Information, Calcul et Communication

CS-119(k) ICC – Théorie
Semaine 2

Rafael Pires
rafael.pires@epfl.ch

Lausanne, 28.02.2025

Précédemment, dans ICC-T 01

- Les ingrédients de base des algorithmes
  - Données (variables)
  - Instructions (affectations, structures de contrôle)
- Problèmes :
  - Calcul du modulo 3 d'un grand nombre
  - Recherche du minimum dans une liste
  - Problème du voyageur du commerce
  - Comparaison tous contre tous
  - L'algorithme d'Euclide (pgdc)

EPFL

Précédemment, dans ICC-T 01

- Les ingrédients de base des algorithmes
  - **Données** (variables)
  - **Instructions** (affectations, structures de contrôle)

Branchements

si delta < 0, alors …
sinon …

Boucles

pour i allant de 1 à n, (répeter …)

tant que $i \leq 10$, (répeter …)

Précédemment, dans ICC-T 01

Boucles

Itérations

pour i allant de 1 à 10,
(répeter ...)

itère **sur une séquence** définie à l'avance

Boucles conditionnelles

$i \leftarrow 1$
tant que $i \leq 10$
(répeter ...)
$i \leftarrow i + 1$

condition peut être **n'importe quelle expression booléenne**

EPFL

Attention !

Boucles

Indices et comparateur d'inégalité

ICC-T : vendredi $\neq$ **ICC-P : lundi**

Itération		Boucle conditionnelle
pour i allant de 1 à 10, (répeter ...)	$\equiv$	$\mathrm{i} \leftarrow 1$ tant que $i \leq 10$ (répeter ...) $\mathrm{i} \leftarrow \mathrm{i} + 1$

Itération		Boucle conditionnelle
for i in range(0,10): (répeter ...)	$\equiv$	$\mathrm{i} \leftarrow 0$ tant que $i < 10$ (répeter ...) $\mathrm{i} \leftarrow \mathrm{i} + 1$

Nombre de itérations

pour i allant de 1 à *n*,
(répeter ...)

$n - 1 + 1 = n$

$=$

for i in range(0,*n*):
(répeter ...)

