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

algorithme principal

entrée : ...
sortie : ...

...
$x \leftarrow$ **algo1**(5)
$k \leftarrow 10$
y $\leftarrow$ **algo2**(k)
$z \leftarrow$ x + **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 (L , n) ?

L = liste de nombres

= (14, 18, 12, 17, 3)

Pour i allant de 2 à n :

si L(i) < L(i-1),

alors permuter L(i) et L(i-1)

Sortir L

Sortie

i=2 : rien

i=3 : (14, 12, 18, 17, 3)

i=4 : (14, 12, 17, 18, 3)

i=5 : (14, 12, 17, 3, 18)

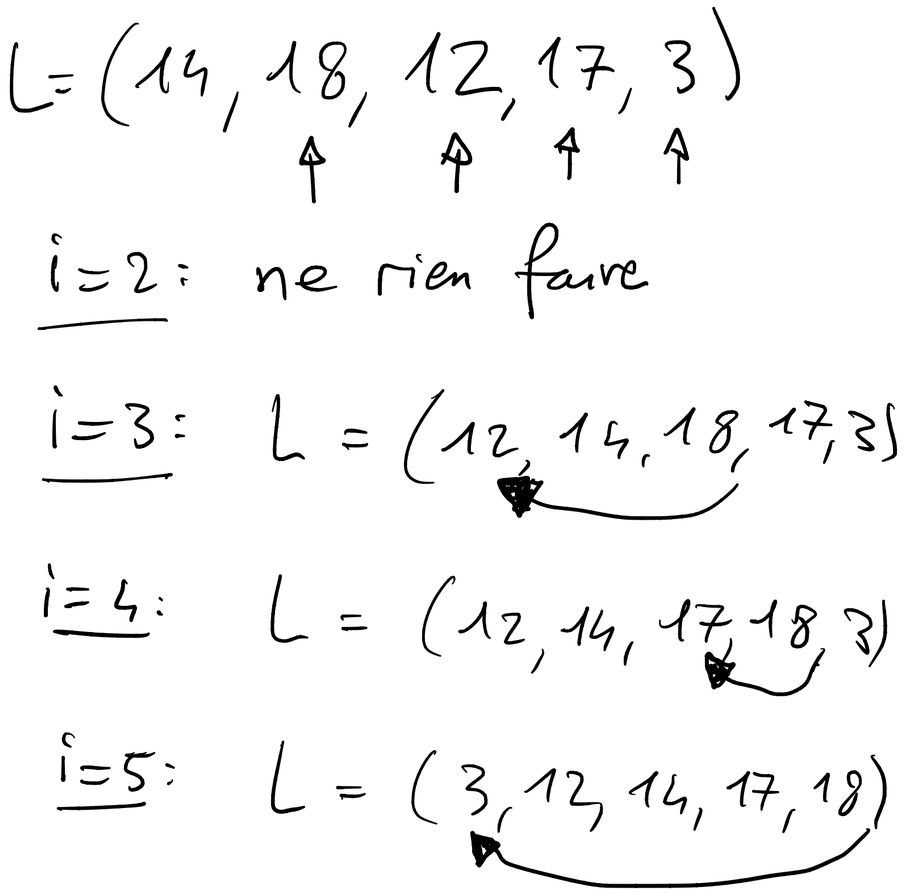
pas un bon algorithme !

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



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

$i=3$

$L = (14, 18, 12, 17, 3)$

$j=3$

$j>1$? & $12<18$? oui

$L \leftarrow (14, 12, 18, 17, 3)$

$j \leftarrow 2$

$j>1$? & $12<14$? oui

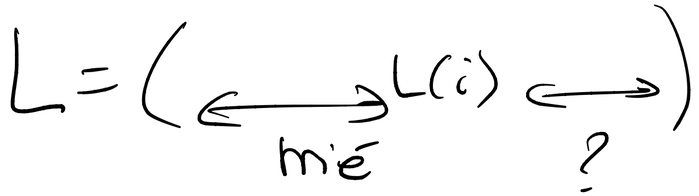
$L \leftarrow (12, 14, 18, 17, 3)$

$j \leftarrow 1$

$j>1$? non

Sortir $L = (12, 14, 18, 17, 3)$

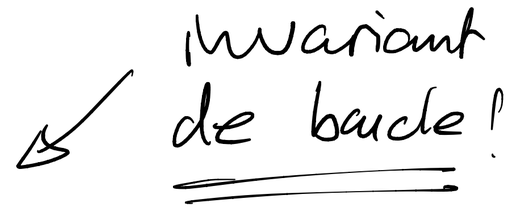
Tri par insertion: sous-algorithme 1



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

$temp \leftarrow L(j)$ temp ← 2
$L(j) \leftarrow L(k)$ L(j) ← 3
$L(k) \leftarrow temp$ L(k) ← 2
Sortir : L

