PGR. Calculabilité et décidabilité

I/ Un programme, une donnée comme une autre

Dans une fonction en Python, on peut utiliser divers types de variables en tant que **paramètres** : types *simples*, types *construits* par exemple.

Exemple d'une fonction simple

```
def accueil(n):
    for k in range(n):
        print("bonjour")
```

L'interpréteur est un programme qui permet de traduire à la volée en langage machine un programme écrit dans un langage de haut niveau comme Python : la machine peut le *comprendre* et *l'exécuter*.

Dans une invite de commande, on peut exécuter le programme *test.py* à l'aide du programme python3 : on peut donc utiliser un programme comme paramètres.

```
gilles@gilles-bureau:~/Bureau$ python3 test.py
bonjour
bonjour
bonjour
bonjour
bonjour
```

Un autre exemple ici : https://python.doctor/page-utiliser-interpreteur-python

Un **compilateur** est un programme permet de traduire un programme en langage machine sous forme d'exécutable. On l'utilise dans les langages de bas niveau comme le C 😊.
A noter qu'un compilateur peut être écrit … en C (ce qui est souvent le cas).

Un compilateur pour le langage C : https://www.mycompiler.io/fr/new/c

A savoir : un programme n'est qu'une donnée comme une autre : on peut donc lancer un programme à partir d'un autre.

II/ Un programme : stop ou encore ?

1/ Quelques exemples

Voici un exemple de programme (à droite)
Il s'agit d'un compte à rebours affichant « fini » lorsque le compteur tombe à zéro.
Question : si *n* est un nombre entier naturel, le programme s'arrête naturellement mais qu'en est-il dans les autres cas ?
(la boucle sera infinie)

```
def countdown(n):
    while n != 0:
        print(n)
        n = n - 1
    print("fini")
```

Un autre exemple : *la conjecture de Syracuse*.
Si un nombre est pair, *n* le divise par deux sinon on lui applique la transformation *3*n + 1*.
Il semble que l'on arrive toujours à 4/2/1 à la fin pour tout *n* mais non démontré à ce jour.
Question : comment sait-on si l'on doit arrêter le programme en cas de contre-exemple ?

```
1  def syracuse(N) :
2      # Si N pair, on le divise par deux, sinon 3*N+1
3      while N != 1 :
4          N = N//2 if not N%2 else 3*N + 1
5          print(N, end = "/")
6
7  syracuse(30)
8
```

```
15/46/23/70/35/106/53/160/80/40/20/10/5/16/8/4/2/1/
```

Pour répondre à ces questions, on pourrait imaginer un *analyseur de programme* qui indiquerait si une boucle est infinie ou pas.

2/ Une programme de prédiction de l'arrêt éventuel d'un programme tiers ?

On pourrait imaginer le programme (simple) à droite pour déterminer si un programme nommé *prog* avec comme paramètre *x* s'arrête ou non.
On supposera ici que l'on sait déterminer la condition « prog(x) s'arrête ».

```
def halt(prog, x):
    if "prog(x) s'arrête":
        return True
    else :
        return False
```

Ainsi, avec le programme à droite, on aurait :

- *halt(prog,10)* => renvoie *True*
- *halt(prog,8.8)* = renvoie *False*

```
def countdown(n):
    while n != 0:
        print(n)
        n = n - 1
    print("fini")
```

On considère le programme suivant :

```
def sym(prog):
    if halt(prog, prog) == True:
        while True:
            print("vers l'infini et au-delà !")
    else:
        return 1
```

On peut remarquer que le programme *halt* est appelé avec comme paramètres *prog*, *prog*, ce qui signifie que *prog* se prend lui-même en paramètre (récursivité).

Ce programme *sym* reçoit donc en paramètre un programme *prog*, et :

- va rentrer dans une **boucle infinie** si *prog(prog)* s'arrête.
- va **renvoyer 1** si *prog(prog)* ne s'arrête pas.

Comment savoir si le programme *sym* **s'arrête** ? Il suffirait alors qu'il s'auto-appelle comme ceci :

```
def sym(sym) :
    if halt(sym,sym) == True :
        while True :
            print("Vers l'infini et au-delà")
    else :
        return 1
```

Deux cas se présentent :

- **cas n°1** : *halt(sym, sym)* renvoie *True*, ce qui signifie que *sym(sym)* devrait s'arrêter. Mais dans ce cas-là, l'exécution de *sym(sym)* rentre dans une boucle infinie. C'est une **contradiction**.
- **cas n°2** : *halt(sym, sym)* renvoie *False*, ce qui signifie que *sym(sym)* rentre dans une boucle infinie. Mais dans ce cas-là, l'exécution de *sym(sym)* se termine correctement et renvoie la valeur 1. C'est une **contradiction**.

Ce programme « universel » halt qui pourrait prédire si un programme s'arrête ou pas ne peut donc pas exister.

A savoir : le problème de l'arrêt ne permet pas de déterminer si un programme avec une entrée va s'arrêter ou non.

Ce résultat a été démontré par ***Alan Turing*** en 1936, le mathématicien ***Church*** y parviendra également à la même époque.

Cela a eu une portée considérable dans le monde de l'algorithmique puisque finalement, on ne peut répondre à des énoncés mathématiques comme « un triangle rectangle peut-il être isocèle » ou « existe-t-il d'autres nombres premiers pairs que 2 » dans le cas général.

Le **problème de l'arrêt** est dit **indécidable** car on **ne peut pas le calculer**.

