Calculabilité – Décidabilité

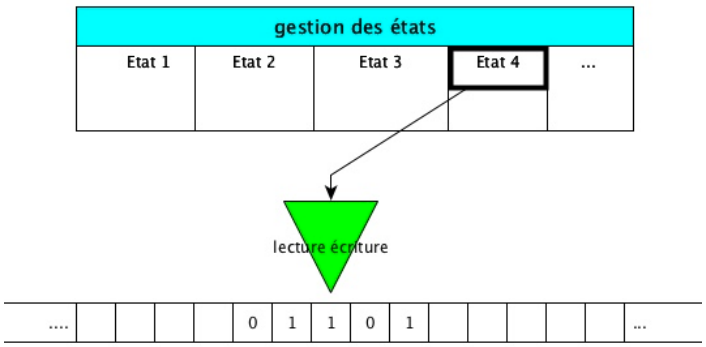
Remarque introductive : ce cours donne une introduction aux problèmes : que veut dire « calculer » ? Que peut-on calculer ? Qu'est-ce qu'un programme ?

1. Introduction : les machines de Turing

Les machines de Turing ne sont pas au programme, mais c'est rigolo et plutôt simple.
Turing a inventé sa machine avant l'invention de l'ordinateur. Elle est censée représenter le fonctionnement du cerveau d'un mathématicien...
Une machine de Turing est une machine théorique pouvant effectuer des calculs. Elle comporte :

- un ruban infini, constitué d'une infinité de cases pouvant contenir des symboles 0, 1 ou □ (case vide) ;
- un ensemble fini d'états ;
- une tête de lecture/écriture, qui à tout instant se trouve au dessus d'une case du ruban et peut se déplacer de cases en cases vers la gauche ou la droite ;
- une table de transition qui définit le comportement de la machine : en fonction de l'état courant et du symbole lu par la tête, la machine écrit un nouveau symbole sur le ruban, déplace sa tête de lecture (d'une case à gauche ou à droite) et passe dans un nouvel état.

En général, l'état initial est sur la gauche du ruban.



Exemples :

- Machine de Turing renvoyant 1 si un nombre est de la forme 0...01...1, avec autant de 0 et de 1 que l'on souhaite, et 0 sinon. Le nombre de 0 et de 1 n'est pas obligatoirement le même
  Algorithme :
  Tant que la case est vide, la tête se déplace à droite et la machine est dans l'état Initialisation
  Si le symbole le plus à gauche est 1, alors la machine s'arrête dans l'état Faux.
  Sinon
  Tant que le symbole lu est 0, alors la tête se déplace à droite, la machine est dans l'état Droite
  Si la machine lit une case vide, alors elle s'arrête dans l'état Faux
  Tant que le symbole lu est 1, alors la tête se déplace à droite, la machine est dans l'état Test
  Si la machine lit une case vide, alors elle s'arrête dans l'état Vrai
  Sinon elle s'arrête dans l'état Faux

Table de transition : la table de transition explique le fonctionnement de l'algorithme

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	☐	*idem*	→	INIT
INIT	1	*idem*		FAUX
INIT	0	*idem*	→	DROITE
DROITE	☐	*idem*		FAUX
DROITE	1	*idem*	→	TEST
TEST	☐	*idem*		VRAI
TEST	0	*idem*		FAUX

- Machine de Turing qui ajoute 1 à un nombre en binaire
  Algorithme :
  Tant que la case est vide, la tête se déplace à droite et la machine est dans l'état Initialisation
  Tant que le symbole lu est 0 ou 1, la machine se déplace à droite dans l'état Lecture
  Si le symbole lu est vide, la machine se déplace à gauche dans l'état Addition
  Tant que le symbole lu est 1, la machine écrit dans la case courante 0, et se déplace à gauche dans l'état Addition.
  Si le symbole lu est 0 ou vide , la machine écrit dans la case courante 1, et s'arrête dans l'état Fin

  Table de transition

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	☐	*idem*	→	INIT
INIT	0/1	*idem*	→	LECTURE
LECTURE	☐	*idem*	←	ADDITION
ADDITION	1	0	←	ADDITION
ADDITION	0/☐	1		FIN

- Une machine de Turing en Lego : https://www.dailymotion.com/video/xrn0yi

2. Calculabilité

Une fonction f est dite calculable si on peut obtenir le résultat $f(x)$ en un nombre fini d'étapes, réalisables par un humain avec un papier et un crayon, sans faire appel à l'intelligence de l'humain si ce n'est pour suivre les instructions. En simplifiant, une fonction calculable est une fonction dont on peut trouver le résultat algorithmiquement.
Remarque : f est une fonction plus dans le sens que vous connaissez en informatique qu'en mathématiques. De même x est un tuple de variables.

