

Rappel : ingrédients de base des algorithmes

Données

- Entrées
- Sorties
- Variables internes

Instructions

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

→ Pour i allant de 1 à n

→ Tant que ...

Pour i allant de 1 à n
(répéter ...)

≡

$i \leftarrow 1$

Tant que $i \leq n$
(répéter ...)

$i \leftarrow i + 1$



Information, Calcul et Communication

Sous-algorithmes

Olivier Lévêque



Kathryn Greenhill [CC BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>)]

Un problème récurrent :
comment préparer ses
valises en famille ?

Solution 1

Algorithme centralisé : une personne se charge de tout: pas idéal...

Solution 2

Chacun prépare sa propre valise: beaucoup mieux !

Et encore mieux: chaque personne prépare séparément :

- ses habits
- sa trousse de toilette
- ses livres
- sa tenue de plongée ...

Un exemple en pseudocode

algorithme principal

entrée : ...
sortie : ...

...
 $x \leftarrow \text{algo1}(5)$
 $k \leftarrow 10$
 $y \leftarrow \text{algo2}(k)$
 $z \leftarrow x + \text{algo1}(y)$
...

algo1

entrée : nombre entier n
sortie : nombre entier $s1$

(instructions)

algo2

entrée : nombre entier n
sortie : nombre entier $s2$

(instructions)

Illustration du principe avec le tri d'une liste

Tri d'une liste de nombres

Comment trier une liste de nombres ? (ou un jeu de cartes, ou encore une liste de noms, ou ...)

Il existe de nombreuses façons de faire, plus ou moins efficaces. Nous allons en voir une: le **tri par insertion**, qui permet de bien illustrer le principe de l'utilisation de sous-algorithmes.

Tri d'une liste ?

$$\underline{\text{Ex}}: L = (10, 3, 14, 5)$$

$$\rightarrow L' = (3, 5, 10, 14)?$$

entrée : liste L de n nombres entiers

sortie : liste L triée dans l'ordre croissant ?

Pour i allant de 2 à n :

si $L(i) < L(i-1)$, alors permuter $L(i)$ et $L(i-1)$

Sortir L

$\underline{\text{Ex}}: L = (10, 3, 14, 5) \xrightarrow{i=2} L = (3, 10, 14, 5)$
$\xrightarrow{i=3} L = (3, 10, 14, 5) \xrightarrow{i=4} L = (3, 10, 5, 14)$

Tri par insertion: algorithme principal

tri par insertion

entrée : liste de nombres L , taille de la liste n
 sortie : liste L triée dans l'ordre croissant

Pour i allant de 2 à n :

Si $L(i) < L(i - 1)$, alors :

$L \leftarrow \text{insérer}(L, i)$

Sortir : L

⚠ de l'ordre de n^2 opérations
à effectuer ici !

$i=4$
 \downarrow
 $L = (3, 10, 14, 5)$
 insertion à
 la bonne place

$L = (10, 12, 3, 7)$

$i=2$: $L = (10, \underline{12}, 3, 7)$

$i=3$: $L = (\underline{3}, 10, 12, 7)$

$i=4$: $L = (3, \underline{7}, 10, 12)$

Tri par insertion: sous-algorithme 1

insérer

entrée : liste de nombres L , nombre entier positif i
 sortie : liste L avec l'élément $L(i)$ bien placé

$j \leftarrow i$

Tant que $j > 1$ & $L(j) < L(j - 1)$:

