Faculté des Sciences
كلية العلوم

Département d'Informatique
Filière MIP - Semestre 2
2023-2024

Module : Informatique II (Algorithmique II / Python)

Chapitre 5

La complexité

Pr. Khadija Louzaoui

Introduction

- Quand on veut résoudre un problème, la question qui se pose c'est le choix du **meilleur algorithme** parmi les algorithmes qui permettent de résoudre ce problème.
- Certains algorithmes sont complexes et le traitement peut nécessiter beaucoup de **temps** et de **ressources de machine**, c'est ce qu'on appelle le "**coût** " (**efficacité** ou **complexité**) de l'algorithme.
- L'analyse de la **complexité** consiste à mesurer ces deux grandeurs (**temps** et **espace mémoire**) pour choisir l'algorithme le mieux adapté pour résoudre un problème (le plus rapide, le moins gourment en place mémoire).

Introduction

- Dans ce cours on ne s'intéresse qu'à la **complexité temporelle** c.à d. qu'au **temps de calcul** (par opposition à **la complexité spatiale**).
- Le temps d'exécution est difficile à prévoir, il peut être affecté par plusieurs facteurs:
  - Le problème à résoudre
  - La taille des données
  - Les structures de données utilisées
  - L'algorithme de résolution
  - L'expertise et l'habilité du programmeur
  - La rapidité de la machine
  - Le langage de programmation
  - Le compilateur

Introduction

- Pour pallier à ces problèmes, une notion de complexité plus simple, mais efficace, a été définie pour un modèle de machine . Elle consiste à compter **les instructions de base** exécutées par l'algorithme. Elle est exprimée en fonction de **la taille** du problème à résoudre.
- Une instruction de base (ou élémentaire) est soit: une affectation, un test, une addition, une multiplication, modulo, ou partie entière,...

Introduction

- On cherche à mesurer la complexité d'un algorithme en fonction de **la taille des données** que l'algorithme doit traiter.
- Exemples :
  - Recherche d'une valeur dans un tableau
    → taille (= nombre d'éléments) du tableau
  - Produit de deux matrices
    → taille (=dimension) des matrices
  - Recherche d'un mot dans un texte
    → taille(= longueur du mot et celle du texte)
  - Calcul d'un terme d'une suite
    → taille(= indice du terme de la suite)

Définition

- **La complexité (temporelle) d'un algorithme** désigne le nombre d'opérations fondamentales (affectations, comparaisons, opérations arithmétiques) qu'il effectue sur un jeu de données.
- La **complexité** s'exprime **en fonction** de la taille **n** des données.
- On note généralement:
  - **n** la taille de données
  - **T(n)** le **temps** (ou le **cout**) de l'algorithme.

  → **T(n)** est une fonction de **IN** dans **IR**$^{+*}$

Type de la complexité

- Lorsque, pour **une valeur donnée du paramètre de complexité**, le temps d'exécution varie selon les données d'entrée, on peut distinguer trois mesures de complexité :
  - **La complexité dans le meilleur cas** : temps d'exécution minimum, dans le cas le plus **favorable** (en pratique, cette complexité n'est pas très utile).
  - **La complexité dans le pire cas** : temps d'exécution maximum, dans le cas le plus **défavorable**.
  - **La complexité dans le cas moyenne :** temps d'exécution dans un cas médian, ou **moyenne** des temps d'exécution.
- Dans la suite nous nous **intéressons particulièrement au pire cas** , car on veut borner le temps d'exécution.

Type de la complexité

Exemple :

la recherche d'une valeur dans un tableau dépend de la position de cette valeur dans le tableau.

```
i ←1
Tantque (i ≤n) et (A[i] <> x) faire
        i ← i+1
FinTantque
Si i >n alors
        retourner(faux)
Sinon
        retourner(vrai)
FinSi
```

Type de la complexité

Exemple :

10	-8	57	-12	23	0	84	-9	15	64

- **Le meilleur cas :** est que l'on trouve l'élément à la première comparaison.

