T[2:] \leftarrow Fusion(T1[2:], T2[1:])$;
  sinon
    $T[1] \leftarrow T2[1]$;
    $T[2:] \leftarrow Fusion(T1[1:], T2[2:])$;
  retourner T

Illustration du Tri Fusion



Récurrences

- Temps d'exécution T dans le pire des cas du tri fusion pour trier un tableau de n entiers
- $$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$
- Complexité du tri fusion : $T(n) = \Theta(n \log n)$
- Comment passer de l'un à l'autre ?
  - Méthode par substitution
  - Méthode générale (théorème maître)

Méthode par substitution

- Il faut avoir une intuition sur la forme de la solution (ici : $\mathcal{O}(n \log n)$)
- On veut montrer que $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n) \in \mathcal{O}(n \log n)$
- On montre par induction qu'il existe $f(n)$ t.q. $\forall n \; T(n) \leq f(n)$
  - On va en déduire **a posteriori** que $T(n) \in \mathcal{O}(f(n))$

Attention !

- L'hypothèse d'induction est $T(n) \leq f(n)$, et **non** $T(n) \in \mathcal{O}(f(n))$
- L'argument inductif "pour tout $x < n$, $T(x) \in \mathcal{O}(f(x))$" ne veut pas dire grand chose puisque la notation $\mathcal{O}$ est définie pour n arbitrairement grand : **on remplace tous les termes en $\mathcal{O}, \Omega, \Theta$ par une fonction élément de l'ensemble**

Méthode par substitution (condition aux limites)

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \leq cn \log n$$

- Il faut montrer que la formule est vraie pour les conditions limites de la récurrence pour des données de petite taille, i.e. $n = 1$
- **Problème :** c'est faux pour $n = 1$ car $c \times 1 \times \log 1 = 0 < T(1) = 1$;
- Mais on cherche à montrer la complexité pour des données de grande taille : $n \geq n_0$ et on a le choix pour $n_0 \implies$ vérifier pour $T(2)$ (et $T(3)$)
- On peut aussi borner par $f(n) = cn \log n + b$ puisque $cn \log n + b \in \mathcal{O}(n \log n)$
  - Ou même $f(n) = cn \log n + an + b$

Méthode par substitution (condition aux limites)

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \leq cn \log n$$

- On vérifie que la formule tient pour $T(2)$ et $T(3)$

$$T(2) = 2T(\lfloor 2/2 \rfloor) + 2$$

$$T(2) = 2T(1) + 2$$

$$T(2) = 2 * 1 + 2 = 4 \leq 2c \log 2 = 2c$$

$$T(2) = 4 \leq 2c$$

$$c \geq 2$$

- On fait la même chose pour $T(3)$...
- ... et on obtient que c doit être ≥ 2.

Méthode par substitution (Induction)

$$T(n) = 2T(\lfloor n/2 \rfloor) + n \leq cn \log n$$

- On suppose maintenant que $T(x) \leq cx \log x$ est vrai pour tout $2 \leq x \leq n-1$; et on vérifie que c'est aussi le cas pour $x = n$

$$T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor$$

- On substitue dans l'expression

$$\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + n \\
&\leq 2c\lfloor \tfrac{n}{2} \rfloor \log(\lfloor \tfrac{n}{2} \rfloor) + n \\
&\leq cn \log(n/2) + n \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - cn + n \\
&\leq cn \log n
\end{aligned}$$

- A condition que $c \geq 1$ (on prendra $c \geq 2$, à cause de $T(2)$ et $T(3)$)

Méthode générale

- Pour les récurrence de la forme $T(n) = aT(n/b) + f(n)$ avec $a \geq 1$ et $b > 1$
- L'algorithme découpe la donnée en a sous-problèmes de taille n/b et les résout récursivement

  La fonction f représente le coût de division et de « fusion » du problème.
- Exemple pour le tri fusion : $a = 2$, $b = 2$ et $f(n) = \Theta(n)$
- Exemple pour la recherche binaire : $a = 1$, $b = 2$ et $f(n) = \Theta(1)$
- Il existe un théorème pour calculer la complexité : le théorème maître

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(n/b) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ pas de domination

- Tri fusion :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T(n/2) + \Theta(n) & \text{si } n > 1 \end{cases}$$

- $a = 2,\ b = 2,\ d = 1,\ \log_2 2 = 1 = d$

  On est donc dans le cas 3 et la complexité en $\Theta(n \log n)$

Théorème maître (général) - version simplifiée

- On ne considère que les récurrences $T(n) = aT(n/b) + \Theta(n^d)$ (ou $\mathcal{O}(n^d)$) avec $a \geq 1, b > 1, d \geq 0$

1. Si $d > \log_b a$, $T(n) = \Theta(n^d)$ — complexité dominée par le coût de fusion
2. Si $d < \log_b a$, $T(n) = \Theta(n^{\log_b a})$ — complexité dominée par le coût du sous-problème
3. Si $d = \log_b a$, $T(n) = \Theta(n^d \log n)$ — pas de domination

- Recherche binaire :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ T(n/2) + \Theta(1) & \text{si } n > 1 \end{cases}$$

- $a = 1,\ b = 2,\ d = 0,\ \log_2 1 = 0 = d$

  On est donc dans le cas 3 et la complexité en $\Theta(\log n)$

Taille de la Donnée

Taille de la donnée $|x|$

- Espace mémoire nécessaire pour la représenter x
- Dans une unité quelconque, fonction de la donnée : $|x| \in \Theta(f(x))$

- Type primitif entier, flottant, etc. sur 32 ou 64 bits : $\Theta(1)$
- Entier non-borné x : $|x| \in \Theta(\log x)$
- Liste L de n caractères : $|L| \in \Theta(n)$

La taille de la donnée : exemple

- La complexité se mesure **en fonction de la taille de la donnée**
- Parfois, on mesure plus facilement la complexité d'un algorithme en fonction d'un autre paramètre que la taille de la donnée (ex : sa **valeur**)
- Si l'algorithme **A** est en $\Theta(f(x))$ pour une donnée x et la taille $|x|$ est en $\Theta(g(x))$, alors la complexité de **A** sera en $\Theta(f(g^{-1}(x)))$

Exemple

Algorithme : Carré(x)

Données : un entier x
Résultat : un entier valant x^2
r : entier;
début
    $r \leftarrow 0$;
    pour *i allant de 1 à x* **faire**
        pour *j allant de 1 à x* **faire**
            $r \leftarrow r + 1$;
    retourner r;

- Complexité : $\Theta(x^2)$
- Mais la taille de la donnée est $|x| = \Theta(\log x)$
- Autrement dit, $x = \Theta(2^{|x|})$
- cet algorithme est donc en $\Theta(2^{2^{|x|}})$ (exponentiel !)