$L \leftarrow \text{permuter}(L, j, j - 1)$

$j \leftarrow j - 1$

Sortir : L

$$L = (3, 7, 10, 12)$$

$j=2$
↓

$$L(2) < L(1)? \text{ non}$$

$$L = (3, 10, 12, 7)$$

$i=4$
↓
insertion

$$=7 \quad =12 \quad j=4$$

$L(4) < L(3)? \text{ oui}$

permuter 7 & 12
 $j \leftarrow 3$

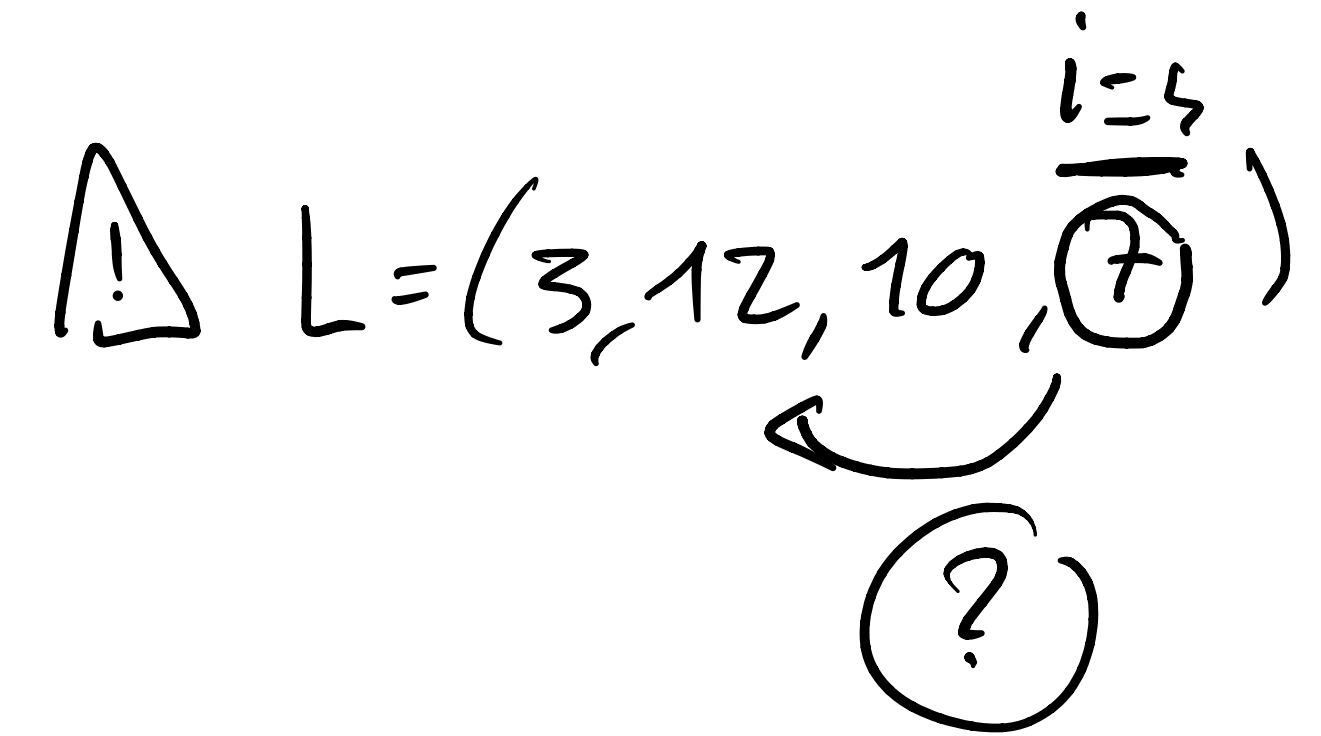
$$L = (3, 10, 7, 12)$$

$$7 \quad 10 \quad j=3$$

$L(3) < L(2)? \text{ oui}$

permuter 10 & 7
 $j \leftarrow 2$

Tri par insertion: sous-algorithme 1



insérer

entrée : liste de nombres L , nombre entier positif i
 sortie : liste L avec l'élément $L(i)$ bien placé

$j \leftarrow i$

Tant que $j > 1$ & $L(j) < L(j - 1)$:

$L \leftarrow \text{permuter}(L, j, j - 1) \equiv \text{permuter } L(j) \text{ \& } L(j-1)$

$j \leftarrow j - 1$

Sortir : L

Remarque importante :

Les éléments $L(1) \dots L(i - 1)$ doivent être déjà triés pour que ce sous-algorithme fonctionne correctement. Heureusement, c'est le cas ici !

Tri par insertion: sous-algorithme 2

entrée = $(L(j) = 2, L(k) = 3)$

permuter

entrée : liste de nombres entiers L , nombres entiers positifs j, k
 sortie : liste L avec les éléments $L(j)$ et $L(k)$ permutés