$T_{min}(n) = \min \{T(d) ; d \text{ une donnée de taille } n\}$

- **Le pire cas :** est qu'on parcours tous les éléments de la liste et que l'élément recherché ne s'y trouve pas.

$T_{max}(n) = \max \{T(d) ; d \text{ une donnée de taille } n\}$

- **Le moyen cas:** est que l'élément recherché se trouve à la 2ème , 3ème position par exemple.

$T_{moy}(n) = \Sigma\, p(d).T(d)$; d de taille n, $p(d)$: probabilité d'avoir la donnée d

$$T_{min}(n) \leq T_{moy}(n) \leq T_{max}(n)$$

Type de la complexité

Pour mesurer la complexité d'un algorithme, il ne s'agit pas de faire un décompte exact du nombre d'opérations T(n), mais plutôt de donner un ordre de grandeur de ce nombre pour **n assez grand**.

- **Première approximation :** on ne considère souvent que la **complexité au pire**
- **Deuxième approximation :** on ne calcule que la **forme générale** de la complexité
- **Troisième approximation :** on ne regarde que le **comportement asymptotique** de la complexité.

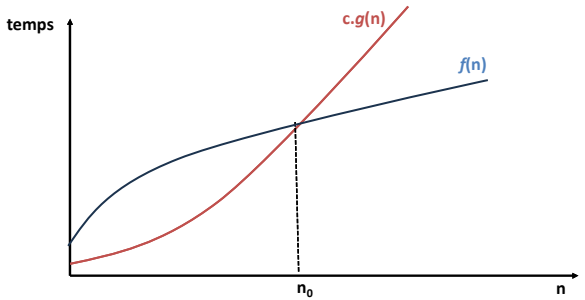
Notation de Landau

- La notation de **Landau** est celle qui est la plus communément utilisée pour expliquer formellement les performances d'un algorithme. Cette notation exprime la limite supérieure d'une fonction dans un facteur constant.
  - "**grand O**":
    $f(n) = O(g(n))$ ssi
    $\exists\, c > 0, \exists\, n_0 \geq 0 \,/\, \forall\, n > n_0 : f(n) \leq c.g(n)$
  - "**grand oméga**":
    $f(n) = \Omega(g(n))$ ssi
    $\exists\, c > 0, \exists\, n_0 \geq 0 \,/\, \forall\, n > n_0 : f(n) \geq c.g(n)$
  - "**grand théta**":
    $f(n) = \Theta(g(n))$ ssi
    $\exists\, c_1 > 0, \exists\, c_2 > 0, \exists\, n_0 \geq 0 \,/\forall\, n > n_0 : c_1.g(n) \leq f(n) \leq c_2.g(n)$

- **Remarque :** Les fonction utilisées dans ce chapitre sont **des suites à valeurs strictement positives** et $(n_0, c) \in \mathbb{N}^* \times \mathbb{R}^{+*}$.

Notation de Landau

f et g étant des suites, $f = O(g)$
s'il existe des constantes c>0 et n_0 telles que $f(n) \leq c.g(n)$ pour tout $n \geq n_0$

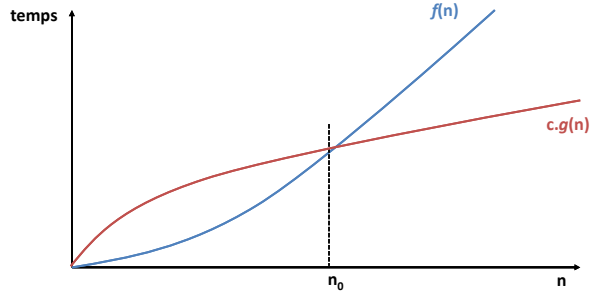


$f = O(g)$ signifie que f est dominée par g.

On dit que f est asymptotiquement majorée ou (dominée) par g.
Signification : pour toutes les grandes entrées pour ($n \geq n_0$), on est assuré que l'algorithme ne prend pas plus que $c.g(n)$ unité de temps : borne supérieure.

