Spécialité NSI en terminale
Calculabilité et décidabilité

Ce sont des notions complexes mais capitales en informatique théorique. Elle sont liées à des théories mathématiques et logiques difficiles. Cette section en présente un aperçu.

Calculabilité

La notion de calculabilité date de 1936.

Si on appelle « programme » la description d'une suite d'opérations à effectuer mécaniquement, alors un programme peut être représenté par une machine de Turing. Un nombre est *calculable* s'il peut être obtenu par une machine de Turing. En fait, Turing utilise le mot *computable* pour ce type de nombre, et le différencie du mot calculable. Cette notion est précisée par la *thèse de Church*. Un nombre calculable au sens intuitif, (avec des objets, des jetons, des cailloux, ou mentalement en faisant abstraction des objets), correspond à un nombre calculable par une machine de Turing.

Le mot calculer vient du latin calculus qui signifie caillou. Que peut-on calculer ?

Une première réponse est : ce que l'on peut calculer avec des objets. Mais si cela se résume à des additions et des soustractions, cela ne nous mène pas très loin.

Une réponse plus réfléchie est de dire qu'une multiplication peut se décomposer en une série d'additions, puis qu'une opération complexe peut se décomposer en une série d'opérations moins complexes et ainsi de suite jusqu'à aboutir à des additions. On peut donc calculer beaucoup de choses, et même des nombres comme π.

Mais alors tous les nombres sont-ils calculables ? La réponse à cette question est non.

Nombres calculables

Un nombre est donc calculable s'il existe une machine de Turing ou bien un programme qui nous permet d'obtenir un par un la suite des chiffres de son écriture décimale qui peut être infinie. Une machine peut nous en donner une approximation à n'importe quelle précision souhaitée.

Les nombres calculables sont par exemple les nombres rationnels, les nombres comme $\sqrt{2}$, $\sqrt{5}$ et plus généralement les nombres qui sont solutions d'une équation $P(x) = 0$ où P est un polynôme à coefficients rationnels. Le nombre π est aussi un nombre calculable.

Cependant il existe beaucoup plus de nombres non calculables que de nombres calculables. La difficulté est d'en exhiber. Grégory Chaitin, qui a travaillé sur la première théorie algorithmique de l'information, comme les mathématiciens Solomonov et Kolmogorov dans les années 1960, a proposé comme exemples les constantes nommées Oméga. Ce sont des nombres aléatoires tels qu'un algorithme ne peut en donner qu'un nombre limité de décimales. Et il en existe une infinité.

Fonctions calculables

Une fonction f est calculable, si on peut obtenir $f(u)$ en un nombre fini d'étapes pour tout u donné, donc avec une succession d'opérations simples. Autrement dit, la fonction f est calculable si on peut calculer $f(u)$ avec une machine de Turing qui termine. Lorsque la machine s'arrête, $f(u)$ est écrit sur le ruban.

Comme pour le cas des nombres, il est démontré qu'il existe des fonctions non calculables. La preuve repose sur le constat que l'ensemble des algorithmes est dénombrable alors que l'ensemble des fonctions ne l'est pas. Précisons la signification du mot dénombrable : nous disons qu'un ensemble est dénombrable

si on peut numéroter ces éléments. La notion de dénombrabilité a été étudiée en particulier par le mathématicien Georg Cantor.

Remarque

Tous les objets utilisés dans les programmes sont codés par des nombres entiers naturels stockés dans une mémoire. Avec un ensemble d'opérations de base, on peut imaginer un modèle mathématique d'un ordinateur. Le premier de ces modèles date de 1936, c'est la machine de Turing. Tous les algorithmes concevables peuvent être exécutés par cette machine. Diverses théories ont été développées utilisant les notions de récursivité ou le lambda-calcul par exemple. Les mathématiciens Alonzo Church et Stephen Kleene ont montré que toutes les théories envisagées définissent les mêmes fonctions calculables.

Problèmes

On peut se demander si un problème peut être résolu à l'aide d'un algorithme. Pour répondre à cette question, on ramène le problème à une fonction. La question est alors de savoir si cette fonction est calculable ou pas.

