Information, Calcul et Communication: Cours 4

Résumé de la dernière leçon: trois principes pour la récursivité:

- condition de terminaison
- appel à une instance plus simple du problème
- (éventuellement) recombinaison

Exemples: somme des n premiers nombres entiers,
recherche par dichotomie, rendu (glouton) de pièces de monnaie

Plan de cette leçon: récursivité (bis) – bienfaits et pièges

- tri par fusion
- "programmation" dynamique: rendu de pièces de monnaie (bis)

Tri d'une liste de nombres

Ce problème, omniprésent en informatique, a été résolu de mille façons!

- Dans la leçon 2, nous avons vu le tri par insertion, mais il existe aussi:
- le tri à bulles
- le tri cocktail
- le tri à peigne
- le tri pair-impair
- le tri par sélection
- le tri par tas
- le tri rapide
- le tri stupide (!)
- ...

↳ complexité temporelle $O(n^2)$

voir aussi:

"Algorithmes de tri – Wikipedia"

"AlgoRythmics – Youtube"

- 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 d'une liste - algorithme de 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 → condition de terminaison

$m \leftarrow \lfloor \frac{n}{2} \rfloor$

$L_1 =$ tri par fusion $(L(1:m), m)$

$L_2 =$ tri par fusion $(L(m+1:n), n-m)$

→ instances plus simples du problème

$L' =$ fusion (L_1, L_2) → recombinaison

Sortir : L'

