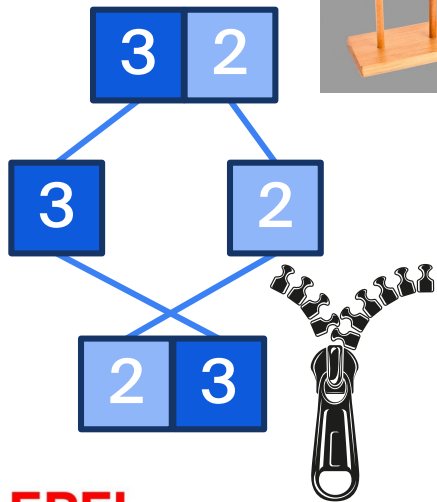


Information, Calcul et Communication

CS-119(g) ICC – Théorie Semaine 12

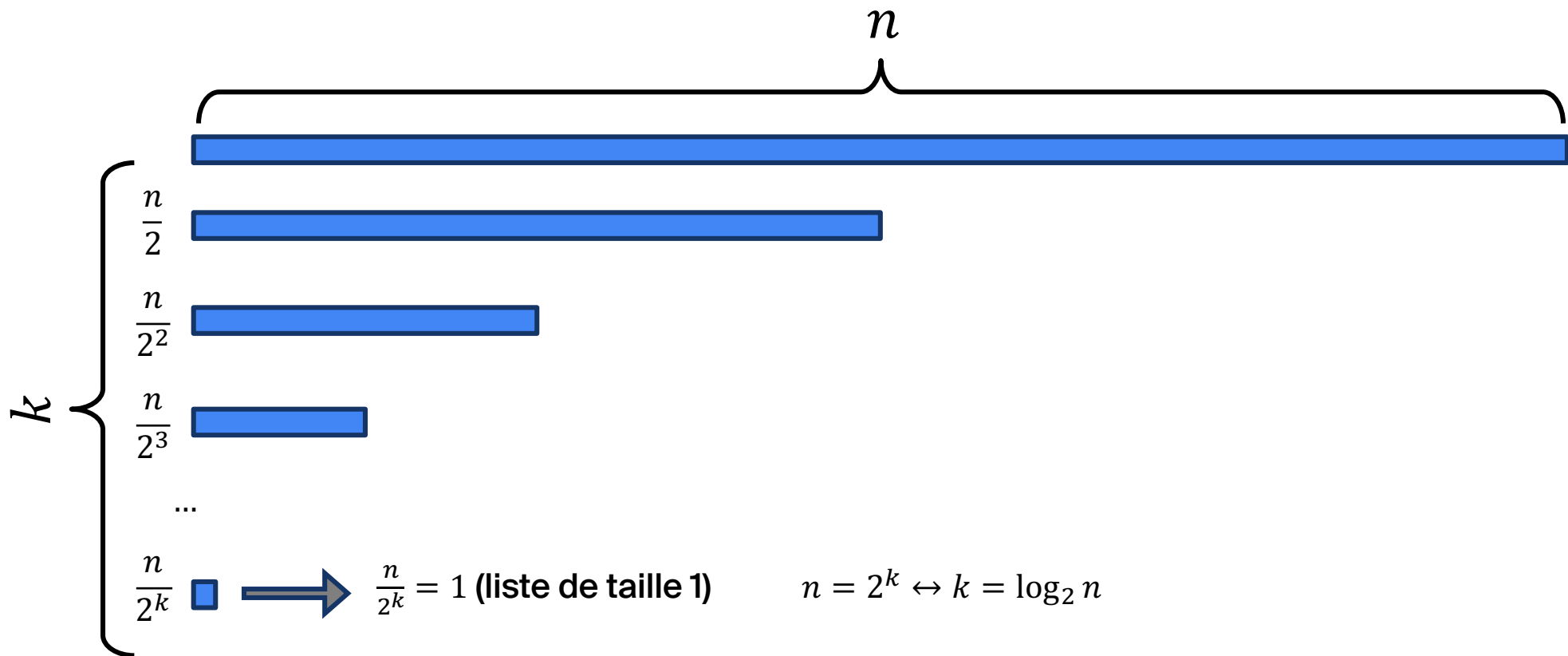
Rafael Pires
rafael.pires@epfl.ch

Précédemment, dans ICC-T 11



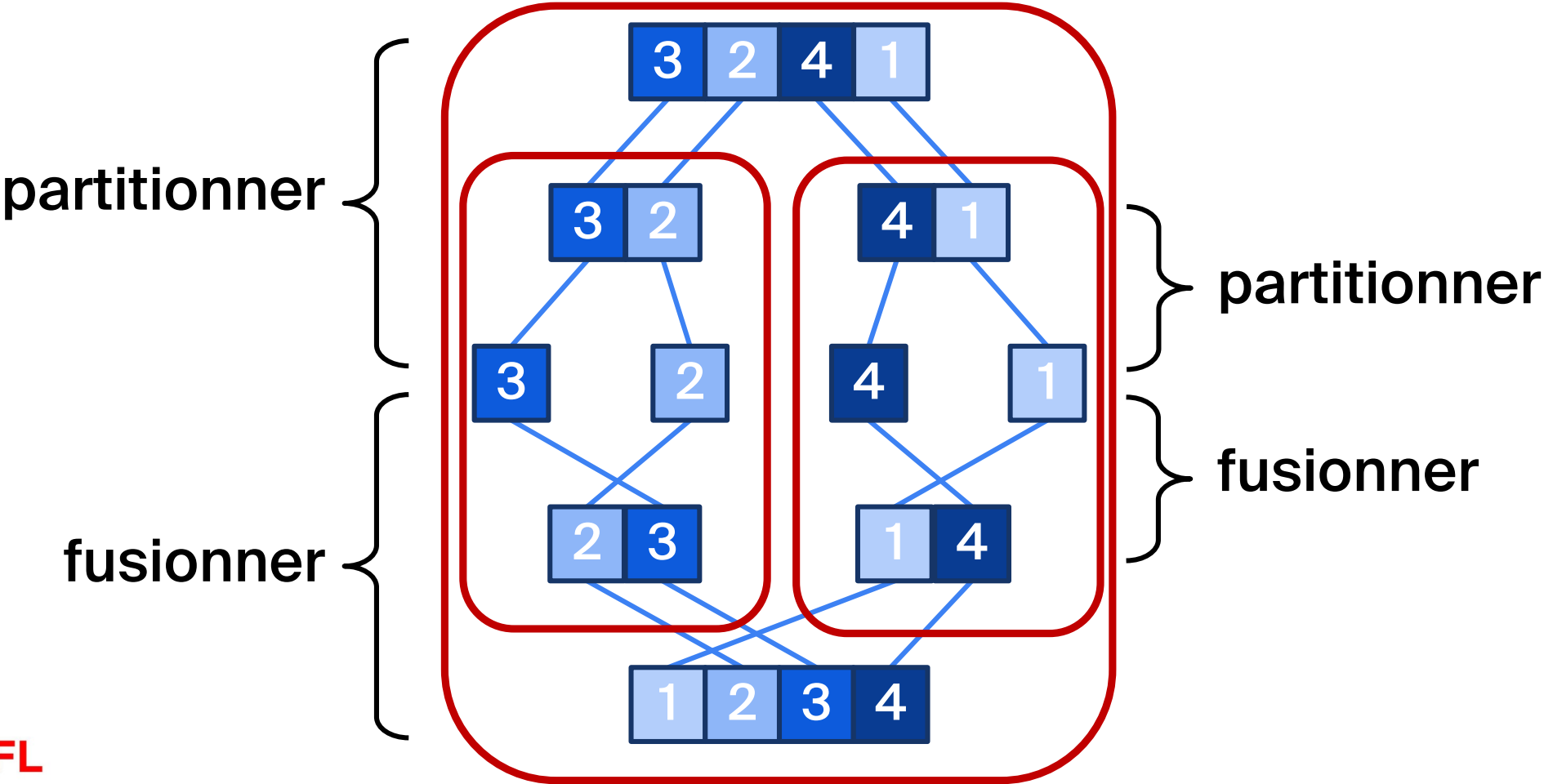
- La récursivité
 - Factorielle - 1 appel récursif
 - Suite de Fibonacci - 2 appels récursifs
 - Tours de Hanoi - 3 appels récursifs
- Le logarithme dans la complexité
 - Recherche dichotomique - $\Theta(\log_2 n)$
 - Tri par fusion - $\Theta(n \log_2 n)$

