Faculté des Sciences
كلية العلوم
Université Ibn Tofail
جامعة ابن طفيل

Département d'Informatique
Filière MIP - Semestre 2
2024-2025

Module : Informatique II (Algorithmique II / Python)

Chapitre 5

La complexité et preuves des algorithmes

Pr. Khadija Louzaoui

La complexité des algorithmes

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.

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

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

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

- **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**":

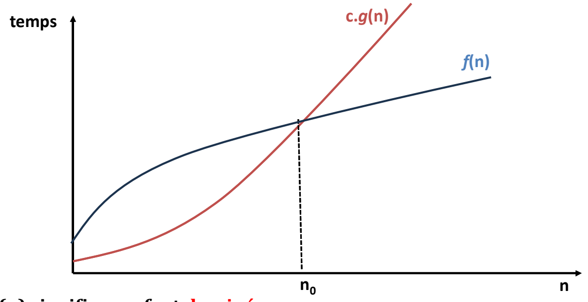
  $f(n) = \Theta(g(n))$ ssi

  $\exists\, c_1 > 0, \exists\, c_2 > 0, \exists\, n_0 \geq 0 \,/\forall\, n > n_0 : c_1.g(n) \leq f(n) \leq c_2.g(n)$

❑ **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 $n \geq n_0$

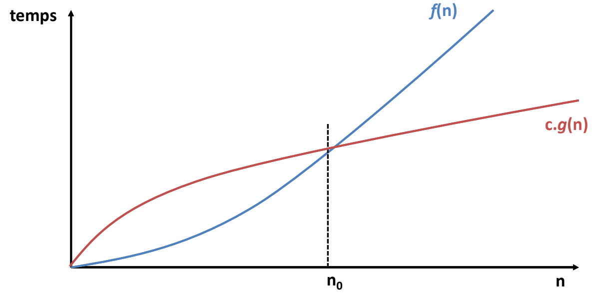


$f = O(g)$ signifie que f est dominée par g.

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 $n \geq n_0$

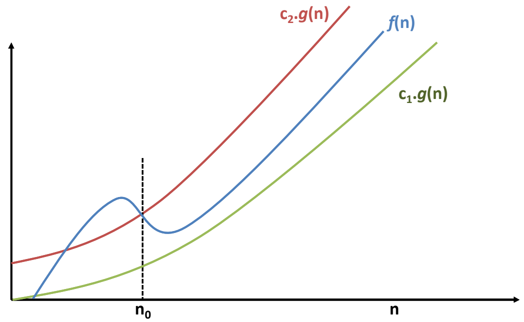


$f = \Omega(g)$ signifie que f domine g.

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.

Pr. Khadija LOUZAOUI

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

Exercice

1- Montrer que $f(n) = 30n^2+5n+10$ est $O(n^2)$

Démonstration :

Il faut trouver $(n_0,c) \in IN^* \times IR^{+*}$ tel que $c > 0, \forall n \geq n_0 : f(n) \leq c.n^2$

On a : $30n^2 \leq 30\, n^2$
$5n \leq 5n^2$
$10 \leq 10n^2$

Alors : $30n^2+5n+10 \leq 45n^2$

Donc : $f(n) \leq 45n^2$

il existe deux constantes:
$c=45$ et $n_0 =1$ telles $c >0$ et que pour tout $n \geq n_0$: $f(n) \leq c.n^2$

Donc *f* est $O(n^2)$

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. $O(f)\,O(g) = O(f\,g)$
10. $f\,O(g) = O(f\,g)$

Notation de Landau

Principales relations entre: O, o, Ω, ω, Θ, ~

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 $\mathbf{T(n) = 3n^2+10n+10}$, 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≥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

