Calculabilité & complexité

DIU Enseignement de l'informatique au lycée
Bloc 5 : Algorithmique avancée

Bruno Grenet – Université de Montpellier

Juin-juillet 2020

C'est quoi un *calcul* ?

Calculabilité

Contexte historique

Vor Allem aber möchte ich unter den zahlreichen Fragen, welche hinsichtlich der Axiome gestellt werden können, dies als das wichtigste Problem bezeichnen, *zu beweisen, daß dieselben untereinander widerspruchslos sind, d. h. daß man auf Grund derselben mittelst einer endlichen Anzahl von logischen Schlüssen niemals zu Resultaten gelangen kann, die miteinander in Widerspruch stehen.*

Eine Diophantische Gleichung mit irgend welchen Unbekannten und mit ganzen rationalen Zahlencoefficienten sei vorgelegt: *man soll ein Verfahren angeben, nach welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist.*

Contexte historique

Mais avant tout, parmi tant de questions soulevées par l'examen des axiomes, je regarde comme la plus importante celle-ci : *Démontrer que les axiomes ne sont pas contradictoires; c'est-à-dire démontrer qu'en se basant sur les axiomes l'on ne pourra jamais arriver à des résultats contradictoires au moyen d'un nombre fini de déductions logiques.*

On donne une équation de Diophante à un nombre quelconque d'inconnues et à coefficients entiers rationnels : *On demande de trouver une méthode par laquelle, au moyen d'un nombre fini d'opérations, on pourra distinguer si l'équation est résolable en nombres entiers rationnels.*

Contexte historique

- 23 problèmes de Hilbert pour le xx$^{\text{ème}}$ siècle (1900) :
  - 2$^{\text{ème}}$: Peut-on prouver la cohérence de l'arithmétique ?
  - 10$^{\text{ème}}$: Peut-on trouver les solutions d'une équation diophantienne *au moyen d'un nombre fini d'opérations* ?
- *Entscheidungsproblem* (Hilbert-Ackermann, 1928) : existe-t-il un algorithme pour décider si un énoncé mathématique est correct ?

Contexte historique

- 23 problèmes de Hilbert pour le xx$^{\text{ème}}$ siècle (1900) :
  - 2$^{\text{ème}}$: Peut-on prouver la cohérence de l'arithmétique ?
  - 10$^{\text{ème}}$: Peut-on trouver les solutions d'une équation diophantienne *au moyen d'un nombre fini d'opérations* ?
- *Entscheidungsproblem* (Hilbert-Ackermann, 1928) : existe-t-il un algorithme pour décider si un énoncé mathématique est correct ?
- Gödel (1931) : solution négative au 2$^{\text{ème}}$ problème de Hilbert :
  « pas avec les outils de l'arithmétique seule »
  - 2$^{\text{ème}}$ de ses théorèmes d'incomplétude
  - plus gl$^{\text{t}}$: dans un système axiomatique *riche*, certains énoncés sont indémontrables

Contexte historique

- 23 problèmes de Hilbert pour le xx$^{\text{ème}}$ siècle (1900) :
  - 2$^{\text{ème}}$: Peut-on prouver la cohérence de l'arithmétique ?
  - 10$^{\text{ème}}$: Peut-on trouver les solutions d'une équation diophantienne *au moyen d'un nombre fini d'opérations* ?
- *Entscheidungsproblem* (Hilbert-Ackermann, 1928) : existe-t-il un algorithme pour décider si un énoncé mathématique est correct ?
- Gödel (1931) : solution négative au 2$^{\text{ème}}$ problème de Hilbert : « pas avec les outils de l'arithmétique seule »
  - 2$^{\text{ème}}$ de ses théorèmes d'incomplétude
  - plus gl$^{\text{t}}$: dans un système axiomatique *riche*, certains énoncés sont indémontrables

Comment formaliser mathématiquement la notion de calcul ?

