Chapitre V

Complexité temporelle

Introduction

On se concentre maintenant sur des problèmes décidables ou calculables (à savoir à des algorithmes qui s'arrêtent toujours), à savoir :

- pour une MTD, elle s'arrête toujours en acceptant ou en rejetant.
- pour une MTND, toutes les branches d'exécutions s'arrêtent toujours en acceptant ou en rejetant.

Donc, on ne considère plus que les problèmes pour lesquels il est possible d'écrire des algorithmes qui les **décident**.

On s'intéresse au **temps** nécessaire pour résoudre ces problèmes.

On va donc maintenant introduire :

- faire un rappel sur les asymptotiques,
- se demander comment mesurer le temps d'exécution d'un programme sur une entrée,
- définir le comportement asymptotique du temps d'exécution,
- la notion de complexité temporelle.
- les différentes classes de complexité (**P**, **NP**, ...) ainsi que les propriétés qui leurs sont associées.

1 Asymptotiques

Soit une fonction $f(n)$ qui décrit le comportement d'un certain phénomène en fonction d'un paramètre n. Par exemple, dans notre cas, $f(n)$ serait le temps qu'un programme met pour trier un tableau d'éléments.

Ce temps dépend non seulement de la taille du tableau, mais du tableau lui-même (par exemple, ce dernier peut être partiellement trié, et cette caractéristique rend le tri plus rapide).

L'idée d'asymptotique est de s'intéresser au comportement de la fonction f quand n devient plus grand, en se débarrassant des caractéristiques de f (par exemple, des oscillations) qui gênent l'analyse, et en la comparant à une autre fonction g exempte de ces scories.

Par exemple :

- f et g sont du même ordre de grandeur.
- majorer f par g : f toujours moins grande que g.
- minorer f par g : f toujours plus grande que f.
- f négligeable devant g.

1.1 Majoration asymptotique (grand O)

Cette partie est un rappel. Voir le début du chapitre précédent pour les exemples.

Définition V.1 (grand O)

Soit f et g des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$.
$f(n) = O(g(n))$ si $\exists c > 0, n_0 \in \mathbb{N}^* \mid \forall n \geq n_0, f(n) \leq c.g(n)$
se lit : g est une borne asymptotique supérieure pour f.
se comprend : f ne grandit pas plus vite que g.

Cette définition s'interprète de la manière suivante :

- On borne supérieurement une fonction $f(n)$ par une fonction $c.g(n)$.
- La constante c ne dépend pas de n. Il n'y a aucune contrainte sur son ordre de grandeur (pourrait être 10^9).
- Le n_0 est le n à partir duquel la relation $f(n) \leq c.g(n)$ est vérifiée.
  On tient compte ainsi de la tendance générale quand n devient assez grand.

1.2 Domination asymptotique (petit o)

Définition V.2 (petit o)

On écrit $f(n) = o(g(n))$ si :
$$\forall \varepsilon > 0, \exists n_0, \forall n > n_0, f(n) \leq \varepsilon g(n).$$
se lit : f est dominée asymptotiquement par g.
se comprend : f est négligeable devant g.

La notation o permet donc d'indiquer des termes qui peuvent être négligés car trop petit devant le terme principal.

Une autre façon d'interpréter cette notation est :
$$f(n) = o\left(g(n)\right) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

À savoir, pour n assez grand, $f(n)$ est négligeable devant $g(n)$.

Exemple 55: $f(n) = \log(n)$ est dominée par $g(n) = n$. On a bien $\lim_{n\to\infty} \frac{\log n}{n} = 0$.

2 Référentiel de mesure temporelle

On remarque que :

- sur une machine de Turing, en comptant le nombre de transitions,
- sur un processeur classique, en nombre de cycles ou d'instructions du processeur.
  par exemple, en utilisant un compteur haute résolution, ou un outil de profiling.

Mais comment avoir une mesure qui représente quelque chose ?

Cela dépend sans doute :

- du type de machine utilisée (variation de machines de Turing, type de processeur, architecture, ...)

- du fait que la machine soit déterministe ou pas (comment compter les transitions ?)

Étudions ce problème sur quelques exemples.

2.1 Temps d'exécution sur une machine déterministe

Donnons une première définition du temps d'exécution.

Définition V.3 (Temps d'exécution d'une machine de Turing M sur une entrée w)

Le **temps d'exécution** de $M(w)$ est le nombre de transitions nécessaires à M pour traiter la chaîne d'entrée w.

Noter que ce résultat est seulement valide pour l'algorithme M appliqué sur l'entrée w.

Lors du comptage du nombre de transitions, on utilisera des majorations asymptotiques. A savoir, lorsqu'on indique qu'une étape prend $O(n)$ transitions pour réaliser une tache sur un mot de taille n, cela signifie donc que cette tache nécessite au plus p parcours de la bande (où p ne dépend pas de n ; par exemple $p = 4$).

La majoration asymptotique nous permettra donc de donner l'ordre de grandeur du pire temps pour l'exécution de la tache.

Exemple 56: Soit le langage $A = \{0^k 1^k | k \geq 0\}$
Ce langage est décidable.
Une MT simple bande qui décide A est :

$M_1(\langle w \rangle) =$

```
1 pour chaque cellule de la bande
2 |   si on trouve un 0 à droite d'un 1 alors rejeter
3 tant que il reste des 0 sur la bande
4 |   barrer un 0
5 |   si il reste un 1 alors le barrer sinon rejeter
6 si il reste des 1 alors rejeter sinon accepter
```

Exécutions :

w	00001011
1-2	rejeter

w	00011111
1-2	00011111
3-4	x00x1111
3-4	xx0xx111
3-4	xxxxxx11
5	rejeter

w	00001111
1-2	00001111
3-4	x000x111
3-4	xx00xx11
3-4	xxx0xxx1
3-4	xxxxxxxx
5	accepter

Quel est le temps mis par la MT M_1 pour décider si son entrée $w \in A$?
Pour cette MT,

- ce temps se mesure en nombre de transitions (= de pas).
- ce dernier dépend de la longueur n de l'entrée w ($n = |w|$).

Analyse de M_1 :

1-2 : prend au plus $O(n)$ pas.
3-5 : répétition au plus $n/2$ fois de $O(n)$ pas $\Rightarrow O(n^2)$.
6 : prend $O(n)$ pas.

Donc, M_1 prend $O(n^2)$ pas pour décider si $w \in A$.

Peut-on trouver une autre MT qui décide le même langage plus rapidement ?

Exemple 57: Soit le code suivant d'une machine de Turing :

$M_2(\langle w \rangle) =$

```
1 pour chaque cellule de la bande
2 |   si on trouve un 0 à droite d'un 1 alors rejeter
3 tant que il reste des 0 sur la bande
4 |   si le nombre de 0 et de 1 restant est impair alors rejeter
5 |   barrer un 0 sur deux
6 |   si il reste un 1 alors barrer un 1 sur deux sinon rejeter
7 si il reste des 1 alors rejeter sinon accepter
```

Exécutions :

w	00001111
1-2	00001111
3-6	x0x0x1x1
3-6	xxxxxxxx
7	accepter

w	000000111111
1-2	000000111111
3-6	x0x0x0x1x1x1
3-6	xxx0xxxxx1xx
3-6	xxxxxxxxxxxx
7	accepter

w	00111111
1-2	00111111
3-6	x0x1x1x1
3-6	xxxxx1xx
7	rejeter

Pour 0^k1^k, à chaque exécution de la boucle 3,

- pour M_1, un seul 0 et un seul 1 sont barrés.
  $\Rightarrow$ la boucle ligne 3 est exécutée k fois.
- pour M_2, le nombre de 0 et de 1 est divisé par 2.
  $\Rightarrow$ la boucle ligne 3 est exécutée $\log_2 k$ fois.
  Rappel : doubler k fois $\Rightarrow 2^k$ fois plus grand.

Analyse de M_2 : $n = |w|$

1-2 : prend au plus $O(n)$ pas.
3-6 : répétition au plus $O(\log n)$ fois de $n/2$ pas $\Rightarrow O(n \log n)$.
7 : prend $O(n)$ pas.

Le temps d'exécution de M_2 est $O(n \log n)$.

Peut-on trouver une autre MT qui s'exécute encore plus rapidement en utilisant plusieurs bandes ?

Exemple 58: (version multibande de $A = \{0^k1^k | k \geq 0\}$) Avec une MT à deux bandes :

$M_3(\langle w \rangle) =$

```
1 pour chaque cellule de la bande 1
2 |   si on trouve un 0 à droite d'un 1 alors rejeter
3 pour chaque cellule de la bande 1 contenant un 0
4 |   déplacer le 0 de la bande 1 vers la bande 2
5 pour chaque cellule de la bande 1 contenant un 1
6 |   barrer le 1 sur la bande 1
7 |   si il reste un 0 sur la bande 2 alors le barrer sinon rejeter
8 si il reste des 0 sur la bande 2 alors rejeter sinon accepter
```

Exécutions :

	bande 1	bande 2
w	00001111	________
1-2	00001111	________
3-4	____1111	0000____
5-7	____xxxx	xxxx____
8	accepter	

	bande 1	bande 2
w	00000111	________
1-2	00000111	________
3-4	_____111	00000___
5-7	_____xxx	00xxx___
8	rejeter	

Analyse de M_3 :

1-2 : prend au plus n pas.
3-4 : prend au plus n pas.
5-7 : prend au plus n pas.
8 : prend au plus n pas.

Le temps d'exécution de M_3 est en $O(n)$.
Si la chaîne est acceptée, il n'est évidemment pas possible de faire plus rapide (puisqu'il faut lire l'entrée et que celle-ci est de taille n).

Conclusion :

Les MTs simple bande et multi-bandes ont donc :

- la même puissance en terme de calculabilité.
  ils peuvent résoudre les mêmes problèmes.
- une puissance différente en terme de complexité.
  ils ne les résolvent pas à la même vitesse.

Il faut standardiser les mesures. On utilisera donc le modèle standard simple bande d'une MT pour calculer les vitesses d'exécution.

On obtient donc comme mesure de référence de temps pour un algorithme déterministe (codé dans la machine de Turing M) :

Définition V.4 (Temps d'exécution déterministe d'un algorithme sur une entrée)

Le **temps d'exécution déterministe** de $M(w)$ est le nombre de transitions nécessaires à pour une machine de Turing déterministe M simple bande codant l'algorithme pour traiter la chaîne d'entrée w.

Mais j'ai calculé la vitesse de mon algorithme sur une MT multibande, comment connaître la vitesse équivalente sur une MT classique ?

Théorème V.1 (Pénalité engendrée par l'utilisation d'une MT multibande)

Soit $t(n)$ une fonction telle que $t(n) \geq n$.
Pour toute MT à k-bandes qui s'exécute en temps $t(n)$ sur une entrée de longueur n, il existe une MT à une bande qui s'exécute en temps $O(k^2 t(n)^2)$.

Démonstration:

Soit M_k une MT à k bandes qui s'exécute en temps $t(n)$.
Construisons une MT à une bande qui s'exécute en temps $O(k^2 t(n)^2)$.
On a vu comment simuler M_k sur M :

- M stocke sur sa bande les k bandes de M_k en les séparant par #.
- position du pointeur sur une bande = symbole marqué (une marque par bande).

Simulation de M comme M_k :

- parcourir la bande pour lire les caractères sous chaque pointeur.
- parcourir la bande pour mettre à jour le caractère et la position de chaque pointeur.
- si l'on dépasse l'extrémité, on ajoute un espace en décalant le contenu de la bande.

Portion active des bandes :

- M_k s'exécute en temps $t(n)$, chacune de ses bandes accèdent **au plus** les $t(n)$ premières cellules.
- M utilise donc **au plus** les $k \times t(n) + k + 1 = O(kt(n))$ premières cellules.
  $k + 1$ = les séparateurs de bandes.

M fait **à chaque pas** (dans le pire des cas) :

1. parcourir la bande pour lire les caractères sous chaque pointeur.
   prend au plus un temps $O(k.t(n))$ où $t(n)$ est le temps d'exécution.
2. ajouter un espace à chaque bande
   un ajout de un espace = un parcours de la bande en temps $O(kt(n))$.
   au pire, k espace sont ajoutés.
3. mettre à jour la position de chaque pointeur et du caractère pointé.
   prend au plus un temps $O(kt(n))$.

Comme ceci est réalisé $O(kt(n))$ fois
$\Rightarrow$ l'exécution de M se fait en temps $O(k^2 t(n)^2)$. □

Exemple 59: de conversion de temps d'exécution Si un programme met un temps $t(n) = O(n)$ sur une machine à $k = 2$ bandes, ce programme mettra un temps $O(2^2 O(n)^2) = O(n^2)$ sur une MT à une bande.

2.2 Temps d'exécution sur une machine non déterministe

Comment définir le temps d'exécution dans le cas d'une machine de Turing non-déterministe ?

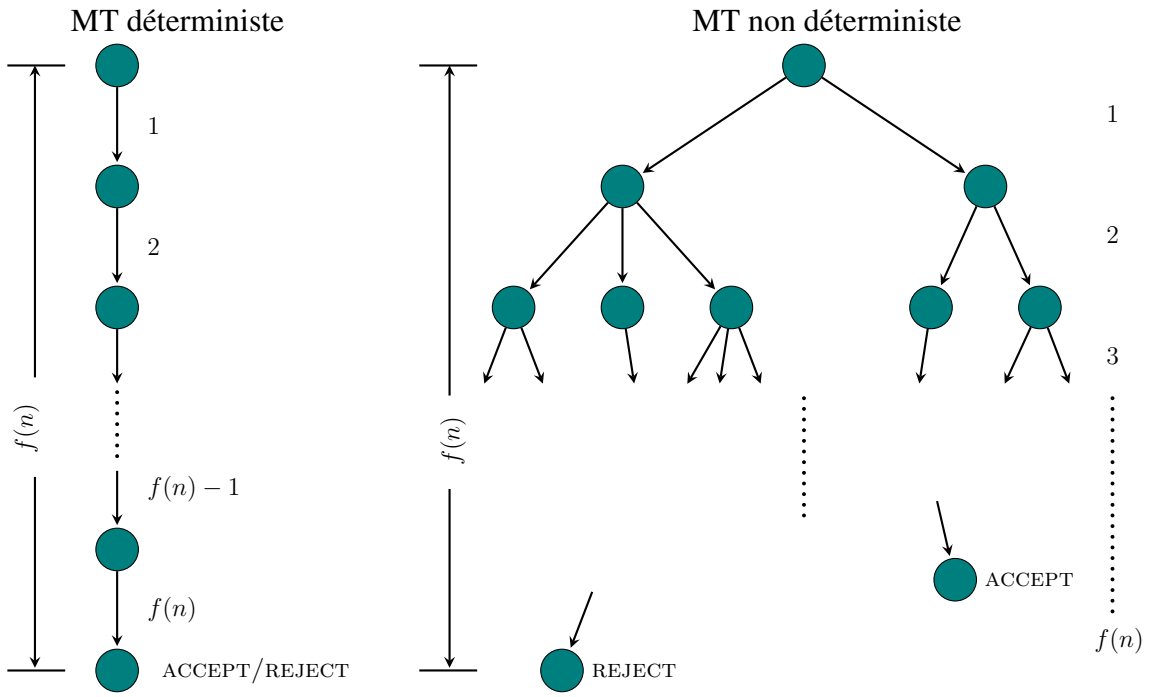
Définition V.5 (Décideur dans le cas d'une MTND (rappel))

une MTND M est un décideur si l'évaluation de toutes ses branches s'arrête pour toutes les entrées.
$\Rightarrow$ M décide $\mathcal{L}(M)$.