$n - 0 = n$

EPFL

Programme du cours

Vous êtes ici

Cours et séries, partie programmation

P	1	2	3	4	5	6	7	8	9		10	11	12	13	14
T	1	2	3	4	5	6	7	8			10	11	12	13	14

Cours et séries, partie théorique

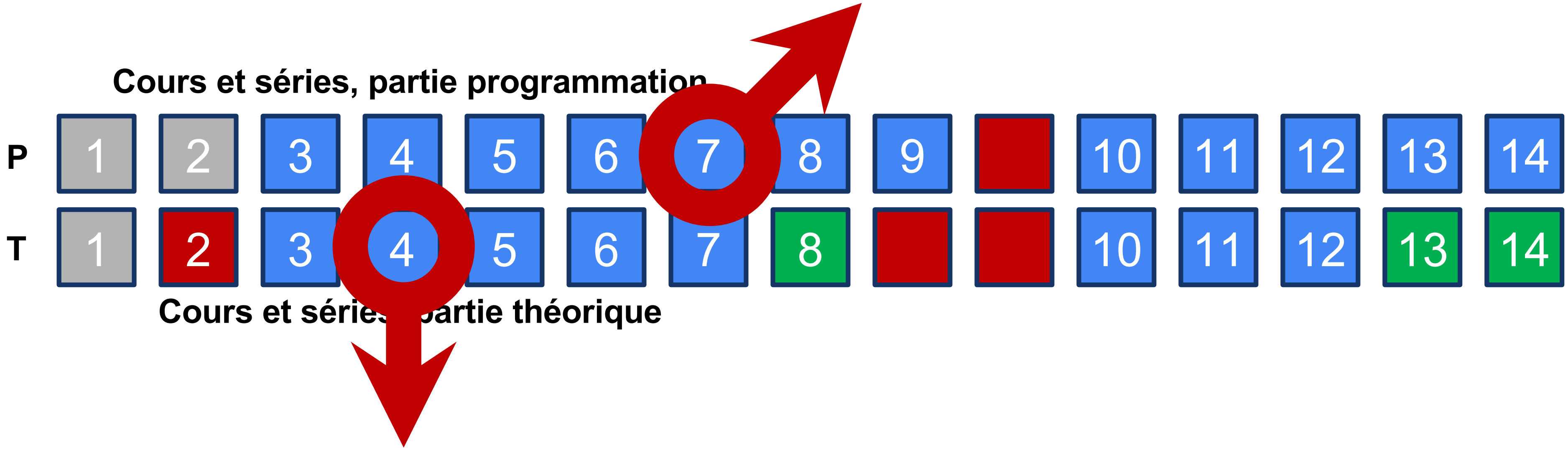
Calcul

Information

Communication

Annonces

31.03 ICC-P Cours par zoom (aussi diffusé en salle)
Séance d’exercices normale



14.03 ICC-T Changement de salle exceptionnel : CM14

Aujourd’hui

- Sous-algorithmes
- Complexité temporelle
- Notation Grand Theta

Sous-algorithmes

~~Sous-algorithmes~~ Sous-recettes

- Préparer le pain → Peut être une sous-recette (faire du pain maison)
- Couper les légumes → Une autre sous-recette

  (utilisable pour une salade aussi)
- Assembler le sandwich → Une tâche qui utilise les résultats des

  sous-recettes précédentes

EPFL

Sous-algorithmes

Algorithme principal

entrée : …
sortie : …

…
$x \leftarrow$ **algo1(5)**
$x \leftarrow$ **10**
$x \leftarrow$ **algo2(5)**
$x \leftarrow x$ **+ algo1(5)**
…

algo1

entrée : nombre entier n
sortie : nombre entier

…

algo2

entrée : nombre entier n
sortie : nombre entier

…

Illustration du principe avec le tri d'une liste

- Comment trier une liste de nombres ?

- 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.

