Algorithmique E3

Daniel Vaz

daniel.ramosvaz@esiee.fr

2025-11-17

ESIEE PARIS

Objectifs et déroulement

Objectifs principaux du cours

- Comprendre et appliquer :
  - Structures de données de base
  - Algorithmes de tri
  - Notions de complexité
- Analyser la complexité d'un algorithme
- Écrire des algorithmes efficaces

Contenu du cours

C1 Introduction aux algorithmes

C2 Complexité

C3 Les tableaux

C4 Algorithmes de tri simples

C5 Algorithmes de tri avancés

C6 Structures de données

Séances TD / TP

9 séances pratiques sous forme de TD et TP. Pendant les séances de TD/TP, vous êtes encouragé.e.s à :

- travailler en binôme,
- programmer en C (dans les TP),
- produire votre propre réponse individuelle après discussion (pas de programmation en binôme).

N'oubliez pas de venir avec :

- des feuilles de brouillon
- de quoi écrire (stylo ou crayon à papier + une gomme)
- TD : des feuilles "propres" pour écrire la solution finale (qui vous servira de support de révision pour l'examen).

Exercices TP

Trois types d'exercices de programmation en C :

1. Libres : vous écrivez le code et testez le résultat
2. Testés : structure du code et tests fournis, vous écrivez certaines fonctions et exécutez les tests
3. PLaTon : exercices évalues par la plateforme PLaTon, qui permettent de "valider" le TP

Exercices TP – Libre

```c
#include <stdio.h>

int multi(int a, int b) {
    return a*b;
}

int main() {
    printf("%d\n", multi(5,3));
    printf("%d x %d = %d\n", 123, 456, multi(123,456));
}
```

```
$ gcc multi.c -Wall -o multi
$ ./multi
15
123 x 456 = 56088
```

Exercices TP – Testé

```
1 // ...
2
3 /* quicksort version de base */
4 int tri_rapide_1(double t[], int n) {
5     /* a ecrire */
6 }
7
8 // ...
```

```
$ make
...
$ ./main
Tri rapide sur tableau aleatoire de 10000 nombres :
        Tri effectue en 0.002 s
```

Exercices TP – PLaTon

https://elearning.univ-eiffel.fr/course/view.php?id=13802

Évaluation

- Examen de 2 heures le 23/01/2025
- Les TD / TP ne sont pas notés
  - Mais il est fortement conseillé de faire tous les exercices
- L'assiduité sera prise en compte lors du jury de semestre.

Intervenants

Cours : Daniel VAZ, daniel.ramosvaz@esiee.fr

TD / TP :

- Mourad BEN HADJ
- Clément CHENEVIÈRE
- Mohamed-Amine KHELASSI
- Moussa LAMAMRA
- Daniel VAZ
- Teng WU
- Jean-Giono ZEHOUNKPE

ChatGPT ?

Il est fortement recomendé de :

- Ne pas utiliser l'IA pour écrire du code
- Poser des questions au chargé de TD/TP
- Utiliser l'IA uniquement comme outil de recherche
    - Trouver une fonction, expliquer une erreur

Si l'IA écrit votre code, vous ne développerez pas vos compétences

- Et vous aurez des difficultés à l'examen

Aperçu

Introduction aux Algorithmes

Qu'est-ce qu'un Algorithme ?

Algorithme : une séquence d'instructions bien définies permettant de résoudre un problème ou d'effectuer une tâche.

Caractéristiques :

- Précis : chaque étape est claire et sans ambiguïté
- Fini : l'exécution se termine après un nombre fini d'étapes
- Un algorithme prend des données en entrée...
- ... et les traite pour produire un résultat en sortie

Généralement spécifié sous forme de pseudo-code

Un Exemple

Algorithme SommeListe

Données: tableau t de taille n

Résultat: somme des éléments du tableau t

1: $s \leftarrow 0$
2: **for** $0 \leq i < n$ **do**
3: $\quad s \leftarrow s + t[i]$
4: **end for**
5: **return** s

Quel est le résultat de l'exécution sur le tableau $[2, 8, 3, 7]$?

