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 …)

$L = (13, 3, 14, 12, 22, 13, 11)$ $n = 7$ nombres

Tri simple ?

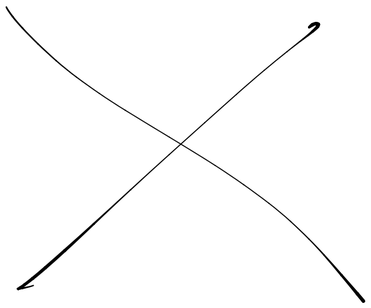
entrée : liste L, taille n

sortie : liste L 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



→ L = (3, 13, 12, 14 ------)

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

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 mal placé en position i

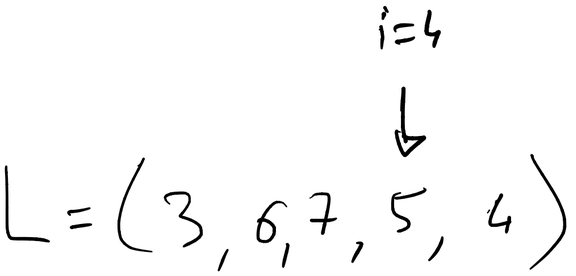
position de l'élément mal placé

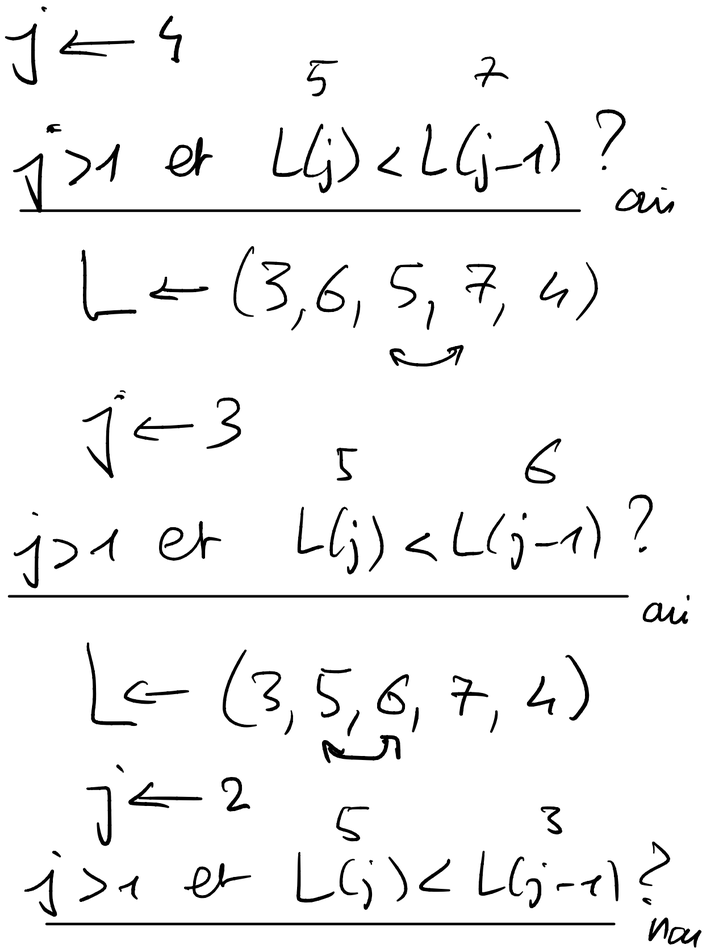
Tri par insertion: sous-algorithme 1

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





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

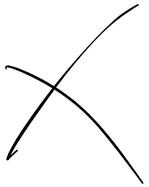
$temp \leftarrow L(j)$
$L(j) \leftarrow L(k)$
$L(k) \leftarrow temp$
Sortir : L