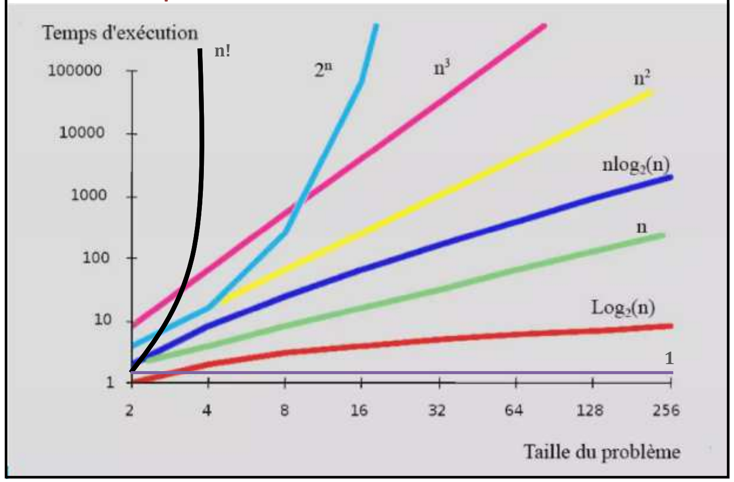
Classe de Complexité	Description	Exemples
O(1)	**Complexité constante :** Le temps d'exécution ne dépend pas de la taille de l'entrée.	Accéder à un élément dans un tableau.
O(log n)	**Complexité logarithmique :** Le temps d'exécution croît logarithmiquement avec la taille de l'entrée.	Recherche binaire.
O(n)	**Complexité linéaire :** Le temps d'exécution est proportionnel à la taille de l'entrée.	Parcourir une liste pour vérifier une condition.
O(n log n)	**Complexité linéaire logarithmique :** C'est typique des algorithmes de tri efficaces.	Tri rapide (Quicksort), Tri fusion (Mergesort).
O(n^2)	**Complexité quadratique :** Le temps d'exécution est proportionnel au carré de la taille de l'entrée.	Tri à bulles, Tri par insertion.
O(n^3)	**Complexité cubique :** Le temps d'exécution est proportionnel au cube de la taille de l'entrée.	Algorithmes de traitement de matrices.
O(2^n)	**Complexité exponentielle :** Le temps d'exécution double à chaque ajout d'élément.	Problèmes de sous-ensemble, Force brute.
O(n!)	**Complexité factorielle :** Le temps d'exécution croît rapidement avec la taille de l'entrée.	Problème du voyageur de commerce (TSP).

Classes de complexité

- Les algorithmes de complexité polynomiale ne sont utilisables que sur des données réduites (p≤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)

33

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 :

$$\left.\begin{array}{ll}\text{Traitement 1} & T_1(n)\\ \text{Traitement 2} & T_2(n)\end{array}\right\} T(n)=T_1(n)+T_2(n)$$

Notation de landau :

$$\left.\begin{array}{ll}\text{Traitement 1} & O(T_1(n))\\ \text{Traitement 2} & O(T_2(n))\end{array}\right\} O(T(n))=O(T_1(n))+O(T_2(n))$$

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

34

Calcul de la complexité: règles pratiques

Exemple:

Temps d'exécution :

$$\left.\begin{array}{ll}a\leftarrow 5 & c_1\\ b\leftarrow (a*3)/2 & c_2\end{array}\right\} T(n)=4$$

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

Donc la complexité est O(1)

Notation de landau :

$$\left.\begin{array}{ll}a\leftarrow 5 & O(1)\\ b\leftarrow (a*3)/2 & O(1)\end{array}\right\} O(1)$$

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

35

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 :

$$\left.\begin{array}{ll}\textbf{Si (condition) Alors} & T_C(n)\\ \quad\text{Traitement 1} & T_1(n)\\ \textbf{Sinon} & \\ \quad\text{Traitement 2} & T_2(n)\\ \textbf{FinSi} & \end{array}\right\} T(n)=T_C(n)+\max(T_1(n),T_2(n))$$

Notation de landau:

$$\left.\begin{array}{ll}\textbf{Si (condition) Alors} & O(T_C(n))\\ \quad\text{Traitement 1} & O(T_1(n))\\ \textbf{Sinon} & \\ \quad\text{Traitement 2} & O(T_2(n))\\ \textbf{FinSi} & \end{array}\right\} O(T(n))=O(\max(T_C(n),T_1(n),T_2(n)))$$

36

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 : $T(n)= 1+max(1,1)=1+1=2$ **donc la complexité est** $O(1)$

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 ├ T(n)=1+(n-2)*2+1
FinPour                                        │
Retourner fact                1                ┘
```

On a : $T(n) =2n-2$ **et** $2n-3\leq 2n+2n$ **alors** $T(n) \leq 4n$
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) ┐ │
        fact ← fact*i     O(1) ┘ ├O(n) ├ 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) ┐
    Traitement           T1(n) ├ m fois
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)) ┐
    Traitement           O(T1(n)) ├ m fois
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          T(n)=7n+6
   FinSi
   i ← i + 1                                2
FinTantQue
Retourner trouve                            1 } 1
```

On a : $T(n) = 7n+6$ **et** $7n+6 \leq 7n+6n$ **alors** $T(n) \leq 13n$

Donc la complexité temporelle dans le pire des cas est $O(n)$.