Allez sur wooclap.com et utilisez le code **FMZYUF**

Introduction aux Algorithmes

Instructions

Un Exemple

Algorithme SommeListe

Données: tableau t de taille n

Résultat: somme des éléments du tableau t

1: $s \leftarrow 0$
2: **for** $0 \leq i < n$ **do**
3: $\quad s \leftarrow s + t[i]$
4: **end for**
5: **return** s

Instructions

Affectation : stocke une valeur dans une variable

$x \leftarrow \langle \texttt{expression} \rangle$

Condition : exécute des instructions si une condition est vraie

if $\langle \texttt{condition} \rangle$ **then**
    $\langle \texttt{instructions} \rangle$
end if

- **else if** $\langle \texttt{condition} \rangle$: si la condition est vraie et toutes les conditions précédentes sont fausses
- **else** : si toutes les conditions précédentes sont fausses

Instructions (2)

Boucle for : répète l'exécution des instructions pour chaque valeur (par défaut dans l'ordre croissant)

for $a \leq i < b$ **do**
    ⟨`instructions`⟩
end for
for $i = a, a+1, a+2, \ldots, b-1$ **do**
    ⟨`instructions`⟩
end for

- **while** ⟨`condition`⟩ : continue à s'exécuter tant que la condition est vraie

Return : termine l'exécution avec le résultat donné

return ⟨`valeur`⟩

Introduction aux Algorithmes

Exemples

Exercice : Test de primalité

Problème : vérifier si un entier n donné est un nombre premier (divisible uniquement par 1 et par lui-même)

Algorithme TestPrimalite

Données: un nombre entier $n > 1$

Résultat: **true** si n est premier, **false** sinon

```
1: for 2 ≤ i < n do
2:     if n%i = 0 then
3:         return false
4:     end if
5: end for
6: return true
```

Exercice 2 : Compter dans un Tableau

Problème : compter combien de fois un élément apparaît dans un tableau

Algorithme Compter

Données: un tableau d'entiers t de taille n, un entier x
Résultat: le nombre d'occurrences de x dans t

```
1: c ← 0
2: for 0 ≤ i < n do
3:     if t[i] = x then
4:         c ← c + 1
5:     end if
6: end for
7: return c
```

Exercice 3 : Recherche dans un tableau

Problème : trouver un élément dans un tableau

Algorithme Recherche

Données: un tableau d'entiers t de taille n, un entier x

Résultat: **true** si x est dans t, **false** sinon

Exercice 3 : Recherche dans un tableau

Problème : trouver un élément dans un tableau

Algorithme Recherche

Données: un tableau d'entiers t de taille n, un entier x

Résultat: **true** si x est dans t, **false** sinon

```
1: for 0 ≤ i < n do
2:     if t[i] = x then
3:         return true
4:     end if
5: end for
6: return false
```