Définition V.6 (Temps d'exécution d'une MTND)

le **temps d'exécution** t de M sur l'entrée w est le nombre maximum de transitions que M traverse dans n'importe laquelle de ses branches de calcul sur l'entrée w.
t est le temps d'exécution de la branche la plus longue de $M(\langle w \rangle)$.

Comparaison de la complexité temporelle : ci-dessous, on note n la longueur de l'entrée w et $f(n)$ le temps d'exécution mesuré.



Mais comment comparer un temps d'exécution obtenu sur une machine de Turing non déterministe à celui d'une machine déterministe ?

Théorème V.2 (Pénalité engendrée par l'utilisation d'une MT non déterministe)

Soit $t(n)$ une fonction telle que $t(n) \geq n$. Pour toute MT non déterministe M' qui s'exécute en temps $t(n)$, il existe une MT déterministe M à une bande qui s'exécute en temps $2^{O(t(n))}$.

Démonstration:

Pour M', chaque branche à un temps de calcul au plus $t(n)$. Chaque nœud de M' a au plus b fils (= nb maximum de choix de transitions). Donc, le nombre de feuilles de M' est au plus de $O(b^{t(n)})$. Le temps de traitement d'une branche est au plus de $O(t(n))$. L'ordre de parcours des nœuds importe peu : dans le cas le pire, on passe dans toutes les branches. Donc, le temps total de simulation de M' par M est en temps $O(t(n)b^{t(n)})$. Or $O(t(n)b^{t(n)}) = O(e^{t(n)\log b}) = O(2^{t(n)\log b/\log 2}) = 2^{O(t(n))}$. □

Les MT non déterministes, s'il était possible de les réaliser physiquement, permettrait donc un gain exponentiel par rapport aux MTs classiques.

3 Temps d'exécution d'un algorithme

3.1 Définition

On veut maintenant une mesure de la performance de l'algorithme M indépendamment de son entrée.

Comment faire ?

Il faut avoir une mesure de la taille n du problème que cet algorithme va traiter.

Exemples 60:

- opération sur un tableau de p éléments : $n = p$
  Exemple : recherche du max, tri, ...
- opération sur un matrice de $p \times p$ éléments : $n = p \times p$
  Exemple : transposition, inversion, ...
- opération sur un graphe non orienté de p nœuds : plus compliqué dans ce cas.
  Exemple : recherche d'un chemin entre deux sommets, recherche d'un chemin Hamiltonien, ...
  Il y a au minimum p arêtes et au maximum p^2.
  Clairement, la complexité va aussi dépendre du nombre d'arêtes.
  On prend $n = p + p \times p$.
- ...

Dans tous les cas ci-dessus, la taille du problème à traiter est en fait la taille du codage de l'entrée.

Relativement intuitif : plus la quantité de données mise en entrée d'un algorithme est importante, plus le temps de traitement devrait être important.

Si ce n'est pas le cas, cela signifie qu'une partie des données mise en entrée de l'algorithme n'est pas nécessaire à son exécution.

Exemple 61: Pour un algorithme M de calcul du nombre de sommets dans un graphe $\langle V, E\rangle$, il est inutile de passer la liste des arêtes $\langle E\rangle$.
On exécute $M(\langle V\rangle)$ en ne passant que la liste des sommets.

On définit donc le temps d'exécution d'un algorithme de la manière suivante :

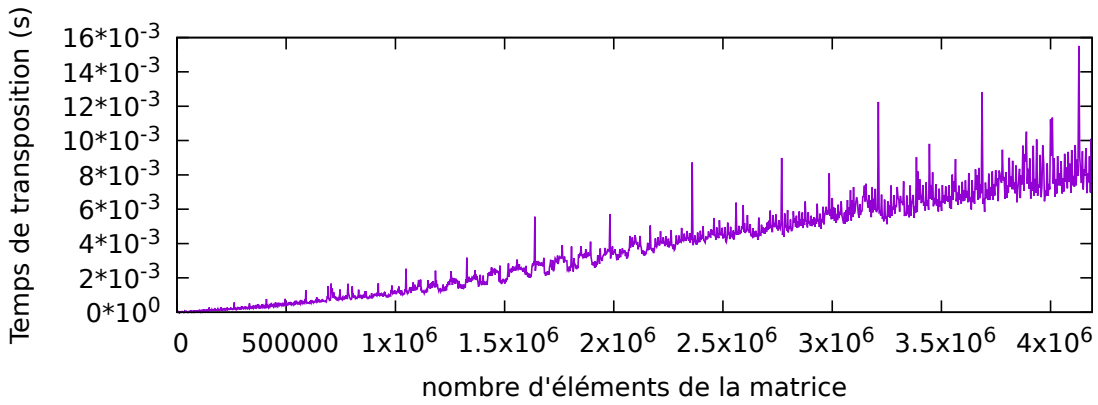
Définition V.7 (Temps d'exécution d'un algorithme M)

Le **temps d'exécution** de M est la fonction $f : \mathbb{N} \to \mathbb{N}$ où $f(n)$ est le nombre maximum de transition (resp. nombre de cycles) nécessaires à M pour traiter une chaîne d'entrée (resp. entrée) de longueur n.

$f(n)$ représente donc le temps **au pire** qui sera nécessaire pour résoudre un problème un problème de taille n (aucune entrée de taille n ne passe plus de temps).

Remarques : on dit alors que l'algorithme M s'exécute en temps $f(n)$ (ou M est à temps $f(n)$).

A noter que les temps d'exécution sur un ordinateur sont souvent perturbés par des mécanismes internes (nombre de registres, mémoire locale sur le CPU ...) et externes (nombre de niveaux de cache, taille des caches, taille des blocs, ...).



L'exemple ci-dessus présente le temps d'exécution de la transposition d'une matrice $p \times p$ en fonction de n.

On constate que :

- la progression est approximativement linéaire (en fonction de la taille de l'entrée $n = p^2$).
- il existe certaines tailles de matrice "maudites" : il y a, à intervalle régulier, des pics correspondants à des cas où les caches font défaut en cascade.

3.2 Comportements asymptotiques

Comme le temps d'exécution $f(n)$ d'un algorithme peut être très irrégulier, on s'intéresse au comportement asymptotique de cette fonction.

Les **ordres de grandeur** asymptotiques les plus courants sont les suivants :

Grand O	vitesse de croissance	exemple
1	constante	10
$\log\log n$	loglogarithmique	
$\log n$	logarithmique	$\log(n^2)$
$\log(n)^c, c > 1$	polylogarithmique	$\log(n)^2$
$n^c, c \in (0,1)$	puissance fractionnaire	$n^{1/2}$
n	linéaire	$3n$
$n \log n$ $\log(n!)$	quasi-linéaire	
n^2	quadratique	$2n^2$
$n^c, c > 1$	polynomiale	$n^{3/2}$
e^n $c^n, c > 1$	exponentielle	2^n
$e^{n^c}, c > 1$ n^n	(poly)exponentielle	$2^{n^2}, n^n$ $n!$

Note : formule de Stirling $n! \sim \sqrt{2\pi n}(n/e)^n$

Exemples d'ordres asymptotiques communs :

- recherche dans une table de correspondance : 1
- recherche dans un arbre binaire équilibré : $\log n$
- recherche séquentielle : n
- tri : naïf = n^2, à bulle = $n \log n < n^2$
- voyageur de commerce : naïf = $n!$, mais possiblement 2^n

Exemple possiblement contre-intuitif :

Considérons la fonction d'addition de deux entiers suivantes :

$\textsc{SillyAdd}(\langle a, b\rangle) =$

```
r=a
POUR i=1 à b : r = r + 1
RETOURNER r
```

Comme déjà indiqué, on utilise un codage raisonnable des entiers a et b non signés. Prenons ici le binaire.

Soit $n = |\langle a, b\rangle|$. En négligeant le séparateur, supposons $n = |\langle a\rangle| + |\langle b\rangle|$. Le maximum de boucles est obtenu pour $|\langle a\rangle| = 0$ et $|\langle b\rangle| = n$.

Donc, si $n = 32$, on est assuré que l'algorithme effectuera $2^{32} - 1 \simeq 4.10^9$ de boucles au pire (nombre maximum de transitions).

A savoir, une entrée de taille n engendre au pire 2^n boucles, soit un nombre exponentiel par rapport à l'entrée. Noter ici que l'on ne compte même pas le coût de l'incrémentation. **L'ordre asymptotique de cet algorithme est exponentiel.**

Évidemment, une addition classique (comme vue dans le chapitre sur les fonctions calculable) est, au pire, à temps $10.n^2$ (= temps $T(n) = n$ sur une MT à $k = 2$ bandes - $5kT(n)^2$ sur une machine à une bande).

Lorsqu'un problème est décidable, si la taille de l'entrée est assez petite, alors, la plupart du temps, il est toujours possible de trouver une solution en un temps raisonnable.

Exemple : pour l'algorithme **SillyAdd** précédent, s'il est envisageable de l'utiliser pour $n = 32$, il devient très rapidement inutilisable lorsque n devient plus grand.

On va donc s'intéresser au **comportement asymptotique** du temps d'exécution d'un algorithme, à savoir à quelle vitesse son temps de calcul augmente lorsque la taille du problème à traiter augmente lui-aussi.

Afin de les quantifier, on va utiliser des ordres asymptotiques (= fonctions de majoration).

Ceci nous permettra de définir des classes particulières de problèmes pour lesquels, même si des algorithmes existent pour les résoudre, ils sont **par nature infaisables** (= prennent trop de temps à calculer) dès que la taille du problème augmente.

3.3 Ordres de grandeur

Temps de calcul (taille du problème en fonction de la complexité)

On suppose qu'un instruction s'exécute en $1\mu s = 10^{-6}s$ (rappel : $1ms = 10^{-3}s$)

ordre de grandeur \ $n =$	10	20	30	40	50	60
n	10μs	20μs	30μs	40μs	50μs	60μs
n^2	100μs	400μs	900μs	1.6m	2.5ms	36ms
n^3	1ms	8ms	27ms	64ms	125ms	216ms
n^5	100ms	$3,2$s	$24,3$s	1,7 min	5,2 min	13 min
2^n	$1ms$	$1s$	$17,6$ min	$12,7$ jours	$35,7$ ans	$36,6$ Kan
3^n	$59ms$	$58'$	$6,5$ ans	$385,5$ Kan	22.7 Gan	$1,3$ Tan

Kan = 1000 ans = un millénaire.
Man = 1.000.000 ans = un million d'années.
Gan = 1.000.000.000 ans = un milliard d'années.
Tan = 1.000.000.000.000 ans = mille milliard d'années.

Rappel : on estime que l'univers n'a "que" 13.8 milliard d'années.

Pour les ordres de grandeur :

- de n^k, le temps d'exécution est raisonnable.
- de k^n, le temps d'exécution devient rapidement trop long dès que la taille du problème augmente.

L'illusion de la puissance :

Attendez ! Donnez-moi un Cray Titan à 17.59 Pflops et je vous le fais.

Rappel : 1 PFlops = 1 petaflops = 1 million de Gflops.

Si la puissance de mon ordinateur est multipliée par ...

Pour un même temps de calcul, la taille n du problème peut augmenter de :

ordre de grandeur	$n =$	10	10^2	10^3	10^6	10^9
	n	$\times 10$	$\times 10^2$	$\times 10^3$	$\times 10^6$	$\times 10^9$
	n^2	$\times 3,16$	$\times 10$	$\times 31,6$	$\times 1000$	$\times 31623$
	n^3	$\times 2,15$	$\times 4,64$	$\times 10$	$\times 100$	$\times 1000$
	n^5	$\times 1,58$	$\times 2,51$	$\times 3,98$	$\times 15,8$	$\times 63$
	2^n	$+3,32$	$+6,64$	$+9,96$	$+19,9$	$+29,89$
	3^n	$+2,09$	$+4,19$	$+6,29$	$+12,57$	$+18,89$

Pour les ordres de grandeur :

- de n^k, la taille du problème peut être multipliée.
- de k^n, la taille du problème progresse de quelques unités.

Autrement dit, dans ce dernier cas, l'augmentation de la puissance de calcul ne permet pas d'augmenter significativement la taille du problème qu'il est possible de traiter.

3.4 Temps d'exécution des opérations standards

On s'intéresse maintenant l'ordre de grandeur du temps que mettent les opérations arithmétiques de base à s'exécuter.

Soit deux entiers x et y tels que $x \leq y$, et on note $n = \log(x)$ et $m = \log(y)$.

Le logarithme est pris dans la base dans laquelle les opérations sont effectuées (exemple : en binaire, $\log = \log_2$). Les explications sont données en binaire mais se généralisent à toute base.