Thèse de Church-Turing : tous les langages de programmations (Python, C, JavaScript, …) donnent les mêmes fonctions calculables. Il en est de même des machines de Turing, du lambda-calcul, et même Minecraft, ou le CSS3[1].
Remarque : une thèse n'est pas un théorème, c'est par principe indémontrable. En effet, on a montré l'équivalence pour les exemples cités ci-dessus, mais pas pour tous les formalismes que l'on ne connaît pas encore.
On a répondu aux questions « que veut dire calculer ? » et « Que peut-on calculer ? ». Calculer, et ce que qu'on peut calculer, c'est tout ce qui se code dans un langage de programmation, ce qui équivalent à « tout ce qui peut se traduire dans le jeu de la vie », tout ce qui peut se « programmer » dans Minecraft, tout ce qui peut se programmer dans une machine de Turing, etc.

[1] En effet celui-ci permet de programmer le jeu de la vie, qui lui-même permet de calculer les mêmes fonctions que Python

Il est trompeur de croire que les fonctions non calculables n'ont pas d'intérêt ou n'existent pas. Avec des arguments mathématiques détaillés ci-après, il y a infiniment plus de fonctions non calculables que de fonctions calculables. Quelques-unes de ces fonctions non calculables peuvent s'exprimer en termes compréhensibles par les humains. Le castor affairé en est une, c'est déjà difficile à comprendre, cf https://fr.wikipedia.org/wiki/Castor_affair%C3%A9 pour voir en fin de page la croissance extrêmement rapide de cette fonction : c'est cette rapidité qui la rend non calculable.
Éléments de preuve pour « il y a infiniment plus de fonctions non calculables que de fonctions calculables » : on admet que l'infini de $\mathbb{R}$ est beaucoup plus grand que celui de $\mathbb{N}$. Si on code les lettres par des entiers, un algorithme est donc une suite finie d'entiers. On peut écrire une infinité de programmes, mais on admet que cet infini sera toujours de la même nature que celui de $\mathbb{N}$, soit une goutte d'eau –infinie certes– dans l'océan infiniment plus grand des fonctions définies sur $\mathbb{R}$. Toutes les fonctions qui ne peuvent pas s'écrire ainsi sont non calculables. Et plus philosophiquement, non imaginables par un esprit humain limité par le langage.

Complément hors programme : les problèmes de décision[2] qui se calculent « bien » sont les problèmes pour lesquels il existe un algorithme polynomial, de complexité $O\left(n^p\right)$. Ces problèmes sont dits de classe P, ce sont les problèmes « faciles ». La classe NP comporte des problèmes qu'on ne sait pas résoudre en temps polynomial, comme le chemin hamiltonien (dans un graphe, existe-t-il un chemin qui passe une fois et une seule par chaque sommet), trouver la solution d'une grille de sudoku, trouver la plus longue sous-séquence commune de caractères avec un nombre quelconque de chaînes, minimiser le temps total de processus de durées connues dans un système multi-processeur, problème du voyageur de commerce (un voyageur doit passer par plusieurs villes, existe-t-il un chemin de longueur inférieure à k ?), éviter les interblocages, etc. Par contre tous ces problèmes ont comme particularité que l'on peut en vérifier une solution facilement, en temps polynomial (vérifiez !). Une des grandes questions en informatique est « est-ce que ces deux classes sont différentes ou égales, autrement dit est-ce que $P \neq NP$? ». En effet, il suffit que l'on trouve un algorithme polynomial pour certains problèmes de classe NP (les problèmes appelés NP-complets[3]), pour que tous les problèmes de cette classe s'y ramènent. Il y aurait alors égalité de P et NP. On n'a jamais trouvé un tel algorithme. A l'inverse personne n'a jamais réussi à prouver la différence. Si vous arrivez à l'un ou l'autre, alors à vous le million de dollars promis pour la résolution de cette question !

