ALGORITHMIQUE

PART II COMPLEXITÉ
CALCUL DE COMPLEXITÉ

Laboratoire Informatique Image Interaction (L3I)
Université de La Rochelle - Pôle Sciences et Technologie - Avenue Michel Crépeau - 17042 LA ROCHELLE CEDEX 1 France
Tél : +33 (0)5 46 45 82 62 – Fax : 05.46.45.82.42 – Site internet : http://l3i.univ-larochelle.fr/

PLAN DU COURS D'ALGORITHMIQUE LICENCE3

Deux sous objectifs

Partie I : Algorithmique des graphes

- Cours 1 : Définition et parcours (largeur, profondeur, tri topo)
- Cours 2 : Algos polynomiaux classiques (PCC, connexité, MST, …)
- Cours 3 : Quelques algos plus complexes (communautés)

Partie II : Complexité et structures de données

- Cours 1 : Complexité et structures de données
- Cours 2 : Calcul de complexité
- Cours 3 : Classes de complexité et décidabilité
- Cours 4 : Heuristiques et approximations

ALGORITHME VS ALGORITHMIQUE

Algorithme:

« *Description d'une méthode de calcul qui, à partir d'un ensemble de données d'entrée (problème) et une suite finie d'étapes, produit un ensemble de données en sortie (solution)* »

Un algorithme ne peut résoudre que des ***problèmes calculables***

Algorithmique:

« *Science qui étudie les algorithmes pour eux-mêmes, indépendamment de tout langage de programmation* »

- Complexité d'un algorithme ?
- Terminaison d'un algorithme ?
- Correction (validité) d'un algorithme ?
- Existence d'un algorithme ?

HISTORIQUE ET CALCULABILITÉ

- 1800 av J.C.: Premier algorithme (Euclide)

Formulation de règles de calcul (algorithme) pour résoudre certaines opérations. Ex: PGCD et algorithme d'Euclide.

- 9ème siècle: Mot algorithme

Mot ***algorithme*** venant d 'un mathématicien perse (al-Khowa-rismi) qui généralise l'utilisation de règles de calcul

- 1930: Concept d'algorithme (Türing)

Concepts d'algorithme et de calculabilité (distinction entre les problèmes calculables et les problèmes non calculables) introduisant les notions de décidabilité et de complexité.

Travaux de Alan Türing (machine de Türing).

- 1945: Première génération d'ordinateurs

ECRITURE D'UN ALGORITHME

- L'écriture d'un algorithme ne suit pas une syntaxe rigoureuse, mais se doit d'être compréhensible.
- L'algorithme s'adresse à un programmeur, non à une machine (métaphore de la recette de cuisine).

- Tout algorithme doit cependant préciser:
  - Les données d'entrée et le résultat
  - Une suite finie d'opérations élémentaires:
    - Expressions (variables, valeurs, opérateurs)
    - Expressions conditionnelles (if) et répétitives (for, while)
    - Mécanisme d'appel de fonctions

COMPLEXITÉ

- Définition de la complexité d'un algorithme:

« Quantité de ressources (temps, mémoire) nécessaire à son exécution »

- En pratique:

Fonction d'estimation O de l'exécution d'un algorithme indépendamment de son implémentation (machine + langage) qui s'exprime en fonction de la donnée d'entrée, souvent notée n (taille d'un tableau, d'un container)

- A quoi ça sert:
  - Permet de vérifier si un algorithme est « raisonnable »
  - Permet de comparer des algorithmes résolvant le même problème
- Plusieurs complexités:
  - dans le **pire des cas** / meilleur des cas / en moyenne ;
  - en **temps** / en espace mémoire

CALCUL DE COMPLEXITÉ EN TEMPS

- Définition formelle:

Une fonction f(n) du temps d'exécution d'un algorithme est dite O(g(n)) s'il existe des constantes c et no telles que, pour tout $n \geq no$: $f(n) \leq c\ g(n)$

- Principe:

Calculer/Estimer le nombre d'itérations (exécutions) des boucles en fonction de la/les données d'entrée, en tenant compte:

- des boucles imbriquées
- des boucles "cachées" (fonctions appelées)
- des appels de fonctions (boucle équivalente)

Puis simplifier par majoration ce nombre d'itérations

CALCUL DE COMPLEXITÉ

Nom: Recherche d'un élément x dans un tableau

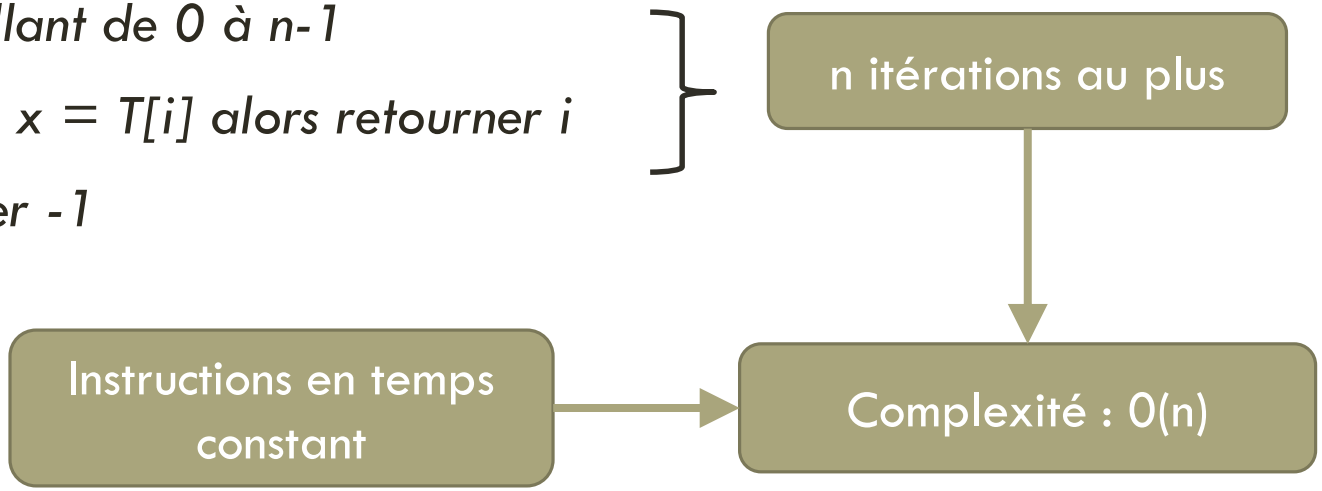
Entrée : un tableau T de n éléments, un élément x