Complexité temporelle : Recherche dichotomique



Complexité : $\Theta(\log_2 n)$

Tri par fusion



Aujourd'hui

- **Algorithmes gloutons**
- **Programmation dynamique**
- **Memoisation**
- **Introduction à la théorie de la calculabilité**

Algorithmes gloutons : 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

$P = \{ 0.10, 0.20, 0.50, 1.0, 2.0 \}$

- 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 :

2.0, 0.50, 0.20 et 0.10

- Un algorithme récursif permet de faire ça !

Algorithmes gloutons : Rendu de pièces de monnaie



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 :
 - ils ne trouvent pas toujours **la meilleure solution**
 - ou ils ne trouvent **pas de solution**, tout court !



Algorithmes gloutons : Rendu de pièces de monnaie



Rendu glouton

entrée : z le montant à rendre

$p = \{p_1 < p_2 < \dots < p_n\}$ l'ensemble des pièces disponibles

sortie : l'ensemble des pièces à rendre

Si $z = 0$

Sortir : \emptyset (ensemble vide)

Si $p = \emptyset$ ou $z < p_1$

Sortir : « Rendu exact impossible »

Si $z < p_n$

Sortir : **Rendu glouton**($z, p \setminus \{p_n\}$) (enlève p_n des pièces disponibles)

Sortir : $\{p_n\} \cup$ **Rendu glouton**($z - p_n, p$)

Algorithmes gloutons : Rendu de pièces de monnaie

Rendu glouton

entrée : $z = 2.80$

$p = \{0.1, 0.2, 0.5, 1., 2.\}$

sortie : *rendu*

Si $z = 0$

Sortir : \emptyset

Si $p = \emptyset$ ou $z < p_1$

Sortir : « Rendu exact impossible »

Si $z < p_n$

Sortir : $\text{Rendu glouton}(z, p \setminus \{p_n\})$

Sortir : $\{p_n\} \cup \text{Rendu glouton}(z - p_n, p)$

Algorithmes gloutons : Rendu de pièces de monnaie



Rendu glouton

entrée : z le montant à rendre

$p = \{p_1 < p_2 < \dots < p_n\}$ l'ensemble des pièces disponibles

sortie : l'ensemble des pièces à rendre

Si $z = 0$

Sortir : \emptyset (ensemble vide)

Si $p = \emptyset$ ou $z < p_1$

Sortir : « Rendu exact impossible »

Si $z < p_n$

Sortir : **Rendu glouton**($z, p \setminus \{p_n\}$) (enlève p_n des pièces disponibles)

Sortir : $\{p_n\} \cup$ **Rendu glouton**($z - p_n, p$)

Algorithmes gloutons

Pas de solution,
tout court !



Rendu glouton

entrée : $z = 2.80$

$p = \{0.2, 0.5, 1.0, 2.0\}$

sortie : ?

Si $z = 0$

Sortir : \emptyset

Si $p = \emptyset$ ou $z < p_1$

Sortir : « Rendu exact impossible »

Si $z < p_n$

Sortir : $\text{Rendu glouton}(z, p \setminus \{p_n\})$

Sortir : $\{p_n\} \cup \text{Rendu glouton}(z - p_n, p)$

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

$P = \{0.20, 0.50, 1.0, 2.0\}$

Résultat :

$\{2.0, 0.50, 0.20\}$ et ensuite « Rendu exact impossible »

Alors que c'est **faux**, puisqu'il aurait été possible de rendre

$\{2.0, 0.20, 0.20, 0.20, 0.20\}$

pour obtenir un montant exact.



Algorithmes gloutons

Pas toujours la solution optimale du problème



Rendu glouton

entrée : $z = 0.60$

$p = \{ 0.10, 0.30, 0.40 \}$

sortie : ?