La taille d'une structures de données

- Dans le cas de structures de données (listes, ensembles, etc.) il faut analyser la structure : la taille de chaque élément, les pointeurs, etc.

Exemple, liste L de n éléments

Il faut $\Theta(1)$ bits par élément, plus la somme des tailles des éléments : donc $|L| \in \Theta(n) + \sum_{e \in} |e| \subseteq \Theta(n) \times \Theta(|e|)$

- Les éléments sont des entiers : $|L| \in \Theta(n)$
- Les éléments sont des entiers non bornés et m est l'élément maximum : $|L| \in \Theta(n \log m)$

La taille d'une structures de données

- La taille des structures va dépendre de leur codage :

Exemple, graphe $G = (S, A)$

Tableau de tableaux : Il faut $\Theta(1)$ bits par arc **potentiel**, donc

$$|G| \in \Theta(|S|^2)$$

Listes d'adjacence : Il faut $\Theta(\log |S|)$ bits par arc dans A, donc

$$|G| \in \Theta(|A| \log |S|)$$

Attention : sujet piégeux !

- Taille d'une liste de n entiers (codés sur 32 bits) : $\Theta(n)$
- Taille d'une liste de n entiers (où l'élément maximum est m) : $\Theta(n \log m)$
- Taille d'un ensemble de n entiers (codés sur 32 bits) : $\Theta(1)$! !
  - Il y a un nombre fini d'entiers représentable avec 32 bits (2^{32}), et donc un nombre fini d'ensembles de tels entiers ($2^{2^{32}}$)

Classes de Problèmes

Vocabulaire

Un algorithme est dit :

- en temps *constant* si sa complexité (dans le pire des cas) est bornée par une constante
- *linéaire* (resp. linéairement borné) si sa complexité (dans le pire des cas) est $\Theta(n)$ (resp. $\mathcal{O}(n)$)
- *quadratique* (resp. au plus quadratique) si sa complexité (dans le pire des cas) est $\Theta(n^2)$ (resp. $\mathcal{O}(n^2)$)
- *polynômial* ou polynômialement borné, si sa complexité (dans le pire des cas) est en $\mathcal{O}(n^p)$ pour un certain $p > 0$
- (au plus) exponentiel si elle est en $\mathcal{O}(2^{n^c})$ pour un certain $c > 0$

Classes de problèmes

- Analyser la complexité des algorithmes permet de faire un choix éclairé, **mais pas seulement**
- Classer les problèmes
  - en fonction de la complexité du meilleur algorithme connu pour les résoudre

$\mathbf{TIME}\ (f(n))$

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(f(n))$

- *Tri* $\in \mathbf{TIME}(n \log n)$
- *Recherche dans un tableau trié* $\in \mathbf{TIME}(\log n)$
- Tout algorithme de tri est en $\Omega(n \log n) \implies$ *Tri* $\notin \mathbf{TIME}(n)$
- **Trier est plus "difficile" que chercher dans un tableau trié**

Classes de problèmes (suite)

- On peut analyser l'espace mémoire utilisé par un algorithme de manière similaire

$\mathbf{SPACE}\ (f(n))$

Ensemble des problèmes pour lesquels il existe un algorithme nécessitant $\mathcal{O}(f(n))$ octets

- On ne compte pas la taille de la donnée, mais on compte la taille de la réponse

Théorème

$\mathbf{TIME}(f(n)) \subseteq \mathbf{SPACE}(f(n))$

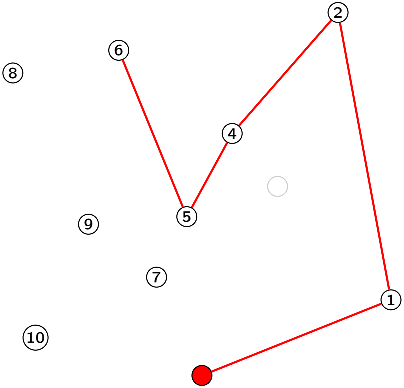
- Problème $\mathbf{A} \in \mathbf{TIME}(f(n)) \implies \mathbf{A} \in \mathbf{SPACE}(f(n))$
- Chaque octet utilisé implique $\Omega(1)$ opération(s)

Parcours de Graham

- Trouver l'enveloppe convexe d'un ensemble de points

Algorithme

1. Trouver le point p de plus petite ordonnée $\mathcal{O}(n)$
2. **Trier** les autres points en fonction de l'angle avec l'axe des abcisses par rapport à p
3. Appliquer le parcours de Graham $\mathcal{O}(n)$
   - retour arrière en cas de "tournant à droite"



Réduction polynômiale

- On peut trouver l'enveloppe convexe en $\mathcal{O}(n)$ plus le coût de trier n entiers
- $\mathit{Tri} \in \mathbf{TIME}(n \log n) \implies \mathit{Enveloppe\ convexe} \in \mathbf{TIME}(n \log n)$
- Réduction de *Enveloppe convexe* vers *Tri* :
  - *Enveloppe convexe* pas plus difficile que *Tri*
  - *Tri* au moins aussi difficile que *Enveloppe convexe*

Réduction polynômiale de A vers B (de Cook / de Turing) :

Un algorithme pour résoudre **A** en $\mathcal{O}(n^c)$ (pour c un constante et n la taille de la donnée) sous l'hypothèse que le problème **B** peut être résolu en $\Theta(1)$

Les Classes P et NP

Classe “Temps Polynômial”

- Rappel :

TIME $(f(n))$

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(f(n))$

- Classe des problèmes pour lesquels il existe un algorithme polynômial :

PTIME ou simplement : P

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(n^c)$ pour une constante c.

$$\mathbf{P} = \bigcup_{c \geq 0} \mathbf{TIME}(n^c)$$

Classe “Temps Exponentiel”

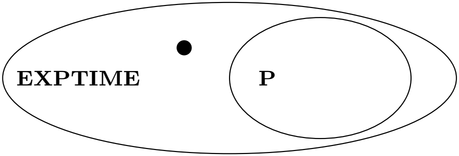
- Classe des problèmes pour lesquels il existe un algorithme exponentiel :

EXPTIME :

Ensemble des problèmes pour lesquels il existe un algorithme en $\mathcal{O}(2^{n^c})$ pour une constante c

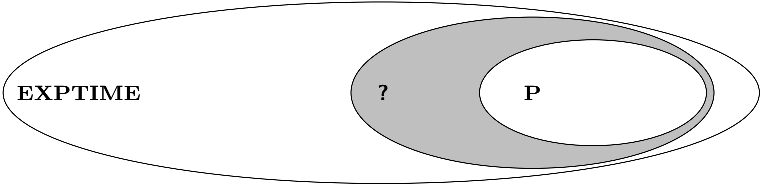
$$\mathbf{EXPTIME} = \bigcup_{c \geq 1} \mathbf{TIME}(2^{n^c})$$

- Evidemment, $\mathbf{P} \subseteq \mathbf{EXPTIME}$
- Est-ce que $\mathbf{P} \subset \mathbf{EXPTIME}$? **Oui !**
- Il existe des problèmes en $\Omega(2^n)$



Problèmes "intermédiaires"

- Une classification trop grossière ?