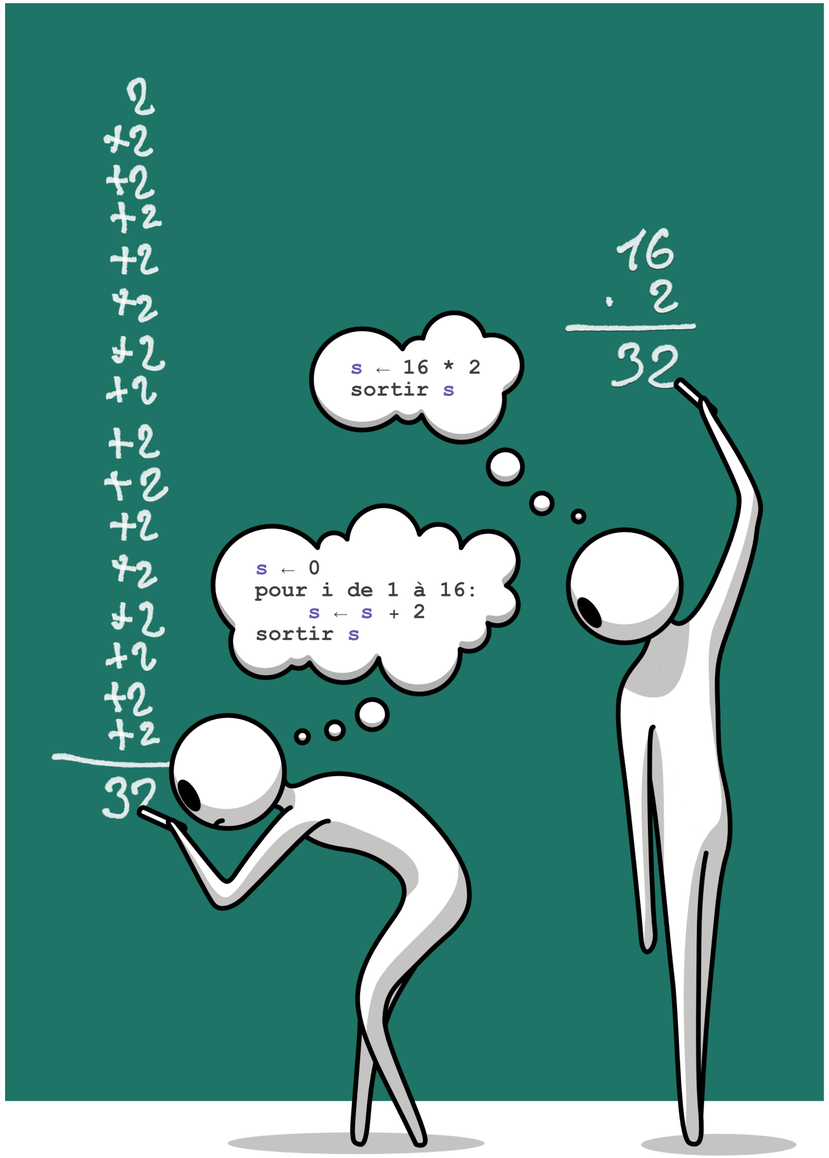
L(j) = 2, L(k) = 3

L(j) = 3, L(k) = 2 ✓

NB : { L(k) ← L(j)
{ L(j) ← L(k)

Pas bon !