Notation de Landau

f et g étant des suites, **$f = \Omega(g)$**
s'il existe des constantes c>0 et n_0 telles que $f(n) \geq c.g(n)$ pour tout $n \geq n_0$

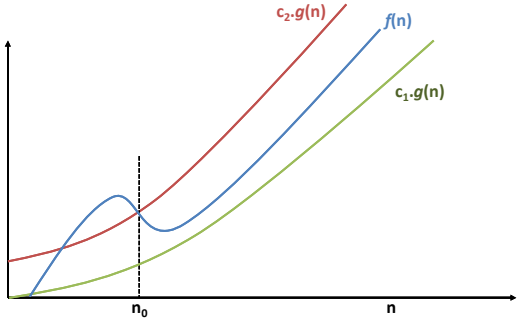


$f = \Omega(g)$ signifie que f domine g.

On dit que f est asymptotiquement minorée ou (domine) par g.
Signification : pour toutes les grandes entrées pour ($n \geq n_0$), on est assuré que l'algorithme prend plus que $c.g(n)$ unité de temps : borne inférieure.

Notation de Landau

f et g étant des suites, **$f = \Theta(g)$**
s'il existe des constantes c_1>0 , c_2>0 et n_0 telles que
$c_1 .g(n) \leq f(n) \leq c_2 .g(n)$ pour tout $n \geq n_0$



$f = \Theta(g)$ signifie que f domine g et f est dominée par g.

Notation de Landau

❑ Notations supplémentaires:

- "**petit o**": $f(n) = o(g(n))$ ssi
  $\forall \varepsilon > 0, \exists n_0 > 0 \ / \ \forall n \geq n_0 : f(n) \leq \varepsilon.g(n)$
  En pratique: **$f(n) = o(g(n))$ ssi**
  $$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$
  On dit f est négligable devant g.
  - " **équivalence à** ": $f(n) \sim g(n)$ ssi
    $\forall \varepsilon > 0, \exists n_0 > 0 \ / \ \forall n \geq n_0 : |f(n) - g(n)| \leq \varepsilon.g(n)$
    En pratique : **$f(n) \sim g(n)$ ssi**
    $$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 1$$
    On dit f est approximativement égale à g.
- "**petit omega** ": $f(n) = \omega(g(n))$ ssi
  $\forall k > 0, \exists n_0 > 0 \ / \ \forall n \geq n_0 : f(n) \geq k.g(n)$
  En pratique : **$f(n) = \omega(g(n))$** ssi
  $$\lim_{n \to +\infty} \frac{g(n)}{f(n)} = 0$$

- grand-**O**, **Θ** et **Ω** sont les plus utilisées en **informatique**
- le petit-**o** est courant en **mathématiques** mais plus rare en informatique
- le ω est rarement utilisés.

Notation de Landau

On a par définition:

$$O(g) = \{ f : \mathbb{N} \longrightarrow \mathbb{R}^{+*} / \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : f(n) \leq c.g(n) \}$$

$$\Omega(g) = \{ f : \mathbb{N} \longrightarrow \mathbb{R}^{+*} / \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : f(n) \geq c.g(n) \}$$

$$\Theta(g) = O(g) \cap \Omega(g)$$

La difficulté, dans la familiarisation avec ces concepts, provient de la convention de notation (de Landau) qui veut que l'on écrive par abus de notation:

f = O(g), ou encore **f(n) = O(g(n))** au lieu de:
f ∈ O(g), ou encore **f est O(g)**

De maniere analogue, on ecrit **O(f) = O(g)** lorsque **O(f)⊂ O(g)**
(il en est de même pour les notations Θ ou Ω)

Notation de Landau

Remarques pratiques:

- Le cas le plus défavorable est souvent utilisé pour analyser un algorithme.
- La notation **O** donne une borne supérieure de la complexité pour toutes les données de même taille (suffisamment grande). Elle est utilisée pour évaluer un algorithme dans le cas le plus défavorable.
- $f(n) \leq c.g(n)$ signifie que le nombre d'opérations ne peut dépasser $c.g(n)$ itérations, pour n'importe quelle donnée de longueur n.
- Pour évaluer la complexité d'un algorithme, on cherche un majorant du nombre d'opérations les plus dominantes.
- Dans les notations asymptotiques, on ignore les constantes.

