DIU Enseigner l'informatique au lycée

Notes de cours

Jean-Christophe Filliâtre

Jean-Christophe.Filliatre@lri.fr

programme de la journée

9h00	complexité exercices
10h30	pause
10h40	calculabilité exercices
12h00	pause déjeuner
13h00	apprentissage exercices
14h20	pause
14h30	algorithmes probabilistes exercices
16h00	fin de la journée

Complexité

performance

Les performances d'un programme se mesurent principalement selon deux critères :

- le temps de calcul : on parle de complexité temporelle ;
- l'espace mémoire utilisé : on parle de complexité spatiale.

Pour un même problème, on peut proposer différents algorithmes, avec des complexités différentes.

exemple

Pour trier un tableau contenant N valeurs, il existe de très nombreux algorithmes, avec des complexités différentes :

- complexité temporelle proportionnelle à N^2, à $N \log N$, voire même à N dans certains cas particuliers (cf plus loin) ;
- complexité spatiale proportionnelle à N (par ex. on utilise un tableau temporaire), à $\log N$ (par ex. on fait des appels récursifs) ou encore constante.

plusieurs cas de figure

Qu'elle soit temporelle ou spatiale, on peut distinguer la complexité

- dans le pire des cas ;
- en moyenne ;
- dans le meilleur des cas.

exemple

Pour la complexité temporelle du tri par insertion de N valeurs,

- dans le pire des cas : proportionnelle à N^2 ;
- en moyenne : proportionnelle à N^2 ;
- dans le meilleur des cas : proportionnelle à N.

Note : les calculs de complexité en moyenne supposent de faire des hypothèses de répartition des entrées et sont en général difficiles.

comparaisons

Un algorithme peut être plus performant qu'un autre dans un cas, mais moins performant dans un autre cas.

	moyenne	pire cas	meilleur cas
insertion	N^2	N^2	N
fusion	$N \log N$	$N \log N$	N
rapide	$N \log N$	N^2	$N \log N$
par tas	$N \log N$	$N \log N$	$N \log N$

La donnée correspondant au pire (resp. meilleur) d'un algorithme ne correspond pas forcément au pire (resp. meilleur) cas d'un autre algorithme.

complexité asymptotique

En théorie de la complexité, on s'intéresse principalement à la complexité asymptotique.

Ainsi, deux algorithmes de tri de complexités temporelles respectives

$$18N\lfloor \log N \rfloor + 28N \quad \text{et} \quad 24N\lfloor \log N \rfloor + 12N$$

unités de temps sont considérés de même complexité $O(N \log N)$.

On parle alors d'un algorithme en $N \log N$.

complexité asymptotique

La constante cachée dans le O peut effectivement être ignorée le plus souvent.

Ainsi, un algorithme en $O(N^2)$ prendra quatre fois plus de temps pour une entrée deux fois plus grande, au moins à partir d'un certain rang.

Et il sera moins performant qu'un algorithme en $O(N)$ à partir d'un certain rang.

complexité asymptotique et complexité en pratique

Il existe cependant des cas où la constante cachée peut avoir une importance.

Deux exemples :

- Le tri par tas et le tri rapide sont tous les deux en $O(N \log N)$ mais on préfère le tri rapide en pratique car la constante est plus petite.

- Un tas de Fibonacci permet d'améliorer la complexité asymptotique de l'algorithme de Dijkstra, mais en pratique on ne l'utilise pas car la constante est trop grande.

mesure du temps d'exécution en Python

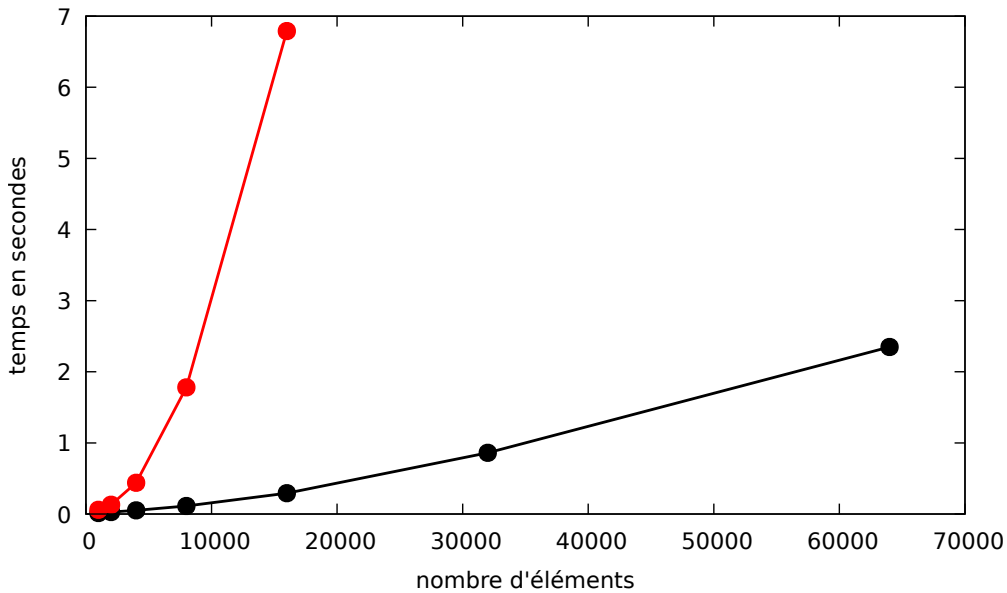
```
from time import perf_counter()
debut = perf_counter()
...
fin = perf_counter()
print("temps écoulé : ", fin - debut, "secondes")
```

activités autour de ces mesures

- mesurer le temps de calcul pour différentes valeurs d'un paramètre
- tracer une courbe
- estimer une complexité
- comparer avec notre intuition / la théorie
- comparer les complexités de deux programmes

