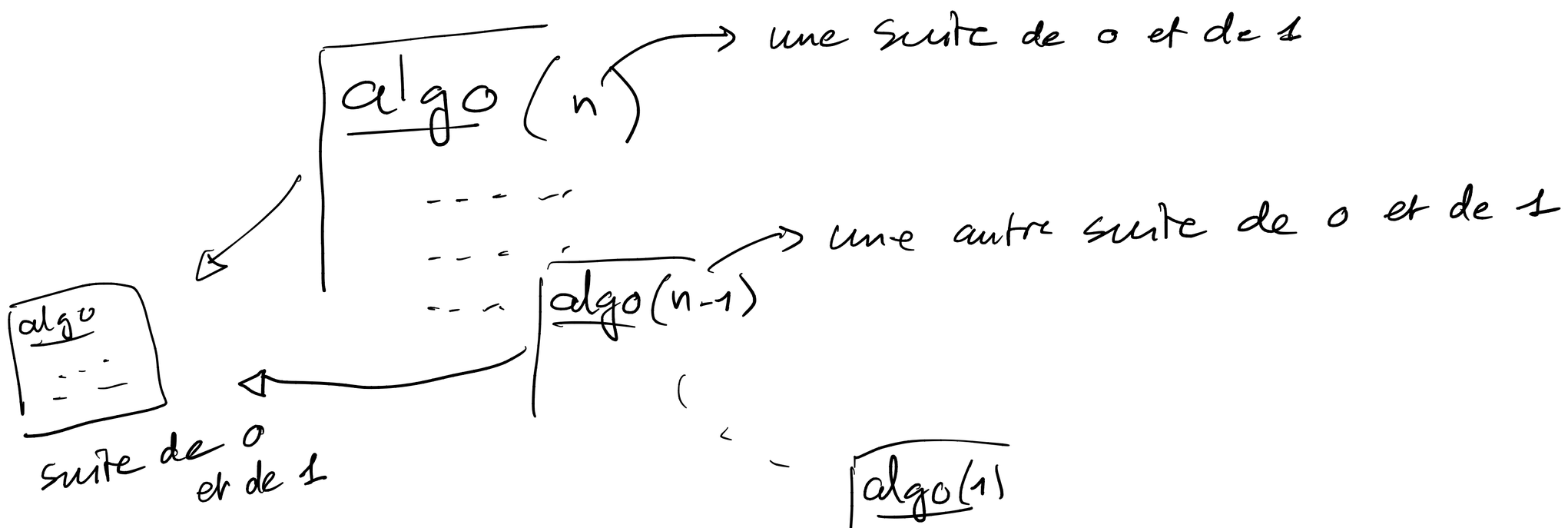
Une note à propos des algorithmes récursifs

- 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

Tri par fusion

École polytechnique fédérale de Lausanne

Olivier Lévêque

Tri d'une liste de nombres

Ce problème, omniprésent en informatique, à été résolu de milles façons !

La semaine dernière, nous avons vu le **tri par insertion**, mais il existe aussi :

- Le tri à bulles
- Le tri à cocktail
- Le tri à peigne
- Le tri pair-impair
- Le tri par sélection
- Le tri par tas
- Le tri rapide
- Le tri stupide (!)
- …

Aujourd'hui, nous allons voir un tri récursif : le **tri par fusion**, qui permet une résolution plus rapide du problème que le tri par insertion.