Illustration du principe avec le tri d’une liste

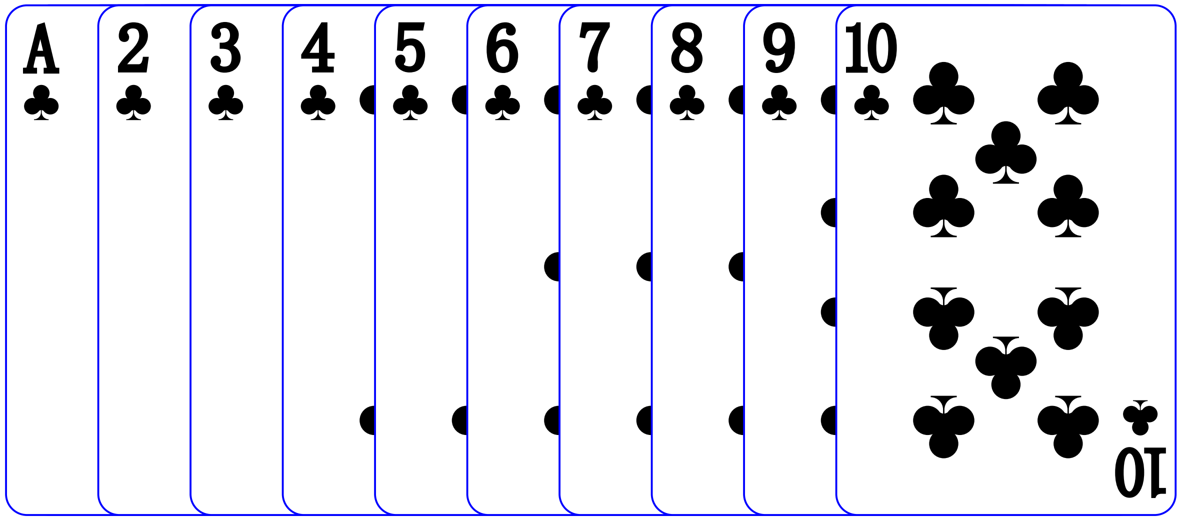


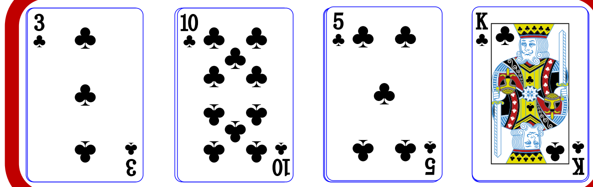
Illustration du principe avec le tri d'une liste

Tri d'une liste – 1er essai

entrée : liste ***L*** de taille ***n***
sortie : liste ***L*** triée dans l'ordre croissant

Pour i allant de 2 à **n** :
    Si L(i) < L(i-1), alors
        L ← **permuter(**L, i, i-1)
Sortir : L

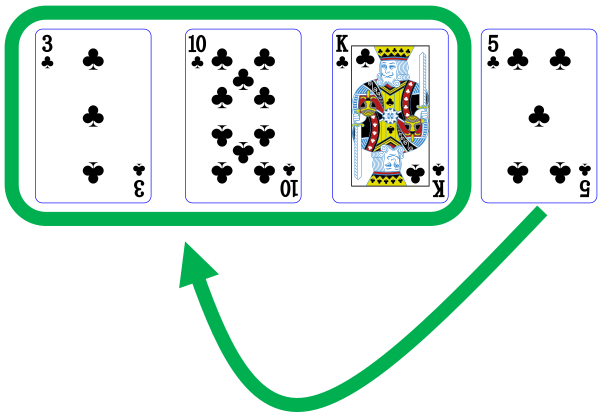
pas triée



Tri par insertion : algorithme principal

Tri par insertion
entrée : liste ***L*** de taille ***n*** ***sortie*** : liste ***L*** triée dans l'ordre croissant

Pour i allant de 2 à n :
    Si L(i) < L(i-1), alors
        L ← insérer(L, i)
Sortir : L



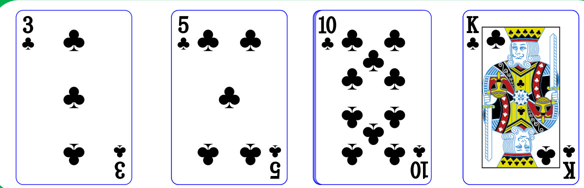
Tri par insertion : sous-algorithme 1

inserer

entrée : liste ***L***, indice ***i***
sortie : liste ***L*** avec l'élement **L**(**i**) bien placé

Tant que i > 1 et L(i) < L(i-1) :
    L ← permuter(L, i, i-1)
    i ← i-1
Sortir : L

triée



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

Tri par insertion : sous-algorithme 2

permuter – 1er essai

entrée : liste ***L***, indices ***j*** et ***k***
sortie : liste ***L*** avec les éléments L(j) et L(k) permutés

L(j) ← L(k)
L(k) ← L(j)
Sortir : L

permuter

entrée : liste ***L***, indices ***j*** et ***k***
sortie : liste ***L*** avec les éléments L(j) et L(k) permutés

temp ← L(j)
L(j) ← L(k)
L(k) ← temp
Sortir : L

Tri par insertion : algorithme entier

Tri par insertion

entrée : liste *L* de taille *n*
sortie : liste *L* triée dans l'ordre croissant

Pour i allant de 2 à n :
    Si L(i) < L(i-1), alors
        L ← **insérer**(L, i)
Sortir : L

insérer

entrée : liste *L*, indice *i*
sortie : liste *L* avec l'élement L(i) bien placé

Tant que i > 1 **et** L(i) < L(i-1) :
    L ← **permuter**(L, i, i-1)
    i ← i-1
Sortir : L

permuter

entrée : liste *L*, indices *j* et *k*
sortie : liste *L* avec les éléments L(j) et L(k) permutés