41

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) } n × max(O(1),O(1))    O(n)
   FinSi
   i ← i + 1                        O(1)
FinTantQue
Retourner trouve                            O(1) } O(1)
```

Donc : $O(T) = O(1) + n \times \max(O(1), O(1)) + O(1) = n \times O(1) = O(n)$

42

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.

43

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                   c1
Pour i de 1 à n faire   c2
    s ← s+i             c3          T(n)=O(?)
FinPour
```

44

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
      s ← s+i                2   2n
FinPour
```

$T(n)=2n+1$

$c_1=1$, $c_3=2$, $c_2=n$
on a $\mathbf{T(n)=c_1+c_3*c_2=2n+1}$ et $2n+1 \le 2n+n$ alors $\mathbf{T(n) \le 3n}$ pour $n\geq1$
Donc la complexité temporelle dans le pire des cas est $\mathbf{O(n)}$.

Notation landau

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

$O(1)+O(n)=O(n)$

La complexité temporelle dans le pire des cas est $\mathbf{O(n)}$.

Calcul de la complexité: règles pratiques

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

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

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

$O(T(n))=O(n)$

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

$T(n)=1+n*n*3+1=3n^2+1$

$O(T(n))=O(n^2)$

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

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

$T(n)=1+(n*(n+1)/2)*3+1$
$=(3/2)n^2+(3/2)n+2$

$O(T(n))=O(n^2)$

Complexité des algorithmes récursifs

On modélise souvent un algorithme récursif à l'aide d'une **relation de récurrence**, puis on la résout pour obtenir la **complexité asymptotique.**

Etapes :

- **Ecrire la relation de récurrence** qui décrit le temps d'exécution.
- **Trouver une solution** (approximative ou exacte) de cette relation.
- **Conclure sur la complexité** (en notation O, Θ ou Ω).

Complexité des algorithmes récursifs

Exemple:

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

Soit $T(n)$ la complexité temporelle de cette fonction.

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

Complexité des algorithmes récursifs

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

Soit $n \in \mathbb{N}^*$ ($T(n)$ suite arithmétique de raison 4).

$$\begin{aligned} T(n) &= 4 + T(n-1) \\ T(n-1) &= 4 + T(n-2) \\ T(n-2) &= 4 + T(n-3) \\ &\vdots \\ T(3) &= 4 + T(2) \\ T(2) &= 4 + T(1) \\ T(1) &= 4 + T(0) \end{aligned}$$

Après la somme, on aura:

$$T(n) = 4n + T(0)$$

$\forall n \in \mathbb{N}$: **T(n)=4n+2** Donc: **T(n)=O(n)**

Preuves des algorithmes

Définition

- Une **preuve d'un algorithme** est une démonstration mathématique qui garantit que **pour toute entrée valide**, l'algorithme **se termine** et **donne un résultat correct** .
- Deux points essentiels à prouver :
  - **Terminaison :** L'algorithme se termine après un nombre fini d'étapes pour toute entrée valide.
    Exemple : une boucle finit toujours car une variable diminue jusqu'à atteindre une condition d'arrêt.
  - **Correction :** Le résultat retourné est conforme au but de l'algorithme.
    Exemple : un algorithme de tri renvoie bien une liste triée.

Méthodes utilisées pour la preuve :

- **L'invariant de boucle :** propriété qui reste vraie à chaque itération d'une boucle.
- **La récurrence :** méthode utilisée pour prouver la correction d'algorithmes récursifs.
- **L'analyse de cas :** on examine tous les cas possibles (cas simples, cas généraux…).
- **Preuve par contre-exemple** (preuve par contradiction): on donne un exemple qui contredit la validité d'un algorithme.

Preuve par invariant de boucle

Définition (Invariant de boucle)

- En algorithmique, **l'invariant de boucle** est une **propriété logique** utilisée pour prouver la **correction** d'un algorithme, en particulier pour les **boucles**.