- Il existe des problèmes pour lesquels on ne connait pas d'algorithme en temps polynômial, sans pouvoir prouver qu'ils n'en n'ont pas
- Ces problèmes sont très nombreux...
- ... et très intéressants : *Voyageur de Commerce*, *Programmation Linéaire en Nombres Entiers*, *SAT*, *Isomorphisme de Graphes*, etc.

Classes non-déterministes

- Puisque on ne peut pas prouver que ces problèmes n'ont pas d'algorithme en temps polynômial, on n'est pas sûr qu'une telle classe **existe**
- Si elle existe, alors cette classe correspond à la notion de **non-déterminisme**
- Rappel de la définition d'un algorithme :

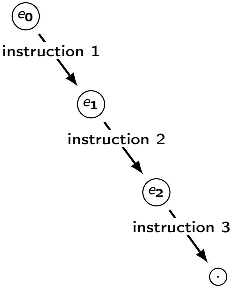
Algorithme : méthode pour résoudre un problème

- Non-ambiguë**Non-ambiguë**, composée d'instructions primitives
- Non-ambiguité = déterminisme

Algorithmes déterministes

Pour un certain **état de la mémoire**, et une certaine **position** dans le code, il n'existe qu'**une seule** instruction possible avec **un seule** effet possible (sur l'état de la mémoire et la position).

- **État** : contenu de la mémoire + position d'un *curseur* dans le code
- **Transition** : Instruction
- **Effet** : modification de la mémoire et déplacement du curseur



Algorithmes déterministes

Pour un certain **état de la mémoire**, et une certaine **position** dans le code, il n'existe qu'**une seule** instruction possible avec **un seule** effet possible (sur l'état de la mémoire et la position).

Algorithme : TriBulles(T)

Données : un tableau T
Résultat : le tableau T trié

```
début
    pour i allant de 2 à |T| faire
        pour j allant de 1 à i − 1 faire
            si T[j] > T[j + 1] alors
                échanger(T[j], T[j + 1]);
```

Mémoire

T :

5	3	4	9

i : 2

j : 1

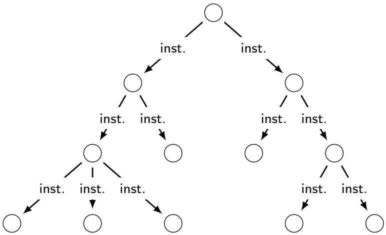
Algorithmes non-déterministes

Pour un certain état de la mémoire, et une certaine position dans le code, il peut y avoir **plusieurs** instructions possibles, avec des effets potentiellement différents (sur l'état de la mémoire et la position).

Algorithme : TriNonDéterministe(T)

Données : un tableau T
Résultat : le tableau T trié

début
  pour *i allant de* 1 *à* $|T| - 1$ **faire**
    Devine $j \in [i, .., |T|]$ t.q. $T[j]$ est minimal;
    Échange $T[i]$ et $T[j]$;
    si $i > 0$ *et* $T[i-1] > T[i]$ **alors** stop;



Algorithmes non-déterministes

Algorithme : TriNonDéterministe(T)

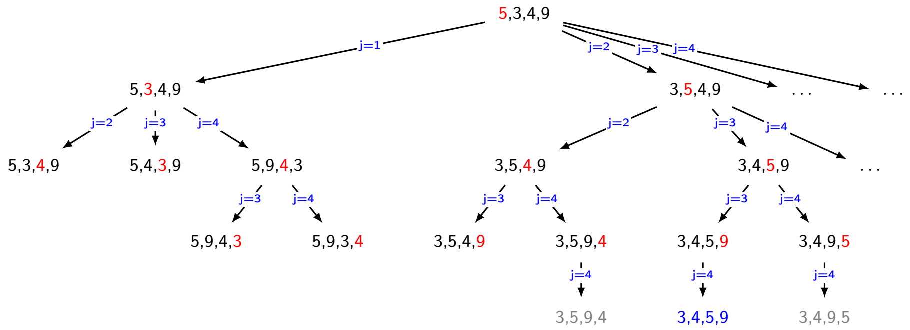
Données : un tableau T
Résultat : le tableau T trié

début
  pour *i allant de* 1 *à* $|T|-1$ **faire**
    Devine $j \in [i, .., |T|]$ t.q. $T[j]$ est minimal;
    Échange $T[i]$ et $T[j]$;
    si $i > 0$ *et* $T[i-1] > T[i]$ **alors stop;**

T :

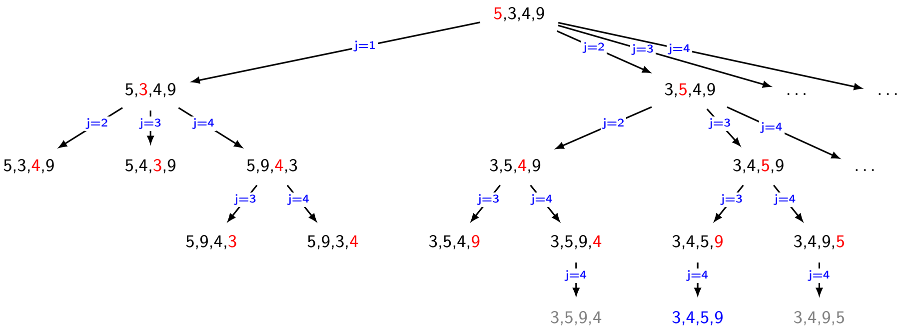
5	3	4	9

i : en rouge



Arbre de calcul non-déterministe

- On suppose que l'algorithme fait **toujours le bon choix** (ou explore tous les choix en parallèle)
- Complexité en temps = **longueur maximale d'une branche**
- TriNonDeterministe est en **temps linéaire** ($\mathcal{O}(n)$) puisqu'aucune branche ne peut avoir une longueur supérieure à n



La classe de problèmes NP

- On peut donc définir une classe de complexité en fonction d'une **ressource** (*temps*, *espace*, etc.), une **borne** ($n^c, 2^{n^c}$, etc.) et un **mode** (*déterministe*, *non-déterministe*)
- On se restreint aux problèmes de **décision** : réponse dans $\{\mathrm{oui}, \mathrm{non}\}$

P

Ensemble des problèmes pour lesquels il existe un algorithme **déterministe** en $\mathcal{O}(n^c)$ pour une constante c.

