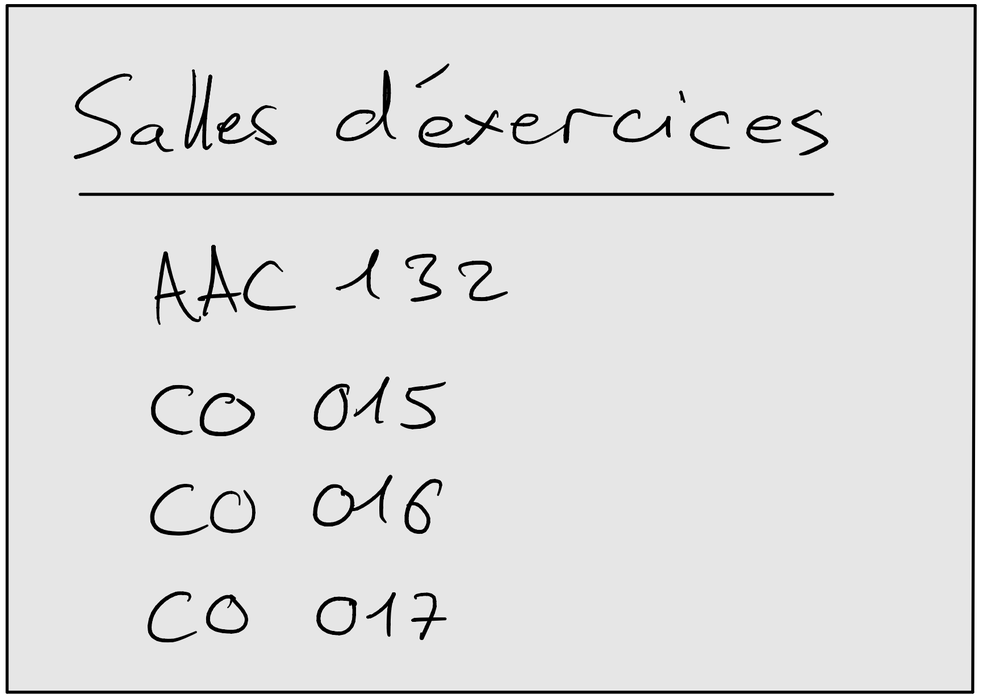
Rappel : ingrédients de base des algorithmes

Données

- Entrées
- Sorties
- Variables internes

Instructions

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



Information, Calcul et Communication

Sous-algorithmes

Olivier Lévêque

École polytechnique fédérale de Lausanne

Sous-algorithmes

Un problème récurrent : comment préparer ses valises en famille ?

