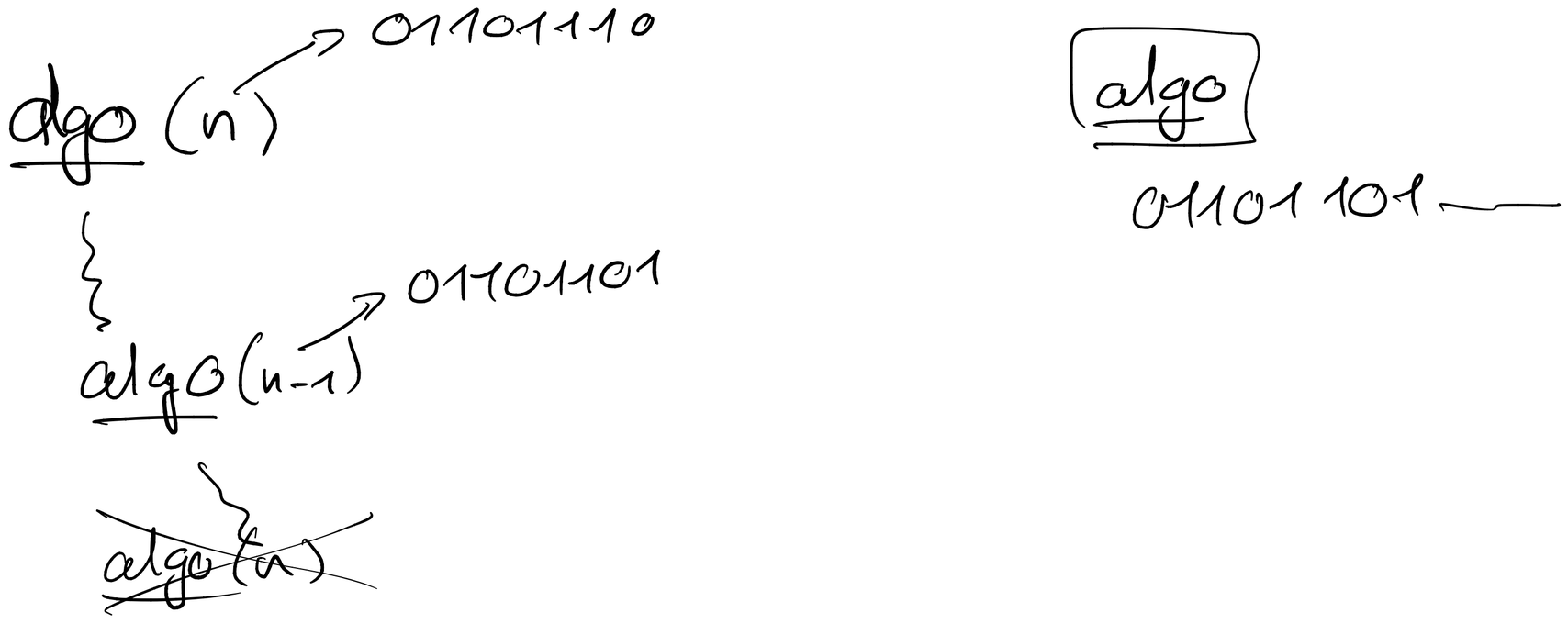
Une note à propos des algorithmes récursifs

- Ils sont très puissants: peu de lignes font beaucoup de choses !
- … mais ils sont aussi difficiles à lire, et à écrire !
- Et au fait, comment ça marche, concrètement, un algorithme récursif ?



Information, Calcul et Communication

Rendu de pièces de monnaie – partie 1

École polytechnique fédérale de Lausanne

Olivier Lévêque

Rendu de pièces de monnaie

Pour rendre une certaine quantité d'argent z, un automate dispose d'un ensemble P de pièces de monnaie, chaque pièce étant disponible en grande quantité.

