Quelques notions de calculabilité et de décidabilité condensées en un temps déraisonnablement court

David Renault

Présentation aux élèves de Terminale du lycée Václav Havel

le 26 mai 2021

Avertissement

Afin de pouvoir tenir en un temps raisonnable, ces slides font un certain nombre d'approximations et de choix sur les thèmes abordés et les personnes citées.
En aucun cas, ils ne peuvent remplacer un cours sérieux sur le sujet, que ce soit historiquement ou scientifiquement.
Pour approfondir le sujet, quelques références sont données à la fin.

Le programme NSI de Terminale

Contenus	Capacités attendues	Commentaires
Notion de programme en tant que donnée. Calculabilité, Décidabilité.	Comprendre que tout programme est aussi une donnée. Comprendre que la calculabilité ne dépend pas du langage de programmation utilisé. Montrer, sans formalisme théorique, que le problème de l'arrêt est indécidable.	L'utilisation d'un interpréteur ou d'un compilateur, le téléchargement de logiciel, le fonctionnement des systèmes d'exploitation permettent de comprendre un programme comme donnée d'un autre programme.

Qu'est-ce que considérer un programme comme une donnée ?

. . . et pourquoi serait-ce si passionnant ?

- Au départ, cela semble quelque chose de peu naturel (les données, usuellement, ce sont des nombres, des chaînes de caractères . . .)

```
def solve_the_riddle(a_number, a_string):
    a_result = a_number * 7
    print(f"The␣{a_string}␣is␣{a_result}")

solve_the_riddle(6, "answer") # Prints "The answer is 42"
```

Qu'est-ce que considérer un programme comme une donnée ?

. . . et pourquoi serait-ce si passionnant ?

- Au départ, cela semble quelque chose de peu naturel (les données, usuellement, ce sont des nombres, des chaînes de caractères . . .)
- Quoique si on réfléchit bien, il est parfois utile de pouvoir considérer les fonctions comme des données.

```
def check_the_function(a_function, a_param, a_result):
    assert a_function(a_param) == a_result, "Something is wrong"

check_the_function(math.sqrt, 4, 2)
check_the_function(math.sqrt, 4, 3) # Error : something is wrong
```

Qu'est-ce que considérer un programme comme une donnée ?

. . . et pourquoi serait-ce si passionnant ?

- Au départ, cela semble quelque chose de peu naturel (les données, usuellement, ce sont des nombres, des chaînes de caractères . . .)
- Quoique si on réfléchit bien, il est parfois utile de pouvoir considérer les fonctions comme des données.
- Et si on considère un programme comme une grosse fonction, cela peut ressembler à quelque chose de sensé.

```
def compile_program(a_program):
    # complex code hidden here

def execute_program(a_program):
    # the same
```

Qu'est-ce que considérer un programme comme une donnée ?

. . . et pourquoi serait-ce si passionnant ?

- Au départ, cela semble quelque chose de peu naturel (les données, usuellement, ce sont des nombres, des chaînes de caractères . . .)
- Quoique si on réfléchit bien, il est parfois utile de pouvoir considérer les fonctions comme des données.
- Et si on considère un programme comme une grosse fonction, cela peut ressembler à quelque chose de sensé.

```
def compile_program(a_program):
    # complex code hidden here

def execute_program(a_program):
    # the same
```

En fait, tout système d'exploitation est un programme conçu pour exécuter d'autres programmes . . .

Les programmes comme des données (1/3)

En pratique, il n'est pas difficile de trouver des exemples de programmes vu comme des objets tangibles :

- Une recette de cuisine ;
- Un manuel d'instructions pour construire un meuble ;

Les programmes comme des données (1/3)

En pratique, il n'est pas difficile de trouver des exemples de programmes vu comme des objets tangibles :

- Une recette de cuisine ;
- Un manuel d'instructions pour construire un meuble ;
- Un jeu de cartes un tant soit peu complexe :

Image : wiki.dominionstrategy.com

Les programmes comme des données (2/3)

Dans chacun de ces exemples, on peut identifier le programme et l'entité exécutant le programme.

- Une recette / Une personne aux fourneaux ;
- Un manuel / Une personne aux outils ;
- Un jeu de cartes / Une personne jouant le jeu ;
- Un programme / Un interprète du programme.

Les programmes comme des données (3/3)

Pourquoi est-ce aussi remarquable ?