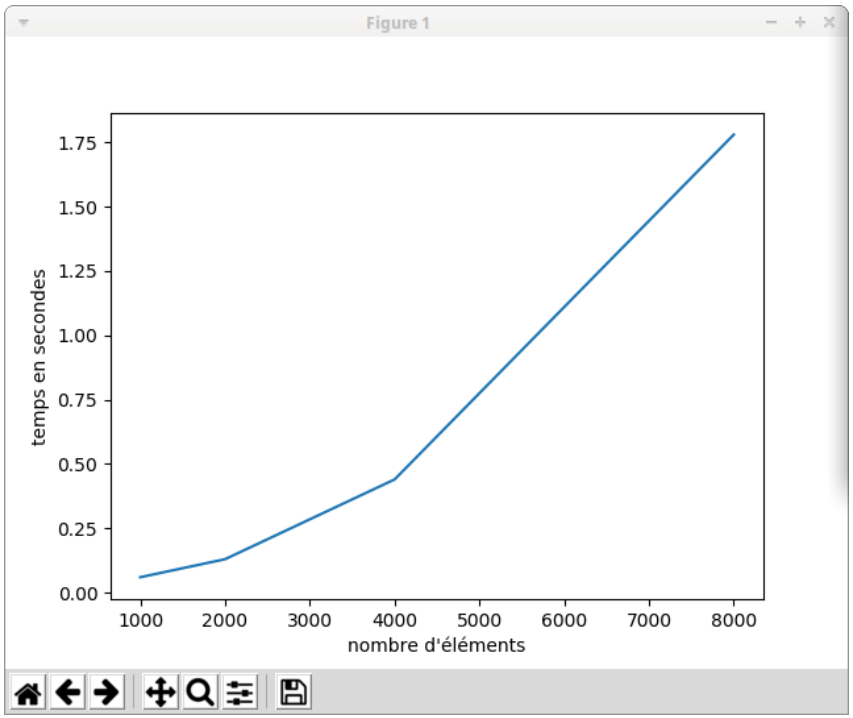
exemple

taille	sélection	fusion
1 000	0,06 s	0,01 s
2 000	0,13 s	0,03 s
4 000	0,44 s	0,05 s
8 000	1,78 s	0,11 s
16 000	6,79 s	0,29 s
32 000	–	0,86 s
64 000	–	2,34 s



tracé avec matplotlib

```
import matplotlib.pyplot as plt
plt.xlabel("nombre d'éléments")
plt.ylabel("temps en secondes")
plt.plot([1000, 2000, 4000, 8000], \
         [0.06, 0.13, 0.44, 1.78])
plt.show()
```



une machine d'aujourd'hui

On observe ceci :

- 1 million d'opérations : instantané
- 1 milliard d'opérations : 30 secondes
- 1000 milliards d'opérations : trop long

Bien entendu, cela dépend de la nature des opérations (addition, appel de fonction, allocation mémoire, etc.) et du langage (interprété, compilé, etc.).

Mais c'est un bon ordre de grandeur.

une machine d'aujourd'hui

Ainsi, le programme Python

```
i = 0
while i < 1_000_000_000:
    i = i + 1
```

s'exécute sur ma machine en 1 min 5 s.

complexités comparées

Comparons quatre complexités qui reviennent souvent :

$\log N$ (exemple : recherche dichotomique parmi N valeurs)

N (exemple : recherche du maximum parmi N valeurs)

$N \log N$ (exemple : tri fusion de N valeurs)

N^2 (exemple : tri par insertion de N valeurs)

complexités comparées

Les valeurs ne sont pas forcément très parlantes :

$\log N$	10^1	2×10^1	3×10^1
N	10^3	10^6	10^9
$N \log N$	10^4	2×10^7	3×10^{10}
N^2	10^6	10^{12}	10^{18}

Note : $\log$ désigne ici le logarithme à base 2 ; ainsi, $\log(128) = 7$.

complexités comparées

Quel est le temps de calcul de mon programme,

en supposant 1 milliard d'opérations par minute ?

N	10^3	10^6	10^9
$\log N$	inst.	inst.	inst.
N	inst.	inst.	1 min
$N \log N$	inst.	$1,2$ s	30 min
N^2	inst.	> 16 h	> 1900 ans

complexités comparées

Jusqu'à quelle valeur de N peut-on aller ?

(toujours en supposant 1 milliard d'opérations par minute)

	1 s	1 min	1 h
$\log N$	∞	∞	∞
N	$1,6 \times 10^7$	10^9	6×10^{10}
$N \log N$	$8,5 \times 10^5$	$4,0 \times 10^7$	$1,9 \times 10^9$
N^2	$4,1 \times 10^3$	$3,2 \times 10^4$	$2,4 \times 10^5$

complexité d'un problème

Prenons l'exemple du problème du tri.

À quelle vitesse peut-on espérer trier ?

Dit autrement :

quelle est la meilleure complexité dans le pire des cas ?

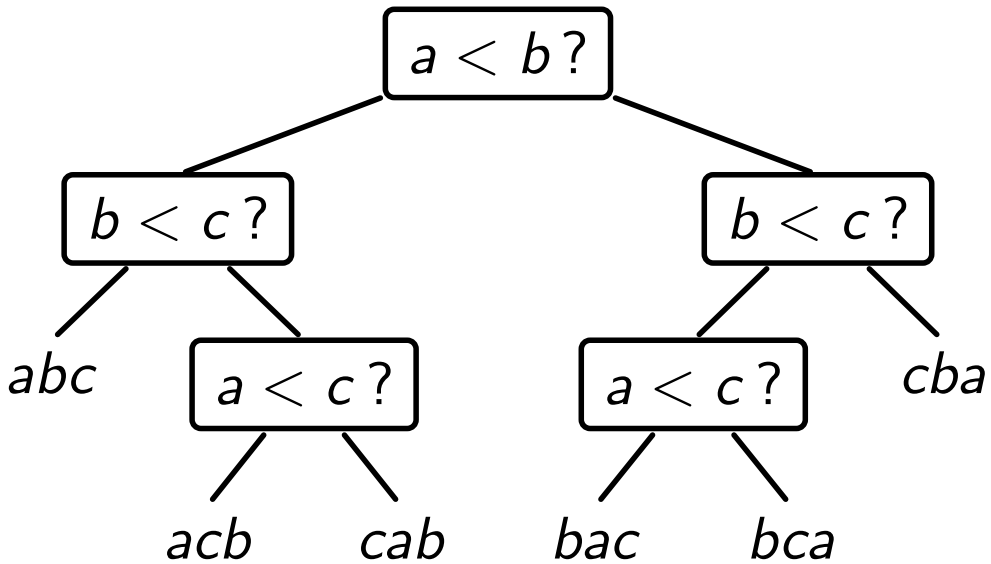
hypothèses

On suppose que l'algorithme

- ne procède que par des comparaisons entre deux éléments ;
- ne possède aucune information quant à la distribution.

le cas de 3 valeurs

On peut trier trois valeurs a, b, c ainsi



Un tel arbre, appelé *questionnaire*, représente l'algorithme.