Notation de Landau

Exemple:
Considérons le programme de détermination du nombre d'occurrences dans un tableau de type list. On veut déterminer la complexité de la fonction nbre_occurrences.

```
def nbre_occurrences (x:object , t: list ) -> int :
    c = 0
    for i in range (len(t)):
        if t[i] == x:
            c = c + 1
    return c
```


Notation de Landau

Solution 1:

1. L'affectation `c = 0` compte pour une opération élémentaire (**1 opération**)
2. La boucle bornée `for i in range(len(t))` se déroule **len(t) fois**, où **len(t)** est la taille du tableau.
3. L'instruction de contrôle de test d'égalité `t[i] == x` se déroulent en temps constant (**1 opération**).
4. Le calcul de l'expression `c+1` ainsi que l'affectation `c = c+1` se déroulent en temps constant (**2 opérations**).
5. `return c` est une opération élémentaire (**1 opération**).

Conclusion :

Si **n = len(t)** (*taille de l'entrée*), le traitement se réalise en un maximum de $3n+2$ opérations élémentaires. Or, $3n+2 \leq 4n$, donc que **la complexité temporelle dans le pire des cas** de la fonction nbre_occurrences est $O(n)$.
On dit aussi que la complexité temporelle est **linéaire**.

Notation de Landau

Solution 2:

Grâce à la notation de Landau :

1. La longueur du tableau **n = len(t)** est une mesure de la taille du problème considéré.
2. L'affectation `c = 0` se déroule en **O(1)** (on dit qu'elle est en O(1), car majorée par une constante).
3. La boucle bornée `for i in range(len(t))` se déroule **n fois**.
4. Les instructions `if t[i] == x` et `c = c + 1` se déroulent en **O(1)**. Donc dans le pire des cas, la boucle for et son bloc de code se déroulent en **O(n)**.
5. `return c` se déroule en **O(1)**.

Conclusion :
par somme, la complexité temporelle dans le pire des cas de la fonction nbre_occurrences est en **O(n)**.

Notation de Landau

Exemple:

```
def nbre_occurrences (x:object , t: list ) -> int :
    c = 0
    for i in range (len(t)):
        if t[i] == x:
            c = c + 1
    return c
```

- Dans le **meilleur des cas**, t[0] == x : la boucle s'arrête après 1 passage. On dit que la complexité temporelle dans le meilleur des cas est en O(1).
- Dans le **pire des cas**, x n'est pas dans t : la boucle s'arrête après n passages. On dit que la complexité temporelle dans le pire des cas est en O(n).

Par défaut, on déterminera la complexité temporelle dans le pire des cas.

Notation de Landau

Propriétés:

En utilisant la notation de Landau (pour les fonctions de IN dans IR^{+*}), on a :

1. $O(c) = O(1)$
2. $c\,O(f) = O(c\,f) = O(f)$ ($c>0$)
3. $f = O(f)$
4. $f = O(g)$ et $g = O(h) \Rightarrow f = O(h)$
5. $O(f) + O(g) = O(f + g) = O(\max(f, g))$
6. $f + O(g) = O(f + g)$
7. $h = f + O(g) \Leftrightarrow h - f = O(g)$.
8. $O(f) + O(f) = O(f)$
9. $O(f)\,O(g) = O(fg)$
10. $f\,O(g) = O(fg)$

Notation de Landau

Principales relations entre: O, o, Ω, ω, Θ, $\sim$

Généralement on ne calcul pas la complexité exacte, mais son ordre de grandeur. Pour ce faire, nous avons besoin de notations asymptotiques:

1. $f = o(g) \Rightarrow f = O(g)$
2. $f = O(g) \Leftrightarrow g = \Omega(f)$
3. $f = \Omega(g) \Leftrightarrow g = O(f)$
4. $f = \omega(g) \Rightarrow f = \Omega(g)$
5. $f \sim g \quad \Rightarrow f = \Theta(g)$
6. $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = l\ (l > 0) \Rightarrow f = \Theta(g)$
7. $f = \Theta(g) \Leftrightarrow f = O(g)$ et $f = \Omega(g)$
8. $f = \Theta(g) \Rightarrow f = O(g)$
9. $f = \Theta(g) \Rightarrow f = \Omega(g)$

Notation de Landau

Supposant que le temps d'exécution d'un algorithme est décrit par la fonction **$T(n) = 3n^2+10n+10$**, Calculer O(T(n))?