- Un invariant de boucle est une **propriété** (ou une condition) qui :
  - Est **vraie avant** la première exécution de la boucle,
  - Reste **vraie à chaque itération** (après chaque passage dans la boucle),
  - Permet de **prouver** que l'algorithme fait ce qu'il est censé faire.
- Un invariant de boucle sert à :
  - **Prouver la correction** d'un algorithme,
  - **Montrer que la boucle progresse** vers un objectif (par exemple, trouver une solution, trier une liste, etc.),
  - **Justifier que le résultat final est correct** une fois la boucle terminée.

Preuve par invariant de boucle

Etapes de preuve avec un invariant de boucle

La démonstration de **la correction d'une boucle** suit une démarche formelle, souvent appelée **une preuve par invariant de boucle**.
L'invariant de boucle est **la propriété centrale à démontrer** tout au long de l'exécution de la boucle.

Cette démarche suit généralement trois étapes :

1. **Initialisation :** Montrer que l'invariant est **vrai** à l'itération n°0 de la boucle (**avant** la première exécution de la boucle).
2. **Conservation (ou Hérédité) :** Montrer que si l'invariant est vrai au **début d'une itération**, alors il **reste** vrai après cette itération.
3. **Sortie de boucle :** Montrer que lorsque la boucle se termine (la condition devient fausse), **l'invariant combiné avec la condition d'arrêt** permet de prouver que **la post-condition** (le but de l'algorithme) est atteinte.

Rq: pour une boucle, l'itération n° 0 définit les valeurs initiales des variables.

Preuve par invariant de boucle

Rappel: Propriété logique

Une **propriété** (ou prédicat) noté $\mathcal{P}(n)$ est une expression logique qui devient une **proposition** vraie ou fausse lorsqu'on :

- Remplace ses variables par des valeurs précises.
- Ajoute des quantificateurs à la propriété $\mathcal{P}(n)$.

Exemple :

Considérons le prédicat : $\mathcal{P}(n)$: $n^2 > 60$, $n \in \mathbb{N}$
Ce n'est pas encore une **proposition** tant que **n n'est pas défini**.

- Si on prend $n = 8$, alors $\mathcal{P}(8)$: $8^2 > 60 \Rightarrow 64 > 60$, donc **Vrai**.
- Si on prend $n = 4$, alors $\mathcal{P}(4)$: $4^2 > 60 \Rightarrow 16 > 60$, donc **Faux**.
- $\forall n \in \mathbb{N}, \mathcal{P}(n)$ est **Faux**.
- $\exists n \in \mathbb{N}, \mathcal{P}(n)$ est **Vrai**.

Preuve par invariant de boucle

Rappel : Raisonnement par récurrence

La **démonstration par récurrence** est une méthode utilisée pour prouver qu'une **propriété** est vraie pour **tous les entiers naturels** à partir d'un certain rang (souvent à partir de 0 ou 1).

Elle se fait en trois étapes :

1. **Initialisation** : On vérifie que la propriété est vraie pour le premier entier (par exemple n=0 ou n=1).
2. **Hérédité** : On suppose que la propriété est vraie pour un entier n (c'est l'**hypothèse de récurrence**), et on démontre qu'elle est aussi vraie pour n+1.
3. **Conclusion** : On en déduit que la propriété est vraie pour **tous les entiers naturels** à partir du rang initial.

Chapitre 5 — Preuves des algorithmes

Preuve par invariant de boucle

Rappel : Raisonnement par récurrence

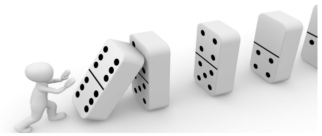
Imagine une suite infinie de dominos alignés.

Pour qu'ils tombent tous, deux conditions sont nécessaires :

1. Faire tomber le premier domino (c'est le point de départ, **l'initialisation**).
2. S'assurer que chaque domino fait tomber le suivant (c'est le principe **d'hérédité**).

Si ces deux conditions sont remplies, alors tous les dominos tombent un à un.

C'est exactement le principe de **la démonstration par récurrence en mathématiques**.



57

Chapitre 5 — Preuves des algorithmes

Preuve par invariant de boucle

Exemple:

Soit l'algorithme suivant :