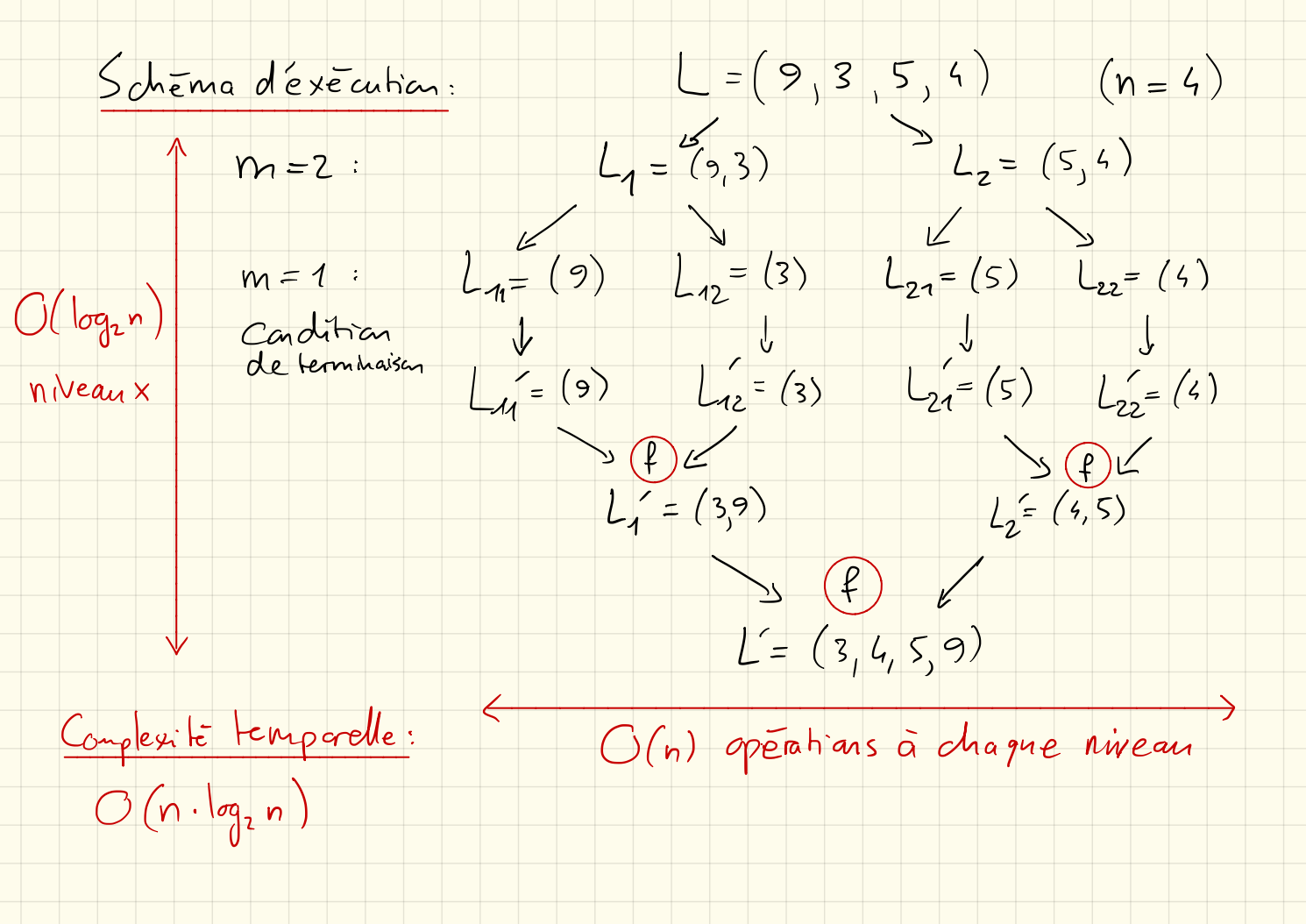


Schéma d'exécution :
L = (9, 3, 5, 4)
(n = 4)
m = 2 :
m = 1 :
Condition de terminaison
O(log₂ n)
niveaux
L₁ = (9, 3)
L₂ = (5, 4)
L₁₁ = (9)
L₁₂ = (3)
L₂₁ = (5)
L₂₂ = (4)
L₁₁' = (9)
L₁₂' = (3)
L₂₁' = (5)
L₂₂' = (4)
f
f
f
L₁' = (3, 9)
L₂' = (4, 5)
L' = (3, 4, 5, 9)
O(n) opérations à chaque niveau
Complexité temporelle :
O(n · log₂ n)

fusion ("fermeture éclair")

Complexité temporelle : $O(m_1 + m_2)$

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

Programmation dynamique

- Jusqu'à présent, nous n'avons vu que des algorithme récursifs où toutes les opérations effectuées sont nécessaires à la résolution du problème.
- Cependant, dans le cas du rendu de pièce 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

1 ou 2 ?

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 et explorer où cela mène, pour finalement conclure que seul un des deux chiffres mène à la solution.

Revenons au rendu (glouton) de pièces de monnaie:

Soit z = le montant à rendre

$P = (p_1, p_2, \dots, p_n)$ la liste des pièces disponibles pour le rendu de monnaie (avec $p_1 < p_2 < \dots < p_n$)

n = le nombre de pièces disponibles

rendu glouton (z, n)

- si $z = 0$, sortir : $()$ (liste 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) + **rendu glouton** $(z - p_n, n)$

Comme nous l'avais 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)

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.