NP

Ensemble des problèmes de **décision** pour lesquels il existe un algorithme **non-déterministe** en $\mathcal{O}(n^c)$ pour une constante c.

- On se restreint aux **problèmes de décision**

Problème de décision Q

Fonction $Q : x \mapsto \{\mathbf{oui}, \mathbf{non}\}$

- Pour un problème dont la réponse n'est pas dans $\{\mathbf{oui}, \mathbf{non}\}$, on peut définir un problème **polynômialement** équivalent :

Voyageur de commerce (optimisation)

- **donnée** : ensemble de villes
- **question** : quel est le **plus court** chemin passant par toutes les villes ?

Voyageur de commerce (décision)

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il existe un chemin passant par toutes les villes de longueur inférieure à k ?

Problème d'optimisation, et sa version décision

Problème d'optimisation $Q = (X, f, m, g) : X \mapsto \bigcup_{x \in X} f(x)$

- X est un ensemble d'**instances**
- Pour tout instance $x \in X$, $f(X)$ est l'ensemble des solutions "faisables"
- $m : X \times f(X) \mapsto \mathbb{R}$ est une fonction "objectif"
- $g \in \{\min, \max\}$
- $Q(x) = arg_g\{m(x, y) \mid y \in f(x)\}$

Problème d'optimisation, et sa version décision

Problème d'optimisation $Q = (X, f, m, g) : X \mapsto \bigcup_{x \in X} f(x)$

- X est un ensemble d'**instances**
- Pour tout instance $x \in X$, $f(X)$ est l'ensemble des solutions "faisables"
- $m : X \times f(X) \mapsto \mathbb{R}$ est une fonction "objectif"
- $g \in \{\min, \max\}$
- $Q(x) = arg_g\{m(x, y) \mid y \in f(x)\}$

Exemple : Voyageur de commerce

- $x \in X$ est un ensemble de villes
- $y \in f(x)$ si et seulement si le chemin y passe par toutes les villes de x
- $m(x, y)$ est la longeur du chemin y (étant données les distances entre les villes de x)
- $g = \min$

Problème d'optimisation, et sa version décision

Problème d'optimisation $Q = (X, f, m, g) : X \mapsto \bigcup_{x \in X} f(x)$

- X est un ensemble d'**instances**
- Pour tout instance $x \in X$, $f(X)$ est l'ensemble des solutions "faisables"
- $m : X \times f(X) \mapsto \mathbb{R}$ est une fonction "objectif"
- $g \in \{\min, \max\}$
- $Q(x) = arg_g\{m(x, y) \mid y \in f(x)\}$

Problème de décision Q_D obtenu à partir de $Q = (X, f, m, \min)$

- **donnée** : $x \in X, k \in \mathbb{R}$
- **question** : Est-ce qu'il existe y tel que $f(x, y)$ et $m(x, y) \leq k$

Certificat polynômial

- On peut considérer des algorithmes non-déterministes de la forme suivante :

Algorithme non-déterministe polynômial

- L'algorithme **devine** une solution y ou répond "non"
  - $\simeq$ pour toute solution possible, il existe une branche de l'arbre de calcul y menant
- La solution y doit être **vérifiable** en temps polynomial : *certificat*
  - $\simeq$ La longueur de chaque branche est polynômiale, i.e., en $\mathcal{O}(n^c)$

- Méthode simple pour déterminer si un problème appartient à la classe $\mathbf{NP}$: est-ce que toute réponse "oui" peut-être accompagnée d'un certificat (la solution) vérifiable en temps polynomial

Certificat polynômial : exemple

- Méthode simple pour déterminer si un problème appartient à la classe **NP** : est-ce que toute réponse "oui" peut-être accompagnée d'un certificat (la solution) vérifiable en temps polynomial

Problème du voyageur de commerce

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il existe un chemin passant par toutes les villes de longueur inférieure à k ?

- Certificat : le chemin (permutation de villes)
  - Vérifier que toutes les villes sont visitées est facile
  - Calculer la longueur totale est facile (somme)

Le problème complément

- Le **complément** d'un problème de décision : inversion des réponses "oui"/"non"
  - Est-ce que le tableau T contient l'élement e ?
  - Est-ce que le tableau T **ne** contient **pas** l'élement e ?
- Le complément d'un problème dans $\mathbf{P}$ est aussi dans $\mathbf{P}$
  - Le même algorithme peut être utilisé

Co-problème du voyageur de commerce

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il **n'**existe **aucun** chemin passant par toutes les villes de longueur inférieure à k ?

- Ce problème est dans $\mathbf{NP}$, si et seulement si il existe un certificat polynômial, quel est le certificat polynômial ?

La classe coNP

Co-problème du voyageur de commerce

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il **n'**existe **aucun** chemin passant par toutes les villes de longueur inférieure à k ?

- Le co-problème du voyageur de commerce ne semble pas avoir de certificat polynômial
- Le complément d'un problème dans **NP** n'est souvent pas dans **NP** !

coNP

Ensemble des problèmes de décision dont le problème complément est dans **NP**

Inclusion dans EXPTIME

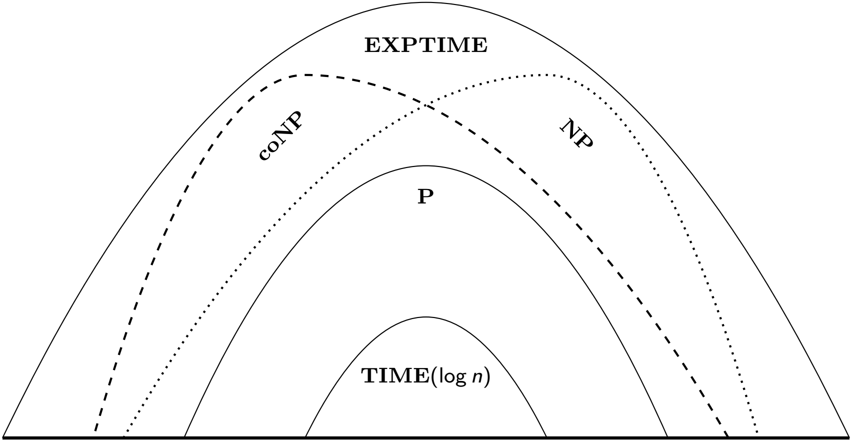
Considéront un problème dans **NP**

- La solution y doit avoir une taille polynômiale : $|y| \in \mathcal{O}(|x|^c)$
- L'arbre non-déterministe a donc au plus $\mathcal{O}(2^{|x|^c})$ feuilles
- Il est donc possible de les explorer séquentiellement en temps $\mathcal{O}(2^{|x|^c})$