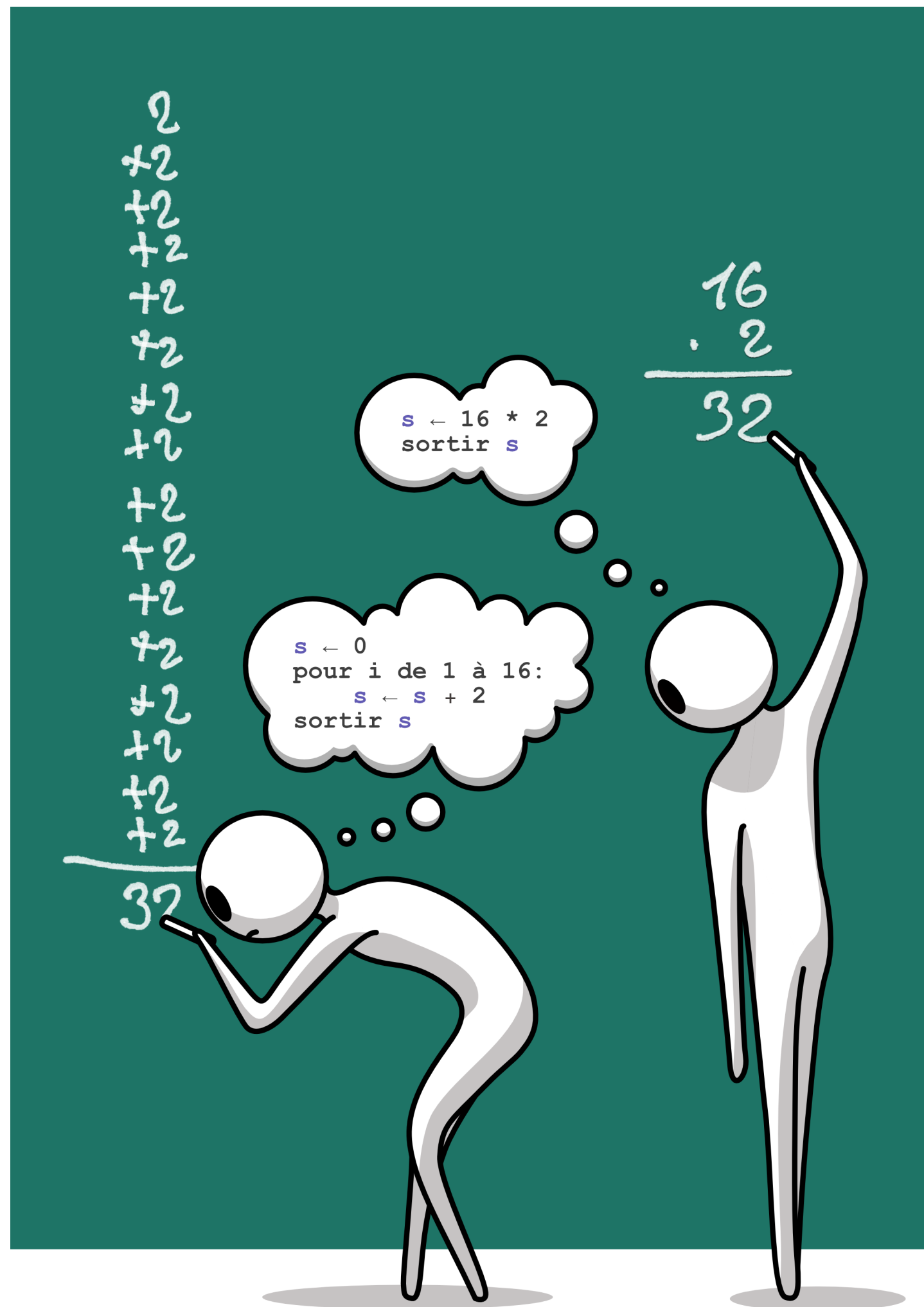
$temp \leftarrow L(j)$
 $L(j) \leftarrow L(k)$
 $L(k) \leftarrow temp$
 Sortir : L

$temp \leftarrow 2$
 $L(j) \leftarrow 3$
 $L(k) \leftarrow 2$

~~$L(j) \leftarrow L(k)$
 $L(k) \leftarrow L(j)$~~

sortie = $(3, 2)$

sortie = $(3, 3)$



Information, Calcul et Communication

Algorithmes :
complexité temporelle

Complexité temporelle d'un algorithme

La complexité temporelle d'un algorithme est son temps d'exécution.

Définition plus précise :

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

- **opération élémentaire** = addition, soustraction, multiplication ou comparaison de deux bits
- **pire des cas**: le temps d'exécution peut en effet dépendre des données d'entrée.

Approximation pour ce cours :

complexité temporelle = nombre d'**instructions** lues par l'algorithme au cours de son exécution (dans le pire des cas)

Algorithme 1

Tant que $1 > 0$:
 Afficher "bonjour"

Sortir 0

Complexité temporelle infinie !

EPFL Exemples

Algorithme 1

Tant que $1 > 0$:
Afficher "bonjour"

Algorithme 2

entrée : L liste de nombres, n taille de la liste
sortie : m moyenne des n nombres de la liste

1 $m \leftarrow 0$
 n Pour i allant de 1 à n :
 $m \leftarrow m + L(i)$
1 Sortir : m/n

$n+2$ opérations

$m \leftarrow 0$
 $i \leftarrow 1$
Tant que $i \leq n$
 $m \leftarrow m + L(i)$
 $i \leftarrow i+1$

Sortir m/n

} 2

} $3n$

} 1

$3n+3$ opérations

Algorithme 3 (version légèrement modifiée de l'algorithme «Tous différents ?»)

entrée : L liste de nombres, n taille de la liste
sortie : *oui* ou *non*

Pour i allant de 1 à $n - 1$:
 Pour j allant de $i + 1$ à n :
 Si $L(i) = L(j)$, alors :
 Sortir : *non*
Sortir : *oui*

Comp. temporelle \sim nombre de paires d'éléments
dans $\{1..n\} = \frac{n(n-1)}{2}$
ordre n^2

Notation $\Theta(\cdot)$: introduction

- En général, on évalue la complexité temporelle d'un algorithme en fonction d'un paramètre lié à la **taille des données d'entrée** (le paramètre n dans les deux exemples précédents).
- Pourquoi tant s'intéresser à cette complexité temporelle ? Voici un exemple concret:

Supposons qu'un algorithme prenne une minute pour s'exécuter avec des données d'entrée de taille $n = 1'000$. On aimerait savoir en combien de temps (au pire) s'exécutera ce même algorithme avec des données d'entrée de taille $n = 10'000$.