coût

Ici, on fait 2 ou 3 comparaisons selon l'entrée,

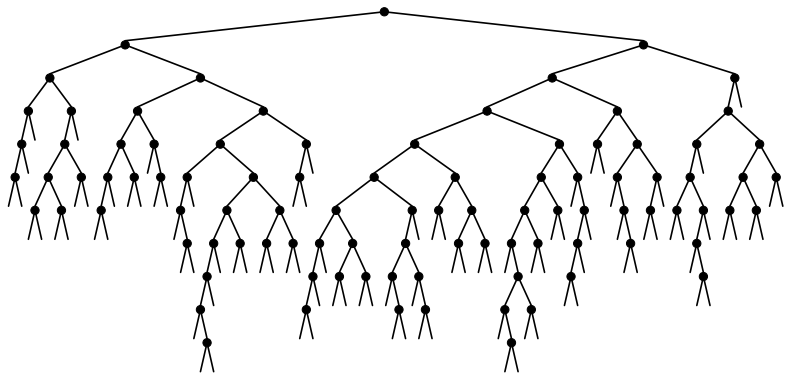
donc 3 comparaisons dans le pire des cas.

On se persuade facilement que c'est optimal pour $N = 3$:

on ne peut trier trois valeurs avec au plus deux comparaisons.

de manière générale

Un algorithme qui trie N valeurs peut être vu comme un arbre binaire où chaque nœud est un test et chaque feuille une permutation.



- Il y a au moins $N!$ feuilles.
- La complexité dans le pire des cas est la hauteur de cet arbre.

minoration

Or,

$$N! \leq \text{nombre de feuilles} \leq 2^{\text{hauteur}}$$

donc

$$\text{hauteur} \geq \log_2(N!) = \sum_{1 \leq i \leq N} \log_2(i) \sim N \log_2(N).$$

complexité du problème du tri

Résultat : Aucun algorithme, procédant par comparaisons uniquement, ne peut trier N valeurs en effectuant toujours moins que

$$N \log N$$

comparaisons.

En particulier, le tri fusion est optimal.

cas particuliers

En revanche, on peut faire mieux quand on possède une information supplémentaire sur les éléments.

Deux exemples :

- Si les N éléments ne prennent que deux valeurs distinctes, alors on peut trier en $O(N)$.
- Si les éléments sont des entiers 64 bits ou des chaînes de caractères, alors on peut trier en $O(N)$.

réduction de problème

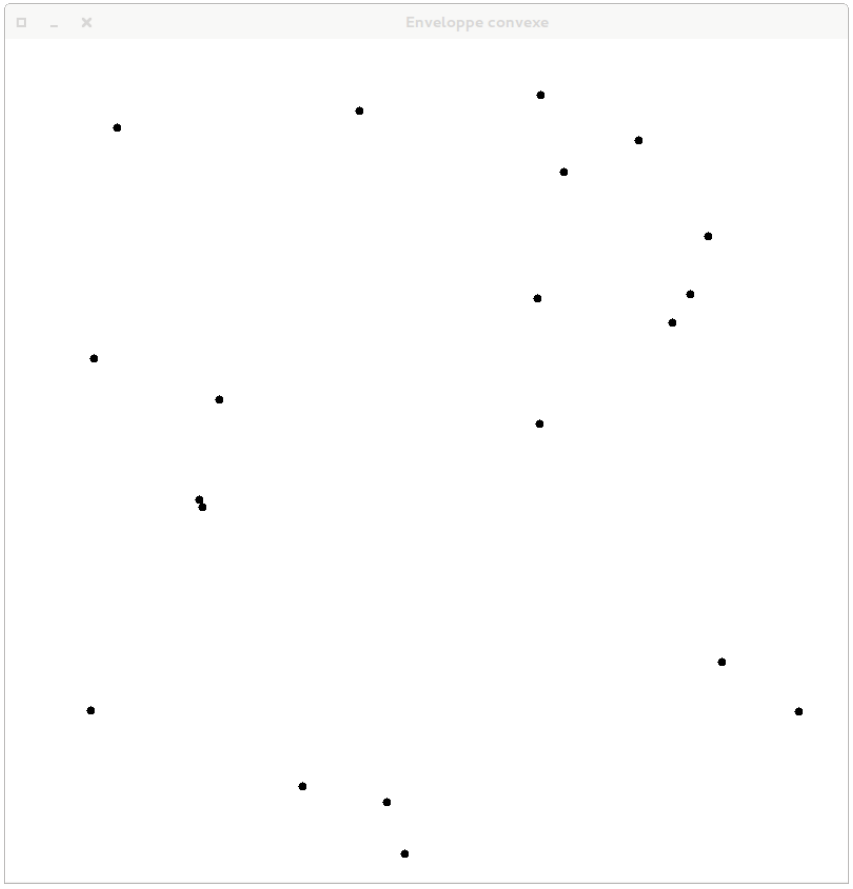
On peut minorer la complexité d'un problème P en réduisant un problème Q, dont la complexité est connue, au problème P.

Exemple : le problème de l'enveloppe convexe de N points est de complexité au moins $N \log N$ car on peut réduire le problème du tri vers le problème de l'enveloppe convexe.

