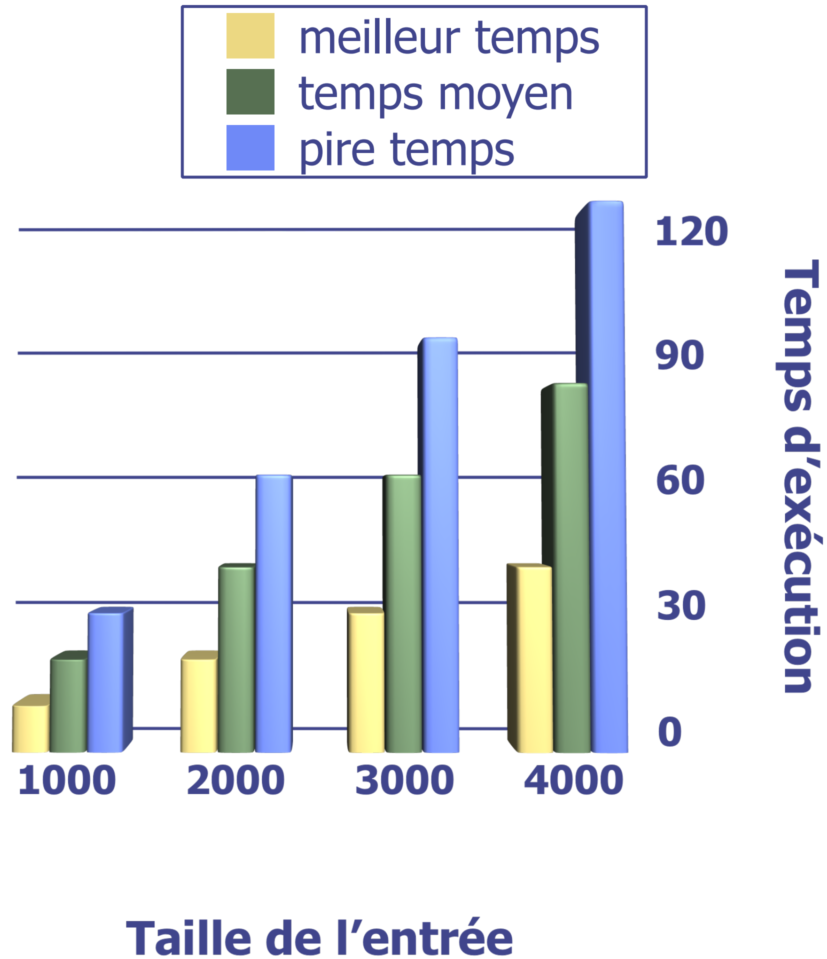
- Analyse empirique
- Analyse théorique
- Pseudo-code
- Sept fonctions importantes
- Opérations primitives
- Estimation du temps d'exécution
- Comparaison de 2 algorithmes
- Facteurs constants
- Notation O
- Analyse asymptotique
- Variantes de O

Analyse empirique

L'analyse empirique nécessite que l'algorithme sous étude a été implémenté. Ainsi, nous pouvons mesurer son temps d'exécution sur différents jeux de données.

```
from time import time
# enregistre le temps au départ
temps_debut = time( )
execute_algorithme
# enregistre le temps à la fin
temps_fin = time( )
# calcule le temps d'exécution
temps_execution = temps_fin - temps_debut
```

Le temps d'exécution la plupart du temps croit avec la taille de l'entrée.



Limitations de l'approche empirique

- Il est nécessaire d'implémenter l'algorithme, ce qui peut s'avérer difficile
- Les résultats peuvent ne pas être représentatifs de tous les jeux de données possibles
- Pour comparer deux algorithmes il faut utiliser le même hardware et le même environnement du système.

Analyse théorique

- Utilise une description haut-niveau de l’algorithme (pseudo-code) plutôt qu’une implémentation
- Caractérise le temps d’exécution en fonction de la taille, n, de l’entrée
- Prend en compte tous les jeux de données possibles
- Permet d’évaluer la vitesse d’un algorithme de manière indépendante du hardware et de l’environnement du système.

