Séance 6 : Décidabilité et Complexité

Objet : Sensibiliser les étudiants aux principes de la décidabilité et de la complexité des algorithmes

---

Plan

- Rappel de Calculabilité

- Décidabilité

- Complexité algorithmique

VI.1 Rappel de calculabilité

La *calculabilité* cherche d'une part à identifier la classe des fonctions qui peuvent être calculées à l'aide d'un algorithme et à appliquer ces concepts à des questions fondamentales des mathématiques. Une bonne appréhension de ce qui est calculcable et de ce qui ne l'est pas permet de voir les limites des problèmes que peuvent résoudre les ordinateurs.

Il peut être démontré qu'il existe des fonctions f qui sont incalculables, c'est à dire qu'il n'existe aucun algorithme qui, étant donné x, retourne toujours la valeur $f(x)$ en un temps fini. L'exemple le plus courant est celui du problème de l'arrêt : il n'existe pas de programme universel qui prenne n'importe quel programme en argument et qui, en temps fini, renvoie « oui » si l'exécution du programme reçu en argument finit par s'arrêter et « non » s'il ne finit pas.

Plusieurs modèles de calcul sont utilisés en calculabilité:

machines de Turing
machine à compteurs
lambda-calcul
automates cellulaires
fonctions récursives
etc.

Malgré la diversité de ces modèles, la classe de fonctions calculables par chacun de ceux-ci est unique et cette constatation mène à la thèse Church-Turing.

La **thèse de Church-Turing**, ou simplement **thèse de Church**, des noms des mathématiciens Alonzo Church et Alan Turing, est une idée qui se rattache au domaine de l'informatique. *Dans sa forme la plus ordinaire, elle affirme que tout traitement réalisable mécaniquement peut être accompli par une machine de Turing. Tout programme d'ordinateur peut donc être traduit en une machine de Turing.*

D'autre part, certaines machines de Turing, dites *universelles*, peuvent effectuer tous les traitements possibles avec une machine de Turing quelconque. La plupart des langages de programmation usuels ont (plus exactement, auraient sur un ordinateur disposant d'une mémoire infinie) les possibilités de calcul d'une machine de Turing universelle, de sorte que toutes les machines de Turing peuvent être simulées par un programme écrit dans l'un de ces langages.

La thèse de Church-Turing affirme donc que n'importe quel langage de programmation (Turing-complet) permet d'exprimer tous les algorithmes.

La thèse de Church-Turing est généralement considérée comme vraie. Mais ce n'est pas un énoncé mathématique : chercher à la démontrer n'a pas de sens. Elle serait en revanche réfutée si un calcul que l'on s'accorde à considérer comme réalisable mécaniquement s'avérait hors de portée des machines de Turing.

La thèse peut être reformulée en disant que les machines de Turing formalisent correctement la notion de méthode effective de calcul. On considère généralement qu'une méthode effective doit satisfaire aux obligations suivantes :

l'algorithme consiste en un ensemble fini d'instructions simples et précises qui sont décrites avec un nombre limité de symboles ;
l'algorithme doit toujours produire le résultat en un nombre fini d'étapes ;
l'algorithme peut en principe être suivi par un humain avec seulement du papier et un crayon ;
l'exécution de l'algorithme ne requiert pas d'intelligence de l'humain sauf celle qui est nécessaire pour comprendre et exécuter les instructions.

Un exemple d'une telle méthode est l'algorithme d'Euclide pour déterminer le plus grand commun diviseur d'entiers naturels.

Il s'agit là d'une définition intuitive assez claire, mais pas d'une définition formelle, faute d'avoir précisé ce qu'on entend par « instruction simple et précise » ou par « l'intelligence requise pour exécuter les instructions ». On peut en revanche *définir* formellement les algorithmes comme les procédés qui peuvent être accomplis par une machine de Turing universelle. À ce stade, la thèse de Church-Turing affirme que les deux définitions, intuitive et formelle, concordent.

VI.2- Décidabilité

Intuitivement, P est décidable s'il existe un algorithme qui pour chaque x dit "OUI" ou "NON" à la question : "Est-ce que P(x) est vrai ?".

Ce problème avait été posé la première fois par David Hilbert, au Congrès International des Mathématiciens qui avait eu lieu à Paris en 1900. Hilbert, comme beaucoup de mathématiciens de l'époque, était assez obsédé par de nombreux problèmes de mathématiques qui ne trouvaient pas de solution. Ainsi celui des *équations diophantiennes*. On appelle équation diophantienne une équation du genre P(x, y, z, ...) = 0, où P est un polynôme à coefficients entiers. Résoudre une telle équation, c'est chercher les solutions sous forme d'entiers.

Par exemple : $x2 + y2 - 1 = 0$ est une équation diophantienne qui admet pour solutions: $x = 1$, $y = 0$ et $x = 0$, $y = 1$.
L'équation $x2 - 991y2 - 1 = 0$ est également diophantienne mais a des solutions beaucoup plus difficiles à trouver (cf J. P. Delahaye, in "Logique, informatique et paradoxes" ed. Belin), la plus petite est : x = 379 516 400 906 811 930 638 014 896 080 et y = 12 055 735 790 331 359 447 442 538 767

Hilbert posait la question : "existe-t-il un procédé mécanique permettant de dire, après un nombre fini d'étapes, si une équation diophantienne donnée possède des solutions entières?".

On doit noter que, parmi ces équations, figurent certaines, qui sont associées à un problème historiquement bien connu: *le problème de Fermat*. Fermat pensait en effet avoir démontré que toute équation de la forme : $xp + yp = zp$ est sans solution pour $p > 2$. (Pour $p = 2$, bien sûr... elle en a au moins une, cf: $x = 3$, $y = 4$, $z = 5$).
De fait, on ne disposait pas, jusqu'à une date récente (Andrew Whiles, 1993) d'une telle démonstration.
Au-delà de ce type de problème, Hilbert posait la question générale de savoir si l'on pouvait trouver un procédé mécanisable capable de résoudre toutes les questions mathématiques récalcitrantes. C'est à partir de cette dernière question que Turing s'est mis au travail. Elle est plus générale que la première question: en effet, on pourrait imaginer qu'il n'y ait pas de procédure uniforme pour résoudre tous les problèmes mathématiques mais qu'il y en ait une pour résoudre la problème particulier des équations diophantiennes. En fait, ni l'une ni l'autre n'existe.

En logique mathématique, le terme **décidabilité** recouvre deux concepts liés : la décidabilité *logique* et la décidabilité *algorithmique*.

L'indécidabilité est la négation de la décidabilité. Dans les deux cas il s'agit de formaliser l'idée qu'on ne peut pas toujours conclure lorsque l'on se pose une question, même si celle-ci est sous forme logique.