Exemples 62: si on veut effectuer $140 + 65 = 205$

- en binaire, on effectue : $10001100 + 1000001 = 11001101$. On a $n = \lceil \log_2(143) \rceil = 8$ et $m = \lceil \log_2(65) \rceil = 6$. L'addition se fait bit à bit. On a besoin de n opérations avec gestion de retenue.
- en base 16, on effectue : $8c + 41 = cd.$. On a $n = \lceil \log_{16}(143) \rceil = 2$ et $m = \lceil \log_{16}(65) \rceil = 2$. L'addition se fait en hexadécimal. On a besoin de deux opérations avec gestion de retenue.

- sur une ALU (Unité de calcul logique et arithmétique d'un processeur) de 64 bits, les calculs s'effectuent par paquets de 64 bits, donc en base 2^{64}. On retrouve bien que le coût de l'addition de deux entiers de 64 bits est 1.

Les opérations sont bien définies à un facteur multiplicatif près suivant la base utilisée.

On a pour les opérations arithmétiques de base :

- **addition :** $x + y$ prend un temps $O(\max(n, m))$ (donc au pire $O(n)$).
  Résultat sur $n + 1$ bits au plus.
- **soustraction :** idem addition (en prenant le complément à deux + 1).
- **multiplication :** $x \times y$ en temps $O(n \times m)$), au pire $O(n^2)$.
  Résultat sur $n + m + 1$ bits au plus.
  Multiplier par 2^i = faire i décalage à gauche. Donc $x \times y$ = sommer $\log(y)$ fois (au plus) un nombre de $\log(x)$ bits décalé à gauche.
- **division entière :** x/y en temps $O(m \times (n - m + 1))$.
  Résultat sur $n - m$ bits.
  Même idée que la multiplication (sauf que l'on fait des soustractions, et que l'on arrête dès que le reste est inférieur au numérateur : d'où le $n - m + 1$).
  cas le pire : $m = n/2$, on est en $O(n^2)$.

Le calcul modulo p permet d'affirmer que tout calcul ci-dessus sera toujours effectué au maximum sur p bits (donc $n = m = p$), avec réduction préalable modulo p si n ou m est plus grand que p.
Pour l'algorithme d'Euclide : $O(n)$, en version étendue : $O(n \times m)$. Résultat sur m bits au plus.

Exemple 63: algorithme d'exponentiation rapide de x^y Étudions ce cas en binaire. Remarquons tout d'abord que si $x = 2^n$ et $y = 2^m$, alors $x^y = (2^n)^{2^m} = 2^{n2^m}$

En conséquence, x^y possède $n2^m$ bits (à un près), soit un **nombre exponentiel de bits**.

L'algorithme d'exponentiation rapide : l'algorithme répète m fois, une élévations au carré (à chaque fois) et une multiplications par x (pour chaque bit de y à 1).

Le nombre total d'opérations est en $O(n^2 \times 2^p)$. La difficulté consiste à tenir compte de l'augmentation progressive du nombre de bits nécessaires pour stocker le résultat de chaque opération, ce qui augmente la complexité des opérations suivantes.

Remarque 28:

Des algorithmes plus rapides existent mais ils ne deviennent efficaces que pour n assez grand (asymptotique) ; autrement dit pour les grands nombres (par exemple 4096 bits).

Pour la multiplication de deux nombres à n bits :

- classique = $O(n^2)$,
- Karatsuba (1962) = $O(n^{\log_2 3})$,
- Schönhage-Strassen (1971) = $O(n \log(n) \log\log(n))$
- Harvey-van der Hoeven (2019) = $O(n \log(n))$,

On pense que ce dernier résultat ne peut pas être amélioré (*i.e.* le gain ne se fera pas sur l'ordre de grandeur).

Mais pourquoi ne serait-il pas possible de faire encore mieux ?

4 Faisabilité

Intéressons-nous maintenant à la faisabilité des problèmes, et donnons une première définition qui semble raisonnable :

Définition V.8 (Problème infaisable)

Un problème est **infaisable** si le comportement asymptotique du temps d'exécution du meilleur algorithme connu pour le résoudre est exponentiel ou subexponentiel.

Mais que signifie subexponentiel ?

Définition V.9 (fonction subexponentielle)

Une fonction $f(x)$ est subexponentielle si :

- $\forall \alpha > 0, \lim_{x\to\infty} f(x)/x^\alpha = +\infty$
- $\lim_{x\to\infty} \log(f(x))/x = 0$

à savoir elle croit beaucoup plus vite que tout polynôme, mais moins vite qu'une exponentielle.

Exemple 64: $f(x) = x^{\log x}$ est une fonction subexponentielle.
La majorité des méthodes de cryptographie moderne ont des difficultés d'attaque d'ordre subexponentiel.

Pour ce type d'algorithme :

- on peut l'exécuter pour une taille d'entrée petite.
- dès que la taille du problème augmente, le temps d'exécution devient tel qu'il devient inutile de lancer l'exécution :
  - il est déraisonnable d'attendre le temps nécessaire (et d'assurer les ressources qui permettre d'assurer le programme de s'exécuter),
  - avec le rythme d'évolution (exponentiel) des puissances des machines (s'il ne change pas), il est plus avantageux de lancer le programme sur une future machine qui n'existe pas encore, que sur une machine actuelle.

    En clair, attendre que l'évolution de la technologie fasse progresser la valeur du n correspondant à un temps d'attente raisonnable.
- La paradigme quantique implique que certains problèmes considérés actuellement comme infaisable pourront se révéler faisable.

Exemple 65: Un problème a un comportement asymptotique en 2^n.
Pour $n = 40$, j'ai un résultat en moins de 13 jours, ce qui est raisonnable.
Je veux maintenant faire un calcul pour $n = 60$. La table précédente nous indique que le temps de calcul sera de l'ordre de 36.600 ans, ce qui l'est beaucoup moins.
L'une des expressions de la loi de Moore dit que la puissance des ordinateurs double tous les 2 ans. Supposons que cette loi reste vraie dans le futur (par amélioration de la technologie, ou augmentation du nombre de transistor, des fréquences, etc ...)
En conséquence, la puissance de calcul progresse de 2^p en $2p$ années. Donc, si j'attends $2 \times 20 = 40$ ans (*i.e.* $p = 60 - 40$), je pourrais exécuter mon calcul avec $n = 20$ en 13 jours.
On notera que pour $n = 1000$, le problème restera infaisable encore longtemps.

Remarque 29:
La progression de la puissance de calcul devrait finir par se heurter à des murs (taille des circuits, fréquences, énergétique, ressources, ...). De nombreux problèmes resteront donc très probablement infaisables.

Pour simplifier :

- un problème est exponentiel s'il est infaisable à long terme,
- un problème est subexponentiel s'il est infaisable à court ou à moyen terme.

Mais alors, quand est-ce qu'un problème est faisable avec un ordinateur ?

Définition V.10 (Problème faisable)
Un problème est considéré comme **faisable** si le comportement asymptotique du temps d'exécution du meilleur algorithme connu pour le résoudre est au plus polynomial.

Afin d'étudier plus précisément les comportements asymptotiques des problèmes, nous allons définir des classes de complexité.

Exercice 37 (Factorielle)**.** *Soit la fonction calculable* $f(p) = p!$.

1. *Écrire le programme (en pseudo-code) qui calcule cette fonction.*
2. *En utilisant la formule de Stirling* $n! \simeq \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$*, calculer l'ordre asymptotique du nombre de bits dans* $n!$.
3. *Lorsque je multiplie* $(n-1)!$ *par n, de combien de bits supplémentaires dois-je disposer afin de stocker le résultat ?*
4. *L'algorithme naïf de multiplication de deux entiers de p bits est de l'ordre de* $O(p^2)$ *(à noter que l'algorithme de Fürer (2007) est en temps* $O(p \log p)$*. Quel est le coût de calcul de* $n!$ *connaissant* $(n-1)!$ *?*
5. *Sachant que pour k grand,* $\sum_{i=1}^{k} i^2 \log^2 i = O(k^3 \log^2 k)$*, calculer la complexité de* $n!$.
6. *En déduire la complexité de calcul de la factorielle si n est un nombre à q bits.*

5 Théorème de Hiérarchie

5.1 Complexité inhérente

Une hypothèse envisageable serait qu'il n'existe pas vraiment une complexité inhérente à un problème.

Par exemple, s'il existe un problème P pour lequel je connais un algorithme permettant de le résoudre en temps 2^n, il y a peut-être un algorithme plus compliqué qui le résout plus vite en $2^{\sqrt{n}}$, et un autre plus compliqué encore plus vite en $n^{\log n}$ ou en n^{1024}, ... jusqu'à $O(n)$.

Évidemment, plus l'algorithme est complexe, plus son code devient long, possiblement tendant vers l'infini à mesure que l'on s'approche de $O(n)$.

Cette hypothèse vous semble absurde ?

Pour le problème de la multiplication de deux matrices :

- "force brute" en $O(n^3)$ avec la définition du produit matriciel.
- algorithme de Strassen (1969) en $O(n^{2.807})$
- algorithme de Coppersmith-Winograd (1980) en $O(n^{2.5})$
- nouvel algorithme de Coppersmith-Winograd (1987) en $O(n^{2.376})$

- algorithme de le Gall (2014) en $O(n^{2.3728639})$
- algorithme de Alman-Williams (2021) en $O(n^{2.3728596})$
- algorithme de Fawzi (2022) utilisant de l'apprentissage pour l'adapter à la taille de la matrice et à la machine (CPU,GPU,...)
- algorithme FBHHRBNRSSSHK (2023) montrant que l'on peut encore améliorer, pour une matrice 5×5, le record établit par Fawzi de 98 multiplications à 96 dans $\mathbb{Z}^2$.
- ...?

Cet exemple montre donc que, même pour un problème classique, on continue à trouver de nouveaux algorithmes qui permettent d'améliorer les temps de calcul.

5.2 Théorème d'accélération

Définissons tout d'abord une mesure de complexité :

Définition V.11 (mesure de complexité de Blum (simplifiée))

Une mesure de complexité de Blum est une paire (ϕ, Φ) telle que :

- ϕ est une énumération de l'ensemble des fonctions calculables qui, comme les machines de Turing, est un ensemble dénombrable. On notera ϕ_i la $i^{\text{ème}}$ fonction calculable, $\phi_i(w)$ l'exécution de ϕ_i sur l'entrée w.
- Φ est un ensemble de fonctions calculables associé à ϕ représentant l'ensemble de mesures de complexité où $\Phi_i(w)$ représente la complexité de $\phi_i(w)$

Par exemple, Φ peut représenter le nombre de transitions lors de l'exécution. Tout type de mesure peut être choisi : le nombre de cellules de la bande utilisés, le nombre de cycles sur le processeur considéré, ...

Théorème V.3 (d'accélération de Blum (1967))

Pour toute fonction totalement calculable r, il existe une fonction f totalement calculable telle que $\phi_i = f$ implique qu'il existe j tel que $\phi_j = f$ et $\Phi_i(w) > r(w, \Phi_j(w))$ presque partout.

Interprétation : pour n'importe quelle accélération r, il existe une fonction totalement calculable f (dont le code est ϕ_i et la complexité est Φ_i) pour laquelle je peux trouver un autre code ϕ_j dont la complexité est inférieure pour presque tous les w.

1. si on prend par exemple une fonction exponentielle pour r, on a $\Phi_i(w) > 2^{\Phi_j(w))}$, à savoir $\phi_j(w)$ va exponentiellement plus vite que $\phi_i(w)$.
2. "presque partout" signifie que pour un nombre fini d'entrées w, l'algorithme ϕ_j n'accélère pas la fonction f.
3. la dépendance de r à w permet de faire dépendre l'accélération à w.

Le théorème est en fait encore plus général et s'applique aux fonctions partiellement calculables.

Il affirme donc l'existence de fonctions que l'on peut "infiniment" optimiser (en se débarrassant possiblement des cas les plus problématiques).

5.3 Théorème de hiérarchie

Montrons maintenant qu'il existe effectivement une hiérarchie des complexités, à savoir :

- il existe des problèmes qui ne peuvent pas être calculé avec une complexité plus faible,

- en conséquence, ces problèmes peuvent être optimisé jusqu'à, potentiellement, leur complexité minimale.

Mais avant de pouvoir le démontrer, une définition et quelques rappels.

5.3.1 Fonction constructible en temps

Définition V.12 (Fonction constructible en temps)

Une fonction $f(n) \leq n$, non décroissante ($f(n) \leq f(n+1)$) est dite constructible en temps si il existe une fonction calculable F qui calcule $F : 1^n \mapsto 1^{f(n)}$ en temps $f(n)$.

à savoir, si il existe une MT M qui prend en entrée $w = 1^n$ et s'arrête en $O(f(n))$ transitions avec $1^{f(n)}$ sur sa bande.

Remarques 30:

- toutes les fonctions classiques n, n^k, 2^n sont constructibles en temps si $f(n) \geq c.n$ où $c = O(1)$ (sinon on n'a pas suffisamment de temps pour lire le contenu de la bande).
- Inversement, toute fonction en $o(n)$ n'est pas constructible en temps (même raison).
- si f_1 et f_2 sont constructibles en temps, alors $f_1 + f_2$ et $f_1.f_2$ aussi. En conséquence, tout polynôme de $\mathbb{N}[X]$ est constructible en temps.

Exemple 66: Considérons $f(n) = n^3$ en se plaçant sur un transducteur :

$M(w) =$
compter le nombre n de symbole 1 sur la bande
calculer n^3
écrire n^3 fois le symbole 1 sur la bande

Chacune de ses opérations est au plus en n^3 :

- **comptage des 1 :** n incrémentations sur $\log(n)$ bits au plus $= O(n \log n)$
- **calcul de n^3 :** calcul n^2 en $O(\log^2(n))$, calcul de n^3 en $O(\log(n)\log(n^2)) = O(\log^2(n))$. Donc le tout en $O(\log^2(n))$.
- **écriture de 1^{n^3} sur la bande de sortie :** en $O(n^3)$ (décrémentation du compteur n^3 en $O(n^3 \log n^3)$)

Tous les termes sont dominés par la dernière étape en $O(n^3) = O(f(n))$.

La dernière étape nécessairement un temps $f(n) = n^3$ puisqu'il faut écrire n^3 fois un 1 sur la bande de sortie.

Donc $f(n)$ est constructible en temps.

