Calculabilité et décidabilité

Cours – Terminale NSI

Remarque (Prérequis et objectifs). **Prérequis :** la récursivité, la notion de fonction comme donnée.
À l'issue de ce chapitre, vous saurez :

- expliquer pourquoi un programme peut être le paramètre d'un autre programme ;
- démontrer par l'absurde que le problème de l'arrêt est indécidable ;
- énoncer le théorème de Rice et donner des exemples de questions indécidables ;
- définir une fonction calculable, décrire une machine de Turing et expliquer ce qu'est un langage Turing-complet ;
- en complément hors programme, distinguer machine déterministe et machine non-déterministe ;
- en complément hors programme, distinguer les classes P, NP et NP-complet et donner des exemples pour chacune.

1 Un programme est une donnée

1.1 Le code source est un texte

Voici une fonction qui calcule la moyenne d'une liste de notes :

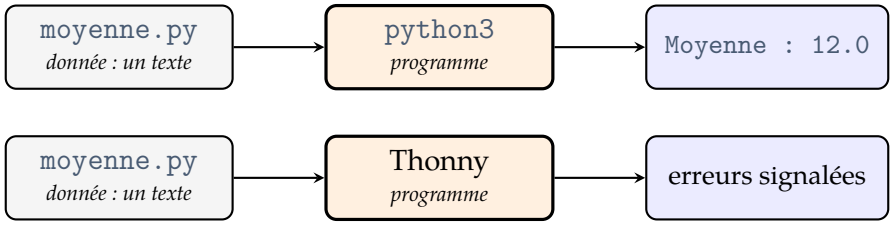
```
def moyenne(notes):
    total = 0
    for note in notes:
        total = total + note
    return total / len(notes)

print(f"Moyenne : {moyenne([12, 15, 9])}")
```

```
Moyenne : 12.0
```

La fonction `moyenne` reçoit une **donnée**, la liste `[12, 15, 9]`, et renvoie un nombre. Mais le code de `moyenne` est lui-même enregistré dans un fichier, par exemple `moyenne.py`, c'est-à-dire une suite de caractères. Ce texte est à son tour la donnée d'autres programmes :

- l'**interpréteur** `python3` lit le fichier `moyenne.py` et exécute ses instructions une à une ;
- l'éditeur **Thonny** colore les mots-clés et signale les erreurs de syntaxe avant toute exécution ;
- un **antivirus** examine les fichiers exécutables pour y repérer des suites d'instructions suspectes ;
- le correcteur automatique de **Capytale** exécute votre code sur des jeux de tests et compare les résultats aux réponses attendues.



1.2 Une fonction qui reçoit une fonction

En Python, une fonction peut recevoir une autre fonction en paramètre. La fonction `verifier` ci-dessous joue le rôle d'un petit correcteur automatique : elle exécute la fonction qu'on lui donne sur une entrée et compare le résultat à la réponse attendue.

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

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

print(f"Test 1 : {verifier(carre, 4, 16)}")
print(f"Test 2 : {verifier(carre, 3, 6)}")
```

```
Test 1 : True
Test 2 : False
```

Dans l'appel `verifier(carre, 4, 16)`, la fonction `carre` n'est pas exécutée par nous : elle est **transmise** à `verifier`, qui décide quand l'exécuter.

1.3 Une fonction qui lit un code source

On peut aussi analyser un programme sans l'exécuter, en lisant son texte. La fonction `nombre_de_boucles` compte les lignes d'un code source qui commencent une boucle `for` ou `while` :

```
def nombre_de_boucles(source):
    nombre = 0
    for ligne in source.split("\n"):
        debut = ligne.strip()
        if debut.startswith("for ") or debut.startswith("while "):
            nombre = nombre + 1
    return nombre

code = """
def moyenne(notes):
    total = 0
    for note in notes:
        total = total + note
    return total / len(notes)
"""
print(f"Boucles : {nombre_de_boucles(code)}")
```