> En informatique, le programme et son interprète peuvent être de la même nature.

On peut donc :

- écrire des programmes qui consomment des programmes (interpréteur Python)
- écrire des programmes qui transforment des programmes en d'autres programmes (compilateur Cython $\rightarrow$ code C)
- écrire des programmes qui génèrent de nouveaux programmes potentiellement inattendus (ex. AlphaGo)

A un moment se pose la question de ce que l'on peut calculer avec un langage de programmation.

- Par exemple, quel est l'ensemble des stratégies possibles dans un jeu de cartes comme Dominion ou comme le Go ?

Nombre de positions légales sur un plateau de Go classique	$\approx 2.1 \times 10^{170}$
Nombre d'atomes de l'univers observable	$\approx 1 \times 10^{80}$

- Mais le problème n'est pas vraiment dans le nombre total d'opérations que l'on peut réaliser ...

Capacité de calcul du 1er calculateur du top 500 en 1993	$\approx 1.3 \times 10^{9}$ flops
en 2020	$\approx 4.4 \times 10^{15}$ flops

- La vraie question qu'on se pose est en fait :

> Existe t'il une limite que l'on ne peut pas et qu'on ne pourra jamais dépasser ?

Que peut-on faire avec un langage de programmation ?

Qu'est-ce qu'un langage de programmation ?

Un langage de programmation est un langage permettant d'exprimer des manières de réaliser des calculs.

Ex. : Python est un bon point de départ (pour les connaisseurs).

Qu'est-ce que la calculabilité ?

Un domaine de l'informatique se posant la question de ce qui est calculable avec un langage de programmation.

Ex. : factoriser un nombre entier en produits de nombres premiers

Dans le cadre qui nous intéresse, l'ensemble de ce qui est calculable est indépendant du langage de programmation.

C, C++, C#, Haskell, Java, Javascript, OCaml, Python, Ruby, Scala ...

Que peut-on faire avec un ensemble de règles logiques ?

En fait, la question se pose aussi pour un système logique.

Qu'est-ce qu'un système logique ?

Un système logique est un ensemble d'axiomes et de règles de déduction dont la finalité est d'établir des théorèmes.

Quelques exemples de systèmes logiques :

- Les mathématiques dans leur entièreté.
- La logique booléenne.
- La géométrie euclidienne.

Qu'est-ce que la décidabilité ?

Un domaine de l'informatique se posant la question des théorèmes qu'un système logique est capable d'exprimer.

Ex. : définir la propriété "être un nombre premier"

Historiquement, ces questions ont été initialement posées pour axiomatiser le système logique que sont les mathématiques :

- l'*Idéographie* de Gottlob Frege en 1879
- les *Principia Mathematica* de Bertrand Russell et Alfred Whitehead en 1910

Ces personnages illustres se sont trouvés confrontés à des chausse-trapes de la logique impossibles à isoler : des paradoxes.

Paradoxe du menteur

Cette phrase est un mensonge.

Paradoxe du menteur

Cette phrase est un mensonge.

- Il n'y a pas moyen de dire si cette phrase affirme une proposition vraie ou fausse. Il s'agit d'un paradoxe logique.
- Une des propriétés importantes de ce paradoxe est son côté auto-référentiel : le mensonge est une caractéristique de la phrase entière, qui contredit qu'elle est un mensonge.

Paradoxe de Berry

Considérons la définition mathématique suivante :

> Le plus petit nombre entier que l'on ne peut pas définir en moins de vingt mots.

Paradoxe de Berry

Considérons la définition mathématique suivante :

> Le plus petit nombre entier que l'on ne peut pas définir en moins de vingt mots.

- L'ensemble des phrases que l'on peut écrire en français sur 20 mots est fini.

Paradoxe de Berry

Considérons la définition mathématique suivante :

> Le plus petit nombre entier que l'on ne peut pas définir en moins de vingt mots.

- L'ensemble des phrases que l'on peut écrire en français sur 20 mots est fini.
- L'ensemble des nombres entiers qu'on peut définir en 20 mots est donc aussi fini.

Paradoxe de Berry

Considérons la définition mathématique suivante :

> Le plus petit nombre entier que l'on ne peut pas définir en moins de vingt mots.

- L'ensemble des phrases que l'on peut écrire en français sur 20 mots est fini.
- L'ensemble des nombres entiers qu'on peut définir en 20 mots est donc aussi fini.
- Son complémentaire existe, et contient un plus petit élément.