Sous-algorithmes

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

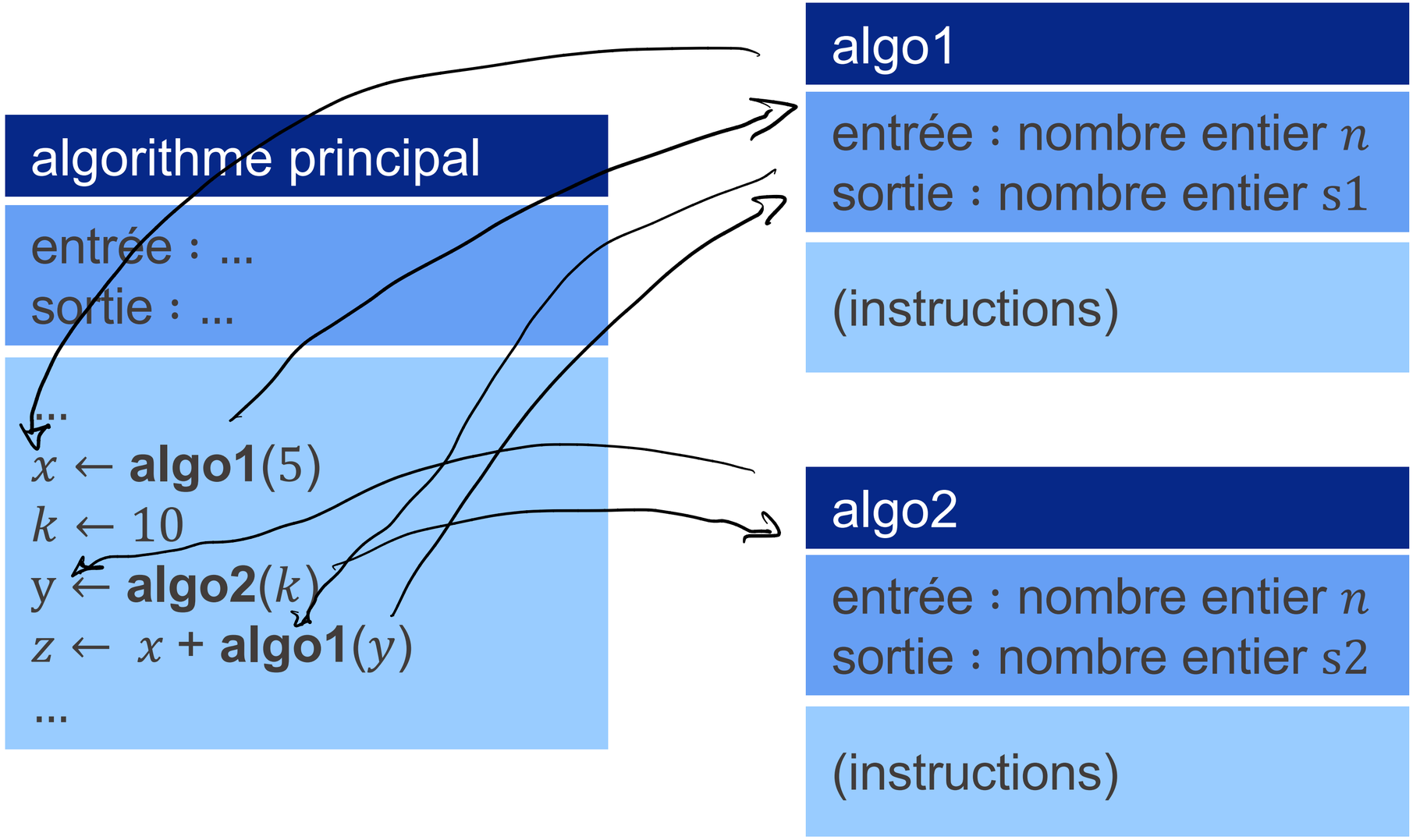


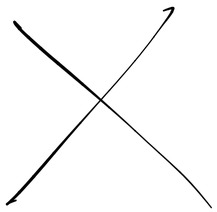
Illustration du principe avec le tri d'une liste

entrée : $L = (17, 18, 3, 45, 22, 12)$

$n = 6$

Tri d'une liste de nombres

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



algo

entrée : liste L, taille n

sortie : liste triée ?

Pour i allant de 1 à n-1 :

Si L(i) > L(i+1), alors permuter(L, i, i+1)

Sortir L

sortie : $L = (17, 3, 18, 22, 12, 45)$

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 par insertion: algorithme principal

Comp. temp. : $1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$
(dans le pire des cas)

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 :
    $L \leftarrow$ **insérer**(L, i)
Sortir : L

élément $L(i)$ mal placé

$L = (2, 7, 6, 5) \xrightarrow{i=2} L = (2, 7, 6, 5) \xrightarrow{i=3} L = (2, 6, 7, 5)$

$\xrightarrow{i=4} L = (2, 5, 6, 7)$

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 $j > 1$ & $L(j) < L(j-1)$:
  $L \leftarrow$ **permuter**$(L, j, j-1)$
  $j \leftarrow j - 1$
Sortir : L

$L = (3, 7, 5, 6)$ $i = 4$

Sortir $(2, 5, 6, 7)$

$L = (2, 6, 7, 5)$

$i = 4$

$j \leftarrow 4$

Tant que $j > 1$ ✓
$5 < 7$ ✓

$L \leftarrow (2, 6, 5, 7)$

$j \leftarrow 3$

Tant que $j > 1$ ✓
$5 < 6$ ✓

$L \leftarrow (2, 5, 6, 7)$

$j \leftarrow 2$

Tant $j > 1$ ✓
$5 < 2$ ✗

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 $j > 1$ & $L(j) < L(j-1)$:
    $L \leftarrow$ **permuter**$(L, j, j-1)$
    $j \leftarrow j - 1$
Sortir : L

