Séq. 20 – Calculabilité et décidabilité

Objectifs

1. Comprendre que la calculabilité ne dépend pas du langage de programmation utilisé
2. Montrer, sans formalisme théorique, que le problème de l'arrêt est indécidable

Cette séquence s'appuie sur :

- https://lecluseo.frama.io/leclusemaths/nsi/NSI_T/langages/calculabilite/

1 Programme en tant que donnée

Les théories de la calculabilité et de la complexité sont deux pans de la théorie du calcul : qu'est-ce que veut dire calculer ? un ordinateur peut-il tout calculer ?

C'est en 1936 que **Allan Turing (1912-1954)** a apporté des réponses à ces questions.

Nous allons tout d'abord expliciter un point important qui sera le fondement de la théorie de la calculabilité : un programme est aussi une donnée.

Cela peut paraître étonnant à première vue puisqu'on est habitué à traiter

- les programmes dans des fonctions
- les données dans des variables.

Fonctions et variables sont des objets de nature différente en apparence : Si on se raccroche à ce que l'on connaît en python, une fonction se déclare avec le mot clé `def` et une variable s'initialise avec l'opérateur d'affectation `=`.

Prenons en exemple l'algorithme d'Euclide - un algorithme vieux de plus de 2500 ans permettant de calculer le PGCD de 2 nombres. On peut l'écrire à l'aide d'une fonction Python:

```
def euclide(a,b):
    if a < b:
        a,b=b,a
    while b:
        a,b=b,a%b
    return a
```

Dans ce programme Python, euclide est une fonction et a et b sont des données. Ils ne semblent pas être de nature comparable :

```
>>> euclide(35, 49)
7
```

Et pourtant, à y regarder de plus près, notre algorithme programmé dans la fonction `euclide` n'est rien d'autre qu'une succession de caractères. On peut même pousser la réflexion jusqu'à créer une chaîne de caractère contenant ce programme :

```
mon_programme = "def euclide(a,b):\n\tif a < b: a,b=b,a\n\twhile b: a,b=b,a%b\n\treturn a"
```

Maintenant mon algorithme est devenu une variable. On peut alors construire une machine universelle capable d'évaluer n'importe quelle donnée contenant un algorithme formalisé dans le langage Python :

```
def universel(algo, *args):
    exec(algo)
    ligne1 = algo.split('\n')[0]
    nom = ligne1.split('(')[0][4:]
    return eval(f"{nom}{args}")
```

A présent je peut invoquer ma machine universelle en lui passant en données :

- la variable contenant mon algorithme
- les arguments sur lequel celui-ci va travailler et obtenir la réponse

```
>>> universel(mon_programme,35,49)
7
```

Dans l'exemple ci-dessus, vous pouvez constater que le programme et les données sur lesquelles il agit sont de même nature : ce sont 3 variables passées en paramètres à ma fonction universelle.

A faire vous même 1.

Saurez-vous citer d'autres exemples ou un programme est considéré comme donnée ?
Quelques réponses possibles :

- Les compilateurs qui prennent en entrée un texte et le transforment en une suite de 0 et 1 exécutable par le microprocesseur de l'ordinateur
- L'interpréteur python fait de même, c'est d'ailleurs lui que nous avons mis à contribution dans notre machine universelle
- Le navigateur internet en est un autre exemple : il reçoit un lot de données d'internet via le protocole http et interprète certaines d'entre elles comme programme (javascript) et d'autres comme données (html) etc...

2 Calculabilité

La calculabilité est la branche de l'algorithmique qui s'intéresse aux questions suivantes :

- Peut-on tout calculer à l'aide d'un ordinateur ?
- Que signifie calculer à l'aide d'un ordinateur ?

Un programme est aussi une donnée. Il est donc manipulable par des algorithmes. Peut-on concevoir un algorithme permettant de savoir si un programme passé en paramètre

- va se terminer ? c'est à dire renvoyer un résultat
- va provoquer une erreur ?
- est correct et ne contient pas de bugs ?

Ce sont des questions fondamentales au cœur de l'algorithmique et de l'informatique en général. Imaginez un programme capable de dire si un autre programme est correct ou bugué ! vous commencez à sentir que cela ne va pas être possible, hélas ...