$O(T(n)) = O(3n^2 + 10n + 10)$
$= O(\max(3n^2, 10n, 10))$
$= O(3n^2)$
$= O(n^2)$

Remarque:

Pour n = 10 nous avons :

- Temps d'exécution de $3n^2$: $3(10)^2$ / $3(10)^2+10(10)+10$ = 73,2%
- Temps d'exécution de 10n : 10(10)/3(10)2+10(10)+10 = 24,4%
- Temps d'exécution de 10 : 10 / 3(10)2+10(10)+10 = 2,4%

Le poids de $3n^2$ devient encore plus grand quand n=100, soit 96,7%.

On peut négliger les quantités 10n et 10.

Ceci explique les règles de la notation O.

Classes de complexité

- La complexité asymptotique est le comportement de la complexité d'un algorithme lorsque la taille de son entrée est asymptotiquement grande.

- Soit **n** un entier naturel non nul. On dit qu'un algorithme de complexité temporelle **T(n)** s'exécute dans le pire des cas :

  - en temps **constant** quand **$T(n) = O(1)$**
  - en temps **logarithmique** quand **$T(n) = O(\log n)$**
  - en temps **linéaire** quand **$T(n) = O(n)$**
  - en temps **quasi-linéaire** quand **$T(n) = O(n \log n)$**
  - en temps **quadratique** quand **$T(n) = O(n^2)$**
  - en temps **cubique** quand **$T(n) = O(n^3)$**
  - en temps **polynomial** quand il existe $p \geq 2$ tel que **$T(n) = O(n^p)$**
  - en temps **exponentiel** quand il existe $a > 1$ tel que **$T(n) = O(a^n)$**
  - en temps **factorielle** quand **$T(n) = O(n!) = O(n^n)$**

Classes de complexité

Classe	Notation O	Exemple
Constante	O(1)	pas d'augmentation du temps d'exécution quand le paramètre croit Affectation, comparaison,…
Logarithmique	O(log(n))	augmentation très faible du temps d'exécution quand le paramètre croit. Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales, etc.
Linéaire	O(n)	augmentation linéaire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). Somme des n premier entiers naturels
Quasi-linéaire	O(n log(n))	augmentation un peu supérieure à O(n). Calculer la somme des chiffres des n premier entiers naturels
Quadratique	$O(n^2)$	quand le paramètre double, le temps d'exécution est multiplié par 4. algorithmes avec deux boucles imbriquées.

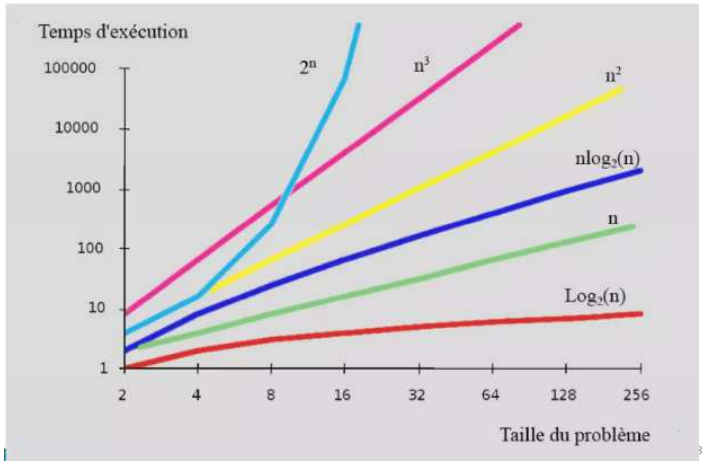
Classes de complexité

Classe	Notation O	Exemple
Polynomiale	$O(n^p)$	quand le paramètre double, le temps d'exécution est multiplié par 2^i. algorithme utilisant i boucles imbriquées.
Exponentielle	$O(a^n)$	quand le paramètre double, le temps d'exécution est élevé à la puissance 2.
factorielle	O(n!)	asymptotiquement équivalente a n^n