L'année 1936

- Années 1920 : fonctions *récursives primitives* (Skolem)
  - Insuffisant (Ackermann, 1928)

L'année 1936

- Années 1920 : fonctions *récursives primitives* (Skolem)
  - Insuffisant (Ackermann, 1928)
- 1931 : *fonctions générales récursives* (Herbrand-Gödel)

L'année 1936

- Années 1920 : fonctions *récursives primitives* (Skolem)
  - Insuffisant (Ackermann, 1928)
- 1931 : *fonctions générales récursives* (Herbrand-Gödel)
- 1936 :
  - λ-calcul (Church)
  - Fonctions μ-récursives (Kleene)
  - Machine de Turing (...)

  - Équivalence des formalismes
  - Réponse négative à l'*Entscheidungsproblem*
  - Indécidabilité du problème de l'arrêt

L'année 1936

- Années 1920 : fonctions *récursives primitives* (Skolem)
  - Insuffisant (Ackermann, 1928)
- 1931 : *fonctions générales récursives* (Herbrand-Gödel)
- 1936 :
  - λ-calcul (Church)
  - Fonctions μ-récursives (Kleene)
  - Machine de Turing (...)

  - Équivalence des formalismes
  - Réponse négative à l'*Entscheidungsproblem*
  - Indécidabilité du problème de l'arrêt

La *bonne* définition de calcul est trouvée !

Thèse de Church (ou Church-Turing)

Les formalismes équivalents (λ-calcul, machine de Turing, fonctions μ-récursives, ...) capturent la notion de *méthode effective de calcul*.

Thèse de Church (ou Church-Turing)

Les formalismes équivalents (λ-calcul, machine de Turing, fonctions μ-récursives, ...) capturent la notion de *méthode effective de calcul*.

- Hypothèse sur le monde physique
- Jamais démentie à ce jour
- Par nature indémontrable
- Précède le développement de l'informatique

Thèse de Church (ou Church-Turing)

Les formalismes équivalents (λ-calcul, machine de Turing, fonctions μ-récursives, ...) capturent la notion de *méthode effective de calcul*.

- Hypothèse sur le monde physique
- Jamais démentie à ce jour
- Par nature indémontrable
- Précède le développement de l'informatique

Autres formalismes :

- machine RAM
- tous les langages de programmation (même CSS ou LaTeX !)
- automates cellulaires, *jeu de la vie*, *Minecraft*, ...

Un peu de vocabulaire de calculabilité

Algorithme : choix d'un *modèle de calcul* ; ici, *les programmes Python*

Un peu de vocabulaire de calculabilité

Algorithme : choix d'un *modèle de calcul* ; ici, *les programmes Python*

Fonction : va d'ensemble d'entrées vers un ensemble de sorties

Un peu de vocabulaire de calculabilité

Algorithme : choix d'un *modèle de calcul* ; ici, *les programmes Python*

Fonction : va d'ensemble d'entrées vers un ensemble de sorties

Fonction partielle : si l'algorithme boucle sur une entrée, la fonction est *indéfinie*

Un peu de vocabulaire de calculabilité

Algorithme : choix d'un *modèle de calcul* ; ici, *les programmes Python*

Fonction : va d'ensemble d'entrées vers un ensemble de sorties

Fonction partielle : si l'algorithme boucle sur une entrée, la fonction est *indéfinie*

Problème de décision : fonction partielle à valeur dans $\{0, 1\}$

Un peu de vocabulaire de calculabilité

Algorithme : choix d'un *modèle de calcul* ; ici, *les programmes Python*

Fonction : va d'ensemble d'entrées vers un ensemble de sorties

Fonction partielle : si l'algorithme boucle sur une entrée, la fonction est *indéfinie*

Problème de décision : fonction partielle à valeur dans $\{0, 1\}$

Fonction ou problème calculable : il existe un algorithme qui la ou le calcule

Un peu de vocabulaire de calculabilité

Algorithme : choix d'un *modèle de calcul* ; ici, *les programmes Python*
Fonction : va d'ensemble d'entrées vers un ensemble de sorties
Fonction partielle : si l'algorithme boucle sur une entrée, la fonction est *indéfinie*
Problème de décision : fonction partielle à valeur dans $\{0, 1\}$
Fonction ou problème calculable : il existe un algorithme qui la ou le calcule