```
Fonction puissance(x:reel, n:entier): réel
Variables p:réel
Début
   P ← 1
   Pour i de 1 à n pas de 1 faire
      P ← P*x
   Fin pour
   retourner P
Fin
```

1- Donner une preuve formelle de terminaison de cette fonction.

2- Démontrer que la propriété: $\mathcal{P}(\boldsymbol{k})$: $\boldsymbol{P_k = x^k}$; $(0 \le k \le n)$ est un invariant de boucle pour cette fonction. ($\boldsymbol{P_k}$: la valeur stockée dans la variable $\boldsymbol{P}$ à l'issue de l'itération $\boldsymbol{k}$ de la boucle)

3- Quelle est la valeur renvoyée par la fonction.

58

Chapitre 5 — Preuves des algorithmes

Preuve par invariant de boucle

Solution :

1- Preuve formelle de terminaison de la fonction.

On analyse la boucle :

```
Pour i de 1 à n pas de 1 faire
      P ← P*x
Fin pour
```

Définition d'une mesure de terminaison :

On définit la variable de contrôle **i**, qui prend des valeurs de **1** à **n**.
A chaque itération, **i** est incrémenté de 1.

Mesure décroissante vers un cas d'arrêt :

- La boucle effectue un nombre fini d'itérations : de **i = 1** à **i = n**.
- Après **n** itérations, la condition de boucle **i ≤ n** n'est plus vraie.
- Il n'y a pas de boucle infinie ou de saut conditionnel imprévisible.

Conclusion : La fonction termine toujours après **n** étapes.

59

Chapitre 5 — Preuves des algorithmes

Preuve par invariant de boucle

Solution :

2- Preuve formelle que $\mathcal{P}(\boldsymbol{k})$: $\boldsymbol{P_k = x^k}$ est un invariant de boucle.

On définit la propriété : $\mathcal{P}(\boldsymbol{k})$: $\boldsymbol{P_k = x^k}$; $(\boldsymbol{0 \le k \le n})$

Initialisation (k=0):

Avant toute itération (k = 0), on a : P ← 1.
Or, $\boldsymbol{x^0 = 1}$. Donc $\boldsymbol{P_0 = 1 = x^0}$.
$\mathcal{P}(\boldsymbol{0})$ est vraie.

Conservation (Hérédité) :

Supposons que $\mathcal{P}(\boldsymbol{k})$est vraie, c'est-à-dire :
Après la k-ième itération, $\boldsymbol{P_k = x^k}$ $(\boldsymbol{0 \le k \le n-1})$.
Mq : à l'itération suivante **(k + 1)** : $\mathcal{P}(\boldsymbol{k+1})$ est vraie

- à l'itération n° k+1 on exécute l'instruction : P ← P * x
- $\boldsymbol{P_{k+1} = P_k \times x = x^k \times x = x^{k+1}}$ alors $\boldsymbol{P_{k+1} = x^{k+1}}$.

$\mathcal{P}(\boldsymbol{k}) \Longrightarrow \mathcal{P}(\boldsymbol{k+1})$ l'hérédité est vérifiée.

Conclusion : Par principe de récurrence

$\forall \boldsymbol{k} \in \{\boldsymbol{0, 1, \dots, n}\}$: $\mathcal{P}(\boldsymbol{k})$ est vraie

$\forall \boldsymbol{k} \in \{\boldsymbol{0, 1, \dots, n}\}$: $\boldsymbol{P_k = x^k}$

60

Preuve par invariant de boucle

Solution :

3- Quelle est la valeur renvoyée par la fonction.

- A la fin de la boucle (Après **n** itérations), on a établi que:

$$P_n = x^n$$

- Donc la fonction retourne la valeur x^n
- L'algorithme est **correct** et retourne **effectivement le résultat attendu :** x^n

Preuve par récurrence

Définition

- La terminaison et la correction d'un algorithme récursif se montrent simultanément par une **preuve mathématique par récurrence**, car la structure d'un algorithme récursif est très similaire à celle d'une preuve récursive :
  - On traite un **cas de base**
  - Puis on s'appuie sur l'hypothèse que l'algorithme fonctionne pour un cas "plus petit" pour démontrer qu'il fonctionne pour un cas "plus grand".

Preuve par récurrence

Etapes de la preuve par récurrence

La **preuve par récurrence** est la méthode classique pour démontrer la correction d'un **algorithme récursif**. Elle ressemble à une démonstration mathématique par récurrence.