- Les algorithmes de complexité polynomiale ne sont utilisables que sur des données réduites ($p \leq 3$), ou pour des traitements ponctuels.
- Les algorithmes exponentiels ou au delà ne sont pas utilisables en pratique.
- On a:

$(O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(e_n) \subset O(n!)$

Classes de complexité



Complexité et temps d'exécution

- Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme.
- On suppose que l'ordinateur utilise peut effectuer 10^6 opérations a la seconde (une opération est de l'ordre de la µs)

n \T(n)	log n	n	n log n	n^2	2^n
10	3 µs	10 µs	30 µs	100 µs	1000µs
100	7 µs	100 µs	700 µs	1/100 s	1014 siècles
1000	10 µs	1000µs	1/100µs	1 s	Astronomique
10000	13 µs	1/100µs	1/7 s	1,7 mn	Astronomique
100000	17 µs	1/10 s	2 s	2,8 h	Astronomique

Il vaut mieux optimiser ses algorithmes qu'attendre des années qu'un processeur surpuissant soit inventé.

Calcul de la complexité: règles pratiques

- Calculer le cout d'un programme revient à calculer le nombre d'**opérations élémentaires** en fonction:
  - de la **taille** des données (par ex. le nombre d'éléments à trier)
  - de **la nature** des données (provoquant par exemple une sortie de boucle prématurée)
- **Configurations caractéristiques :**
  - le meilleur des cas,
  - **le pire des cas**,(on veut borner le temps d'exécution)
  - la configuration en moyenne
- **Notations :**
  - **n** : la taille des données,
  - **T(n)** : le nombre d'opérations élémentaires
- Pour déterminer le cout d'un algorithme, on se fonde en général sur le modèle de complexité suivant:

Calcul de la complexité: règles pratiques

1. **Cas d'une instruction de base:** (écriture, lecture, affectation, comparaison ou évaluation d'une expression arithmétique (+,-,/,*,div,%,^) ayant en général un faible temps d'exécution considéré comme l'unité de mesure du coût ***c*** d'un algorithme.

T(n)=c

O(T)=O(c)=O(1)

Exemple:

1- a←5	c_1	$T(n)=c_1$
2- b←a+5	c_2	$T(n)=c_2$
3- b←(a+5)/3	c_3	$T(n)=c_3$

le coût total est : $T(n)=c_1=1$ Donc la complexité est $O(T)=O(1)=O(1)$

le coût total est : $T(n)=c_2=2$ Donc la complexité est $O(T)=O(2)=O(1)$

le coût total est : $T(n)=c_3=3$ Donc la complexité est $O(T)=O(3)=O(1)$

Calcul de la complexité: règles pratiques

2. **Cas d'une suite d'instructions simples**: Le coût des instructions en séquence est la somme des coût des instructions.

Temps d'exécution :

Traitement 1 $T_1(n)$
Traitement 2 $T_2(n)$
$\Big\}$ $T(n)=T_1(n)+T_2(n)$

Notation de landau :

Traitement 1 $O(T_1(n))$
Traitement 2 $O(T_2(n))$
$\Big\}$ $O(T(n))=O(T_1(n))+O(T_2(n))$

$$O(T) = O(T_1)+O(T_2) = O(T_1 + T_2) = O(\max(T_1,T_2))$$

Calcul de la complexité: règles pratiques

Exemple:

Temps d'exécution :

```
a←5              1
b← (a*3)/2       3      } T(n)= 4
```

Le coût total est : $T(n)= c_1 + c_2$
$=1+3$
$=4$
$=O(4)$
$=O(1)$

Notation de landau :

```
a←5              O(1)
b← (a*3)/2       O(1)   } O(1)
```

la complexité est : $O(T) = O(1)+ O(1)$
$= O(\max(O(1),O(1)))$
$=O(1)$

Calcul de la complexité: règles pratiques

3. **Cas d'un traitement conditionnel:** Le coût d'un test est égal au maximum des coûts des instructions, plus le temps d'évaluation de la condition.