Exercice 3 : Recherche dans un tableau

Problème : trouver un élément dans un tableau

Algorithme Recherche

Données: un tableau d'entiers t de taille n, un entier x

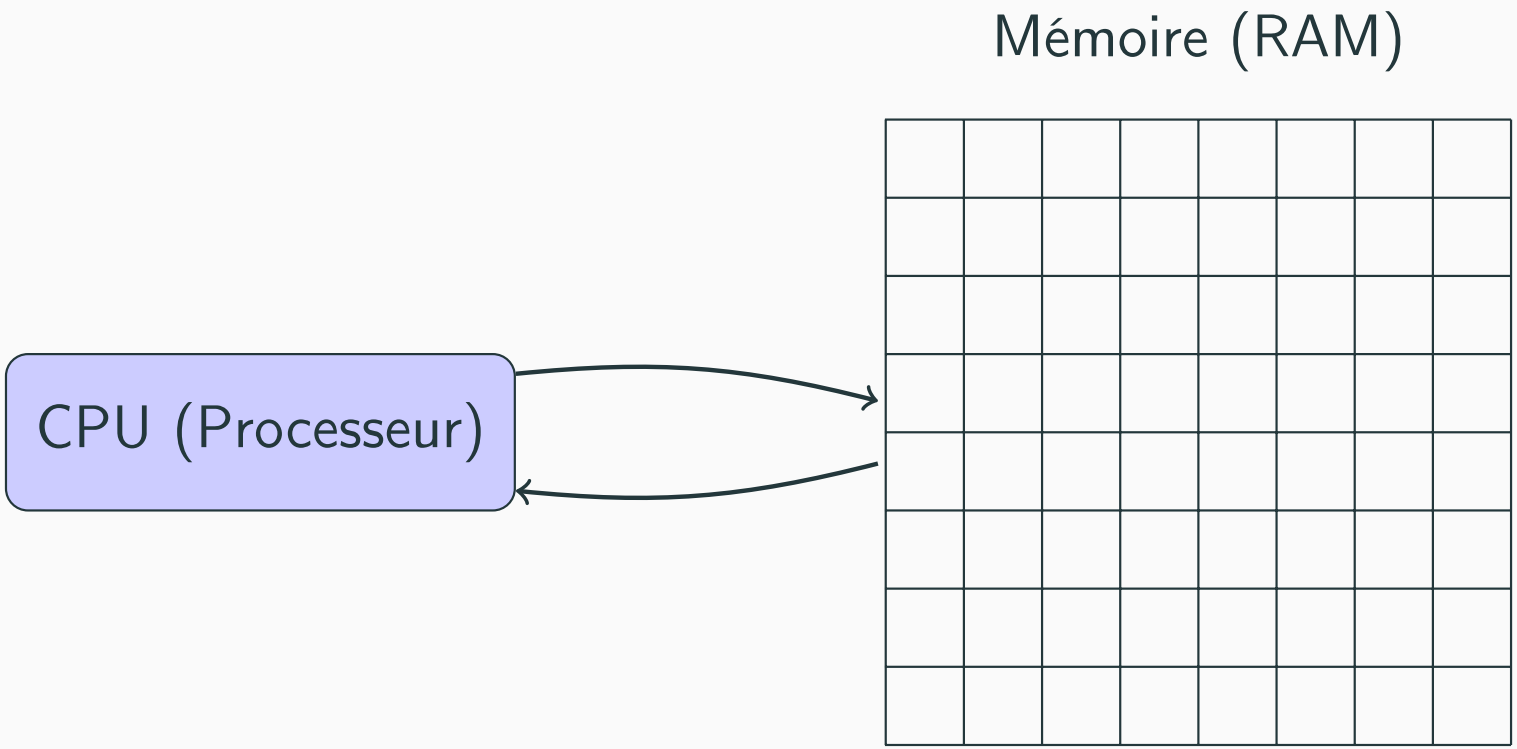
Résultat: **true** si x est dans t, **false** sinon

```
1: f ← false
2: for 0 ≤ i < n do
3:     if t[i] = x then
4:         f ← true
5:     end if
6: end for
7: return f
```

Introduction aux Algorithmes

Exécution dans un ordinateur

Architecture CPU et Mémoire



La cuisine informatique : CPU et Mémoire

CPU : charge le code et exécute les instructions

- et charge les valeurs depuis la mémoire si nécessaire

Mémoire : stocke toutes les valeurs utilisées dans le programme

- Lorsqu'une variable est créée, la place nécessaire est allouée dans la mémoire
- Le processeur sait où se trouve chaque variable dans la mémoire

Exemple : Addition de deux nombres

$a \leftarrow 5$
$b \leftarrow 10$
$c \leftarrow a + b$

a	5
b	10
c	15

Opérations de la ligne 3

1. **Charger** la valeur 5 de la mémoire vers le CPU
2. **Charger** la valeur 10 de la mémoire vers le CPU
3. **Additionner** les deux valeurs
4. **Stocker** le résultat (15) dans la mémoire à l'adresse de c

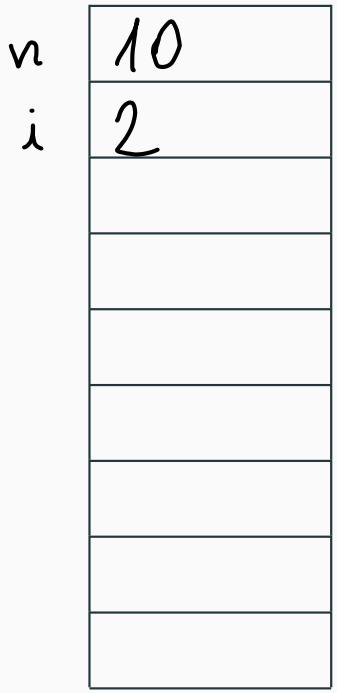
Exercice : Test de primalité

Algorithme TestPrimalite

Données: un nombre entier $n > 0$

Résultat: **true** si n est premier, **false** sinon

```
1: for 2 ≤ i < n do
2:     if n%i = 0 then
3:         return false
4:     end if
5: end for
6: return true
```



Introduction aux Algorithmes

Récursivité et Appels à fonctions

Les fonctions

Fonction : Unité de code

- qui sépare une portion de code pour le structurer
- qui peut être appelée pour exécuter son code
- qui accepte des paramètres et peut retourner une valeur