Si $z = 0$

Sortir : \emptyset

Si $p = \emptyset$ ou $z < p_1$

Sortir : « Rendu exact impossible »

Si $z < p_n$

Sortir : $\text{Rendu glouton}(z, p \setminus \{ p_n \})$

Sortir : $\{ p_n \} \cup \text{Rendu glouton}(z - p_n, p)$

Considérons maintenant $z = 0.60$, avec l'ensemble des pièces:

$$P = \{ 0.10, 0.30, 0.40 \}$$

Résultat :

$$\{ 0.40, 0.10, 0.10 \}$$

Ce qui est certes un rendu exact, mais **ne minimise pas le nombre de pièces rendues**, car il aurait mieux valu rendre :

$$\{ 0.30, 0.30 \}$$

Heureusement, dans la pratique, les ensembles de pièces utilisés suivent souvent le schéma « 1, 2, 5 », répété à plusieurs échelles. On peut démontrer que ces ensembles possèdent la propriété d'être canoniques, ce qui signifie que l'algorithme glouton décrit précédemment trouve toujours la solution optimale dans ces cas !

Algorithmes gloutons : l'essentiel



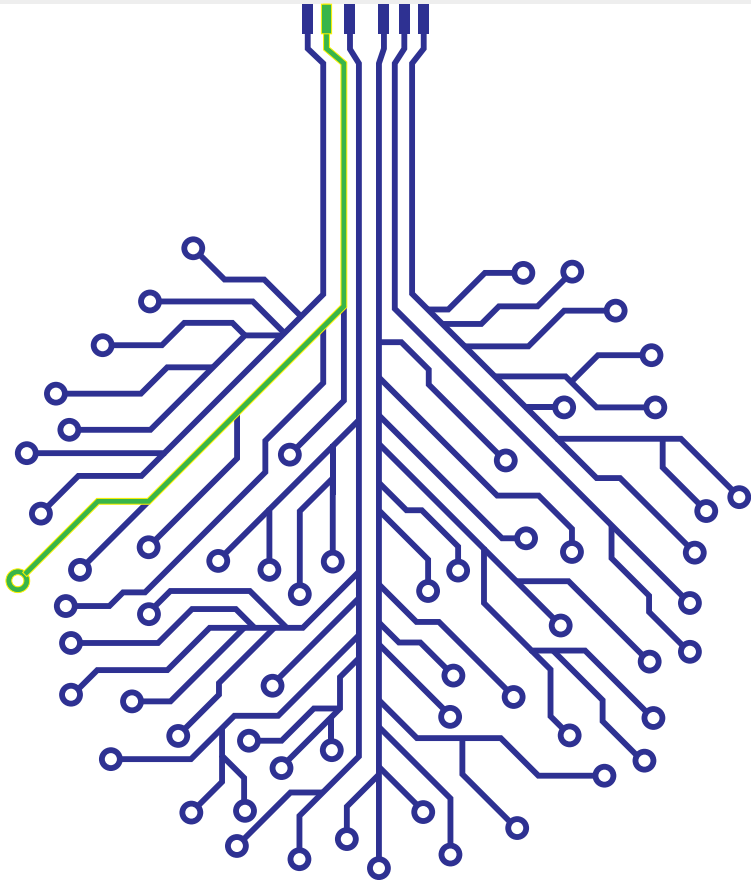
1. Ne remettent jamais en question les **décisions** prises précédemment.
2. Économies en temps de calcul
3. Mais :
 - ils ne trouvent pas toujours **la meilleure solution**
 - parfois ils ne trouvent **pas de solution**



Aujourd'hui

- Algorithmes gloutons
- **Programmation dynamique**
- Memoisation
- Introduction à la théorie de la calculabilité

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

Programmation dynamique :

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

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

Programmation dynamique :

Rendu de pièces de monnaie



- 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 »
- Nous souhaitons plutôt un algorithme capable de :
 1. Rendre le montant exact, lorsque c'est possible.
 2. Minimiser le nombre de pièces utilisées.
(en donnant toujours la priorité au premier critère)

Programmation dynamique :

Rendu de pièces de monnaie



- L'erreur de l'algorithme glouton est de toujours choisir, *sans réflexion*, la pièce de plus grande valeur possible, ce qui *mène parfois à des problèmes*.
- Un algorithme plus prudent consiste à envisager, à chaque étape, deux options :
 1. Utiliser la pièce de plus grande valeur disponible
 2. Ne pas utiliser cette pièce et l'exclure également des choix futurs

