Algorithmique E3

Introduction aux algorithmes

Daniel Vaz
daniel.ramosvaz@esiee.fr

2026-11-16

Objectifs et déroulement

Objectifs principaux du cours

Développer la logique et le raisonnement algorithmique

- 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 ; algorithmes naifs
- **C3** : Algorithmes itératifs
- **C4** : Récursivité (en ligne)
- **C5** : Structures de données
- **C6** : Révision ; arbres et tas

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. Unitaires : structure du code et tests fournis
    - Vous n'avez qu'à écrire quelques fonctions et lancer les tests
3. Testés : Tests fournis, structure libre
    - Plus difficiles que d'autres dans les fiches TP

Le code doit toujours être testé ! ! !

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 – Unitaire

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

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

Exercices TP – Testé

```c
#include <stdio.h>
// ...
int main() {
  int n;
  scanf("%d", &n);

  for (int i = 2; i < n; i++) {
    if (isprime(i))
      printf("%d\n");
  }
}
```

```
$ runtests.py
Tests de la liste des premiers : 10 / 10 reussis
```

Intervenants

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

TD / TP :

- Mehdi ACHEMAOUI
- Mourad BEN HADJ
- Ouided BRAOUI
- Clément CHENEVIÈRE
- Mohamed FIDMA
- Ahmed TABIA
- Daniel VAZ
- Teng WU

Évaluation

- Examen de 2 heures le 19/01/2027
- Les TD / TP ne sont pas notés
  - Mais il est fortement conseillé de faire tous les exercices
  - Un bonus de 0,3/20 est possible pour vos travaux TP
- L'assiduité sera prise en compte lors du jury de semestre.

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

Git et Bonus TP

Il est recommandé de sauvegarder votre travail dans un repo Git

- Guide disponible sur Blackboard
- Demandez de l'aide auprès de votre chargé de TP ou par mail

Vous pouvez obtenir un bonus de 0,3/20 pour vos efforts en TP :

- Vérification en fin ~~d'année~~ du semestre, lors de la revue d'examen
- Il faut comprendre parfaitement tout votre travail
- Une seule ligne non comprise : bonus = 0
- Si le code est entièrement le vôtre, cela devrait être facile

Aperçu

Objectifs et déroulement

Introduction aux Algorithmes

- Pseudo-Code
- Écrire des algorithmes
- Exécution dans un ordinateur
- Fonctions

Complexité des algorithmes

- Analyse asymptotique

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

Exemple : Somme des éléments d'un tableau

Algorithme SommeListe

Données: tableau t de taille n d'entiers

Résultat: somme des éléments du tableau t

s prend la valeur 0

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

Exemple : Somme des éléments d'un tableau

Précis : l'algorithme est défini en langage mathématique

Fini : il suffit qu'il n'y ait pas de boucle infinie

Données et résultat : indiqués dans l'en-tête

Introduction aux Algorithmes

Pseudo-Code

Exemple : Somme des éléments d'un tableau

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

Pseudo-code

Une façon de spécifier des algorithmes en langage mathématique :

- Une sorte de programmation indépendante du langage
- Spécifie généralement les entrées et les sorties
- Pour les humains, mais clair et sans ambiguïté

Exécution de pseudo-code : comme en programmation :

- Partir de la ligne 1 et progresser linéairement
- Conditions : sauter à l'instruction correspondante
- Boucles : répéter les instructions
- Appel de fonction : sauter aux instructions, puis revenir

Exemple : Somme des éléments d'un tableau

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

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

Return : termine l'exécution avec le résultat donné

return $\langle \texttt{expression} \rangle$

Boucle for : répète l'exécution des instructions pour chaque valeur

for $a \leq i < b$ **do** *// par défaut dans l'ordre croissant*

  $\langle \texttt{instructions} \rangle$ *// exécute pour $i = a$, puis $i = a + 1$, ...*