temp ← L(j)
L(j) ← L(k)
L(k) ← temp
Sortir : L

Aujourd'hui

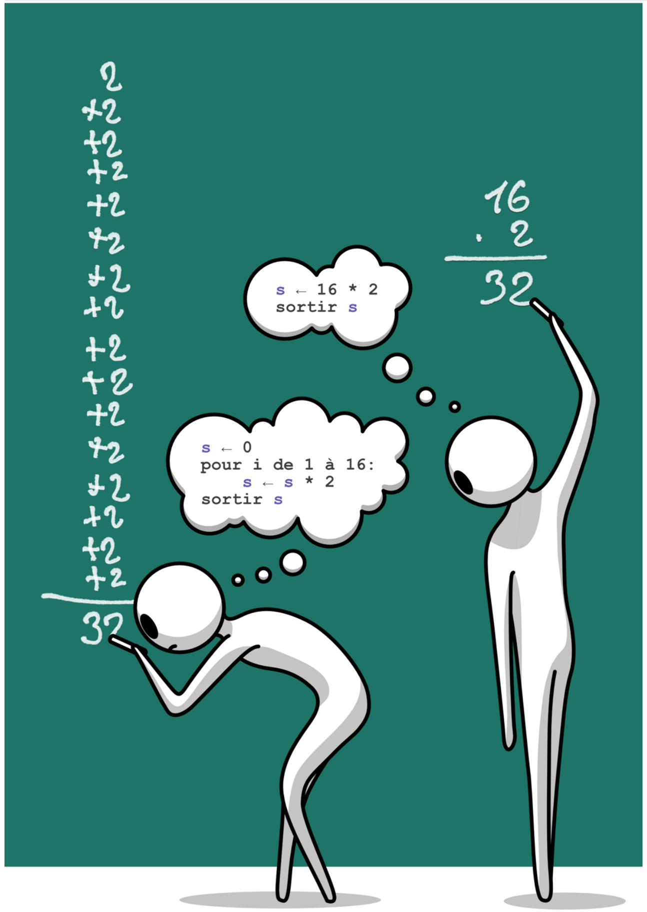
- Sous-algorithmes 

- Complexité temporelle

- Notation Grand Theta

EPFL

Algorithmes : Complexité temporelle



- 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
- **pire des cas :** le temps d'exécution peut en effet dépendre des données d'entrée.

Complexité temporelle : Exemples

Algorithme 1

Tant que 1 > 0 :
    Afficher “bonjour”

Algorithme 2

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

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

EPFL

ICC-T 01 : Tous différents ?

- **Problème à résoudre :**
  - Parmi une liste de n objets, identifier si ceux-ci sont tous différents les uns des autres.

i \ k	1	2	3
1	1,1	1,2	1,3
2	2,1	2,2	2,3
3	3,1	3,2	3,3

Tous différents

entrée : liste ***L*** de ***n*** objets
sortie : valeur binaire oui/non

```
s ← oui
Pour i allant de 1 à n-1 :
    Pour k allant de i+1 à n :
        Si L(i) = L(k), alors :
            Sortir non
Sortir : s
```

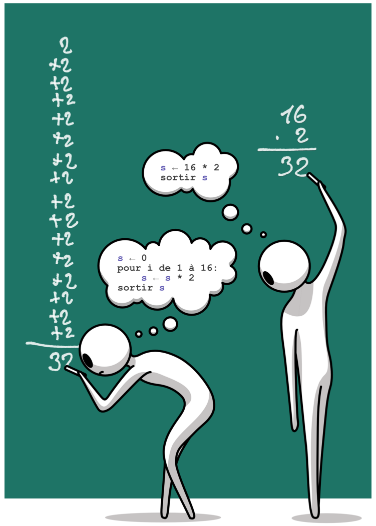
i = 1:	k = 2, 3, 4, …, n	n-1 comp.
i = 2:	k = 3, 4, …, n	n-2 comp.
i = 3:	k = 4, …, n	n-3 comp.
⋮	⋮	⋮
i = n-2:	k = n-1, n	2 comp.
i = n-1:	k = n	1 comp.