VI.2.1- Décidabilité, indécidabilité d'un énoncé dans un système logique

Définition VI.2.1.1- décidabilité
Une proposition (on dit aussi énoncé) est dite **décidable** dans une théorie axiomatique, si on peut la démontrer ou démontrer sa négation dans le cadre de cette théorie.

Un énoncé mathématique est donc indécidable dans une théorie s'il est impossible de le déduire, ou de déduire sa négation, à partir des axiomes.

Remarque :
Pour distinguer cette notion d'indécidabilité de la notion d'indécidabilité algorithmique (voir ci-dessous), on dit aussi que l'énoncé est *indépendant* du système d'axiomes.

En termes plus concrets, cela veut dire qu'on demande au système de fournir une conclusion sans lui avoir fourni suffisamment d'hypothèses. Ainsi, l'âge du capitaine d'un bateau est indécidable en fonction du tonnage et de la vitesse du navire.

Proposition VI.2.1.1-
En logique classique, d'après le théorème de complétude, une proposition est indécidable dans une théorie s'il existe des modèles de la théorie où la proposition est fausse et des modèles où elle est vraie.

On utilise souvent des modèles, pour montrer qu'un énoncé est indépendant d'un système d'axiomes (dans ce cadre on préfère employer indépendant qu'indécidable). La propriété utilisée dans ce cas n'est pas le théorème de complétude mais sa réciproque, très immédiate, appelée parfois fidélité. Probablement est-ce là d'ailleurs la première apparition de la notion de modèle, avec la construction au XIXème siècle de modèles des géométries non classiques, ne vérifiant pas l'axiome des parallèles.

Exemple :
Si l'on admet le fait assez intuitif que la géométrie euclidienne est cohérente — la négation de l'axiome des parallèles ne se déduit pas des autres axiomes — l'axiome des parallèles est bien alors indépendant des autres axiomes de la géométrie, ou encore indécidable dans le système formé des axiomes restant.

Définition VI.2.1.2- complétude
Une théorie mathématique pour laquelle tout énoncé est décidable est dite complète, sinon elle est dite *incomplète*.

Beaucoup de théories mathématiques sont naturellement incomplètes, parce qu'il y a évidemment des énoncés qui ne sont pas déterminés par les axiomes (théorie des groupes, des anneaux, ...) . Certaines théories, comme la théorie des corps algébriquement clos, celle des corps réels clos, ou encore l'arithmétique de Presburger sont complètes.

Le premier théorème d'incomplétude peut être énoncé de la façon encore un peu approximative suivante :

Premier Théorème d'incomplétude de Gödel :
Dans n'importe quelle théorie récursivement axiomatisable, cohérente et capable de « formaliser l'arithmétique », on peut construire un énoncé arithmétique qui ne peut être ni prouvé ni réfuté dans cette théorie.

De tels énoncés sont dits indécidables dans cette théorie. On dit également *indépendant* de la théorie.

Toujours dans l'article de 1931, Gödel en déduit le second théorème d'incomplétude :

Deuxième Théorème d'incomplétude de Gödel :
Si T est une théorie cohérente qui satisfait des hypothèses analogues, la cohérence de T, qui peut s'exprimer dans la théorie T, n'est pas démontrable dans T.

Ces deux théorèmes ont été prouvés pour l'arithmétique de Peano et donc pour les théories plus fortes que celle-ci, en particulier les théories destinées à fonder les mathématiques, telles que la théorie des ensembles, ou les Principia Mathematica...

Le théorème d'incomplétude de Gödel nous garantit que toute théorie axiomatique cohérente, et suffisamment puissante pour représenter l'arithmétique de Peano (l'arithmétique usuelle), est incomplète, pourvu qu'elle soit axiomatisée de façon que l'on puisse décider au sens

algorithmique si un énoncé est ou non un axiome. Cette dernière hypothèse qui semble un peu compliquée à énoncer est très naturelle et vérifiée par les théories axiomatiques usuelles en mathématiques.

VI.2.2- Décidabilité, indécidabilité d'un problème de décision

Définition VI.2.2.1 : décidabilité algorithmique
Un problème de décision est dit décidable s'il existe un *algorithme*, une *procédure mécanique* qui termine en un nombre fini d'étapes, qui le décide, c'est-à-dire qui réponde par oui ou par non à la question posée par le problème.
S'il n'existe pas de tels algorithmes, le problème est dit indécidable.

Par exemple, le problème de l'arrêt est indécidable. On peut formaliser la notion de fonction calculable par algorithme, ou par procédure mécanique de diverses façons, comme par exemple en utilisant les machines de Turing. Toutes les méthodes utilisées se sont révélées équivalentes dès qu'elles étaient suffisamment générales, ce qui constitue un argument pour la thèse de Church-Turing : les fonctions calculables par une procédure mécanique sont bien celles qui sont calculées selon l'un de ces modèles de calcul. La thèse de Church-Turing est indispensable pour interpréter de la façon attendue une preuve d'indécidabilité.

Remarque :
En cas d'ambiguïté possible, on peut parler d'*indécidabilité algorithmique*, pour distinguer cette notion de l'*indécidabilité logique* exposée dans le paragraphe précédent (ou parfois de décidabilité au sens de Turing pour la décidabilité algorithmique, et de décidabilité au sens de Gödel pour la décidabilité logique).

Le problème décidable algorithmique réfère à la notion de *calculabilité* en cherchant un algorithme qui s'arrête à un temps fini et fournit la réponse OUI ou NON à la question posée.

Dire qu'un problème est indécidable ne veut pas dire que les questions posées sont insolubles mais seulement qu'il n'existe pas de méthode unique et bien définie, applicable d'une façon mécanique, pour répondre à toutes les questions, en nombre infini, rassemblées dans un même problème.

Définition VI.2.2.2 : Ensembles décidables et indécidables
Un sous-ensemble des entiers naturels est dit décidable, quand le problème de l'appartenance d'un entier quelconque à cet ensemble est décidable, indécidable sinon.
On généralise directement aux n-uplets d'entiers. On dit aussi d'un ensemble décidable qu'il est récursif. Le complémentaire d'un ensemble décidable est décidable.
On montre en théorie de la calculabilité qu'un ensemble récursivement énumérable dont le complémentaire est récursivement énumérable est récursif (c'est-à-dire décidable).

On généralise ces notions aux langages formels, par des codages « à la Gödel ». Il est possible aussi de les définir directement.

Remarque :
Dans le cas des théories logiques (closes, donc par déduction), on parle alors de théorie décidable, ou de théorie indécidable. Ces notions ne doivent pas être confondues avec celles de *théorie complète* et *théorie incomplète* vues au paragraphe précédent. Quand on parle d'une théorie décidable ou indécidable, il s'agit *forcément* de décidabilité algorithmique et *jamais* de

décidabilité logique. A contrario, quand on parle d'énoncé ou de proposition décidable ou indécidable, c'est forcément de décidabilité logique qu'il s'agit.