end for

for $i = a, a + 1, a + 2, \ldots, b - 1$ **do**

  $\langle \texttt{instructions} \rangle$

end for

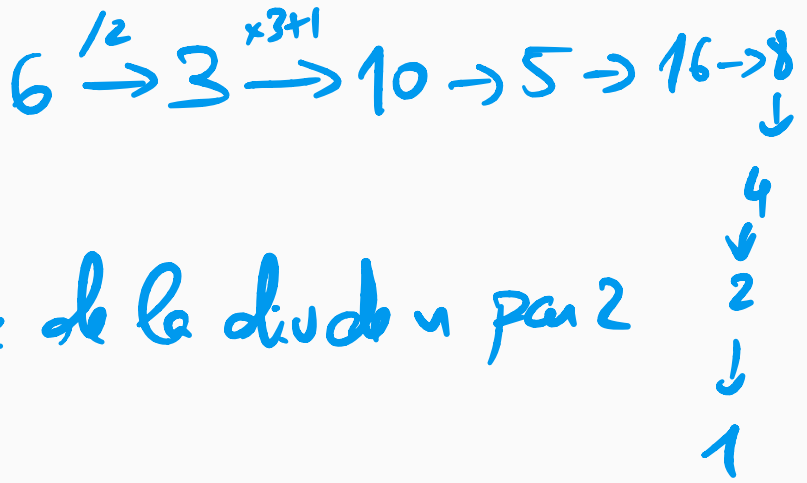
Exemple : Suite de Syracuse

Algorithme CompteSuiteSyracuse

Données: un entier n

Résultat: entier p

```
1:  p ← 0
2:  while n ≠ 1 do
3:      if n%2 = 0 then
4:          n ← n/2
5:      else
6:          n ← 3n + 1
7:      end if
8:      p ← p + 1
9:  end while
10: return p
```



Exemple : Suite de Syracuse

Algorithme CompteSuiteSyracuse

Données: un entier n

Résultat: nombre de pas p de la suite de Syracuse pour arriver à 1

```
1:  p ← 0
2:  while n ≠ 1 do
3:      if n%2 = 0 then
4:          n ← n/2
5:      else
6:          n ← 3n + 1
7:      end if
8:      p ← p + 1
9:  end while
10: return p
```

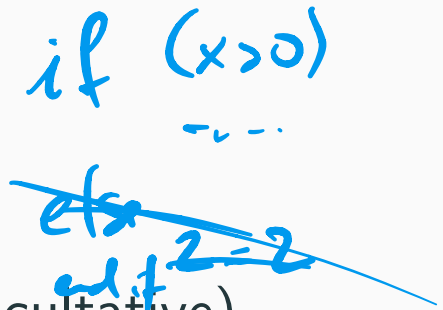
Instructions (2)

Boucle while : répète instructions tant que la condition est vraie

while ⟨`condition`⟩ **do**
    ⟨`instructions`⟩
end while

Condition : exécute les instructions si la condition est vraie

if ⟨`condition`⟩ **then**
    ⟨`instructions`⟩
end if



- **else if** ⟨`condition`⟩ : ajoute une alternative (facultative)
  - uniquement si toutes les conditions précédentes sont fausses
- **else** : si toutes les conditions précédentes sont fausses

facultative

Notation

Opérations :

	Pseudo-code	Exemple	C
et	$\wedge$	$x > 1 \wedge x < 3$	`x > 1 && x < 3`
ou	$\vee$	$x < 0 \vee x > 2$	`x < 0 || x > 2`
non	$\neg$	$\neg(x < 0)$	`!(x < 0)`
égal	$=$	$x = 0$	`x == 0`
affecter	$\leftarrow$	$x \leftarrow 0$	`x = 0`
reste div.	%	$x\%3$	`x % 3`
div. entière	//	$x//3$	`x / 3`

Spécification d'intervalles :

- **for** $0 \leq i < n$ **do**
- `tableau`$[a : b]$: éléments de indice a à $b - 1$