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

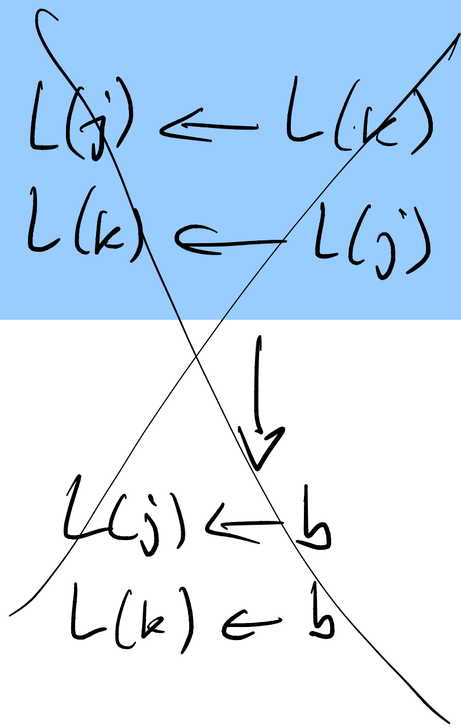
Tri par insertion: sous-algorithme 2

permuter

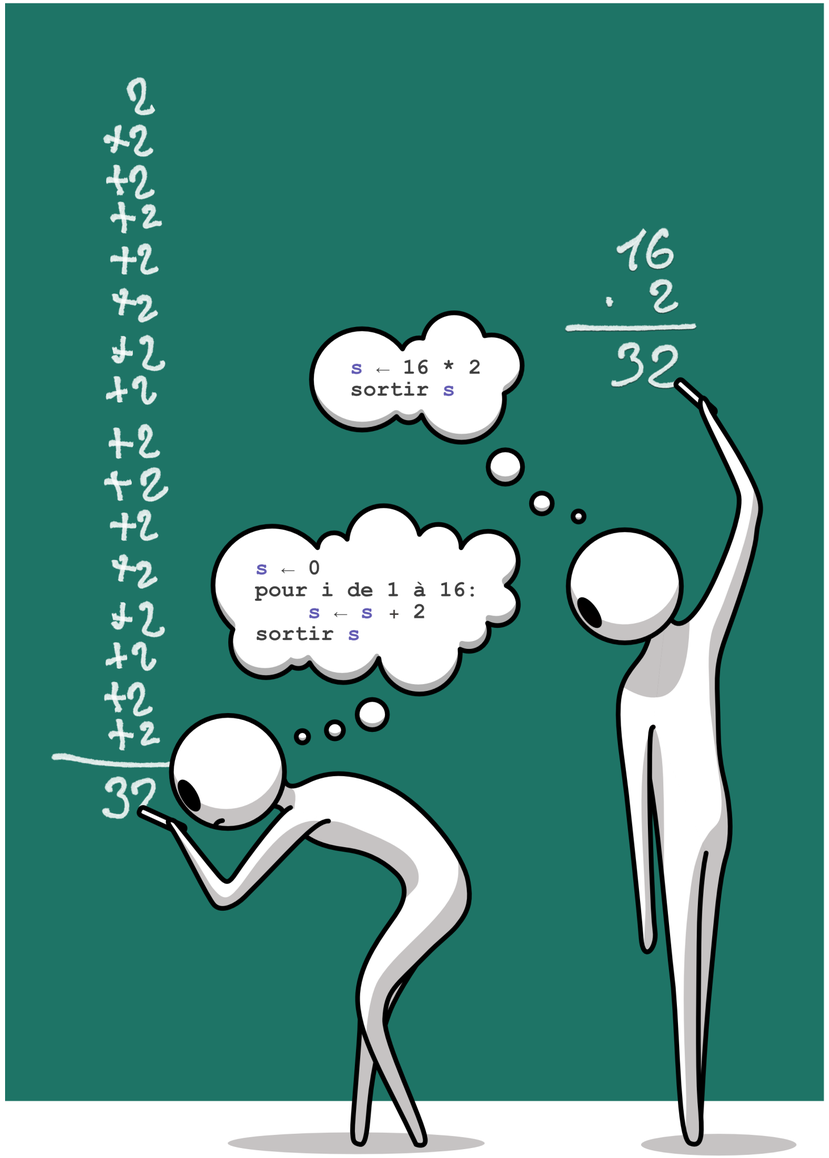
entrée : liste de nombres entiers L, nombres entiers positifs j, k
sortie : liste L avec les éléments $L(j)$ et $L(k)$ permutés

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

temp ← a
L(j) ← b
L(k) ← a



L(j) = a
L(k) = b



Information, Calcul et Communication

Algorithmes : complexité temporelle

École polytechnique fédérale de Lausanne

Olivier Lévêque

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)

Exemples

Algorithme 1

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

Comp. temp. infinie !