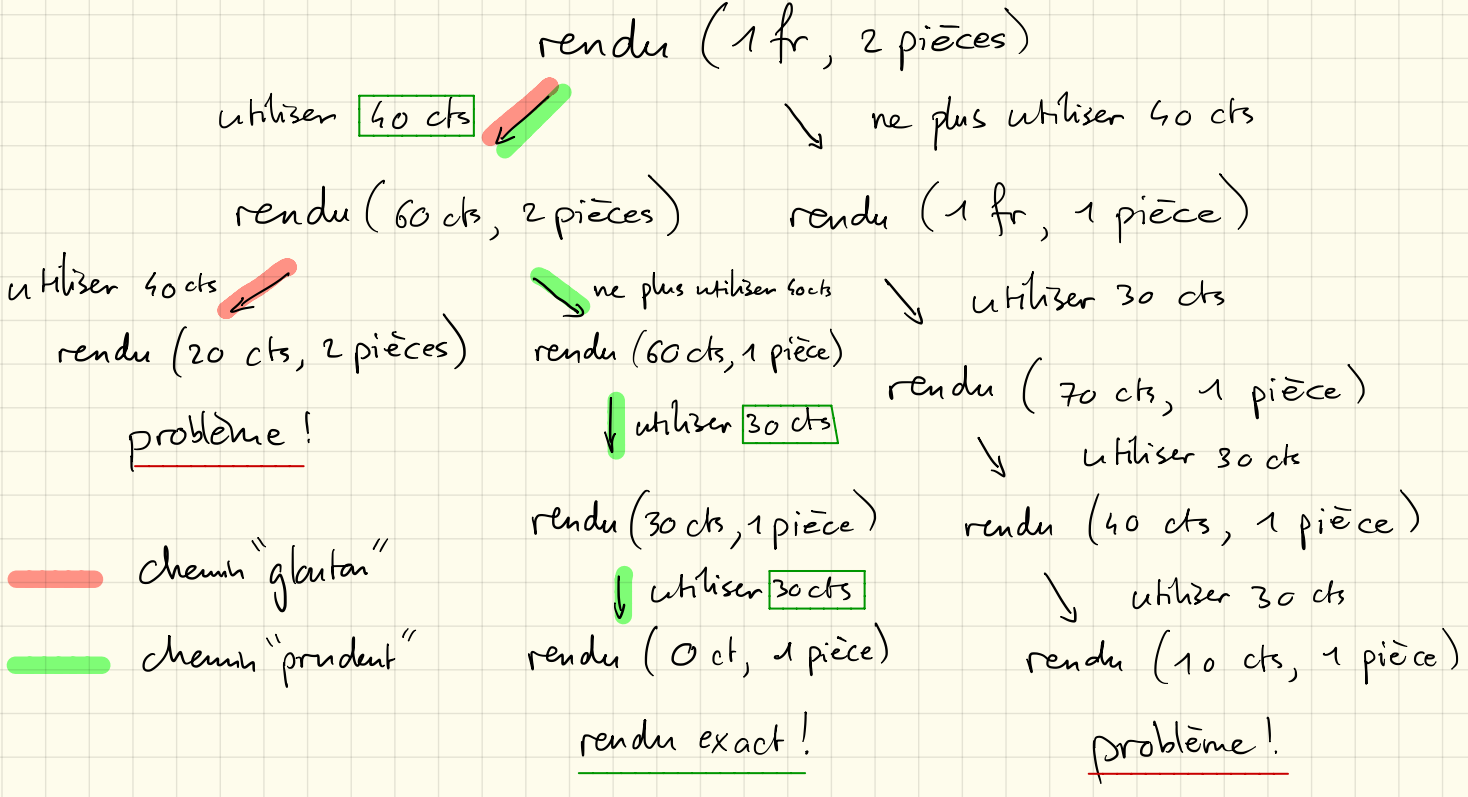
Un algorithme plus circonspect consiste à choisir, à chaque étape, entre deux options :

1) rendre la pièce avec la plus grande valeur disponible

ou 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 : $P = (30\text{ cts}, 40\text{ cts})$, $n = 2$, $Z = 1\text{ fr}$



Ecriture formelle de l'algorithme :

rendu dynamique(z, n)

liste L arbitrairement longue de nombres

- Si $z = 0$, sortir : $()$ (liste vide)
- Si $n = 0$ ou $z < p_1$, sortir : "~~rendu exact~~ impossible"
- Si $z < p_n$, sortir : rendu dynamique$(z, n-1)$
- $R_1 = (p_n)$ + rendu dynamique$(z - p_n, n)$
- R_2 = rendu dynamique $(z, n-1)$
- Si $|R_1| < |R_2|$, sortir : R_1
- Sinon, sortir : R_2

nouveauté

NB : $|R|$ = taille de R

Ceci garantit que la solution avec le moins de pièces est retenue.

Remarque :

Vu que l'algorithme explore tous les chemins possibles, il importe, à chaque fois que celui-ci aboutit à un cul de sac, de ne pas sortir "rendu exact impossible !". S'il se trouve qu'aucun chemin ne mène à la solution du problème (et que l'algorithme retourne donc la liste arbitrairement longue L), alors il faut afficher "rendu exact impossible !" (une fois sorti de l'algorithme).