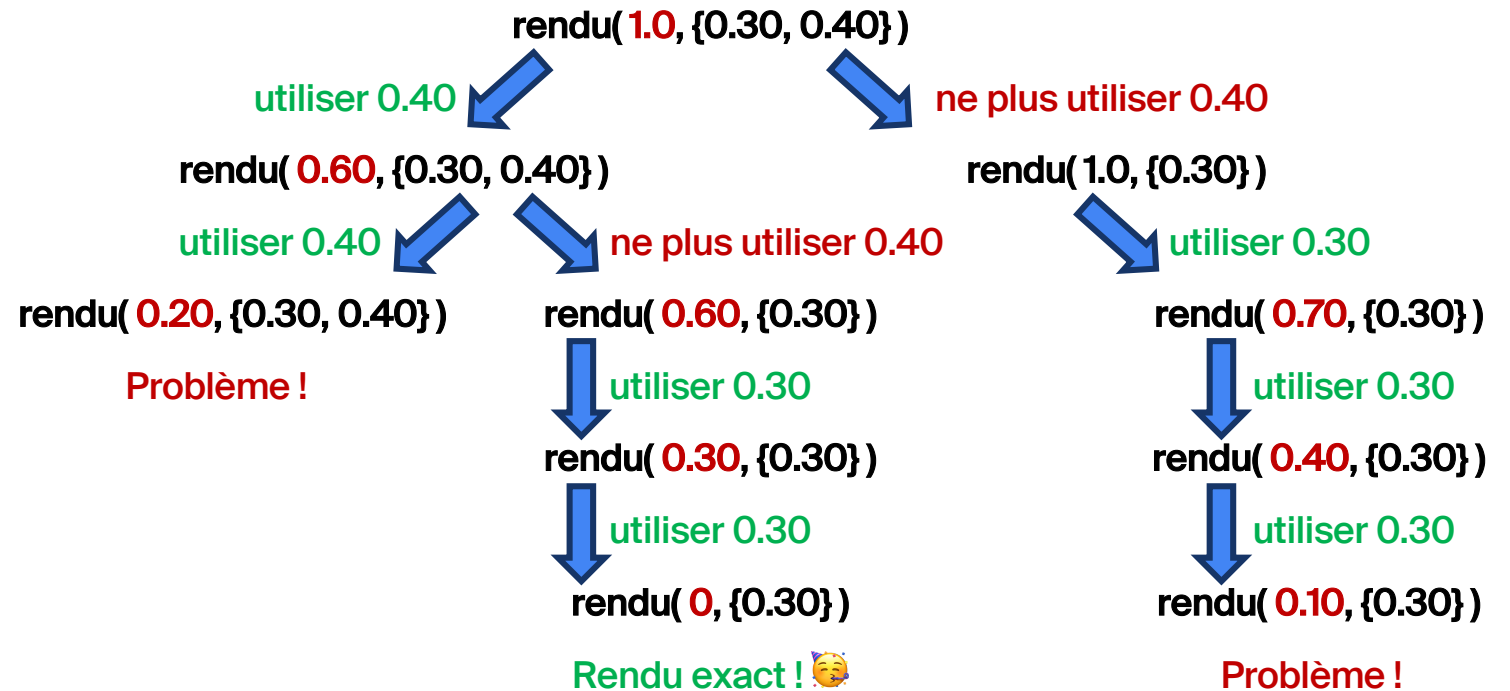
Avant de prendre une décision, il est crucial d'examiner les conséquences de chaque choix (comme dans le Sudoku)

Programmation dynamique : Rendu de pièces de monnaie



Exemple :

$$P = \{ 0.30, 0.40 \}, z = 1.0$$



Programmation dynamique :

Rendu de pièces de monnaie



Rendu dynamique

entrée : z le montant à rendre

$p = \{p_1 < p_2 < \dots < p_n\}$ l'ensemble des pièces disponibles

sortie : l'ensemble des pièces à rendre

Si $z = 0$

Sortir : \emptyset (ensemble vide)

Si $p = \emptyset$ ou $z < p_1$

Sortir : E (ensemble d'une taille arbitrairement grande)