Pseudo-code

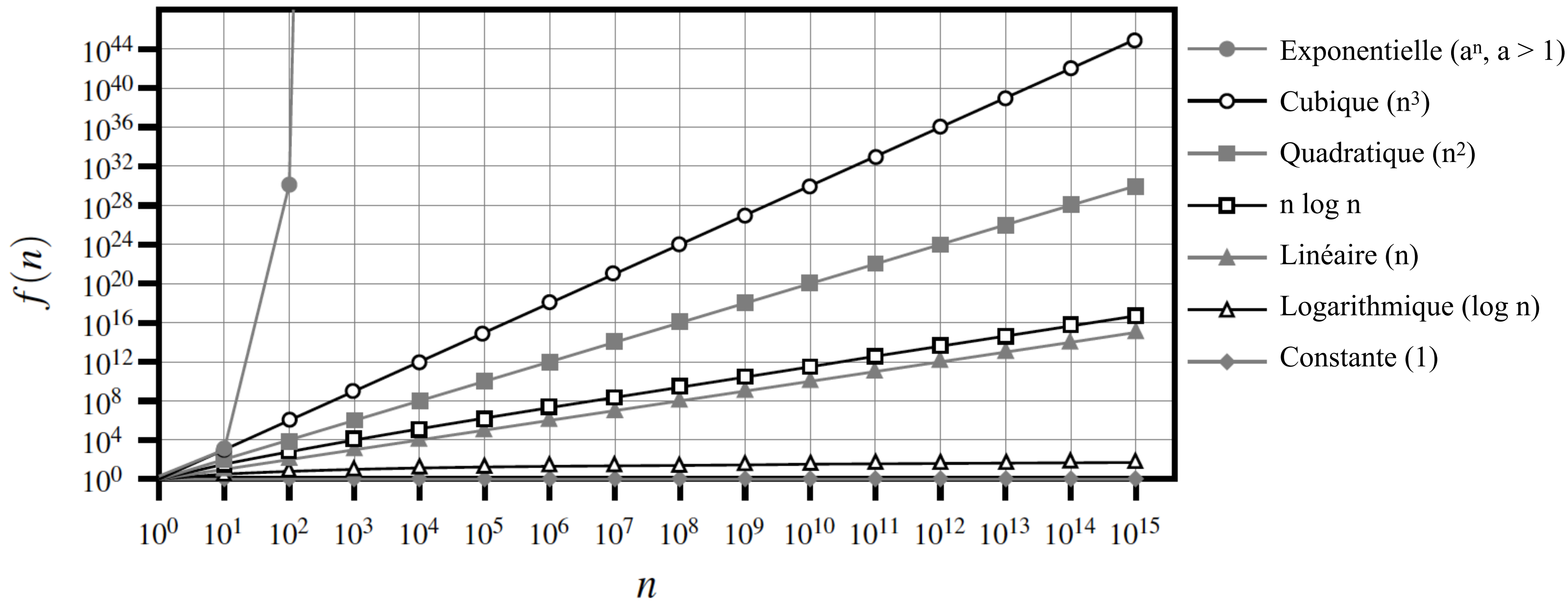
- Description haut-niveau d’un algorithme
- Plus structuré qu’une langue naturelle
- Moins détaillé qu’un programme
- Masque les problèmes de conception de programme

Exemple de pseudo-code

```
# code en python
def recherche_binaire_iterative( data, cible ):
    min = 0
    max = len( data ) - 1
    while min <= max:
            milieu = ( min + max ) // 2
            if cible == data[milieu]:
                return True
            elif cible < data[milieu]:
                max = milieu - 1
            else:
                min = milieu + 1
    return False
```

```
# pseudo code
recherche_binaire_iterative( data, cible )
    min = 0
    max = n – 1
    while min <= max do
        milieu = ( min + max ) / 2
        if cible = data[milieu]
            return true
        else if cible < data[milieu]
            max = milieu – 1
        else min = milieu + 1
    return false
```

Sept fonctions importantes



Les taux de croissance pour les 7 fonctions fondamentales utilisées dans l'analyse algorithmique. La base a = 2 est utilisée pour la fonction exponentielle. Les fonctions sont tracées sur un graphique log-log pour comparer les taux de croissance principalement en tant que pentes. La fonction exponentielle se développe trop rapidement pour afficher toutes ses valeurs sur le graphique.