VI.2.3- Exemples d'ensembles et de problèmes décidables indécidables

Exemple 1 : problèmes décidables

- Tous les sous-ensembles finis des entiers sont décidables (il suffit de tester l'égalité à chacun des entiers de l'ensemble).
- On peut construire un algorithme pour décider si un entier naturel est pair ou non (on fait la division par deux, si le reste est zéro, le nombre est pair, si le reste est un, il ne l'est pas), donc l'ensemble des entiers naturels pairs est décidable ;
- il en est de même de l'ensemble des nombres premiers.
- Le problème de savoir si une proposition est vraie dans l'arithmétique de Presburger, c'est-à-dire dans la théorie des nombres entiers naturels avec l'addition mais sans la multiplication, est décidable.
- La question de savoir si oui ou non un énoncé de la logique du premier ordre est universellement valide (démontrable dans toute théorie), dépend de la signature du langage choisie (les symboles d'opération ou de relation ...). Ce problème, parfois appelé problème de la décision. Pour un langage égalitaire du premier ordre ne contenant que des symboles de prédicat unaires (calcul des prédicats égalitaire monadique), le problème est décidable.

Remarque :

*Notons qu'un ensemble peut être théoriquement décidable sans qu'en pratique la décision puisse être faite, parce que celle-ci nécessiterait trop de temps (plus que l'âge de l'univers) ou trop de ressources (plus que les atomes de l'univers). L'objet de la **théorie de la complexité** est d'étudier les problèmes de décision en prenant en compte ressource et temps de calcul.*

Exemple 2 : problèmes indécidables

- Le problème de l'arrêt.

Dans ce cas, les questions portent sur tous les programmes informatiques (dans un langage suffisamment puissant, tel que tous ceux qui sont utilisés en pratique) et sur tous les états initiaux possibles de la mémoire (définis par une quantité finie d'information). Il s'agit de savoir si oui ou non un ordinateur s'arrêtera lorsqu'il exécute un programme à partir de l'état initial de la mémoire. L'indécidabilité du problème de l'arrêt a été prouvée par Alan Turing.

- Plus généralement le théorème de Rice énonce que toute question sur les programmes informatiques qui ne dépend que du résultat du calcul (terminer ou non, valeur calculée etc.) est indécidable ou triviale (ici, « trivial » s'entend par : la réponse est toujours oui ou toujours non).
- La question de savoir si oui ou non un énoncé de la logique du premier ordre est universellement valide (démontrable dans toute théorie), dépend de la signature du langage choisie (les symboles d'opération ou de relation ...). Ce problème, parfois appelé problème de la décision, est indécidable pour le langage de l'arithmétique, et plus généralement pour n'importe quel langage égalitaire du premier ordre qui contient au moins un symbole de relation binaire (comme $<$ ou $\leq$).
- La question de savoir si oui ou non une proposition énoncée dans le langage de l'arithmétique (il faut les deux opérations, $+$ et $\times$) est vraie dans le modèle standard de l'arithmétique est indécidable (c'est une conséquence du premier théorème d'incomplétude de Gödel).
- La prouvabilité d'un énoncé à partir des axiomes de l'arithmétique de Peano est indécidable. Gödel a montré que cet ensemble est strictement inclus dans le précédent. Comme l'axiomatique de Peano a une infinité d'axiomes, cela ne se déduit pas directement de

l'indécidabilité du problème de la décision dans le langage, énoncée précédemment. Les deux résultats se déduisent d'un résultat général pour les théories arithmétiques qui satisfont certaines conditions.
- La prouvabilité d'un énoncé à partir des axiomes d'une théorie des ensembles cohérente, et plus généralement de toute théorie cohérente qui permet d'exprimer « suffisamment » d'arithmétique formelle est indécidable.
- La question de savoir si oui ou non une équation diophantienne a une solution. La preuve de son indécidabilité est le théorème de Matiyasevich (1970)
La question de savoir si oui ou non deux termes du lambda-calcul sont β-équivalents, ou de façon similaire, l'identité de deux termes de la logique combinatoire. Son indécidabilité a été prouvée par Alonzo Church.

VI.2.4- Logiques décidables

Une logique est décidable s'il existe un algorithme qui réponde toujours par oui ou non à la question de savoir si un énoncé donné est démontrable dans cette logique. Un tel algorithme peut être facilement étendu en un algorithme de recherche de démonstration formelle : une fois que l'on sait qu'un énoncé est démontrable, il suffit d'énumérer toutes les démonstrations bien formées jusqu'à trouver une démonstration de cet énoncé. Cet algorithme de recherche n'a bien sûr qu'un intérêt théorique, sauf dans des cas particulièrement simples.

Remarque :
Même si une logique est décidable, la complexité algorithmique de sa décision peut être rédhibitoire.

Exemples de logiques décidables :

la théorie des corps réels clos, avec des algorithmes basés sur l'élimination des quantificateurs, qui produisent au vu d'un énoncé un énoncé équivalent sans quantificateurs $\forall$ou $\exists$; les énoncés sans quantificateurs de la théorie sont trivialement décidables ; la complexité des algorithmes connus est élevée et ne permet que la décision d'énoncés très simples ;
l'arithmétique de Presburger (arithmétique entière sans multiplication, ou, ce qui revient au même, avec multiplication restreinte au cas où l'un des opérandes est une constante)[1]

Les deux notions de décidabilité (logique et algorithmique) interprètent chacune la notion intuitive de décision dans des sens clairement différents. Elles sont cependant liées.

En effet, on considère en mathématiques qu'une démonstration, si elle peut être difficile à trouver, doit être « facile » à vérifier, en un sens très informel (et discutable — mais ce n'est pas l'objet de cet article). Quand on formalise, on traduit ceci en demandant que le problème de reconnaître si un assemblage de phrases est une démonstration formelle, est décidable. Pour que ceci soit exact, il faut supposer que l'ensemble des axiomes de la théorie est décidable, ce qui, on l'a déjà mentionné, est très naturel.

Sous cette hypothèse, l'ensemble des théorèmes d'une théorie devient récursivement énumérable ; une telle théorie, si elle est complète, est alors décidable (voir article théorie axiomatique pour des justifications et détails supplémentaires). Par contre une théorie décidable, n'est pas forcément complète. Ainsi la théorie des corps algébriquement clos n'est pas complète, puisque la caractéristique n'est pas précisée[2], elle est pourtant décidable. La

théorie des corps algébriquement clos d'une caractéristique donnée est elle complète et décidable.

VI.2.5- Théorème de Rice

En théorie de la calculabilité, le **théorème de Rice** dit que toute propriété non triviale (c.-à-d. qui n'est pas toujours vraie ou toujours fausse) sur la sémantique dénotationnelle d'un langage de programmation Turing-complet est **indécidable**. Il s'agit d'une généralisation du problème de l'arrêt.