Lorsqu'une fonction est appelée :

- un espace est réservé dans la mémoire pour l'appel
- les paramètres sont copiés dans cet espace
- les variables créées dans la fonction sont également stockées dans cet espace
- cet espace disparaît (est libéré) lorsque la fonction se termine

Test de primalité

Algorithme TestPrimalite

Données: un nombre entier $n > 0$

Résultat: **true** si n est premier, **false** sinon

```
1: for 2 ≤ i < n do
2:     if n%i = 0 then
3:         return false
4:     end if
5: end for
6: return true
```

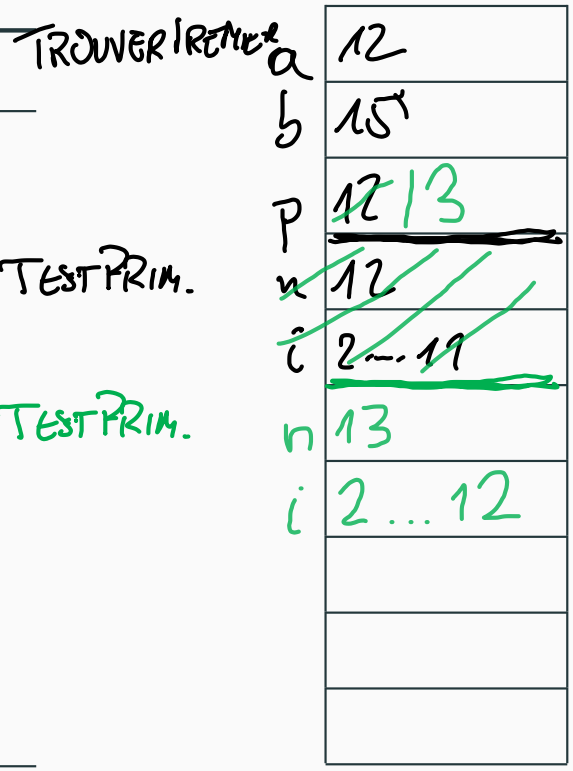
Exercice : Trouver un nombre premier

Algorithme TrouverPremier

Données: des entiers $0 < a < b$

Résultat: un premier $a \leq p < b$

```
1: for a ≤ p < b do
2:     if TestPrimalite(p) then
3:         return p
4:     end if
5: end for
6: return 0
```



Récursivité

Peut-on écrire une fonction qui s'appelle elle-même ?

Récursivité :

- une fonction est récursive si elle contient des appels à elle-même
- généralement, avec des paramètres différents
- chaque appel contient son un espace de mémoire indépendant

Deux exemples :

- Factorielle
- Plus grand commun diviseur

Factorielle

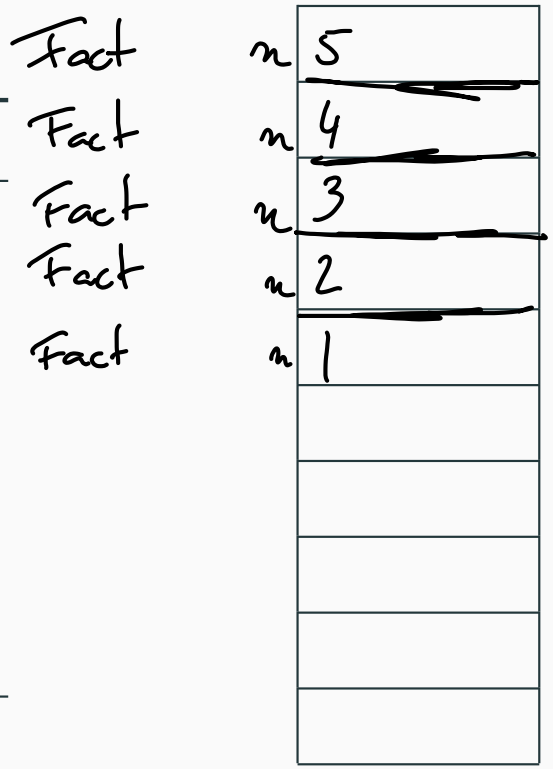
$n! = n \times (n-1) \cdot (n-2) \times \cdots \times 1.$

Algorithme Factorielle

Données: un entier $n > 0$

Résultat: la factorielle de n

```
1: if n = 1 then
2:     return 1
3: end if
4: return n · FACTORIELLE(n − 1)
```



