

EPFL Rappel : ingrédients de base des algorithmes

Données

- Entrées
- Sorties
- Variables internes

Instructions

- Affectations
- Structures de contrôle
 - Branchements conditionnels (tests)
 - Itérations (boucles)
 - Boucles conditionnelles

Sous-algorithmes

Rappel: complexité temporelle et notation $\Theta(\cdot)$

Définition 1

La complexité temporelle d'un algorithme est le nombre **d'opérations élémentaires** effectuées au cours de son exécution, dans le **pire des cas**.

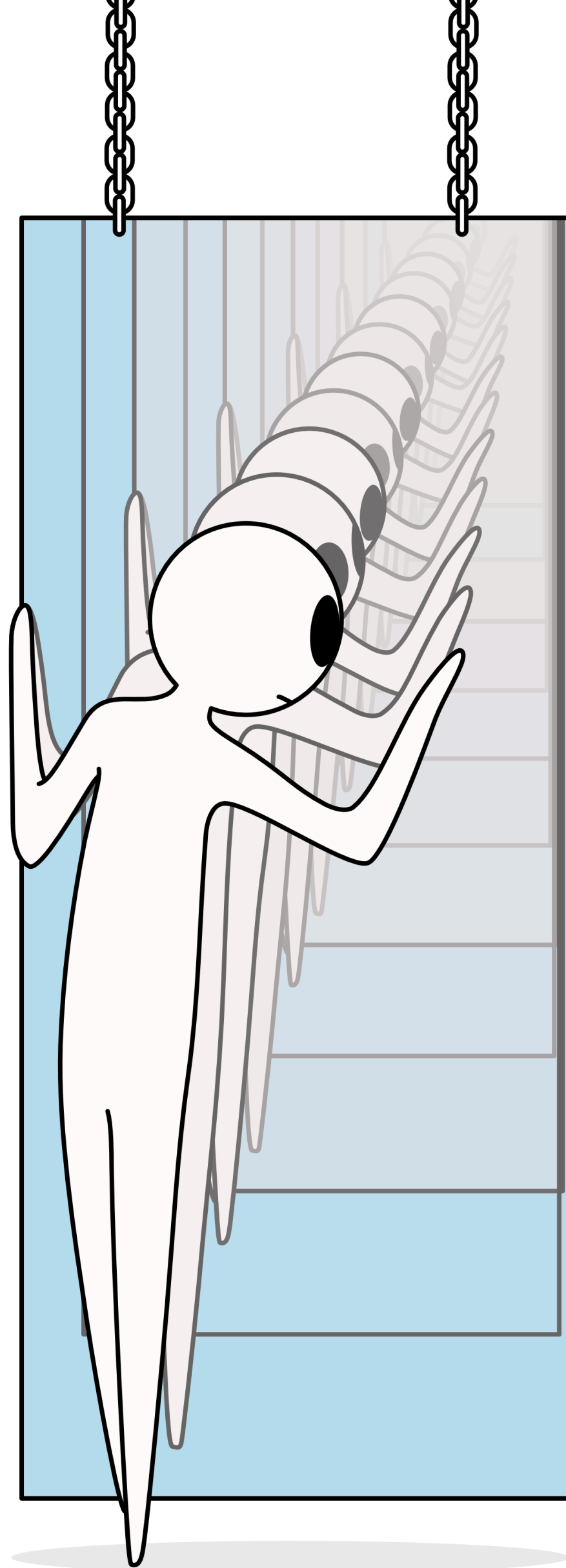
Définition 2

Soient $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$ deux fonctions non-négatives
On dit que " $f(n)$ est un **grand theta** de $g(n)$ " et on écrit " $f(n) = \Theta(g(n))$ "
s'il existe $0 < C_1 < C_2 < \infty$ et $N \geq 1$ tels que

$$C_1 g(n) \leq f(n) \leq C_2 g(n) \quad \text{pour tout } n \geq N$$

Deux exemples :

- Les fonctions $f(n) = n + 2$ et $f(n) = 3n + 3$ sont toutes deux des $\Theta(n)$
- La fonction $f(n) = \frac{n(n-1)}{2} + 1$ est un $\Theta(n^2)$



Information, Calcul et Communication

Récurtivité : introduction

Olivier Lévêque

Un algorithme récursif est un algorithme qui résout un problème en calculant des solutions d'instances plus petites du même problème (l'algorithme s'invoquant lui-même de façon répétitive).