Théorème

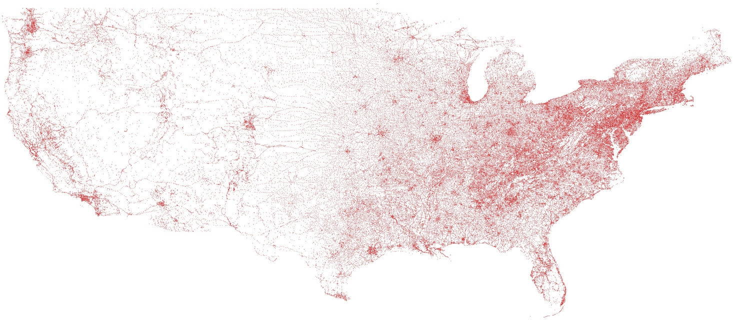
$$\mathbf{NP} \subseteq \mathbf{EXPTIME}$$

Résumé : les classes de complexité



L'importance de la classe NP

- Ces problèmes sont partout : en intelligence artificielle, en cryptographie, dans l'industrie...
- Savoir adapter la méthode à la complexité du problème ; le problème du voyageur de commerce n'a pas d'algorithme polynômial connu, mais...
  - Domaine de recherche très actif, des algorithmes "intelligent" peuvent résoudre (optimalement) de très grandes instances



115 475 villes

L'importance de la classe $\mathbf{NP}$ (suite)

Conjecture $\mathbf{P} \neq \mathbf{NP}$

- Un des 7 "problèmes du millénaire" du Clay Mathematics Institute
  Mise-à-prix 1 000 000 $

- Preuve de $\mathbf{P} \neq \mathbf{NP}$: un problème dans $\mathbf{NP}$ mais pas dans $\mathbf{P}$
  - Montrer que le problème est dans $\mathbf{NP}$ est facile : certificat polynômial
  - Montrer que le problème n'est pas dans $\mathbf{P}$ est difficile : tout algorithme est en $\Omega(2^n)$

- Dans le prochain cours, on verra comment prouver que $\mathbf{P} = \mathbf{NP}$

Certificat polynômial

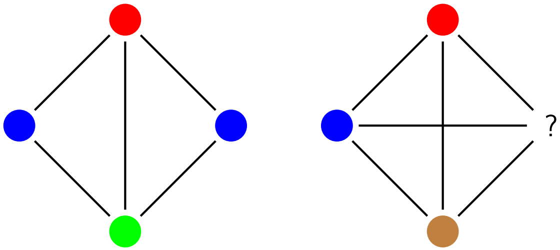
Rappel : Pour montrer qu'un problème fait partie de la classe $\mathbf{NP}$, il faut montrer qu'il existe un **certificat**

- De taille polynômiale (dans la TDLD)
- Vérifiable en temps polynômial (dans la TDLD) : algorithme

Exemple : Le problème de *3-coloration*

Donnée : Un graphe $G = (S, A)$ (sommets S ; arêtes A)

Question : Est-ce qu'il est possible de colorier les sommets de G avec au plus 3 couleurs en évitant que deux sommets adjacents partagent la même couleur.



3-Coloration est dans *NP*

Certificat : La coloration (tableau – couleur du i-ème sommet dans la case i)

- De taille polynômiale (dans la TDLD)
  - quelle est la taille de la donnée du problème ?
    taille : $|G| = \Theta(|S| + |A|)$
  - quelle est la taille du certificat ?
    taille : $\Theta(|S|)$
- Vérifiable en temps polynômial (dans la TDLD) : algorithme

Algorithme de vérification

Algorithme : verification(L, G)

Données : un graphe $G = (S, A)$ et un tableau T
Résultat : `vrai` si T est une 3-coloration de G, `faux`sinon
début

  pour *chaque arrête* (i, j) *de* A **faire**
    si $T[i] = T[j]$ **alors**
      retourner $faux$;
  retourner $vrai$;

Complexité : $\mathcal{O}(|A|)$ (liste d'adjacence)

$\Rightarrow$ *3-Coloration* est bien dans $\mathbf{NP}$

Théorème de Cook

Caractériser la difficulté

- L'appartenance à la classe **NP** est un indice de "facilité" (borne supérieure)
  - Un problème dans **NP** peut être résolu en temps polynômial "non-déterministe"
  - Pour un problème dans **NP**, il est facile de vérifier la réponse "oui"

- Comment peut-on caractériser la difficulté des problèmes de **NP** qui n'ont pas d'algorithme polynômial connu ? **Réduction ?**

Réduction polynômiale de A vers B (de Cook / de Turing) :

Un algorithme pour résoudre **A** en $\mathcal{O}(n^c)$ (pour c un constante et n la taille de la donnée) sous l'hypothèse que le problème **B** peut être résolu en $\Theta(1)$

- *Enveloppe convexe* peut se résoudre à condition de savoir résoudre *Tri*
  - *Enveloppe convexe* n'est pas beaucoup **plus difficile** que *Tri*
  - *Tri* est à peu près **aussi difficile que** *Enveloppe convexe*

- **Problème** : il n'y a pas de point d'appui ; les problème dans $\mathbf{NP}$ sont potentiellement tous dans $\mathbf{P}$

Théorème de Cook–Levin

SAT est $\mathbf{NP}$-complet

- Un problème est dit "**C**-complet" s'il est plus difficile que tous les problèmes de **C**
- "Plus difficile" est défini en utilisant une opération de **réduction**
- S'il existe un algorithme (déterministe) polynômial pour *SAT*, alors il en existe un pour **tous** les problèmes dans $\mathbf{NP}$!