Paradoxe de Berry

Considérons la définition mathématique suivante :

> Le plus petit nombre entier que l'on ne peut pas définir en moins de vingt mots.

- L'ensemble des phrases que l'on peut écrire en français sur 20 mots est fini.
- L'ensemble des nombres entiers qu'on peut définir en 20 mots est donc aussi fini.
- Son complémentaire existe, et contient un plus petit élément.
- Problème : sa définition tient en moins de 20 mots . . .

- Si la logique veut offrir des bases stables pour les mathématiques, elle doit pouvoir rendre de tels paradoxes clairement identifiables.
- Mais les choses ne sont malheureusement pas si simples.
  Kurt Gödel, en 1931, publia ses théorèmes d'incomplétude :

Théorème d'incomplétude de Gödel

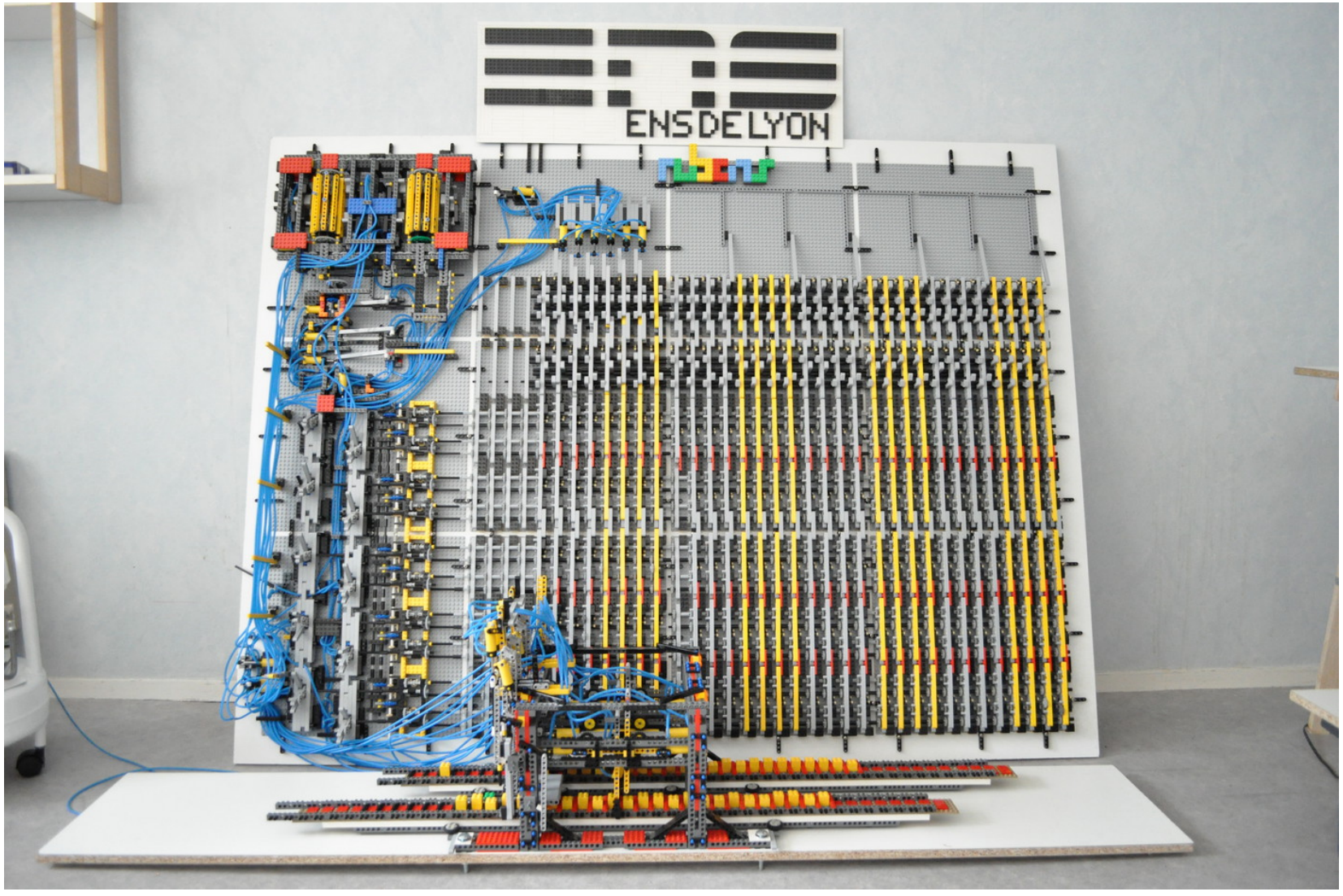
Dans tout système logique non trivial, et sans inconsistance, il existe des théorèmes vrais qu'il n'est pas possible de prouver.