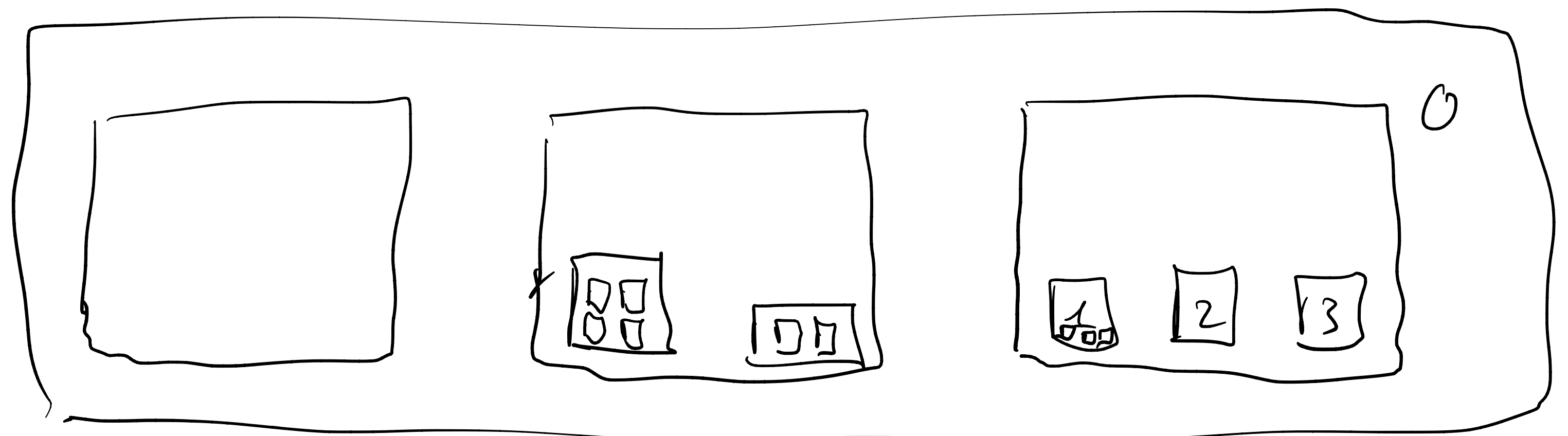
Exemple: recherche de clé



recherche de clé (dans un carton donné)

est-ce que je vois la clé quelque part dans le carton?

- si oui, c'est bon: sortir de l'algorithme
- si non, est-ce que le carton contient au moins un autre carton ?
 - si oui, parcourir la liste des autres cartons et effectuer une **recherche de clé** dans chacun d'entre eux
 - si non, sortir de l'algorithme



recherche de clé (dans un carton donné)

est-ce que je vois la clé quelque part dans le carton?

- si oui, c'est bon: sortir de l'algorithme
- si non, est-ce que le carton contient au moins un autre carton ?
 - si oui, parcourir la liste des autres cartons et effectuer une **recherche de clé** dans chacun d'entre eux
 - si non, sortir de l'algorithme

Deux remarques

- cet algorithme se termine toujours (mais pas forcément avec succès, si la clé n'est pas dans le carton donné)
- remplacer "carton" par "camionnette" pour lancer l'algorithme au départ

La récursivité: trois principes

1. Un algorithme récursif a toujours une **condition de terminaison** (correspondant à l'instance la plus simple du problème à résoudre).
2. Pour résoudre un problème donné, un algorithme récursif fait d'abord appel à lui-même avec des données de taille plus petite en entrée (résolvant ainsi une **instance plus simple** du problème).
3. Un **processus de reconstruction** est généralement nécessaire ensuite pour obtenir la solution du problème d'origine.

Illustration

Calcul de la somme des n premiers nombres entiers

Exemple: lorsque $n = 3$, $s(3) = 1 + 2 + 3 = 6$

somme (version itérative)

entrée : nombre entier positif n

sortie : $s(n)$ = somme des n premiers nombres entiers

$s \leftarrow 0$

Pour i allant de 1 à n :

$s \leftarrow s + i$

Sortir : s

Complexité temporelle: $\Theta(n)$ (une boucle)

$$\begin{array}{c} S(n-1) \\ \hline = 1+2+3+\dots+(n-1)+n \\ \quad \quad \quad \downarrow \\ \quad \quad \quad \underbrace{\hspace{1cm}} \end{array}$$

- Résolution incrémentale : $s(n) = n + s(n - 1)$

instance plus simple du problème



- Résolution incrémentale : $s(n) = n + s(n - 1)$



recombinaison

- Condition de terminaison : $n = 1$ (sortie correspondante : $s(1) = 1$)

somme récursive

entrée : nombre entier positif n

sortie : $s(n)$ = somme des n premiers nombres entiers

Si $n = 1$:

Sortir : 1

Sortir : $n + \text{somme récursive}(n - 1)$

recursion

Instance plus simple

cond. de terminaison

somme réursive

entrée : nombre entier positif n

sortie : $s(n) =$ somme des n premiers nombres entiers

Si $n = 1$:

Sortir : 1

Sortir : $n + \text{somme réursive}(n - 1)$

Complexité temporelle: également $\Theta(n)$ (comme nous allons le voir)

Schéma d'exécution pour $n = 3$

Somme rec (3)

$n=1?$ non

Sortir $3 +$ Somme rec (2)

$n=1?$ non

Sortir $2 +$ Somme rec (1)

$n=1?$ oui $\rightarrow$ sortir (1)

(3)

(6)

somme récursive

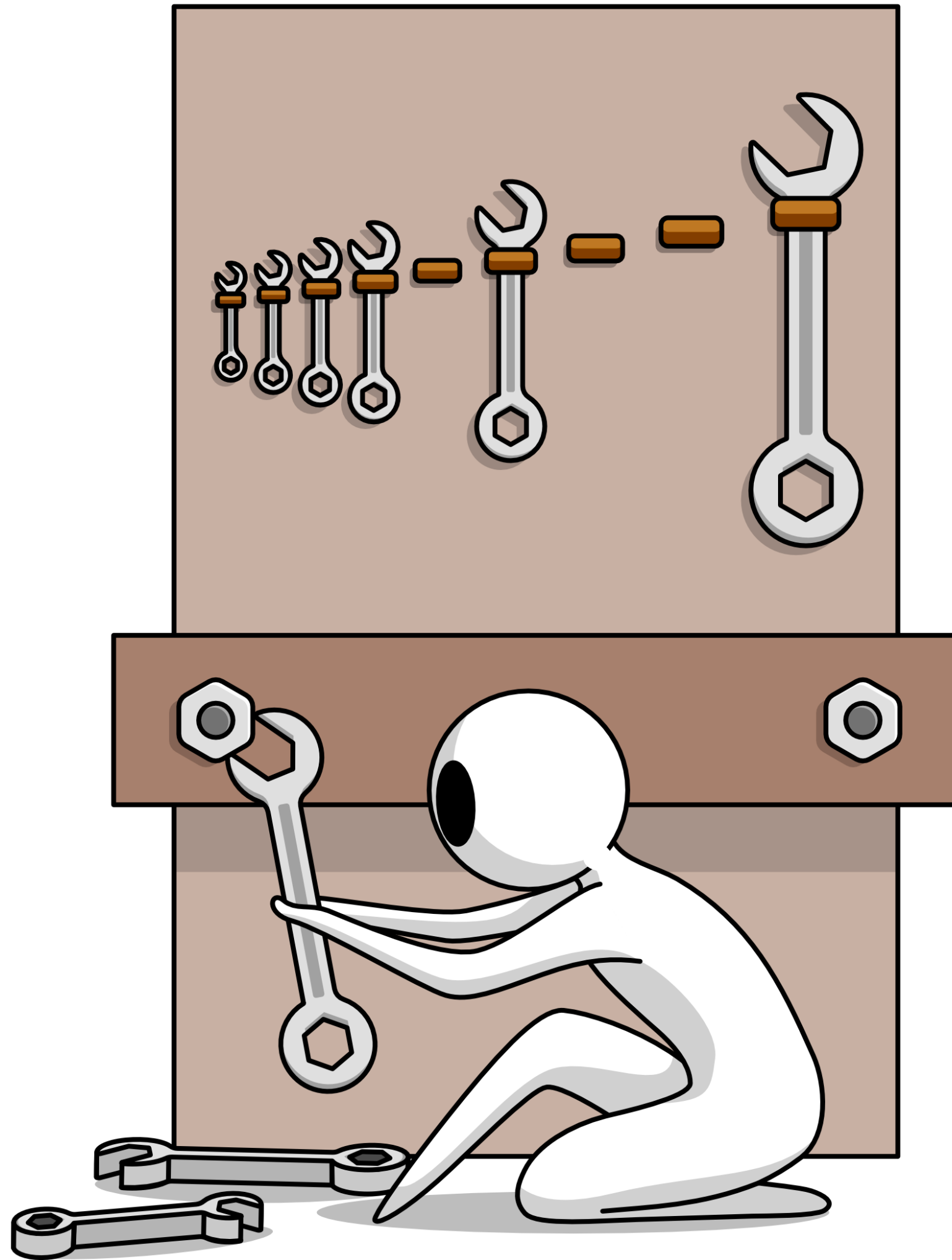
entrée : nombre entier positif n

sortie : $s(n)$ = somme des n premiers nombres entiers

Si $n = 1$:

Sortir : 1

Sortir : $n +$ **somme récursive**($n - 1$)



Information, Calcul et Communication

Recherche par dichotomie

Olivier Lévêque

Problème

Identifier si un élément fait partie d'une liste donnée est une composante essentielle de nombreux algorithmes. On distingue deux cas principaux :

1. Recherche dans une liste non-ordonnée

- Est-ce qu'un ingrédient donné (ex: gluten) fait partie ou non de la liste des ingrédients d'un produit ?
- Est-ce que la feuille de papier sur laquelle j'avais écrit ce numéro de téléphone se trouve dans cette pile de feuilles que j'ai devant moi ?