$1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{n(n-1)}{2}$ comp.

Aujourd'hui

- Sous-algorithmes

- Complexité temporelle
- **Notation Grand Theta**



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 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 « Tous différents » qui effectue $\frac{n(n-1)}{2}$ opérations lors de son exécution, dans le pire des cas, alors on peut répondre à la question ci-dessus.

EPFL

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.

> 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 $0 < C1 < C2 < \infty$ et $N \geq 1$ tels que
>
> $$C1\ g(n) \leq f(n) \leq C2\ g(n) \quad \text{pour tout } n \geq N$$

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 $\Theta(n^2)$

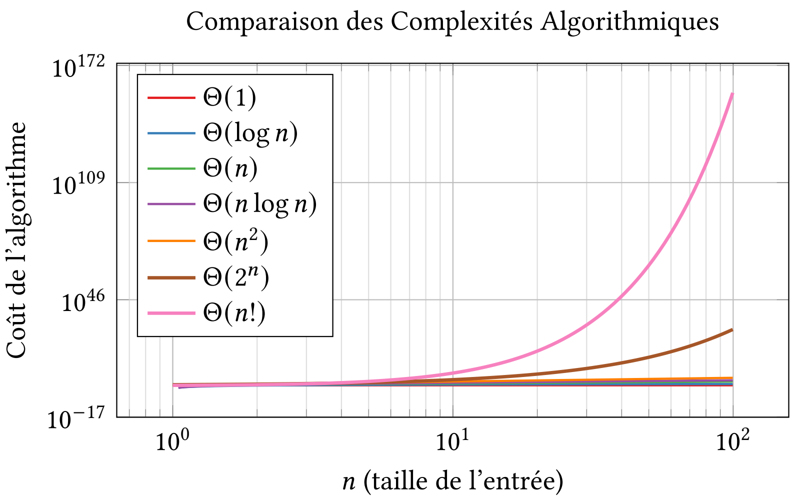
EPFL

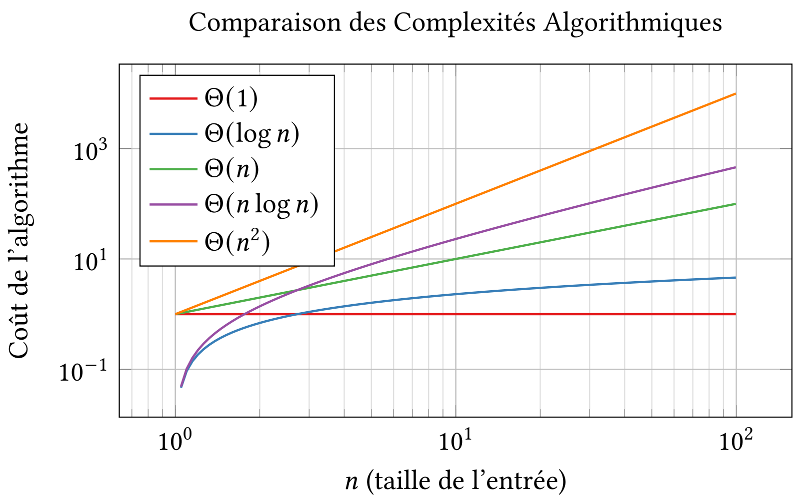
Notation $\Theta(\cdot)$: application

- 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 $\boldsymbol{\Theta}(n)$ et il prend 1 minute avec une entrée de taille n=1'000, alors son temps d'exécution avec n=10'000 en entrée vaudra (approximativement) 10 minutes.
- Si sa complexité temporelle est un $\boldsymbol{\Theta(n^2)}$, alors alors son temps d'exécution avec n=10'000 en entrée vaudra (approximativement) 10 × 10 = 100 minutes = 1 h 40 min.