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

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

1
1
n
n
n
1

n+2 instructions

3n+3 instructions

Exemples

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. temp. : $\frac{n(n-1)}{2} + 1$

$i=1$: $j = 2, 3, 4, 5 \ldots n$ — $n-1$
$i=2$: $j = 3, 4, 5 \ldots n$ — $n-2$
$i=3$: $j = 4, 5, \ldots n$ — $n-3$
⋮ ⋮ ⋮
$i=n-1$: $j=n$ — 1

Nb de comparaisons $= (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$

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.

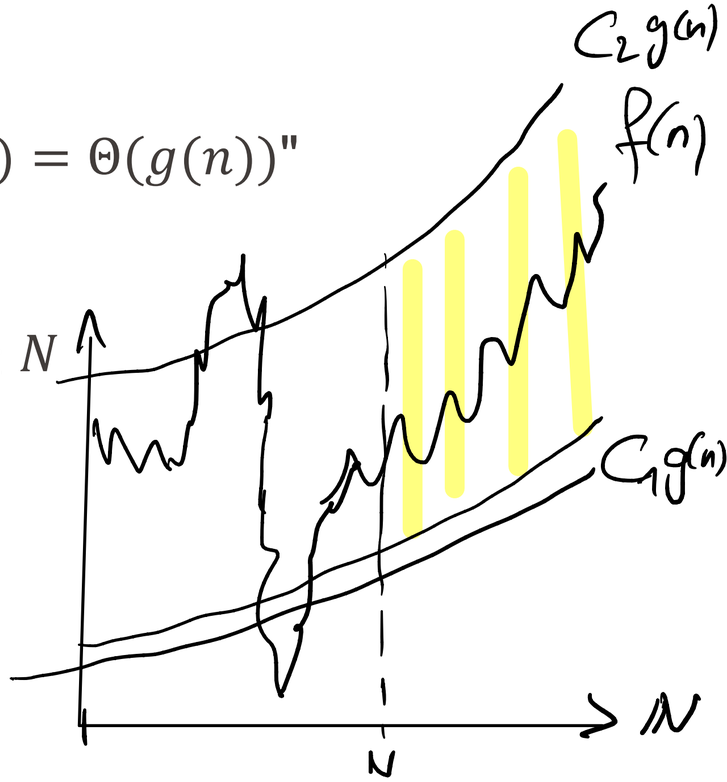
Notation $\Theta(\cdot)$: définition

- 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 $f, g : \mathbb{N} \to \mathbb{R}_+$ 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

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



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 $f, g : \mathbb{N} \to \mathbb{R}_+$ 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 $\Theta(n)$ [cf. algorithme 2]
- La fonction $f(n) = \frac{n(n-1)}{2} + 1$ est un $\Theta(n^2)$ [cf. algorithme 3]

$\neq \Theta(n^2)$

$\neq \Theta(n)$

Notation $\Theta(\cdot)$: application

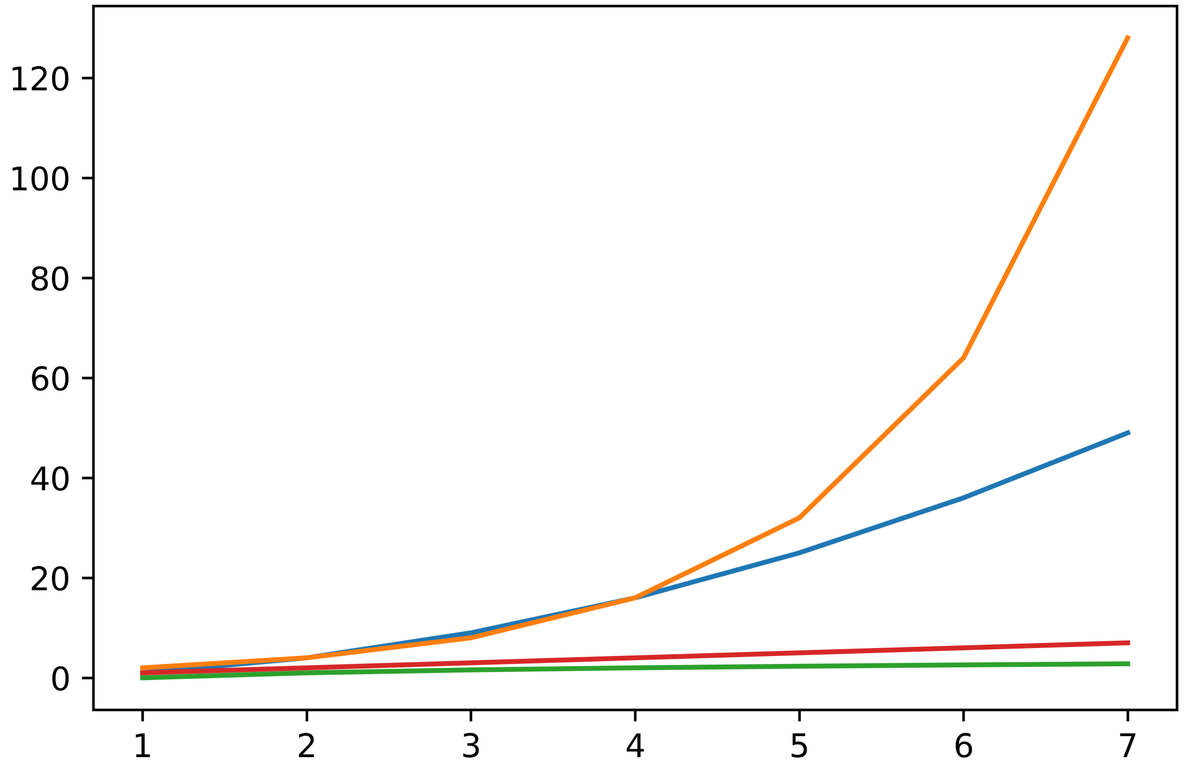
- Revenons à notre exemple :