VI.2.5.1- Théorème

Théorème (Rice) : *Toute propriété non triviale des langages récursivement énumérables est indécidable.*

Preuve
Nous entendons par propriétés triviales les propriétés qui sont vraies de tous les langages ainsi que celles qui ne sont vraies d'aucun. En fin de compte, une propriété non triviale des langages r.e. est une propriété P telle que P soit vraie d'au moins un langage r.e. mais pas de tous.

C'est toujours la technique de réduction que nous employons. Soit P une propriété non triviale. Alors il existe au moins un langage r.e. possédant la propriété P, et tous ne la possédent pas. On peut supposer que le langage ∅ ne la possède pas. En effet: si c'est le cas... tant mieux (!), si ce n'est pas le cas, alors ∅ ne possède pas la propriété ¬P, qui elle aussi est non triviale. Et nous pouvons raisonner avec ¬P à la place de P. Prouver l'indécidabilité de P est en effet équivalent à prouver l'indécidabilité de ¬P. D'autre part, il existe au moins une machine de Turing TP qui accepte un langage ayant la propriété P, que nous noterons LP. Admettons donc qu'il existe une machine T telle que :

T[Tn] = 1 si le langage accepté par Tn a la propriété P
= 0 sinon

Nous allons démontrer que dans ce cas, LU serait récursif. En effet considérons un mot τ^w qui représente la concaténation du code d'une machine de Turing et de celui d'une entrée. Considérons la machine Taux qui construit à partir de τ^w une machine qui simule Tτ sur w mais ne tient aucun compte de son propre mot d'entrée et est faite de sorte que :
Taux[τ^w][w'] = si Tτ accepte w, alors simule TP sur w' (et puisqu'elle simule Tτ sur w, si Tτ n'accepte pas w, elle n'accepte aucun mot)
alors: Taux[t^w] accepte LP ou ∅ selon que Tτ accepte w ou ne l'accepte pas
mais nous avons :

T[Taux[τ^w]] = 1 si le langage accepté par Taux[τ^w] a la propriété P
T[Tn] = 0 sinon

donc il vient :

T[Taux[τ^w]] = 1 si Taux[τ^w] accepte LP, donc si Tτ accepte w

= 0 si Taux[τ^w] accepte ∅, donc si Tτ n'accepte pas w

Ce qui donne alors un algorithme pour décider de l'appartenance à LU.

———

Tout d'abord, rappelons qu'il n'existe pas de programme, noté *halt*, permettant de savoir si un programme, noté *prog*, s'arrête ou non avec une chaîne de bits *ch* en entrée.

On peut obtenir un résultat plus général.

Précisons quelques notations:
- *prog(ch)* est la chaîne obtenue en sortie si on ne boucle pas en lançant *prog* avec *ch* en entrée,
- on note *ch_prog* la chaîne codant le programme *prog*, et
- *ch1,ch2* désigne la chaîne obtenue en concaténant les deux chaînes *ch1* et *ch2*.

Supposons par exemple, qu'il existe un programme *square* tel que
- *square(ch_prog,ch)* retourne 1 si on ne boucle pas et que
- *prog(ch)* code le carré de l'entier codé par *ch*, et 0 sinon.

On peut alors construire le programme *trouble* suivant, dans lequel *prog* est un programme quelconque et *ch2* une chaîne quelconque:

trouble(ch1)

1. **faire** *prog(ch2)*

2. **retourne** le carré de l'entier codé par *ch1*

Nous obtenons que *trouble(ch1)* existe et code le carré de l'entier codé par *ch1* si et seulement si *prog* s'arrête avec *ch2* en entrée. Ainsi, si *square* existe, alors *halt* existe; or on sait que *halt* n'existe pas.

Dans notre exemple au-dessus on peut remplacer *square* par n'importe quelle fonction, ce qui nous permet de prouver le théorème de Rice: « Pour toute fonction *f* et tout programme *prog*, il n'existe pas de programme pour vérifier que *prog* est une implantation de *f*. »

VI.2.5.2- Impact pratique

La formulation pratique de ce théorème est qu'il n'existe, pour aucune propriété intéressante (non triviale) portant sur le résultat final d'un programme, un algorithme permettant, au vu du code source d'un programme, de décider s'il satisfait cette propriété.

De telles propriétés indécidables incluent:

« le programme ne termine pas par une erreur d'exécution »
« le programme calcule un résultat correct par rapport à la spécification »

En d'autres termes, aucune méthode d'analyse statique de programmes ne peut donner des résultats sûrs et dans un temps fini pour *tous les programmes quels qu'ils soient*, et ce quelle

que soit la propriété à tester. La restriction ne s'applique en revanche pas forcément aux seuls programmes de taille donnée.

Exemple 1 :
Par exemple, un ramasse-miettes (logiciel qui libère la mémoire inutilisée) optimal et sûr est impossible;
En effet, savoir si l'on doit libérer ou non la mémoire dépend du comportement futur du système, qu'il est impossible de prévoir. Ceci explique que toutes les techniques de ramasse-miettes soient conservatrices, et puissent garder en mémoire des blocs inutiles, permettant ainsi des fuites de mémoire.

Exemple 2 :
Ainsi, dans le programme suivant en langage C (supposons que `p` n'intervienne pas ailleurs), déterminer si l'on doit libérer le bloc mémoire sitôt après son allocation est équivalent à décider le problème de l'arrêt sur `f()`.

```
char *p = malloc(1000);
f();
*p = 1;
```

VI.3- Complexité

La théorie de la **complexité algorithmique** s'intéresse à l'estimation de l'efficacité des algorithmes. Elle s'attache à la question :

entre différents algorithmes réalisant une même tâche, quel est le plus rapide et dans quelles conditions ?

Dans les années 1960 et au début des années 1970, alors qu'on en était à découvrir des algorithmes fondamentaux (tris tels que *quicksort*, arbres couvrants tels que les algorithmes de Kruskal ou de Prim), on ne mesurait pas leur efficacité. On se contentait de dire :

« cet algorithme (de tri) se déroule en 6 secondes avec un tableau de 50 000 entiers choisis au hasard en entrée, sur un ordinateur IBM 360/91. Le langage de programmation PL/I a été utilisé avec les optimisations standards ».

Une telle démarche rendait difficile la comparaison des algorithmes entre eux. La mesure publiée était dépendante du processeur utilisé, des temps d'accès à la mémoire vive et de masse, du langage de programmation et du compilateur utilisé, etc.

Une approche indépendante des facteurs matériels était nécessaire pour évaluer l'efficacité des algorithmes. Donald Knuth fut un des premiers à l'appliquer systématiquement dès les premiers volumes de sa série *The Art of Computer Programming*. Il complétait cette analyse de considérations propres à la théorie de l'information : celle-ci par exemple, combinée à la formule de Stirling, montre qu'il ne sera pas possible d'effectuer un tri *général* (c'est-à-dire uniquement par comparaisons) de N élements en un temps croissant moins rapidement avec N que *N ln N* sur une machine *algorithmique*.