problème de l'enveloppe convexe

entrée $= N$ points dans le plan

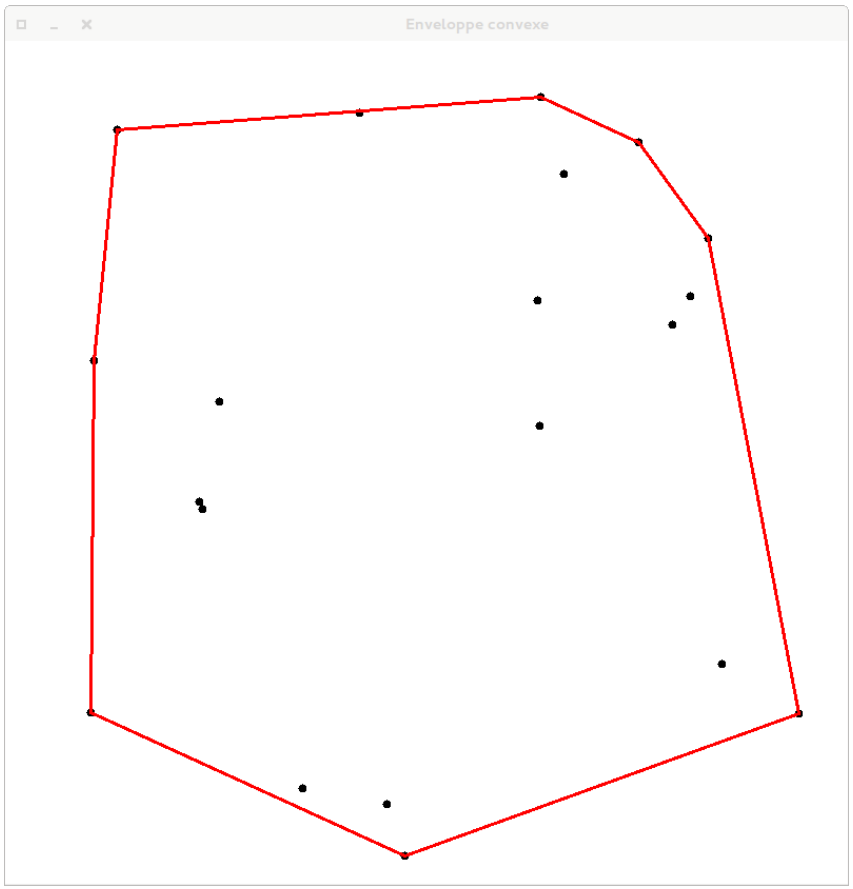
sortie = suite de points formant l'enveloppe convexe



problème de l'enveloppe convexe

entrée $=$ N points dans le plan

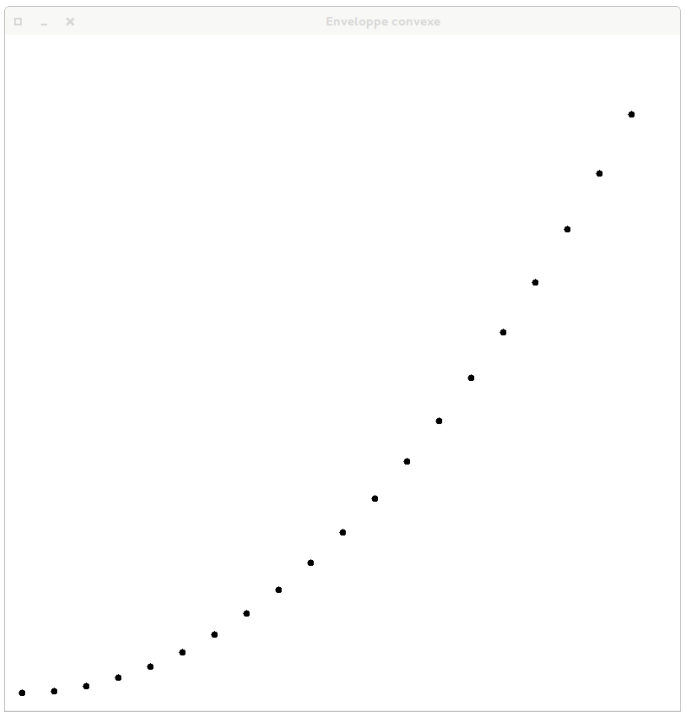
sortie $=$ suite de points formant l'enveloppe convexe

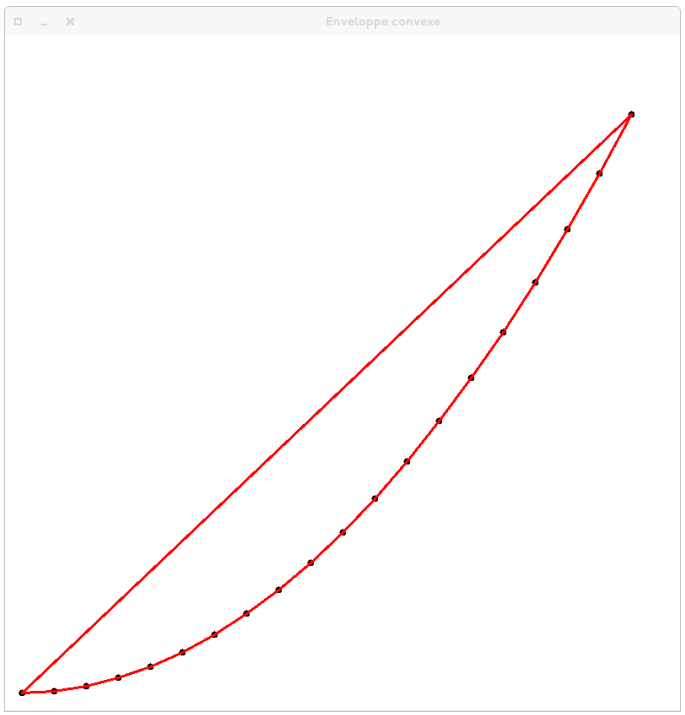


réduction tri $\rightarrow$ enveloppe

Soient N entiers $x_1, \ldots, x_N$.

Trouver l'enveloppe convexe des points (x_i, x_i^2) revient à les trier.

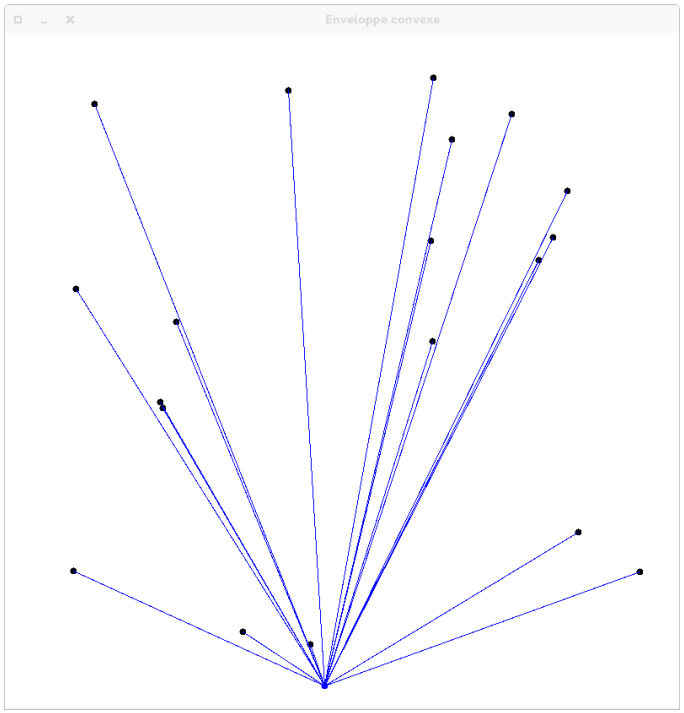




algorithme de Graham

1 déterminer le point p le plus bas

2 trier tous les points selon l'angle fait avec p



3 considérer les points dans cet ordre,
en les ajoutant/retirant de l'enveloppe convexe

démo

complexité

- L'étape 1 est clairement en $O(N)$.
- L'étape 2 est en $O(N \log N)$ (tri de N valeurs).
- L'étape 3 est $O(N)$, car chaque point
  - est ajouté une fois dans l'enveloppe,
  - est retiré au plus une fois de l'enveloppe.