5.3.2 Temps d'exécution d'une MT universelle

On rappelle que :

Théorème V.4 (temps d'exécution d'une MT universelle)

Soit une machine de Turing M.
Notons :

- $T_M(w)$ le temps d'exécution de M sur l'entrée w.
- $poly(n)$ une fonction polynomiale de variable n.

Il existe une MT universelle capable de simuler une machine M sur l'entrée w telle que :

$$T_U(\langle M, w\rangle) \leq poly(|\langle M\rangle|).T_M(w)\log T_M(w)$$

Nous avons déjà vu une variation de ce théorème avec une pénalité en $T \log T$ (cf machine de Turing universelle efficace).

Celui-ci précise que la constante multiplicative peut être majorée par un polynôme qui ne dépend que de la longueur de l'encodage de la machine.

5.3.3 Théorème de hiérarchie temporelle

Théorème V.5 (Hiérarchie temporelle)

Soient :

- G une fonction constructible en temps
- g une autre fonction constructible en temps telle que $g(n) \log g(n) = o(G(n))$

Il existe des langages en temps $G(n)$ qui ne sont reconnus par aucune machine en temps $g(n)$.

Démonstration:

Soient deux fonctions $g(n)$ et $G(n)$, telle que :

- $\lim_{n\to\infty} \frac{g(n)\log g(n)}{G(n)} = 0$ (à savoir $g(n)\log g(n) = o(G(n))$).
- $g(n)$ et $G(n)$ sont constructibles en temps,

Soit l'ensemble :

$$E_G = \{M \in \mathcal{R} \mid \mathcal{L}(M) \text{ en temps } G(n)\}.$$

E_G est l'ensemble des machines décidables en temps $G(n)$.

Donc, E_G contient aussi l'ensemble des machines en temps $g(n)$ (i.e. $E_g \subseteq E_G$).

On veut montrer que cette inclusion est stricte ($E_g \subsetneq E_G$), à savoir qu'il existe des langages qui ne sont pas décidable par n'importe quelle machine en temps $g(n)$.

Considérons la MT D suivante qui utilise la MTU U :

$D(\langle M\rangle) =$

```
CALCULER n = |⟨M⟩|
SIMULER U(⟨M, ⟨M⟩⟩) sur au plus G(n) transitions
SI elle s'est arrêtée
ALORS RETOURNER l'opposé de la décision
SINON ACCEPTER // peu importe pour la démonstration
```

Comme $G(n)$ est constructible en temps (i.e. $G(n)$ peut être calculé à partir d'une entrée de taille n qui est utilisé pour compter les transitions), et que U exécute au plus $G(n)$ transitions, D **est en temps** $G(n)$.

Prenons maintenant une machine de Turing R en temps $g(n)$.

Quel temps prend $D(\langle R\rangle)$ à s'exécuter ? Comme R en temps $g(n)$, la simulation est faite en temps $poly(|\langle R\rangle|)(g(n)\log(g(n)))$

Le terme $poly(|\langle R\rangle|)$ est ennuyeux puisqu'il ajoute un facteur multiplicatif en $poly(n)$, ce qui se produit parce qu'on exécute R sur sa propre description (*i.e.* $n = |\langle R\rangle|$).

En conséquence, la taille de l'entrée (dont dépend le temps d'exécution) est proportionnelle à la description de la machine.

On voudrait pouvoir regarder avoir un comportement asymptotique du temps d'exécution de R.

Remarquons que $\langle R \rangle$ et $\langle R \rangle 10^k$ représentent la même machine de Turing (puisque le padding est ignoré).

Donc pour une entrée de $n = |\langle R \rangle 10^k| = |\langle R \rangle| + k + 1$, et pour n assez grand, $k/n \simeq 1$, on a $poly(|\langle R \rangle 10^k|) = poly(|\langle R \rangle|)$ car seul compte l'encodage effectif de la machine (voir V.4).

Pour toute machine R en temps $g(n)$, il existe donc une constante k_R, telle que $poly(|\langle R \rangle|)(g(n)\log(g(n))) < G(n)$ où $poly(|\langle R \rangle|)$ est une constante "indépendante" de n.

En conséquence, pour tout R en temps $g(n)$, il existe un mot $w_R = \langle R \rangle 10^k$ tel que $D(w_R)$ simule $U(R, \langle R \rangle 10^k)$ en temps $g(n)\log(g(n))$.

Nous pouvons maintenant utiliser l'argument de diagonalisation.

Nous avons construit une MT $D(w)$ où, pour toute machine R en temps $g(n)$, il existe un mot w_R tel que :

- la simulation $U(R, \langle w_R \rangle)$ s'arrête en temps $g(n)\log(g(n)) = o(G(n))$.
- $D(w_R)$ prend la décision opposée de $R(w_R)$.

Toute machine R en temps $g(n)$ reconnaît un langage différent de D, car la décision de D sur w_R est l'opposée de celle de R (diagonalisation).

Comme D est en temps $G(n)$, il existe donc bien dans des langages en temps $G(n)$ qui ne peuvent être reconnus par aucun langage en temps $g(n)$. □

Nous venons donc de démontrer qu'il existe bien des problèmes qui ont une complexité minimale (ils ne peuvent pas être résolus en moins de temps).

Évidemment, on pense que la plupart des problèmes sont de cette nature. Le théorème d'accélération démontre, quand à lui, que ce n'est pas toujours le cas.

6 Classes de complexité en temps déterministe

Grâce au théorème de Hiérarchie (voir théorème V.5), nous savons qu'il y a un sens à classifier les problèmes en fonction de leurs complexités.

On peut donc définir des classes de complexité.

6.1 Définition

On rappelle qu'une machine de Turing M est à temps $t(n)$ si, pour toute entrée w de longueur n, le nombre de transitions que prend M pour décider w est en $O(t(n))$.

A savoir, $t(n)$ est la borne asymptotique supérieure de son temps d'exécution.

Définition V.13 (classe de complexité temporelle)

Soit $t : \mathbb{N} \to \mathbb{R}^+$ une fonction.
La classe de complexité temporelle $\texttt{TIME}(t(n))$ est l'ensemble de tous les langages décidables par une MT déterministe à temps $O(t(n))$.

Exemple 67: $\texttt{TIME}(n^2)$ est contient l'ensemble des langages (*i.e.* des problèmes) décidables par un algorithme s'exécutant sur une machine de Turing en temps $O(n^2)$ pour une entrée de taille n.

On notera que :

- On donne une borne supérieure asymptotique pour le temps d'exécution d'une machine de Turing M. Elle peut être sur-évaluée (majoration trop large).
- De la même façon, si un langage $L \in \textbf{TIME}(n^3)$, cela signifie qu'il existe une MT M en temps n^3 qui reconnaît L.

  Néanmoins, il existe peut-être un autre algorithme M' qui reconnaît L de manière plus efficace (par exemple en temps $n^2 \log n$).
- Pour un langage A (voir par l'exemple 56), y-a-t-il une contradiction entre $A \in \texttt{TIME}(n^2)$ et $A \in \texttt{TIME}(n \log n)$?

  A appartient aux deux. Il est toujours possible de faire plus lent (par exemple en faisant autre chose).

Nous verrons par la suite que, souvent, il n'est pas nécessaire d'obtenir la meilleure borne possible avec le meilleur algorithme possible pour se faire une idée de la complexité d'un problème.

On remarquera que la notion de complexité est en général lié à un algorithme, et non directement à un problème.

Lorsque que l'on dit qu'un problème possède une certaine complexité, on signifie par là :

- soit qu'il a été démontré (mathématiquement) qu'il n'existe pas d'algorithme plus rapide pour ce problème,
- soit qu'il s'agit de la complexité du meilleur algorithme **connu**, impliquant que sa complexité réelle est peut-être plus faible.

6.2 Stabilité du temps polynomial

Définition V.14 (Temps polynomial)

si un MT M s'exécute en temps $t(n) = O(n^c)$ avec $c > 1$,
alors on dit que M s'exécute en temps polynomial.

Corollaire V.1 (complexité d'une MT multi-bandes)

Toute MT M_k à k-bandes à temps polynomial possède
une MT M à une bande équivalente à temps polynomial.

Démonstration:

Si M_k s'exécute en temps $t_k(n) = O(n^c)$, alors M s'exécute en temps $t(n) = O(k^2 t_k(n)^2) = O(k^2 n^{2c}) = O(n^{c'})$ où $c' > 2c > 1$.
Donc, M s'exécute aussi en temps polynomial. □

Conséquence : tout algorithme s'exécutant en temps polynomial sur une MT à k-bandes s'exécute aussi en temps polynomial sur un MT à une bande.

Plus généralement : tout algorithme s'exécutant en temps polynomial sur toute variation d'une MT déterministe s'exécute aussi en temps polynomial sur un MT à une bande. Mieux, la pénalité est au plus quadratique.

6.3 Complexité polynomiale déterministe

On voit donc que des langages polynomiaux (donc décidés par un algorithme en temps polynomial) sont fondamentaux pour l'étude de la complexité.

On définit donc :

Définition V.15 (classe de complexité **P**)

P est la classe des langages qui sont décidables en temps polynomial par une MT à simple bande.
Autrement dit : $\mathbf{P} = \bigcup_k \texttt{TIME}(n^k)$

Remarques 31:

- comme vu précédemment, **P** correspond à la classe des problèmes qui sont solvables sur un ordinateur en un temps réaliste (= faisable).
- **P** est invariant pour tous les modèles de machine que l'on peut simuler sur une MT simple bande en un temps polynomial.
  si un modèle de machine M' résout un problème en temps polynomial et que la simulation de M' sur une MT M se fait en temps polynomial, **alors** M peut résoudre le même problème en temps polynomial.

Conséquence : comment déterminer si un algorithme est à temps polynomial ?

- exprimer les entrées de l'algorithme, et voir comment celles-ci participent à la taille n de l'entrée.
- écrire l'algorithme en pseudo-code
- pour chaque étape, donner une borne supérieure de sa complexité en fonction de n.
- pour des étapes successives, les complexités s'additionnent.
- la complexité d'une boucle est la complexité du corps de boucle multiplié par le nombre de répétitions.
  on s'assurera que le nombre de boucles est toujours fini (condition d'arrêt).
- on conservera que le terme dominant du cumul des complexités.

Exemple 68: Pour un code séquentiel où l'on aura trouvé les complexités n^5, n^2 et 4.
$O(n^5 + n^2 + 4) = O(n^5)$ car $\lim_{n\to\infty} n^5 + n^2 + 4 = \lim_{n\to\infty} n^5(1 + 1/n^3 + 4/n^5) = \lim_{n\to\infty} n^5$. Le terme en n^5 domine donc tous les autres.

L'algorithme s'exécute alors en temps polynomial (car tout ordinateur classique peut être simulé sur une machine de Turing avec une pénalité au plus d'ordre polynomiale).

Attention : l'affirmation précédente n'est valide que pour les machines physiquement réalisables [1] à l'exception des machines quantiques.

Donnons maintenant quelques exemples.

1. ce qui exclut les machines de Turing non déterministes

Exemple : langage PATH

Définition V.16 (langage PATH)

PATH = $\{\langle G, s, t\rangle \mid G$ est un graphe avec un chemin de s vers $t\}$

Donc $\langle G, s, t\rangle \in$ PATH ssi il existe un chemin entre les sommets s et t dans le graphe $G = (V, E)$, ce qui ne peut être le cas que si $(s, t) \in V^2$.

Proposition V.1

PATH $\in$ **P**

Démonstration: PATH $\in$ **P**

Montrons qu'il existe un décideur M à temps polynomial pour PATH.

$M(\langle G, s, t\rangle) =$

```
   // les marquages se font dans la liste des sommets de G
1  marquer le sommet s
2  répéter
3  |   pour chaque arête de G
4  |   |   si un seul sommet est marqué alors marquer l'autre
5  jusqu'à ce que plus aucun nouveau sommet ne soit marqué
6  si t est marqué alors accepter sinon rejeter
```

Temps d'exécution de M : soit m le nombre de nœuds dans G.

- Les lignes 1 et 6 parcourent l'entrée une fois en temps $O(m)$.
- Les lignes 2-5 s'exécutent au plus m fois (temps maximum de propagation entre s et t). Chaque exécution a au plus $O(m^2)$ étapes (chaque nœud est lié avec au plus $m - 1$ autre nœud), et s'exécute en temps $O(m^3)$.
- $m = O(n)$ où $n =$ longueur de la chaîne d'entrée.

Dont M s'exécute en temps $O(n + n^3 + n) = O(n^3)$.
Donc, PATH $\in$ TIME$(n^3) \subset$ **P**. □

Exemple : langage RELPRIME

Définition V.17 (langage RELPRIME)

RELPRIME = $\{\langle x, y\rangle \mid x$ et y sont entiers et PGCD$(x, y) = 1\}$

Donc $\langle x, y\rangle \in$ RELPRIME ssi (x, y) sont des entiers premiers entre eux.

Proposition V.2

RELPRIME $\in$ **P**

Démonstration: RELPRIME $\in$ **P**

Montrons qu'il existe un décideur M à temps polynomial pour RELPRIME.
On utilise l'algorithme d'Euclide.

$M(\langle x, y\rangle) =$

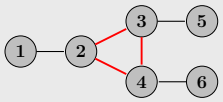
```
1  si x < y alors échanger x et y
2  répéter
3  |   x = x  mod y
4  |   échanger x et y
5  jusqu'à jusqu'à ce que y = 0
6  si x = 1 alors accepter sinon rejeter
```

Temps d'exécution de M :

- Les lignes 1 et 5 sont exécutées une seule fois.
- Chaque exécution de la ligne 3, réduit la valeur de x par aux moins 2. Donc le nombre d'exécutions est au plus de $O(z)$ où $z = \log_2 x + \log_2 y$.
- Toute opération arithmétique (comparaison, modulo) peut être exécutée en temps polynomial du codage des entrées, donc polynomial en z.

Au total, M est polynomial en z.
Comme $z = O(n)$, RELPRIME $\in$ TIME$(n) \subset \mathbf{P}$. □

Exercice 38 (Triangle)**.** *Dans un graphe non orienté, un triangle est un ensemble de 3 arêtes connectées entre elles.*



Soit : $\texttt{TRIANGLE} = \{\langle G\rangle \mid G \text{ est un graphe non orienté contenant un triangle}\}$
Montrer que $\texttt{TRIANGLE} \in \boldsymbol{P}$.