- Si on peut caractériser le nombre d'opérations effectuées par l'algorithme en fonction de n (comme par exemple pour l'algorithme 3 qui effectue $\frac{n(n-1)}{2} + 1$ opérations lors de son exécution, dans le pire des cas), alors on peut répondre à la question ci-dessus.

Notation $\Theta(\cdot)$: définition

- Dans de nombreuses applications, on a affaire à des données d'entrée de **grande** taille.
- Dans ce cas, on aimerait obtenir des **ordres de grandeur** plutôt que de devoir faire des calculs détaillés.

Définition

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 " $f(n) = \Theta(g(n))$ "

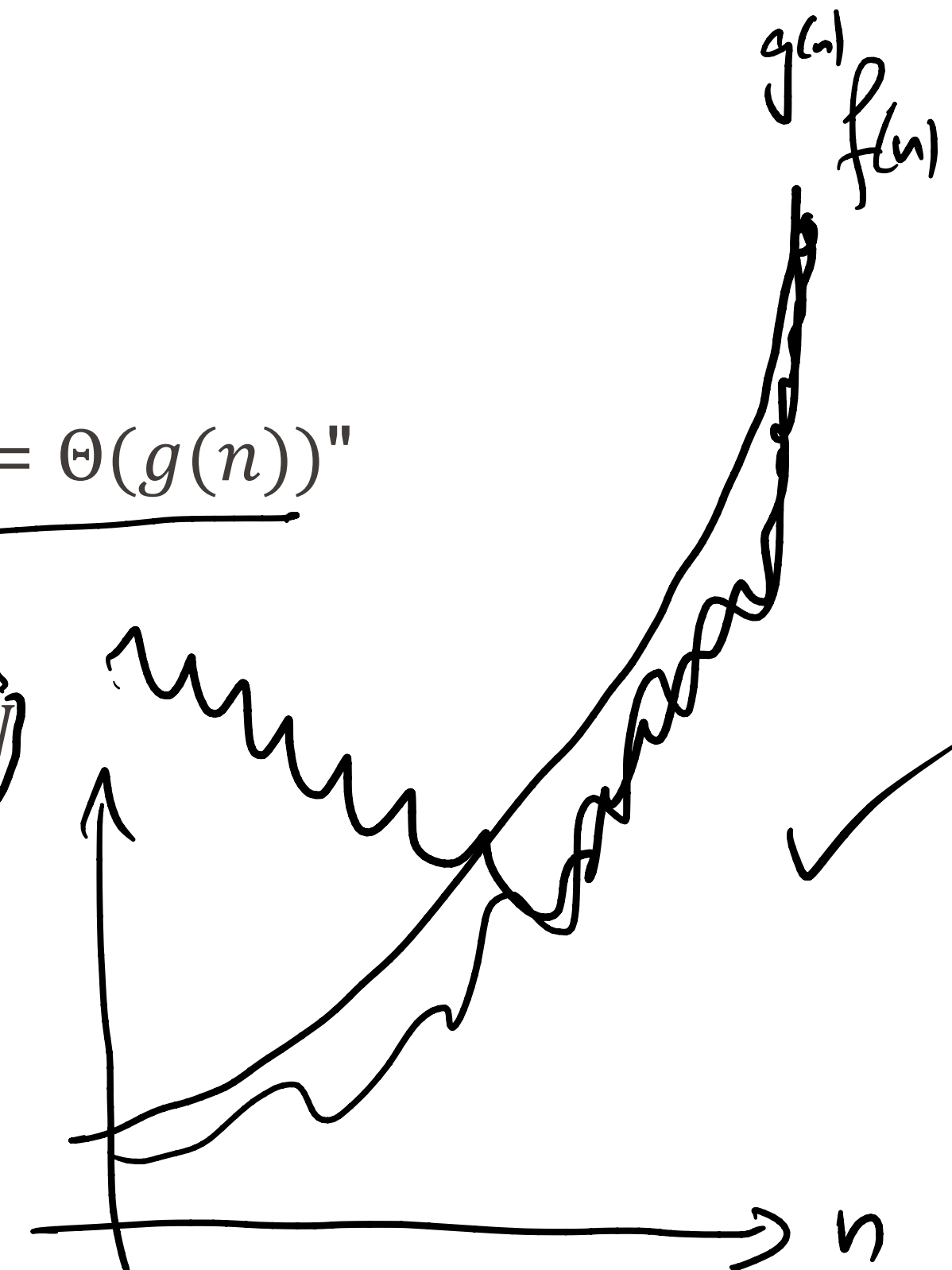
s'il existe $0 < C_1 < C_2 < \infty$ et $N \geq 1$ tels que