Exemple: z = 2.80 francs

P = {10 cts, 20 cts, 50 cts, 1 fr, 2 frs}

Le but est de rendre le montant exact avec le moins de pièces possibles. Dans l'exemple ci-dessus, il faudrait donc rendre les pièces:

2 frs, 50 cts, 20 cts et 10 cts

Un algorithme récursif permet de faire ça!

Description informelle de l'algorithme

$P = \{ p_1 \dots p_n \}$

roots 2.–

- si $z = 0$, s'arrêter
- si $z < p_1$, déclarer "rendu exact impossible"
- Si $z < p_n$, relancer l'algorithme avec le même montant mais en s'empêchant d'utiliser la pièce p_n

- rendre la plus grande pièce disponible p_n
  et relancer l'algorithme avec un montant à rendre $z' = z - p_n$

Description informelle de l'algorithme

1. Est-ce que $z = 0$? Si oui, s'arrêter.
2. Est-ce qu'il n'y a plus de pièces à disposition ou est-ce que z est plus petit que la valeur de la plus petite pièce disponible ? Si oui, déclarer que le rendu exact est impossible et s'arrêter.
3. Est-ce que z est plus petit que la valeur de la plus grande pièce disponible ? Si oui, recommencer en 1. en enlevant la possibilité d'utiliser cette plus grande pièce.
4. Rendre la plus grande pièce disponible p et recommencer en 1. en ayant mis à jour la valeur de z à $z - p$.

Algorithmes gloutons

L’algorithme que nous venons de décrire en mots fait partie de la classe des algorithmes dits "**gloutons**" ("**greedy**" en anglais) qui ne remettent jamais en question les décisions prises précédemment.

De tels algorithmes sont généralement économes en temps de calcul (ils vont droit au but !), mais comme nous allons le voir, le prix à payer est qu’ils ne trouvent pas toujours la meilleure solution du problème (ou même ne trouvent pas la solution tout court !).

Description formelle de l'algorithme

Soit : $z =$ le montant à rendre

$P = \{p_1 < p_2 < \cdots < p_n\}$ l'ensemble des pièces disponibles pour le rendu de monnaie

$n =$ le nombre de pièces disponibles

rendu glouton

entrée : z, n
sortie : l'ensemble de pièces de monnaie à rendre

si $z = 0$, sortir $\emptyset$ (ensemble vide)
si $(n = 0$ ou $z < p_1)$, sortir "rendu exact impossible"
si $z < p_n$, sortir **rendu glouton**$(z, n-1)$
Sortir : $\{p_n\} \cup$ **rendu glouton**$(z - p_n, n)$

Exemple

La sortie de cet algorithme (implémenté dans tous les distributeurs de boissons, barres de chocolat et billets de train qu'on peut rencontrer) est donc un ensemble de pièces, éventuellement ponctué d'un message "rendu exact impossible".

Dans notre exemple, on a :

z = 2.80 frs

P = {10 cts, 20 cts, 50 cts, 1 fr, 2 frs}

n = 5 au départ

Dans ce cas, la sortie de l'algorithme est l'ensemble des pièces :

{2 frs, 50 cts, 20 cts, 10 cts}

et le rendu de monnaie est exact (= 2.80 frs).

Schéma d'exécution de l'algorithme

$P = \{0.10, 0.20, 0.50, 1.-, 2.-\}$

$z = 2.80$, $n = 5$

↓ rendre 2.–

$z = 0.80$, $n = 5$

↓ — rien

$z = 0.80$, $n = 4$

↓ — rien

$z = 0.80$, $n = 3$

↓ rendre –.50

$z = 0.30$, $n = 3$

↓ — rien

$z = 0.30$, $n = 2$

↓ rendre –.20

$z = 0.10$, $n = 2$

rien ↳ $z = 0.10$, $n = 1$ → rendre –.10 → $z = 0$, $n = 1$ → s'arrête

rendu glouton

entrée : z, n
sortie : l'ensemble de pièces de monnaie à rendre

si $z = 0$, sortir $\emptyset$ (ensemble vide)
si $n = 0$ ou $z < p_1$, sortir "rendu exact impossible"
si $z < p_n$, sortir **rendu glouton**$(z, n-1)$
Sortir : $\{p_n\} \cup$ **rendu glouton**$(z - p_n, n)$

Mais…

Comme mentionné plus haut, cet algorithme ne trouve pas toujours la solution optimale du problème. Voici deux exemples :

Exemple 1

Gardons $z = 2.80$ frs, mais changeons l'ensemble des pièces:

$P = \{20 \text{ cts}, 50 \text{ cts}, 1 \text{ fr}, 2 \text{ frs}\}$ (et donc $n = 4$).