Au total, c'est donc une solution en $O(N \log N)$.

Et on a vu qu'on ne pouvait pas faire mieux.

question

En Python, on peut ajouter un élément à la fin d'un tableau :

```
t.append(42)
```

Mais quelle est la complexité de cette opération ?

(et par quelle magie cela est-il possible ?)

curiosité

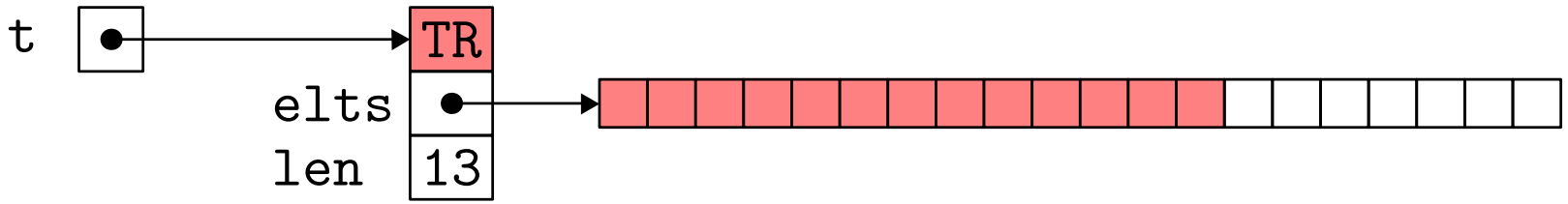
Livrons-nous à une petite expérience :

```
t = []
for i in range(100):
    print(t.__sizeof__())
    t.append(i)
```

```
40
72
72
72
72
104
104
104
104
168
168
168
168
168
168
168
168
168
240
...
```

tableau redimensionnable

En interne, Python utilise un tableau traditionnel, avec des cases éventuellement inutilisées.

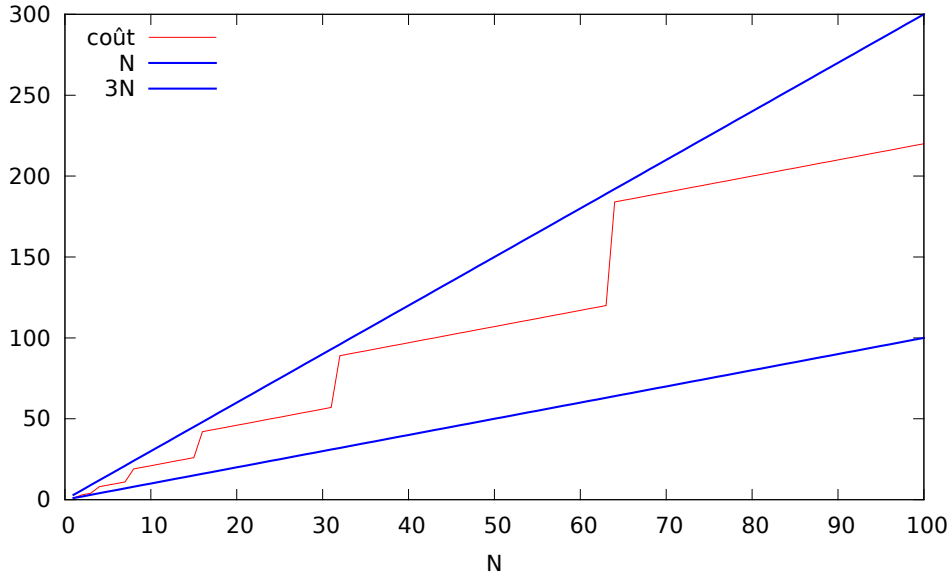


Quand on exécute `t.append(v)`, il y a deux cas de figure :

- Si le (vrai) tableau n'est pas encore plein, on y ajoute `v`. Le coût est constant.
- Si le (vrai) tableau est plein, on alloue un tableau plus grand (par exemple deux fois plus grand), on y copie toutes les valeurs, puis on ajoute `v`. Le coût est proportionnel à `len(t)`.

complexité linéaire au total

La plupart des ajouts ont un coût constant mais d'autres ont un coût proportionnel à la taille du tableau (1, 2, 4, 8, 16, etc.).



Au total, cependant, le coût de l'ajout successif de N éléments reste proportionnel à N.

complexité amortie

Le coût moyen d'un ajout, ramené au nombre total N d'opérations, est donc constant.

On dit que `append` a une complexité amortie constante.

Exercices

Calculabilité

question

On vient de voir que certains problèmes ne peuvent être résolus en-dessous d'une certaine complexité.

Mais existe-t-il des problèmes qui ne peuvent pas être résolus du tout, c'est-à-dire pour lesquels il n'est pas possible d'écrire un programme, quel que soit le temps que celui-ci prendrait ?

d'autres questions

Ceci nous amène à d'autres questions :

- qu'est-ce qu'un programme ?
- est-il important de fixer le langage de programmation ?
- est-il important de fixer l'ordinateur sur lequel il tourne ?

La calculabilité est là pour répondre à ces questions.

des programmes qui manipulent des programmes

Un programme est une donnée comme une autre.

L'interprète Python, par exemple, est un programme (appelé `python` sur ma machine) qui reçoit en entrée, comme donnée, un programme (ici le fichier `fichier.py`) :

```
$ python fichier.py
...
```

à notre tour

Écrivons une fonction Python qui reçoit en argument un programme, sous la forme d'un nom de fichier, et qui renvoie `True` si et seulement si l'exécution de ce programme s'interrompt avec `ZeroDivisionError`.

```
def detecteur(nom_de_fichier):
    fichier = open(nom_de_fichier)
    programme = fichier.read()
    ...
```

C'est peut-être (sûrement) très compliqué, mais on dispose de tout le texte du programme dans la chaîne de caractères `programme`.

première tentative

```
def detecteur(nom_de_fichier):
    fichier = open(nom_de_fichier)
    programme = fichier.read()
    try:
        exec(programme) # exécution du programme
        return False
    except ZeroDivisionError:
        return True
    except:
        return False
```

On se sert ici de exec, qui permet d'exécuter le code contenu dans une chaîne de caractères.