```
Boucles : 1
```

Rien n'empêche d'ailleurs de donner à `nombre_de_boucles` son propre code source : un programme peut prendre **lui-même** comme donnée.

Propriété 1 **(Un programme est une donnée)** Un programme est un texte, donc une donnée comme une autre. Il peut être lu, transmis, analysé ou exécuté par un autre programme, et même par lui-même. C'est le principe de l'**interprétation** (Python, JavaScript) et de la **compilation** (C, Rust), et c'est aussi l'idée qui permet de démontrer les **limites** du calcul dans la suite du chapitre.

② Le problème de l'arrêt

2.1 Un programme qui ne s'arrête jamais

Une personne rembourse une dette de 3 € par mois. La fonction suivante compte le nombre de mois nécessaires :

```
def rembourser(dette):
    mois = 0
    while dette != 0:
        dette = dette - 3
        mois = mois + 1
    return mois

print(f"Mois nécessaires : {rembourser(12)}")
```

```
Mois nécessaires : 4
```

Avec une dette de 12 €, la variable `dette` prend les valeurs 12, 9, 6, 3 et 0 : la boucle s'arrête après 4 mois.

Avec une dette de 10 €, la variable `dette` prend les valeurs 10, 7, 4, 1, puis −2, −5, −8… Elle ne vaut **jamais** 0 : l'appel `rembourser(10)` ne s'arrête pas, c'est une **boucle infinie**. Ici, une lecture attentive du code suffit à le prévoir, et la condition `dette > 0` corrige l'erreur.

2.2 Un programme dont personne ne sait s'il s'arrête

La fonction suivante applique la règle de la **suite de Syracuse** : si `n` est pair, on le divise par 2 ; sinon, on le remplace par `3 * n + 1`. Elle compte les étapes nécessaires pour atteindre 1.

```
def syracuse(n):
    etapes = 0
    while n != 1:
        if n % 2 == 0:
            n = n // 2
        else:
            n = 3 * n + 1
        etapes = etapes + 1
    return etapes

print(f"Étapes pour 6 : {syracuse(6)}")
print(f"Étapes pour 27 : {syracuse(27)}")
```