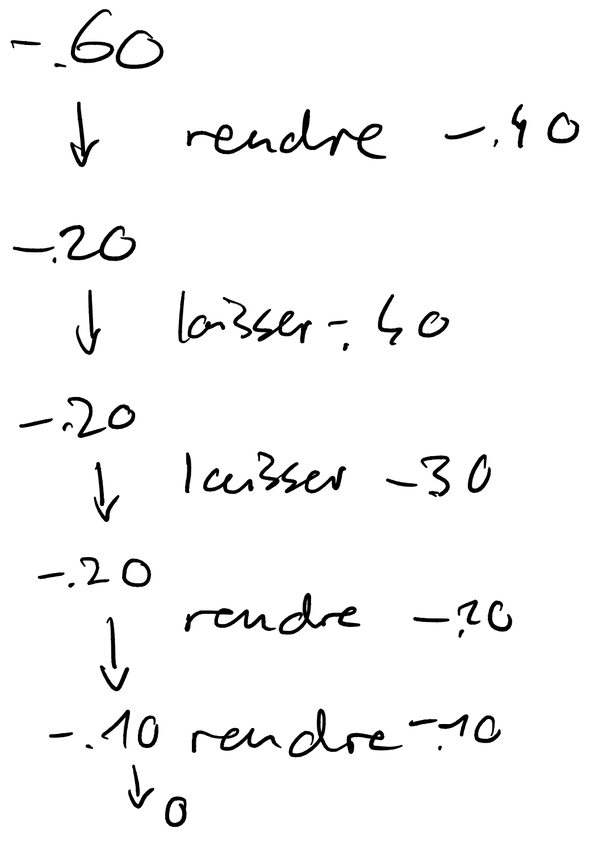
Dans ce cas, l'algorithme glouton rend successivement les pièces {2 frs, 50 cts, 20 cts}, puis affiche le message "rendu exact impossible", alors que c'est faux, puisqu'il aurait été possible de rendre {2 frs, 20 cts, 20 cts, 20 cts, 20 cts} pour obtenir un montent exact.

L'absence de la pièce de 10 cts pose ici un problème à l'algorithme.

Mais...

Exemple 2

Considérons maintenant $z = 60$ cts avec $P = \{10 \text{ cts}, 30 \text{ cts}, 40 \text{ cts}\}$ et $n = 3$.



Mais…

Exemple 2

Considérons maintenant $z = 60$ cts avec $P = \{10 \text{ cts}, 30 \text{ cts}, 40 \text{ cts}\}$ et $n = 3$.

Dans ce cas, l'algorithme glouton rend les pièces {40 cts, 10 cts, 10 cts}, ce qui est certes un rendu exact, mais ne minimise pas le nombre de pièces rendues, car il aurait mieux valu rendre {30 cts, 30 cts}.

Fort heureusement dans la pratique, les ensembles de pièces utilisées suivent souvent le schéma *"1, 2, 5"*, répété à plusieurs échelles. On peut montrer que ces ensembles ont la propriété d'être **canoniques**, ce qui signifie que l'algorithme glouton décrit précédemment trouve toujours la solution optimale du problème !