Introduction aux Algorithmes

Écrire des algorithmes

Exercice : Test de primalité

Problème : vérifier si un entier n donné est un nombre premier

deux diviseurs : 1 et n

Exercice : Test de primalité

Problème : vérifier si un entier n donné est un nombre premier

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
```

Attention aux returns effectués trop tôt ! ! !

Exercice 2 : Compter dans un Tableau

Problème : Compter les occurrences d'un élément dans un tableau

- On utilise une variable pour compter les occurrences

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

- En utilisant return direct ou variable drapeau

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

- En utilisant return direct ou variable drapeau

Algorithme Recherche

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

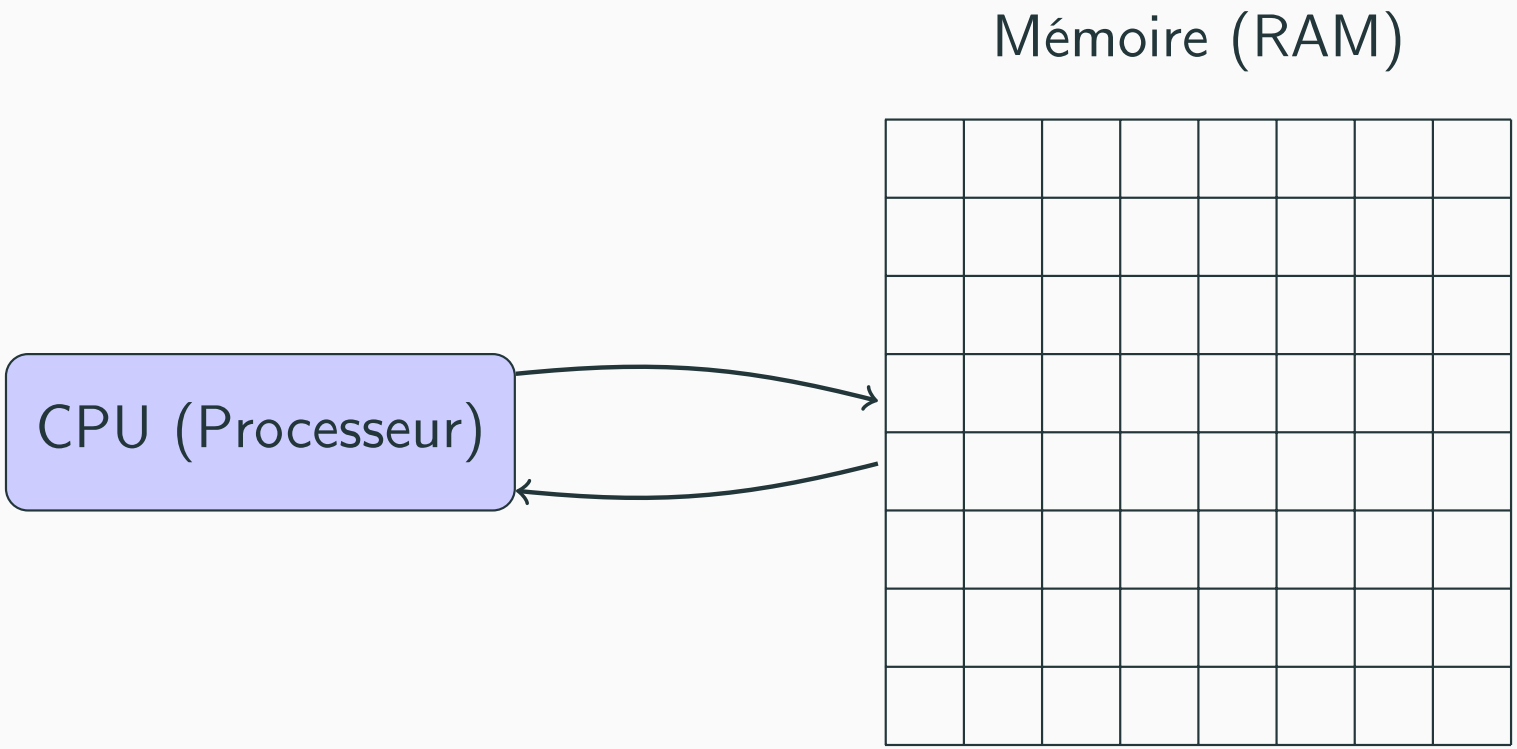
Résultat: **true** si x est dans t, **false** sinon

```
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



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

- Chaque variable dispose de son propre espace dans la mémoire
- Cet espace est alloué (réservé) lors de la création de la variable
- Le processeur connaît l'adresse (position) de chaque variable

Exemple : Addition de deux nombres

1: $a \leftarrow 5$
2: $b \leftarrow 10$
3: $c \leftarrow a + b$

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

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
  - Cet espace s'appelle *cadre de pile*
- les paramètres sont copiés dans le cadre
- les variables créées dans la fonction sont stockées dans le cadre
- le cadre est libéré (disparaît) lorsque la fonction se termine

Fonctions et algorithmes

L'algorithme :

- La méthode ou idée pour résoudre un problème
- Peut être représenté en pseudo-code
- Implémenté par une (ou plusieurs) fonctions

La fonction :

- Unité de code réutilisable qui exécute une tache
- Sa idée peut correspondre à un algorithme