Si $z < p_n$

Sortir : **Rendu dynamique**($z, p \setminus \{p_n\}$) (enlève p_n des pièces disponibles)

$R_1 \leftarrow \{p_n\} \cup$ **Rendu dynamique**($z - p_n, p$)

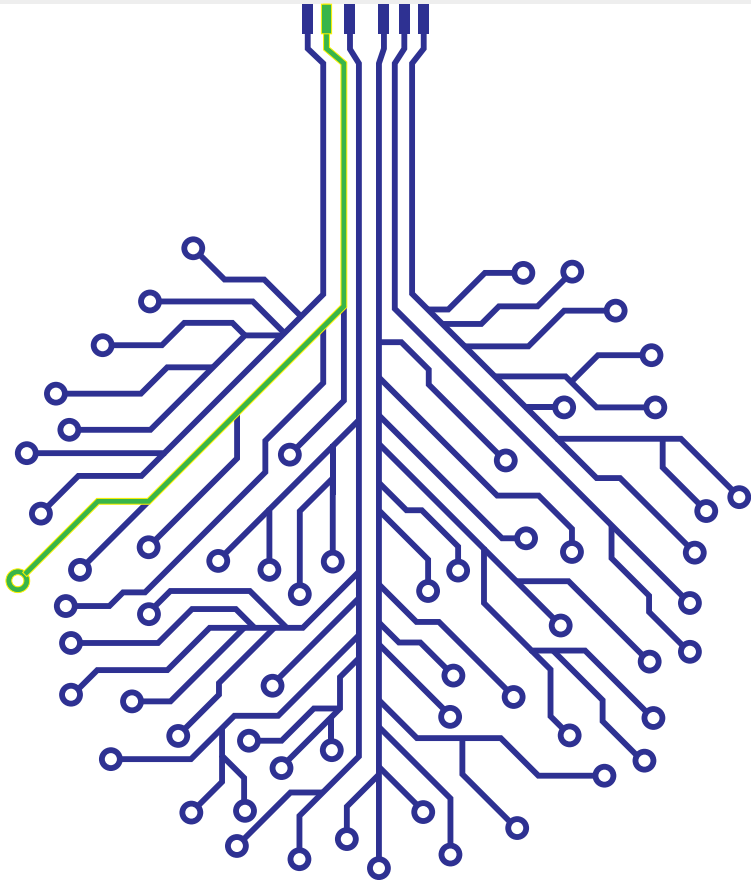
$R_2 \leftarrow$ **Rendu dynamique**($z, p \setminus \{p_n\}$)

Si $|R_1| < |R_2|$, Sortir : R_1

Sinon, Sortir : R_2

$|R|$ désigne la taille de l'ensemble R

Programmation dynamique : l'essentiel



1. Explore **plusieurs solutions possibles** en décomposant le problème en sous-problèmes plus petits
2. Exploite les **liens entre ces sous-problèmes** pour éviter de tout recalculer
3. Explorer toutes les possibilités peut être très **coûteux**