Exemples

— Est-ce qu'il existe un algorithme ou un programme qui permet de savoir si un énoncé mathématique quelconque est un théorème (on peut le prouver), ou pas ? La fonction associée prend la valeur 1 si c'est un théorème et 0 sinon. On démontre que cette fonction n'est pas calculable.

— Est-ce qu'il existe un programme qui peut prendre en argument un programme P quelconque et détermine si l'exécution du programme P termine ou pas ? La fonction associée prend la valeur 1 si l'exécution du programme P termine, et 0 si l'exécution ne termine pas. On démontre aussi que cette fonction n'est pas calculable. Des précisions sont données par la suite.

Décidabilité

Notion

Les deux problèmes donnés en exemple ci-dessus sont des problèmes de décision.

La *décidabilité* est une notion utilisée en logique. Dans le cadre d'une théorie axiomatique, étant donnée une proposition, il s'agit de démontrer qu'elle est vraie ou fausse. Si cette démonstration est possible, on dit que la proposition est décidable. Sinon, on dit que la proposition est indécidable.

En algorithmique, un *problème de décision* est une question à laquelle on répond par oui ou par non. On dit qu'un problème est *décidable* s'il existe un algorithme, c'est-à-dire une méthode qui se termine en un nombre fini d'étapes, qui permet de répondre par oui ou par non à la question posée par le problème. Sinon on dit que le problème est *indécidable*.

Examinons un exemple simple avec le programme qui suit où `n` est un entier naturel.

```
def pair(n):
    while n > 0:
        n = n - 2
    return n == 0
```

Ce programme permet de déterminer pour tout entier naturel `n` s'il est pair ou pas. La fonction renvoie `True` si `n` est pair et `False` si `n` est impair.

Le problème, ou la question posée, est : est-ce qu'il existe un programme qui prend en argument un entier naturel `n` quelconque et détermine si cet entier naturel est pair ou pas ?

C'est un problème de décision et la réponse est oui. Donc ce problème est décidable. La fonction associée prend la valeur 1 ou 0 suivant que le nombre est pair ou non et cette fonction est calculable. Cette fonction s'appelle la fonction caractéristique de l'ensemble des nombres pairs.

La fonction caractéristique d'un ensemble est une fonction K telle, pour tout n, $K(n) = 1$ si n appartient à l'ensemble et $K(n) = 0$ sinon.

Nous pouvons aussi écrire un programme qui détermine pour n'importe quel entier si cet entier est premier ou pas.

On dit que ces deux propriétés, être pair et être premier, sont des propriétés décidables.

Si nous nous plaçons dans le domaine de la calculabilité, nous disons que l'ensemble des nombres pairs et l'ensemble des nombres premiers sont des ensembles décidables. Nous disons aussi que les fonctions caractéristiques de ces ensembles sont des fonctions calculables.

Hilbert a posé en 1928 la question en logique mathématique qu'on appelle le *problème de la décision*, en allemand « Entscheidungsproblem ». Peut-on déterminer par un algorithme si un énoncé quelconque est vrai ou faux, si c'est un théorème ?

Gödel a démontré en 1931 qu'il existe des propriétés mathématiques non décidables dans n'importe quel système définissant l'arithmétique.

Turing a aussi répondu négativement à la question de Hilbert en utilisant l'indécidabilité d'un problème nommé le *problème de l'arrêt*. Le problème de la décision et le problème de l'arrêt sont les deux problèmes évoqués en exemple précédemment.

Dans son papier de 1936, Turing montre que si le problème de la décision est décidable, alors la question de l'arrêt ou non d'une machine de Turing donnée peut être résolue par un algorithme. Autrement dit, la terminaison ou non d'un programme quelconque peut être déterminée par une machine de Turing. Or, il a démontré dans ce même papier qu'il n'existe pas de machine, ou de fonction, qui pour tous les programmes écrirait 1 si le programme donné termine et 0 sinon. Cette fonction n'est pas calculable. Donc le problème de la décision n'est pas décidable.

Indécidabilité du problème de l'arrêt

La question est donc de savoir si on peut écrire un programme dont la fonction est d'analyser un programme quelconque et déterminer si ce programme termine dans tous les cas, quelles que soient les données fournies en entrées. Et ce programme doit permettre de répondre à la question pour tous les programmes en un temps fini.

Rappelons que si par exemple un programme entre dans une boucle sans fin, alors il ne termine pas. On dit que l'algorithme n'est pas valide.