3. Decidabilite

On s'intéresse ici à : étant donné un problème de décision, peut-trouver un algorithme qui le résout systématiquement ? Cela répond partiellement à « que peut-on calculer ? »
On va tout de suite voir que certains problèmes ne sont pas calculables[4].
On cherche à répondre au problème suivant : « je viens de taper mon code, qui par exemple met plusieurs jours à tourner, et j'aimerais bien le passer dans une moulinette qui me dit si mon programme s'arrête ou pas, avec les données que vais lui fournir ». Plus formellement, on cherche à construire une fonction `arrêt(f, x)`, qui renvoie `True` si le calcul de `f(x)` se termine et `False` sinon. Ce problème est connu sous le nom de problème de l'arrêt.

On va montrer que cette fonction n'existe pas, en raisonnant par l'absurde, c'est à dire en supposant qu'elle existe, et en arrivant à un résultat contradictoire.
On suppose donc que quelqu'un a réussi à programmer la fonction arrêt. Nous allons utiliser cette fonction pour programmer la fonction étrange :

[2] Problème dont le résultat est soit Vrai, soit Faux.
[3] Tous les problèmes précités en font partie, sauf le sudoku.
[4] La quasi-totalité des problèmes de décision est incalculable. Comme en mathématiques, on ne fait en informatique que des choses très simples, l'immense majorité étant beaucoup trop compliquée. Et ce n'est pas une blague.

```
def étrange(f, x) :
      if arrêt(f,x) :
            while True :
                  pass
```

Cette fonction teste si le calcul de f(x) termine, et si c'est le cas, rentre dans une boucle infinie qui ne fait rien[5]. A partir de cette fonction, on en crée encore une autre : la fonction paradoxe.

```
def paradoxe(f, f) :
      étrange(f,f)
```

Le calcul de `paradoxe(f)` se termine si se seulement si le calcul de `f(f)` ne se termine pas (*)
Examinons ce qui se passe quand on lance `paradoxe(paradoxe)`. Dans la phrase (*), on remplace `f` par `paradoxe` et on obtient :

Le calcul de `paradoxe(paradoxe)` se termine si se seulement si le calcul de `paradoxe(paradoxe)` ne termine pas !
On est donc arrivé à une contradiction, donc notre hypothèse de départ est fausse : la fonction n'existe pas. On a montré le :
Théorème de l'arrêt : le problème de l'arrêt est incalculable.

Deux vidéos supplémentaires pour peut-être mieux comprendre la preuve de théorème :
https://www.youtube.com/watch?v=92WHN-pAFCs (en anglais simple)
https://www.youtube.com/watch?v=13O1qhX4Bqo (basée sur la démonstration utilisée dans ce cours)

Remarque : Ce théorème résonne avec la question « Entscheidungsproblem[6] » du mathématicien David Hilbert (1862-1943), à savoir « montrer de façon mécanique si un théorème mathématique est vrai ou faux ». Turing et Church ont montré indépendamment que ce problème est indécidable.

4. Programme en tant que donnee

Qu'est-ce qu'un programme ? Un programme est une donnée comme une autre tout simplement. Tous les programmes Python -c'est aussi valable pour les autres langages- sont définis par leur chaîne de caractères. Cela signifie que l'on peut faire des calculs sur un programme/algorithme que l'on a crée. Cela doit vous rappeler la programmation fonctionnelle, où l'on a vu que des fonctions, donc des programmes, peuvent être des paramètres d'entrée, ou bien des résultats de fonctions.
Le programme suivant permet d'interpréter un algorithme sur une entrée donnée.