Information, Calcul et Communication

Rendu de pièces de monnaie – partie 2

École polytechnique fédérale de Lausanne

Olivier Lévêque

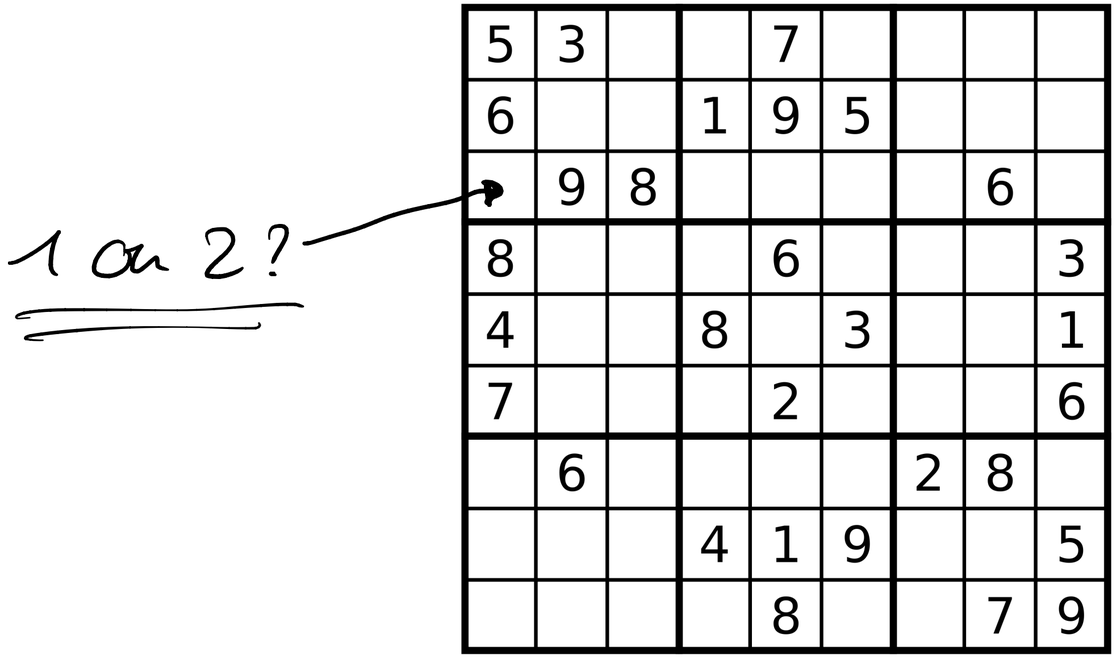
Programmation dynamique

- Jusqu’à présent, nous n'avons vu que des algorithmes récursifs où toutes les opérations effectuées sont nécessaires à la résolution du problème.

- Cependant, dans le cas du rendu de pièces de monnaie, l'algorithme glouton ne trouve pas toujours la meilleure solution du problème, voire pas de solution tout court.

- Pour réparer cela, il est nécessaire de développer un algorithme plus **exploratoire**.

Un exemple simple : résolution d'un sudoku



Pour résoudre un sudoku difficile, on doit parfois essayer deux chiffres différents (ou plus) dans une case donnée et explorer où cela mène, pour finalement conclure que seul un des deux chiffres mène à la solution.

Rappel: rendu glouton de pièces de monnaie

Soit : $z =$ le montant à rendre

$P = \{p_1 < p_2 < \cdots < p_n\}$ l'ensemble des pièces disponibles pour le rendu de monnaie

$n =$ le nombre de pièces disponibles

rendu glouton

entrée : z, n
sortie : l'ensemble de pièces de monnaie à rendre

si $z = 0$, sortir $\emptyset$ (ensemble vide)
si $n = 0$ ou $z < p_1$, sortir "rendu exact impossible"
si $z < p_n$, sortir **rendu glouton**$(z, n-1)$
Sortir : $\{p_n\} \cup$ **rendu glouton**$(z - p_n, n)$

Problème de l'algorithme glouton

Comme nous l'avons vu, l'algorithme glouton ne fonctionne pas toujours correctement si la liste P des pièces de monnaie à disposition n'est pas "standard".

