Calculabilité, décidabilité, problème de l'arrêt

1. Définition et idée générale

Une **fonction calculable** est une fonction pour laquelle il existe un algorithme qui donne le résultat *pour toute entrée* de son *domaine de définition* (comme en maths, une fonction informatique n'est définie que pour certaines entrées : une fonction travaillant sur les entiers ne devrait *jamais* recevoir une liste en paramètre).

Un **problème décidable** est un problème auquel on peut répondre par oui ou non, et pour lequel il existe un algorithme qui répond *toujours* correctement.

> ⚠️ **Attention !**
>
> Un problème peut être formulé clairement sans pour autant être calculable ou décidable. Le problème de l'arrêt en est l'exemple central.

2. Le problème de l'arrêt

Question fondamentale : existe-t-il un programme `HALT` qui, lorsqu'on lui donne en entrée le code source d'un *autre* programme `P` et des *arguments* `args` *pour cet autre programme*, répond `True` lorsque `P(args)` se termine, ou `False` lorsque `P(args)` ne se termine pas ? C'est la version informatique d'une question simple en apparence, mais qui n'admet pas de solution algorithmique générale.

> 💭 **Remarque : signification précise de « se terminer »** ▼
>
> Cela signifie ici s'arrêter, *quelle qu'en soit la raison* — résultat correct ou non, erreur ou exception. *Seule la boucle infinie* correspond à « ne pas se terminer ». Quand on parle de terminaison, on ne juge pas la *qualité de la terminaison*, mais *seulement son existence*.

Réponse : le problème de l'arrêt est **INDÉCIDABLE**. Il **n**'existe **pas** d'algorithme qui dise, pour tout programme et toute entrée, si ce programme s'arrêtera. Cette limite est importante parce qu'elle montre qu'il existe des questions mathématiques et informatiques auxquelles *aucun* algorithme ne peut répondre à tous les coups.

> ✅ **La logique de la preuve** ▼
>
> La connaître n'est pas nécessairement indispensable.
>
> Supposons qu'il existe un tel programme `HALT`. Cela signifie que :

- si `P(args)` se termine, `HALT(P, args)` répond `True` ;
- sinon, `HALT(P, args)` répond `False`.

On construit alors le programme suivant, nommé `absurde` :

```
def absurde(x):                 # x est une chaîne de caractères représentant un programme
    if HALT(x, x):              # HALT(x, x) renvoie True si x(x) termine, False sinon
        while True: pass        # Lance une boucle infinie -> NE termine PAS
    # Sinon, le programme se termine (implicitement, par retour normal)
```

Ce programme `absurde` prend un argument `x` en entrée (une chaîne de caractères représentant un programme) :

- si `x` termine, alors `absurde(x)` ne termine pas, car une boucle infinie est lancée ;
- si `x` **ne** termine **pas**, alors `absurde(x)` termine.

Examinons alors les deux seules possibilités sur l'exécution `absurde(absurde)` — oui, on passe à `absurde` son *propre code source* en argument (on assimile le programme à son code source) :

- si `absurde(absurde)` se termine, alors `HALT` répond `True`, et `absurde` rentre dans une boucle infinie… donc `absurde(absurde)` ne termine pas : contradiction !
- si `absurde(absurde)` ne se termine pas, alors `HALT` répond `False`… et `absurde(absurde)` se termine : contradiction encore !

Ces deux cas sont exhaustifs (pour toute exécution, soit elle termine, soit elle ne termine pas), et dans les deux cas on aboutit à une contradiction. On a donc une contradiction systématique, ce qui est… absurde, et permet de conclure que le programme `HALT` n'existe pas.

Remarque importante : un point mérite d'être souligné, car il est rarement explicité. Le programme `absurde` n'est pas un programme quelconque — il est conçu *exprès* pour prendre en entrée une chaîne de caractères représentant un programme. C'est une contrainte délibérée — et c'est précisément elle qui rendra la contradiction possible ! Or `HALT` lui-même repose déjà sur cette convention : il reçoit un code source en premier argument. Passer un code source à `absurde` est donc parfaitement admissible — et passer à `absurde` *son propre code source* l'est tout autant. C'est *précisément ce choix de conception* qui rend l'auto-application `absurde(absurde)` licite, et c'est là que la contradiction surgit.

3. Liens avec d'autres notions

- La **récursivité** permet de construire des programmes qui peuvent s'arrêter ou boucler.
- La **complexité** mesure le coût d'un algorithme, mais ne dit pas si un problème est calculable.
- Le **problème de l'arrêt** éclaire la frontière entre ce qu'un algorithme peut et ne peut pas faire.