- Plusieurs algorithmes peuvent calculer la même fonction
- La quasi-totalité des fonctions est non calculable : (in)dénombrabilité

L'algorithme : une donnée comme une autre

- Quelque soit le formalisme :
  - l'algorithme est un *objet fini*
  - lui-même manipulable par un algorithme

L'algorithme : une donnée comme une autre

- Quelque soit le formalisme :
  - l'algorithme est un *objet fini*
  - lui-même manipulable par un algorithme
- Formalisme des programmes Python :
  - l'algorithme est une chaîne de caractères

    ```
    ch_pg = "def f(x):\n \treturn 3*x+1"
    ```
  - donc on peut chercher un motif (y a-t-il une boucle `while` ?), ...

L'algorithme : une donnée comme une autre

- Quelque soit le formalisme :
  - l'algorithme est un *objet fini*
  - lui-même manipulable par un algorithme
- Formalisme des programmes Python :
  - l'algorithme est une chaîne de caractères
    `ch_pg = "def f(x):\n \treturn 3*x+1"`
  - donc on peut chercher un motif (y a-t-il une boucle `while` ?), ...
- Exemples :
  - Compilateurs, interpréteurs, ...
  - Éditeur de texte, IDE, ...
  - Analyse statique de programme

Les programmes universels

Dans tout formalisme de calcul, il existe un *programme universel* qui prend en paramètre un algorithme et une entrée, et simule l'algorithme sur l'entrée.

Les programmes universels

Dans tout formalisme de calcul, il existe un *programme universel* qui prend en paramètre un algorithme et une entrée, et simule l'algorithme sur l'entrée.

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

```
>>> ch_pg='def f(x):\n\treturn 3*x'
>>> universel(ch_pg,17)
51
```

Les programmes universels

Dans tout formalisme de calcul, il existe un *programme universel* qui prend en paramètre un algorithme et une entrée, et simule l'algorithme sur l'entrée.

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

```
>>> ch_pg='def f(x):\n\treturn 3*x'
>>> universel(ch_pg,17)
51
```

- Un peu de triche... mais c'est possible sans tricher !
- Programme universel $\simeq$ interpréteur écrit dans le langage lui-même

Le problème de l'arrêt

Entrées : Un algorithme A et une entrée x
Sortie : oui/non : l'algorithme A termine-t-il sur l'entrée x ?

Le problème de l'arrêt

Entrées : Un algorithme A et une entrée x
Sortie : oui/non : l'algorithme A termine-t-il sur l'entrée x ?

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

Le problème de l'arrêt

Entrées : Un algorithme A et une entrée x
Sortie : oui/non : l'algorithme A termine-t-il sur l'entrée x ?

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

- Problème important en pratique !

Le problème de l'arrêt est indécidable

```
def arret(ch_algo, m):
    """
    Renvoie True si algo termine
    sur l'entrée m, False sinon
    """
    ... # comment faire ?
```

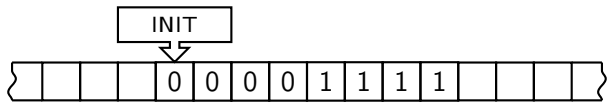
Le problème de l'arrêt est indécidable

```
def arret(ch_algo, m):
    """
    Renvoie True si algo termine
    sur l'entrée m, False sinon
    """
    ... # comment faire ?
```

```
def diagonal(ch_algo):
    # not arret(ch_algo, ch_algo)
    if not universel(ch_arret, ch_algo, ch_algo):
        return True
    while True: # on boucle
        pass
```

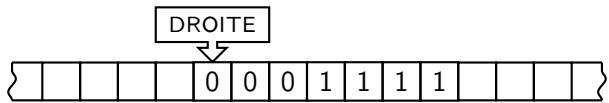
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