Sortie : le rang de x dans le tableau, -1 sinon

Pour i allant de 0 à n-1

Si x = T[i] alors retourner i

Retourner -1



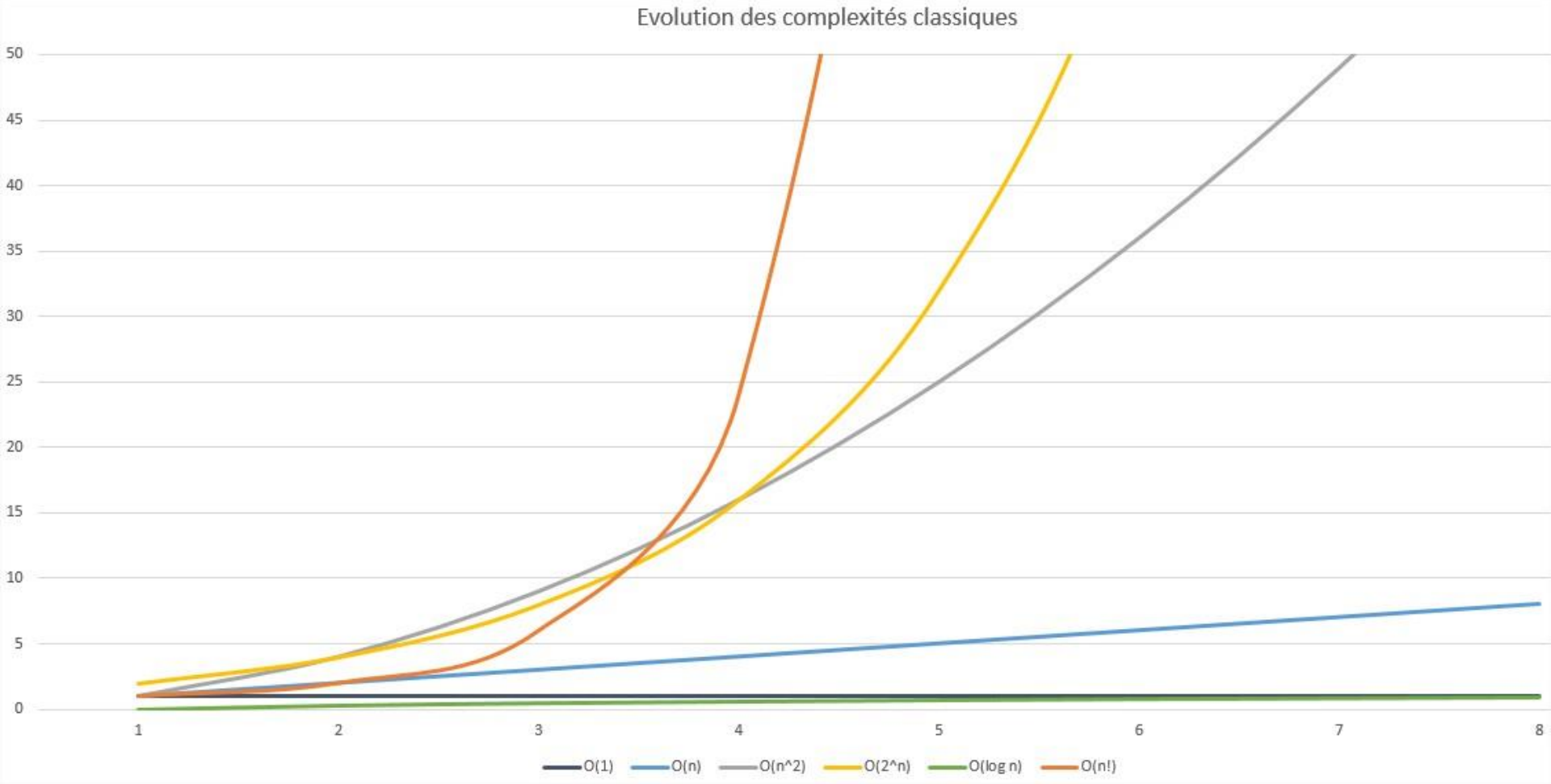
TYPES DE COMPLEXITÉS

- Principaux types de complexité:
  - Complexité constante : O(1)
  - Complexité logarithmique: O(log n)
  - Complexité linéaire: O(n)
  - Complexité polynomiale: $O(n^k)$
  - Complexité exponentielle: $O(2^n)$

- Cas pratiques:
  - Les applications temps-réels nécessitent des algorithmes linéaires
  - Les algorithmes exponentiels sont à manipuler avec précaution pour des données de grande taille (big data)
  - Limiter la complexité pour une *conception responsable des services numériques* (nouvelle posture de conception afin d'intégrer des enjeux de développement durable)

=> Notion d'algorithme polynomial

EVOLUTION DES COMPLEXITÉS CLASSIQUES



EVOLUTION DES COMPLEXITÉS CLASSIQUES

Temps	Type de complexité	n=5	n=20	n=50	n=250	n= 1000	n= 10000	n= 100000
O(1)	Constante	10ns	10ns	10ns	10ns	10ns	10ns	10ns
O(logn)	Logarithmique	10ns	10ns	20ns	30ns	30ns	40ns	40ns
O(n)	Linéaire	50ns	200ns	500ns	2,5µs	10µs	100µs	10ms
$O(n^2)$	Polynomiale	250ns	4µs	25µs	625µs	10ms	1s	2,8h
$O(n^3)$	Polynomiale	1,25µs	80µs	1,25ms	156ms	10s	2,7h	316ans
$O(2^n)$	Exponentielle	320ns	10ms	130jours	10^{59}ans			