Temps d’exécution souhaitables

Idéalement, on voudrait que les opérations de structure de données s'exécutent dans des temps proportionnels à la fonction constante ou logarithme, et que nos algorithmes s'exécutent en temps linéaire ou n log n.

Les algorithmes avec des temps d’exécution quadratiques ou cubiques sont moins pratiques, et les algorithmes avec des temps d’exécution exponentiels sont irréalisables pour toutes les entrées sauf très petites.

Opérations primitives

- Calculs de base exécuté par un algorithme
- Identifiables dans le pseudo-code
- Définitions précises non importante (nous verrons pourquoi plus tard)
- Prennent une quantité de temps constante

- Examples:
  - Évaluer une expression
  - Assigner une valeur à une variable
  - Indexer dans un tableau
  - Appeler une méthode
  - Terminer une méthode

Compter les opérations primitives

En inspectant le pseudocode, on peut déterminer le nombre maximum d'opérations primitives exécutées par un algorithme, en fonction de la taille d'entrée

```
def trouve_max( data ):                                                        2 opérations
    """Retourne l'élément maximum d'une liste Python non vide."""
    max = data[0]     #valeur initiale à battre                                 2 opérations
    for _ in data:    #pour tous les éléments                                  2n opérations
        if _ > max:   #si il est plus grand que max                             3n opération
            max = _   #on a trouvé un nouveau max                         0 à 2n opérations
    return max        #quand la boucle termine, max est l'élément maximum       2 opérations
```

- **trouve_max** exécute $7n + 6$ opérations primitives dans le pire cas et $5n + 6$ dans le meilleur des cas. Définissons :

  $\boldsymbol{a}$ = Temps d'exécution de la plus rapide des opérations primitives

  $\boldsymbol{b}$ = Temps d'exécution de la plus lente des opérations primitives

- Prenons $\boldsymbol{T}(\boldsymbol{n})$ comme étant le temps du pire cas de **trouve_max**. Alors

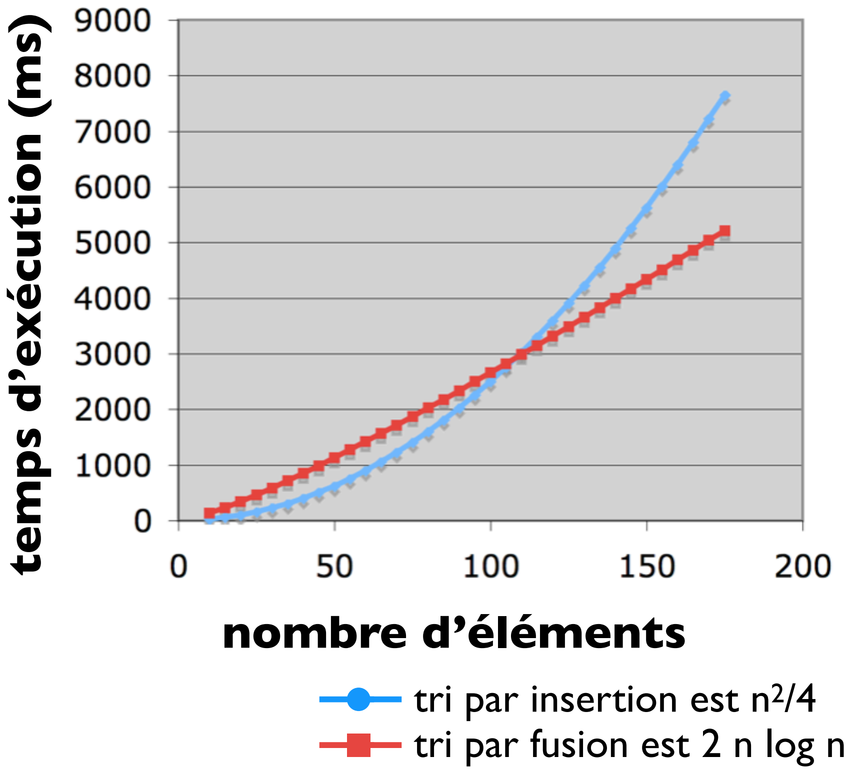
$$\boldsymbol{a}\,(5\boldsymbol{n} + 6) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(7\boldsymbol{n} + 6)$$

- Donc, le temps d'exécution $\boldsymbol{T}(\boldsymbol{n})$ est borné par deux fonctions linéaires.