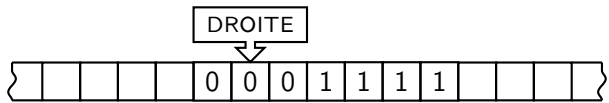
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



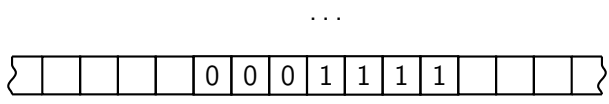
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



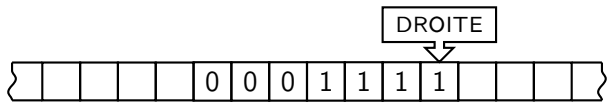
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



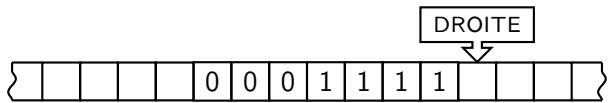
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



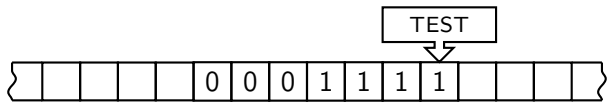
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



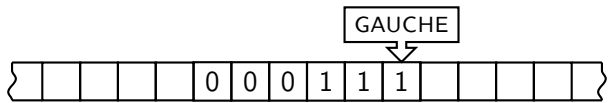
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	☐	☐	$\rightarrow$	VRAI
INIT	0	☐	$\rightarrow$	DROITE
DROITE	0 ou 1	*idem*	$\rightarrow$	DROITE
DROITE	☐	☐	$\leftarrow$	TEST
TEST	1	☐	$\leftarrow$	GAUCHE
GAUCHE	0 ou 1	*idem*	$\leftarrow$	GAUCHE
GAUCHE	☐	☐	$\rightarrow$	INIT



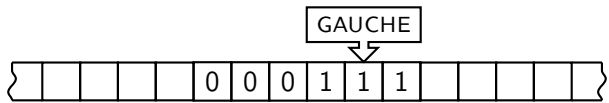
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



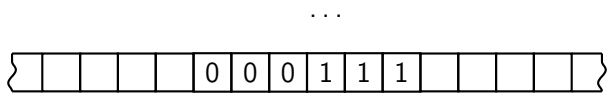
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



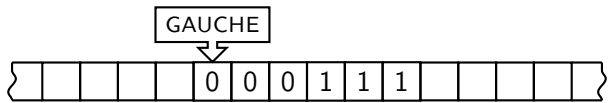
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



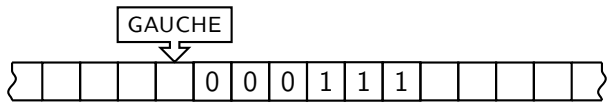
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



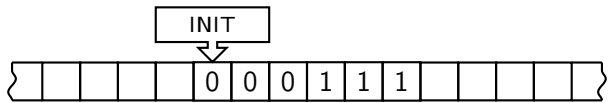
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	☐	☐	$\rightarrow$	VRAI
INIT	0	☐	$\rightarrow$	DROITE
DROITE	0 ou 1	*idem*	$\rightarrow$	DROITE
DROITE	☐	☐	$\leftarrow$	TEST
TEST	1	☐	$\leftarrow$	GAUCHE
GAUCHE	0 ou 1	*idem*	$\leftarrow$	GAUCHE
GAUCHE	☐	☐	$\rightarrow$	INIT



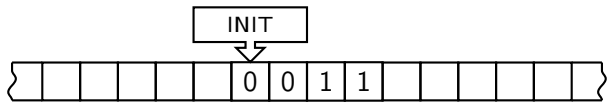
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



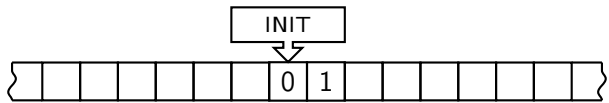
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