III/ Calculabilité

1/ Notion de calculabilité

Le *calcul* mathématique peut se réduire à une succession d'opérations élémentaires (songez à la multiplication entière comme une série d'additions). Les nombres calculables sont les nombres qui sont **générables** en un nombre **fini d'opérations élémentaires**. De la même manière, une fonction mathématique sera dite **calculable** s'il existe une **suite finie d'opérations élémentaire**s permettant de passer d'un nombre **x** à **son image f(x).**

On retrouve cette notion d'opérations élémentaires dans les machines de Turing. Cette machine (théorique) permet de simuler tout ce qu'un programme informatique (une suite d'instructions) est capable d'exécuter.
Un algorithme peut se réduire à une suite d'opérations élémentaires, comme une fonction mathématique peut se réduire à une suite de calculs. Dès lors, on pourra considérer un algorithme comme une fonction.

Turing a démontré que l'ensemble des **fonctions calculables**, au sens de ***Church***, était équivalent à l'ensemble des **fonctions programmables** sur sa machine. Certaines fonctions peuvent être calculables, *ou ne pas l'être* : c'est notamment le cas de notre fonction du problème de l'arrêt.

A savoir : tous les langages de programmation suivent les principes énoncés par Turing : la calculabilité ne dépend pas du langage choisi.

2/ Calculable ou pas ?

En informatique, un autre critère en ligne de compte pour la calculabilité que la dualité calculable/non calculable vue précédemment : la **complexité**.
L'utilisation de la récursivité est un bon exemple, on peut citer la suite de Fibonacci ici, https://github.com/lmayer65/NSI_T/blob/main/Programmation/Programmation_dynamique/PRG.Programmation_dynamique.pdf où l'on peine à calculer le centième terme 😟 : ce n'est pas un bon algorithme.

On distingue deux types d'algorithmes :

Classe P	Classe NP
- Algorithmes de **complexité polynomiales** donc calculables. - Tous les problèmes à complexité logarithmique, linéaires, quadratiques mais pas exponentielle. Exemples : algorithmes de tris, de recherche d'éléments, test de primalité …	- Algorithmes de recherche **non déterministe polynomial.** - Complexité polynomiale avec une **machine non déterministe**, explorant plusieurs solutions à la fois. - Complexité polynomiale avec une **machine déterministe** pour la vérification d'une solution.

Lien vers une machine non déterministe :
https://fr.wikipedia.org/wiki/Machine_de_Turing_non_d%C3%A9terministe

Lien vers le test de primalité :
https://fr.wikipedia.org/wiki/Test_de_primalit%C3%A9_AKS#:~:text=Le%20test%20de%20primalit%C3%A9%20AKS,Agrawal%2C%20Neeraj%20Kayal%20et%20Nitin

Pour le résumer très grossièrement, un problème de classe NP est un problème dont on sait vérifier facilement si une solution proposée marche ou pas :

- la résolution d'un sudoku est dans NP : si quelqu'un vous montre un sudoku rempli, vous pouvez très rapidement lui dire si sa solution est valable ou pas.
- la factorisation d'un nombre est dans NP : si quelqu'un vous propose 4567*6037 comme décomposition de 27570979, vous pouvez très rapidement lui dire s'il a raison. (oui.)
- le problème du sac à dos est dans NP. Une proposition de butin peut facilement être examinée pour savoir si elle est possible ou non.
- le problème du voyageur de commerce, en version décisionnelle, est dans NP. Si on vous propose un trajet, vous pouvez facilement vérifier que sa longueur est (par exemple) inférieure à 150 km.

Malheureusement, aucun de ces problèmes cités n'a (à ce jour) d'algorithme de **résolution** meilleur qu'exponentiel...

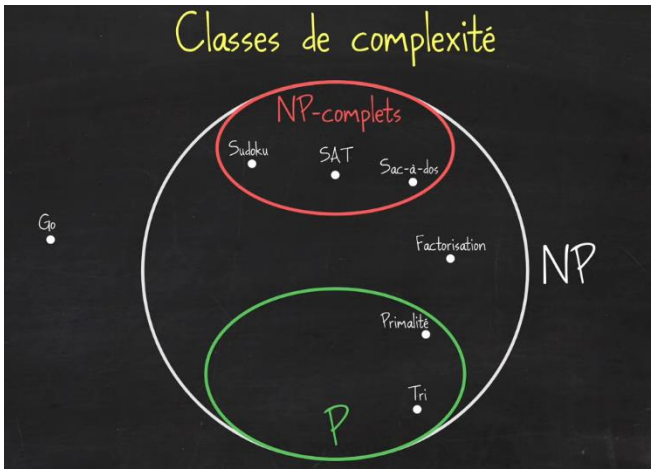
Lien vers le problème du voyageur de commerce :
https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_voyageur_de_commerce

3/ P = NP ?

Tous les problèmes de P ont une solution qui peut être **trouvée** de manière polynomiale. Donc évidemment, la vérification de cette solution est aussi **polynomiale**. Donc tous les problèmes de P sont dans NP. On dit que P est inclus dans NP, que l'on écrit **P ⊂ NP**.

Une vidéo très bien faite (et la question à 1 million de dollars) : https://www.youtube.com/watch?v=AgtOCNCejQ8
Voici la page d'introduction :

On y retrouve (en **vert**) la classe P, qui contient les algorithmes de tri.
En **blanc**, la classe NP, qui contient les problèmes de factorisation, du sudoku, du sac-à-dos...
En **rouge**, les NP avec une propriété remarquable : la résolution polynomiale d'un seul d'entre eux ferait ramener la **totalité** des problèmes NP dans P.
Si on trouve une résolution polynomiale du sudoku, alors on aura démontré que P = NP … et la fin des secrets bancaires !



Actuellement, les chercheurs ne pensent pas que P = NP sauf un certain ***Donald Knuth***, l'un des artisans de l'algorithmique moderne.

Lien vers Donald Knuth, https://fr.wikipedia.org/wiki/Donald_Knuth