ALGORITHMES SUR LES TABLEAUX

- Recherche d'une donnée dans un tableau non trié de n éléments:
  - Recherche simple: O(n)

- Recherche d'une donnée dans un tableau trié de n éléments:
  - Recherche simple: O(n)
  - Recherche dichotomique (tableau trié): **O(log n)**

- Tri d'un tableau de n éléments:
  - Tri par sélection / par insertion : $O(n^2)$
  - Tri rapide / tri à bulles / tri fusion: **O (n log n)**
  - Tri par rang / à casier : O(n+m),

(m: nb de modalités du tableau)

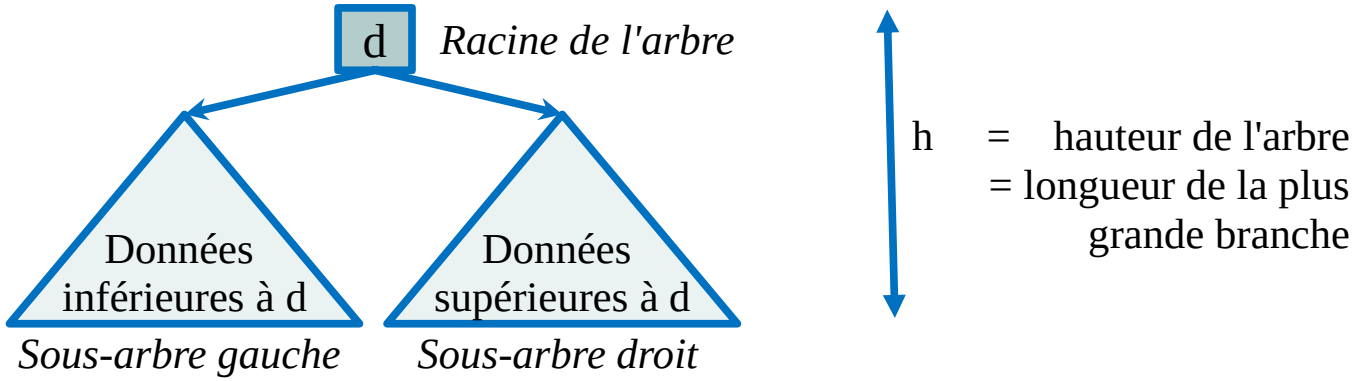
STRUCTURES DE DONNÉES

- Structure de données:

  « *Structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement* »

- Opérations classiques de gestion des données:
  - Recherche d'une donnée (contains, …)
  - Ajout d'une nouvelle donnée (add, put, ...)
  - Suppression d'une donnée existante (remove, get, ...)
- Structures de données classiques:
  - Tableau (avec capacité)
  - Liste (doublement) chaînée, pile, file
  - Arbre binaire de recherche
  - Table de hachage
  - Arbres d'indexation (B-Tree & co)
  - Graphes

ARBRE BINAIRE DE RECHERCHE



- Permet de stocker des données ***comparables*** entre elles
- <u>Opérations de gestion: O(log n)</u>
  - Simple parcours d'une branche de l'arbre: O(h)
  - Lorsque l'arbre est équilibré, on a: $h \approx \log n$
  - Des opérations locales de rotations droite et gauche permettent de maintenir un arbre équilibré après ajout ou suppression:

$$O(h) = O(\log n)$$

TABLE DE HACHAGE

- Description : tableau de m listes stockant n données
  - Nécessite une fonction de hachage h qui détermine, pour chaque donnée id, une et une seule des m listes:
  - h(id) = n° de liste (entre 0 et m-1)

    Ex: h(id) = id modulo (m-1)