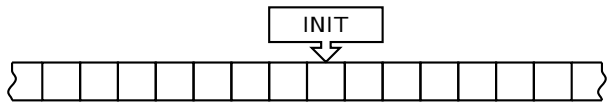
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



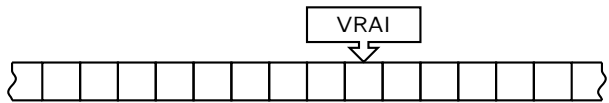
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	☐	☐	$\rightarrow$	VRAI
INIT	0	☐	$\rightarrow$	DROITE
DROITE	0 ou 1	*idem*	$\rightarrow$	DROITE
DROITE	☐	☐	$\leftarrow$	TEST
TEST	1	☐	$\leftarrow$	GAUCHE
GAUCHE	0 ou 1	*idem*	$\leftarrow$	GAUCHE
GAUCHE	☐	☐	$\rightarrow$	INIT



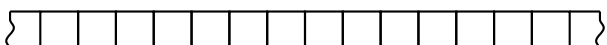
Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



Une formalisation : la machine de Turing

État courant	Symbole lu	Symbole écrit	Déplacement	Nouvel état
INIT	□	□	→	VRAI
INIT	0	□	→	DROITE
DROITE	0 ou 1	*idem*	→	DROITE
DROITE	□	□	←	TEST
TEST	1	□	←	GAUCHE
GAUCHE	0 ou 1	*idem*	←	GAUCHE
GAUCHE	□	□	→	INIT



Complexité

Complexité des algorithmes / complexité des problèmes

- Étant donné un algorithme, on définit sa complexité en temps, ou en espace, ...

La complexité (en temps, en espace, ...) d'un problème est la *plus petite* complexité d'un algorithme qui résout le problème.

Classes de complexité

Théorie de la complexité

- regrouper les problèmes par *classes* qui ont une même *borne supérieure* de complexité
- comparer les classes entre elles

Objectif : expliquer pourquoi certains problèmes ont des algorithmes efficaces et d'autres non

Classes de complexité

Théorie de la complexité

- regrouper les problèmes par *classes* qui ont une même *borne supérieure* de complexité
- comparer les classes entre elles

Objectif : expliquer pourquoi certains problèmes ont des algorithmes efficaces et d'autres non

Exemples

- Problèmes linéaires : algo. de complexité $O(n)$
- Problèmes quadratiques : algo. de complexité $O(n^2)$
- linéaire $\subset$ quadratique

Classe P

La classe P est la classe des problèmes de décision qui ont un algorithme de complexité polynomiale en la taille d'entrée.

Classe P

La classe P est la classe des problèmes de décision qui ont un algorithme de complexité polynomiale en la taille d'entrée.

- Recherche de motif dans un texte
- Est-ce que m divise n ?
  - entrées : $O(\log(m) + \log(n))$
  - complexité : $O(\log(m)\log(n))$
- ...

La classe P peut être considérée comme la classe des problèmes *faciles à résoudre*

Classe NP

La classe NP est la classe des problèmes de décision qui ont un algorithme de *vérification* V de complexité polynomiale : pour toute entrée x,

- si l'entrée est *positive*, il existe un *certificat* y tel que $V(x, y) = 1$
- si l'entrée est *négative*, pour tout certificat y, $V(x, y) = 0$

Classe NP

La classe NP est la classe des problèmes de décision qui ont un algorithme de *vérification* V de complexité polynomiale : pour toute entrée x,

- si l'entrée est *positive*, il existe un *certificat* y tel que $V(x, y) = 1$
- si l'entrée est *négative*, pour tout certificat y, $V(x, y) = 0$

Somme partielle

Entrées : T : tableau d'entiers ; t : entier cible

Question : Est-il possible de sélectionner un sous-ensemble des éléments de T dont la somme vaut t ?