L(j) = 2, L(k) = 2 ✗



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"
```

Complexité 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

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

complexité temporelle

$n+2$ $(\sim n)$

$3n+3$?

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*

$i=1$: $n-1$ valeurs de j
$i=2$: $n-2$ valeurs de j
⋮
$i=n-1$: 1 valeur de j

Complexité temporelle : $\underbrace{\frac{n(n-1)}{2}}_{1+2+3+\dots+(n-1)} + 1 \sim n^2$

taille des données d'entrée $n = 1'000$

⟶ algorithme s'exécutant en 1 minute

et pour des données d'entrée de taille $n = 10'000$?

1. Algorithme de complexité temporelle n.

⟶ temps d'exécution : 10 minutes

2. Algorithme de complexité temporelle n^2

⟶ $1'000^2 = 1'000'000$ opérations ⟷ 1 minute

$10'000^2 = 100'000'000$ opérations ⟷ 100 minutes

3. Algorithme de complexité temporelle $\frac{n^2}{2}$

pour $n = 1'000$ $\longrightarrow$ 1 minute de temps d'exec.

$\frac{n^2}{2} = 500'000$ opérations

pour $n = 10'000$ $\longrightarrow$ $X = 1 \cdot \frac{50'000'000}{500'000} = 100$ minutes

$\frac{n^2}{2} = 50'000'000$ opérations

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

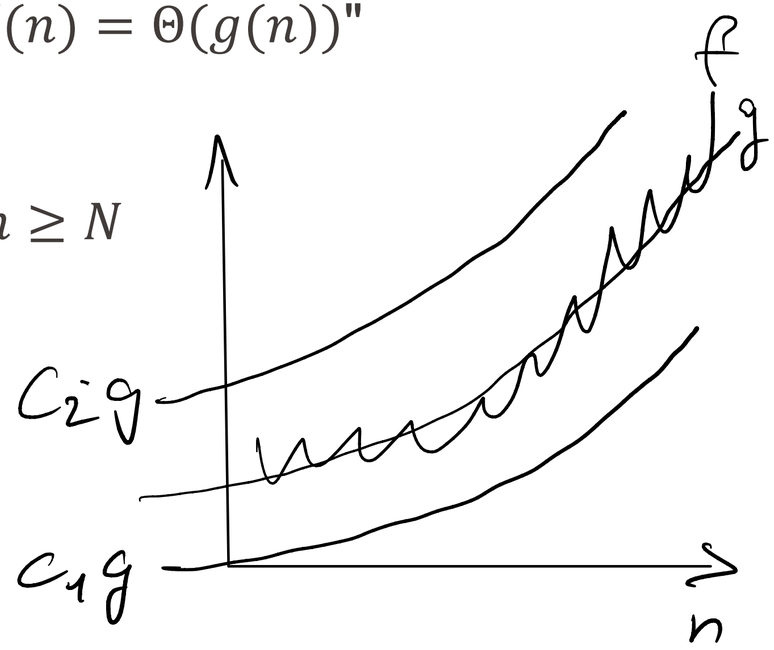
Soient $f, g : \mathbb{N} \to \mathbb{R}_+$ deux fonctions non-négatives
On dit que "$f(n)$ est un **grand theta** de $g(n)$" et on écrit "$f(n) = \Theta(g(n))$"