Temps d'exécution :

```
Si (condition) Alors     Tc (n)
    Traitement 1         T1 (n)   } T(n)=Tc(n)+max(T1 (n),T2 (n))
Sinon
    Traitement 2         T2(n)
FinSi
```

Notation de landau:

```
Si (condition) Alors     O(Tc (n))
    Traitement 1         O(T1 (n))   } O(T(n))=O(max(Tc(n),T1 (n),T2 (n)))
Sinon
    Traitement 2         O(T2(n))
FinSi
```


Calcul de la complexité: règles pratiques

Exemple:

Temps d'exécution :

```
Si (a>b) alors     1
    max←a          1     } T(n)= 1 + max(1 , 1 )
Sinon
    max←b          1
FinSi
```

Le coût total : $T(n)= 1+\max(1,1)=1+1=O(1)$

Notation de landau:

```
Si (a>b) alors     O(1)
    max←a          O(1)     } O(max(1,1,1))
Sinon
    max←b          O(1)
FinSi
```

Donc la complexité est : $O(T) = O(\max(O(1), O(1) , O(1))=O(1)$

Calcul de la complexité: règles pratiques

4. **Cas d'un traitement itératif : boucle Pour**

Le coût d'une boucle **Pour** est égal au nombre de répétitions multiplié par le coût du bloc d'instructions du traitement.
Quand le coût du traitement dépend de la valeur de **i**, le cout total de la boucle est la somme des coût du traitement pour chaque valeur de **i**.

Temps d'exécution :

```
Pour i de in à fin faire
    Traitement     T1(n)   } (fin-in+1) fois
FinPour
```

$T(n)= T_1(n)\times(fin-in+1)$

Notation landau

```
Pour i de in à fin faire
    Traitement     O(T1(n))   } (fin-in+1) fois
FinPour
```

$O(T(n))= O((fin-in+1) \times O(T_1(n)))$

Calcul de la complexité: règles pratiques

Exemple: Calcul la factorielle d'un entier

Temps d'exécution :

```
fact ← 2                          1
Pour i de 3 à n faire             (n-2) itération
      fact ← fact*i               2          } (n-2)*2
FinPour                                      } T(n)=1+(n-2)*2+1
Retourner fact                    1
```

On a : T(n) =2n-2 et 2n-3≤ 2n+2n alors T(n) ≤ 4n
Donc la complexité temporelle dans le pire des cas est O(*n*).

Notation landau

```
fact ← 2                          O(1)
Pour i de 3 à n faire             O(n)   } O(n)
      fact ← fact*i               O(1)   } O(n)=O(1)+O(n)+O(1)
FinPour
Retourner fact                    O(1)
```

La complexité temporelle dans le pire des cas est O(*n*).

Calcul de la complexité: règles pratiques

4. Cas d'un traitement itératif : boucle Tantque

Le coût d'une boucle **Tantque** est plus complexe à traiter puisque le nombre de répétitions n'est en général pas connu a priori.
On peut majorer le coût de l'exécution de la boucle par le nombre de répétitions effectuées

Temps d'exécution :

```
Tantque condition faire  Tc(n)   } m fois
      Traitement         T1(n)   }
FinTantque
```

$$T(n) = 1 + \sum_{k=1}^{m} (T_1(n)+T_c(n))$$

$$T(n) = 1 + m\times(T_1(n)+T_c(n))$$

Notation landau

```
Tantque condition faire  O(Tc(n))   } m fois
      Traitement         O(T1(n))   }
FinTantque
```

$$O(T(n)) = O(m\times(O(T_1(n))+O(T_c(n))))$$

Calcul de la complexité: règles pratiques

Exemple: Recherche séquentielle

Temps d'exécution :

```
i← 1                                  1   } 2
Trouve ← faux                         1   }
Tant que ((i≤n) et (non trouve)) faire 3  } 3(n+1)
  Si (T[i] = x) Alors                 1
     Trouve ← vrai                    1   } 4 n
  FinSi                                  }
  i ← i + 1                           2   }
FinTantQue                                } T(n)=7n+6
Retourner trouve                      1   } 1
```

On a : T(n) =7n+6 et 7n+6≤ 7n+6n alors T(n) ≤ 13n
Donc la complexité temporelle dans le pire des cas est O(*n*).