Une catastrophe : établir un socle complet des mathématiques est impossible.

A peu près au même moment, Alonzo Church d'un côté et son élève Alan Turing se posent la question d'une manière un peu différente :

- Plutôt que d'axiomatiser toute la logique, ils proposent de construire des modèles de calcul (des langages de programmation).
- Ils proposent chacun de leur côté :
  - le lambda-calcul proposé en 1936 par Church,
  - les machines de Turing proposées en 1937 par Turing.
- Ces modèles se veulent plus simples que les modèles de logique au sens où ils sont constructifs.
- Ils décrivent des procédures qu'on peut appliquer pas à pas avec des règles "simples".

Une machine de Turing moderne :)



Source : `http://rubens.ens-lyon.fr`

Vidéo : `https://interstices.info/comment-fonctionne-une-machine-de-turing`

Leurs modèles sont en fait équivalents et suffisamment expressifs pour qu'ils puissent affirmer :

Thèse de Church-Turing

Toute fonction "effectivement calculable" est calculable dans le lamda-calcul et les machines de Turing.

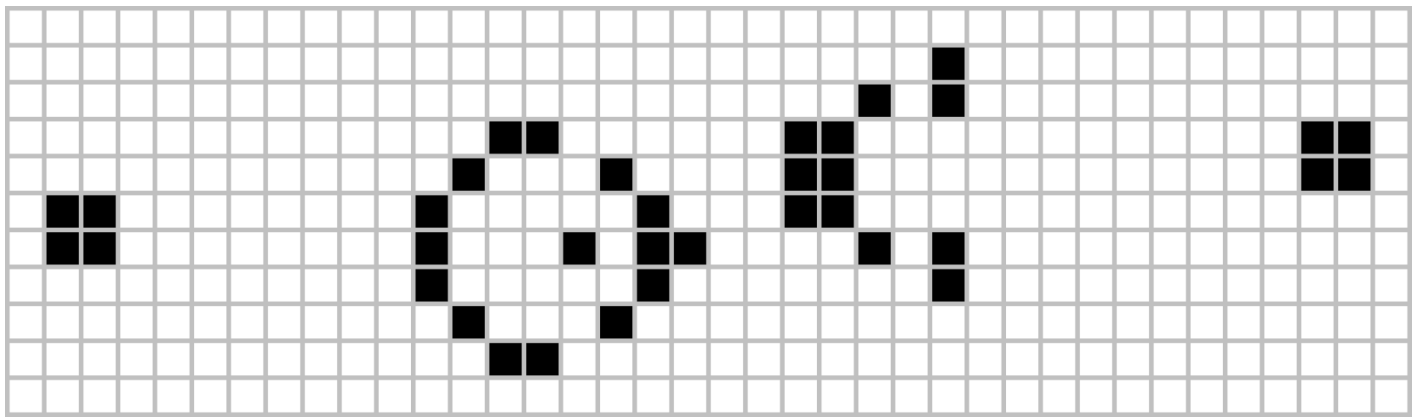
Cela pose évidemment la question de ce qui est "effectivement calculable", mais cette thèse est une sorte de définition de ce qui est calculable.

- Un langage de programmation est dit Turing-puissant s'il est capable dans ses programmes de réaliser tout ce qui est calculable.
- Les langages de programmation contemporains comme Python sont Turing-puissants et équivalents en terme d'expressivité.

Conway's Game of Life

Un exemple de modèle de calcul Turing-puissant : le jeu de la vie.

- Conçu par le mathématicien John Conway en 1970
- Basé sur l'idée de cellules disposées sur une grille infinie, et évoluant au cours du temps.
- A chaque seconde, une cellule de la grille survit selon le nombre de voisins vivants autour d'elle à la génération précédente :
  - 0 ou 1 voisins : elle meurt du sous-population ;
  - 2 ou 3 voisins : elle survit ;
  - 3 voisins exactement : elle naît ;
  - 4 voisins ou plus : elle meurt de surpopulation.