```
Étapes pour 6 : 8
Étapes pour 27 : 111
```

Partant de 6, on obtient 6, 3, 10, 5, 16, 8, 4, 2, 1. Partant de 27, la suite monte jusqu'à 9 232 avant de redescendre à 1. Les ordinateurs ont vérifié que `syracuse(n)` s'arrête pour tous les entiers jusqu'à plus de 10^{20}, mais **personne n'a jamais démontré** qu'il s'arrête pour tout entier `n` : c'est la **conjecture de Syracuse**, ouverte depuis les années 1930.

Cinq lignes de code suffisent donc à poser une question que les mathématiciens ne savent pas résoudre. D'où la question de ce chapitre : un programme pourrait-il répondre à notre place, en lisant le code d'un autre programme ?

2.3 Hypothèse : il existe un programme `arret`

Supposons qu'il existe une fonction `arret(programme, donnee)` qui, pour **tout** programme et **toute** donnée :

— renvoie `True` si l'appel `programme(donnee)` finit par s'arrêter ;

— renvoie `False` si l'appel `programme(donnee)` ne s'arrête jamais.

On aurait par exemple `arret(rembourser, 12) == True` et `arret(rembourser, 10) == False`. Et `arret` réglerait au passage la conjecture de Syracuse.

Nous allons montrer, par un **raisonnement par l'absurde**, qu'une telle fonction ne peut pas exister : en la supposant possible, on aboutit à une contradiction.

2.4 Le programme `contraire`

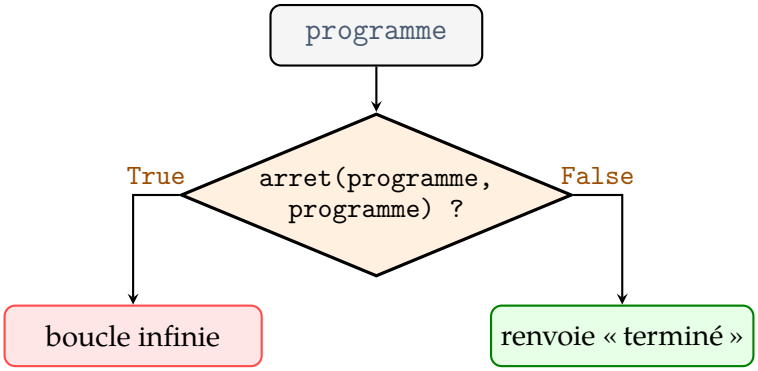
Imaginez une voyante qui prétend prédire, sans jamais se tromper, si vous viendrez au lycée demain. Il suffit de lui demander sa prédiction, puis de faire l'inverse : elle se trompe forcément. La démonstration de Turing repose sur la même idée. Avec la fonction `arret`, on écrit le programme `contraire` :

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

Le programme `contraire` reçoit un programme, demande à `arret` si ce programme s'arrête quand on lui donne **son propre code** comme donnée, puis fait l'inverse de la prédiction :

— si `arret` répond `True` (« il s'arrête »), `contraire` entre dans une boucle infinie ;

— si `arret` répond `False` (« il ne s'arrête pas »), `contraire` s'arrête aussitôt.



2.5 La contradiction : `contraire(contraire)`

Puisqu'un programme est une donnée, on peut donner à `contraire` son propre code : que se passe-t-il lors de l'appel `contraire(contraire)` ? Tout dépend de la réponse de `arret(contraire, contraire)`.

Réponse de `arret`	**Ce que `arret` affirme**	**Ce que fait réellement `contraire(contraire)`**
`True`	`contraire(contraire)` s'arrête	il entre dans la boucle infinie et ne s'arrête jamais
`False`	`contraire(contraire)` ne s'arrête jamais	il renvoie `"terminé"`, donc il s'arrête

Dans les deux cas, `arret` se trompe sur `contraire(contraire)`. Or on avait supposé que `arret` ne se trompe jamais : l'hypothèse de départ est fausse. **La fonction `arret` ne peut pas exister.**

2.6 Le théorème de l'arrêt

Théorème 1 **(Théorème de l'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. On dit que le problème de l'arrêt est **indécidable**.