Exercice 39 (Fermeture de **P** par les opérateurs réguliers)**.** *Montrer que* ***P*** *est fermé par :*

1. *union.*
2. *concaténation.*
3. *opérateur de Kleene *.*

Exercice 40 (CONNECTED)**.**
Soit $\texttt{CONNECTED} = \{\langle G\rangle \mid G$ *est un graphe tels que toutes paires de sommets est connectée par une arête*$\}$
Montrer que $\texttt{CONNECTED} \in \boldsymbol{P}$.

Exercice 41 (BIPARTITE)**.**
Un graphe bipartite est un graphe tel que les sommets peuvent être partitionnés en deux ensembles A et B tels que toute arête relie un sommet dans A à un sommet dans B (il n'y a pas d'arête entre deux sommets de A ou entre deux sommets de B).
$\texttt{BIPARTITE} = \{\langle G\rangle \mid G \text{ est un graphe bipartite}\}$
Montrer que $\texttt{BIPARTITE} \in \boldsymbol{P}$.

Exercice 42 (TREE)**.**
Un graphe est un arbre s'il est connecté et ne contient aucun cycle.
Alternativement, un graphe G est un arbre si tout couple $(u, v) \in G$ *est connecté par un chemin simple (= sans répétition de sommets).*
$\texttt{TREE} = \{\langle G\rangle \mid G \text{ est un arbre}\}$
Montrer que $\texttt{TREE} \in \boldsymbol{P}$.

6.4 Complexité subexponentiel et exponentiel déterministe

Les classes de complexité au-delà de **P** sont :

Définition V.18 (classe de complexité **SUBEXP**)

SUBEXP est la classe des langages qui sont décidables en temps subexponentiel par une MT à simple bande.
Autrement dit :**SUBEXP** = $\texttt{TIME}(2^{o(n)})$

Définition V.19 (classe de complexité **EXP**)

EXP est la classe des langages qui sont décidables en temps exponentiel par une MT à simple bande.
Autrement dit : **EXP** = $\bigcup_k \texttt{TIME}(2^{n^k})$

Nous avons vu que les problèmes subexponentiels ou exponentiels étaient généralement infaisables.

Mais, serait-il possible de définir une sous-classe de problèmes pour laquelle la résolution du problème pourrait paraître envisageable ?

7 Classes de complexité en temps non déterministe

On sait déjà que les machines déterministes sont capables de résoudre des problèmes exponentiels en des temps bien inférieurs à une machine déterministe (une MTND en temps $f(n)$ se simule sur une MT déterministe en temps $2^{O(f(n))}$).

On voudrait créer une classe de complexité juste au-dessus de **P** mais moins compliquées que **EXP**.

La complexité d'un problème pour laquelle on cherche une solution peut venir de plusieurs facteurs, notamment de la complexité :

1. à trouver une solution candidate,
2. à vérifier qu'un candidat solution est bien solution du problème.

A priori, les problèmes pour lesquels il est simple de vérifier qu'une solution est bien une solution est plus simple, car il localise la complexité sur la recherche de solution.

Tentons de formaliser cette notion.

7.1 Vérificateur et certificat

Avant de pouvoir parler de la classe **NP**, nous devons aborder la notion de vérificateur.

Définition V.20 (vérificateur)

Un **vérificateur** V pour un langage A est un algorithme tel que :
$A = \{w \mid \exists c; V(\langle w, c\rangle) \text{ accepte}\}$

Remarques 32:

- V utilise une information c supplémentaire pour vérifier que $w \in A$.
  c s'appelle un certificat (ou preuve) d'appartenance à A.
- le temps d'exécution du vérificateur se mesure en terme de longueur de w
  i.e. la longueur de c ne compte pas dans la longueur de l'entrée.

Définition V.21 (vérification)

- un **vérificateur à temps polynomial** est un vérificateur qui s'exécute en temps polynomial sur la longueur de w.
- un langage A est un **langage polynomialement vérifiable** si il possède un vérificateur à temps polynomial.

Exemple 69: de vérificateur
Soit $S = \{s_1, s_2, \ldots, s_n\}$ un ensemble d'entiers et t un entier.
Existe-t-il un sous-ensemble de S tel que la somme de ses entiers soit égal à t ?
On définit le langage associé : SUBSET-SUM= $\left\{\langle S, t\rangle \mid \exists S' \subset S, \sum_{s_i \in S'} s_i = t\right\}$
Exemples de certificats c **pour** $S = \{5, 3, 4, 2, 1, 6\}$ et $t = 13$
$\{6, 4, 3\}$, $\{5, 4, 3, 1\}$, ...
ce sont toutes des valeurs possibles pour c.

Reformulation :

un vérificateur est donc un algorithme qui :

- $\forall w \in A$, il existe (au moins) un certificat c tel que $V(w, c)$ accepte.
- $\forall w \notin A$, il n'existe aucun certificat c tel que $V(w, c)$ accepte.

On peut considérer c comme la solution au problème w,
V est une façon de vérifier que cette solution est valide.

7.2 Classe NP

7.2.1 Définition

Définition V.22 (Classe **NP**)
La classe **NP** est la classe des langages qui sont polynomialement vérifiables.

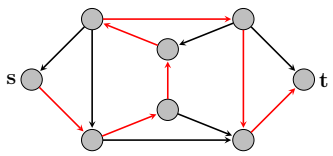
Remarques 33:

- ne provient pas de la complexité de vérification qu'une solution est acceptable.
- mais bien du problème lui-même.

7.2.2 Exemples

Exemples 1 : HAMPATH et $\overline{\text{HAMPATH}}$

Un chemin Hamiltonien dans un graphe orienté est un chemin orienté qui passe exactement une seule fois par tous les sommets :
HAMPATH = $\{\langle G, s, t\rangle \mid G$ graphe orienté avec un chemin Hamiltonien de s à $t\}$



HAMPATH est polynomialement vérifiable.
Un chemin Hamiltonien C d'un graphe $G = (V, E)$ est une liste de sommets consécutifs de C.
Comment vérifier qu'un chemin C est un graphe Hamiltonien de G ?
On utilise le décideur suivant :

$M(\langle G, C\rangle) =$

```
   // vérifie que C contient tous les sommets de G
1  pour chaque sommet s de C
2  |  si s ∈ V alors marquer s dans V sinon rejeter
3  pour chaque sommet s de V
4  |  si s n'est pas marqué alors rejeter
   // vérifie que le chemin dans C existe dans E
5  pour chaque paire de sommets (u, v) consécutive de C
6  |  si (u, v) ∉ E alors rejeter
7  accepter
```

Sa complexité est en $O(n^2)$ où $n = |\langle G, C\rangle|$, donc polynomialement vérifiable.
En revanche, trouver une solution est beaucoup plus difficile :

- complexité de résolution par force brute : $n! = O(n^n)$ (test de tous les chemins possibles).
- pas d'algorithme connu résolvant HAMPATH en temps polynomial.

Définition V.23 (Langage "graphe non Hamiltonien")

$\overline{\texttt{HAMPATH}}$ = graphes sans chemin Hamiltonien.
non polynomialement vérifiable : impossible de valider sans tout vérifier.

Proposition V.3

HAMPATH $\in$ **NP**.

Démonstration:

Définissons un vérificateur V à temps polynomial. Il faut montrer que :

- $\forall \langle G\rangle \in$ HAMPATH, alors $\exists c \mid V(\langle G, c\rangle)$ accepte,
- $\forall c$ tel que $V(\langle G, c\rangle)$ accepte, alors $\langle G\rangle \in$ HAMPATH

Définissons un vérificateur V comme suit :

$V(\langle G, c\rangle) =$

```
1  si c est un chemin hamiltonien alors accepter sinon rejeter
```

V s'exécute en temps polynomial (vérifier que chaque transition du chemin est dans la liste des arêtes, puis vérifier que chaque sommet n'est traversé qu'une seule fois).
Soit H $= \{\langle G\rangle \mid V(\langle G, c\rangle)$ accepte$\}$.

- $\forall \langle G\rangle \in$ H, il existe c qui accepte $\langle G, c\rangle$. Ceci implique que c est un chemin hamiltonien. Donc, $\langle G\rangle$ est un graphe Hamiltonien et H $\subseteq$ HAMPATH.
- Pour tout $\langle G\rangle \in$ HAMPATH, soit c le cycle Hamiltonien dans ce graphe, alors V accepte $\langle G, c\rangle$. Donc, HAMPATH $\subseteq$ H.

Donc, HAMPATH $\in$ **NP**. □

Exemple 2 : COMPOSITES

Définition V.24 (Langage "composite")

Un entier est composite s'il est le produit de deux entiers plus grands que 1 :
On définit le langage :
$\texttt{COMPOSITES} = \{x \mid x = pq, \text{ pour deux entiers } p, q > 1\}$

Proposition V.4

$\texttt{COMPOSITES} \in \mathbf{NP}$.

Démonstration:

Soit le vérificateur V défini par :

$V(\langle x, c\rangle) =$

```
1 si (c n'est pas 1 ou x) et (c divise x) alors accepter sinon rejeter
```

V s'exécute en temps polynomial $O(|\langle x\rangle|^2)$ (division en n^2).
Soit $\texttt{C} = \{\langle x\rangle \mid V(\langle x, c\rangle) \text{ accepte}\}$.

- $\forall \langle x\rangle \in \texttt{C}$, il existe c tel que V accepte $\langle x, c\rangle$. Donc, x est un nombre composite, et $\texttt{C} \subseteq \texttt{COMPOSITES}$
- $\forall \langle x\rangle \in \texttt{COMPOSITES}$, soit c un diviseur de x avec $1 < c < x$. Alors V accepte $\langle x, c\rangle$, $\texttt{COMPOSITES} \subseteq \texttt{C}$.

Donc, $\texttt{COMPOSITES} \in \mathbf{NP}$. □

7.2.3 Caractérisation la classe NP

Théorème V.6 (caractérisation de la classe **NP**)

(Un langage A est dans **NP**) si et seulement si (il est décidé par une MT non-déterministe en temps polynomial).

Autrement dit, s'il existe un vérificateur à temps polynomial qui décide A.

Démonstration: ($A \in \mathbf{NP}$) $\Rightarrow$ (A **décidé par une MTND en temps polynomial)**

Si A est dans **NP**, alors il existe un vérificateur V à temps polynomial qui permet de le vérifier.
Soit n^k le temps d'exécution de V.
Soit N, la MT non-déterministe suivante :

$N(\langle w\rangle) =$

```
1 choix non déterministe d'une chaîne c de longueur au plus n^k
2 exécuter V(⟨w, c⟩)
3 si V accepte alors accepter sinon rejeter
```

Alors, sur la MTND N :

1. générer l'ensemble des chaînes de longueur n^k se fait en temps polynomial $O(n^k)$ (arbre avec $|\Sigma|$ fils par nœud et de profondeur n^k). On obtient ainsi tous les certificats possibles.
2. vérifier chaque certificat avec $V(\langle w, c\rangle)$ prend lui-aussi un temps polynomial (revient à poursuivre chaque feuille de l'arbre précédent et l'accepter ou la rejeter).
3. acceptation des branches. $O(1)$ sur une MTND.

V accepte si l'une quelconque des branches accepte. Donc, si un certificat c existe pour w, alors N le trouve nécessairement.
N permet donc de décider pour tout w de A en temps polynomial. □

Démonstration: (*A* décidé par une MTND en temps polynomial) ⇒ (*A* ∈ NP)

Soit A un langage accepté par une MTND N à temps polynomial. On construit un vérificateur à temps polynomial de la façon suivante :

$V(\langle w, c\rangle) =$

1 **simuler** $N(\langle w\rangle)$ en utilisant c comme la description du choix non-déterministe de N à chaque étape.
2 **si** *la branche d'exécution de N accepte* **alors** accepter **sinon** rejeter

Si un MTND est à temps polynomial, alors chacune de ses branches s'exécute en temps polynomial. Or, le certificat c pris est celui qui permet de choisir l'une des branches acceptante à chaque nœud de la MTND. Donc, l'exécution de V ne revient à évaluer qu'une seule branche de la MTND. Par conséquent, V s'exécute en temps polynomial. □

Définition V.25 (NTIME($t(n)$))

NTIME($t(n)$) = ensemble des langages qui peuvent être décidés par une MT non-déterministe à temps $O(t(n))$.

Corollaire V.2

NP = $\bigcup_k$ NTIME(n^k)

Démonstration:

conséquence directe du théorème précédent. □

Nous avons donc 2 façons de démontrer qu'un problème appartient à **NP** :

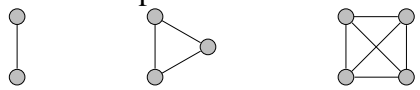
- en utilisant un vérificateur.
- en utilisant une MT non-déterministe à temps polynomial.

NP a, en fait, pour signification **P**olynomial en temps **N**on déterministe, et non pas **N**on **P**olynomial. Maintenant, application pour montrer l'appartenance de nouveaux langages à **NP**.

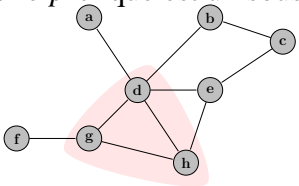
Exemple 1 : CLIQUE

Définition V.26 (Langage Clique)

Un graphe complet K_p est un graphe qui contient p noœuds, et tel que chaque nœud soit connecté au $p-1$ autres nœuds par une arête.



Une p-clique est un sous-graphe complet K_p d'un graphe G non orienté.



Le graphe ci-contre contient deux 3-cliques : $\{g, h, d\}$ et $\{d, e, h\}$.

On définit le langage :

CLIQUE = $\{\langle G, k\rangle \mid G$ est un graphe avec une k-clique$\}$

Proposition V.5

CLIQUE ∈ **NP**.

DÉMONSTRATION:

Méthode 1 : en utilisant un vérificateur.

Le certificat c du vérificateur V sont les nœuds constituant une k-clique.

$V(\langle G, k, c\rangle) =$

1 **tester si** c est un sous-ensemble de k nœuds de G.
2 **tester si** chaque couple de nœuds de c est connecté dans G.
3 **si** *les deux tests sont vrais* **alors** accepter **sinon** rejeter

Trivialement, V accepte tout $\langle G, k\rangle \in$ `CLIQUE` en temps polynomial.