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 $a \leq p < b$

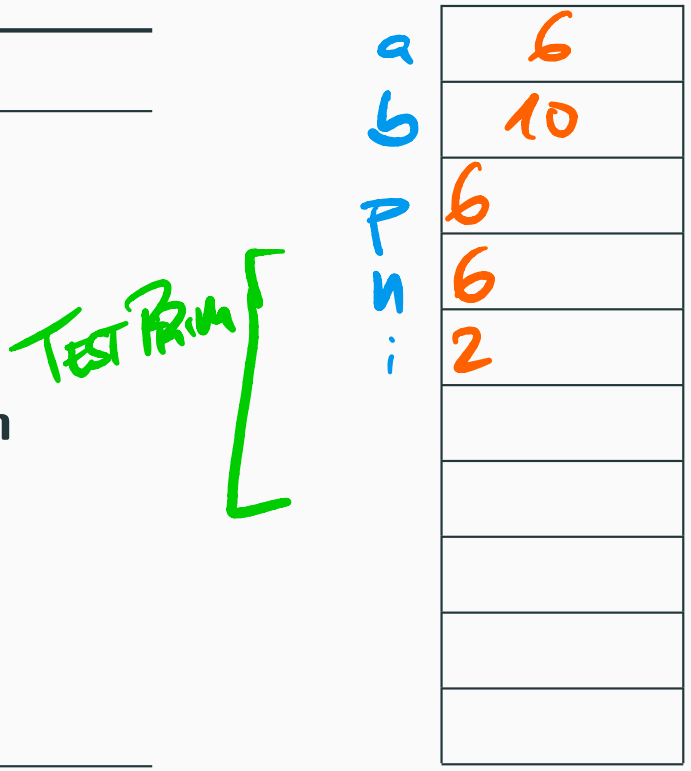
Exercice : Trouver un nombre premier

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
```



Exercice : Trouver un nombre premier

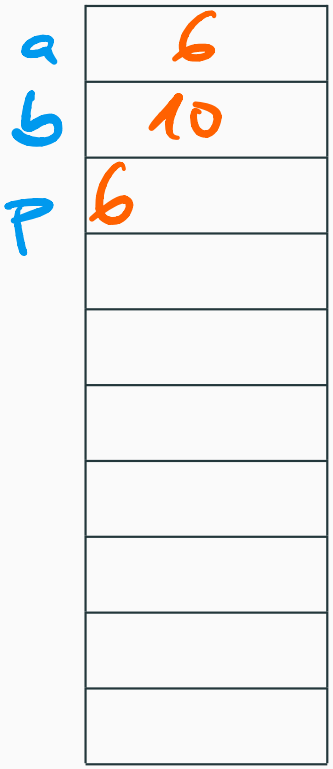
Algorithme TrouverPremier

Données: des entiers $0 < a < b$

Résultat: un premier $a \leq p < b$

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

- false



Exercice : Trouver un nombre premier

Algorithme TrouverPremier

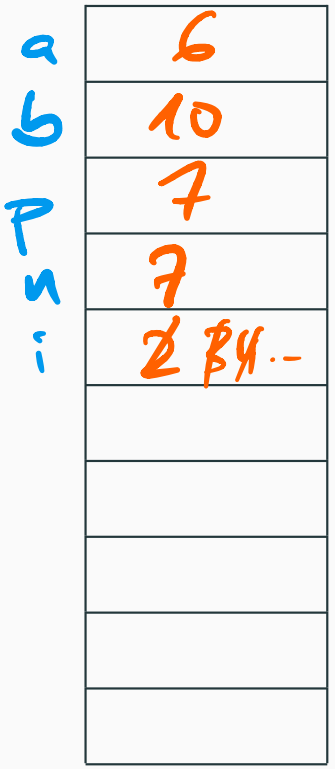
Données: des entiers $0 < a < b$

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
```

„true

TestPrim[



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
- Chaque fonction a son cadre dans la mémoire
    - Qui stocke les paramètres et variables

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.
- 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 si cette fonction croît lentement.
  - Petit polynôme de la taille de l'entrée

TESTPRIMALITE (n)

↳ $\approx n$

↳ $\approx n^2$

↳ $\approx n^3$

↳ $\approx 2^n$

Analyse de la complexité d'un algorithme

Comment analyser la complexité :

1. Complexité de temps ou d'espace ?
2. Quelles données utiliser ?
3. Hypothèses simplificatrices

Complexité de temps ou d'espace ?