Ce théorème ne dit pas qu'on ne peut jamais savoir si un programme s'arrête : pour `rembourser`, la réponse est évidente. Il dit qu'**aucune méthode automatique unique** ne peut répondre correctement pour **tous** les programmes.

Le résultat est démontré en 1936 par le mathématicien britannique Alan Turing, dans l'article *On Computable Numbers, with an Application to the Entscheidungsproblem*. Pour donner un sens précis au mot « programme », Turing y décrit une machine abstraite, appelée depuis **machine de Turing** (section 4.2). La même année, l'Américain Alonzo Church obtient un résultat équivalent avec un autre formalisme, le lambda-calcul.

Ces travaux répondent à une question posée en 1928 par le mathématicien allemand David Hilbert, le **problème de la décision** (*Entscheidungsproblem*) : existe-t-il une méthode automatique qui décide, pour tout énoncé mathématique, s'il est vrai ou faux ? Une telle méthode réglerait par exemple la **conjecture de Goldbach** (« tout entier pair supérieur à 2 est la somme de deux nombres premiers »), toujours ouverte : il suffirait d'écrire un programme qui essaie 4, 6, 8, 10… et s'arrête dès qu'il trouve un entier pair qui n'est pas une telle somme, puis de demander à `arret` si ce programme s'arrête. Turing et Church montrent que la réponse à la question de Hilbert est **non**.

③ Le théorème de Rice

3.1 Énoncé

En 1953, le mathématicien américain Henry Gordon Rice généralise le théorème de l'arrêt.

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

Une propriété du comportement porte sur ce que le programme **fait** quand on l'exécute (ce qu'il renvoie, ce qu'il affiche, s'il s'arrête). Elle est non triviale si certains programmes la vérifient et d'autres non. Pour démontrer le théorème, on montre qu'un programme capable de décider une telle propriété permettrait d'écrire la fonction `arret`, qui n'existe pas : on dit que le problème de l'arrêt se **réduit** à cette propriété.

3.2 Questions indécidables et questions décidables

Les questions suivantes portent sur le comportement d'un programme : aucun programme ne peut y répondre correctement dans tous les cas.

— « Cette fonction de tri renvoie-t-elle une liste triée pour **toute** liste ? »

— « Ce programme affiche-t-il un jour le mot de passe de l'utilisateur ? »

— « Ce fichier exécutable est-il un virus ? » (Fred Cohen l'a démontré dès 1987 : aucun antivirus ne peut détecter tous les virus sans jamais se tromper.)

— « Les fonctions `f` et `g` renvoient-elles le même résultat pour toute entrée ? »

En revanche, une question sur le **texte** du programme, et non sur son comportement, peut être décidable : « ce code contient-il une boucle `while` ? » se décide en lisant le code, comme le fait la fonction `nombre_de_boucles` de la section 1.

Remarque 1 (Conséquence pratique). Il est impossible d'écrire un outil qui détecte, dans tout programme, toutes les boucles infinies, tous les bugs ou toutes les failles de sécurité. Les outils réels (Thonny, les analyseurs comme pylint, les antivirus) donnent des **approximations** : ils laissent passer certaines erreurs ou signalent de fausses alertes.