2.1 Théorème fondamental de la calculabilité

Il existe des problèmes non calculables par des fonctions

L'argument central pour justifier cette affirmation peut s'énoncer en deux phrases :

- l'ensemble des algorithmes que l'on peut programmer par des fonctions (en python par exemple) est dénombrable puisqu'une fonction est une suite finie de caractères
- l'ensemble des problèmes est un ensemble indénombrable (cela peut être démontré) Il n'y a donc tout simplement pas assez d'algorithmes pour résoudre tous les problèmes.

Pour rappel, un infini dénombrable est un infini que l'on peut compter : $\mathbb{N}\mathbb{Z}\mathbb{Q}$ sont dénombrables par opposition à $\mathbb{R}$ qui est indénombrable. Il est impossible de numéroter les nombres réels. L'infini indénombrable des nombres réels est bien plus grand que l'infini dénombrable des entiers.

De même que la majorité des nombres sont des nombres réels (bien plus nombreux que les entiers), la majorité des problèmes ne sont pas calculables !

2.2 Le problème de l'arrêt

Le premier problème explicite non calculable a été décrit par Turing en 1936 : Il s'agit du problème de l'arrêt qui s'énonce ainsi :

étant donné un algorithme A et une prenant en paramètre m,
existe t-il un algorithme permettant de décider si A(m) s'arrête ou pas ?

C'est une question cruciale et pourtant, elle est indécidable : Il n'existe aucun moyen de savoir en général si un algorithme quelconque va s'arrêter ou non.

La conjecture de Syracuse

Un exemple de cette indécision est la conjecture de Syracuse, encore non résolue par les mathématiciens à ce jour. Et pourtant son énoncé est très simple :

Soit un entier n . On définit la suite U_n par récurrence ainsi

- si n est pair, $U_{n+1}=\frac{n}{2}$
- si n est impair, $U_{n+1}=3n+1$

U_0 étant donné, on peut ainsi calculer les termes de proche en proche. Par exemple si on part de 7, on obtient la suite suivante :

```
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

Les derniers termes sont 4, 2, 1. On s'arrête ici dans l'énumération des termes car cela constitue un cycle. En effet 1 est impair donc on fait $3\times1+1=4$ ce qui tourne en boucle.

A ce jour, tous les nombres essayés conduisent inévitablement à ce cycle 4, 2, 1 mais nul n'a été capable de le démontrer.

S'il existait une solution au problème de l'arrêt, alors la conjecture de Syracuse serait résolue car on saurait prédire l'arrêt de l'algorithme. Ce n'est pas le cas;

Preuve de Turing - Attention : Maths inside :)
Pour prouver que le problème de l'arrêt n'est pas calculable, Turing en 1936 a fait ce raisonnement :
On raisonne par l'absurde et on suppose qu'il existe une fonction arrêt qui prend en paramètres un algorithme A et des arguments m (on se rappelle qu'un algorithme est une donnée). Cela pourrait prendre cette forme en Python :

```
def arret(A, m):
    ... # Tout est ici !
    if ...:
        return True
    else:
        return False
```

Le vrai problème est bien sûr de compléter les ... mais on suppose ici que quelqu' un à su le faire.
On peut alors créer un paradoxe à l'aide de l'algorithme suivant (écrit en pseudo python approximatif...):