- La complexité temporelle est le facteur limitant
  - Un algorithme efficace n'a pas le temps d'utiliser trop d'espace
- On développe d'abord des algorithmes rapides,
  - puis des algorithmes économes en espace

Quelles données utiliser ?

Différentes données 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



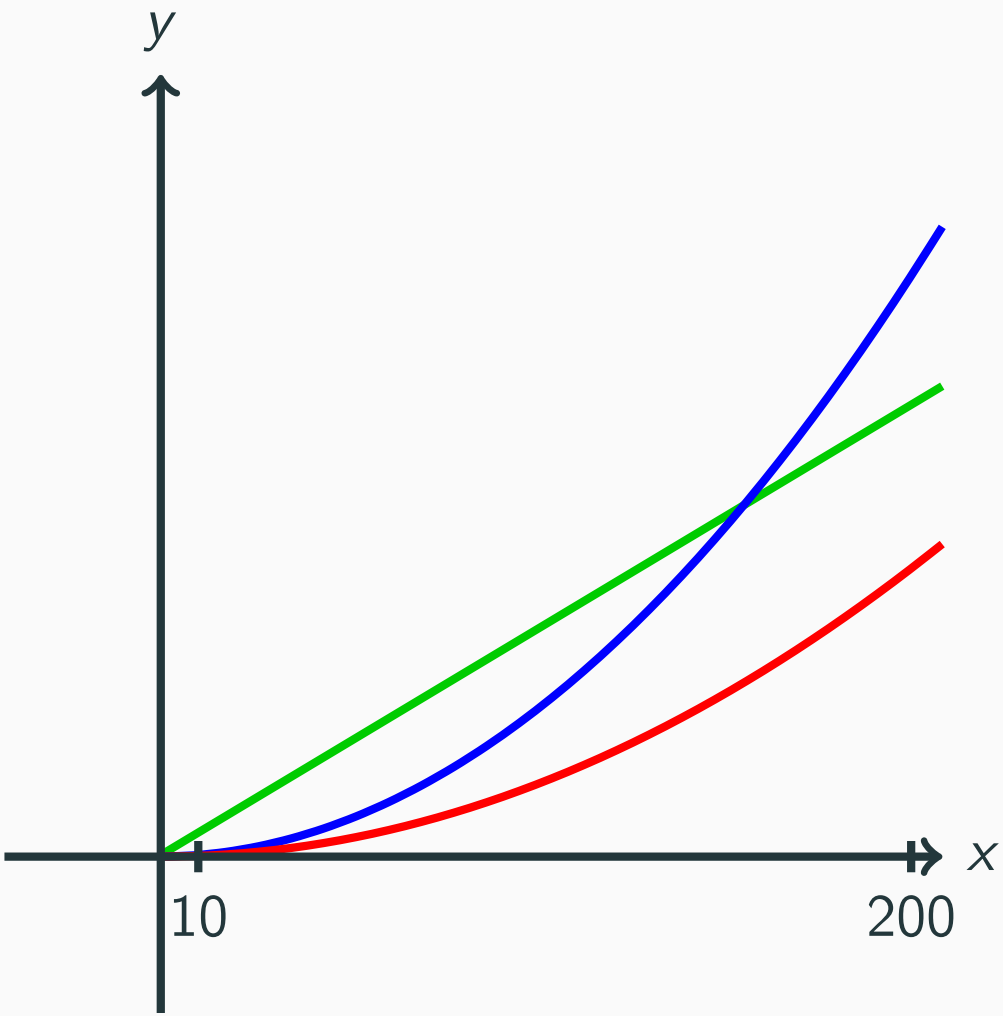
$300x + 200$

$2x^2 + 3x$

x^2

$x \in [0, 10]$

Analyse asymptotique



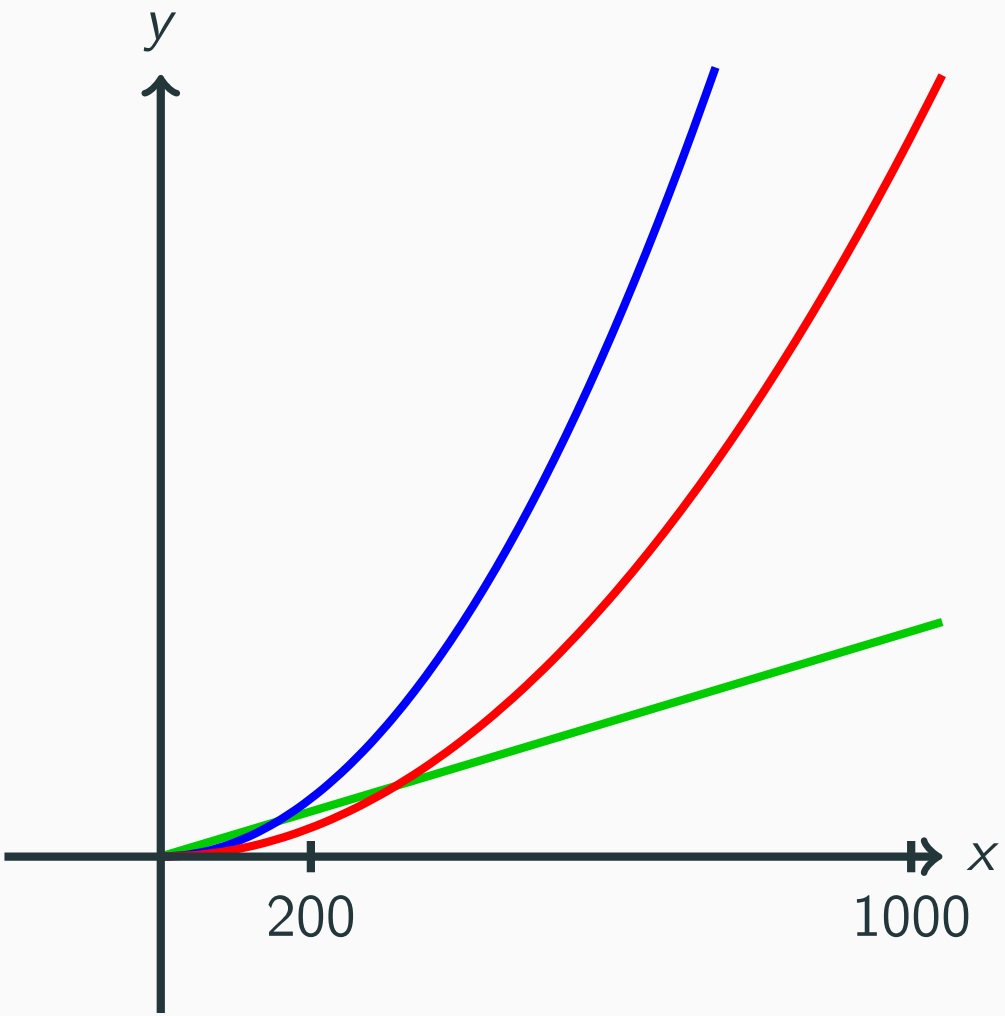