④ Calculabilité

4.1 Fonction calculable

Le problème de l'arrêt est indécidable parce que la fonction `arret` ne peut être programmée : on dit qu'elle n'est pas **calculable**.

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'étapes en donnant $f(x)$.

Chaque étape d'un algorithme est une **opération élémentaire**, si simple qu'elle ne se décompose pas : ajouter 1 à un nombre, comparer deux symboles, lire ou écrire un symbole en mémoire, choisir l'instruction suivante selon le résultat d'une comparaison. Une multiplication d'entiers, par exemple, se ramène à une suite d'additions.

Sont calculables, par exemple :

— $f(a, b)$ = le PGCD de a et b, par l'algorithme d'Euclide ;

— $f(n)$ = le n-ième nombre premier, en testant les entiers un par un ;

— $f(\text{L})$ = la liste L triée par ordre croissant.

N'est pas calculable : la fonction qui associe à un programme et à une donnée la réponse à la question « ce programme s'arrête-t-il sur cette donnée ? ».

4.1.1 Plus de fonctions que de programmes.

En réalité, **la plupart** des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ne sont pas calculables. Un programme est un texte de longueur finie : on peut ranger tous les programmes dans une liste infinie (d'abord ceux d'un caractère, puis ceux de deux caractères, etc.), on dit que l'ensemble des programmes est **dénombrable**. L'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, lui, ne l'est pas (argument diagonal de Cantor). Il y a donc strictement plus de fonctions que de programmes, et certaines fonctions ne sont calculées par aucun programme. Les fonctions qui servent en pratique sont calculables, puisqu'on les programme.

4.2 La machine de Turing

Définition 2 **(Machine de Turing)** Une **machine de Turing** comporte :

— un **ruban** infini découpé en cases, chacune contenant un symbole (`0`, `1` ou une case vide notée `_`) ;

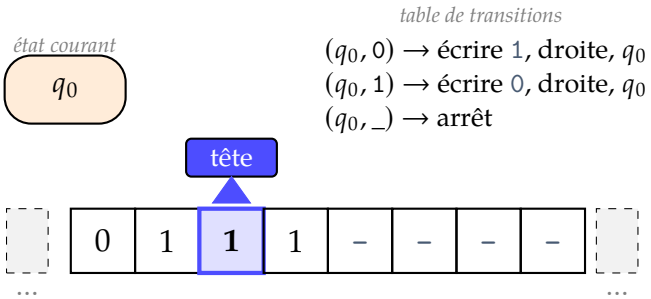
— une **tête** de lecture et d'écriture, placée sur une case, qui lit le symbole, peut le remplacer, puis se déplace d'une case vers la gauche ou vers la droite ;

— un **état** courant, choisi dans un ensemble fini d'états ;

— une **table de transitions** qui, pour chaque état et chaque symbole lu, donne le symbole à écrire, le déplacement et le nouvel état.

La machine s'arrête lorsqu'elle atteint un état final.

Voici une machine qui **inverse les bits** d'un mot binaire : dans l'état q_0, elle remplace chaque `0` par `1` et chaque `1` par `0` en avançant vers la droite, puis s'arrête sur la première case vide. La figure la montre après deux transitions, en partant du ruban `1011`.



Les deux premières cases ont déjà été inversées (`1` est devenu `0`, puis `0` est devenu `1`) ; la tête lit maintenant le troisième symbole. Le programme suivant simule cette machine : la table de transitions est un dictionnaire dont les clés sont les couples (état, symbole lu).