Donc, `CLIQUE` est polynomialement vérifiable et `CLIQUE` $\in$ **NP**. □

Méthode 2 : en utilisant une MT non-déterministe à temps polynomial qui accepte `CLIQUE`.

La MTND N suivante permet de décider `CLIQUE` :

$N(\langle G, k\rangle) =$

1 **choix non-déterministe** de c contenant k-nœuds de G.
2 **tester si** chaque couple de nœuds de c est connecté.
3 **si** *le test est vrai* **alors** accepter **sinon** rejeter

Toutes les étapes de N s'exécutent en temps polynomial.

Donc, `CLIQUE` $\in$ **NP**. □

EXEMPLE 2 : `SUBSET-SUM`

Proposition V.6

`SUBSET-SUM` $\in$ **NP**.

DÉMONSTRATION:

Méthode 1 : en utilisant un vérificateur

Le certificat c du vérificateur V sont des entiers de S dont la somme est t.

$V(\langle S, t, c\rangle) =$

1 **tester si** c est un sous-ensemble d'entiers de S.
2 **tester si** la somme des nombres de c fait t.
3 **si** *les deux tests sont vrais* **alors** accepter **sinon** rejeter

Trivialement, V accepte tout $\langle S, t\rangle \in$ `SUBSET-SUM` en temps polynomial.

Donc, `SUBSET-SUM` est polynomialement vérifiable et `SUBSET-SUM` $\in$ **NP**. □

Méthode 2 :en utilisant une MT non-déterministe à temps polynomial qui accepte `SUBSET-SUM`.

La MTND N suivante permet de décider `SUBSET-SUM` :

$N(\langle G, k\rangle) =$

1 **choix non-déterministe** d'un sous-ensemble c de S.
2 **tester si** la somme des nombres de c fait t.
3 **si** *le test est vrai* **alors** accepter **sinon** rejeter

Toutes les étapes de N s'exécutent en temps polynomial.

Donc, `SUBSET-SUM` $\in$ **NP**. □

Exercice 43 (Fermeture de **NP**). *Montrer que **NP** est fermé par union :*

1. *avec un vérificateur à temps polynomial.*
2. *avec une machine de Turing non déterministe à temps polynomial.*

Exercice 44 (SUBSET_SUM). *Montrer que SUBSET_SUM est dans **NP** :*

1. *avec un vérificateur à temps polynomial.*
2. *avec une machine de Turing non déterministe à temps polynomial.*

Exercice 45 (UNARY_SUM).
Soit : $\texttt{UNARY\_SUM} = \Big\{\langle S, t\rangle$ *où* $S = \{x_1, \ldots, x_k\}$ *sont des entiers stockés en unaire et* $\exists S' \subseteq S$ *tel que* $\sum_{x_i \in S'} x_i = t\Big\}$

1. *Montrer que* $\texttt{UNARY\_SUM} \in \boldsymbol{P}$.
2. *Serait-il possible de trouver un algorithme équivalent pour SUBSET_SUM montrer qu'il est dans **NP** ?*

7.2.4 Autres propriétés en temps non déterministe

Le théorème de Hiérarchie existe aussi en temps non déterministe :

Théorème V.7 (Hiérarchie en temps non-déterministe)

Si $g(n)$ est croissante et constructible en temps, et $f(n+1)\log f(n+1) = o(g(n))$ alors $\texttt{NTIME}(f(n)) \subsetneq \texttt{NTIME}(g(n))$.

Nous ne démontrerons pas ce résultat.

Théorème V.8

Pour toute fonction $t : \mathbb{N} \to \mathbb{N}$, $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n)) \subseteq \texttt{TIME}\left(2^{O(t(n))}\right)$

Démonstration:

- $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n))$: trivial, car toute machine déterministe est une machine non déterministe avec une seule branche.
- $\texttt{NTIME}(t(n)) \subseteq \texttt{TIME}(2^{O(t(n))})$: l'ensemble des branches de calcul d'une machine non déterministe en $t(n)$ est en $2^{O(t(n))}$. Comme chaque branche prend un temps, au plus de $t(n)$, cela peut être simulé en temps $t(n).2^{O(t(n))} = 2^{O(t(n))}$ sur une machine déterministe.

□

7.3 Temps non-déterministe exponentiel

De manière similaire à **EXP**, on peut définir l'équivalent pour le temps non déterministe :

Définition V.27 (classe de complexité **NEXP**)

NEXP est la classe des langages qui sont décidables en temps exponentiel par une MT à simple bande.

Autrement dit : $\textbf{NEXP} = \bigcup_k \texttt{NTIME}(2^{n^k})$

8 NP-complétude

8.1 Comparaison entre P et NP

Si on compare les caractéristiques de **P** et de **NP** :

P classe des algorithmes qui peuvent être décidés "rapidement".
NP classe des algorithmes qui peuvent être vérifiés "rapidement".

où "rapidement" = en temps polynomial.

Réflexions :

La puissance de la vérifiabilité à temps polynomial
(i.e. d'une MT non-déterministe à temps polynomial)
semble plus importante que la décidabilité à temps polynomial
(i.e. d'une MT déterministe à temps polynomial)
Autrement dit, que $\mathbf{P} \subset \mathbf{NP}$
Mais ...

Malheureusement,

- personne n'a jamais réussi à démontrer si $\mathbf{P} = \mathbf{NP}$
  un des grands problèmes d'informatique/mathématiques
- beaucoup pensent que $\mathbf{P} \neq \mathbf{NP}$
  comme cela n'a jamais été démontré, cela est peut-être faux.

En 1971, Cook & Levin démontrent séparément que :

- Il existe des problèmes caractéristiques de la classe **NP** (dit problèmes **NP**-complets).
- Pour un problème **NP**-complets,
  si **n'importe lequel** d'entre eux est décidable par un MT déterministe en temps polynomial,
  alors **tous** les problèmes de **NP** peuvent être décidés par une MT déterministe en temps polynomial.
- **Reformulation** :
  il existe certains langages L tels que si $L \in \mathbf{P}$, alors $\mathbf{P} = \mathbf{NP}$.

et ils trouvent un exemple de problème **NP**-complet.

8.2 Réduction en temps polynomial

Rappel

Si un problème A peut être réduit en un autre problème B, alors :

- si B est décidable, alors A est décidable.
- si B est énumérable, alors A est énumérable.

On remarquera qu'une réduction est une transformation en un **nombre fini de pas**.

Définition V.28 (fonction calculable en temps polynomial)

Une fonction $f : \Sigma^* \to \Sigma^*$ est une **fonction calculable en temps polynomial** s'il existe une MT M à temps polynomial qui s'arrête avec $f(w)$ sur sa bande lorsque son entrée est w.

Définition V.29 (langage réductible en temps polynomial)

Un langage A est **réductible en temps polynomial** en un langage B, s'il existe une fonction f calculable en temps polynomial telle que : $w \in A \Leftrightarrow f(w) \in B$

Notation : $A \leq_{\mathbf{P}} B$

On appelle f une réduction en temps polynomial de A à B

Théorème V.9 (Réduction en temps polynomial à **P**)

Soit $A \leq_P B$ et $B \in \mathbf{P}$.
Alors $A \in \mathbf{P}$.

Démonstration:

Soit un algorithme M (= une MT) qui décide B en temps polynomial.
Soit f une réduction en temps polynomial de A vers B.
Soit N l'algorithme suivant :

$N(\langle w \rangle) =$

1 **calculer** $f(w)$.
2 **exécuter** $M(< f(w) >)$
3 **décider** comme M

N décide B car M accepte $f(w)$ pour tout $w \in A$, puisque f est une réduction de A dans B et M un décideur pour A.
N s'exécute en temps polynomial puisque la réduction f et le décideur M s'exécutent consécutivement, chacun en temps polynomial.

□

Maintenant, un exemple de réduction en temps polynomial.

8.3 Rappel de logique des propositions

Tables de vérité : opérateurs logiques OU $\vee$, ET $\wedge$ et NON $\neg$

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

x	$\overline{x}$
0	1
1	0

Définition :

littéral = variable booléenne (1/vrai ou 0/faux).
clause = plusieurs littéraux connectés par $\vee$
exemple : $v_1 \vee \overline{v_2} \vee \overline{v_3} \vee v_4$
forme normale conjonctive = plusieurs clauses connectées par $\wedge$
exemple : $(v_1 \vee \overline{v_2}) \wedge (\overline{v_3} \vee v_4)$
note 1 : aussi nommée FNC.
note 2 : FNCk = FNC où chacune des clauses a k littéraux.

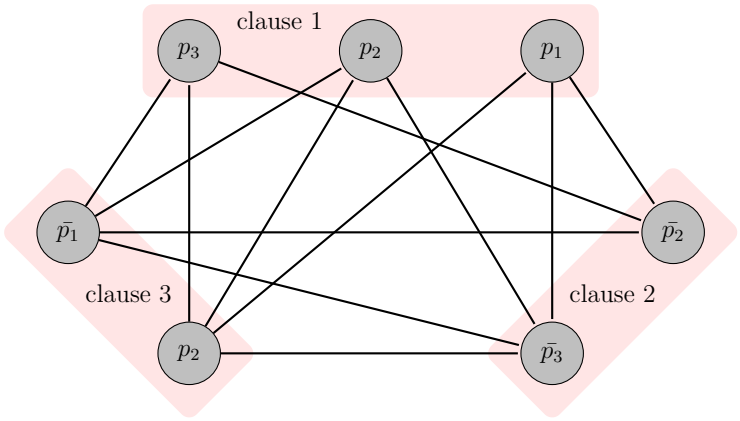
Forme normale conjonctive satisfiable :

Une FNC F est satisfiable s'il existe une combinaison de littéraux telle que F soit vrai.
exemple : $F = (v_1 \vee \overline{v_2}) \wedge (\overline{v_3} \vee v_4)$ est satisfiable ($F = 1$)
avec $v_1 = 1$, $v_2 = 1$, $v_3 = 0$ et $v_4 = 0$.

Graphe associé à une FNC :

- chaque littéral représente un nœud pour chacun des littéraux d'une clause.
- chaque clause représente un ensemble de nœuds non connectés entre eux.
- chaque littéral x d'une clause est connecté à tous les littéraux de toutes les autres clauses, sauf si ce littéral est $\overline{x}$.

Exemple : $(p_1 \vee p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_1} \vee p_2)$



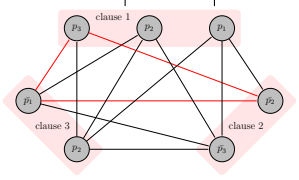
La FNC représente des "blocs" de relations similaires entre nœuds d'un graphe.

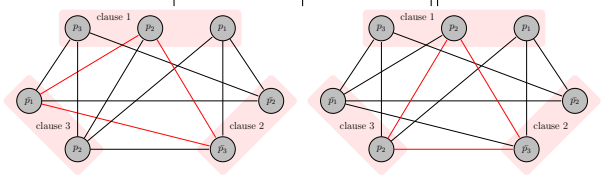
Lien entre FNC et clique

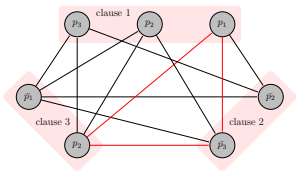
Un FNC à k clauses peut être transformé en un graphe. Si on trouve une k-cliques dans ce graphe, alors la FNC est satisfiable.

Exemple : $(p_1 \vee p_2 \vee p_3) \wedge (\overline{p_2} \vee \overline{p_3}) \wedge (\overline{p_1} \vee p_2)$ (donc ici $k = 3$)

p_1	p_2	p_3	$p_1 \vee p_2 \vee p_3$	$\overline{p_2} \vee \overline{p_3}$	$\overline{p_1} \vee p_2$	p
faux	faux	faux	faux	vrai	vrai	faux
faux	faux	vrai	vrai	vrai	vrai	vrai
faux	vrai	faux	vrai	vrai	vrai	vrai
faux	vrai	vrai	vrai	faux	vrai	faux
vrai	faux	faux	vrai	vrai	faux	faux
vrai	faux	vrai	vrai	vrai	faux	faux
vrai	vrai	faux	vrai	vrai	vrai	vrai
vrai	vrai	vrai	vrai	faux	vrai	faux


$p_1 = p_2 = 0, p_3 = 1$


$p_1 = 0, p_2 = 1, p_3 = 0$


$p_1 = p_2 = 1, p_3 = 0$

Rappel

- $\texttt{SAT}_3 = \{\langle F \rangle \mid F$ est une $\texttt{FNC}_3$ satisfiable$\}$
- $\texttt{CLIQUE} = \{\langle G, q \rangle \mid G$ est un graphe avec une q-clique$\}$

Proposition V.7

$\texttt{SAT}_3 \leq_P \texttt{CLIQUE}$.

Démonstration:

Définissons la fonction f qui construit un graphe G associé à une FNC comme vu dans l'exemple précédent.

① Pour tout $F \in \texttt{SAT}_3$, alors $f(F) \in \texttt{CLIQUE}$, puisque pour une FNC à n clauses, $f(F) \in \texttt{CLIQUE}_n$ et que $\texttt{CLIQUE}_n \subset \texttt{CLIQUE}$.

② Inversement, la construction associant un littéral par nœud, f est clairement inversible. Donc, $f(F) \in \texttt{CLIQUE} \Rightarrow F \in \texttt{SAT}_3$.

③ La construction du graphe se fait en temps polynomial (générer les sommets = $O(3n)$, générer les arêtes = $O(3n^2)$).

Donc f est bien une réduction de $\texttt{SAT}_3$ vers $\texttt{CLIQUE}$ calculable en temps polynomial. □

Conséquence :

- on a vu que : $\texttt{SAT}_3 \leq_P \texttt{CLIQUE}$
- et que si $A \leq_P B$ et $B \in \mathbf{P}$ alors $A \in \mathbf{P}$.

Donc, si $\texttt{CLIQUE} \in \mathbf{P}$ alors $\texttt{SAT}_3 \in \mathbf{P}$ aussi.

Mais peut-on dire quelque chose si $\texttt{CLIQUE} \in \mathbf{NP}$?