Satisfiabilité booléenne (*SAT*)

Formule booléenne

- Ensemble fini de variable booléennes $\{x_1, \ldots, x_n\}$ qui peuvent prendre deux valeurs : $\mathbf{vrai}$ ou $\mathbf{faux}$
- Connecteurs logiques "non" ($\neg$) ; "ou" ($\vee$) ; "et" ($\wedge$)
- Une **formule booléenne** ϕ peut-être :
  - Une variable x
  - La **négation** $\neg\phi_1$ d'une formule booléenne ϕ_1
  - La **disjonction** $\phi_1 \vee \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2
  - La **conjonction** $\phi_1 \wedge \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2

- Une **interprétation** σ d'une formule booléenne ϕ sur les variables $\{x_1, \ldots, x_n\}$ est une fonction de $\{x_1, \ldots, x_n\}$ vers $\{\mathbf{vrai}, \mathbf{faux}\}$; on écrit $\sigma \vdash \phi$ pour "l'interprétation σ **satisfait** la formule ϕ", avec :
  - $\sigma \vdash x$ si et seulement si $\sigma(x) = \mathbf{vrai}$
  - $\sigma \vdash \neg\phi$ si et seulement si $\sigma \not\vdash \phi$
  - $\sigma \vdash \phi_1 \vee \phi_2$ si et seulement si $\sigma \vdash \phi_1$ ou $\sigma \vdash \phi_2$
  - $\sigma \vdash \phi_1 \wedge \phi_2$ si et seulement si $\sigma \vdash \phi_1$ et $\sigma \vdash \phi_2$

Satisfiabilité booléenne : exemple

Formule booléenne

- Ensemble fini de variable booléennes $\{x_1, \dots, x_n\}$ qui peuvent prendre deux valeurs : **vrai** ou **faux**
- Connecteurs logiques "non" ($\neg$) ; "ou" ($\vee$) ; "et" ($\wedge$)
- Une **formule booléenne** ϕ peut-être :
  - Une variable x
  - La **négation** $\neg\phi_1$ d'une formule booléenne ϕ_1
  - La **disjonction** $\phi_1 \vee \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2
  - La **conjonction** $\phi_1 \wedge \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2

$$((\text{“suivre le cours”} \wedge \text{“travailler”}) \vee \text{“tricher”} \vee \neg\text{“réussir l'exam”}) \quad \wedge$$
$$((\neg\text{“tricher”} \wedge \text{“réussir l'exam”}) \vee \neg\text{“passer en 4ème année”})$$

Satisfiabilité booléenne : exemple

Formule booléenne

- Ensemble fini de variable booléennes $\{x_1, \ldots, x_n\}$ qui peuvent prendre deux valeurs : **vrai** ou **faux**
- Connecteurs logiques "non" ($\neg$) ; "ou" ($\vee$) ; "et" ($\wedge$)
- Une **formule booléenne** ϕ peut-être :
  - Une variable x
  - La **négation** $\neg\phi_1$ d'une formule booléenne ϕ_1
  - La **disjonction** $\phi_1 \vee \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2
  - La **conjonction** $\phi_1 \wedge \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2

$$((\text{“suivre le cours”} \wedge \text{“travailler”}) \vee \text{“tricher”} \vee \neg\text{“réussir l'exam”}) \quad \wedge$$
$$((\neg\text{“tricher”} \wedge \text{“réussir l'exam”}) \vee \neg\text{“passer en 4ème année”})$$

	"suivre le cours"	"travailler"	"tricher"	"réussir l'exam"	"passer en 4ème année"
σ_1	**faux**	**faux**	**vrai**	**vrai**	**vrai**

Satisfiabilité booléenne : exemple

Formule booléenne

- Ensemble fini de variable booléennes $\{x_1, \ldots, x_n\}$ qui peuvent prendre deux valeurs : **vrai** ou **faux**
- Connecteurs logiques "non" ($\neg$) ; "ou" ($\vee$) ; "et" ($\wedge$)
- Une **formule booléenne** ϕ peut-être :
  - Une variable x
  - La **négation** $\neg\phi_1$ d'une formule booléenne ϕ_1
  - La **disjonction** $\phi_1 \vee \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2
  - La **conjonction** $\phi_1 \wedge \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2

$$((\text{“suivre le cours”} \wedge \text{“travailler”}) \vee \text{“tricher”} \vee \neg\text{“réussir l'exam”}) \quad \wedge$$
$$((\neg\text{“tricher”} \wedge \text{“réussir l'exam”}) \vee \neg\text{“passer en 4ème année”})$$

	"suivre le cours"	"travailler"	"tricher"	"réussir l'exam"	"passer en 4ème année"
σ_1	**faux**	**faux**	**vrai**	**vrai**	**vrai**
σ_2	**vrai**	**vrai**	**faux**	**vrai**	**vrai**

Satisfiabilité booléenne : exemple

Formule booléenne

- Ensemble fini de variable booléennes $\{x_1, \ldots, x_n\}$ qui peuvent prendre deux valeurs : **vrai** ou **faux**
- Connecteurs logiques "non" $(\neg)$; "ou" $(\vee)$; "et" $(\wedge)$
- Une **formule booléenne** ϕ peut-être :
  - Une variable x
  - La **négation** $\neg\phi_1$ d'une formule booléenne ϕ_1
  - La **disjonction** $\phi_1 \vee \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2
  - La **conjonction** $\phi_1 \wedge \phi_2$ de deux formules booléennes ϕ_1 et ϕ_2

$$((\text{“suivre le cours”} \wedge \text{“travailler”}) \vee \text{“tricher”} \vee \neg\text{“réussir l'exam”}) \quad \wedge$$
$$((\neg\text{“tricher”} \wedge \text{“réussir l'exam”}) \vee \neg\text{“passer en 4ème année”})$$

	"suivre le cours"	"travailler"	"tricher"	"réussir l'exam"	"passer en 4ème année"
σ_1	**faux**	**faux**	**vrai**	**vrai**	**vrai**
σ_1	**vrai**	**vrai**	**faux**	**vrai**	**vrai**
σ_3	**vrai**	**vrai**	**faux**	**faux**	**faux**

Satisfiabilité booléenne : le problème de décision

SAT

- **donnée** : Une formule booléenne ϕ
- **question** : Est-ce qu'il existe une interpretation satisfaisant ϕ ?