ça marche !

Avec le fichier `don_jose.py` contenant

```
i = 3
while i > 0:
    print("Prends garde à toi !")
    i = i - 1
print(1 / i)
print("Carmen épouse-moi !")
```

un appel à

```
detecteur_naif('don_jose.py')
```

renvoie bien `True` comme attendu.

mais pas toujours

Malheureusement, sur cet autre programme,

```
while True:
    print("Veillez sur vous !")
print(1 / 0)
```

un appel à `detecteur` ne termine pas, et donc ne répond pas à la question (ici la réponse devrait être `False`).

On comprend qu'il faudrait être plus malin.

Mais c'est peut-être possible (cherchons les `while True`, les fonctions récursives qui ne terminent pas, etc.)

impossibilité

Supposons avoir réussi à écrire en Python une fonction `detecteur_parfait`.

On considère alors le programme `carmen.py` contenant

```
def detecteur_parfait(fichier):
    ...

if detecteur_parfait('carmen.py'):
    print("Tu crois le tenir, il t'évite")
else:
    print("Tu crois l'éviter, il te tient")
    print(1 / 0)
```

et on obtient une contradiction !

impossibilité

La seule hypothèse était l'existence de `detecteur_parfait`.

On vient donc de montrer qu'il n'est pas possible d'écrire une telle fonction.

Est-ce là une limitation de Python ? Pourrait-on écrire un détecteur (toujours pour des programmes Python) dans un autre langage que Python ?

Et quid d'autres ≪ détecteurs ≫ ?

fonction calculable

On cherche donc à définir ce qui est calculable par un programme, ce qui correspond à un algorithme.

On appelle cela une fonction calculable.

(Il n'est en fait pas restrictif de se limiter même à des fonctions de $\mathbb{N}$ vers $\mathbb{N}$ car on peut encoder les objets complexes dans des entiers — cf le codage de Gödel.)

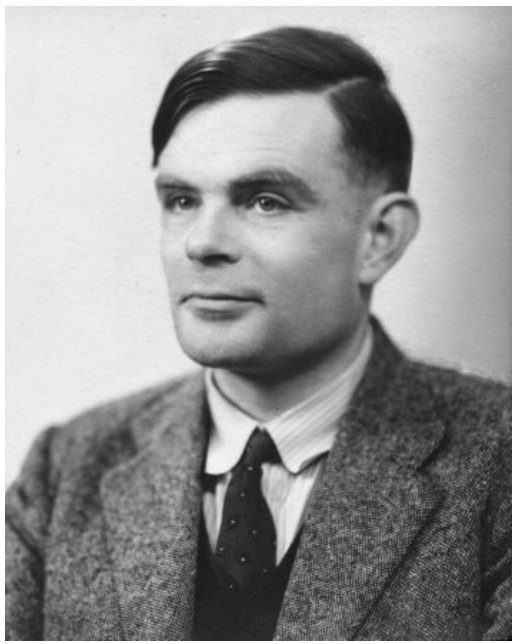
histoire

La notion de fonction calculable est apparue dans la première moitié du XX^{e} siècle, sous l'impulsion donnée par le programme de Hilbert.

En 1928, Hilbert et Ackermann posent le problème de la décision (Entscheidungsproblem) : existe-t-il un algorithme qui détermine, pour tout énoncé mathématique, s'il est vrai ou faux ?

histoire

En 1936, Alonzo Church et Alan Turing y apportent une réponse négative, indépendamment.



le λ-calcul de Church

Il s'agit d'un langage où une expression e est

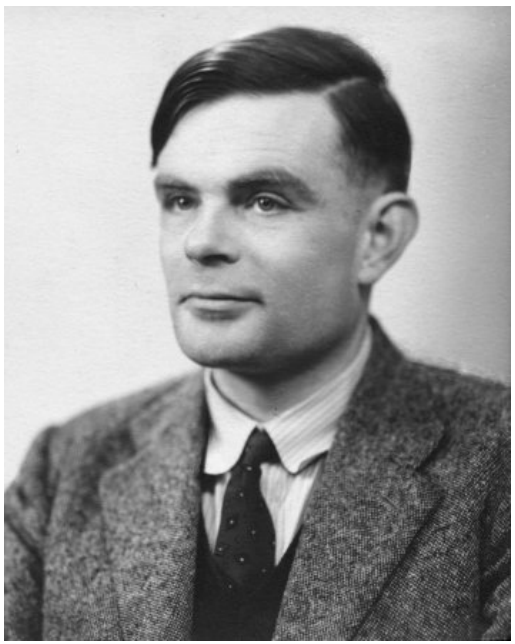
- soit une variable x,
- soit une fonction $x \mapsto e$, notée $\lambda x.\, e$,
- soit une application $e(e')$.

Il y a une seule règle de calcul : l'application de $\lambda x.\, e$ à e' donne l'expression e dans laquelle chaque occurrence de x est remplacée par e'.

Bien que minimal, ce langage permet de représenter des booléens, des paires, des entiers, etc., et de fait tout algorithme.

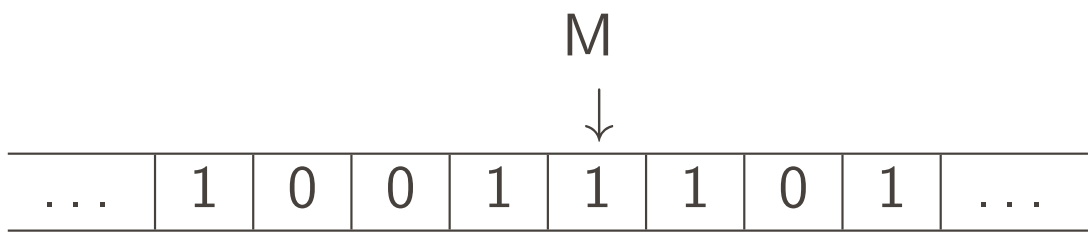
À la base des langages fonctionnels et de langages logiques.

la machine de Turing



Elle est constituée

- d'un ruban de mémoire infini,
- d'un ensemble fini d'états,
- d'un ensemble fini d'instructions de la forme « si la machine est dans l'état S et lit un caractère C sur le ruban, alors écrire le caractère C' sur le ruban, se déplacer éventuellement (à gauche ou à droite) et passer dans l'état S' ».



la machine de Turing

Une machine définit une fonction f.