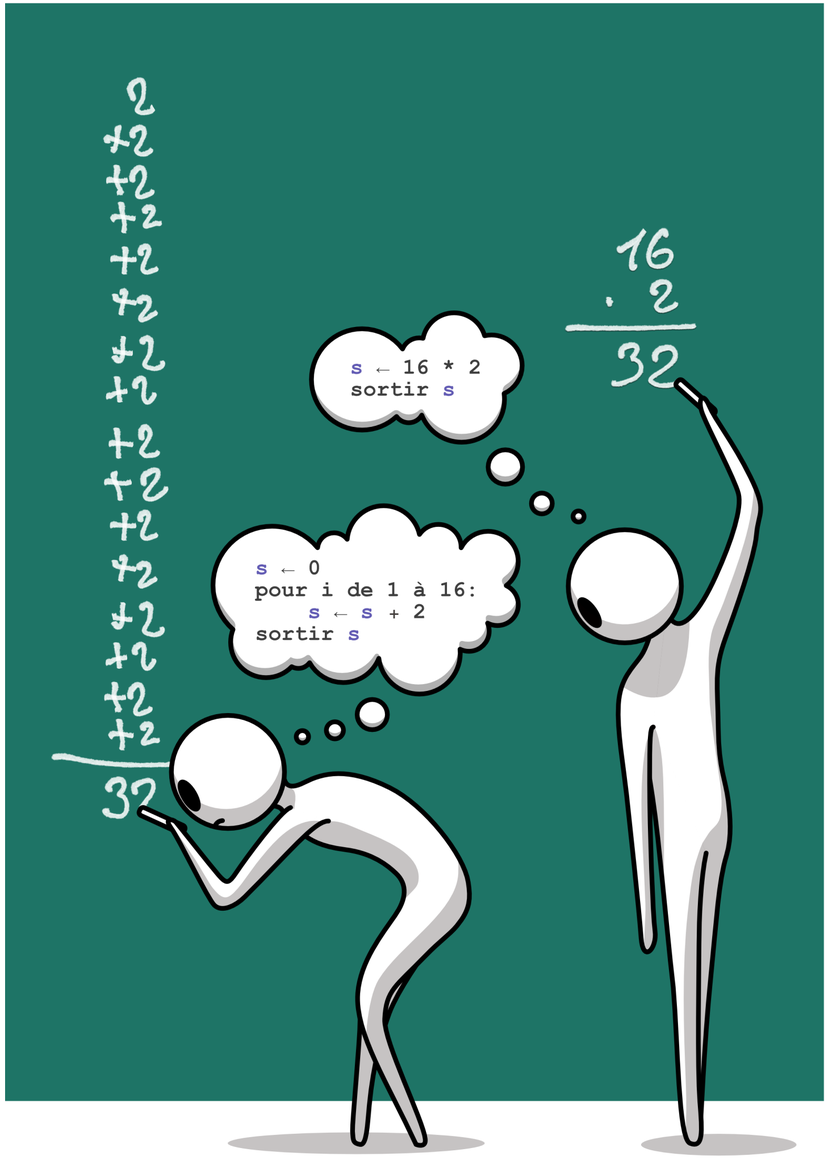
temp ← 4
L(j) ← 5
L(k) ← 4

L(j) ← L(k)
L(k) ← L(j)