VI.3.1- Principe de calcul de la complexité algorithmique

Pour qu'une analyse ne dépende pas de la vitesse d'exécution de la machine ni de la qualité du code produit par le compilateur, il faut utiliser comme unité de comparaison des « opérations élémentaires » en fonction de la taille des données en entrée.

Voici quelques exemples d'opérations élémentaires :

accès à une cellule mémoire
comparaison de valeurs
opérations arithmétiques (sur valeurs à codage de taille fixe)
opérations sur des pointeurs.

Il faut souvent préciser quelles sont les opérations élémentaires pertinentes pour le problème étudié : si les nombres manipulés restent de taille raisonnable, on considérera que l'addition de deux entiers prend un temps constant, quels que soient les entiers considérés (ils seront en effet codés sur 32 bits). En revanche, lorsque l'on étudie des problèmes de calcul formel où la taille des nombres manipulés n'est pas bornée, le temps de calcul du produit de deux nombres dépendra de la taille de ces deux nombres.

On définit alors ***la taille de la donnée*** sur laquelle s'applique chaque problème par un entier lié au nombre d'éléments de la donnée. Par exemple, le nombre d'éléments dans un algorithme de tri, le nombre de sommets et d'arcs dans un graphe.

On évalue le ***nombre d'opérations élémentaires*** en fonction de la taille de la donnée : si 'n' est la taille, on calcule une fonction $t(n)$.

Les critères d'analyse : le nombre d'opérations élémentaires peut varier substantiellement pour deux données de même taille. On retiendra deux critères :

analyse au sens du plus mauvais cas : $t(n)$ est le temps d'exécution du plus mauvais cas et le maximum sur toutes les données de taille n. Par exemple, le tri par insertion simple avec des entiers présents en ordre décroissants.
analyse au sens de la moyenne : comme le « plus mauvais cas » peut en pratique n'apparaître que très rarement, on étudie tm(n), l'espérance sur l'ensemble des temps d'exécution, où chaque entrée a une certaine probabilité de se présenter. L'analyse mathématique de la complexité moyenne est souvent délicate. De plus, la signification de la distribution des probabilités par rapport à l'exécution réelle (sur un problème réel) est à considérer.

La **théorie de la complexité** s'intéresse à l'étude formelle de la *difficulté* des problèmes en informatique. Elle se distingue de la théorie de la calculabilité qui s'attache à savoir si un problème peut être résolu par un ordinateur. La théorie de la complexité se concentre donc sur les problèmes qui peuvent effectivement être résolus, la question étant de savoir s'ils peuvent être résolus efficacement ou pas en se basant sur une estimation (théorique) ***des temps de calcul*** et des ***besoins en mémoire*** informatique.

VI.3.2- Machines déterministe et non déterministe

Une machine déterministe est le modèle formel d'une machine telle que nous les connaissons (nos ordinateurs sont des machines déterministes). Le modèle le plus utilisé en informatique théorique est la machine de Turing.

Les machines déterministes font toujours un seul calcul à la fois. Ce calcul est constitué d'étapes élémentaires; à chacune de ces étapes, pour un état donné de la mémoire de la machine, l'action élémentaire effectuée sera toujours la même. Pour la suite, on pourra imaginer sans perte de généralité qu'une machine de Turing déterministe correspond à l'ordinateur favori du lecteur, programmé dans un langage impératif quelconque.

Une *machine de Turing non-déterministe* est une variante purement théorique des machines de Turing: on ne peut pas construire de telle machine. À chaque étape de son calcul, cette machine peut effectuer un **choix non-déterministe**: elle a le choix entre plusieurs actions, et elle en effectue une. Si l'un des choix l'amène à accepter l'entrée, on considère qu'elle a fait ce choix-là. En quelque sorte, elle devine toujours juste. Une autre manière de voir leur fonctionnement est de considérer qu'à chaque choix non-déterministe, elles se dédoublent, les clones poursuivent le calcul en parallèle suivant les branches du choix. Si l'un des clones accepte l'entrée, on dit que la machine accepte l'entrée.

Il semblerait donc naturel de penser que les machines de Turing non-déterministes sont beaucoup plus puissantes que les machines de Turing déterministes, autrement dit qu'elles peuvent résoudre en un temps raisonnable des problèmes que les machines ordinaires ne savent pas résoudre à moins de prendre des années. L'Ordinateur à ADN (les années 90) est

un cas particulier de machines non-déterministes, puisqu'il est capable de traiter un calcul en temps constant quelle que soit la taille des données. Notons toutefois que la difficulté est ici transposée en termes de réalisabilité de la machine, qui nécessite des quantités exponentielles d'ADN en fonction de la taille des données.

VI.3.3- Complexités d'un algorithme en temps et en espace

1/ Complexité en temps

La complexité d'un algorithme se mesure essentiellement en calculant le *nombre d'opérations élémentaires* pour traiter une donnée de t*aille n*. Les opérations élémentaires considérées sont

le nombre de comparaisons (algorithmes de recherche)
le nombre d'affectations (algorithmes de tris)
Le nombre d'opérations (+,*) réalisées par l'algorithme (calculs sur les matrices ou les polynômes).

On désigne par n la **taille** de la donnée.
Pour les machines déterministes, on définit la classe **TIME(t(n))** des problèmes qui peuvent être résolus en temps **t(n)**. C'est-à-dire pour lesquels il existe au moins un algorithme sur machine déterministe résolvant le problème en temps t(n) (le temps étant le nombre de transitions sur machine de Turing).

TIME(t(n)) = { L | L peut être décidé en temps t(n) par une machine déterministe }

Pour les machines non déterministes, on définit la classe **NTIME(t(n))** des problèmes qui peuvent être résolus en temps **t(n)**.

NTIME(t(n)) = { L | L peut être décidé en temps t(n) par une machine non-déterministe }

Le coût d'un algorithme A pour une donnée d est le nombre d'opérations élémentaires n nécessaires au traitement de la donnée d et est noté $COUT_A(d)$

Complexité dans le pire des cas : $Max_A(n) = \max \{COUT_A(d), d \in Dn\}$ (Dn est ensemble de données de taille n)
Complexité dans le meilleur des cas : $Min_A(n) = \min \{COUT_A(d), d \in Dn\}$
Complexité en moyenne $Moy_A(n) = \sum_{d \in Dn} p(d)COUT_A(d)$

où p(d) est la probabilité d'avoir en entrée la donnée d parmi toutes les données de taille n. Si toutes les données sont équiprobables, alors on a,

$Moy_A(n) = (1/|Dn|) \sum_{d \in Dn} COUT_A(d)$

2/ Complexité en espace mémoire

Il est quelquefois nécessaire d'étudier la complexité en mémoire lorsque l'algorithme requiert de la mémoire supplémentaire (tableau auxiliaire de même taille que le tableau donné en entrée par exemple).