Dans ce cas, pour identifier si l'élément qu'on recherche fait partie de la liste ou non, on n'a pas d'autre choix que de parcourir toute la liste.

recherche linéaire

entrée : liste L d'objets, de taille n , objet x

sortie : est-ce que $x \in L$? (*oui/non*)

Pour i allant de 1 à n :

 Si $L(i) = x$, alors : Sortir : *oui*

Sortir : *non*

La complexité temporelle de l'algorithme ci-dessus est $\Theta(n)$.

(Dans le pire des cas, i.e., lorsque $x \notin L$, il faut parcourir toute la liste.)

EPFL Recherche d'un élément dans une liste (bis)

2. Recherche dans une liste ordonnée

- Identification d'un numéro de carte de crédit
- Recherche d'un nom dans un annuaire (ordre lexicographique)

Dans ce cas, on peut bien sûr effectuer la même recherche linéaire que précédemment, mais on peut aussi faire beaucoup mieux grâce à la récursivité. C'est l'algorithme de **recherche par dichotomie**.

$$L = (\cancel{1}, \cancel{3}, \cancel{7}, \cancel{8}, \cancel{32}, \cancel{45}, \cancel{46}, 47, 99) \quad n=9$$

$x = 52 \in L?$

$\uparrow$
 $52 > 32?$

$\uparrow$
 $52 > 46?$

dichotomie

entrée : liste ordonnée L de nombres entiers, de taille n , objet x
 sortie : est-ce que $x \in L$? (oui/non)

cas de
termin :

si $n=1$, alors : si $x=L(1)$, sortir oui
 sinon, sortir non

$m \leftarrow \lfloor \frac{n}{2} \rfloor$ = partie entière inférieure de $\frac{n}{2}$

si $x \leq L(m)$, sortir dichotomie ($L(1:m)$, m , x)

sinon, sortir dichotomie ($L(m+1:n)$, $n-m$, x)

$$\frac{n}{2} = 10.5 \\ \Rightarrow \lfloor \frac{n}{2} \rfloor = 10$$

dichotomie

entrée : liste ordonnée L de nombres entiers, de taille n , objet x

sortie : est-ce que $x \in L$? (*oui/non*)

Si $n = 1$, alors : si $x = L(1)$, alors : Sortir *oui*
sinon : Sortir *non*

$m \leftarrow \lfloor n/2 \rfloor$

Si $x \leq L(m)$, alors : Sortir **dichotomie**($L(1:m), m, x$)

Sinon : Sortir **dichotomie**($L(m+1:n), n-m, x$)

Schéma d'exécution de l'algorithme

Avec $L = (1, 7, 9, 14)$, $n = 4$ et $x = 10$:

dicho ($L, 4, 10$)

$n = 1$? non

$m \leftarrow 2$

$10 \leq 7$? non $\rightarrow$

Sortir dicho ($(9, 14), 2, 10$)

$n = 1$? non

$m \leftarrow 1$

$10 \leq 9$? non

$\rightarrow$ Sortir dicho ($(14), 1, 10$)

$n = 1$? oui $\rightarrow 10 = 14$? non

$\rightarrow$ Sortir non

dichotomie

entrée : liste ordonnée L de nombres entiers,
de taille n , objet x

sortie : est-ce que $x \in L$? (oui/non)

Si $n = 1$, alors : si $x = L(1)$, alors : Sortir *oui*
sinon : Sortir *non*

$m \leftarrow \lfloor n/2 \rfloor$

Si $x \leq L(m)$, alors : Sortir **dichotomie**($L(1:m), m, x$)

Sinon : Sortir **dichotomie**($L(m+1:n), n-m, x$)

Complexité temporelle

$\mathcal{O}(\log_2(n))$ [$n = 10^3 : \sim 10$, $n = 10^6 : \sim 20$, $n = 10^9 : \sim 30$]

Complexité temporelle de l'algorithme

- A chaque étape, la taille de la liste est divisée (approx.) par 2.
- Partant d'une liste de taille n , le nombre d'étapes nécessaires pour arriver à une liste de taille 1 est donc (approx.) $\log_2(n)$.
- → complexité temporelle $\Theta(\log_2(n))$:
bien plus efficace que l'algorithme de recherche linéaire vu avant !