Tri par fusion

Soit L une liste non-triée de nombres entiers, de taille n.
On aimerait trier cette liste dans l'ordre croissant.

tri par fusion

entrée : Liste L non-triée de nombres entiers, de taille n
sortie : Liste L' triée

Si $n = 1$, sortir : L
$m \leftarrow \lfloor n/2 \rfloor$
$L_1 \leftarrow$ **tri par fusion**$(L(1:m), m)$
$L_2 \leftarrow$ **tri par fusion**$(L(m+1:n), n-m)$
$L' \leftarrow$ **fusion**(L_1, L_2)
Sortir : L'

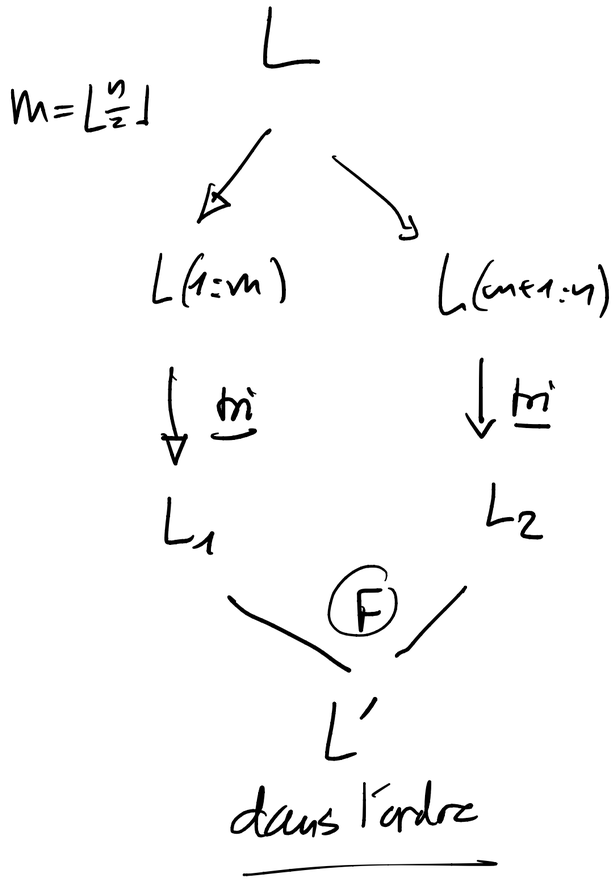


Schéma d'exécution de l'algorithme

Avec $L = (9,3,5,4)$ et $n = 4$:

tri fusion (L, n)

n=1 ? non

m ← 2

L_1 ← tri fusion ((9,3), 2)

L_2 ← tri fusion ((5,4), 2)

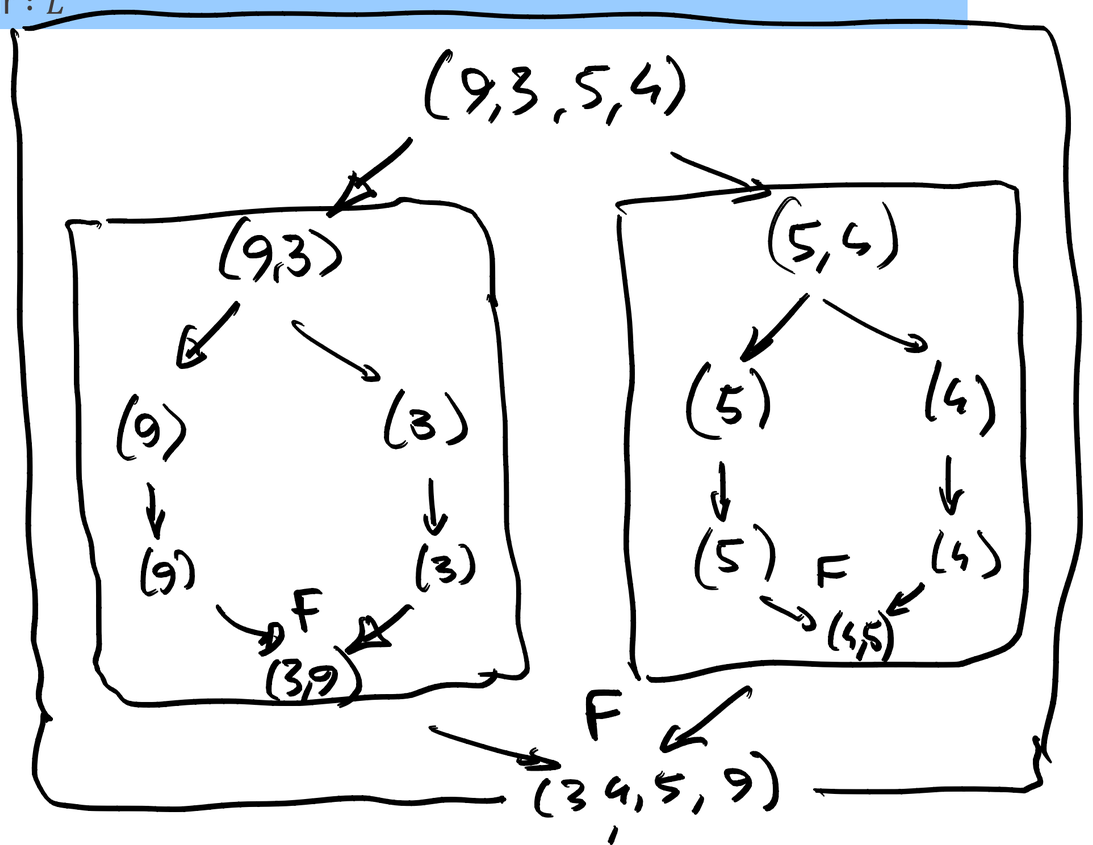
L' ← fusion (L_1, L_2)