L(j) = 4
L(k) = 5

? L(j) ← 5
L(k) ← 5





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"
```

$cstr 1

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

1 $m \leftarrow 0$
2n Pour i allant de 1 à n :
    $m \leftarrow m + L(i)$
1 Sortir : m/n

$2n + 2$

m←0 1
i←1 1
Tant que i≤n
    m←m+L(i) } 3n
    i←i+1
Sortir m/n 1

?

$3n + 3$

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*

$\sim n^2$

= nombre de passages dans le cœur des 2 boucles imbriquées

$i=1$: $j=2 \ldots j=n$ — $n-1$ valeurs de j

$i=2$: $j=3 \ldots j=n$ — $n-2$ valeurs de j

⋮

$i=n-1$: $j=n$ — 1 valeur de j

Au total :

$(n-1)+(n-2)+\ldots+1$

$= \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} + 1$

Notation $\Theta(\cdot)$: motivation

- 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).

- 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** (en n) plutôt que de devoir faire des calculs détaillés.

- 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 $n = 10'000$.

- La réponse à cette question dépend de la complexité temporelle de l'algorithme ! Considérons trois exemples.

Notation $\Theta(\cdot)$: motivation

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 $n = 10'000$.

- Exemple 1: algorithme de complexité temporelle n.

$n = 1'000 \quad \hookrightarrow \quad 1$ minute

$n = 10'000 \quad \hookrightarrow \quad x$ minutes

$\rightarrow x = 10$ minutes

Notation $\Theta(\cdot)$: motivation

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 $n = 10'000$.

- Exemple 1: algorithme de complexité temporelle n.