NB: $\log_3(n) = \frac{\log_2(n)}{\log_2(3)} = \frac{1}{\log_2(3)} \cdot \log_2(n)$

$$\Theta(\log_3(n)) = \Theta(\log_2(n))$$



Information, Calcul et Communication

Tri par fusion

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
 $m \leftarrow \lfloor \frac{n}{2} \rfloor$
 $L_1 \leftarrow \text{tri par fusion}(L(1:m), m)$
 $L_2 \leftarrow \text{tri par fusion}(L(m+1:n), n-m)$
 $L' \leftarrow \text{fusion}(L_1, L_2)$
 sortir L'

$= (L(1), L(2), \dots, L(m))$

EPFL 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
 $m \leftarrow \lfloor n/2 \rfloor$
 $L_1 \leftarrow \text{tri par fusion}(L(1:m), m)$
 $L_2 \leftarrow \text{tri par fusion}(L(m+1:n), n-m)$
 $L' \leftarrow \text{fusion}(L_1, L_2)$
Sortir : L'

Schéma d'exécution de l'algorithme

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

tri par fusion

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

sortie : Liste L' triée

Si $n = 1$, sortir : L

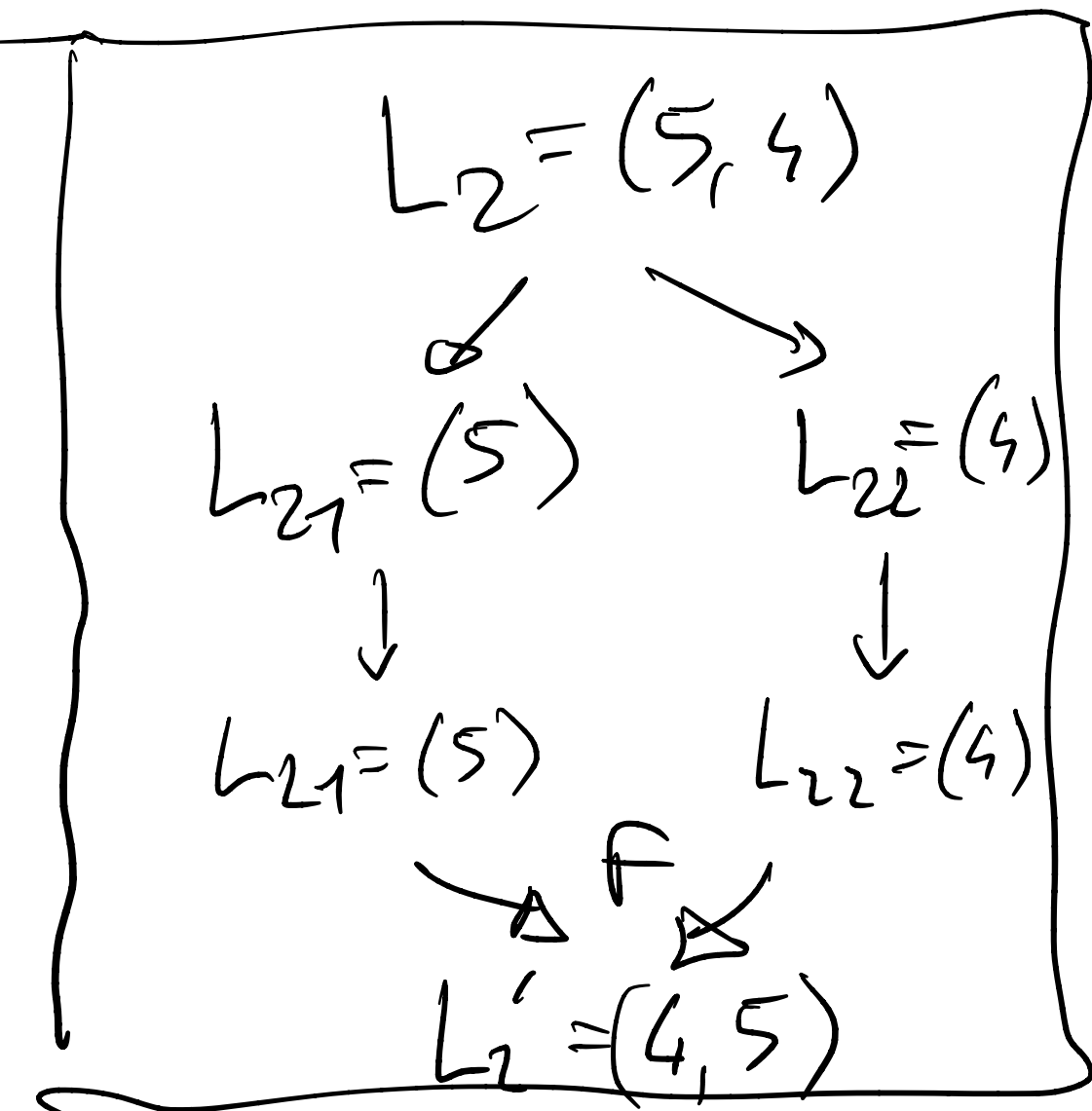
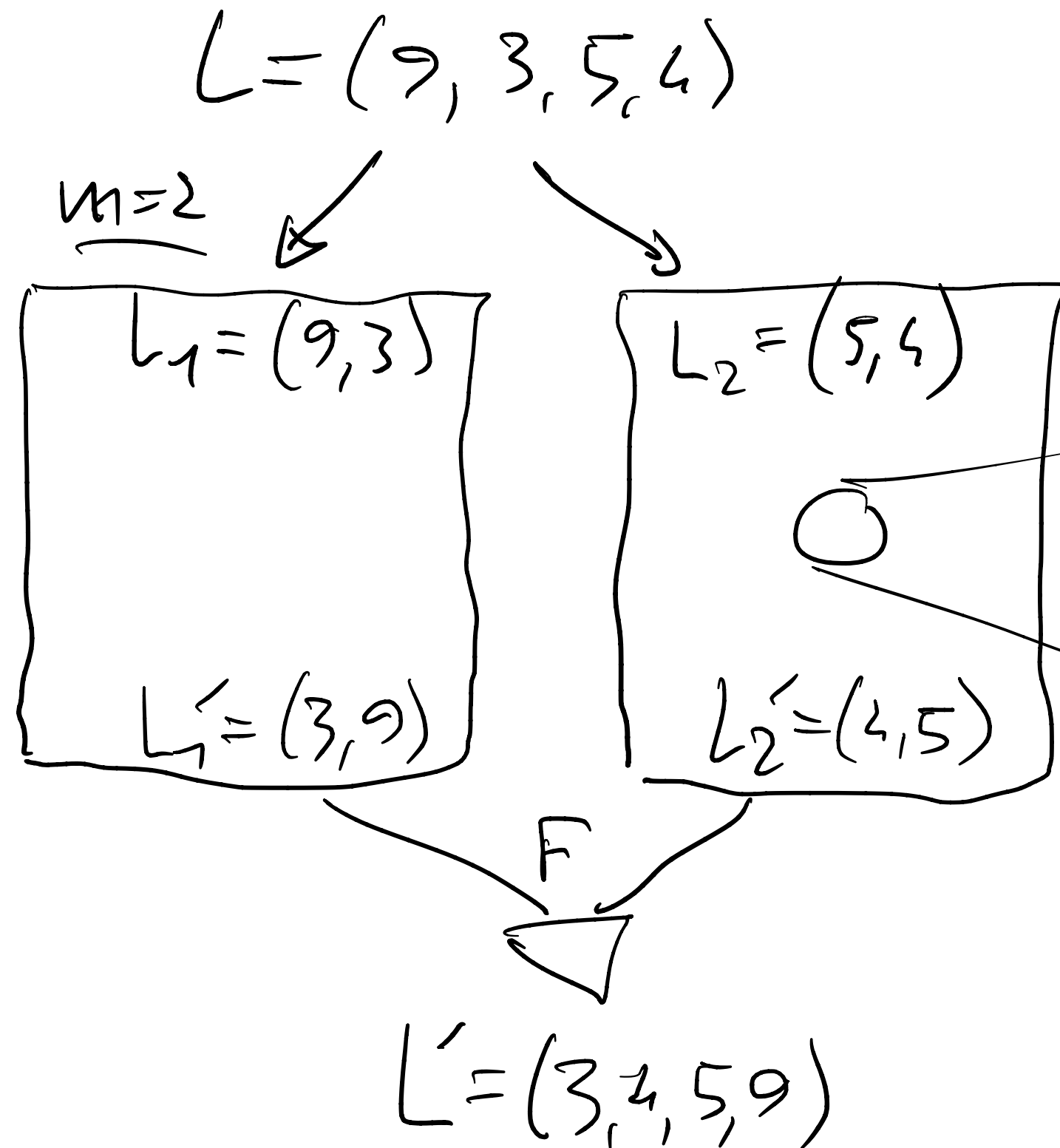
$m \leftarrow \lfloor n/2 \rfloor$

