

Calculabilité et décidabilité

Fiche de synthèse

① UN PROGRAMME EST UNE DONNÉE

Le code d'un programme est un texte : il peut être lu, transmis, analysé ou exécuté par un autre programme. L'interpréteur `python3` exécute le fichier `moyenne.py` ; un éditeur colore ses mots-clés ; un antivirus examine les fichiers exécutables ; le correcteur de Cypitale exécute un programme sur des tests.

```
def carre(x):
    return x * x

def verifier(fonction, entree, attendu):
    return fonction(entree) == attendu

verifier(carre, 4, 16)    # True
```

La fonction `carre` est transmise à `verifier`, qui l'exécute. Un programme peut même recevoir son propre code comme donnée.

② LE PROBLÈME DE L'ARRÊT

Certains programmes ne s'arrêtent pas : avec `dette = 10` et un remboursement de 3 par mois, la condition `dette != 0` reste toujours vraie. Pour d'autres, personne ne sait : on ignore si la boucle de la suite de Syracuse s'arrête pour tout entier de départ.

Théorème 1 (Arrêt, Turing, 1936) Il n'existe aucun programme capable de déterminer, pour tout programme et toute donnée, si ce programme s'arrête sur cette donnée. Le problème de l'arrêt est **indécidable**.

✂ **Démonstration par l'absurde.** Supposons qu'il existe une fonction `arret(programme, donnee)` qui renvoie `True` si l'appel `programme(donnee)` s'arrête, `False` sinon. On écrit :

```
def contraire(programme):
    if arret(programme, programme):
        while True:
            pass
    else:
        return "terminé"
```

Étudions l'appel `contraire(contraire)`.

- Si `arret` répond `True`, l'appel entre dans une boucle infinie : il ne s'arrête pas.
- Si `arret` répond `False`, l'appel renvoie `"terminé"` : il s'arrête.

Dans les deux cas, `arret` se trompe : la fonction `arret` ne peut pas exister.

Le théorème n'interdit pas de savoir si un programme particulier s'arrête : il affirme qu'aucune méthode automatique unique ne répond correctement pour tous les programmes. Il répond au problème de la décision posé par Hilbert en 1928.

③ LE THÉORÈME DE RICE

Théorème 2 (Rice, 1953) Toute propriété non triviale du **comportement** d'un programme est indécidable.

Une propriété porte sur le comportement quand elle concerne ce que fait le programme à l'exécution ; elle est non triviale quand certains programmes la vérifient et d'autres non. Aucun programme ne peut donc décider dans tous les cas si une fonction de tri trie toute liste, si un fichier est un virus, ou si deux fonctions renvoient toujours le même résultat. Une question sur le **texte** du programme reste décidable : « ce code contient-il une boucle `while` ? ». Les analyseurs de code et les antivirus ne donnent que des approximations.

④ CALCULABILITÉ

Définition 1 (Fonction calculable) Une fonction f est **calculable** s'il existe un algorithme qui, pour chaque entrée x , s'arrête après un nombre fini d'opérations élémentaires en donnant $f(x)$.

Le PGCD, le n -ième nombre premier ou le tri d'une liste sont calculables ; la fonction qui répond au problème de l'arrêt ne l'est pas. Il y a bien plus de fonctions que de programmes : la plupart des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ne sont pas calculables.

Définition 2 (Machine de Turing) Une **machine de Turing** comporte un ruban infini découpé en cases, une tête qui lit et écrit un symbole puis se déplace d'une case, un état courant pris dans un ensemble fini, et une table de transitions : pour chaque état et chaque symbole lu, elle donne le symbole à écrire, le déplacement et le nouvel état.

0	1	1	1	-	-
---	---	---	---	---	---

↑
tête, état q_0

La machine qui inverse les bits remplace chaque 0 par 1 et chaque 1 par 0 en avançant vers la droite, et s'arrête sur la première case vide. Partie du ruban 1011, elle a déjà inversé deux cases; elle finira sur 0100.

- Une **machine universelle** lit sur son ruban la description d'une autre machine et de sa donnée, puis la simule; un ordinateur exécute de même des programmes rangés en mémoire.
- **Thèse de Church-Turing** : les fonctions calculables par un algorithme sont exactement celles que calcule une machine de Turing. C'est une thèse, non un théorème : la notion d'algorithme n'a pas de définition mathématique.
- Un langage est **Turing-complet** s'il peut simuler toute machine de Turing : Python, C, JavaScript ou Scratch le sont, HTML seul ne l'est pas. Tous ces langages calculent les mêmes fonctions.

⑤ P, NP ET NP-COMPLET (HORS PROGRAMME)

- La classe **P** regroupe les problèmes résolus par un algorithme de complexité polynomiale, majorée par n^k pour un k fixé : trier une liste, chercher dans une liste triée, plus court chemin avec Dijkstra, test de primalité.
- La classe **NP**, pour non déterministe polynomial, regroupe les problèmes dont une solution proposée se **vérifie** en temps polynomial. NP ne signifie pas « non polynomial ».
- Exemples de NP : trouver les facteurs de 3 127 demande des essais, vérifier que $53 \times 59 = 3\,127$ une seule multiplication; colorier un graphe, organiser une tournée, remplir un sac à dos 0/1.
- $P \subset NP$: si l'on sait résoudre un problème, on vérifie une solution en le résolvant.
- Un problème **NP-complet** est un problème de NP auquel tout problème de NP se ramène en temps polynomial : SAT, sac à dos 0/1, voyageur de commerce, coloration avec trois couleurs. En résoudre un seul en temps polynomial prouverait que $P = NP$.

La question « $P = NP$? » est l'un des sept problèmes du prix du millénaire, doté d'un million de dollars.

📌 À retenir.

- Un programme est une donnée que d'autres programmes peuvent lire et exécuter.
- Le problème de l'arrêt est indécidable; la preuve par l'absurde utilise `contraire(contraire)`.
- Théorème de Rice : toute propriété non triviale du comportement d'un programme est indécidable.
- Une fonction est calculable si un algorithme la calcule en un nombre fini d'étapes; tous les langages Turing-complets calculent les mêmes fonctions.