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

pour tout $n \geq N$

Ex: $f(n) = 2n$ & $g(n) = n$ ✓

Contre-ex: $f(n) = n^2$ & $g(n) = n$ ✗



Notation $\Theta(\cdot)$: définition

- Dans de nombreuses applications, on a affaire à des données d'entrée de **grande** taille.
- Dans ce cas, on aimerait obtenir des **ordres de grandeur** plutôt que de devoir faire des calculs détaillés.

Définition

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 $f(n) = \textcircled{n} + 2$ et $f(n) = 3\textcircled{n} + 3$ sont toutes deux des $\Theta(n)$ [cf. algorithme 2]

- La fonction $f(n) = \frac{n(n-1)}{2} + 1$ est un $\Theta(n^2)$ [cf. algorithme 3]

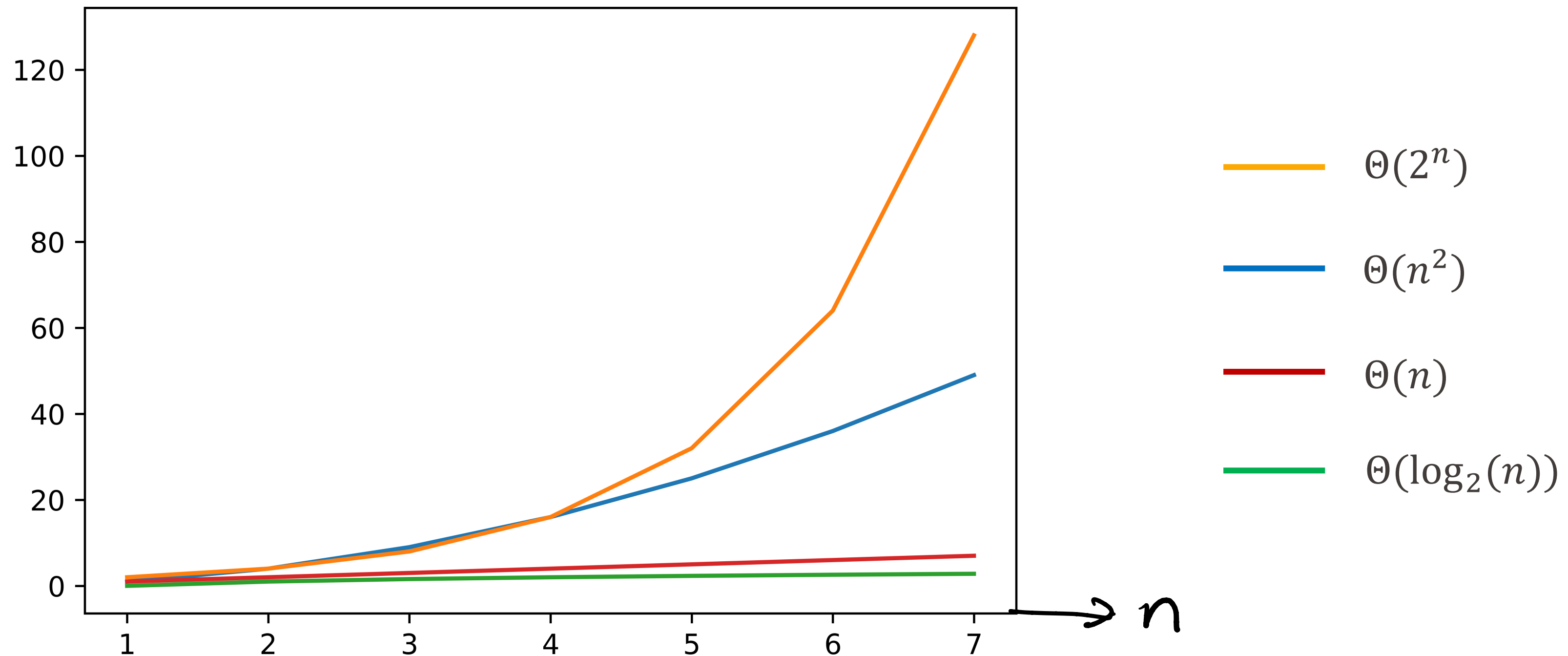
$$\underbrace{\frac{n(n-1)}{2}}_{\text{handwritten}} = \frac{n^2}{2} - \frac{n}{2} + 1 \longrightarrow \frac{n^2}{2} \longrightarrow \textcircled{n^2}$$

Notation $\Theta(\cdot)$: application

- Revenons à notre exemple :

Supposons qu'un algorithme prenne une minute pour s'exécuter avec des données d'entrée de taille $n = 1'000$. On aimerait savoir en combien de temps (au pire) s'exécutera ce même algorithme avec des données d'entrée de taille $n = 10'000$.