- Opérations de gestion : O(1)
  - Nécessitent le parcours de la liste h(id): O(L) (L: longueur liste max)
  - Des fonctions de hachage sophistiquées garantissent une bonne répartition en moyenne des n données dans les m listes:

    $$L \approx n/m$$
  - Lorsque le nombre de données n évolue, on peut également faire évoluer le nombre m de listes de façon à garantir:

    $$\min < n/m < \max$$

    $$O(L) = O(1)$$

COMPLEXITÉ DES STRUCTURES DE DONNÉES

n: nb de données	**Ajout**	**Recherche**	**Suppression**
Liste chaînée	O(1)	O(n)	O(n)
Tableau trié	O(n)	O(log n)	O(n)
Tableau	O(n) / O(1)	O(n) / O(1)	O(n) / O(1)
Pile/File	O(1)	---	O(1)
Arbre de recherche	O(log n)	O(log n)	O(log n)
Table de hachage	O(1) / O(n)	O(1)	O(1) / O(n)

CONTAINERS

- Les containers sont utilisés pour stocker des données selon des règles spécifiques
- Plusieurs types de containers:
  - Séquence : collection de données ordonnées par un index
  - Dictionnaire (structure associative) : collection de données où chaque donnée est une paire (clé-valeur)
  - Ensemble : collection de données uniques et non ordonnées.
- Dans la plupart des langages de programmation, les containers sont implémentés par différentes *structures de données*.
- On choisit le container approprié en fonction :
  - Du type des données (comparables, indexées, clé, ..)
  - De la structure de données :
    - Complexité des opérations à réaliser (ajout, suppression, recherche)
    - Stabilité de la taille du container

CONTAINERS JAVA (PACKAGE JAVA.UTIL)

- Collection : séquence ou ensemble de données
  - Set : ensemble de données (pas d'index, pas de doublons)
    - ***HashSet*** **:** données non ordonnés, table de hachage (fonction de hachage)
    - ***TreeSet*** **:** données ordonnés, arbre binaire de recherche (fonction de comparaison)
  - List : séquence de données (index, possibilité de doublons)
    - ***LinkedList*** : liste doublement chaînée
    - ***ArrayList/Vector*** : tableau, avec une capacité

- Map : dictionnaire (données associatives (clé,valeur))
  - ***HashMap*** : couples non ordonnés, table de hachage (sur la clé)
  - ***TreeMap*** : couple ordonnés (selon la clé), arbre binaire de recherche

STRUCTURES DE DONNÉES PLUS COMPLEXES

Il existe de nombreuses structures de données plus complexes que les collections et les dictionnaires de données:

- Arbre S-play : arbre équilibré dont les données les plus récemment ajoutées sont proches de la racine
- Arbre R-Tree : arbre d'indexation implémenté sous MySQL
- Graphes : modélisent de nombreux problèmes (réseaux informatiques, sémantiques, électriques, sociaux, ordonnancement, données relationnelles, ...)
- ……

REPRÉSENTATIONS D'UN GRAPHE

	Espace	Recherche arc/arête	Ajout arc	Ajout sommet	
Liste de listes	O(n+m)	O(n)	O(1)	O(1)	Graphes peu denses et dynamiques
Matrice carrée	$O(n^2)$	O(1)	O(1)	$O(n^2)$	Graphes denses et statiques
Liste d'arêtes (triplets)	O(m)	O(m)	O(1)	0(m)	Graphes du web

COMPLEXITÉ DU PARCOURS EN LARGEUR

Nom: Parcours en largeur

Entrée : un graphe G de n sommets et m arêtes, un sommet s

Initialiser une file F et y ajouter s

etat (s) = dansfile

tant que F n'est pas vide

t = tête de F

pout tout u ∈ adj(t)

si etat(u) = inexploré

ajouter u dans F

etat(u) = dansfile

défiler F

état(t) = exploré

n itérations au plus
(chaque sommet une fois dans la file)

n itérations au plus
(chaque sommet a au plus n voisins)