Aujourd'hui

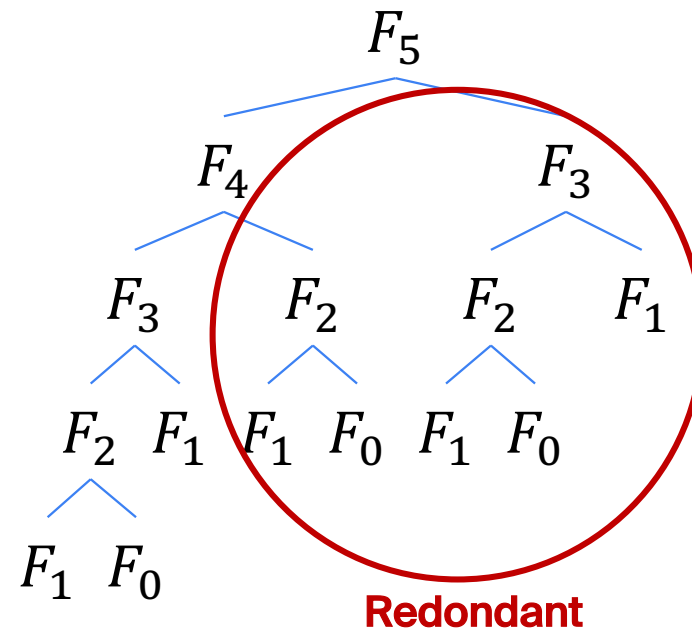
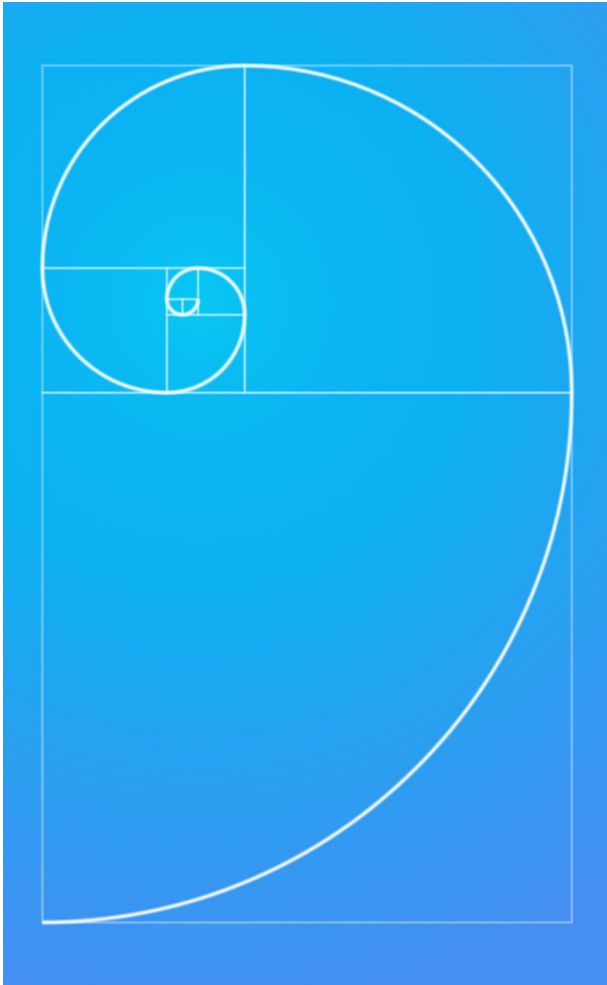
- Algorithmes gloutons
- Programmation dynamique
- **Memoisation**
- Introduction à la théorie de la calculabilité

Memoisation



- 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

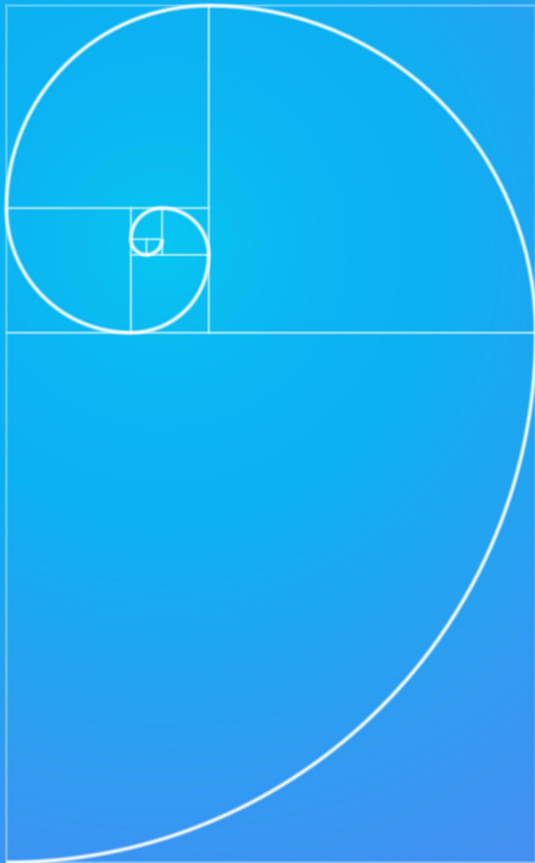
ICC-T 11 : Fibonacci



Complexité :

$$\Theta(\phi^n)$$

Fibonacci : memoisation



$\Theta(\phi^n)$

Fibonacci naïve

entrée : entier naturel **n**

sortie : nième nombre dans la suite de Fibonacci

Si $n \leq 1$

Sortir : 1

Sinon

Sortir : **Fibonacci naïve**($n - 1$) + **Fibonacci naïve**($n - 2$)