Au lieu de cela, nous désirerions un algorithme capable de :

1. rendre le montant exact (lorsque cela est possible)
2. rendre le nombre minimum de pièces

(en privilégiant toujours le point n° 1)

Problème de l'algorithme glouton

L'erreur de l'algorithme glouton est de toujours choisir "sans réfléchir" la pièce avec la plus grande valeur possible, ce qui mène parfois à des problèmes.

$$\begin{cases} R_1 \leftarrow \{p_n\} \cup \text{rendu}(z - p_n, n) \\ R_2 \leftarrow \text{rendu}(z, n-1) \end{cases}$$

Si $|R_1| < |R_2|$, sortir R_1

("taille de l'ensemble")

Sinon, sortir R_2

Problème de l'algorithme glouton

L'erreur de l'algorithme glouton est de toujours choisir "sans réfléchir" la pièce avec la plus grande valeur possible, ce qui mène parfois à des problèmes.

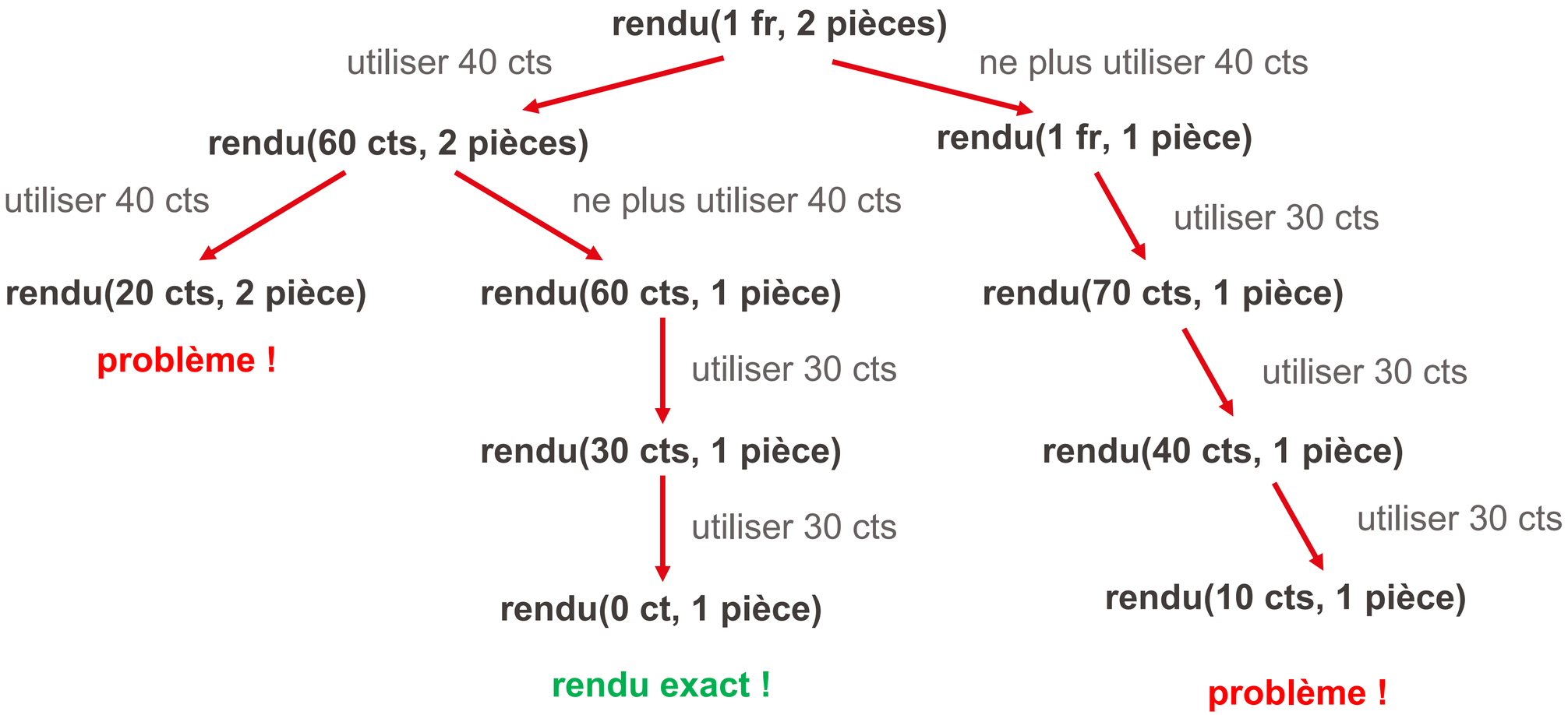
Un algorithme plus circonspect consiste à choisir, à chaque étape, entre **deux options** :