```
def paradoxe():
    if arret(paradoxe): # paradoxe est une donnée passée en paramètre a arrêt
        while True:
            pass # on fait une boucle infinie si paradoxe s'arrête
     else:
        return True # On s'arrête si paradoxe ne s'arrête pas
```

Que donne la fonction `arret(paradoxe)` (on cherche à savoir si paradoxe s'arrête) ?

- Si paradoxe s'arrête, alors paradoxe ne s'arrête pas car on rentre dans la boucle infinie `while True`.
- Si paradoxe ne s'arrête pas, alors paradoxe s'arrête car le `return True` met fin au programme.

Il est donc impossible de connaître le résultat de l'appel `arret(paradoxe, paradoxe)` car il ne peut valoir `True` et `False` en même temps !

Cette contradiction montre donc que le problème de l'arrêt est indécidable. Il en est de même pour la majorité des problèmes en algorithmique !

Retrouvez cette preuve dans cette vidéo (en anglais, facile à comprendre) :
https://www.youtube.com/watch?v=92WHN-pAFCs

A faire vous même 2.
Visionnez cette vidéo : http://ninoo.fr/LC/Term_NSI/seq20_calculabilite/arte_tv_calculabilite_decidabilite.mp4

A faire vous même 3. Pour les plus motivés
Retrouvez la preuve de Turing dans cette vidéo (en anglais) : https://www.youtube.com/watch?v=92WHN-pAFCs

3 Conclusion

La détection de bugs (comme des boucles infinies par exemple) est une activité cruciale en informatique. Il existe des programmes capables de détecter des erreurs dans d'autres programmes : les environnements de développement modernes comme VScode ou PyCharm sont capable de souligner vos erreurs en python alors même que vous tapez le programme, et cela constitue un grand gain de temps pour le développeur.

En France, depuis l'accident du vol 501 d'Ariane 5 en 1996 du à une erreur de programmation, le développement de programmes de preuves de correction a connu une forte croissance.

Un bel exemple de progrès réalisé grâce à ces programmes est le logiciel de la ligne de métro automatique 14 (Météor) à Paris: cette ligne a été mise en service sans test en condition réelles : son programme a été mathématiquement prouvé sans fautes. Depuis sa mise en service, aucun bugs n'a été à déplorer.

Malheureusement, un algorithme général permettant de prouver qu'un programme fonctionne n'existe pas, cela fait partie des très nombreux problèmes indécidables. Il n'y aura donc jamais de système permettant de s'assurer que n'importe quel programme est fiable, même si c'est réalisable pour quelques exemples particiuliers comme le Météor.

A faire vous même 4.
Pour chacun des problèmes suivant, indiquer s'il est décidable ou indécidable (aucune justification):

Problème 1 : Déterminer pour tout nombre s'il est premier ou pas.
Problème 2 : Déterminer si la politique du Président Trump est bonne pour les Etats-Unis ou pas.
Problème 3 : Déterminer pour tout programme informatique s'il s'arrête ou pas.
Problème 4 : Déterminer pour toute liste de nombres si elle est triée en ordre décroissant ou pas.

A faire vous même 5.

On donne la définition suivante utilisée dans cet exercice: « Deux fonctions `f()` et `g()` sont dites équivalentes si pour tout paramètres d'entrée e, `f(e)` et `g(e)` donnent le même résultat : `f(e) = g(e).` »

1. Analyser les deux fonctions `f` et `g` données ci-dessous. Expliquer en quelques phrases le fonctionnement de chacune d'elles. Les deux fonctions sont-elles équivalentes ? Justifier.

```
def f(n) :
    assert (type(n) == int) & (n >= 0), 'n doit être un entier positif'
    if n == 0:
        return True
    elif n == 1:
        return False
    else:
        return f(n - 2)
def g(n) :
    assert (type(n) == int) & (n >= 0), 'n doit être un entier positif'
    return n%2 == 0
```

2. On cherche maintenant à écrire un programme `equivalent()` pour déterminer d'une façon systématique l'équivalence de deux programmes p1 et p2 pour certaines valeurs x. Les deux programmes p1 et p2 et les valeurs x sont des paramètres du programme `equivalent()` qui renvoie:

- `True` si les programmes p1 et p2 (qui utilisent les données x) produisent le même résultat ; et
- `False` si les résultats sont différents.

Admettons qu'on a déjà écrit entièrement la fonction `equivalent()` dont le code est en partie donné ici:

```
def equivalent(p1, p2, x):
    ''' p1 et p2 sont des fonctions, x est un paramètre -> bool
    renvoie True si p1 et p2 sont équivalentes
    '''
    if …… :
    # teste si p1(x) == p2(x)
        return True
    else:
        return False
```

On réalise maintenant les programmes `boucle_infinie()` et `test(p, x)` faisant appel à `equivalent()`.

```
def boucle_infinie():
    while True :
        print('et ca continue et encore et encore…')
def test(p, x):
    ''' p est une fonction, x est un paramètre -> bool '''
    if equivalence(p, boucle_infinie, x):
        return False
    else:
        return True
```

3. Que renvoie la fonction `test(p, x)` quand p est un programme qui s'arrête? Même question quand p ne s'arrête jamais? En déduire ce que fait le programme `test()`? Que peut-on conclure sur le programme `equivalent()`?