- Exemple 2: algorithme de complexité temporelle n^2.

$n = 1'000 \leftrightarrow n^2 = 1'000'000$ opérations $\leftrightarrow$ 1 minute

$n = 10'000 \leftrightarrow n^2 = 100'000'000$ opérations $\leftrightarrow$ x minutes

$\Longrightarrow x = 100$ minutes

Notation $\Theta(\cdot)$: motivation

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 $n = 10'000$.

- Exemple 1: algorithme de complexité temporelle n.
- Exemple 2: algorithme de complexité temporelle n^2.
- Exemple 3: algorithme de complexité temporelle $n^2/2$.

$n = 1000 \iff 500\,000$ opérations $\iff$ 1 minute

$n = 10'000 \iff 50'000'000$ opérations $\iff$ x minutes

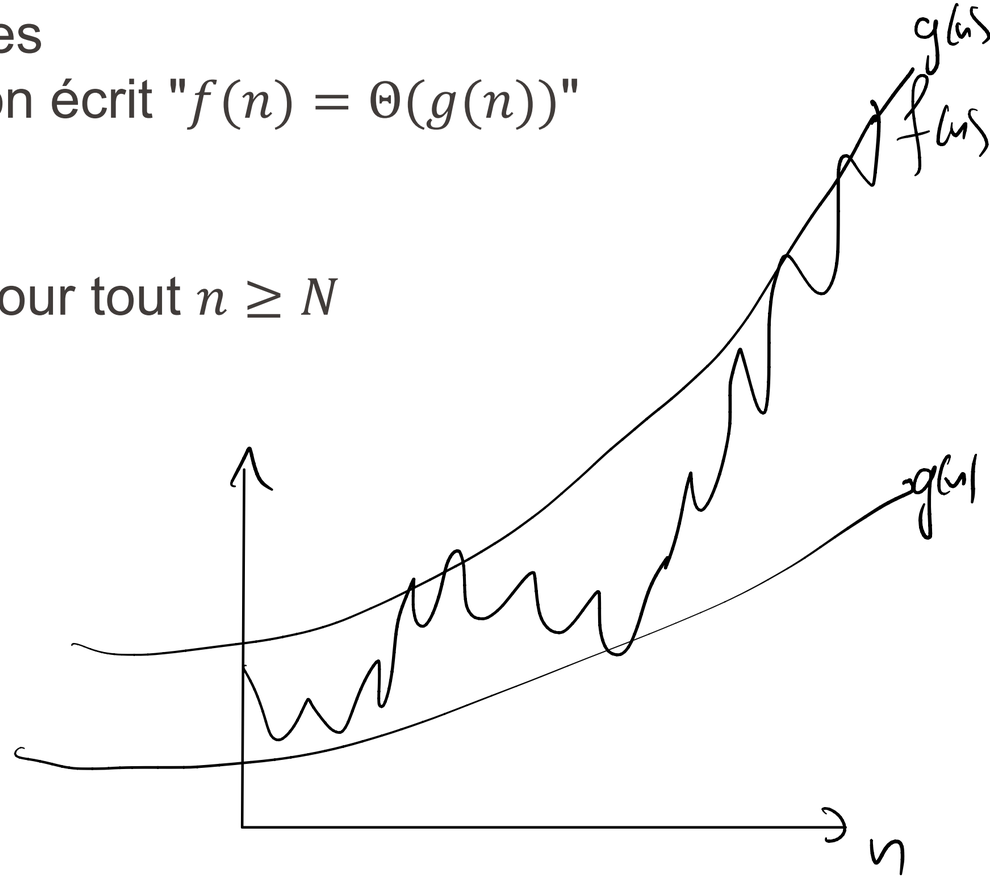
$\Rightarrow x = 100$ minutes

Notation $\Theta(\cdot)$: définition

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

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

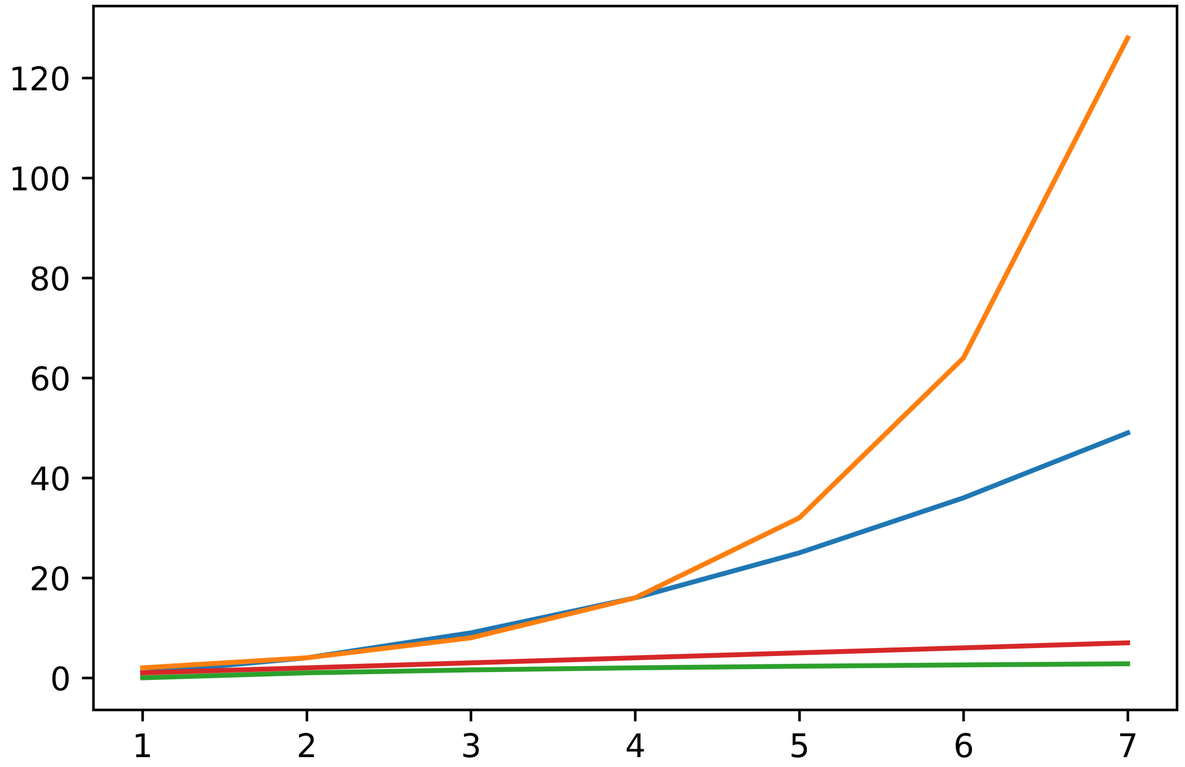
$$C_1\, g(n) \leq f(n) \leq C_2\, g(n) \quad \text{pour tout } n \geq N$$

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]

Tri par insertion → comp. temp. = $\Theta(n^2)$

Ordres de grandeur



$\Theta(2^n)$

$\Theta(n^2)$

$\Theta(n)$

$\Theta(\log_2(n))$

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

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

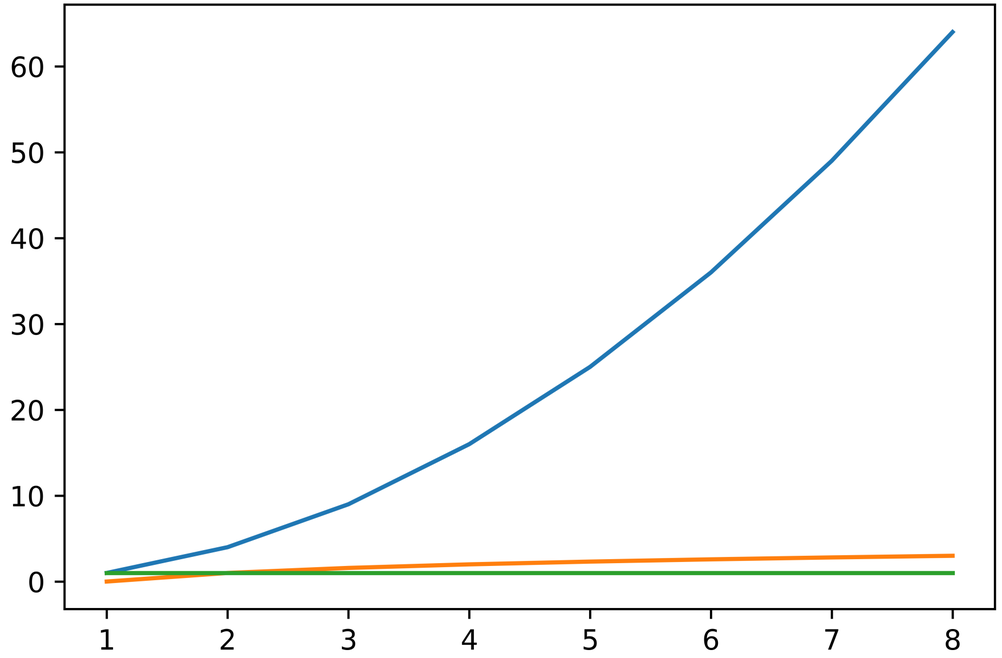
$s \leftarrow \frac{n(n-1)}{2}$
Sortir : s

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 $\Theta(1)$



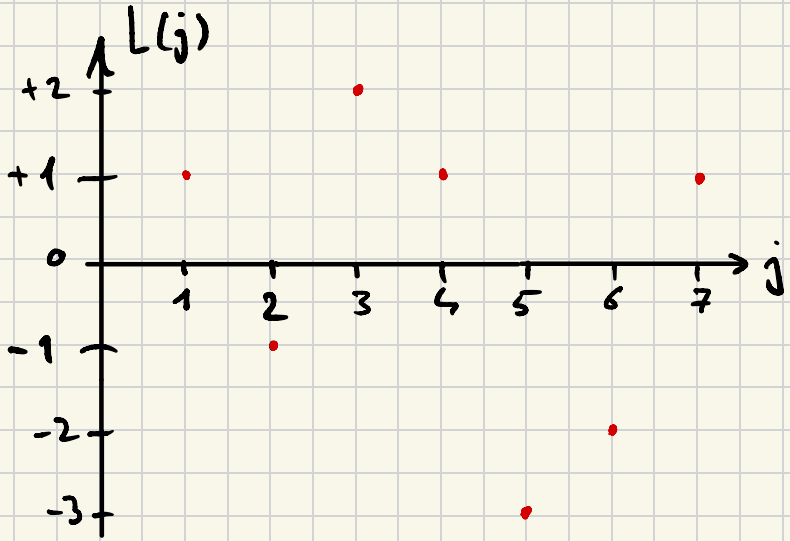
de nbs entiers relatifs

Problème : Etant donné une liste L ↑ de taille n trouver le minimum parmi toutes les sommes partielles