EPFL

Notation $\Theta(\cdot)$: Ordres de grandeur





Impraticables : $\boldsymbol{\Theta(2^n), \Theta(n!)}$

Plus lents, mais souvent acceptés : $\boldsymbol{\Theta(n^2)}$ … $\boldsymbol{\Theta(n^k), \Theta(n.\log(n))}$

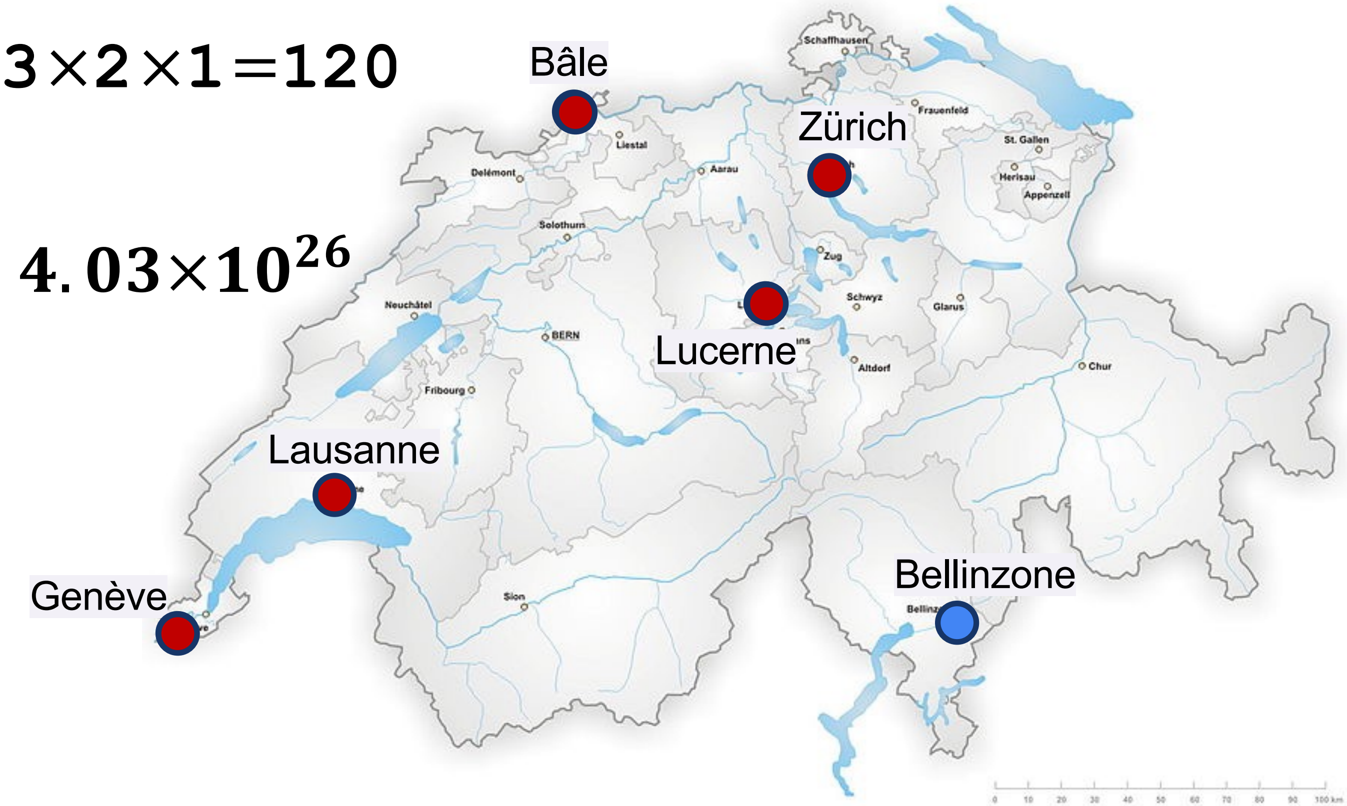
Rapides: $\boldsymbol{\Theta(1), \Theta(\log n), \Theta(n)}$