Taux de croissance du temps d'exécution

- Changer le hardware ou l'environnement système affecte $T(n)$ par un facteur constant, mais ne change pas le taux de croissance de $T(n)$
- Le taux de croissance du temps d'exécution $T(n)$ est une propriété intrinsèque de l'algorithme **trouve_max**

Comparaison de 2 algorithmes

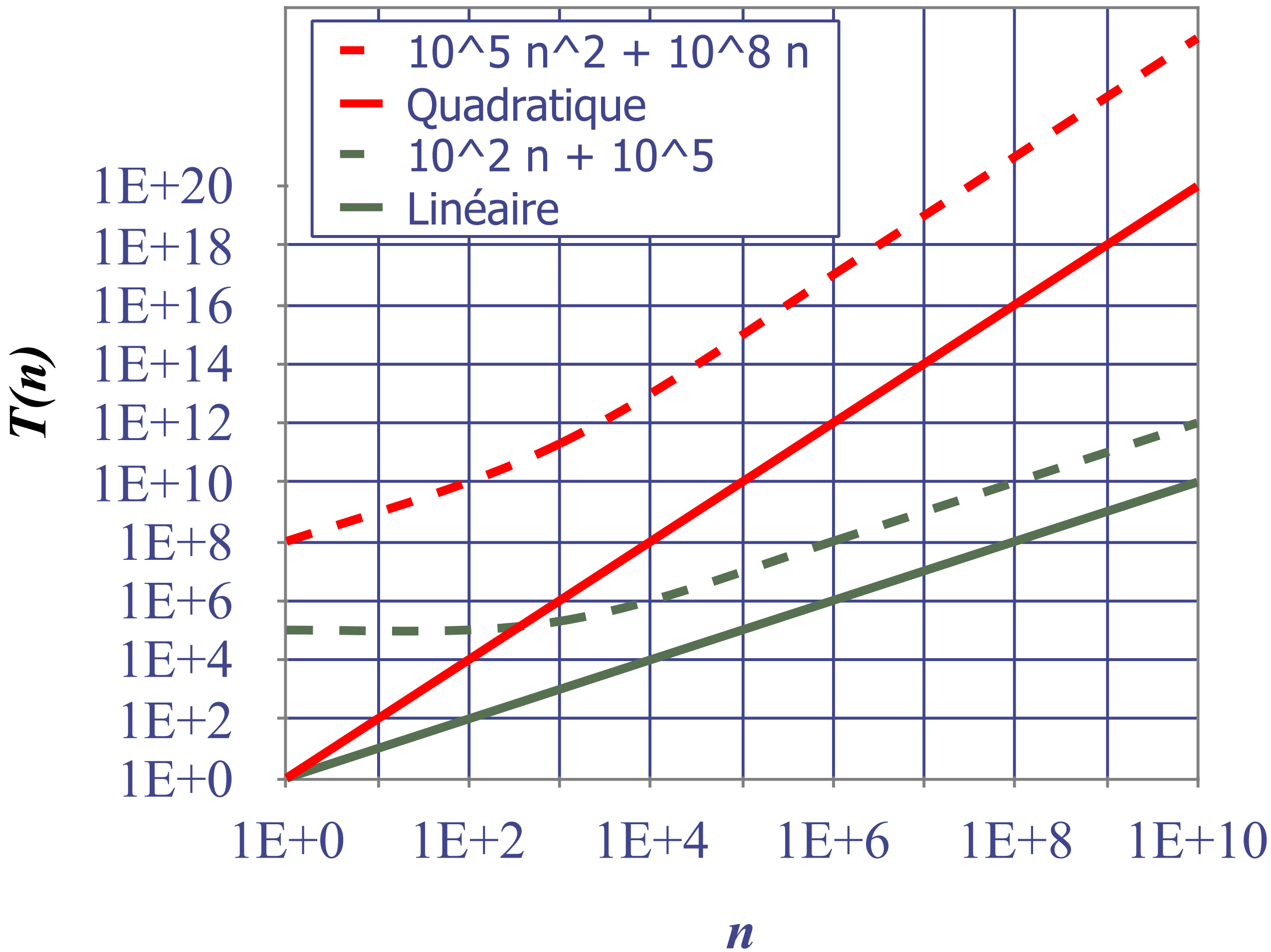


Trier un million d'éléments par insertion prend environ 70 heures alors que par fusion prend 40 secondes.