1. On écrit l'entrée e sur le ruban.
2. On lance la machine.
3. Lorsqu'elle s'arrête (un état final indique que le calcul est terminé), on lit le résultat $f(e)$ sur le ruban.

En particulier, l'entrée et la sortie peuvent être des entiers écrits en binaire :

...	•	1	0	0	1	1	•	...

exemple de machine de Turing

Définissons une machine pour incrémenter un tel entier écrit en binaire.

Principe :

1. d'abord se déplacer jusqu'à l'extrémité droite,
2. puis ajouter 1 au bit de poids faible, et revenir vers la gauche autant que nécessaire pour propager les retenues.

...	•	1	0	0	1	1	•	...

exemple de machine de Turing

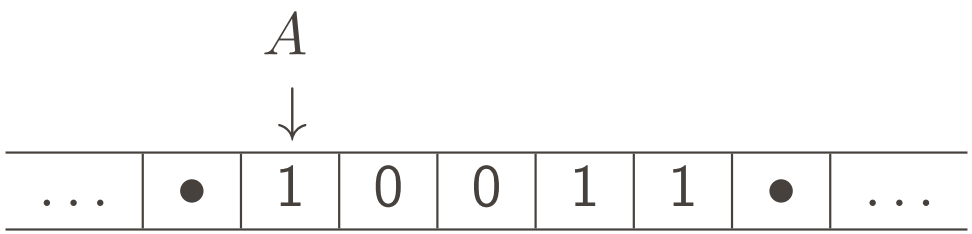
On se donne trois états :

- un état de départ A pour la première phase de déplacement ;
- un état B pour la phase d'incrément ;
- un état final F indiquant que le calcul est terminé.

...	•	1	0	0	1	1	•	...

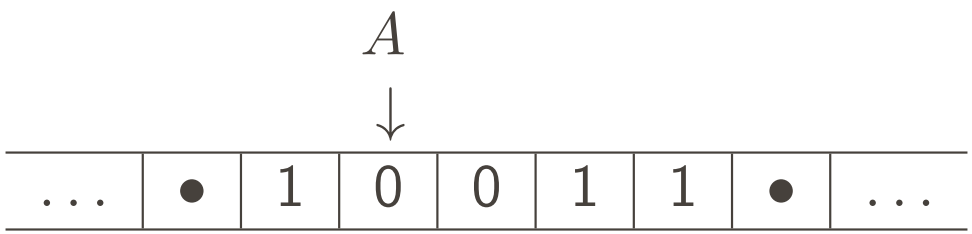
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0			
A	1			
A	•			
B	0			
B	1			
B	•			



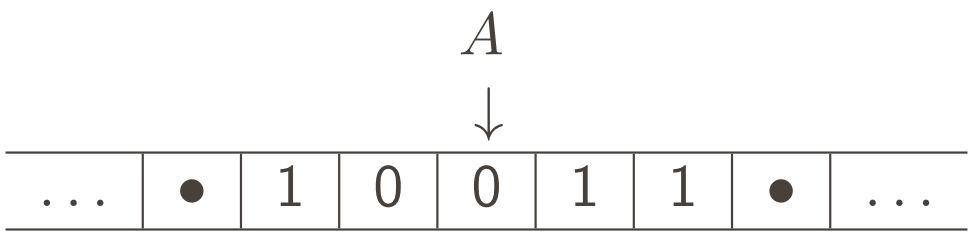
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	1	→	A
A	1			
A	•			
B	0			
B	1			
B	•			



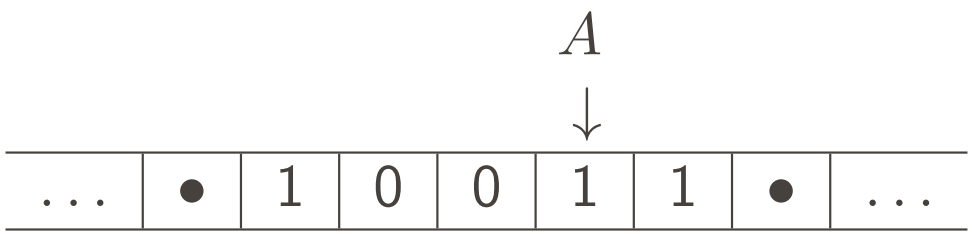
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•			
B	0			
B	1			
B	•			



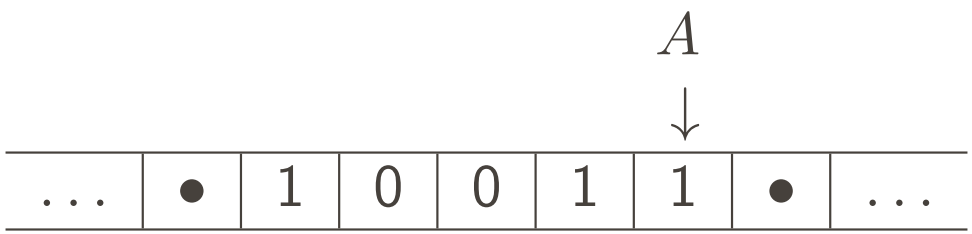
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•			
B	0			
B	1			
B	•			



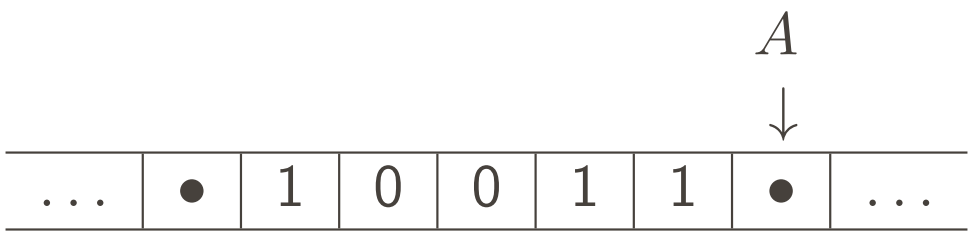
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•			
B	0			
B	1			
B	•			