Instructions en temps constant

Complexité : $O(n^2)$

COMPLEXITÉ DU PARCOURS EN LARGEUR

Nom: Parcours en largeur

Entrée : un graphe G de n sommets et m arêtes, un sommet s

Initialiser une file F et y ajouter s

etat (s) = dansfile

tant que F n'est pas vide }

t = tête de F

pout tout u ∈ adj(t) }

si etat(u) = inexploré

ajouter u dans F

etat(u) = dansfile

défiler F

état(t) = exploré

n itérations au plus
(chaque sommet une fois dans la file)

m itérations en tout
(chaque arête visitée une seule fois)

Instructions en temps constant

Complexité : O(n+m)

RÈGLES DE CALCUL

Boucle for interne:

- n itérations au plus : **$O(n^2)$**
- m itérations en tout : **$O(n+m)$**

La seconde analyse est plus fine et permet d'obtenir une meilleure complexité

Pour calculer finement la complexité d'un algorithme, il est nécessaire d'utiliser des règles de calcul:

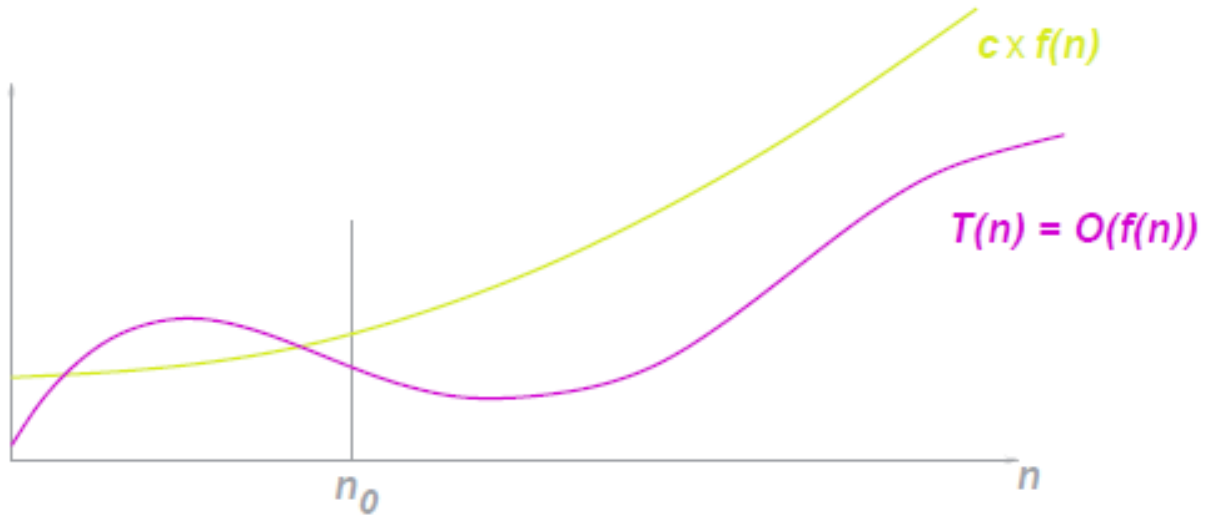
- Sommes
- Equations récursives

PRINCIPE GÉNÉRAL DU CALCUL DE COMPLEXITÉ

- n : taille des données
- T(n) : nombre d'opérations élémentaires
- f(n) : complexité (constante, linéaire, polynomiale, exponentielle)

T(n) = O(f(n))

si il existe des constantes c et no telles que, pour tout $n \geq no$:
$T(n) \leq c\ f(n)$



EXEMPLES

$T(n) = n^3 + 2n^2 + 4n + 2$

$= O(n^3)$ car $T(n) \leq 8n^3$ pour n strictement positif

$T(n) = n \log n + 12n - 10$

$= O(n \log n)$ car $T(n) \leq n \log n$ pour $n > 5/6$

$T(n) = 10n^{10} + 8n^8 + 2^n / 1000$

$= O(2^n)$

CONCLUSION

- **Cours 1 :** Complexité et structures de données
  - Savoir choisir la structure de données - le container appropriée aux données
  - Savoir choisir un algorithme en fonction de sa complexité

- **Cours 2 :** Calcul de complexité
  - Savoir calculer la complexité d'un algorithmes (fonctions récursives)