$300x + 200$

$2x^2 + 3x$

x^2

$x \in [0, 200]$

Analyse asymptotique



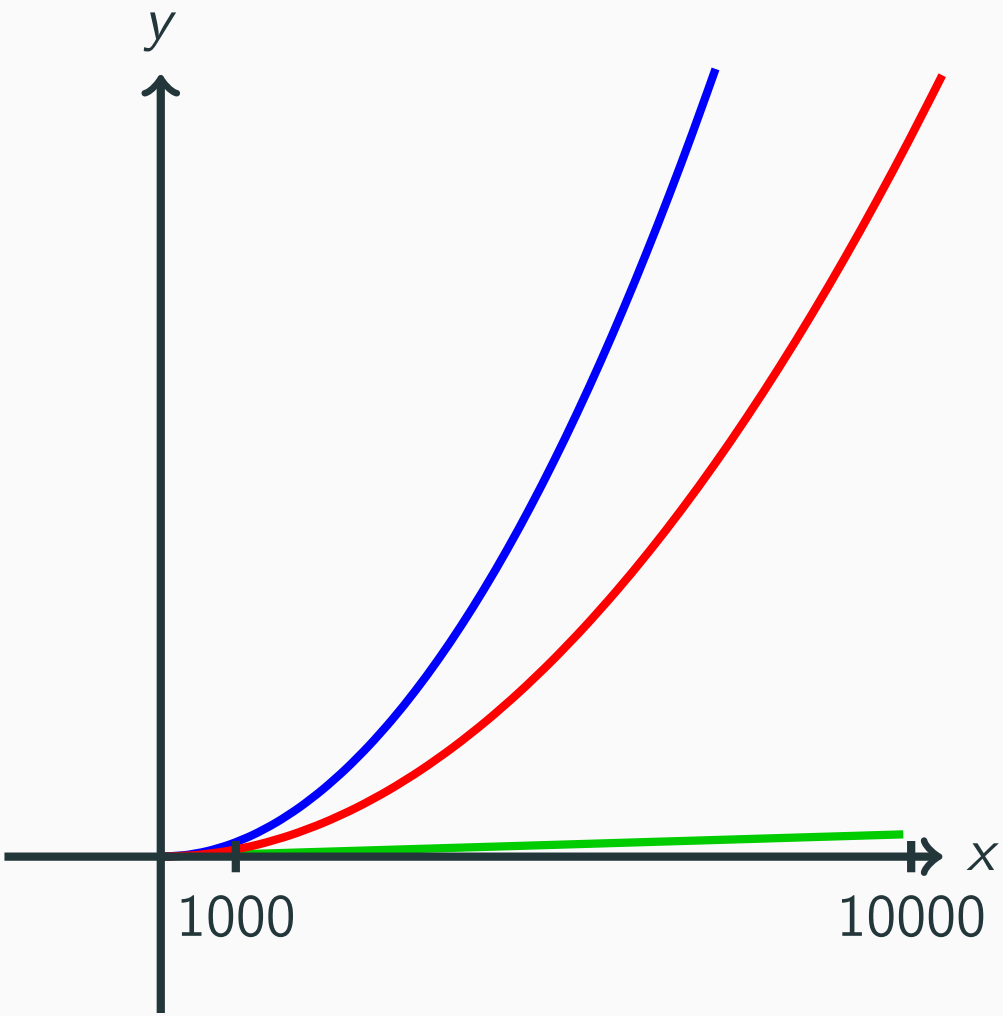
$300x + 200$

$2x^2 + 3x$

x^2

$x \in [0, 1000]$

Analyse asymptotique



$300x + 200$

$2x^2 + 3x$

x^2

$x \in [0, 10000]$

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 $f(n) \in O(n^2)$

↳ grand-O de 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,
- $\Omega(\cdot)$ ("grand omega") : limite inférieure,