Source : https://en.wikipedia.org/wiki/Conway's_Game_of_Life
Voir aussi : https://interstices.info/a-la-decouverte-des-automates-cellulaires

Autant les modèles logiques avaient leur paradoxes, autant les langages de programmation ont leurs incongruités propres :

- Dans tout langage Turing-puissant, il existe des programmes qui ne terminent pas.

```
def loop():
    cpt = 0
    while True:
        cpt = cpt + 1
```

- Si la finalité d'un programme est de faire un calcul produisant un résultat, ce comportement est difficilement acceptable.

Remarque : il existe des programmes ne terminant pas pour de bonnes raisons.

- Ne pas s'arrêter, c'est ne pas fournir de réponse au calcul, une indécision équivalente à celle des paradoxes logiques.

Qu'à cela ne tienne : inventons une procédure (un programme ?) qui, étant donné un programme, renvoie dans tous les cas un booléen disant si ce programme termine ou pas

```
def does_it_stop(a_program):
    # some probably complicated code
```

Par exemple :

```
def loop_long():
    cpt = 0
    while cpt >= 0:
        cpt = cpt + 1

does_it_stop(loop_long) # → False
```

```
def loop_short():
    cpt = 100
    while cpt >= 0:
        cpt = cpt - 1

does_it_stop(loop_short) # → True
```

- L'écriture de `does_it_stop` n'est pas évidente.
- Mais on pourrait imaginer une sorte de compteur de boucles `while` intelligent qui . . .
- En fait, il existe des programmes pour lesquels la terminaison est vraiment difficile à déterminer, comme la suite de Syracuse :

```
def syracuse():
    start = 1000
    while start > 1:
        if start % 2 == 0:
            start = start / 2
        else:
            start = 3*start + 1
```

Les 25 premières valeurs de `start` :

1000, 500, 250, 125, 376, 188 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364 182, 91, 274, 137, 412, 206, 103 . . .

- L'écriture de `does_it_stop` n'est pas évidente.
- Mais on pourrait imaginer une sorte de compteur de boucles `while` intelligent qui . . .
- En fait, il existe des programmes pour lesquels la terminaison est vraiment difficile à déterminer, comme la suite de Syracuse :

```
def syracuse():
    start = 1000
    while start > 1:
        if start % 2 == 0:
            start = start / 2
        else:
            start = 3*start + 1
```

Les 25 premières valeurs de `start` :

1000, 500, 250, 125, 376, 188 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364 182, 91, 274, 137, 412, 206, 103 . . .

Il faut 111 itérations pour que le programme termine.

Considérons le code suivant basé sur le paradoxe du menteur :

```
def paradoxical_code(a_program):
    if does_it_stop(a_program):
        while True:
            pass
    else:
        return

def strange_program():
    return paradoxical_code(strange_program)
```

Que peut retourner l'appel `does_it_stop(strange_progam)` ?

Considérons le code suivant basé sur le paradoxe du menteur :

```
def paradoxical_code(a_program):
    if does_it_stop(a_program):
        while True:
            pass
    else:
        return

def strange_program():
    return paradoxical_code(strange_program)
```

Que peut retourner l'appel `does_it_stop(strange_progam)` ?

- s'il renvoie `True`, `paradoxical_code` entre dans une boucle infinie ;

Considérons le code suivant basé sur le paradoxe du menteur :

```
def paradoxical_code(a_program):
    if does_it_stop(a_program):
        while True:
            pass
    else:
        return

def strange_program():
    return paradoxical_code(strange_program)
```

Que peut retourner l'appel `does_it_stop(strange_progam)` ?

- s'il renvoie `True`, `paradoxical_code` entre dans une boucle infinie ;
- s'il renvoie `False`, alors le programme termine.

Considérons le code suivant basé sur le paradoxe du menteur :

```
def paradoxical_code(a_program):
    if does_it_stop(a_program):
        while True:
            pass
    else:
        return

def strange_program():
    return paradoxical_code(strange_program)
```

Que peut retourner l'appel `does_it_stop(strange_progam)` ?

- s'il renvoie `True`, `paradoxical_code` entre dans une boucle infinie ;
- s'il renvoie `False`, alors le programme termine.

Impossible dans les deux cas !

Seule conclusion possible : l'appel en question ne termine pas.