- Si la complexité temporelle de cet algorithme est un $\Theta(n)$, alors son temps d'exécution avec $n = 10'000$ en entrée vaudra (approximativement) 10 minutes.
- Si sa complexité temporelle est un $\Theta(n^2)$, alors son temps d'exécution avec $n = 10'000$ en entrée vaudra (approximativement) $10 \times 10 = 100$ minutes = 1 heure 40.



$n = 1000$: $\log_2(n) \approx 10$, $n = 1000$, $n^2 = 1'000'000$, $2^n \sim 10^{300}$

Calcul du nombre de paires d'éléments dans l'ensemble $\{1, \dots, n\}$

Pour calculer ce nombre, il existe plusieurs façons de faire :

- Utilisation de deux boucles imbriquées
→ complexité $\Theta(n^2)$
- Utilisation d'une seule boucle
→ complexité $\Theta(n)$
- Utilisation de la formule mathématique
→ complexité $\Theta(1)$

```
s ← 0
Pour i allant de 1 à n - 1 :
    Pour j allant de i + 1 à n :
        s ← s + 1
Sortir : s
```

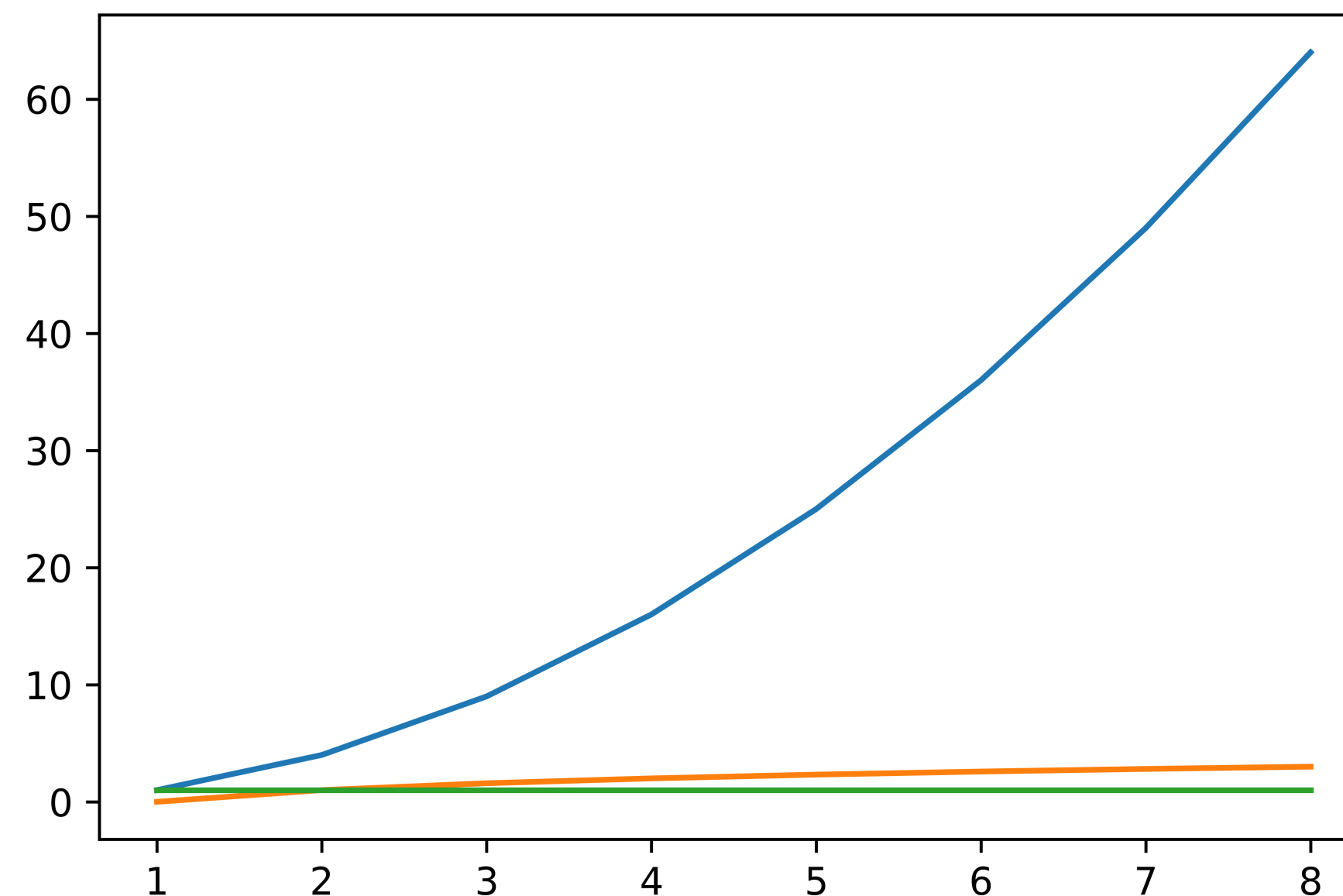
```
s ← 0
Pour i allant de 1 à n - 1 :
    s ← s + n - i
Sortir : s
```