La complexité en espace évalue l'espace mémoire utilisé en fonction de la taille des données ; elle est définie de manière analogue :

SPACE(s(n)) = { L | L peut être décidé par une machine déterministe en utilisant au plus s(n) cellules de mémoire}
NSPACE(s(n)) = { L | L peut être décidé par une machine non-déterministe en utilisant au plus s(n) cellules de mémoire}

Remarque 1 : problème de codage
Le codage influence la complexité des problèmes. Il est bon de se rappeler que les données sur lesquelles travaillent les algorithmes sont nécessairement stockées en mémoire (on parle ici de la mémoire de l'ordinateur, mais aussi de la bande de la machine de Turing par exemple). Si le codage d'une donnée est exponentiel par rapport à la taille de la donnée initiale, l'ensemble des complexités des algorithmes sera sans doute caché par la complexité du codage : il faut par exemple s'interdire de coder le résultat dans l'entrée...
On ne s'intéressera ici qu'aux codages *raisonnables*.

Remarque 2 : problème de décision
Chaque problème informatique peut se réduire à un problème de décision, c'est à dire un problème formulé comme une question dont la réponse est *Oui* ou *Non*, plutôt que par exemple *la taille du plus court chemin est 42*. Un problème qui n'est pas formulé de cette manière peut être très simplement transformé en un problème de décision *équivalent*.
par exemple, le *problème du voyageur de commerce*, qui cherche, dans un graphe, à trouver la taille du cycle le plus court passant une fois par chaque sommet, peut s'énoncer en un problème de décision ainsi :
« *Existe-t-il un cycle passant une et une seule fois par chaque sommet tel que la somme des coûts des arcs utilisés soit inférieure à B, avec* $B \in \mathbb{N}$ »
Ce problème est équivalent au problème du voyageur de commerce au sens où si l'on sait résoudre efficacement l'un, on sait aussi résoudre efficacement l'autre

Exemple de calcul de complexité d'un algorithme : apparition d'un nombre dans un tableau.

Problème : Soit T un tableau de taille N contenant des nombres entiers de 1 à k. Soit a un entier entre 1 et k. La fonction suivante retourne la position du premier a rencontré s'il existe, et 0 sinon.

```
Trouve:=proc(T,n,a)
        local i;
        for i from 1 to n do
                if T[i]=a then RETURN(i) fi;
        od;
        RETURN(0);
end;
```

Cas le pire : N (le tableau ne contient pas a)
Cas le meilleur : 1 (le premier élément du tableau est a)
Complexité moyenne : Si les nombres entiers de 1 à k apparaissent de manière équiprobable, on peut montrer que le cout moyen de l'algorithme est $k(1 - (1 - 1/k)^N)$

De fait les cas où l'on peut explicitement calculer la complexité en moyenne sont rares. Cette étude est un domaine à part entière de l'algorithmique que nous aborderons assez peu ici.

Toutefois, il est indispensable, après avoir écrit un algorithme, de calculer sa complexité dans le pire des cas et dans le meilleur des cas.

VI.3.4- Analyse asymptotique

On étudie systématiquement la complexité *asymptotique*, notée grâce aux *notations de Landau*. La notion grand O, aussi appelée **symbole de Landau**, est un symbole utilisé en théorie de la complexité, en informatique, et en mathématiques pour décrire le comportement asymptotique des fonctions. Fondamentalement, elle indique avec quelle rapidité une fonction « augmente » ou « diminue ».

Le *symbole de Landau* porte le nom du mathématicien allemand spécialisé en théorie des nombres Edmund Landau qui utilisa cette notation introduite primitivement par Bachmann. La lettre O est utilisée parce que la course de la « croissance » d'une fonction est aussi appelée l'*ordre*.

Informellement, cette notion vient de deux idées simples

- idée 1 : évaluer l'algorithme sur des données de grande taille. Par exemple, lorsque n est '***grand***', $3n^3 + 2n^2$ est essentiellement $3n^3$.

- idée 2 : on élimine les constantes multiplicatrices, car deux ordinateurs de puissances différentes diffèrent en temps d'exécution par une constante multiplicatrice. De $3 * n^3$, on ne retient que n^3

L'algorithme est dit en $O(n^3)$.

L'idée de base est donc qu'un algorithme en $O(n^a)$ est « meilleur » qu'un algorithme en $O(n^b)$ si $a < b$.

1/ La notation grand O (avec la lettre majuscule O, et non pas zéro)

Défintion VI.3.4.1 : Grand O

Supposons que f et g soient deux fonctions définies sur une partie de l'ensemble des nombres réels. Nous écrivons :

$f(x) = O(g(x))$ (ou $f(x) = O(g(x))$ quand $x \to \infty$ pour être plus précis)

si et seulement s'il existe des constantes N et C telles que

pour tout $x>N$, $|f(x)| \leq C\,|g(x)|$.

Défintion VI.3.4.2 : Grand O généralisé

Plus généralement, si a est un nombre réel, nous écrivons

$f(x) = O(g(x))$ quand $x \to a$

si et seulement s'il existe des constantes $d > 0$ et C telles que

pour tout x tel que $|x-a| < d$, $|f(x)| \leq C\,|g(x)|$.

Remarque

- Intuitivement, cela signifie que f ne croît pas plus vite que g.

- La relation O n'est pas symétrique
- La première définition est la seule utilisée en informatique (où typiquement seules les fonctions positives à variable entière n sont considérées; les valeurs absolues pouvent être ignorées), tandis que les deux définitions sont utilisées en mathématiques.

2/ Autre notations asymptotique: Ω, Θ, o

Défintion VI.3.4.3 : notion Ω

Supposons que f et g soient deux fonctions définies sur une partie de l'ensemble des nombres réels. Nous écrivons :
$f(x) = \Omega(g(x))$ (ou $f(x) = \Omega(g(x))$ quand $x \to \infty$)
si et seulement s'il existe des constantes N et C telles que

pour tout $x>N$, $C\,|g(x)| \leq |f(x)|$.

Remarque : Ω n'est pas symétrique

Défintion VI.3.4.4 : notion Θ

Supposons que f et g soient deux fonctions définies sur une partie de l'ensemble des nombres réels. Nous écrivons :
$f(x) = \Theta(g(x))$ (ou $f(x) = \Theta(g(x))$ quand $x \to \infty$)
si et seulement s'il existe des constantes N et *C1 et C2* telles que

pour tout $x>N$, $C1\,|g(x)| \leq |f(x)| \leq C2\,|g(x)|$

Remarque : Θ est une relation symétrique

Défintion VI.3.4.5 : notion petit o

Dans le même esprit de manipulation des ordres de grandeur, la notation $f(x) = o(g(x))$ (cette fois avec un ***petit o***) signifie que la fonction f est négligeable devant la fonction g, quand x tend vers une valeur particulière. Formellement, pour $a \in \mathbb{R} \cup \{+\infty; -\infty\}$,et pour f et g deux fonctions de la variable réelle x, avec g qui ne s'annule pas sur un voisinage de a, on dit que $f(x) = o(g(x))$ quand $x \to a$ si et seulement si $f(x) / g(x) \to 0$ quand $x \to a$.