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

pg = 'def f(x):\n\treturn 3*x'
print(universel(pg,17))
```

Que renvoie l'exemple donné ?

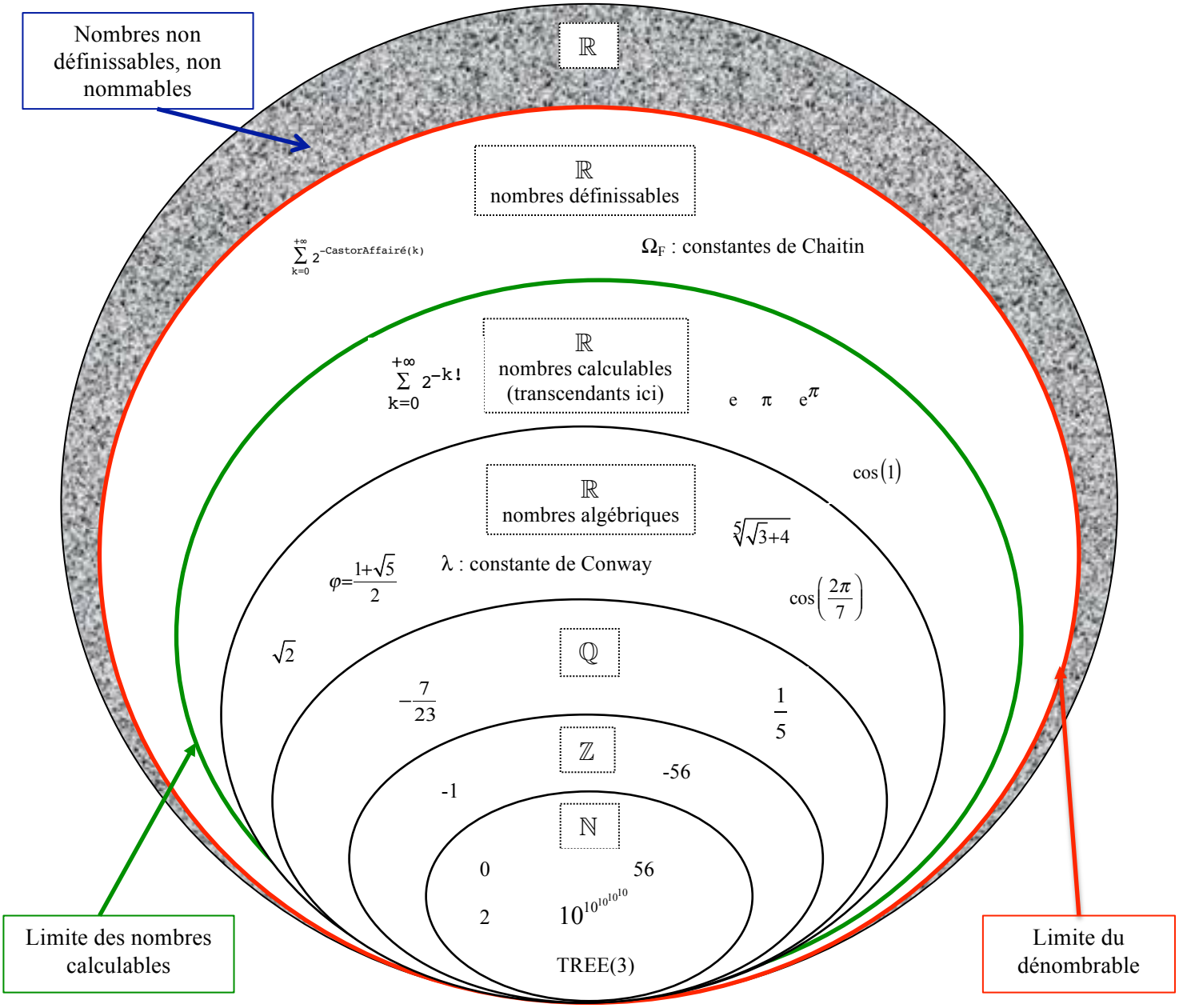
Remarques :

- Il est possible d'écrire les fonctions `exec` et `eval` en Python, c'est-à-dire que l'on peut écrire un interpréteur Python en Python !
- L'exemple le plus connu de programme manipulant un autre programme est celui des compilateurs ;
- mais un navigateur fait de même.

[5] L'instruction `pass` ne fait rien, elle est très pratique quand en cours de programmation on veut faire tourner un code sans avoir tout écrit.
[6] Toujours la poésie magique du vocabulaire allemand

5. Complement : lien entre mathematiques et calculabilite

Le schéma ci-dessous donne les nombres calculables (légende en page suivante).
Pour en comprendre les bases, il faut savoir qu'il y a plusieurs tailles d'infinis, comme dit au paragraphe 2. Le plus petit infini est celui de $\mathbb{N}$ (entiers naturels), on dit que son cardinal –son nombre d'éléments- est $\aleph_0$ (aleph 0). $\mathbb{Z}$ (entiers relatifs), et $\mathbb{Q}$ (rationnels) ont le même cardinal $\aleph_0$ (ce qui est plutôt surprenant), par contre $\mathbb{R}$ est plus grand, de cardinal $c = 2^{\aleph_0}$ (cf. note [7]). Un ensemble de cardinal $\aleph_0$ est dit dénombrable : on peut numéroter ses éléments 0, 1, 2, … Entre autres, toutes les formulations que l'on peut donner avec le langage donnent un ensemble dénombrable, puisqu'il suffit de les coder (a -> 0, b -> 1 etc.) pour les obtenir. En conséquence de tout ce qui précède, $\mathbb{R}$ n'est pas dénombrable et il est impossible de nommer tous ses éléments.



[7] Pour les curieux : cours disponible auprès du professeur sur demande. Sur internet, voir sur Wikipedia les articles « nombres rationnels », « nombres réels », partie propriétés, et « argument diagonal de Cantor » pour la démonstration du cardinal.
Par ailleurs, un théorème mathématique indécidable –dont on ne peut démontrer qu'il est vrai ou faux- dans la théorie usuelle des ensembles est qu'il existe un cardinal infini entre $\aleph_0$ (entiers naturels) et $2^{\aleph_0}$ (nombres réels)

Explications supplémentaires :

- Les nombres algébriques sont solutions d'une équation polynomiale à coefficients entiers (ou rationnels, c'est équivalent en multipliant par le ppcm des dénominateurs).
- Les nombres réels qui ne sont pas algébriques sont transcendants. Donc les nombres des deux plus grands ensembles sont également transcendants (nombres définissables et non définissables).
- L'ensemble des réels non définissables est donc infiniment plus grand que celui des réels définissables, et a fortiori des réels calculables. Ces deux derniers ont un poids (une « mesure » du point de vue mathématique) de 0 dans $\mathbb{R}$.