  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 alors son temps d'exécution avec $n = 10'000$ en entrée vaudra (approximativement) $10\ x\ 10 = 100$ minutes = 1 heure 40.

Ordres de grandeur



$\Theta(2^n)$

$\Theta(n^2)$

$\Theta(n)$

$\Theta(\log_2(n))$

Ex : $n = 1000$: $\log_2(n) \sim 10$, $n = 10^3$, $n^2 = 10^6$, $2^n \sim 10^{300}$

Illustration

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 $\Theta(1)$

$s \leftarrow 0$
Pour i allant de 1 à $n-1$:
  Pour j allant de $i+1$ à n :
    $s \leftarrow s + 1$
Sortir : s

$s \leftarrow 0$
Pour i allant de 1 à $n-1$:
  $s \leftarrow s + n - i$
Sortir : s

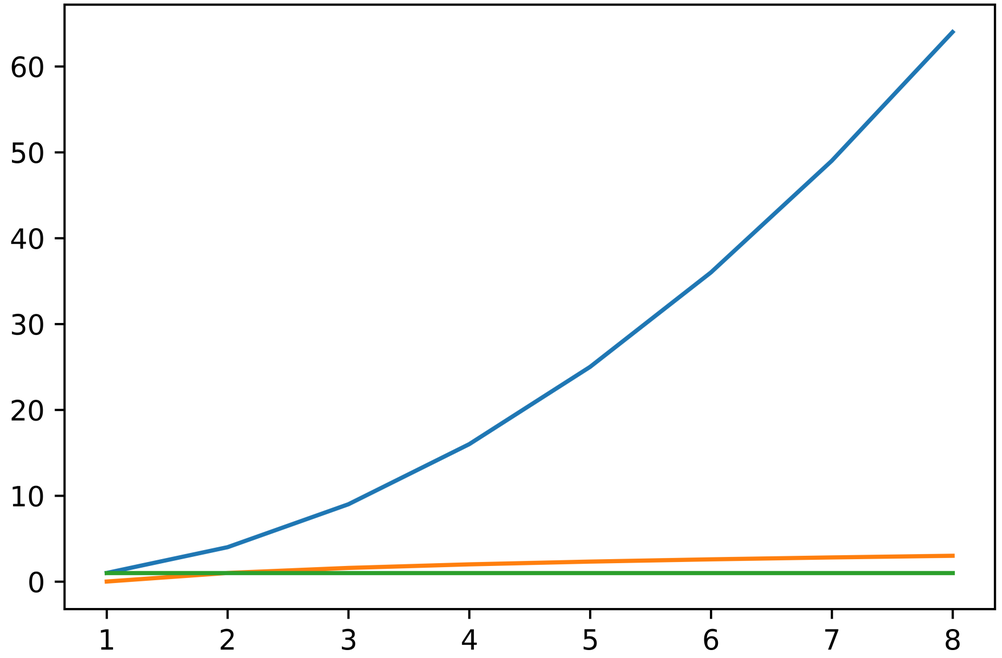
$s \leftarrow \frac{n(n-1)}{2}$
Sortir : s