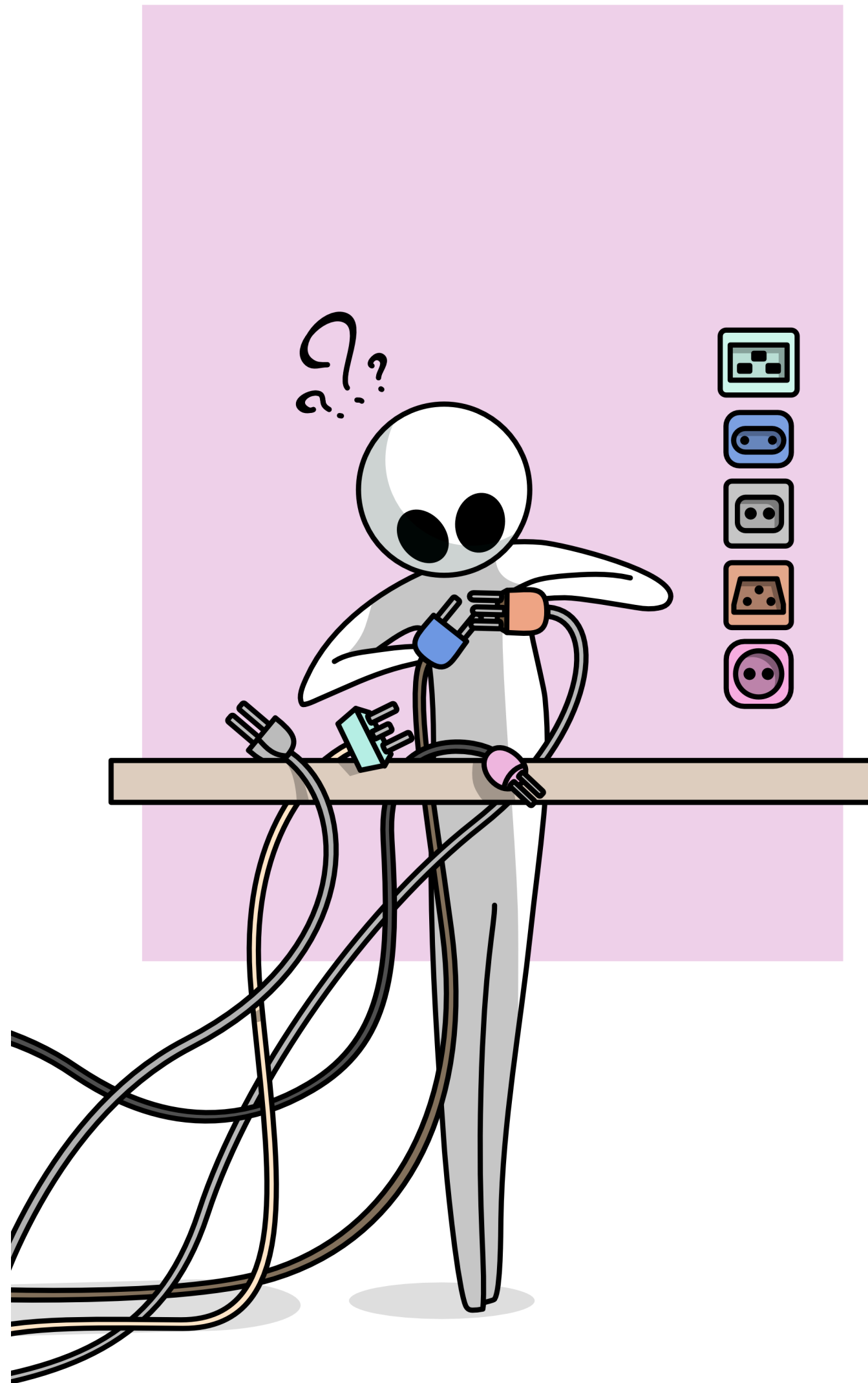
```
s ←  $\frac{n(n-1)}{2}$ 
Sortir : s
```

Calcul du nombre de paires d'éléments dans l'ensemble $\{1, \dots, n\}$

Pour calculer ce nombre, il existe plusieurs façons de faire :

- Utilisation de deux boucles imbriquées
→ complexité $\Theta(n^2)$
- Utilisation d'une seule boucle
→ complexité $\Theta(n)$
- Utilisation de la formule mathématique
→ complexité $\Theta(1)$





Information, Calcul et Communication

Complexité temporelle :
un (autre) exemple concret

Olivier Lévêque

Deux font la paire



Question : Parmi toutes les fiches et prises ci-dessus, y a-t-il une paire qui s'adapte l'une à l'autre?