Calcul de la complexité: règles pratiques

Exemple: Recherche séquentielle

Notation landau

```
i← 1                                  O(1)   } O(1)
Trouve ← faux                         O(1)   }
Tant que ((i ≤ n) et (non trouve)) faire O(1)
  Si (T[i] = x) Alors                 O(1)
     Trouve ← vrai                    O(1)   } O(1)
  FinSi                                  }
  i ← i + 1                           O(1)   } n × max(O(1),O(1))
FinTantQue                                   } O(n)
Retourner trouve                      O(1)   } O(1)
```

Donc : O(T) = O(1)+ n ×max(O(1) , O(1)) +O(1)= n × O(1)=O(n)

Calcul de la complexité asymptotique d'un algorithme

❑ Pour calculer la complexité d'un algorithme :

1. on calcule la complexité de chaque "partie" de l'algorithme.
2. on combine ces complexités conformément aux règles qu'on vient de voir.
3. on simplifie le résultat grâce aux règles de simplifications qu'on a vues.
   - élimination des constantes.
   - conservation du (des) termes dominants.

Calcul de la complexité: règles pratiques

Exercice :
Donner la complexité temporelle dans le pire des cas du code ci-dessous :

Calcul de la somme 1+2+…+n

```
s ← 0
Pour i de 1 à n faire
    s ← s+i
FinPour
```

T(n)=O(?)

Calcul de la complexité: règles pratiques

Solution : **Calcul de la somme 1+2+…+n**

Temps d'exécution :

```
s ← 0                          1
Pour i de 1 à n faire          2 } 2n   } T(n)=2n +1
    s ← s+i
FinPour
```

on a **T(n)=2n +1** et 2n+1 ≤ 2n+n alors **T(n) ≤ 3n** pour n≥1
Donc la complexité temporelle dans le pire des cas est **O(*n*)**.

Notation landau

```
s ← 0                          O(1)
Pour i de 1 à n faire          O(1) } (∑O(1))n  } O(1)+O(n)=O(n)
    s ← s+i                        O(n)
FinPour
```

La complexité temporelle dans le pire des cas est **O(*n*)**.

Calcul de la complexité: règles pratiques

Exercice :
Donner la complexité temporelle dans le pire des cas du code ci-dessous :

```
def somme1(n):                 1
    s=0                        n
    for i in range(1,n+1):      1+1
        s=s+i                  1
    return (s)
```

T(n)=1+n*2+1=2n+1
O(T(n))=O(n)

```
def somme2(n):                 1
    s=0                        n
    for i in range(1,n+1):      n
        for j in range(1,n+1):  1   1+1+1
            s=s+i*j
    return (s)
```

T(n)=1+n*n*3+1=3n²+1
O(T(n))=O(n²)

```
def somme3(n):                 1
    s=0                        n
    for i in range(1,n+1):      i
        for j in range(1,i+1):  1   1+1+1
            s=s+i*j
    return (s)                 1
```

$$T(n) = 1 + \sum_{i=1}^{n} 3 + 1$$

T(n)=1+(n*(n+1)/2)*3+1
=(3/2)n²+(3/2)n+2

O(T(n))=O(n²)

Complexité des algorithmes récursifs

La complexité d'un algorithme récursif peut être calculée par la résolution d'une équation de récurrence

```
Fonction factoriel (n:entier):entier
Début
      Si (n=0) alors
          retourner 1
      Sinon
          retourner (n*factoriel(n-1))
      FinSi
Fin
```

$$\begin{cases} T(0) = 2 \\ T(n) = 4 + T(n-1) \quad , n \geq 1 \end{cases}$$

T(n)=4n+2

T(n)=O(n)

45