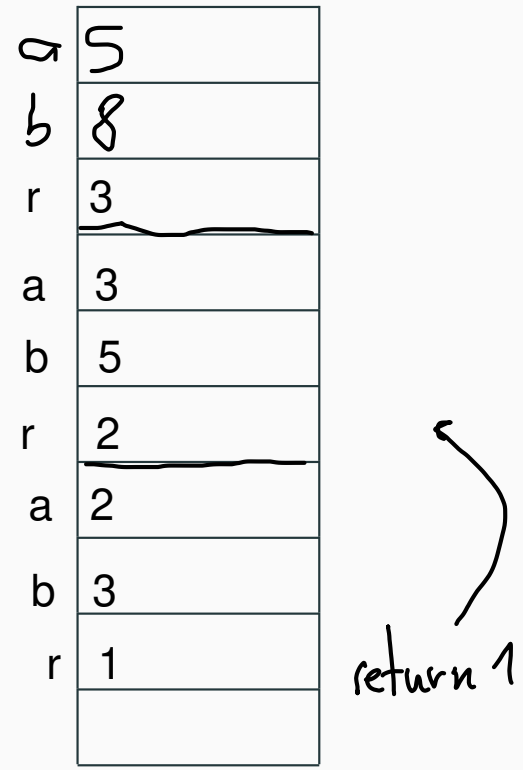
Plus grand commun diviseur

Algorithme PGCD

Données: deux entiers $0 \leq a < b$

Résultat: le plus grand commun diviseur de a et b

```
1: if a = 0 then
2:     return b
3: else
4:     r ← b%a
5:     return PGCD(r, a)
6: end if
```



Plus grand commun diviseur

PGCD

```
while a!= 0
    r = b%a
    b = a
    a = r
end while
```

Introduction aux Algorithmes

Conclusion

Conclusion

Algorithme :

- séquence d'instructions bien définies
- généralement en pseudo-code
- avec des instructions spécifiques ($\leftarrow$, if, for, return)

Exécution :

- CPU exécute le code, charge et modifie valeurs de la mémoire

Récursivité :

- Une fonction peut s'appeler à elle-même
- Chaque appel a son espace de mémoire séparé

Complexité des algorithmes

Problème de tri

On veut trier une liste :

- étant donné une liste d'entiers $(a_1, a_2, \ldots, a_n)$
- les placer par ordre croissant

$$a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_n}$$

Exemple :

$$(31, 41, 59, 26, 41, 58) \quad \rightarrow \quad (26, 31, 41, 41, 58, 59)$$

Il existe différents algorithmes de tris et le choix du meilleur algorithme dépend de plusieurs facteurs, entre eux :

- nombre d'éléments de la séquence
- valeurs pré-triées
- support de stockage

Analyse de la complexité d'un algorithme

Analyse de la complexité :

- L'étude de la quantité de ressources nécessaire à l'exécution.
- Cela implique définir une fonction qui met en rapport la taille des données avec :
  - le temps d'exécution (complexité de temps)
  - la quantité d'espace en mémoire (complexité d'espace).
- Un algorithme est *efficace* quand cette fonction croît lentement.

Complexité de temps ou d'espace ?

- La complexité temporelle est le facteur limitant
  - Il n'y a pas d'algorithmes rapides qui utilisent trop d'espace
- On développe d'abord des algorithmes rapides,
  - puis des algorithmes économes en espace

Quelles données utiliser ?

Différentes données de même taille produisent des comportements différents :

Il existe différentes façons d'analyser la complexité :

- pire cas : limite supérieure du temps d'exécution
  (garantit que l'algorithme ne prendra jamais plus)
- cas moyen : espérance du temps d'exécution
  (nécessite une supposition sur la distribution des données)
- meilleur cas : le nombre minimal d'étapes
  (peu informative).