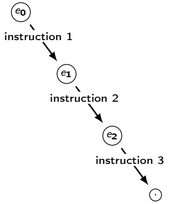
- *SAT* est dans $\mathbf{NP}$, le certificat est l'interpretation σ
  - $|\sigma| \in \mathcal{O}(|X(\phi)|)$ où $X(\phi)$ est l'ensemble de variables qui apparaissent dans ϕ
  - On peut vérifier le certificat en temps $\mathcal{O}(|\phi|)$ en utilisant la définition récursive de $\vdash$

(Horn) SAT est $\mathbf{P}$-complet

Représentation d'un algorithme par une formule

- Un algorithme **déterministe** peut se coder par une formule SAT ϕ
  - État de la mémoire / position dans le code représenté par des variables
  - Les instructions sont représentées par des formules logiques
- La solution de la formule donne la réponse y (et l'état de la mémoire à tout instant)
- Espace mémoire, temps et nombre d'instructions en $\mathcal{O}(|x|^c)$ donc $|\phi| \in \mathcal{O}(|x|^{c'})$

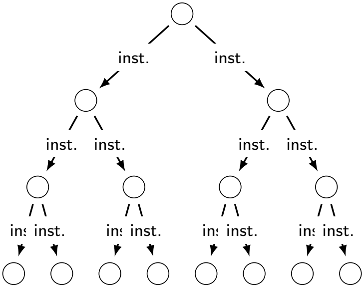
temps	mémoire			curseur		
t_1	x_1^1	$\dots$	x_1^m	z_1^1	$\dots$	z_1^c
t_2	x_2^1	$\dots$	x_2^m	z_2^1	$\dots$	z_2^c
t_3	x_3^1	$\dots$	x_3^m	z_3^1	$\dots$	z_3^c
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
t_n	x_n^1	$\dots$	x_n^m	z_n^1	$\dots$	z_n^c



- Peut-on représenter un algorithme non-déterministe par une formule booléenne ?

Cas non-déterministe

- Pour un arbre de calcul **non-déterministe**, il existe un arbre binaire équivalent
- Une branche correspond à un algorithme déterministe (donc représentable par une formule)
- Toutes les branches partagent le même code et travaillent sur la même mémoire, seul le **choix** d'instruction change
- On rajoute **une variable pour chaque niveau** dans l'arbre pour représenter ce choix



Problème $\mathbf{NP}$-complet

Théorème de Cook-Levin

SAT est $\mathbf{NP}$-complet

- On ne sait toujours pas si $\mathbf{P} \neq \mathbf{NP}$: SAT pourrait être polynômial
- Mais *SAT* est "plus difficile" que **tous** les problèmes dans NP
  - S'il existe un algorithme polynômial pour résoudre *SAT* alors il en existe pour tous les problèmes dans $\mathbf{NP}$
- Point d'appui pour montrer une certaine difficulté :

Réduction

Si on peut **réduire** *SAT* à un problème **A**, alors **A** est "au moins aussi difficile" que *SAT*, et donc $\mathbf{NP}$-complet

Théorème de Cook-Levin

SAT est **NP**-complet

- S'il existe un algorithme polynômial pour résoudre *SAT* alors il en existe pour tous les problèmes dans **NP**

Réduction

S'il existe un algorithme polynômial pour *SAT*, alors $\mathbf{P} = \mathbf{NP}$

Contre-exemple pour la conjecture $\mathbf{P} \neq \mathbf{NP}$, et donc prix de 1 000 000 $

Réduction de Karp

Réduction de Cook / Turing

Réduction polynômiale de A à B (de Cook / de Turing) :

Un algorithme pour résoudre **B** en $\mathcal{O}(n^c)$ (pour c un constante et n la taille de la donnée) sous l'hypothèse que le problème **A** peut être résolu en $\Theta(1)$

- Si on peut résoudre **A** en résolvant **B** $\mathcal{O}(n^c)$ fois, alors **B** est "plus difficile" : un algorithme pour **B** sera au mieux $\mathcal{O}(n^c)$ fois plus rapide qu'un algorithme pour **A**

- Malheureusement, $\mathbf{NP}$ n'est pas fermé par réduction de Cook : il est possible de réduire un problème **A** $\notin \mathbf{NP}$ à **B** $\in \mathbf{NP}$

Réduction polynômiale de *coTSP* à *TSP*

TSP

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il existe un chemin passant par toutes les villes de longueur inférieure à k ?

coTSP

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il **n'**existe **aucun** chemin passant par toutes les villes de longueur inférieure à k ?

- Supposons qu'il existe une algorithme qui résout *TSP* en $\mathcal{O}(1)$
- On peut résoudre *coTSP* en appelant cet algorithme et en inversant le résultat
- Mais *coTSP* $\notin \mathbf{NP}$!

- On a donc besoin d'un type de réduction plus restrictif

Réduction de Karp de A à B

Algorithme $\in \mathbf{P}$ qui pour toute donnée α du problème **A** renvoie β tel que **B**$(\beta) = \text{oui} \iff$ **A**$(\alpha) = \text{oui}$

- Exemple : réduction de *SAT* au cas particulier *CNF-SAT*

CNF-SAT

- **donnée** : Une formule booléenne ϕ en **forme normale conjonctive**
- **question** : est-ce que ϕ est satisfiable ?

- **Litéral** : variable ou sa négation (x, $\neg x$)
- **Clause** : disjonction de litéraux $(x \vee \neg y \vee \neg z)$
- **CNF** : conjonction de clauses $((x \vee \neg y \vee \neg z) \wedge (\neg x \vee y))$

Réduction de *SAT* à *CNF-SAT*

Litéral : variable ou sa négation $(x, \neg x)$

Clause : disjonction de litéraux $(x \vee \neg y \vee \neg z)$

CNF : conjonction de clauses $((x \vee \neg y \vee \neg z) \wedge (\neg x \vee y))$

- On peut transformer une disjonction de conjonctions en conjonction de disjonctions équivalente en distribuant :

$$(a \wedge b \wedge c) \vee (x \wedge y) \iff (a \vee x) \wedge (b \vee x) \wedge (c \vee x) \wedge (a \vee y) \wedge (b \vee y) \wedge (c \vee y)$$

- Mais la transformation peut produire un nombre exponentiel de clauses
- Solution : utiliser des **variables additionnelles**

- Une variable $D = \mathbf{vrai}$ si et seulement si $(a \wedge b \wedge c) = \mathbf{vrai}$
- Une variable $Z = \mathbf{vrai}$ si et seulement si $(x \wedge y) = \mathbf{vrai}$
- Une nouvelle clause $D \vee Z$

$$\begin{aligned}(\neg a \vee \neg b \vee \neg c \vee D) \quad &\wedge \\ (a \vee \neg D) \quad &\wedge \\ (b \vee \neg D) \quad &\wedge \\ (c \vee \neg D) \quad &\end{aligned}$$

Preuve de $\mathbf{NP}$-complétude

Pour montrer qu'un problème est $\mathbf{NP}$-complet, on doit :

1. Montrer qu'il appartient à $\mathbf{NP}$ (certificat polynômial)
   - L'**interpretation** σ de la formule ϕ est un certificat polynômial (cf. définition de $\sigma \vdash \phi$)
2. Montrer qu'il est "$\mathbf{NP}$-difficile", i.e., il existe une réduction de Karp d'un problème $\mathbf{NP}$-complet vers notre problème
   - On vient de montrer que pour toute formule de *SAT*, il existe une formule équivalente de *CNF-SAT*

Attention !