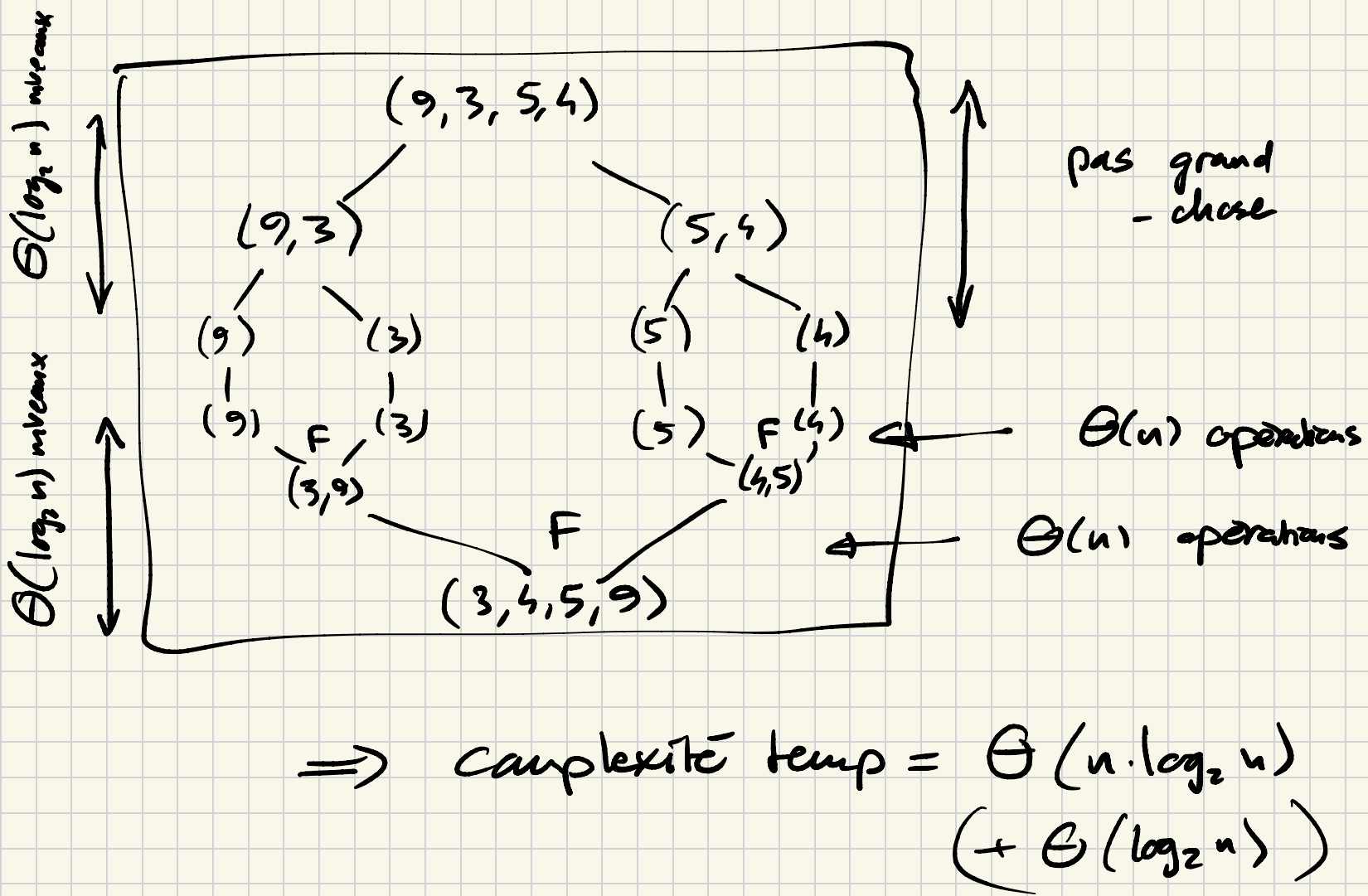
Sortir L'