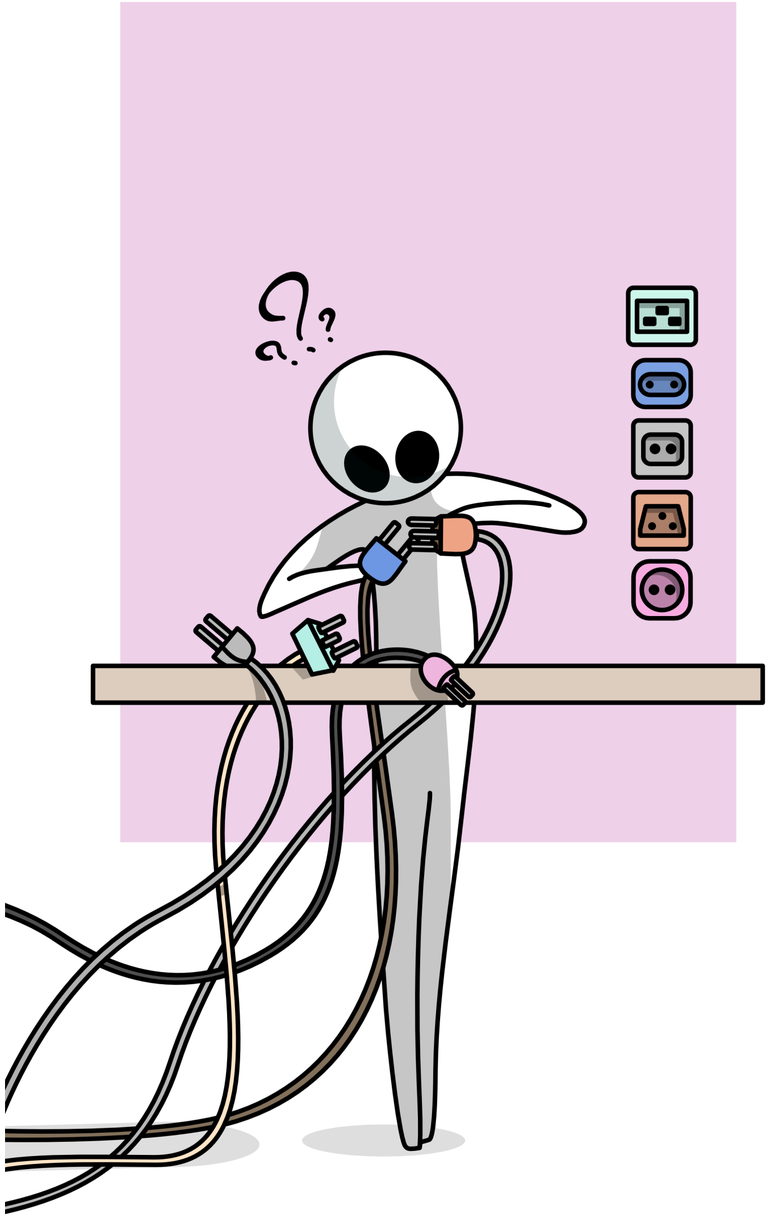
Illustration

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

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

École polytechnique fédérale de Lausanne

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 $i, j \in \{1, \ldots, n\}$ 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 $i, j \in \{1, \ldots, n\}$ 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

Première méthode de résolution

Etant donnée une liste L de n nombres entiers positifs et négatifs, existe-t-il $i, j \in \{1, \ldots, n\}$ 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

comp. temp $\Theta(n^2)$

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

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

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

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.

Mieux ?

NB : la liste L est ordonnée

$\Theta(n)$ { (a) Trouver la position k tq $\begin{cases} L(i) < 0 & \text{si } i < k \\ L(j) \geq 0 & \text{si } j \geq k \end{cases}$

$\Theta\left(\frac{n}{2} \cdot \frac{n}{2}\right) = \Theta(n^2)$ { (b) Pour i allant de 1 à k-1
Pour j allant de k à n
Si $L(i) + L(j) = 0$, sortir ok

Sortir non

pire des "k" : $k = \frac{n}{2}$

$$\Theta(n + n^2) = \Theta(n^2)$$

Encore mieux !

$L = (-13, -12, -5, -4, +3, +5, +17, +28)$

$-13+28 = +15$

$-13+17 = +4$

$-13+5 = -8$

$-12+5 = -7$

$-5+5 = 0$

→ $\Theta(n)$ op.

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 temporelle de ce dernier algorithme: $\Theta(n)$ (une seule boucle !)

Conclusion

- 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.