```
def executer(table, ruban, etat):
    cases = list(ruban) + ["_"]
```

```
    position = 0
    while etat != "fin":
        symbole = cases[position]
        ecrit, deplacement, etat = table[(etat, symbole)]
        cases[position] = ecrit
        position = position + deplacement
        if position == len(cases):
            cases.append("_")
    return "".join(cases).strip("_")

inverser = {
    ("q0", "0"): ("1", 1, "q0"),
    ("q0", "1"): ("0", 1, "q0"),
    ("q0", "_"): ("_", 0, "fin"),
}
print(f"Résultat : {executer(inverser, '1011', 'q0')}")
```

```
Résultat : 0100
```

Avec une autre table, le même programme `executer` simule une autre machine : additionner, comparer, trier… Aussi rudimentaire soit-il, ce modèle permet d'exécuter **n'importe quel algorithme**.

Théorème 3 **(Machine de Turing universelle)** Il existe une **machine universelle** : une machine de Turing unique, à table fixée, qui lit sur son ruban la description d'une autre machine et de sa donnée, puis simule cette machine.

Notre fonction `executer` en donne l'idée : la table `inverser` est une donnée qu'on lui transmet. Un ordinateur fonctionne de la même manière : son processeur exécute des programmes rangés en mémoire comme des données. C'est l'idée de la section 1.

4.3 Thèse de Church-Turing

Propriété 2 **(Thèse de Church-Turing)** Les fonctions calculables par un algorithme sont exactement les fonctions calculables par une machine de Turing.

C'est une **thèse** et non un théorème : la notion intuitive d'« algorithme » n'a pas de définition mathématique qui permettrait de la démontrer. Mais tous les modèles de calcul inventés depuis 1936 (lambda-calcul, machines à registres, automates cellulaires, ordinateurs réels) calculent exactement les mêmes fonctions que la machine de Turing, ni plus ni moins.

4.4 Langages Turing-complets

Un langage de programmation est **Turing-complet** s'il permet de simuler n'importe quelle machine de Turing. Il lui suffit pour cela de disposer d'instructions conditionnelles, de boucles ou de récursivité, et d'une mémoire aussi grande que nécessaire. Python, C, Java, JavaScript et Scratch sont Turing-complets ; le langage HTML seul ne l'est pas, car il décrit une page sans pouvoir répéter un calcul.

Conséquence : une fonction calculable dans un langage Turing-complet est calculable dans tous les autres. Le choix du langage change la facilité d'écriture et la vitesse d'exécution, jamais l'ensemble des problèmes que l'on peut résoudre.

⑤ Classes de complexité : P, NP et NP-complet (hors programme)

Parmi les fonctions calculables, certaines se calculent en quelques millisecondes, d'autres demanderaient des milliards d'années. La théorie de la **complexité** classe les problèmes selon la croissance du temps de calcul en fonction de la taille n de la donnée.

5.1 La classe P

Un algorithme est de **complexité polynomiale** si son nombre d'opérations est majoré par n^k pour un entier k fixé : les complexités logarithmique, linéaire, en $n \log n$, quadratique ou cubique sont polynomiales ; les complexités en 2^n ou en $n!$ ne le sont pas.

Définition 3 **(Classe P)** La **classe P** regroupe les problèmes qui admettent un algorithme de résolution de complexité polynomiale.

Sont dans P :

— trier une liste (tri fusion en $O(n \log n)$) ;

— chercher une valeur dans une liste triée (dichotomie en $O(\log n)$) ;

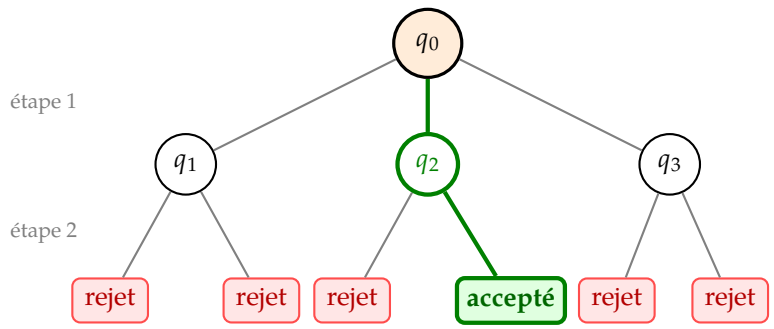
— calculer le plus court chemin entre deux villes d'un graphe routier (algorithme de Dijkstra) ;

— tester si un entier est premier : longtemps, aucun algorithme polynomial n'était connu, jusqu'à ce que trois chercheurs indiens, Agrawal, Kayal et Saxena, en publient un en 2002 (le test AKS).

5.2 La classe NP