tri par fusion

entrée : Liste L non-triée de nombres entiers, de taille n
sortie : Liste L' triée

Si $n = 1$, sortir : L
$m \leftarrow \lfloor n/2 \rfloor$
$L_1 \leftarrow$ **tri par fusion**$(L(1:m), m)$
$L_2 \leftarrow$ **tri par fusion**$(L(m+1:n), n-m)$
$L' \leftarrow$ **fusion**(L_1, L_2)
Sortir : L'





$\Rightarrow$ complexité temp = $\Theta(n \cdot \log_2 n)$

$(+ \Theta(\log_2 n))$

Complexité temporelle de l'algorithme

- Comme pour la recherche par dichotomie, (approx.) $\log_2(n)$ niveaux sont nécessaires pour arriver à des listes de taille 1.

- A chaque, niveau, il faut fusionner n éléments, ce que se fait en $\Theta(n)$ opérations (comme nous allons le voir).

- → complexité temporelle $\Theta(n \cdot \log_2(\mathrm{n}))$:
  plus efficace que le tri par insertion (dans le pire des cas)

$\sim \log_2(n!)$

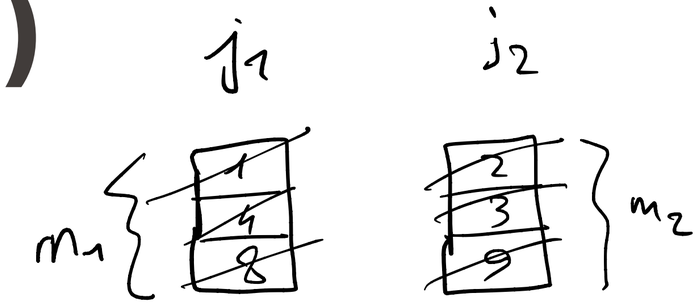
Algorithme fusion ("fermeture éclair")