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
  $\longrightarrow$ 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
  $\longrightarrow$ réduction du temps de calcul

[10]:
```python
# version "programmation dynamique" de l'algorithme de rendu de pièces de monnaie
def rendu_dynamique(z,n):
    if z==0:
        return []
    if n==0 or z<P[0]:
        return L
    if z<P[n-1]:
        return rendu_dynamique(z,n-1)
    R1=[P[n-1]]+rendu_dynamique(z-P[n-1],n)
    R2=rendu_dynamique(z,n-1)
    if len(R1)<=len(R2):
        if show:
            print("G",end='')
        return R1
    else:
        if show:
            print("D",end='')
        return R2
```

[11]:
```python
from time import *
show=0
L=[0]*1000
P=[2,5,10,20,50]
z=int(input("Montant à rendre: "))
t0=time()
R=rendu_dynamique(z,len(P))
t=time()-t0
if show:
    print("")
if sum(R)==z:
    print("Pièces rendues:",R)
else:
    print("Rendu exact impossible!")
print("Temps de calcul:",t,"secondes")
```

```
Montant à rendre: 141
Pièces rendues: [50, 50, 20, 10, 5, 2, 2, 2]
Temps de calcul: 0.049443960189819336 secondes
```

[10]:
```python
# version "programmation dynamique" de l'algorithme de rendu de pièces de monnaie
def rendu_dynamique(z,n):
    if z==0:
        return []
    if n==0 or z<P[0]:
        return L
    if z<P[n-1]:
        return rendu_dynamique(z,n-1)
    R1=[P[n-1]]+rendu_dynamique(z-P[n-1],n)
    R2=rendu_dynamique(z,n-1)
    if len(R1)<=len(R2):
        if show:
            print("G",end='')
        return R1
    else:
        if show:
            print("D",end='')
        return R2
```

[17]:
```python
from time import *
show=0
L=[0]*1000
P=[2,5,10,20,50,100,200]
z=int(input("Montant à rendre: "))
t0=time()
R=rendu_dynamique(z,len(P))
t=time()-t0
if show:
    print("")
if sum(R)==z:
    print("Pièces rendues:",R)
else:
    print("Rendu exact impossible!")
print("Temps de calcul:",t,"secondes")
```

```
Montant à rendre: 458
Pièces rendues: [200, 200, 50, 2, 2, 2, 2]
Temps de calcul: 11.631322622299194 secondes
```

[2]:
```
# version "programmation dynamique" de l'algorithme de rendu de pièces de monnaie, avec memoisation
def rendu_dynamique(z,n):
    if z==0:
        return []
    if n==0 or z<P[0]:
        return L
    if z<P[n-1]:
        return rendu_dynamique(z,n-1)
    R1=[P[n-1]]+rendu_dynamique(z-P[n-1],n)
    R2=rendu_dynamique(z,n-1)
    if len(R1)<=len(R2):
        if show:
            print("G",end='')
        return R1
    else:
        if show:
            print("D",end='')
        return R2
```

[3]:
```
def memoize(f):
    memo={}
    def helper(z,n):
        if (z,n) not in memo:
            memo[z,n]=f(z,n)
        return memo[z,n]
    return helper
```

[4]:
```
from time import *
show=0
L=[0]*1000
P=[2,5,10,20,50,100,200]
z=int(input("Montant à rendre: "))
rendu_dynamique=memoize(rendu_dynamique)
t0=time()
R=rendu_dynamique(z,len(P))
t=time()-t0
if show:
    print("")
if sum(R)==z:
    print("Pièces rendues:",R)
else:
    print("Rendu exact impossible!")
print("Temps de calcul:",t,"secondes")
```

```
Montant à rendre: 458
Pièces rendues: [200, 200, 50, 2, 2, 2, 2]
Temps de calcul: 0.003373384475708008 secondes
```