Hypothèses simplificatrices

Tous les détails de l'exécution ne sont pas pris en compte

- Accès mémoire en temps constant
  - pas de mémoire cache, pas de mémoire virtuelle
- Opérations sur types simples en temps constant
  - Opérations arithmétiques $(+, -, \times, /)$
- Pas d'exécution parallèle (dans ce cours)
- Temps d'exécution simplifié : analyse asymptotique
  - On ne garde que le terme dominant d'une expression
  - $50n^2 + 7n + 3\log(n) \approx n^2$

Dans ce cours

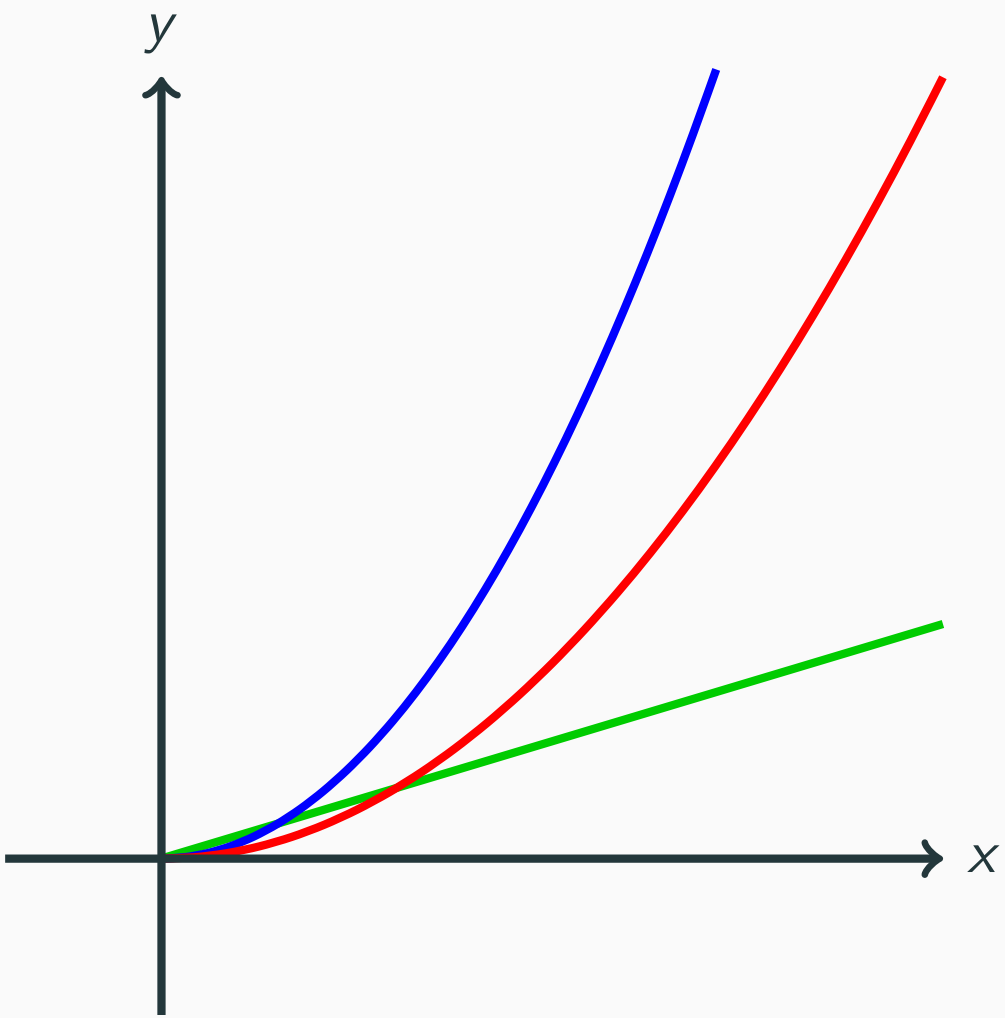
On va s'intéresser dans ce cours à

- la complexité de temps
- dans le pire cas
- analysée asymptotiquement

Complexité des algorithmes

Analyse asymptotique

Analyse asymptotique



$300x + 200$

$2x^2 + 3x$

x^2

$x \in [0, 1000]$

Analyse asymptotique

- Utilisé pour les données de grande taille
- Décrit le comportement limite de la fonction
- Exemple : $f(n) = 2n^2 + 3n$
  - Pour n très grand ($n \to +\infty$), similaire à n^2
  - $3n$ devient insignifiant par rapport à n^2
- Ce qui nous intéresse, c'est le taux de croissance
  - On dit que $f(n) \in O(n^2)$

Ordre de grandeur, comparatif (Source : Wikipedia)