La démonstration de Turing, basée sur les machines et la calculabilité est complexe. Ce qui suit ne constitue pas une preuve mais donne quelques idées de la démarche suivie.

- **Avec des programmes**

Nous supposons que nous disposons d'un programme `T` qui prend en argument un programme `P` quelconque avec des données `d` quelconques. Le programme `P` est représenté par son code `prog` et les données `d` par un code `data`. Le programme `T` renvoie `True` si le programme `P` termine avec les données `d` et `False` sinon.

`T(prog, data)` vaut `True` si `P(d)` termine et `False` sinon.

Considérons alors un programme `D` qui prend en entrée un programme `P` quelconque. Le programme `D` contient le code du programme `T` et produit le code `prog` du programme `P`. Ensuite il se comporte comme le programme `T`. Puis, si le programme `T` renvoie `True` avec le code programme `prog` et le code données `prog`, alors `D` rentre dans une boucle infinie, donc ne termine pas. (Il affiche par exemple `True`, `True`, ...). Et si `T` renvoie `False`, alors le programme `D` affiche faux et termine.

`D(P)` boucle si `T(prog, prog)` vaut `True` et `D(P)` termine si `T(prog, prog)` vaut `False`.

Que se passe-t-il si on donne au programme `D` le programme `D` lui-même ? Soit `T` renvoie `True`, ce qui signifie que `D` termine, mais alors dans ce cas `D` ne termine pas ! Soit `T` renvoie `False`, ce qui signifie que `D` ne termine pas, mais alors dans ce cas `D` termine !

Si `T(D, D)` vaut `True`, ce qui signifie que `D(D)` termine, alors `D(D)` boucle et si `T(D, D)` vaut `False`, ce qui signifie que `D(D)` boucle, alors `D(D)` termine.

Ceci est absurde et on en déduit que le programme `D` ne peut pas exister, donc que le programme `T` ne peut pas exister.

- **Transposition avec des machines de Turing**

Supposons que le problème de l'arrêt soit décidable. Il existe donc une machine M_1 qui, lorsqu'on lui fournit les instructions d'une machine M, disons son numéro n, et une donnée initiale d, s'arrête, par exemple en écrivant 1 si M s'arrête avec la donnée d, et en écrivant 0 dans le cas contraire.

Construisons alors (par la pensée) une machine M_2 fonctionnant comme suit. En présence d'une donnée n, M_2 forme les instructions de la machine M de numéro n, suivi de n.

Nous avons donc le premier n qui identifie la machine et le deuxième n qui identifie les données.

Ensuite, M_2 a la même fonctionnement que M_1, mais si M_1 aurait écrit 1, alors M_2 calcule indéfiniment, et si M_1 aurait écrit 0, alors M_2 s'arrête.

Ainsi, M_2 ne s'arrête pas si et seulement si on lui donne un n qui est l'identifiant d'une machine qui s'arrête pour la donnée n.

Nous arrivons à la contradiction : M_2 est elle-même une machine de Turing, et a donc un identifiant n_2. Donc si on fournit à M_2 la donnée n_2, M_2 devrait calculer sans arrêt si et seulement si n_2 est le numéro d'une machine qui s'arrête pour la donnée n_2. Mais la machine correspondant à l'identifiant n_2 est la machine M_2 ! Donc M_2 ne peut pas exister.

- **Avec des fonctions en Python**

En Python, on ne peut évidemment pas écrire une fonction `termine` qui renvoie `True` si le programme `prog` avec les données `data` s'arrête et renvoie `False` sinon. Nous allons donc le faire en deux étapes.

```
def machine(prog):
    if termine(prog, prog):
        print("termine")
        while True: print("boucle")
    else:
        print("boucle indéfiniment")

def termine(prog, data):
    return prog is data
```

L'expression `termine(machine, machine)` vaut `True`, autrement dit le programme termine et pourtant `machine(machine)` boucle indéfiniment. (Appuyer sur Ctrl+C pour l'arrêter).

Modifions la fonction `termine` :

```
def termine(prog, data):
    return prog is not data
```

L'expression `termine(machine, machine)` vaut `False`, autrement dit le programme boucle indéfiniment, et pourtant `machine(machine)` termine.