Sur une machine 100 x plus rapide, on aura 40 minutes versus 0.5 seconde.

Facteurs constants

- Le taux de croissance n'est pas affecté par :
  - facteurs constants ou
  - termes de plus petits ordres
- Exemples
  - $10^2 n + 10^5$ est une fonction linéaire
  - $10^5 n^2 + 10^8 n$ est une fonction quadratique

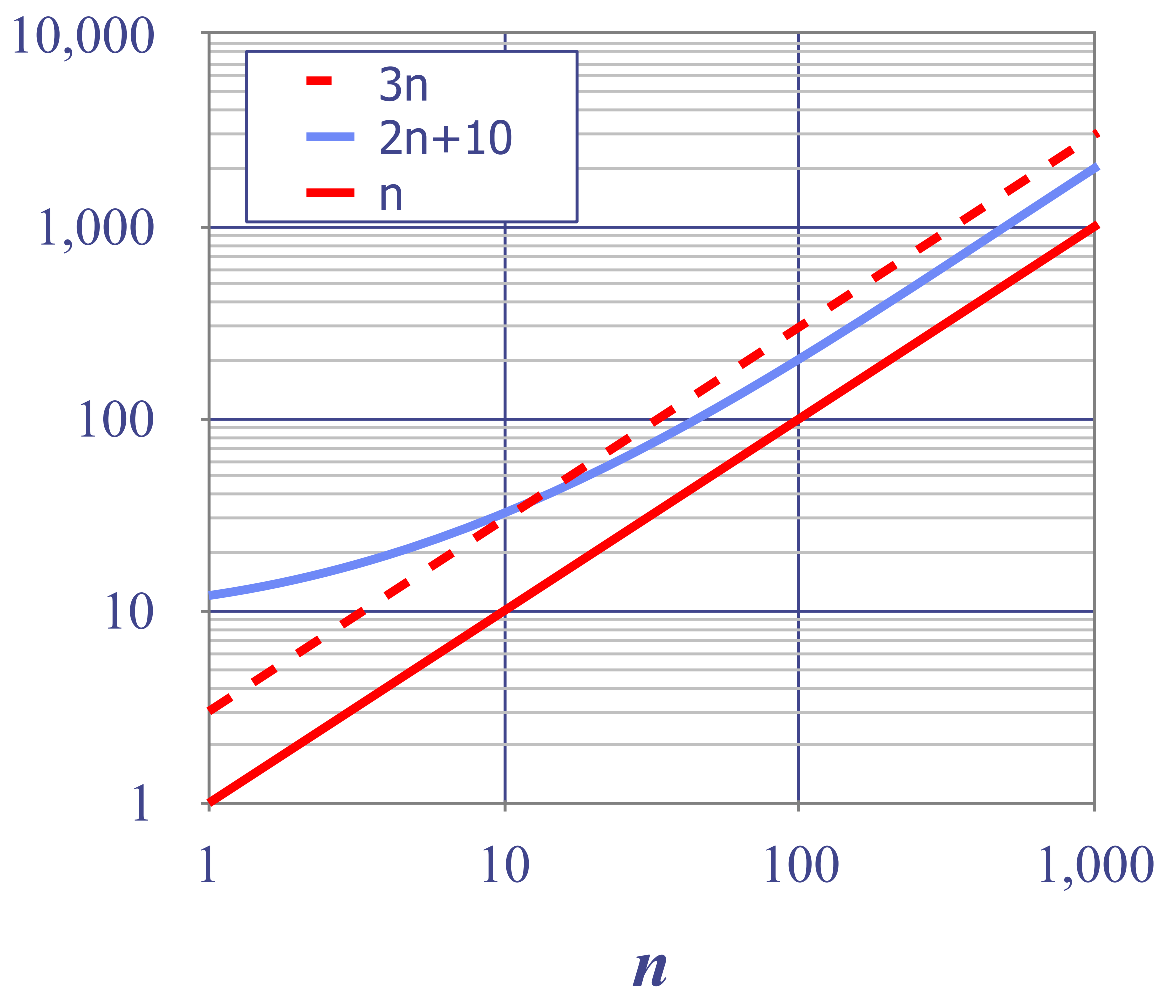


Notation O

- Soit deux fonctions $f(n)$ et $g(n)$, on dit que la fonction $f(n)$ est dans l'ordre de $g(n)$, $O(g(n))$, si il existe des constantes positives c et n_0 tel que