Etapes de la preuve par récurrence pour un algorithme récursif:

2. Cas de base (initialisation): Montrer que l'algorithme fonctionne correctement pour le cas le plus simple (souvent l'appel récursif avec une valeur minimale, comme n= 0 ou n=1).
Ce cas **ne fait pas appel à la récursion**, donc il doit directement produire le bon résultat.

2. Hypothèse de récurrence: On suppose que l'algorithme fonctionne correctement pour une certaine valeur **n=k**.
C'est **l'hypothèse** sur laquelle on s'appuie pour démontrer le cas suivant.

3. Hérédité (Etape de récurrence): En utilisant l'hypothèse de récurrence, on **montre que l'algorithme est correct** pour **n=k+1**.
On vérifie que **l'appel récursif** utilise des valeurs **plus petites**, et que les résultats sont **correctement combinés** pour produire la bonne réponse.

Preuve par récurrence

Exemple:

Montrer la correction de l'algorithme suivant qui renvoie n!.

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

Preuve par récurrence

Solution 1:

Notons, $\forall \boldsymbol{n} \in \mathbb{N}$, $\mathcal{P}(\boldsymbol{n})$, la propriété $\boldsymbol{f_n = n!}$

Initialisation (n=0):

Pour (n = 0), on a : factorielle(0) ← 1.

Or, $\boldsymbol{0}! = \boldsymbol{1}$. Donc $\boldsymbol{f_0 = 1 = 0}!$.

$\mathcal{P}(\boldsymbol{0})$ est vraie.

Hérédité :

Soit $\boldsymbol{n} \in \mathbb{N}$, supposons que $\mathcal{P}(\boldsymbol{n})$ est vraie, c'est-à-dire :

Après le n-ième appel de la fonction, $\boldsymbol{f_n = n}!$.

Mq : A l'appel suivant $(\mathbf{n+1})$: $\mathcal{P}(\boldsymbol{n+1})$ est vraie

- On exécute : factorielle(n+1) ← (n+1)*factorielle(n)
- factorielle(n) termine, donc factorielle(n+1) termine.
- $\boldsymbol{f_{n+1}} = (\boldsymbol{n+1}) \times \boldsymbol{f_n} = (\boldsymbol{n+1}) \times \boldsymbol{n}! = (\boldsymbol{n+1})!$ Alors $\boldsymbol{f_{n+1}} = (\boldsymbol{n+1})!$

$\mathcal{P}(\boldsymbol{n}) \Longrightarrow \mathcal{P}(\boldsymbol{n+1})$ l'hérédité est vérifiée.

Conclusion : Par principe de récurrence

$\forall \boldsymbol{n} \in \{\boldsymbol{0}, \boldsymbol{1}, \dots, \boldsymbol{n}\}$: $\mathcal{P}(\boldsymbol{n})$ est vraie

$\forall \boldsymbol{n} \in \{\boldsymbol{0}, \boldsymbol{1}, \dots, \boldsymbol{n}\}$: $\boldsymbol{f_n = n}!$

L'algorithme est **correct.**

Preuve par récurrence

Solution 2:

On pose $\boldsymbol{f_n}$ la valeur retournée par la fonction factorielle ayant $\boldsymbol{n}$ comme paramètre $(\boldsymbol{n} \geq \boldsymbol{0})$.

On a $\begin{cases} \boldsymbol{f_0 = 1} \\ \boldsymbol{f_n = n \times f_{n-1}} \quad , \boldsymbol{n} \geq \boldsymbol{1} \end{cases}$

Soit $\boldsymbol{n} \geq \boldsymbol{1}$:

On a

$$
\begin{aligned}
\boldsymbol{f_n} &= \boldsymbol{n} \times \cancel{\boldsymbol{f_{n-1}}} \\
\cancel{\boldsymbol{f_{n-1}}} &= (\boldsymbol{n-1}) \times \cancel{\boldsymbol{f_{n-2}}} \\
\cancel{\boldsymbol{f_{n-2}}} &= (\boldsymbol{n-2}) \times \cancel{\boldsymbol{f_{n-3}}} \\
&\vdots \\
\cancel{\boldsymbol{f_4}} &= (\boldsymbol{4}) \times \cancel{\boldsymbol{f_3}} \\
\cancel{\boldsymbol{f_3}} &= (\boldsymbol{3}) \times \boldsymbol{f_2} \\
\cancel{\boldsymbol{f_2}} &= (\boldsymbol{2}) \times \cancel{\boldsymbol{f_1}} \\
\cancel{\boldsymbol{f_1}} &= (\boldsymbol{1}) \times \boldsymbol{f_0}
\end{aligned}
$$

$$\boldsymbol{f_n} = \left(\prod_{i=1}^{n} \boldsymbol{i}\right) \times \boldsymbol{f_0} \qquad (\boldsymbol{f_0 = 1})$$

$\boldsymbol{f_n = n}!$ Pour tout $\boldsymbol{n} \geq \boldsymbol{1}$

$\boldsymbol{f_0 = 1 = 0}!$ Alors

$\boldsymbol{f_n = n}!$ Pour tout $\boldsymbol{n} \in \mathbb{N}$