8.4 Définition et premières propriétés

Définition V.30 (langage **NP**-complet)

Un langage est B est **NP**-complet si :

1. $B \in \mathbf{NP}$
2. $\forall A \in \mathbf{NP}$, $A \leq_P B$ (= B est **NP**-difficile)

Remarque 34:

Les langages **NP**-complets sont les langages les plus difficiles de **NP**.

Théorème V.10

Si B est **NP**-complet et $B \in \mathbf{P}$ alors $\mathbf{P} = \mathbf{NP}$.

Démonstration:

Si B est **NP**-complet, alors pour tout $A \in \mathbf{NP}$, il existe une réduction en temps polynomial vers B.
Or si $B \in \mathbf{P}$, B est décidable en temps polynomial.
Décider A = réduire à B + décider B ; faisable en temps polynomial. □

Théorème V.11

Si B est **NP**-complet et $B \leq_P C$ et $C \in \mathbf{NP}$ alors **C** est **NP**-complet.

Démonstration:

si B est **NP**-complet, alors $\forall A \in \mathbf{NP}$, $A \leq_P B$. Or $B \leq_P C$ implique $A \leq_P C$ (les deux réductions consécutives sont effectuées en temps polynomial). Donc, si $C \in \mathbf{NP}$, alors C est **NP**-complet puisque $\forall A \in \mathbf{NP}$, $A \leq_P C$. □

8.5 Théorème de Cook-Levin

Avant de passer au théorème, considérons le langage suivant :

Définition V.31

$\texttt{SAT} = \{\langle F \rangle \mid F \text{ est une expression logique satisfiable}\}$

Proposition V.8

$\texttt{SAT} \in \mathbf{NP}$.

Démonstration:

- **avec un vérificateur à temps polynomial** : prendre comme certificat c une chaîne contenant la valeur de vérité de chaque littéral unique de l'expression logique. L'évaluation de l'expression logique à partir des valeurs des littéraux se fait en temps polynomial $O(n)$.

- **avec une MT non déterministe** : choix non déterministe de la valeur de vérité de chaque littéral unique, puis évaluation de l'expression logique. Chaque branche s'exécute trivialement en temps polynomial.

□

Problème principal : nous n'avons pas de problème **NP**-complet. Ceci est résolu avec le théorème suivant.

Théorème V.12 (Cook-Levin)

SAT est **NP**-complet.

DÉMONSTRATION:

Il faut montrer que :

- SAT $\in$ **NP** (fait : voir théorème précédent)
- $\forall A \in$ **NP**, $A \leq_P$ SAT
  Il faut montrer que tout langage de **NP** peut être réduit en temps polynomial en SAT.

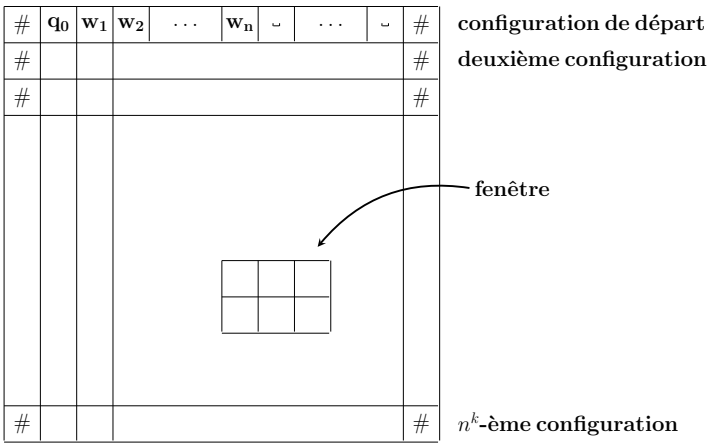
Soit N une MT non déterministe qui décide le langage A.
On veut construire une réduction f telle que :
N accepte $w \Leftrightarrow F$ est satisfiable.
$N \in$ **NP** signifie qu'une branche d'exécution de N s'exécute en temps n^k pour une entrée de taille n, donc :

- une branche d'exécution de N passe par n^k configuration différente
  Rappel : configuration = bande + état courant à sa position.
- N visite au plus n^k cases : une configuration fait au plus n^k symboles.

On définit un tableau (voir page suivante) :

- de taille $n^k \times n^k$ qui représente une branche d'exécution de N sur w.
- qui utilise un alphabet étendu $C = \Gamma \cup Q \cup \{\#\}$ (alphabet de la bande, états et marqueur d'extrémité).

Note : devrait être $n^k + 3$ avec # au début et à la fin + état courant.



La branche d'exécution de N est stockée comme suit dans le tableau :

- La première ligne représente la configuration de depart.
- Les lignes suivantes représentent la suite des configurations dans l'ordre d'exécution de la branche. Chaque ligne peut être déduite de la précédente par les règles de transition de N.
- Si n'importe quelle ligne du tableau est une configuration acceptante, alors le tableau est acceptant.

Donc, un tableau accepte w si N accepte w. Ainsi,

- décider si N accepte w équivaut à décider s'il existe un tableau acceptant pour l'exécution de $N(\langle w \rangle)$.
- il faut trouver une formule logique F permettant de trouver si un tableau acceptant existe.

On veut construire une expression logique F qui permet de décider si un tableau est acceptant lorsque F est satisfiable. F doit garantir que toutes les conditions suivantes sont vraies :

1. Chaque cellule est occupée par exactement 1 symbole (ϕ_{cell}).
2. Le tableau est dans une configuration acceptante (ϕ_{accept}).
3. La première ligne est la configuration d'entrée avec w (ϕ_{start}).
4. La suite des configurations suivantes est cohérente avec une exécution de N (ϕ_{move}).

Autrement dit, F s'écrit sous la forme :

$F = \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}}$

On définit le littéral $x_{i,j,s}$ de la façon suivante.

$x_{i,j,s} = 1$ ssi la cellule (i, j) du tableau stocke le symbole $s \in C$ et 0 sinon.

Expression de ϕ_{cell} :

$f_{i,j,1}$ = la cellule (i, j) contient **au moins** un symbole = $\bigvee_{s \in C} x_{i,j,s}$

$f_{i,j,2}$ = la cellule (i, j) contient **au plus** un symbole = $\bigwedge_{s,t \in C, s \neq t} (\overline{x}_{i,j,s} \vee \overline{x}_{i,j,t})$

Ces conditions doivent être vérifiées pour toutes les cellules du tableau : $\phi_{\text{cell}} = \bigwedge_{1 \leq i,j \leq n^k} f_{i,j,1} \wedge f_{i,j,2} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\overline{x}_{i,j,s} \vee \overline{x}_{i,j,t}) \right) \right]$.

Expression de ϕ_{accept} :

Si il y a l'état q_{accept} n'importe où dans la table.

$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}$

Expression de ϕ_{start} :

Si la première ligne commence par #, suivi par q_0, puis $w = w_1 w_2 \dots w_n$, et complété par des blancs $\sqcup$, et fini par #.

$\phi_{\text{start}} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \dots \wedge x_{1,n+2,w_n}$
$\wedge x_{1,n+3,\sqcup} \wedge \dots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$

Expression de ϕ_{move} :

Fenêtre : définie par sa position (i, j) et sa taille 2×3.

Fenêtre légale : transformation possible par N sur la fenêtre.

a	q_1	b
q_2	a	c

légale si N a une transition $\delta(q_1, b) = (q_2, c, L)$

a	q_1	b
a	a	q_2

légale si N a une transition $\delta(q_1, b) = (q_2, a, R)$

a	a	q_1
a	a	b

légale si N a une transition $\delta(q_1, c) = (q_2, b, R)$

#	a	b
#	a	b

légale (pas de transition dans cette fenêtre)

a	b	a
a	b	q_2

légale si N a une transition $\delta(q_1, b) = (q_2, c, L)$

a	a	a
b	a	a

légale si N a une transition $\delta(q_1, a) = (q_2, b, L)$

a	b	b
a	a	b

non légale (la transition devrait être dans la fenêtre)

a	q_1	b
q_2	a	q_2

non légale (2 transitions sur la 2ème ligne)

a	q_1	a
q_2	c	b

non légale (transition à gauche, caractère à droite modifié)

Expression de ϕ_{move} :

Validité du tableau : si toutes les fenêtres sont légales.

Soit le prédicat Legal(i, j) qui est vrai si la fenêtre (i, j) est légale.

$$\text{Legal}(i, j) = \bigvee_{\{a_1,\ldots,a_6\} \text{ est légale}} x_{i,j,a_1} \wedge x_{i,j,a_2} \wedge \ldots \wedge x_{i,j,a_6}$$

Autrement dit, Legal(i, j) est l'union de toutes les configurations légales sur une fenêtre 3×2 en accord avec le fonctionnement de N sur deux configurations consécutives.

En conséquence, $\phi_{\text{move}} = \bigwedge_{1 \leq i,j \leq n^k} \text{Legal}(i, j)$

Note : raison pour laquelle l'application du prédicat ϕ_{move} produit une suite de configuration valide correspondant à l'exécution d'une MT :

- la première ligne du tableau est une ligne valide (entrée w de la MT assurée par ϕ_{start}).
- la fenêtre 3×2 permet de produire (par récurrence) une configuration valide (la suivante) à partir de la configuration précédente (valide pour i=1).

Conséquence de cette construction :

$\boxed{\Rightarrow}$ F satisfiable

$\Rightarrow$ F représente l'affectation d'un tableau acceptant.

$\Rightarrow$ le tableau acceptant représente une branche d'exécution de N sur w.

$\Rightarrow$ N accepte l'entrée w.

$\boxed{\Leftarrow}$ Inversement, si N accepte l'entrée w,

$\Rightarrow$ la suite des configurations se représente comme un tableau acceptant.

$\Rightarrow$ la formule F équivalente est satisfiable.

$\Rightarrow F$ satisfiable.

Donc, pour tout MT N non déterministe à temps polynomial :
N accepte $w \Leftrightarrow F$ est satisfiable
Cette construction est donc une réduction de tout langage de **NP** vers SAT.

Montrons maintenant que cette réduction peut être faite en temps polynomial.
Notons tous d'abord que :

- l'alphabet étendu C utilisé pour coder le tableau (on notera $c = \#C$).
- la MT N

ne dépendent pas de n.
Alors, le temps de calcul est de l'ordre de :

ϕ_{cell} : $f_{i,j,1}$ et $f_{i,j,2}$ ne dépendent que de c. Donc ϕ_{cell} est en $O(n^{2k})$.
ϕ_{accept} : Trivialement de l'ordre de $O(n^{2k})$.
ϕ_{start} : Trivialement de l'ordre de $O(n^k)$.
ϕ_{move} : Legal(i, j) ne dépend que de c. Donc ϕ_{move} est en $O(n^{2k})$.

Par conséquent la réduction s'effectue en temps polynomial.
Donc, $\forall A \in \mathbf{NP}$, $A \leq_P$ SAT $\Rightarrow$ SAT.
On en déduit que SAT est **NP**-complet. □

8.6 Autres problèmes NP-complets

8.6.1 FNC-SAT

Définition V.32
FNC-SAT = $\{\langle F \rangle \mid F$ est une FNC satisfiable$\}$

Théorème V.13
FNC-SAT est **NP**-complet.

Démonstration:

- FNC-SAT $\in$ **NP** Même ligne de preuve que pour SAT.
- $\forall A \in \mathbf{NP}$, $A \leq_P$ FNC-SAT. la preuve du théorème du Cook-Levin peut être directement réutilisée car la réduction utilise des FNCs.

Donc, FNC-SAT est **NP**-complet. □

Définition V.33
$\text{SAT}_3 = \{\langle F \rangle \mid F$ est une FNC_3 satisfiable$\}$

Proposition V.9
SAT_3 est **NP**-complet.

Démonstration:

- $\text{SAT}_3 \in \mathbf{NP}$. Même ligne de preuve que pour SAT.
- $\forall A \in \mathbf{NP}$, $A \leq_P \text{SAT}_3$. Pour se faire, on réduit FNC-SAT à SAT_3. Soit $F = \bigwedge_i C_i$ où C_i sont les clauses de F. On a alors 3 cas :

— C_i **a moins de 3 littéraux** : il suffit de dupliquer l'un des littéraux.
Exemple : $C_i = L_1 \Rightarrow C'_i = L_1 \vee L_1 \vee L_1$
— C_i **a 3 littéraux** : on le laisse tel quel.
— C_i **a plus de 3 littéraux** : notons $C_i = L_1 \vee L_2 \vee \ldots \vee L_m$.
Introduire des nouveaux littéraux z_i de la façon suivante :

$$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee z_3) \wedge \ldots \wedge (\overline{z}_{m-3} \vee L_{m-1} \vee L_m)$$

Si C_i est satisfiable, alors il existe $(z_1, z_2, \ldots, z_{m-3})$ tels que C'_i soit aussi satisfiable.
Exemple :

$C'_i = (L_1 \vee L_2 \vee z_1) \wedge (\overline{z}_1 \vee L_3 \vee z_2) \wedge (\overline{z}_2 \vee L_4 \vee L_5)$
Si C_i n'est pas satisfiable, alors il n'existe aucune combinaison de (z_1, z_2) qui rende C'_i satisfiable, sinon il existe une combinaison de (z_1, z_2) qui rend C'_i satisfiable.

Soit $F \in$ FNC-SAT satisfiable. Soit F' construit à partir de F par la méthode indiquée. Alors $F' \in \mathtt{SAT}_3$ est satisfiable. Inverse évident car $\mathtt{SAT}_3 \subset$ FNC-SAT. Comme FNC-SAT $\leq_P \mathtt{SAT}_3$ et FNC-SAT **NP**-complet, alors $\mathtt{SAT}_3$ aussi.

□

8.6.2 CLIQUE

Rappel : CLIQUE $= \{\langle G, k\rangle \mid G$ est une $k -$ CLIQUE $\}$

Proposition V.10
CLIQUE est **NP**-complet.

Démonstration:

- CLIQUE $\in$ **NP** : déjà démontré.
- $\forall A \in$ **NP**, $A \leq_P$ CLIQUE. On a déjà montré que $\mathtt{SAT}_3 \leq_P$ CLIQUE. Comme $\mathtt{SAT}_3$ **NP**-complet, alors CLIQUE aussi.

□

On voit que l'on dispose d'une méthode facile pour montrer qu'un problème K est **NP**-complet.