- TREE(3) est le plus grand nombre utilisé dans une démonstration mathématique (à ma connaissance, et sachant que TREE(4) est plus grand etc.). Si par exemple vous connaissez la notation des flèches de Knuth, ou bien la fonction d'Ackermann, les nombres obtenus par ces méthodes sont ridiculement petits par rapport aux nombres TREE. Ces nombres sont liés à la théorie des arbres, et étonnamment on a TREE(1) = 1, TREE(2) = 3, et après la croissance est phénoménalement rapide.
- La constante λ de Conway est liée à la suite « look and say », de premiers termes 1, 11, 21, 1221 etc. Elle est racine d'un polynôme de degré 71 et donne la vitesse de croissance asymptotique entre un terme et le suivant (cf. référence sur la page sujets de grand oral).
- Castor affairé : cf. ci-dessus paragraphe 2.
- Constantes de Chaitin : premier exemple de nombre réellement aléatoire non calculable. Elles sont définies par la probabilité qu'un programme, ayant une fin de code clairement identifiée, généré aléatoirement, finisse par s'arrêter. C'est en fait une manière de coder le problème de l'arrêt.

Exercices

A part l'exercice 1 et l'exercice tiré du bac, ces exercices ne portent que sur des notions hors-programme !

1. Reprendre la fonction `universel`, et donner une chaîne de caractères permettant de calculer le nombre d'ordres possibles des n premiers arrivés dans une course comportant N personnes.
2. Donner la table de transition d'une machine de Turing qui teste si un entier écrit en binaire est pair.
3. Donner la table de transition d'une machine de Turing qui multiplie un entier écrit en binaire par 2.
4. L'entrée d'une machine de Turing est donnée par une suite de 1, une case vide et un nombre binaire. Donner la table de transition d'une machine de Turing qui multiplie l'entier écrit en binaire par 2^n, où n est le nombre de 1.
5. Donner la table de transitions de la machine de Turing renvoyant Vrai si un nombre est de la forme 0...01...1, avec autant de 0 et de 1 que l'on souhaite, et Faux sinon. Le nombre de 0 et de 1 est le même.
6. Donner la table de transitions de la machine de Turing renvoyant Vrai si l'entrée est un palindrome.

Plus difficiles :

7. Donner la table de transition d'une machine de Turing qui détermine si une suite de 1 est de longueur paire ou impaire.
8. Donner la table de transition d'une machine de Turing qui double la longueur d'une suite de 1.
9. Donner la table de transition d'une machine de Turing qui ajoute deux nombres de deux bits.
10. Donner la table de transition d'une machine de Turing qui multiplie un nombre en binaire par 11 (3 en décimal)

Un exercice du bac 2024

Dans cet exercice, on dira qu'un appel `f(x)`, où `f` est une fonction Python prenant un argument et `x` est une valeur, **termine**[8] lorsque l'évaluation de `f(x)` renvoie toujours une valeur au bout d'un nombre fini d'étapes. A l'opposé, un tel appel ne termine pas s'il est possible qu'il effectue des instructions à l'infini.

Partie A : boucle **`while`**

Commençons par un premier exemple, avec une fonction prenant un entier en argument et utilisant une boucle **`while`**.

[8] L'erreur de français préférée de l'informatique théorique. On écrit « se termine » et non termine, qui ne veut rien dire.