L'algorithme qui produit la formule équivalente doit être **polynômial**, et donc en particulier il doit produire une formule de **taille** polynômiale

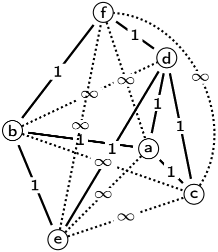
Réduction par généralisation

TSP (voyageur de commerce)

- **donnée** : ensemble de villes, entier k
- **question** : est-ce qu'il existe un chemin passant par toutes les villes de longueur inférieure à k ?

Chemin Hamiltonien

- **donnée** : Un graphe G
- **question** : est-ce qu'il existe un chemin passant une et une seule fois par chaque sommet de G ?



- *TSP* est une **généralisation** de *Chemin Hamiltonien*
- Si *Chemin Hamiltonien* est $\mathbf{NP}$-complet alors *TSP* est $\mathbf{NP}$-complet

Réduction de *CNF-SAT* à *3SAT*

- *3SAT* : cas particulier de *CNF-SAT* où les clauses ont au plus 3 litéraux
- Essayons de réduire *CNF-SAT* à *3SAT*

"Gadget"

- Notion de gadget : reformulation d'un partie du problème réduit (*CNF-SAT*) dans une donnée du type accepté par le problème cible (*3SAT*)

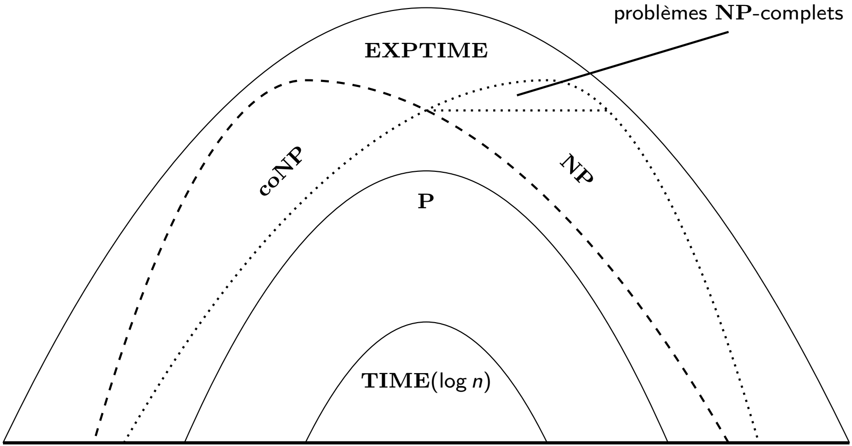
- On peut transformer une clause de taille de taille n en deux clauses, une de taille 3 et une de taille $n-1$. Soit R une dijonction d'un nombre quelconque de litéraux :

$$(a \vee b \vee R) \iff (a \vee b \vee x) \wedge (\neg x \vee R)$$

  Où x est une nouvelle variable

- On peut répéter ce gadget jusqu'à ce qu'il n'y ait plus de clause de taille 4 ou plus

Résumé : les classes de complexité



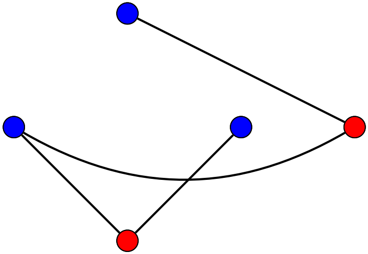
Réduction par équivalence

Stable

donnée : Un graphe $G = (S, A)$, un entier k.

question : G contient-il un stable de cardinalité k ou plus ?
C'est-à-dire existe-t'il $I \subseteq S$ tel que :

1. $\forall i, j \in I, (i, j) \notin A$
2. $|T| \geq k$

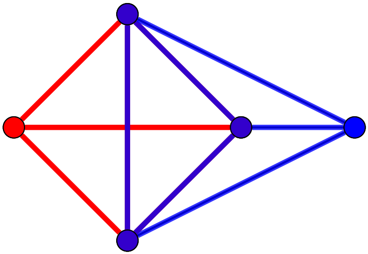


Clique

donnée Un graphe $G = (S, A)$, un entier k.

question G contient-il une clique de cardinalité k ou plus ?
C'est-à-dire existe-t'il $C \subseteq S$ tel que :

1. $\forall i, j \in C, (i, j) \in A$
2. $|C| \geq k$

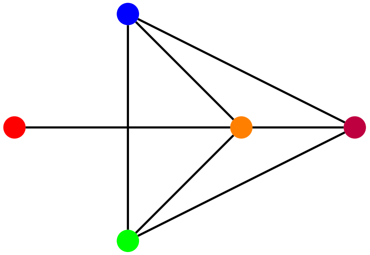


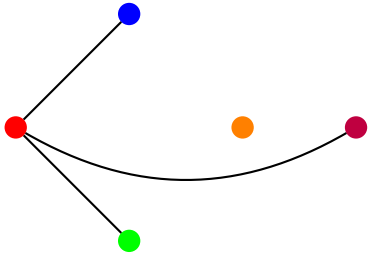
Instance de *Clique*

- un graphe $G = (S, A)$
- un entier k

Instance *Stable*

- $G' = (S, \overline{A})$ (graphe complémentaire de G)
  - $A' = \{(x, y) : (x, y) \notin A\}$
- k' = k





Complexité : $O(|S|^2)$ (matrice d'adjacence)

Problèmes NP-complets (jusqu'à maintenant)

$$
\begin{tikzcd}
\textit{Clique} \arrow[d, leftrightarrow] & \textit{Chemin Hamiltonien} & \boxed{\textit{SAT}} \\
\textit{Stable} & \textit{TSP} \arrow[u] & \textit{CNF-SAT} \arrow[u] \\
 & & \textit{3SAT} \arrow[u]
\end{tikzcd}
$$

$\uparrow$ "réduction depuis", i.e., au moins aussi difficile que