s'il existe $0 < C_1 < C_2 < \infty$ et $N \geq 1$ tels que

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

en particulier : $f(n) = n+3$ est un $\Theta(n)$

$f(n) = 2n$ est un $\Theta(n)$

$f(n) = \frac{n}{2}$ est un $\Theta(n)$



Notation $\Theta(\cdot)$: définition

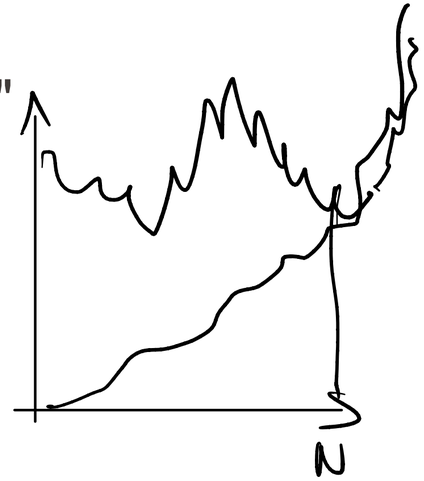
- Dans de nombreuses applications, on a affaire à des données d'entrée de **grande** taille.

- 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

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



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]

$\frac{n^2}{4} \leq f(n) \leq n^2 \quad \forall n$

Notation $\Theta(\cdot)$: application

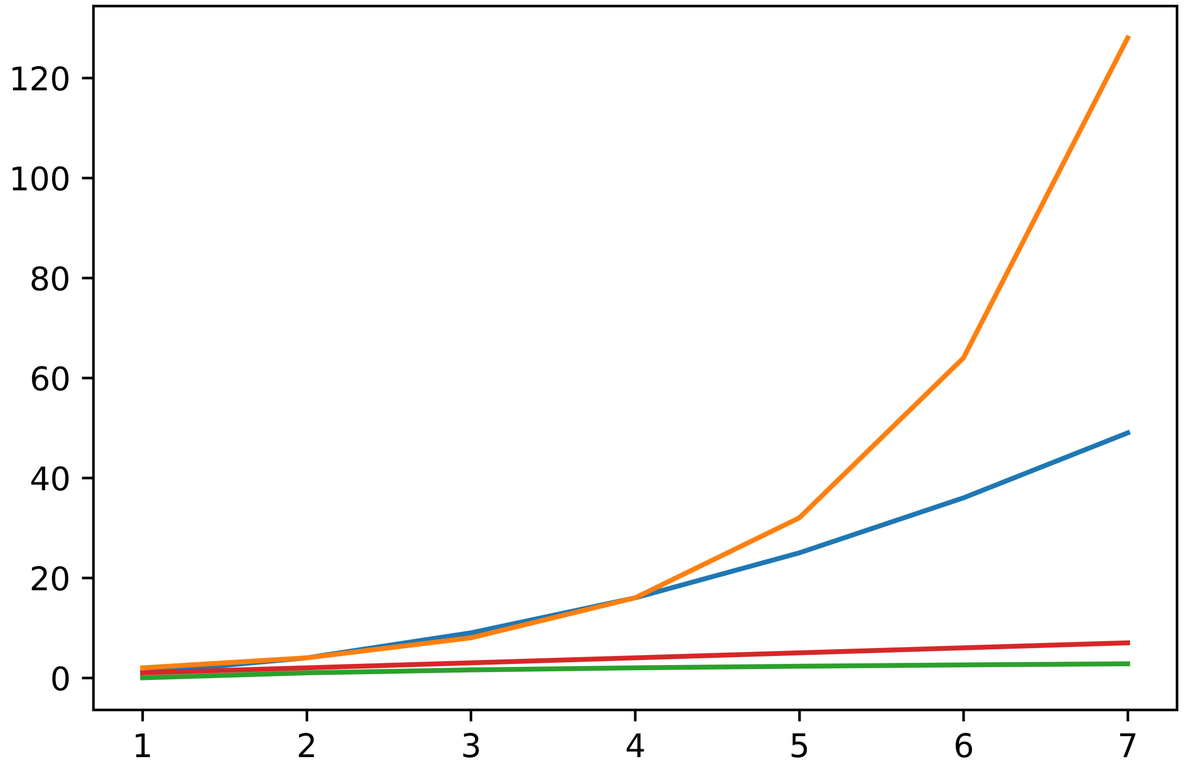
- Revenons à notre exemple :

  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



$\Theta(2^n)$

$\Theta(n^2)$

$\Theta(n)$

$\Theta(\log_2(n))$

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

Problème : Etant donné une liste L de nombres entiers relatifs, de taille n

trouver $\min_{1 \leq k \leq n} L(1) + \dots + L(k)$

Ex : $L = (-3, -5, +6, -4)$

$\longrightarrow$ réponse $-3-5 = -8$ ($k=2$)

Solution 0 ? non

min ← L(1)

Pour i allant de 2 à n

si L(1)+...+L(i) < min,

alors min ← L(1)+...+L(i)

Sortir min

?

Solution 1 :

Complexité temporelle $\Theta(n^2)$