Réécriture du problème avec des nombres entiers

En remplaçant les fiches et les prises par des nombres entiers positifs et négatifs, respectivement, la question précédente se transforme en :

- Etant donnée une liste L de n nombres entiers positifs et négatifs, existe-t-il $i, j \in \{1, \dots, n\}$ tels que $i < j$ et $L(i) + L(j) = 0$?

Exemple : Si $L = (-15, -12, -3, -1, +5, +17, +23)$, alors la réponse est non.

Note : Vu que nous avons affaire ici à des nombres entiers, nous allons supposer de plus que la liste L en entrée est *ordonnée*.

Première méthode de résolution

Etant donnée une liste L de n nombres entiers positifs et négatifs, existe-t-il $i, j \in \{1, \dots, n\}$ tels que $i < j$ et $L(i) + L(j) = 0$?

Deux font la paire

entrée : liste ordonnée L de nombres entiers
sortie : valeur binaire *oui / non*

$s \leftarrow non$

Pour i allant de 1 à $n - 1$:

 Pour j allant de $i + 1$ à n :

 Si $L(i) + L(j) = 0$, alors : $s \leftarrow oui$

Sortir : s

Complexité temporelle de cet algorithme: $\Theta(n^2)$

Les deux boucles imbriquées explorent toutes les paires possibles d'indices $i < j$ dans $\{1 \dots n\}$, qui sont au nombre de

$$(n - 1) + (n - 2) + \dots + 2 + 1 = \frac{n(n - 1)}{2}$$

donc la complexité temporelle de l'algorithme est $\Theta(n^2)$.

Question : Peut-on faire mieux ?

Deuxième méthode de résolution

Deux font la paire

entrée : liste ordonnée L de nombres entiers
sortie : valeur binaire *oui / non*

Pour i allant de 1 à $n - 1$:
 Pour j allant de $i + 1$ à n :
 Si $L(i) + L(j) = 0$, alors : Sortir : *oui*
Sortir : *non*

→ Complexité temporelle $\Theta(n^2)$ également : dans le pire des cas, l'algorithme doit parcourir toutes les paires (i, j) avant de sortir.

Remarque :

Aucun des deux algorithmes précédents n'exploite **l'ordre** de la liste L .

entrée : liste L , de taille n , ordonnée

Sortie : "est-ce que $\exists i < j$ tq $L(i) = L(j)$ " ?

trouver le "milieu" de la liste

(ie le nombre k tq $L(i) \leq 0 \quad \forall i \leq k$
 $L(j) \geq 0 \quad \forall j \geq k+1$)

Pour i allant de 1 à k

Pour j allant $k+1$ à n

si $L(i) + L(j) = 0$, sortir oui

Sortir non

$\left. \vphantom{\begin{matrix} L(i) \leq 0 \\ L(j) \geq 0 \end{matrix}} \right\} \Theta(n)$

+

$\left. \vphantom{\begin{matrix} \text{si } L(i) + L(j) = 0 \\ \text{Sortir oui} \end{matrix}} \right\} \Theta(k \cdot (n-k))$

$k = \frac{n}{2}$

$\Theta(n^2)$

Troisième méthode de résolution

Deux font la paire

entrée : liste ordonnée L de nombres entiers

sortie : valeur binaire *oui / non*

$i \leftarrow 1$

$j \leftarrow n$

Tant que $i < j$:

Si $L(i) + L(j) = 0$, alors : Sortir : *oui*

Si $L(i) + L(j) < 0$, alors : $i \leftarrow i + 1$

Si $L(i) + L(j) > 0$, alors : $j \leftarrow j - 1$

Sortir : *non*

→ Complexité $\Theta(n)$!

$n=6$

$L = (\cancel{-4}, -3, -1, +3, \cancel{+7}, \cancel{+10})$

$\uparrow \quad \uparrow \quad \quad \uparrow \quad \uparrow \quad \uparrow$
 $i=1 \quad i=2 \quad \quad j=6 \quad j=5 \quad j=6$

$$L(i) + L(j) = -4 < 0$$

$$\rightarrow i=2$$

$$L(i) + L(j) = +7 > 0$$

$$\rightarrow j=5$$

$$L(i) + L(j) = +4 > 0$$

$$\rightarrow j=4$$

$$L(i) + L(j) = 0 \rightarrow \underline{\underline{\text{oui}}}$$

- Pour un problème donné, il existe souvent *plusieurs* algorithmes de résolution différents.
- En général, des données d'entrée *structurées* permettent une résolution *plus efficace* du problème.