Temps	Type de complexité	Temps pour n = 1	Temps pour n = 5	Temps pour n = 10	Temps pour n = 20	Temps pour n = 50	Temps pour n = 250	Temps pour n = 1 000	Temps pour n = 10 000	Temps pour n = 1 000 000	Problème exemple
$O(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	accès à une cellule de tableau
$O(\log(n))$	complexité logarithmique	10 ns	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns	recherche dichotomique
$O(\sqrt{n})$	complexité racinaire	10 ns	22 ns	32 ns	45 ns	71 ns	158 ns	316 ns	1 µs	10 µs	test de primalité naïf
$O(n)$	complexité linéaire	10 ns	50 ns	100 ns	200 ns	500 ns	2.5 µs	10 µs	100 µs	10 ms	parcours de liste
$O(n\ log^*(n))$	complexité quasi linéaire	10 ns	50 ns	100 ns	200 ns	501 ns	2.5 µs	10 µs	100,5 µs	10,05 ms	triangulation de Delaunay
$O(n\log(n))$	complexité linéarithmique	10 ns	40 ns	100 ns	260 ns	850 ns	6 µs	30 µs	400 µs	60 ms	tris par comparaisons optimaux (comme le tri fusion ou le tri par tas)
$O(n^2)$	complexité quadratique (polynomiale)	100ns	250 ns	1 µs	4 µs	25 µs	625 µs	10 ms	1 s	2.8 heures	parcours de tableaux 2D
$O(n^3)$	complexité cubique (polynomiale)	1 µs	1.25 µs	10 µs	80 µs	1.25 ms	156 ms	10 s	2.7 heures	316 ans	multiplication matricielle naïve

Notation asymptotique

La notation asymptotique (ou Bachmann–Landau) décrit le comportement asymptotique d'une fonction, lorsque l'argument tend vers une valeur particulière ou vers l'infini.

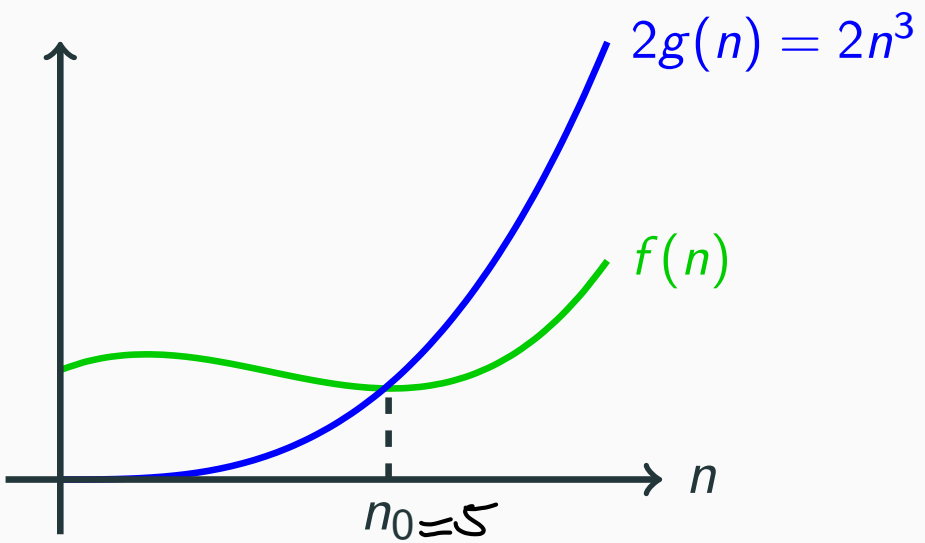
- $O(\cdot)$ ("grand O") : limite supérieure, $f \in O(n^2)$ "au plus"
- $\Omega(\cdot)$ ("grand omega") : limite inférieure, $f \in \Omega(n^2)$ "au moins"
- $\Theta(\cdot)$ ("grand theta") : ordre de grandeur asymptotique. $f \in \Theta(n^2)$ croissance comme n^2

Définition de O

Soient f et g deux fonctions à variable réelle n.

On dit que $f(n) \in O(g(n))$, lorsque

$$\exists\, c > 0, n_0 : \forall\, n > n_0, \quad |f(n)| \leq c|g(n)|$$



$g(n) = n^3$
$\downarrow$
$f \in O(n^3)$

Intuitivement, cela signifie que f ne croît pas plus vite que g.