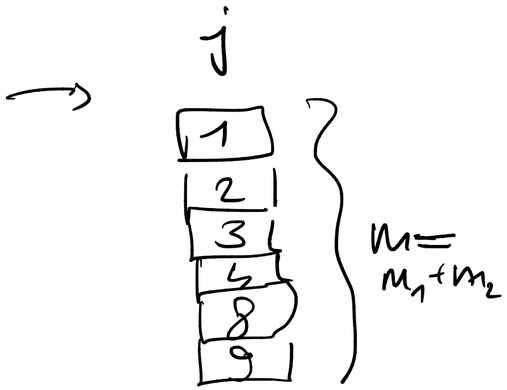
Fusion

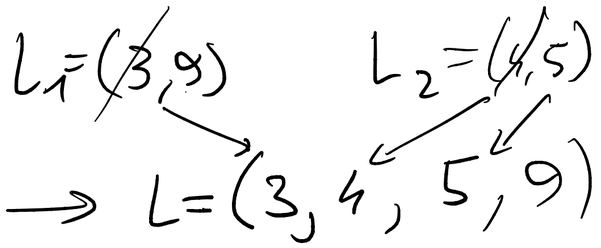
entrée : Listes ordonnées L_1, L_2 (de tailles m_1 et m_2 resp.)
sortie : Liste L (de taille $m_1 + m_2$, également ordonnée)

$j_1 \leftarrow 1$
$j_2 \leftarrow 1$
$j \leftarrow 1$
Tant que $j_1 \leq m_1$ et $j_2 \leq m_2$:
    Si $L_1(j_1) \leq L_2(j_2)$:
        $L(j) \leftarrow L_1(j_1)$
        $j_1 \leftarrow j_1 + 1$
        $j \leftarrow j + 1$
    Sinon :
        $L(j) \leftarrow L_2(j_2)$
        $j_2 \leftarrow j_2 + 1$
        $j \leftarrow j + 1$

Si $j_1 = m_1 + 1$:
    Tant que $j_2 \leq m_2$:
        $L(j) \leftarrow L_2(j_2)$
        $j_2 \leftarrow j_2 + 1$
        $j \leftarrow j + 1$
Sinon :
    Tant que $j_1 \leq m_1$:
        $L(j) \leftarrow L_1(j_1)$
        $j_1 \leftarrow j_1 + 1$
        $j \leftarrow j + 1$
Sortir : L







Algorithme fusion ("fermeture éclair")

Fusion

entrée : Listes ordonnées L_1, L_2 (de tailles m_1 et m_2 resp.)
sortie : Liste L (de taille $m_1 + m_2$, également ordonnée)

$j_1 \leftarrow 1$
$j_2 \leftarrow 1$
$j \leftarrow 1$
Tant que $j_1 \leq m_1$ et $j_2 \leq m_2$:
    Si $L_1(j_1) \leq L_2(j_2)$:
        $L(j) \leftarrow L_1(j_1)$
        $j_1 \leftarrow j_1 + 1$
        $j \leftarrow j + 1$
    Sinon :
        $L(j) \leftarrow L_2(j_2)$
        $j_2 \leftarrow j_2 + 1$
        $j \leftarrow j + 1$

Si $j_1 = m_1 + 1$:
    Tant que $j_2 \leq m_2$:
        $L(j) \leftarrow L_2(j_2)$
        $j_2 \leftarrow j_2 + 1$
        $j \leftarrow j + 1$
Sinon :
    Tant que $j_1 \leq m_1$:
        $L(j) \leftarrow L_1(j_1)$
        $j_1 \leftarrow j_1 + 1$
        $j \leftarrow j + 1$
Sortir : L