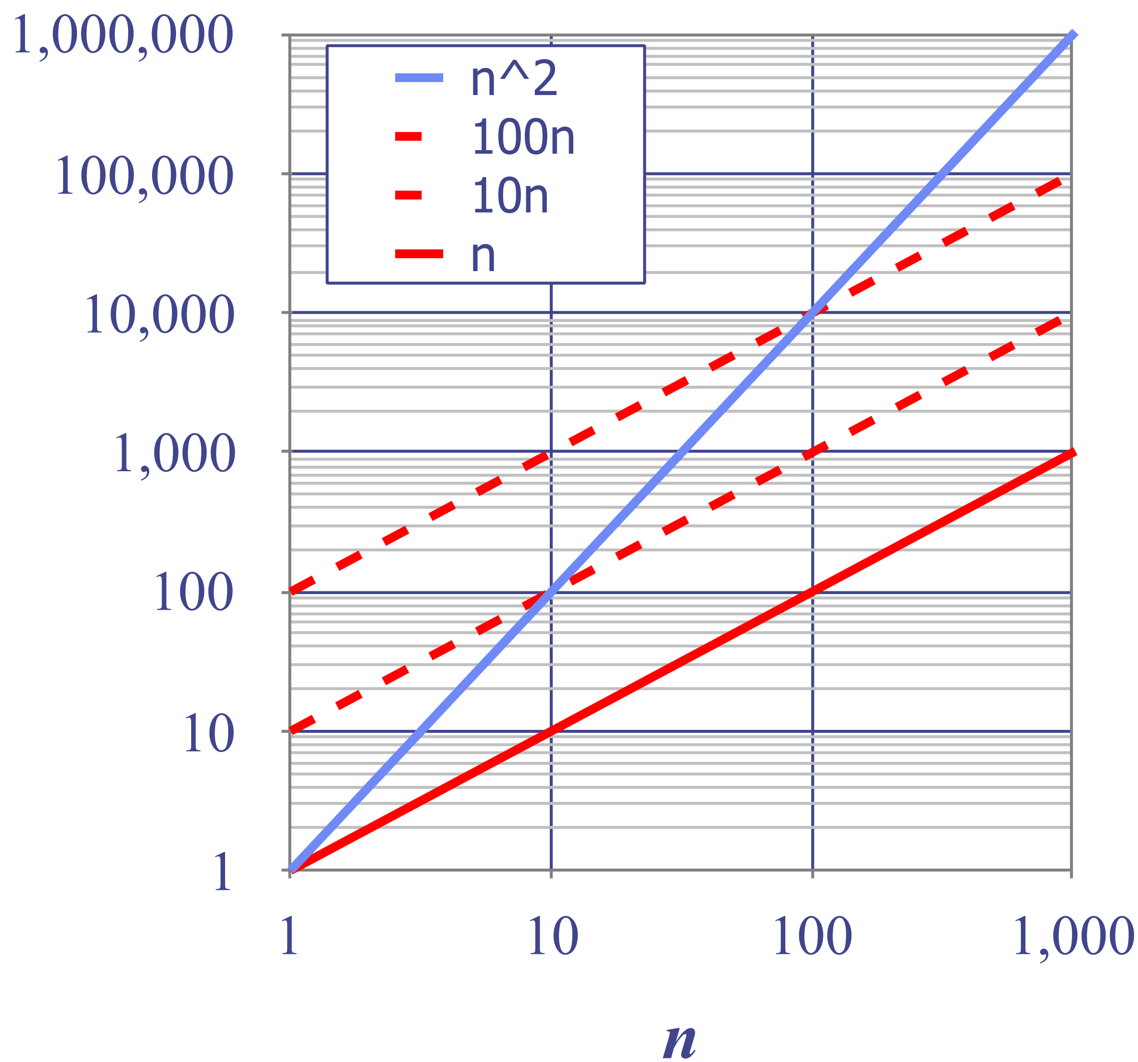
  $f(n) \leq cg(n)$ pour $n \geq n_0$

- Exemple: $2n + 10$ est dans $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Prenons $c = 3$ et $n_0 = 10$



Un exemple de n'est pas dans O de

- La fonction n^2 n'est pas dans $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - Puisque c est une constante, l'inégalité ne peut pas être satisfaite.



O et taux de croissance

- O donne une borne supérieure sur le taux de croissance d'une fonction
- L'énoncé "$f(n)$ est dans $O(g(n))$" veut dire que le taux de croissance de $f(n)$ n'est pas plus grand que celui de $g(n)$
- On utilise la notation O pour classer les fonctions selon leur taux de croissance

	$f(n)$ est dans $O(g(n))$	$g(n)$ est dans $O(f(n))$
$g(n)$ croit plus rapidement que	Oui	Non
$f(n)$ croit plus rapidement que	Non	Oui
possède la même croissance que	Oui	Oui

Règles de simplification de O

- Si $f(n)$ est un polynôme de degré d, alors $f(n)$ est dans $O(n^d)$, i.e.,
  - On peut laisser tomber les termes d'ordre inférieur
  - On peut laisser tomber les facteurs constants
- On utilise la plus petite classe de fonctions
  - Disons que "$2n$ est dans $O(n)$" plutôt que "$2n$ est dans $O(n^2)$" (même si c'est vrai)
- On utilise la plus simple expression de la classe
  - On dit "$3n + 5$ est dans $O(n)$" plutôt que "$3n + 5$ est dans $O(3n)$"(même si c'est vrai)

Analyse asymptotique

- L'analyse asymptotique d'un algorithme détermine son temps d'exécution avec $\boldsymbol{O}$