$$L(1) + L(2) + \dots + L(k) \quad \text{avec } 1 \leq k \leq n$$

Ex:

$n = 7$



$L(1) = +1$

$L(1) + L(2) = 0$

$L(1) + \dots + L(3) = +2$

$L(1) + \dots + L(4) = +3$

$L(1) + \dots + L(5) = 0$

$L(1) + \dots + L(6) = -2$ ✓

$L(1) + \dots + L(7) = -1$

Algo 1 (L, n)

???

min ← L(1)

Pour i allant de 2 à n

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

alors min ← L(1)+...+L(i)

Sortir min

Problème: "..." n'existe pas en algorithmique ! (non plus en Python)

Algo 2 (L, n)

$min \leftarrow L(1)$

Pour i allant de 2 à n

$S \leftarrow L(1)$

Pour j allant de 2 à i] calcul de la somme partielle $L(1)+\dots+L(i)$

$S \leftarrow S + L(j)$

Si $S < min$,

alors $min \leftarrow S$

Sortir min

Problème : Complexité $\Theta(n^2)$

(2 boucles imbriquées)

Algo 3 (L, n)

Complexité : $\Theta(n)$

M ← ( ) (création d'une liste vide M)

s ← 0

Pour i allant de 1 à n

    s ← s + L(i)

    M ← M ⊕ (s) (on ajoute l'élément s à M)

Sortir min(M)

↑

algo de recherche du minimum de la liste M (vu au premier cours)

Algo 4 (L, n)

```
S ← L(1)
min ← L(1)
Pour i allant de 2 à n
    S ← S + L(i)
    si S < min,
        alors min ← S
Sortir S
```

Complexité : $\Theta(n)$

Petit avantage :

Évite la création d'une autre liste M contenant les sommes partielles (cf. Algo 3)

Algo 4 (L, n) [version légèrement différente faite en cours]

$z \leftarrow L(n)$

$min \leftarrow z$

Pour i allant de 2 à n

$z' \leftarrow z + L(i)$

si $z' < z$, alors $min \leftarrow z'$

$z \leftarrow z'$

Sortir min

Complexité : $\Theta(n)$