```
1 def f1(n):
2     i = n
3     while i != 10:
4         i = i + 1
5     return i
```

1. Sur votre copie, donner les valeurs successives de la variable `i` lors de l'exécution de `f1(7)`, et indiquer si `f1(7)` termine.
2. Indiquer si l'appel `f1(-2)` se termine. Si oui, indiquer la valeur renvoyée.
3. Sur votre copie, donner les 5 premières valeurs prises par la variable `i` lors de l'exécution de `f1(12)`, et indiquer si l'appel `f1(12)` va terminer ou non.
4. Préciser pour quels entiers `n` l'appel `f1(n)` se termine.

Partie B : fonction récursive

Prenons maintenant un deuxième exemple, avec une fonction récursive (prenant elle aussi un entier en argument).

```
1 def f2(n):
2     if n == 0:
3         return 0
4     else:
5         return n + f2(n-2)
```

5. L'appel `f2(4)` termine-t-il ? Si oui, indiquer la valeur renvoyée par `f2(4)` ; sinon, justifier brièvement.
6. L'appel `f2(5)` termine-t-il ? Si oui, indiquer la valeur renvoyée par `f2(5)` ; sinon, justifier brièvement.
7. Déterminer l'ensemble des entiers naturels `n` pour lesquels l'appel `f2(n)` termine.
8. Écrire une fonction Python `infini`, récursive, telle que l'appel `infini(n)` ne termine pour aucun entier `n`.

Partie C : le problème de l'arrêt

On se demande maintenant s'il est possible d'écrire une fonction `arret` qui prend en arguments une chaîne de caractères `code_f` contenant le code d'une fonction `f` et un argument `x` de `f`, et tel que `arret(code_f, x)` renvoie `True` si l'appel `f(x)` va terminer et `False` sinon.

Dans la suite de cet exercice, on suppose disposer d'une telle fonction `arret` et on implémente la fonction suivante, utilisant cette fonction `arret`, ainsi que la fonction `infini` de la question précédente dont l'appel `infini(n)` ne termine jamais quelle que soit la valeur de `n`.

```
1 def paradoxe(x):
2 if arret(x, x):
3 infini(42)
4 else:
5 return 0
```

De même, on suppose disposer d'une variable `code_paradoxe` contenant le code de la fonction `paradoxe` sous la forme d'une chaîne de caractères, et on s'intéresse à l'appel `paradoxe(code_paradoxe)`.

Cet appel de fonction commence par effectuer le test `arret(code_paradoxe, code_paradoxe)` dans le `if` de la ligne 2.

9. Dans le cas où `arret(code_paradoxe, code_paradoxe)` renvoie `True`, préciser la prochaine instruction à être exécutée. Dans ce cas, expliquer si l'appel `paradoxe(code_paradoxe)` termine.
10. Dans le cas où `arret(code_paradoxe, code_paradoxe)` renvoie `False`, préciser la prochaine instruction à être exécutée. Dans ce cas, expliquer si l'appel `paradoxe(code_paradoxe)` termine.
11. En déduire qu'une telle fonction `arret` ne peut exister.