Définition 4 **(Machine déterministe)** Une machine de Turing **déterministe** n'a, à chaque étape, qu'une seule transition possible : l'état courant et le symbole lu fixent entièrement l'action suivante. Les ordinateurs réels sont déterministes.

Définition 5 **(Machine non déterministe)** Une machine de Turing **non déterministe** peut avoir **plusieurs** transitions possibles pour un même état et un même symbole lu. Ses exécutions forment un arbre : elle **accepte** une entrée si au moins une branche mène à un état d'acceptation, et la rejette si toutes les branches échouent. Cette machine n'existe pas physiquement : c'est un outil théorique.



Définition 6 **(Classe NP)** La **classe NP** (pour *non déterministe polynomial*) regroupe les problèmes qu'une machine de Turing non déterministe résout en temps polynomial.

Remarque 2 (Attention). NP ne signifie **pas** « non polynomial ».

Une définition équivalente est plus parlante. La machine non déterministe peut suivre la branche qui « devine » une solution, puis vérifier cette solution. D'où :

Propriété 3 **(Caractérisation de NP)** Un problème est dans **NP** si l'on peut **vérifier** en temps polynomial qu'une solution proposée est correcte.

Trouver une solution peut être très long ; vérifier une solution qu'on vous donne est rapide. Par exemple :

— **Factoriser un entier.** Trouver deux facteurs de 3 127 demande des essais ; vérifier que $53 \times 59 = 3\,127$ demande une seule multiplication. Pour les nombres de plusieurs centaines de chiffres utilisés par le chiffrement RSA, aucun ordinateur ne sait trouver les facteurs en un temps raisonnable.

— **Construire le calendrier des épreuves du bac blanc.** Deux épreuves passées par un même élève ne doivent pas avoir lieu dans le même créneau, et l'on dispose de k créneaux. Vérifier un calendrier proposé consiste à examiner chaque paire d'épreuves en conflit ; en trouver un est un problème de **coloration de graphe**.

— **Organiser la tournée d'un livreur** (voyageur de commerce, version de décision : existe-t-il une tournée passant par toutes les adresses de longueur inférieure à L ?). Une tournée proposée se vérifie en additionnant ses distances.

— **Remplir un sac à dos** 0/1 (version de décision, voir ci-dessous). Un choix d'objets se vérifie en additionnant poids et valeurs.

— **Satisfaire une formule booléenne** (problème SAT) : une affectation des variables proposée se vérifie en évaluant la formule.

5.3 Le sac à dos : un piège classique

Le sac à dos est l'exemple parfait pour saisir la frontière entre P et NP, à condition de bien distinguer ses deux versions.

5.3.1 Sac à dos fractionnaire.

On peut prendre une fraction d'un objet. L'algorithme glouton qui trie les objets par rapport valeur/poids décroissant puis les prend dans cet ordre est **optimal**. Sa complexité est celle du tri : $O(n \log n)$. Le problème est **dans P**.

5.3.2 Sac à dos 0/1.

Chaque objet est pris en entier ou pas du tout. L'algorithme glouton s'exécute toujours en $O(n \log n)$, mais il **ne donne plus la solution optimale**.

Exemple 1 (Un contre-exemple). Sac de capacité 10, trois objets :

Objet	Poids	Valeur	Rapport
A	6	10	1,67
B	5	8	1,60
C	5	8	1,60

Glouton : il prend d'abord A (meilleur rapport). Il reste 4 unités de capacité, insuffisantes pour B ou C qui pèsent 5 chacun. Valeur totale obtenue : **10**.
Optimal : prendre B et C (poids total 10, valeur **16**). Le glouton a choisi l'objet au meilleur rapport, mais il a laissé de la place inutilisée.

5.3.3 *Résoudre le sac* 0/1 *correctement.*

— **Force brute** : tester les 2^n sous-ensembles, en $O(2^n)$.

— **Programmation dynamique** : $O(n \cdot W)$, dite *pseudo-polynomiale* : polynomiale en la *valeur* de W, mais comme W s'écrit avec $\log_2 W$ bits, elle est en réalité **exponentielle** en la taille de l'entrée.

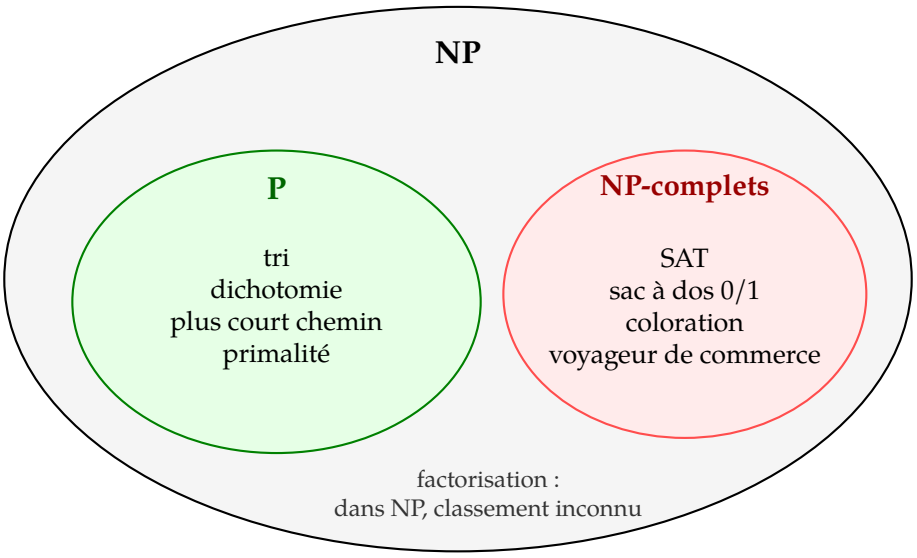