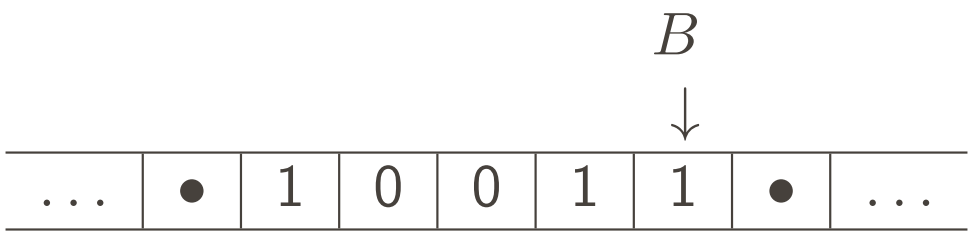
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•			
B	0			
B	1			
B	•			



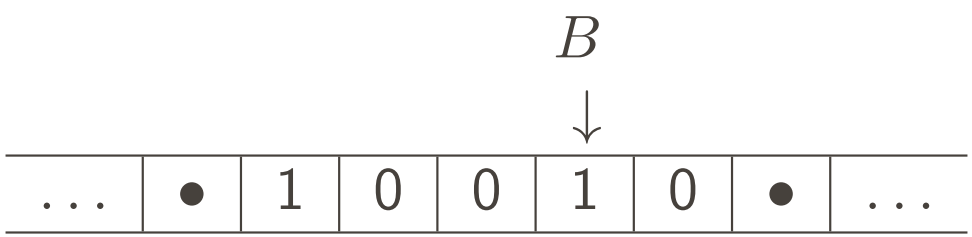
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•	•	$\leftarrow$	B
B	0			
B	1			
B	•			



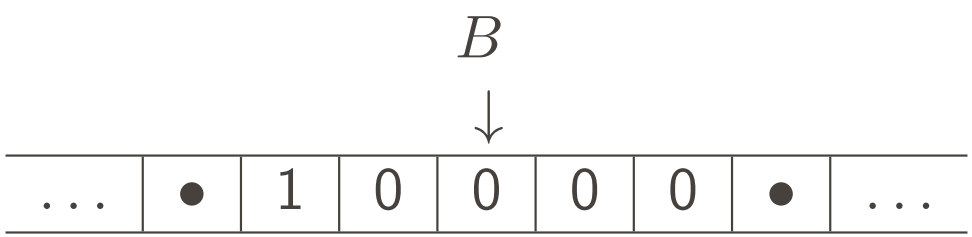
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•	•	$\leftarrow$	B
B	0			
B	1	0	$\leftarrow$	B
B	•			



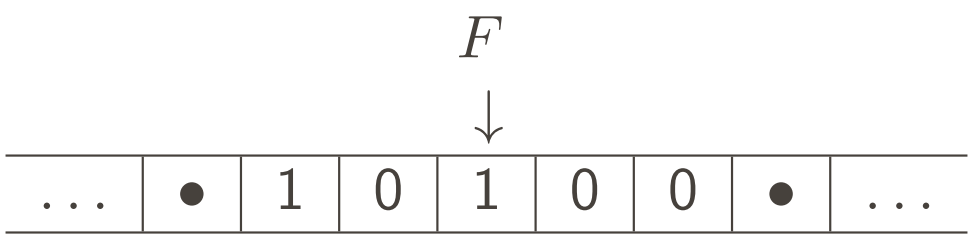
exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	•	•	$\leftarrow$	B
B	0			
B	1	0	$\leftarrow$	B
B	•			



exemple de machine de Turing

état	lu	écrit	dépl.	suiv.
A	0	0	$\rightarrow$	A
A	1	1	$\rightarrow$	A
A	$\bullet$	$\bullet$	$\leftarrow$	B
B	0	1	$\downarrow$	F
B	1	0	$\leftarrow$	B
B	$\bullet$	1	$\downarrow$	F



la thèse de Church-Turing

Le λ-calcul de Church et les machines de Turing ont le même pouvoir d'expressivité (i.e. une fonction calculable par l'un l'est aussi par l'autre).

La thèse de Church-Turing affirme qu'il n'existe aucun modèle de calcul plus expressif.

définition

Un modèle de calcul aussi expressif que les machines de Turing est dit Turing-complet.

Exemples :

- Le langage Python est Turing-complet.
- Le jeu de cartes *Magic : L'Assemblée* est Turing-complet ! Cf `https://youtu.be/OGeOQzGq4LE`

limites de la calculabilité

Church et Turing ont exhibé chacun un problème dont la solution n'est pas calculable.

- Pour Church : il n'y a pas de fonction calculable qui, recevant deux fonctions f et g en argument, renvoie vrai si f et g sont équivalentes (c'est-à-dire, pour tout e, on a $f(e) = g(e)$) et faux sinon.

- Pour Turing : il n'y a pas de machine de Turing qui, recevant sur son ruban la description d'une machine de Turing M et d'un ruban initial pour cette machine, détermine si l'exécution de M termine ou non. C'est le problème de l'arrêt.

retour sur le problème de la décision

Church et Turing en ont déduit, chacun dans son système, que le problème de la décision ne pouvait être résolu par un algorithme.

Autrement dit, il n'existe pas de programme qui, pour tout énoncé mathématique donné en entrée, termine et renvoie vrai si et seulement si cet énoncé est vrai.

C'est le théorème anti-chômage pour les mathématiciens !

problème indécidable

On dit qu'un problème est indécidable s'il n'existe pas d'algorithme (de fonction calculable, de programme Python, etc.) qui, pour toute entrée, détermine en un temps fini si la réponse est oui ou non.

Exemples :

- le problème de la décision de Hilbert/Ackermann
- le problème de l'arrêt
- le problème « ce programme va-t-il provoquer `ZeroDivisionError` ? »

remarque importante

Les deux conditions « pour toute entrée » et « en un temps fini » sont importantes.

Si on s'autorise à ne pas traiter certaines entrées, alors le problème peut redevenir calculable. Ainsi, on peut imaginer un programme qui signale (correctement) la possibilité de `ZeroDivisionError` dans beaucoup de cas, mais répond « je ne sais pas » de temps en temps.

De même, on peut imaginer un programme qui répond correctement, y compris dans de très nombreux cas, mais parfois ne termine pas. On pourrait tester par exemple toutes les preuves possibles d'un énoncé mathématique successivement par taille croissante !

autre remarque importante

Si les configurations possibles sont en nombre fini, il suffit de les explorer toutes, ce qui permet de répondre à la question en un temps fini.

En théorie, cela s'applique à notre ordinateur et donc à nos programmes.

Mais avec 1 Go de RAM, une machine possède $2^{2^{33}}$ états possibles !

Les résultats d'indécidabilité s'appliquent toujours en pratique.

Exercices