Classe NP

La classe NP est la classe des problèmes de décision qui ont un algorithme de *vérification* V de complexité polynomiale : pour toute entrée x,

- si l'entrée est *positive*, il existe un *certificat* y tel que $V(x, y) = 1$
- si l'entrée est *négative*, pour tout certificat y, $V(x, y) = 0$

Somme partielle

Entrées : T : tableau d'entiers ; t : entier cible

Question : Est-il possible de sélectionner un sous-ensemble des éléments de T dont la somme vaut t ?

- On peut tester toutes les possibilités $\rightsquigarrow O(2^n)$ où $n = |T|$
- Si on fournit un sous-ensemble de T, *facile* de vérifier si sa somme vaut t

La question à 1 000 000 de dollars

Est-ce que $P = NP$?

- Autrement dit, si on dispose d'un algorithme de vérification polynomial, a-t-on forcément un algorithme polynomial ?
- On sait que $P \subset NP$ mais l'inverse ?
- Conjecture : $P \neq NP$

Réductions entre problèmes

- Réduction d'un problème A à un problème B : algorithme R qui envoie une entrée x de A sur une entrée $y = R(x)$ de B, tel que x est une entrée positive si et seulement si y l'est.

Réductions entre problèmes

- Réduction d'un problème A à un problème B : algorithme R qui envoie une entrée x de A sur une entrée $y = R(x)$ de B, tel que x est une entrée positive si et seulement si y l'est.

Partition

Entrée : Tableau d'entiers T

Question : Existe-t-il une partition de T en $T_1 \sqcup T_2$ telles que $\sum_{x \in T_1} x = \sum_{x \in T_2} x$?

Réduction et classes

Soit A et B deux problèmes, et R une réduction polynomiale de A à B.

- Si B a un algo. polynomial, alors A également. $B \in \mathrm{P} \Rightarrow A \in \mathrm{P}$
- Si B a un algo. de vérification polynomial, alors A également. $B \in \mathrm{NP} \Rightarrow A \in \mathrm{NP}$

NP-complétude (1971)

Il existe des problèmes B dans NP tels que *quelque soit* A *dans* NP, il existe une réduction polynomiale de A à B. On les appelle les problèmes NP-complets.

- Somme partielle, partition, ... et des dizaines de milliers d'autres

NP-complétude (1971)

Il existe des problèmes B dans NP tels que *quelque soit* A *dans* NP, il existe une réduction polynomiale de A à B. On les appelle les problèmes NP-complets.

- Somme partielle, partition, ... et des dizaines de milliers d'autres
- Si B est NP-complet et que B a un algo. polynomial : P = NP !
- Si B est NP-complet et qu'il existe une réduction de B à $C \in$ NP : C est NP-complet.

NP-complétude (1971)

Il existe des problèmes B dans NP tels que *quelque soit* A *dans* NP, il existe une réduction polynomiale de A à B. On les appelle les problèmes NP-complets.

- Somme partielle, partition, ... et des dizaines de milliers d'autres

- Si B est NP-complet et que B a un algo. polynomial : P = NP !
- Si B est NP-complet et qu'il existe une réduction de B à $C \in$ NP : C est NP-complet.

Résoudre « P = NP ? »

- Il *suffit* de résoudre la question pour *un* problème NP-complet
- Il *faut* résoudre la question pour au moins un problème...

Conclusion

- Calculabilité :
  - Introduite dans les années 1930
  - Qu'est-ce que veut dire calculer ?
  - A précédé l'informatique
  - Encore active aujourd'hui !

« Qu'est-ce que veut dire être aléatoire ? »

Conclusion

- Calculabilité :
  - Introduite dans les années 1930
  - Qu'est-ce que veut dire calculer ?
  - A précédé l'informatique
  - Encore active aujourd'hui ! « Qu'est-ce que veut dire être aléatoire ? »
- Complexité :
  - Introduite dans les années 1960
  - Que peut-on calculer efficacement ?
  - Motivée par la pratique de l'informatique
  - Très active, implications fortes en cryptographie, mais aussi en physique théorique
  - Question « P = NP » ?