Aucun algorithme **polynomial** et **correct** n'est connu : le sac à dos 0/1 (en version de décision) est **NP-complet**.

Propriété 4 **(À retenir)** Un algorithme **rapide** existe (le glouton), mais il est **incorrect** pour la version 0/1. Un algorithme **correct** existe (la programmation dynamique), mais il n'est **pas réellement polynomial**. Cette tension entre rapidité et exactitude caractérise les problèmes NP-complets.

Pour les autres problèmes de la liste (factorisation, calendrier, tournée, SAT), aucun algorithme de résolution polynomial n'est connu à ce jour. Pour la factorisation, les meilleurs algorithmes connus sont plus rapides qu'un algorithme exponentiel, mais ils ne sont pas polynomiaux.

5.4 P ⊂ NP et les problèmes NP-complets

Un problème de P se résout en temps polynomial ; pour vérifier une solution proposée, il suffit de résoudre le problème et de comparer. Donc P ⊂ NP : tout problème de P est aussi dans NP.



La figure représente la situation **si** P ≠ NP, ce que croient la plupart des chercheurs sans l'avoir démontré.

Définition 7 **(Problème NP-complet)** Un problème de NP est **NP-complet** si tout problème de NP peut s'y ramener en temps polynomial.

Ce sont les problèmes les plus difficiles de NP : un algorithme polynomial pour **un seul** d'entre eux donnerait un algorithme polynomial pour **tous** les problèmes de NP, et prouverait que P = NP.

Le premier problème reconnu NP-complet est **SAT** (théorème de Cook-Levin, 1971). Pour montrer qu'un nouveau problème est NP-complet, on transforme en temps polynomial un problème déjà connu NP-complet (SAT par exemple) en ce nouveau problème : on dit qu'on effectue une **réduction**.

Sont NP-complets, par exemple : SAT et sa variante 3-SAT, le sac à dos 0/1 (version de décision), le voyageur de commerce (version de décision), la coloration d'un graphe avec 3 couleurs ou plus, le sudoku généralisé à une grille $n^2 \times n^2$.

Pour approfondir, la vidéo de David Louapre *Nos algorithmes pourraient-ils être BEAUCOUP plus rapides ? (P = NP ?)*, sur la chaîne Science étonnante, présente ces classes avec de nombreux exemples.

5.5 Une question ouverte

La question « P = NP ? » fait partie des sept problèmes du prix du millénaire choisis en 2000 par l'Institut Clay, qui offre un million de dollars pour sa solution. Une preuve que P = NP, assortie d'algorithmes efficaces, rendrait le chiffrement RSA inutilisable ; une preuve que P ≠ NP garantirait qu'aucun progrès des algorithmes ne rendra faciles les problèmes NP-complets.

⑥ Synthèse

Propriété 5 **(À retenir)**

— Un **programme** est un texte, donc une donnée : il peut être lu, analysé ou exécuté par un autre programme, et même par lui-même.

— Le **problème de l'arrêt** est **indécidable** (Turing, 1936) : aucun programme `arret` ne peut prédire, pour tout programme et toute donnée, si l'exécution s'arrête. La preuve construit un programme `contraire` qui fait l'inverse de la prédiction de `arret`.

— Le **théorème de Rice** (1953) étend l'indécidabilité à toute propriété non triviale du comportement d'un programme ; les questions sur le seul texte du programme peuvent rester décidables.

— Une fonction est **calculable** s'il existe un algorithme qui la calcule en un nombre fini d'étapes. La **machine de Turing** (ruban, tête, états, table de transitions) est le modèle de référence ; tous les langages **Turing-complets** calculent les mêmes fonctions.

HP La classe **P** regroupe les problèmes résolubles en temps polynomial ; la classe **NP**, ceux dont une solution se vérifie en temps polynomial, et P ⊂ NP.

HP Les problèmes **NP-complets** (SAT, sac à dos 0/1, coloration, voyageur de commerce) sont les plus difficiles de NP : en résoudre un en temps polynomial prouverait que P = NP.