$\Theta(n)$

Fibonacci

entrée : entier naturel **n**, liste des résultats **memo**

sortie : nième nombre dans la suite de Fibonacci

Si $n \leq 1$

Sortir : 1

Si **n** est dans **memo**

Sortir : **memo**[**n**]

Sinon

memo[**n**] \leftarrow **Fibonacci**($n - 1$, **memo**) + **Fibonacci**($n - 2$, **memo**)

Sortir : **memo**[**n**]

Memoisation : l'essentiel



1. Technique qui consiste à **enregistrer les résultats intermédiaires** pour éviter les calculs inutiles
2. Très utile pour améliorer l'efficacité des algorithmes avec des **sous-problèmes récurrents**
3. Peut être **coûteuse en mémoire** si trop de résultats intermédiaires sont stockés



Aujourd'hui

- Algorithmes gloutons
- Programmation dynamique
- Memoisation
- **Introduction à la théorie de la calculabilité**

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 : 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 problème avec un nombre infini d'instances



- 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



Cette assertion est **fausse.**

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 !

Le problème de l'arrêt



- Supposons, par hypothèse, 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

P
entrée : entier x sortie : aucune
pas ← 0 Tant que pas ≠ x : pas ← pas + 1

❖ **A(P, 5)**

oui

❖ **A(P, -2)**

non

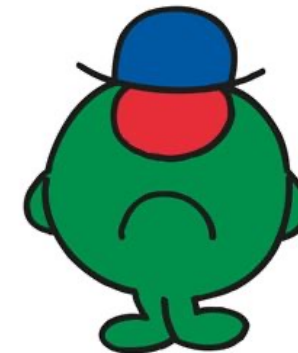
Le problème de l'arrêt



- Supposons, par hypothèse, 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 qu'on appelle « **M. Non** » :

M. Non
entrée : algorithme P
sortie : aucune
Si A(P, P) = oui, alors : Effectue une boucle infinie
Sinon S'arrêter

M. NON



Le problème de l'arrêt : démonstration par l'absurde

M. Non

entrée : algorithme **P**
sortie : aucune

Si $A(P, P) = \text{oui}$, alors :
Effectue une boucle infinie
Sinon
S'arrêter



- Que se passe-t-il si on exécute l'algorithme « **M. Non** » avec lui-même en entrée ?
 - En d'autres termes, que fait **M. Non(M. Non)** ?
- Si $A(\text{M. Non}, \text{M. Non}) = \text{oui}$ (i.e., si **M. Non(M. Non)** s'arrête d'après A), alors **M. Non(M. Non)** effectue une boucle infinie.
- Si $A(\text{M. Non}, \text{M. Non}) = \text{non}$ (i.e., si **M. Non(M. Non)** continue indéfiniment d'après A), alors **M. Non(M. Non)** s'arrête.
- Dans les deux cas, on a clairement une **contradiction** !
- **Conclusion** :
L'hypothèse effectuée (l'**algorithme A existe**) est donc **fausse**.

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



1. Certains problèmes sont fondamentalement **indécidables** : aucun algorithme ne peut toujours y répondre correctement.
2. Le **problème de l'arrêt** en est l'exemple le plus célèbre.
3. Cette limite démontre que l'informatique a des frontières **qu'aucun algorithme ne peut franchir.**



Aujourd'hui

- Algorithmes gloutons
- Programmation dynamique
- Memoisation
- Introduction à la théorie de la calculabilité

Résumé Cours 12 – ICC-T

- **Les algorithmes gloutons** prennent des décisions immédiates sans retour en arrière : ils sont rapides mais peuvent échouer à trouver la solution optimale.
- **La programmation dynamique** explore de nombreuses possibilités et nécessite une structure adaptée pour éviter les calculs redondants.
- **La mémorisation** optimise les algorithmes en stockant des résultats intermédiaires.
- Certains problèmes sont **fondamentalement insolubles** : le problème de l'arrêt montre qu'il existe des questions qu'aucun algorithme ne peut trancher avec certitude.

rafael.pires@epfl.ch



EPFL

Merci