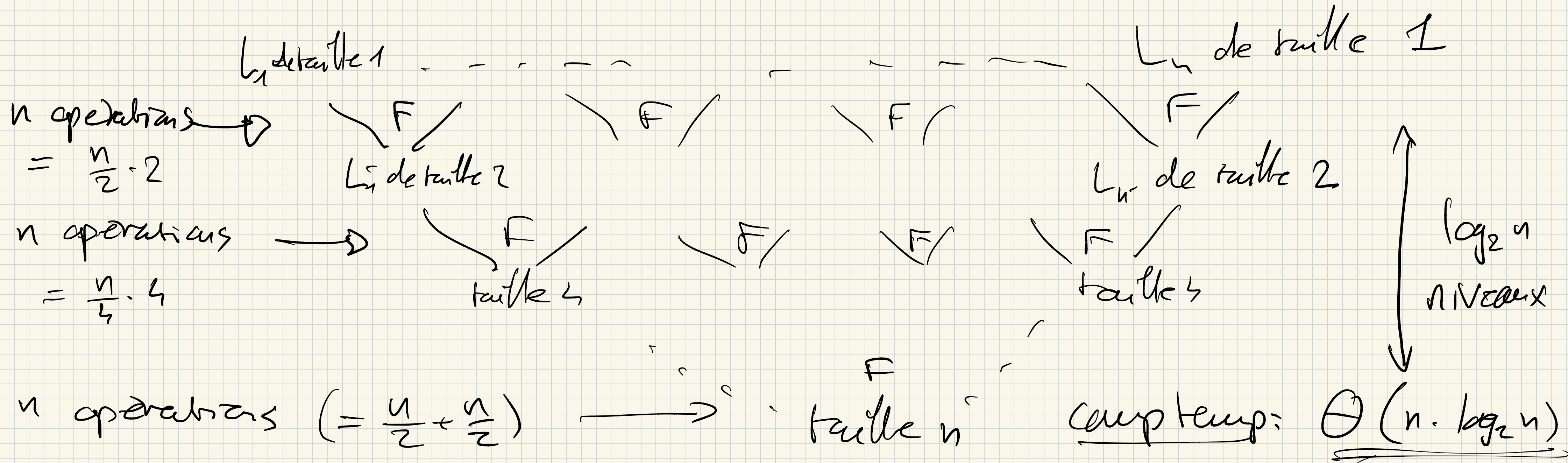
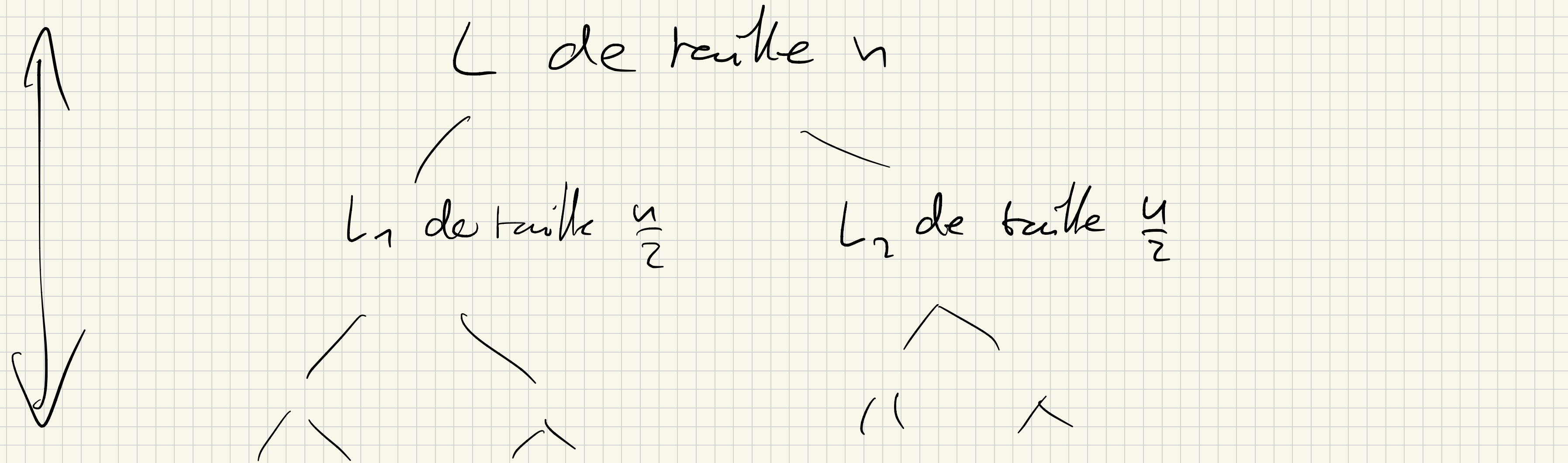
$L_1 \leftarrow \text{tri par fusion}(L(1:m), m)$

$L_2 \leftarrow \text{tri par fusion}(L(m+1:n), n-m)$

$L' \leftarrow \text{fusion}(L_1, L_2)$

Sortir : L'