Remarque :
- Grand-O est la notation asymptotique la plus utilisé
- Après grand-O, les notations Θ et Ω sont les plus utilisées en informatique ;
- Le petit-o est courant en mathématique mais plus rare en informatique.

3/ Applications analytique

Voici une liste de catégories de fonctions qui sont couramment rencontrées dans les analyses d'algorithmes. Les fonctions de croissance les plus lentes sont listées en premier. c est une constante arbitraire.

notation	complexité
O(1)	constante
$O(\log(n))$	logarithmique
$O((\log(n))^c)$	polylogarithmique
$O(n)$	linéaire

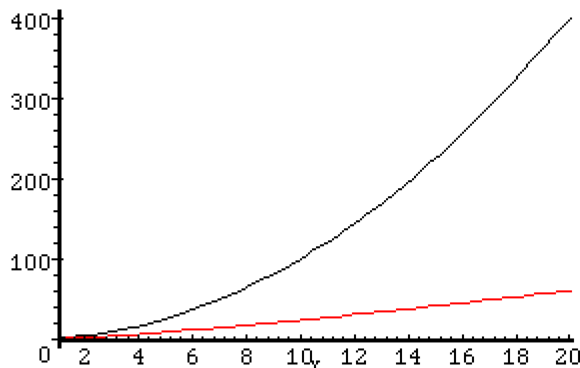
O(n log(n))	parfois appelée « linéarithmique »
O(n^2)	quadratique
O(n^c)	polynomiale, parfois « géometrique »
O(c^n)	exponentielle
O($n!$)	factorielle

Remarque :
- Notons que O(n^c) et O(c^n) sont très différents. Le dernier exprime une croissance bien plus rapide, et ce pour n'importe quelle constante c>1. Une fonction qui croît plus rapidement que n'importe quel polynôme est appelée *super-polynomiale*. Une fonction qui croît plus lentement que toute exponentielle est appelée *sous-exponentielle*. Il existe des fonctions à la fois super-polynômiales et sous-exponentielle comme par exemple la fonction $n^{\log(n)}$. Certains algorithmes ont un temps de calcul de ce type, comme celui de la factorisation d'un nombre entier.
- Remarquons aussi, que O(log n) est exactement identique à O(log(n^c)). Les logarithmes diffèrent seulement d'un facteur constant, et que la notation grand « ignore » les constantes. De manière analogue, les logarithmes dans des bases constantes différentes sont équivalents.
- La liste précédente est utile à cause de la propriété suivante : si une fonction f est une somme de fonctions, et si une des fonctions de la somme grimpe plus vite que les autres, alors celle qui croît le plus vite détermine l'ordre de $f(n)$.

ex.: si $f(n) = 10 \log(n) + 5 (\log(n))^3 + 7 n + 3 n^2 + 6 n^3$,

alors $f(n) = O(n^3)$.

Exemple
- Un polynôme est de l'ordre de son degré. On distingues les fonctions linéaires (en O(n)), les fonctions quadratiques (en O(n^2)) et les fonctions cubiques (en O(n^3)).
- Les fonctions d'orde exponentiell sont les fonctions en O(a^n) ou a >1.
- Les fonctions d'ordre logarithmique sont les fonctions en O(log(n)) (Remarquons que peu importe la base du logarithme).
- Une classe intéressante d'algorithme est en n log(n). Comparaison de nlog(n) et de n^2.



VI.3.5- Classes de complexité

La théorie de la complexité repose sur la définition de classes de complexité qui permettent de classer les problèmes en fonction de la complexité des algorithmes qui existent pour les résoudre.

1/ Classe L (LOGSPACE)
Un problème de décision qui peut être résolu par un algorithme déterministe en espace *logarithmique* par rapport à la taille de l'instance est dans L.

2/ Classe NL
Cette classe s'apparente à la précédente mais pour un algorithme non-déterministe.

3/ Classe P
Un problème de décision est dans *P* s'il peut être décidé par un algorithme déterministe en un temps *polynomial* par rapport à la taille de l'instance. On qualifie alors le problème de polynomial.

Exemple :
Prenons par exemple le problème de la connexité dans un graphe. Étant donné un graphe à *s* sommets, il s'agit de savoir si toutes les paires de sommets sont reliées par un chemin. Pour le résoudre, on dispose de l'algorithme de parcours en profondeur qui va construire un arbre couvrant du graphe à partir d'un sommet. Si cet arbre contient tous les sommets du graphe, alors le graphe est connexe. Le temps nécessaire pour construire cet arbre est au plus s^2, donc le problème est bien dans la classe P.

Les problèmes dans P correspondent en fait à tous les problèmes facilement solubles.

4/ Classe NP
Un problème NP est *N*on-déterministe *P*olynomial (et non pas *Non polynomial*, erreur très courante).

La classe NP réunit les problèmes de décision pour lesquels la réponse *oui* peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance.

De façon équivalente, c'est la classe des problèmes qui admettent un algorithme dans *P* qui, étant donné une solution du problème *NP* (un *certificat*), est capable de répondre *oui* ou *non*.

Intuitivement, les problèmes dans *NP* sont tous les problèmes qui peuvent être résolus en énumérant l'ensemble des solutions possibles et en les testant avec un algorithme polynomial.

Exemple
Par exemple, la recherche de cycle hamiltonien dans un graphe peut se faire avec deux algorithmes :

- le premier génère l'ensemble des cycles (ce qui est exponentiel) ;
- le second teste les solutions (en temps polynomial).

5/ Classe Co-NP (Complémentaire de NP)
C'est le nom parfois donné pour l'équivalent de la classe *NP*, mais avec la réponse *non*.

6/ Classe PSPACE
Elle regroupe les problèmes décidables par un algorithme déterministe en espace polynomial par rapport à la taille de son instance.

7/ Classe NSPACE ou NPSPACE
Elle réunit les problèmes décidables par un algorithme non-déterministe en espace polynomial par rapport à la taille de son instance.

8/ Classe EXPTIME
Elle rassemble les problèmes décidables par un algorithme déterministe en temps exponentiel par rapport à la taille de son instance.

9/ Classe NEXPTIME
Elle rassemble les problèmes décidables par un algorithme non déterministe en temps exponentiel par rapport à la taille de son instance.