- Pour ce faire :
  - On trouve le nombre d'opérations primitives exécutées dans le pire cas en fonction de la taille des données entrées
  - On exprime cette fonction avec la notation $\boldsymbol{O}$
- Exemple:
  - On dit que l'algorithme **trouve_max** vu précédemment "s'exécute en temps dans $\boldsymbol{O}(\boldsymbol{n})$"
- Comme les facteurs constants et le termes d'ordre inférieur sont finalement supprimés, on peut les ignorer lorsqu'on compte le numbre d'opérations primitives.

Variantes de O

- **O (grand O)**

  f(n) est dans O(g(n)) si il existe une constante $c > 0$ et une constante entière $n_0 \geq 1$ tel que

  $$f(n) \leq c \times g(n) \text{ pour } n \geq n_0$$

- **Ω (Omega)**

  f(n) est dans Ω(g(n)) si il existe une constante réelle $c > 0$ et une constante entière $n_0 \geq 1$ tel que

  $$f(n) \geq c \times g(n) \text{ pour } n \geq n_0$$

- **Θ (Theta)**

  f(n) est dans Θ(g(n)) si il existe des constantes $c' > 0$ et $c'' > 0$ et une constante entière $n_0 \geq 1$ tel que

  $$c' \times g(n) \leq f(n) \leq c'' \times g(n) \text{ pour } n \geq n_0$$

Intuition sur les notations

- ***O* (grand O)**
  f(n) est dans O(g(n)) si f(n) est asymptotiquement <u>moins ou égale</u> à g(n)

- **Ω (Omega)**
  f(n) est dans Ω(g(n)) si f(n) est asymptotiquement <u>plus grand ou égale</u> à g(n)

- **Θ (Theta)**
  f(n) est dans **Θ**(g(n)) si f(n) est asymptotiquement égale à g(n)