ICC-T 01 : problème du voyageur de commerce

$$5\times4\times3\times2\times1=120$$

$$n!$$

$$26! \approx 4.03\times10^{26}$$



Factoriale	Résultat
1!	1
2!	2
3!	6
4!	24
5!	120
6!	720
7!	5040
8!	40320
9!	362880
10!	3628800
11!	39916800
12!	479001600
13!	6227020800
14!	87178297200
15!	1307674368000
16!	20922789888000
17!	355687428096000
18!	6402373705728000
19!	121645100408832000
20!	2432902008176640000
21!	51090942171709440000
22!	1124000727777607680000
23!	25852016738884976640000
24!	620448401733239439360000
25!	15511210043330985984000000

EPFL

Notation $\Theta(\cdot)$: Illustration

i \ k	1	2	3
1	1,1	1,2	1,3
2	2,1	2,2	2,3
3	3,1	3,2	3,3

- Calcul du nombre de paires d'éléments dans l'ensemble $\{ 1, 2, \ldots, n \}$

- Pour calculer ce nombre, il existe plusieurs façons de faire :
  - Utilisation de deux boucles imbriquées
    - Complexité $\boldsymbol{\Theta(n^2)}$
  - Utilisation d'une seule boucle
    - Complexité $\boldsymbol{\Theta(n)}$
  - Utilisation de la formule mathématique
    - Complexité $\boldsymbol{\Theta(1)}$

$s \leftarrow 0$
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

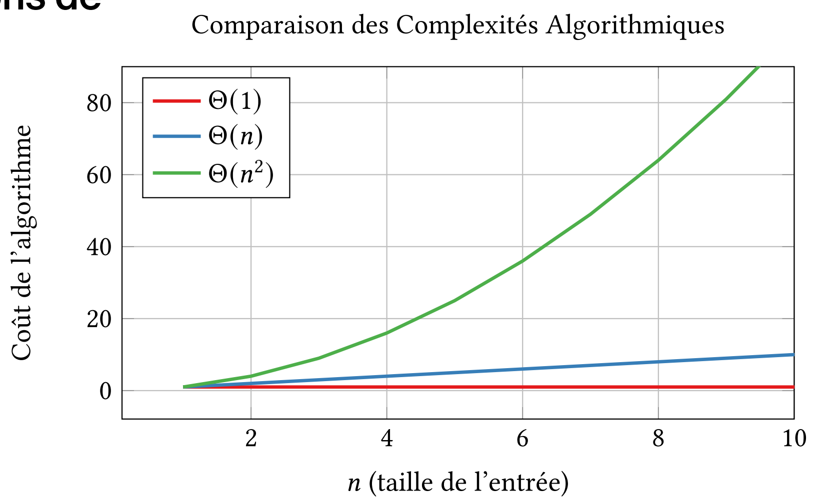
$s \leftarrow \frac{n(n-1)}{2}$
Sortir : s