1. rendre la pièce avec la plus grande valeur disponible
2. choisir de ne pas utiliser cette pièce de plus grande valeur (et donc de ne plus l'utiliser dans la suite non plus)

Il importe d'étudier où chacun de ces choix mène (comme pour le sudoku) avant de prendre une décision.

Exemple

$P = \{30 \text{ cts}, 40 \text{ cts}\}$, $n = 2$, $z = 1$ fr



Ecriture formelle de l'algorithme

rendu dynamique

entrée : z, n
sortie : l'ensemble de pièces de monnaie à rendre

si $z = 0$, sortir $\emptyset$ (ensemble vide)
si $n = 0$ ou $z < p_1$, sortir ~~"rendu exact impossible"~~ un ensemble L de 1000 pièces
si $z < p_n$, sortir : **rendu dynamique**$(z, n-1)$
$R_1 \leftarrow \{p_n\} \cup$ **rendu dynamique**$(z - p_n, n)$
$R_2 \leftarrow$ **rendu dynamique**$(z, n-1)$
Si $|R_1| < |R_2|$, sortir : R_1
Sinon, sortir : R_2

Note : $|R|$ désigne la taille l'ensemble R

Mémoïsation

- La programmation dynamique permet une exploration systématique de tous les chemins possibles pour arriver à la solution du problème (si celle-ci existe).

- Cependant, cette façon de faire a un gros défaut : à chaque étape, il faut choisir entre 2 chemins : si chaque chemin est de longueur n, le nombre de chemins à explorer est donc de l'ordre de 2^n

  → **temps de calcul prohibitif !**

- De plus, beaucoup de calculs sont répétés **inutilement** lors de l'exploration.

- Une solution : mémoriser les calculs effectués au fur et à mesure

  → **réduction du temps de calcul**

Information, Calcul et Communication

La théorie de la calculabilité et le problème de l'arrêt

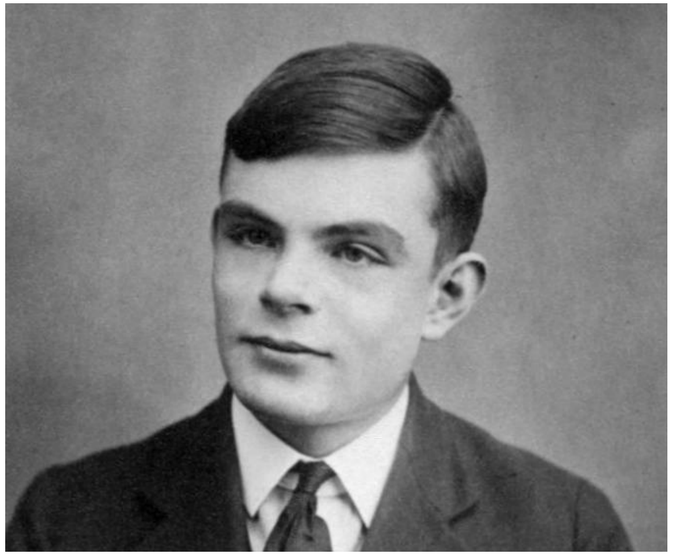
Olivier Lévêque

École polytechnique fédérale de Lausanne

Introduction à la théorie de la calculabilité

Question : Tout problème est-il soluble par un algorithme ?

Réponse : Non ! (Alan Turing, 1936)



Pour bien comprendre cette question, on doit d'abord définir ce qu'on entend par "problème".