Complexité temporelle
$\Theta(m_1 + m_2)$

Information, Calcul et Communication

Rendu de pièces de monnaie – partie 1

École polytechnique fédérale de Lausanne

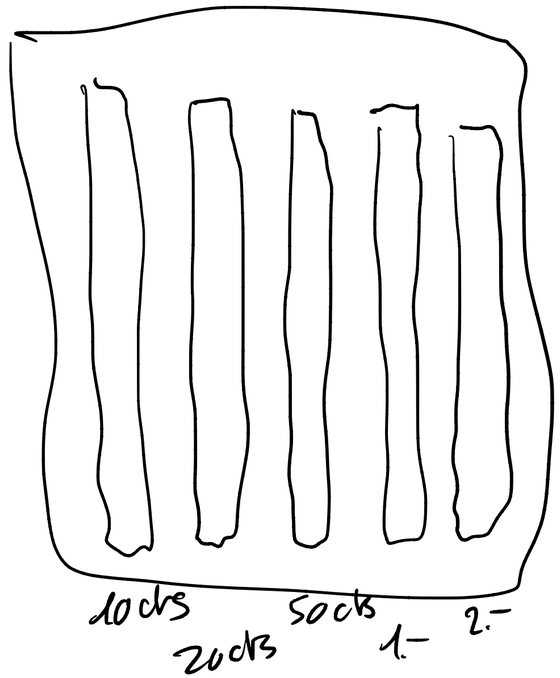
Olivier Lévêque

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}



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

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 $z - p$.

$z = 2.80 \longrightarrow p = 2.- \longrightarrow z = 0.80$

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

fixé

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

$n =$ le nombre de pièces disponibles

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

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

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

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**$(z - p_n, n)$

$z = 2.80,\ n = 5$

↓ rendre 2.–

$z = 0.80,\ n = 5$

↓ laisser tomber les 2.–

$z = 0.80,\ n = 4$

↓ laisser tomber les 1.–

$z = 0.80,\ n = 3$

↓ rendre .50

$z = 0.30,\ n = 3$

↓ laisser tomber les –.50

⋮

⋮

$z = 0.30,\ n = 2$

↓ rendre 0.20

$z = 0.10,\ n = 2$

↓ laisser tomber 0.20

$z = 0.10,\ n = 1$

↓ rendre 0.10

$z = 0,\ n = 1$ | STOP

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 montant 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 $n = 3$.

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

Olivier Lévêque

École polytechnique fédérale de Lausanne

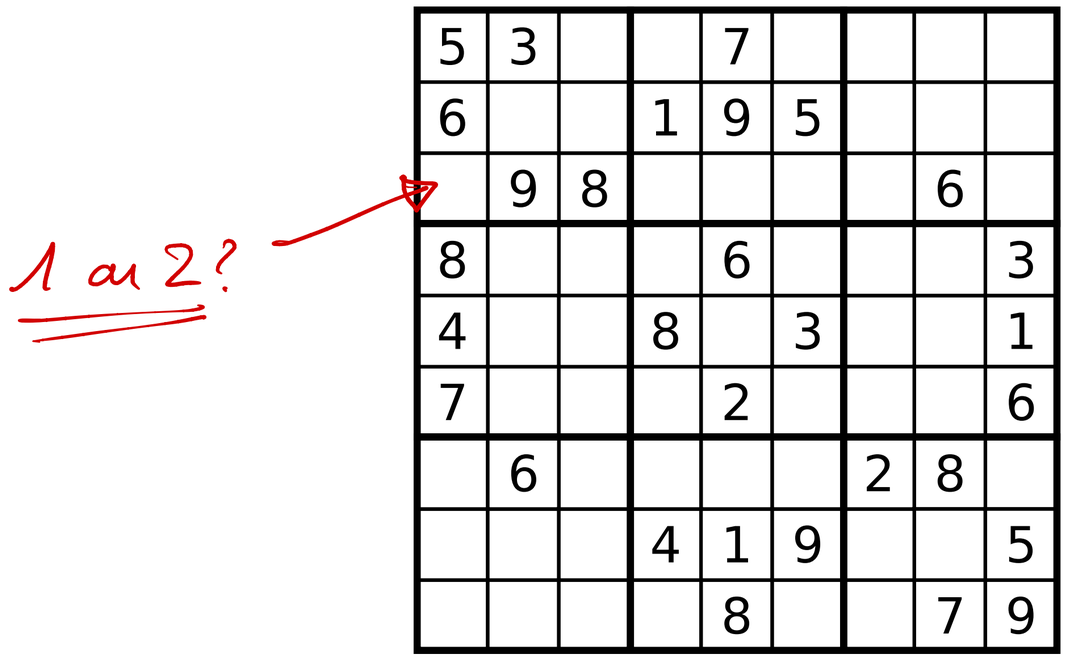
Programmation dynamique

- 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.