Le théorème de l'arrêt

- Une réflexion plus poussée amène à prouver qu'un programme comme `does_it_stop` ne peut pas exister.
- Il s'agit du théorème fondamental suivant :

Indécidabilité du problème de l'arrêt

Il n'existe pas de programme capable de dire, étant donné un programme P, si P s'arrêtera ou non.

Le problème de l'arrêt est un problème indécidable.

Pourquoi une telle quantité de réflexion ? (1/3)

- Savoir qu'une frontière existe, avec impossibilité de la dépasser pour tout un ensemble de problèmes, est important.
- En fait, il existe même un théorème de Rice qui dit que la plupart des propriétés (dites non triviales) des programmes sont indécidables.
- 3 exemples de propriétés indécidables :
  - "Décider si un programme écrit dans un langage comme Python termine sans erreur"
  - "Décider si deux programmes Python renvoient les mêmes valeurs sur les mêmes entrées"
  - "Décider si une suite d'entiers calculable par un programme Python converge"

Pourquoi une telle quantité de réflexion ? (2/3)

Si les problèmes sont indécidables en toute généralité, la notion de décidabilité n'est pas forcément non plus très pratique :

- Dans un cadre borné (mémoire, temps), beaucoup de problèmes sont décidables.
  - Trouver une stratégie gagnante (si elle existe) à partir d'une position donnée aux échecs, au go.
  - Déchiffrer une communication chiffrée utilisant n'importe quel protocole de chiffrement actuel.
- Il existe des problèmes décidables dont la complexité est une tour d'exponentielle en la taille des données en entrée . . .

$$n \to \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n \text{ fois}}$$

Pourquoi une telle quantité de réflexion ? (3/3)

Alors à quoi ça peut bien servir au final ?

- A trouver d'autres manières de raisonner sur les problèmes
- Des cadres plus restreints, plus sécurisés, sur lesquels les questions qu'on se pose ont des réponses.
  - "Programmer en Python avec des types pour assurer une forme de fiabilité des programmes."
- A ne pas s'attaquer bille en tête à des questions trop générales sous peine de se casser les dents.

Au final, cela encourage à faire preuve d'inventivité et de méthode dans la façon de penser les problèmes de l'informatique.

Un peu de lecture pour la route. . .

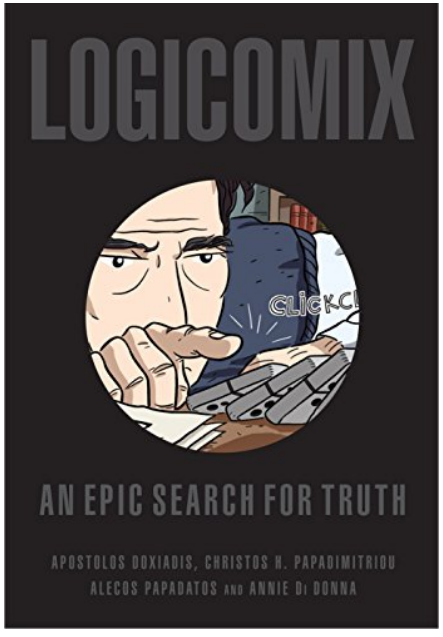


Image : www.amazon.fr

Logicomix : An epic search for truth
par Apostolos Doxiadis et Christos Papadimitriou

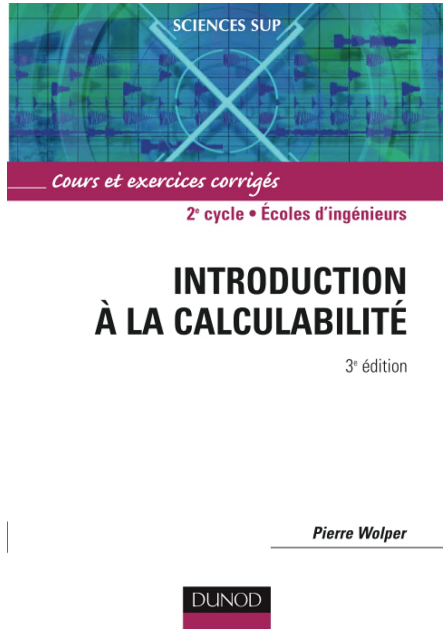


Image : www.amazon.fr

Introduction à la calculabilité
par Pierre Wolper