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 $\Theta(n \cdot \log_2(n))$:
plus efficace que le tri par insertion (dans le pire des cas)

$$\log_2 n := k \quad \text{tg} \quad 2^k = n$$

Algorithme fusion ("fermeture éclair")

Fusion

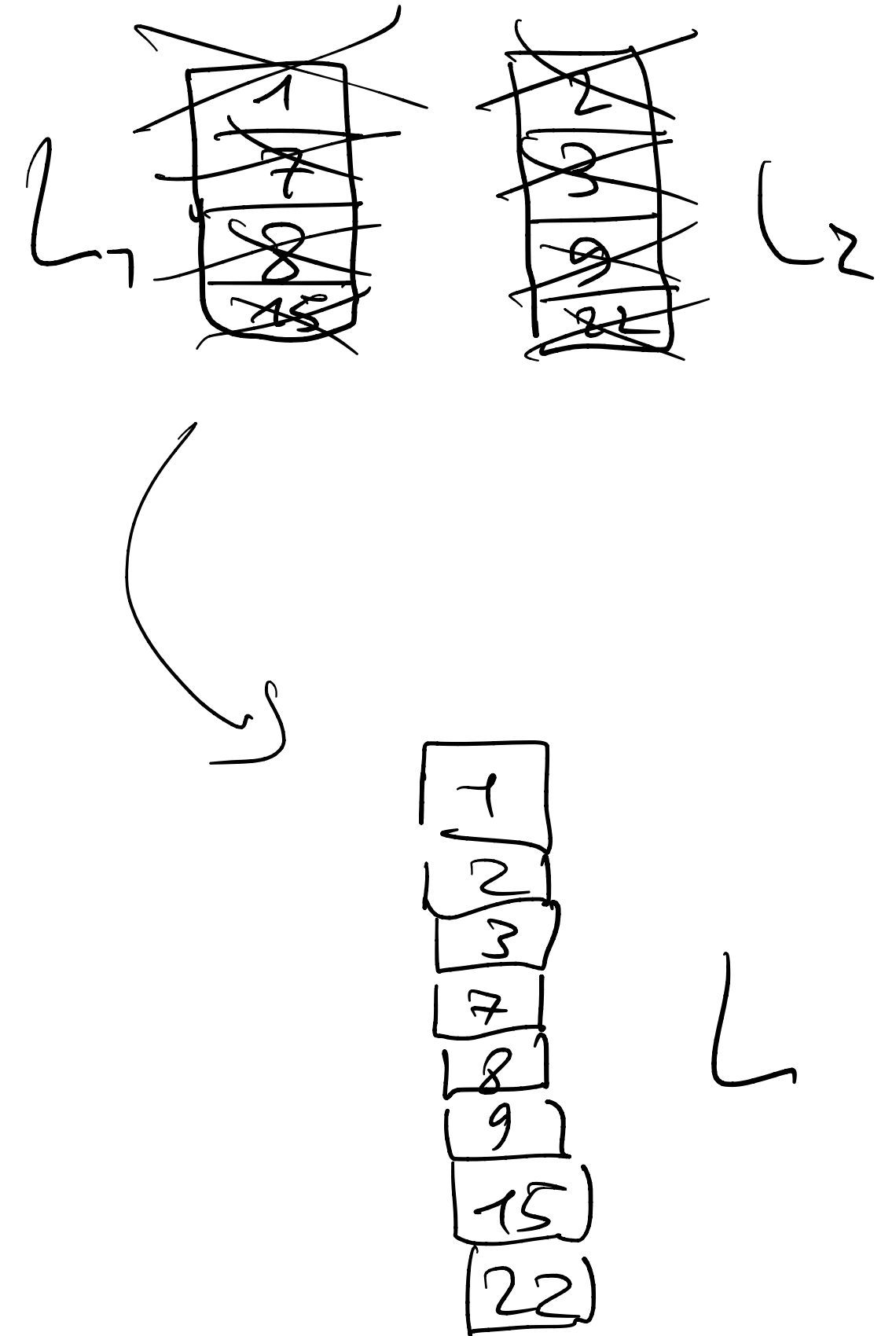
entrée : Listes ordonnées L_1, L_2 (de tailles m_1 et m_2 resp.)
 sortie : Liste L (de taille $m_1 + m_2$, également ordonnée)

```

 $j_1 \leftarrow 1$ 
 $j_2 \leftarrow 1$ 
 $j \leftarrow 1$ 
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  $j_2 \leq m_2$  :
     $L(j) \leftarrow L_2(j_2)$ 
     $j_2 \leftarrow j_2 + 1$ 
     $j \leftarrow j + 1$ 
Sortir :  $L$ 
  
```



EPFL 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
 $\Theta(m_1 + m_2)$