```
min ← L(1)
Pour i allant de 2 à n
    somme ← 0
    Pour j allant de 1 à i
        somme ← somme + L(j)
    Si somme < min, alors min ← somme
Sortir min
```

Solution 2 :

Complexité $\Theta(n)$

```
min ← L(1)
somme ← L(1)
Pour i allant de 2 à n :
    somme ← somme + L(i)
    Si somme < min, alors min ← somme
Sortir min
```

Autre ex :

Etant donné L une liste de n nombres entiers relatifs triés dans l'ordre croissant, identifier si oui ou non, il existe $i<j$ (avec $i, j \in \{1 \dots n\}$) tels que $L(i) + L(j) = 0$.

Ex : $L = (-17, -12, -5, -3, +4, +22)$

$\longrightarrow$ réponse : non

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

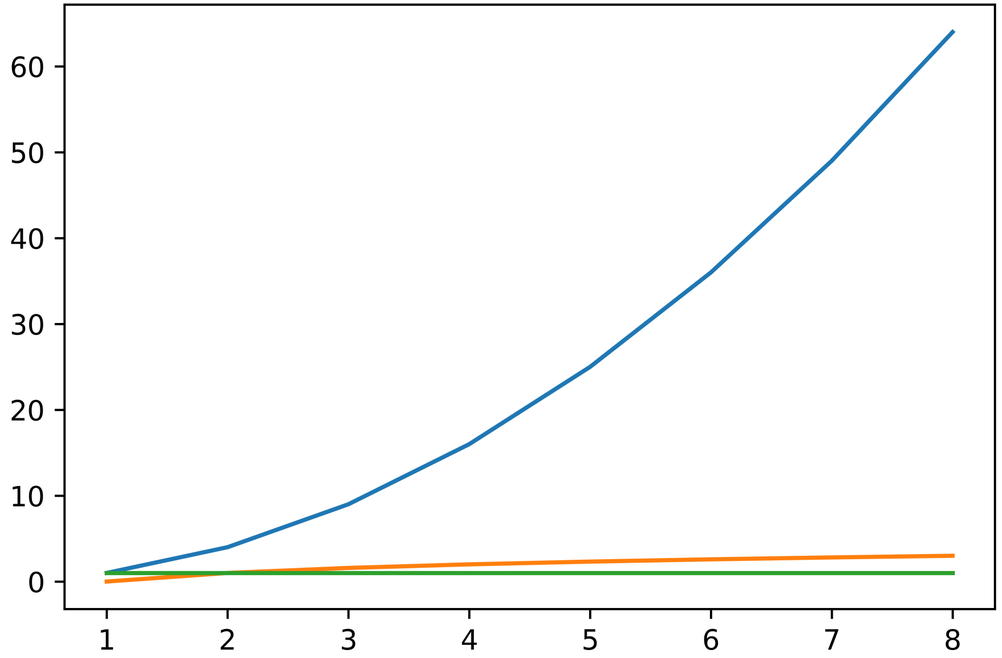
$s \leftarrow \frac{n(n-1)}{2}$
Sortir : s

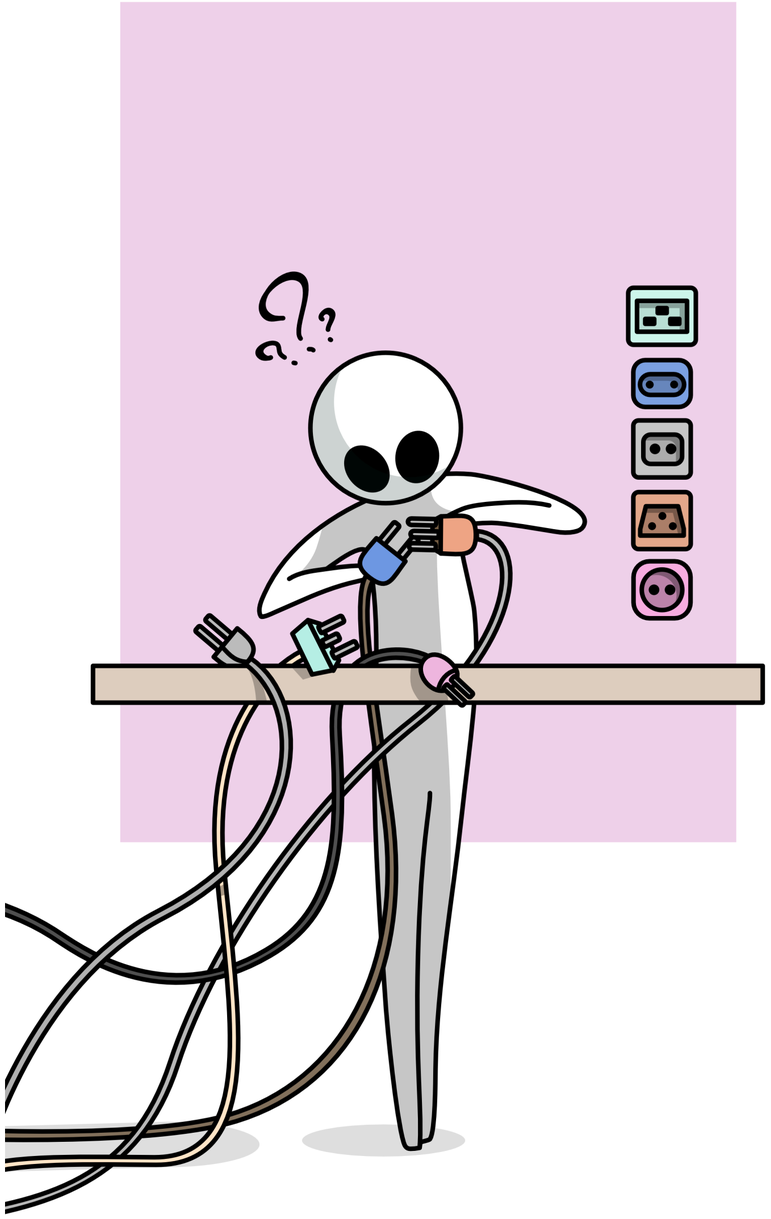
Illustration

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

$s \leftarrow non$
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 $\{1 \ldots n\}$, 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.

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.