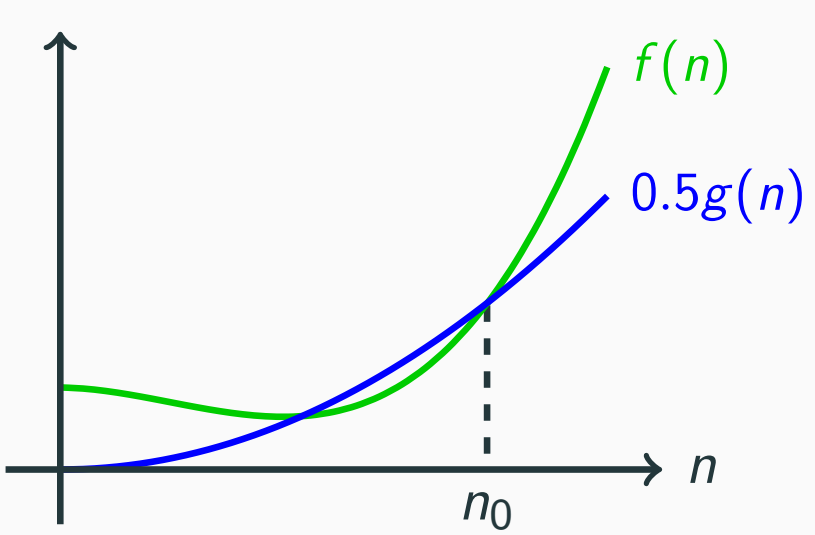
On dit que g est une borne supérieur de f.

Définition de Ω

Soient f et g deux fonctions à variable réelle n.

On dit que $f(n) \in \Omega(g(n))$, lorsque

$$\exists\, c > 0, n_0 : \forall\, n > n_0, \quad |f(n)| \geq c|g(n)|$$



$g(n) = n^2$

Intuitivement, cela signifie que f ne croît pas plus lent que g.

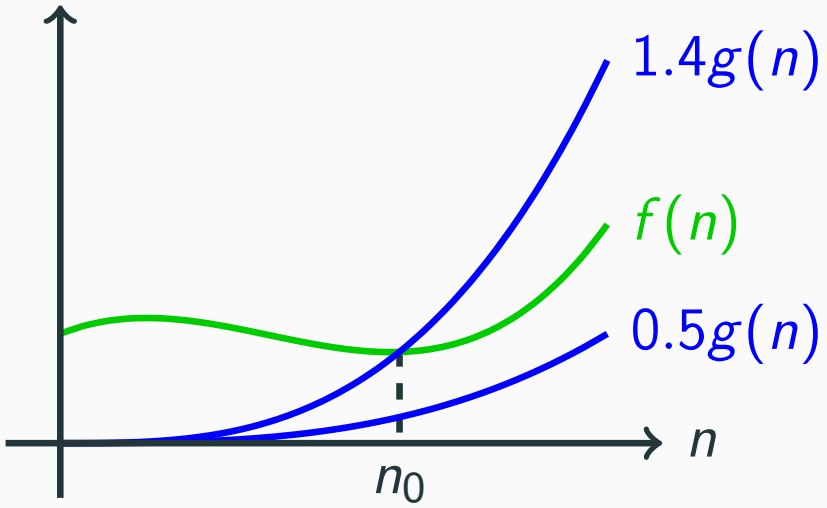
On dit que g est une borne inférieur de f.

Définition de Θ

Soient f et g deux fonctions à variable réelle n.

On dit que $f(n) \in \Theta(g(n))$, lorsque

$$\exists\, c_1, c_2 > 0, n_0 : \forall\, n > n_0, \quad c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$$



Intuitivement, cela signifie que f et g ont une croissance similaire.

On dit que g est une borne serrée de f (*tight bound*).

Exemples

$3n+4 \leq 4n$, $n \geq 100$

$3n+4 \approx n \in O(n)$
$\in \Omega(n)$
$\in \Theta(n)$

1. $f_1(n) = 3n + 4$
2. $f_2(n) = 2n^2 + n + 3$
3. $f_3(n) = n + \log n$
4. $f_4(n) = (n+2)\log n$

$f_3, f_1 \in$ a. $O(n)$?
$f_3, f_1 \in$ b. $O(n \log n)$?
$f_3, f_2, f_1 \in$ c. $O(n^2)$?
$f_2 \in$ d. $\Theta(n^2)$?
$f_3, f_2, f_1 \in$ e. $\Omega(n)$?
$f_2 \in$ f. $\Omega(n^2)$?

$n \cdot \log n$
$\| \quad \wedge$
$n \cdot n$

$f_2(n) = 2n^2 + n + 3 \approx n^2 \in \Theta(n^2) \rightarrow O(n^2), \Omega(n^2)$

$f_3(n) = n + \log n \in \Theta(n)$

$f_4(n) = (n+2)\log n = n\log n + 2\log n$