VI.3.6- Propriété des Classes de complexité

Proposition VI.3.6.1- Complexité en temps et en espace séparément

- P ⊆ NP et symétriquement P ⊂ co-NP
- P ⊆ NP ⊆ EXPTIME
- LOGSPACE ⊆ PSPACE = NPSPACE

Proposition VI.3.6.2- Liaison entre temps et espace
Si *SPACE(s(n))* ≥ n alors *SPACE(s(n))* ≤ *TIME(s(n))*.
(Si M fonctionne en temps t(n) alors M fonctionne en espace au plus t(n))

Proposition VI.3.6.3- Complexité en temps et espace

- LOGSPACE ⊆NLOGSPACE ⊆ P ⊆ NP ⊂ PSPACE = NPSPACE et symétriquement

Co-NP ⊂ PSPACE

- P ⊆ NP⊂ PSPACE = NPSPACE ⊆ EXPTIME

VI.3.7- Problème C-Complet et réduction

Définition VI.3.7.1- C-Difficile
Un problème est **C-difficile** si ce problème est au moins aussi dur que tous les problèmes dans **C**.

Définition VI.3.7.2- C-Complet
Soit **C** une classe de complexité (comme **P**, **NP**, etc.). On dit qu'un problème est **C-complet** si

- il est dans **C**, et

- il est **C-difficile** (on utilise parfois la traduction incorrecte **C-dur**).

Définition VI.3.7.3- Réduction
Formellement on définit une notion de réduction : Soient *Π* et *Π'* deux problèmes ; Une réduction de *Π'* à *Π* est un algorithme transformant toute instance de *Π'* en une instance de *Π*. Ainsi, si l'on a un algorithme pour résoudre *Π*, on sait aussi résoudre *Π'*. *Π* est donc au moins aussi difficile à résoudre que *Π'*.

Remarque *:*

- *Π* est alors **C-difficile** si pour tout problème *Π'* de **C**, *Π'* se réduit à *Π*.
- Quand on parle de problèmes NP-difficiles on s'autorise uniquement des réductions dans P, c'est-à-dire que l'algorithme qui calcule le passage d'une instance de *Π'* à une instance de *Π* est polynomial (Réduction polynomiale).
- Quand on parle de problèmes P-difficiles on s'autorise uniquement des réductions dans LOGSPACE.

Principe de transformation de problème :
La réduction la plus simple (ce n'est d'ailleurs pas vraiment une réduction) consiste simplement à transformer le problème à classer en un problème déjà classé.

Par exemple, démontrons ici que le problème de la recherche de cycle hamiltonien dans un graphe orienté est NP-Complet.

- Le problème est dans *NP* : On peut trouver de façon évidente un algorithme pour le résoudre avec une machine non-déterministe, par exemple en énonçant tous les cycles puis en sélectionnant le plus court.
- Nous disposons du problème de la recherche de cycle hamiltonien pour les graphes non-orientés. Un graphe non-orienté peut se transformer en un graphe orienté en « doublant » chaque arête de manière à obtenir, pour chaque paire de nœuds adjacents, des chemins dans les deux sens. Il est donc possible de ramener le problème connu, NP-difficile, au problème que nous voulons classer. **Le nouveau problème est donc NP-difficile**.
- Le problème étant dans *NP* et *NP-difficile*, il est **NP-complet**.

VI.3.7- Problème ouvert P = NP ?

La recherche travaille activement à déterminer si **NP** $\subset$ **P** (concluant à **P** = **NP**) ou si, au contraire **P** $\neq$ **NP** ?

On a trivialement $P \subseteq NP$ car un algorithme déterministe est un algorithme non déterministe particulier. En revanche la réciproque : $NP \subseteq P$, que l'on résume généralement à P = NP du fait de la trivialité de l'autre inclusion, est l'un des problèmes ouverts les plus fondamentaux et intéressants en informatique théorique. Cette question a été posée en 1970 pour la première fois ; celui qui arrivera à décider si *P* et *NP* sont différents ou égaux recevra le prix Clay (plus de 1 000 000 $).

Le **problème P = NP** revient à savoir si on peut résoudre un problème NP-Complet avec un algorithme polynomial.

Les problèmes étant tous classés les uns à partir des autres — un problème est NP-Complet si l'on peut réduire un problème NP-Complet connu en ce problème —, faire tomber **un seul** de ces problèmes dans la classe *P* fait tomber **l'ensemble de la classe NP**, ces problèmes étant massivement utilisés, en raison de leur difficulté, par l'industrie, notamment en cryptographie — cf. la factorisation en nombres premiers). Ceci fait qu'on conjecture cependant que les problèmes NP-complets ne sont pas solubles en un temps polynomial. À partir de là plusieurs approches sont tentées :

- Des algorithmes d'approximation permettent de trouver des solutions approchées de l'optimum en un temps raisonnable pour un certain nombre de programmes. Dans le cas d'un problème d'optimisation on trouve généralement une réponse correcte, sans savoir s'il s'agit de la meilleure solution ;
- Des algorithmes stochastiques : en utilisant des nombres aléatoires on peut «forcer» un algorithme à ne pas utiliser les cas les moins favorables, l'utilisateur devant préciser une probabilité maximale admise que l'algorithme fournisse un résultat erroné. Citons notamment comme application des algorithmes de test de primalité en temps polynomial en la taille du nombre à tester. A noter qu'un algorithme polynomial non stochastique a été proposé pour ce problème en août 2002 par Agrawal, Kayal et Saxena ;
- Des heuristiques permettent d'obtenir des solutions généralement bonnes mais non exactes en un temps de calcul modéré ;
- Des algorithmes par séparation et évaluation permettent de trouver la ou les solutions exactes. Le temps de calcul n'est bien sûr pas borné polynomialement mais, pour certaines classes de problèmes, il peut rester modéré pour des instances relativement grandes ;
- On peut restreindre la classe des problèmes d'entrée à une sous-classe suffisante, mais plus facile à résoudre.

Si ces approches échouent, le problème est non soluble en pratique dans l'état actuel des connaissances.

Pour le cas de L et NL, on ne sait pas non plus si L = NL. Mais cette question est moins primordiale car $\mathrm{L} \subset \mathrm{NL} \subset \mathrm{P} \subset \mathrm{NP}$. De fait, les problèmes dans L et dans NL sont solubles efficacement.

Inversement on sait que PSPACE = NPSPACE. Par contre $\mathrm{NP} \subset \mathrm{PSPACE}$. Donc, avant de résoudre NP = PSPACE, il faut résoudre P = NP.

Pour résumer, on a $\mathrm{L} \subset \mathrm{NL} \subset \mathrm{P} \subset \mathrm{NP} \subset \mathrm{PSPACE} = \mathrm{NPSPACE}$. On sait de plus que NL est strictement inclus dans PSPACE. Donc deux classes au moins entre NL et PSPACE ne sont pas égales.