Un problème est un ensemble de questions

- **Exemple :** Combien de musées trouve-t-on dans chaque ville de Suisse ?

  Algorithme de résolution: aller consulter la liste des musées de chaque ville et compter à chaque fois le nombre de ceux-ci
  → Genève : 26, Lausanne : 23, etc.

- Mais le nombre de villes en Suisse est un nombre fini !
  On peut donc établir *une fois pour toutes* une **table de correspondance :**

Ville	Genève	Lausanne	…
Nombre de musées	26	23	…

Après ça, **plus besoin d'algorithme** pour résoudre ce problème !

Un autre exemple de problème

- Etant donné un nombre entier positif N, celui-ci est-il un nombre premier ? (c'est-à-dire un nombre admettant exactement deux diviseurs distincts : 1 et le nombre en lui-même)

  Ex: si $N = 7$, alors la réponse est oui ; si $N = 8$, alors la réponse est non.

- Ce problème a un **nombre infini d'instances**. On ne peut donc pas établir une fois pour toutes une table de correspondance.

- Pour autant, existe-t-il un algorithme qui permette de le résoudre ?

  Oui : tester tous les nombres entre 2 et $N - 1$; si aucun ne divise N (et si N est différent de 1), alors N est premier.

Problèmes de décision

- Le problème précédent est un **problème de décision**, qui ne demande qu'une réponse "oui" ou "non" pour chaque valeur de N.

- Turing (1936) :

  "Il existe des problèmes de décision qu'il est impossible de résoudre au moyen d'un algorithme ; ces problèmes sont donc **indécidables**."

- Exemple : le **problème de l'arrêt**.

Le problème de l'arrêt

"Etant donné un algorithme P prenant en entrée des données X,
sait-on si l'algorithme $P(X)$ s'exécute en un temps fini ou non ?"

- Evidemment, pour certains algorithmes P et certaines données d'entrée X, la réponse est connue!

- Plus précisément:

"Existe-t-il un algorithme A prenant en entrée un autre algorithme P
et des données X, et dont la sortie soit **oui** si $P(X)$ s'exécute en
un temps fini, et **non** dans le cas contraire?"

- Ce que Turing démontre en 1936, c'est qu'un tel algorithme A **n'existe pas** !

Démonstration (par l'absurde)

Supposons qu'un tel algorithme A existe, c'est-à-dire :

- $A(P, X)$ sort *oui* si $P(X)$ s'arrête
- $A(P, X)$ sort *non* si $P(X)$ continue indéfiniment

A partir de cet algorithme A, on construit un autre algorithme B :

algorithme B

entrée : algorithme P
sortie : aucune

Si $A(P, P) = oui$, alors : effectuer une boucle infinie
Sinon : s'arrêter

Démonstration (par l'absurde)

Que se passe-t-il si on exécute l'algorithme B avec **lui-même** en entrée ?
En d'autres termes, que fait $B(B)$?

- $A(B, B) = oui$?

  Si oui, B effectue une boucle infinie

  ie. B(B) s'arrête

  Si non B s'arrête

  ie. B(B) effectue une boucle infinie

algorithme B
entrée : algorithme P sortie : aucune
Si $A(P, P) = oui$, alors : effectuer une boucle infinie Sinon : s'arrêter

Démonstration (par l'absurde)

Que se passe-t-il si on exécute l'algorithme B avec **lui-même** en entrée ?
En d'autres termes, que fait $B(B)$?

- si $A(B,B) = \textit{oui}$ (i.e., si $B(B)$ s'arrête), alors $B(B)$ effectue une boucle infinie ?
- si $A(B,B) = \textit{non}$ (i.e., si $B(B)$ continue indéfiniment), alors $B(B)$ s'arrête ?

algorithme B
entrée : algorithme P sortie : aucune
Si $A(P,P) = \textit{oui}$, alors : effectuer une boucle infinie Sinon : s'arrêter

Dans les deux cas, on a clairement une contradiction !

Conclusion : L'hypothèse effectuée (l'algorithme A existe) est donc fausse. #