EPFL

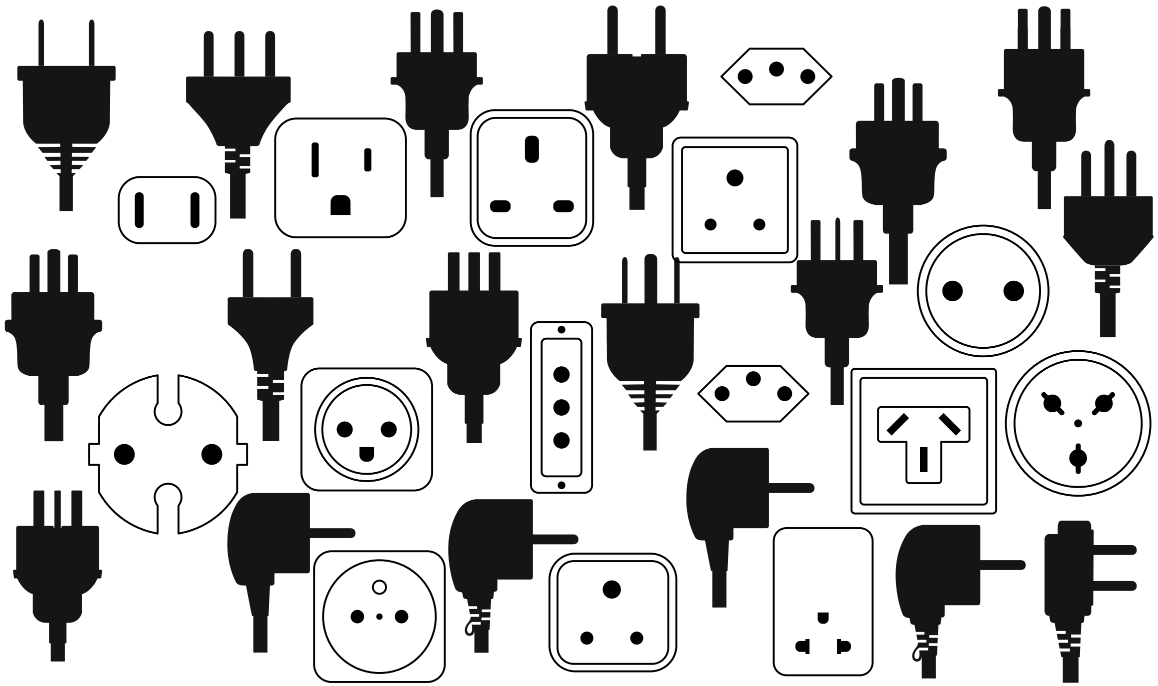
Notation $\Theta(\cdot)$: Illustration

- Calcul du nombre de paires d'éléments dans l'ensemble
  $$\{ 1, 2, \ldots, n \}$$

- 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 $\Theta(1)$



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.
  - Si $L = (-14, -3, -1, +3, +7, +10)$,
    alors la réponse est oui.

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

EPFL

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 $L(i) + L(j) = 0$?

Tous différents

entrée : liste ***L*** de ***n*** objets
sortie : valeur binaire oui/non

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

→

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

Première méthode de résolution

Deux font la paire – 1er essai

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

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

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

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

- donc la complexité temporelle de l'algorithme est $\boldsymbol{\Theta(n^2)}$.
- Question : Peut-on faire mieux ?

- Remarque :
  - L'algorithme précédent n'exploite pas **l'ordre** de la liste L.

EPFL

Deuxième 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 ***n*** 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

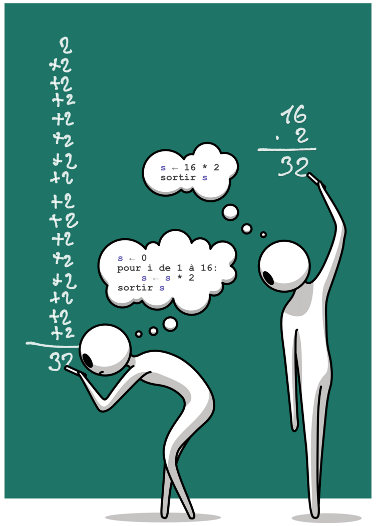
i=2 **j=4**

$$L = (-14, -3, -1, +3, +7, +10)$$

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

Aujourd'hui

- Sous-algorithmes



- Complexité temporelle

- Notation Grand Theta

Θ

EPFL

Résumé Cours 2 – ICC-T

- Les sous-algorithmes permettent de décomposer un problème en sous-problèmes plus simples, favorisant ainsi l'abstraction, la réutilisation du code, une meilleure lisibilité et une maintenance facilitée des algorithmes.
  - Tri par insertion
- La notation Grand Theta permet de caractériser précisément l’ordre de complexité d’un algorithme.
- 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.

rafael.pires@epfl.ch

Merci