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.

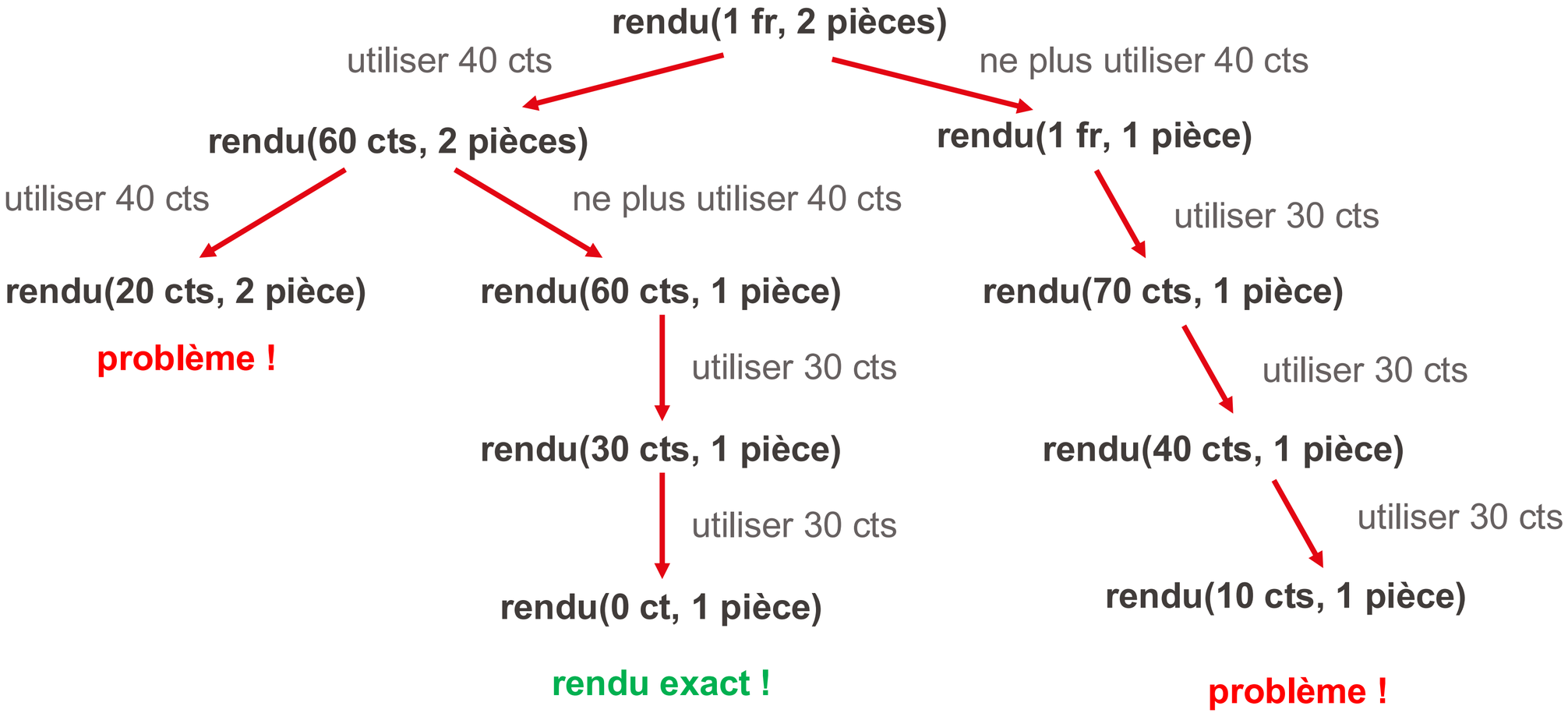
Un algorithme plus circonspect consiste à choisir, à chaque étape, entre **deux options** :

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 arbitrairement grand de 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**