- trouver un autre problème K' qui soit **NP**-complet et "proche" de K
- montrer que $K \in$ **NP**.
- montrer qu'il existe une réduction en temps polynomial de K' vers K.
- en déduire que K est **NP**-complet.

8.6.3 SUBSET-SUM

Rappel : SUBSET-SUM $= \{\langle S, t\rangle \mid \exists S' \subseteq S = \{s_1, s_2, \ldots, s_n\} / \Sigma_{s_i \in S'} s_i = t\}$

Proposition V.11
SUBSET-SUM est **NP**-complet.

DÉMONSTRATION:

- SUBSET-SUM $\in$ **NP** : déjà démontré.
- Par réduction en temps polynomial de $\mathtt{SAT}_3$ vers SUBSET-SUM.

  Soit $F = \bigwedge_{i=1\ldots n} C_i$ une expression logique sous forme de conjonction de clauses C_i, chaque clause étant composée de la disjonction de 3 littéraux parmi $x_1, x_2, \ldots, x_p$ ou de leurs négations.

  On cherche une transformation telle que :

  — chaque littéral x_j possède une valeur unique (vrai ou faux).

  — chaque clause C_i soit vraie.

  Construisons un ensemble d'entiers à $p + n$ chiffres où l'on affecte une propriété à chaque chiffre.

Le rôle de chaque chiffre est le suivant :

- les p premiers chiffres représentent les littéraux x_j. Le $j^{\text{ème}}$ chiffre représente le littéral x_j.
- les n chiffres suivants représentent les clauses C_i. Le $(p+i)^{\text{ème}}$ chiffre représente la clause C_i.

On va donc créer un premier ensemble de $2p$ entiers à $n + p$ chiffres, où tous leurs chiffres sont à 0, sauf pour :

a_j (associé à x_j) son $j^{\text{ème}}$ chiffre à 1 si x_j est **vrai**.
et partout où x_j apparaît dans C_i, son $(p + i)^{\text{ème}}$ chiffre est à 1.

$\overline{a}_j$ (associé à $\overline{x}_j$) son $j^{\text{ème}}$ chiffre à 1 si x_j est **faux**.
et partout où $\overline{x}_j$ apparaît dans C_i, son $(p + i)^{\text{ème}}$ chiffre est à 1.

Quelle valeur de t faut-il alors choisir ?
Il faut nécessairement que chaque a_j ou (exclusif) $\overline{a}_j$ soit dans S'.
$\Rightarrow$ les n premiers chiffres de t sont 1.
$\Rightarrow$ les p chiffres suivants de t sont supérieurs à 1.

Exemple : $F = C_1 \wedge C_2 = (x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_2 \vee \overline{x}_3 \vee x_4)$

	x_1	x_2	x_3	x_4	C_1	C_2
a_1	1	0	0	0	1	0
$\overline{a}_1$	1	0	0	0	0	0
a_2	0	1	0	0	0	1
$\overline{a}_2$	0	1	0	0	1	0
a_3	0	0	1	0	1	0
$\overline{a}_3$	0	0	1	0	0	1
a_4	0	0	0	1	0	1
$\overline{a}_4$	0	0	0	1	0	0

Exemples de choix :

$a_1 + \overline{a}_2 + a_3 + a_4 =$	1	1	1	1	3	1	valide / F vrai
$a_1 + a_2 + \overline{a}_3 + \overline{a}_4 =$	1	1	1	1	1	2	valide / F vrai
$\overline{a}_1 + a_2 + \overline{a}_3 + a_4 =$	1	1	1	1	0	3	valide / F faux
$a_1 + \overline{a_1} + a_3 =$	2	0	1	0	2	0	invalide / F indéfini

Pour la partie attribut : le n premiers chiffres sont 1.
Pour la partie clause : Chaque chiffre est entre 1 et 3.
Pas de t unique si on fait comme cela.
Solution : Choisir $t = 1 \ldots 13 \ldots 3$ en ajoutant 2 variables identiques supplémentaires par clause b_j et b'_j (donc $2p$ variables) définies par le $(n+j)^{\text{ème}}$ chiffre est à 1 et les autres à 0.
Il y a alors 3 cas différents pour la $(n+j)^{\text{ème}}$ colonne :

$= 3$ pas de problème (les 3 littéraux sont vrais).

$= 2$ (resp. 1), alors ajouter b_j **ou** b'_j (resp. b_j **et** b'_j) pour arriver à 3.

$= 0$ alors même en ajoutant b_j et b'_j, impossible de faire 3.

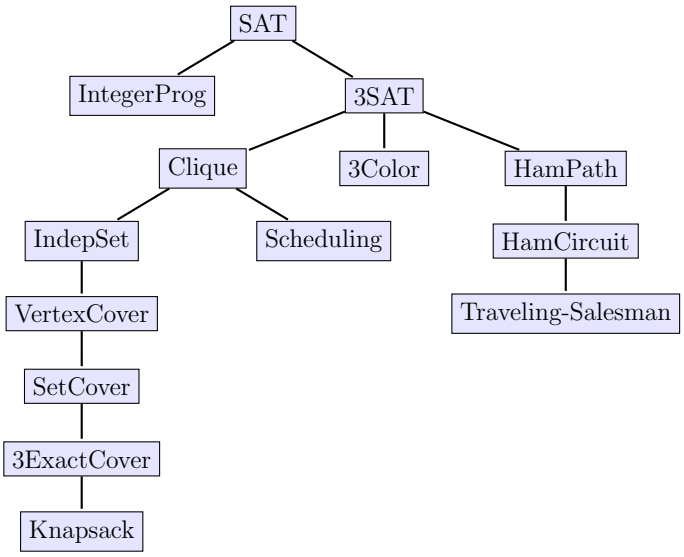
Donc, avec $S = \{a_1,\ldots,a_n,\bar{a}_1,\ldots,\bar{a}_n,b_1,\ldots,b_p,b'_1,\ldots,b'_p\}$ et $t = \underbrace{1\ldots1}_{n \text{ fois}}\underbrace{3\ldots3}_{p \text{ fois}}$.
On a donc trivialement ($\langle F\rangle \in \texttt{SAT}_3 \Leftrightarrow f(\langle F\rangle) \in \texttt{SUBSET-SUM}$).
La réduction f s'effectue clairement en temps polynomial. En conséquence, $\texttt{SAT}_3 \leq_P \texttt{SUBSET-SUM}$ et $\texttt{SAT}_3$ **NP**-complet implique `SUBSET-SUM` **NP**-complet.

□

8.7 Hiérarchies des problèmes NP-complet

De nombreux problèmes **NP**-complets ont été trouvés à ce jour.

Ils constituent une hiérarchie formée par les réductions en temps polynomial pour passer de l'un à l'autre.



Exercice 46 (`SET_PARTITION`)**.**
Soit :

$$\texttt{SET\_PARTITION} = \left\{\langle S\rangle \mid \exists A \subset S \ tq \ \sum_{x_i \in A} x_i = \sum_{x_j \in \overline{A}} x_j\right\} \text{ où } S = \{x_1, \ldots x_n\}$$

est un ensemble d'entiers et $\overline{A} = S \setminus A$.

Montrer que `SET_PARTITION` *est* ***NP****-complet.*

indice : *on pourra utiliser une réduction à* `SUBSET_SUM`*.*

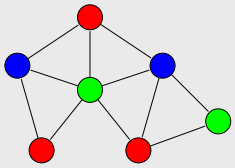
Exercice 47 (DOUBLE-SAT). *Soit DOUBLE-SAT = {⟨Φ⟩ | Φ est une formule booléenne avec 2 assignations vraies }.*

À savoir soit $B = \{b_1, \ldots, b_p\}$ les littéraux de Φ alors il existe 2 assignations de Φ (par exemple $B_1 = \{1, 0, \ldots, 1\}$ et $B_2 = \{0, 1, \ldots, 1\}$) telles ques $\Phi(B_1)$ et $\Phi(B_2)$ soient vraies.

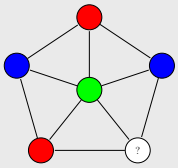
*Montrer que DOUBLE-SAT est **NP**-complet.*

Exercice 48 (3COLOR). *Soit 3COLOR = {⟨G⟩ | G est un graphe dont les nœuds peuvent être colorés avec 3 couleurs tel que aucun couple de nœuds reliés par une arête ne soit de la même couleur }.*

Exemple :



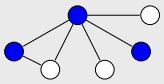
coloriage en 3 couleurs



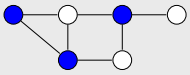
coloriage impossible

*Montrer que 3COLOR est **NP**-complet.*
***indice :** on pourra utiliser une réduction de 3SAT à 3COLOR.*

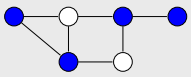
Exercice 49 (VERTEX-COVER). *Une couverture de sommets dans un graphe est un sous-ensemble de sommets tel que chaque arête touche l'un de ses sommets. **Exemple :***



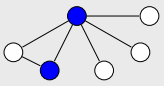
couverture



couverture



couverture minimale



couverture minimale

Soit VERTEX-COVER = {⟨G, k⟩ | G est un graphe non orienté qui a une couverture à k sommets}.

1. *Montrer que VERTEX-COVER ∈ **NP**.*
   ***a)** avec un vérificateur à temps polynomial.*
   ***b)** avec une machine de Turing non déterministe à temps polynomial.*
2. *Montrer que $3SAT \leq_{\mathbb{P}}$ VERTEX-COVER.*
3. *En déduire que VERTEX-COVER est **NP**-complet.*

8.8 *co*NP-complétude

On peut également définir les ensembles complémentaires pour les ensembles **NP**.

Définition V.34 (classe *co*NP)

Un langage C appartient à *co*NP si $\overline{C} \in$ NP.

Définition V.35 (*co*NP-complétude)

Un langage C est *co*NP-complet si :

- $\overline{C} \in$ NP.
- $\forall D \in$ *co*NP, $D \leq_p C$.

9 Liens entre classes de complexité

Nous donnons maintenant différentes relations qui peuvent être déduites des propriétés précédentes.

Proposition V.12

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$$

Démonstration:

Conséquences du théorème V.8 :

- $\mathbf{P} \subseteq \mathbf{NP}$: par le théorème V.8, $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n))$, et l'utiliser avec $t(n) = n^k$, $\forall k$. Conclure avec $\cup_k \texttt{TIME}(n^k) \subseteq \cup_k \texttt{NTIME}(n^k) \Leftrightarrow \mathbf{P} \subseteq \mathbf{NP}$.
- $\mathbf{NP} \subseteq \mathbf{EXP}$: par le théorème V.8, $\texttt{NTIME}(t(n)) \subseteq \texttt{TIME}(2^{O(t(n))})$, et l'utiliser avec $t(n) = n^k$, $\forall k$. Conclure avec $\cup_k \texttt{NTIME}(n^k) \subseteq \cup_k \texttt{TIME}(2^{n^k}) \Leftrightarrow \mathbf{NP} \subseteq \mathbf{EXP}$.
- $\mathbf{EXP} \subseteq \mathbf{NEXP}$: par le théorème V.8, $\texttt{TIME}(t(n)) \subseteq \texttt{NTIME}(t(n))$, et l'utiliser avec $t(n) = 2^{n^k}$, $\forall k$. Conclure avec $\cup_k \texttt{TIME}(2^{n^k}) \subseteq \cup_k \texttt{NTIME}(2^{n^k}) \Leftrightarrow \mathbf{EXP} \subseteq \mathbf{NEXP}$.

□

Nous donnons maintenant différentes relations qui peuvent être déduites des propriétés précédentes.

Proposition V.13

$$\mathbf{P} \subsetneq \mathbf{EXP}$$

Démonstration:

Conséquences du théorème V.8 : On a $\mathbf{P} \subseteq \texttt{TIME}(2^n)$. Par le théorème de Hiérarchie, on a $\texttt{TIME}(2^n) \subsetneq \texttt{TIME}(2^{n^2})$ (avec $f(n) = 2^n$ et $g(n) = 2^{n^2}$). Comme $\texttt{TIME}(2^{n^2}) \subseteq \mathbf{EXP}$, on en déduit que $\mathbf{P} \subsetneq \mathbf{EXP}$. □

Proposition V.14

$$\mathbf{NP} \subsetneq \mathbf{NEXP}$$

Démonstration:

Conséquences du théorème V.8 : On a $\mathbf{NP} \subseteq \texttt{NTIME}(2^n)$. Par le théorème de Hiérarchie non déterministe, on a $\texttt{NTIME}(2^n) \subsetneq \texttt{NTIME}(2^{n^2})$ (avec $f(n) = 2^n$ et $g(n) = 2^{n^2}$). Comme $\texttt{NTIME}(2^{n^2}) \subseteq \mathbf{NEXP}$, on en déduit que $\mathbf{NP} \subsetneq \mathbf{NEXP}$. □

Proposition V.15

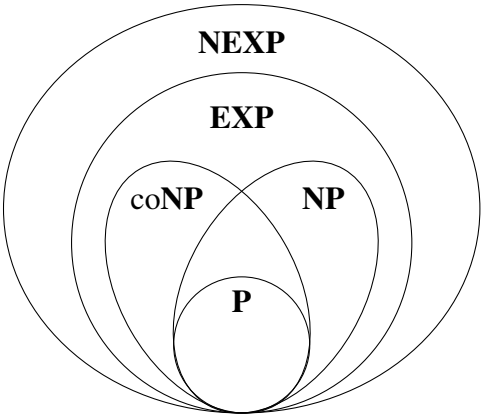
$$\mathbf{P} \subset \mathbf{NP} \cap co\mathbf{NP}$$

Démonstration:

On sait que $\mathbf{P} \subset \mathbf{NP}$ et que $co\mathbf{P} \subset co\mathbf{NP}$. Comme $\mathbf{P} = co\mathbf{P}$, on en déduit $\mathbf{P} \subset \mathbf{NP} \cap co\mathbf{NP}$. □

Ces nouveaux ensembles conduisent à la définition du monde de la complexité algorithmique tel qu'on le pense actuellement, c'est à dire sous les hypothèses que :

- $\mathbf{P} \neq \mathbf{NP}$
- $\mathbf{EXP} \neq \mathbf{NEXP}$
- $\mathbf{NP} \neq co\mathbf{NP}$


NEXP
EXP
coNP
NP
P