- $\Theta(\cdot)$ ("grand theta") : ordre de grandeur asymptotique.

"(inférieur ou égal)"

"(supérieur ou égal)"

"égal"

Définition de O

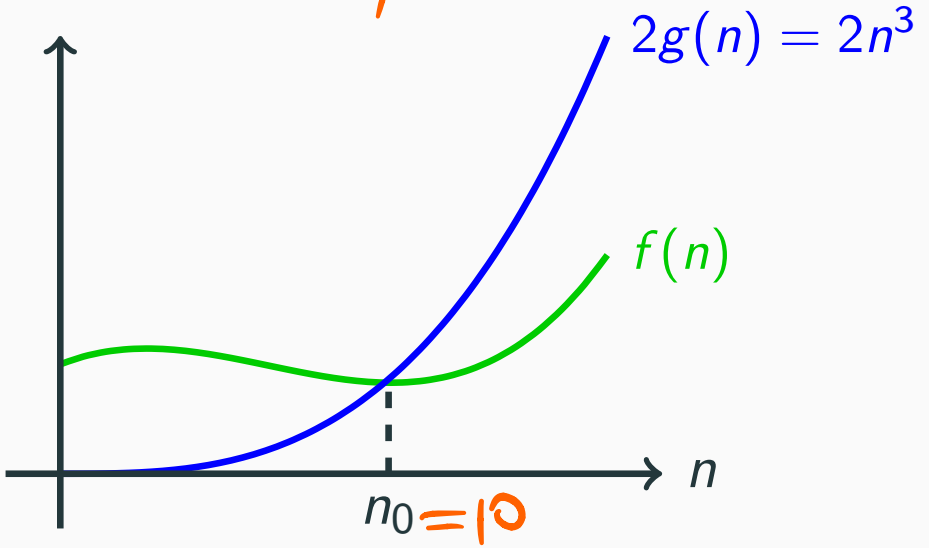
On dit que $f \in O(g)$ si f ne croît pas plus vite que g.

- $2n \in O(n)$, $n^2 \in O(n^3)$
- g est une borne supérieur de f

Formellement : Soient f et g deux fonctions à variable réelle n.

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

existe c , n_0



$f \in O(n^3)$?

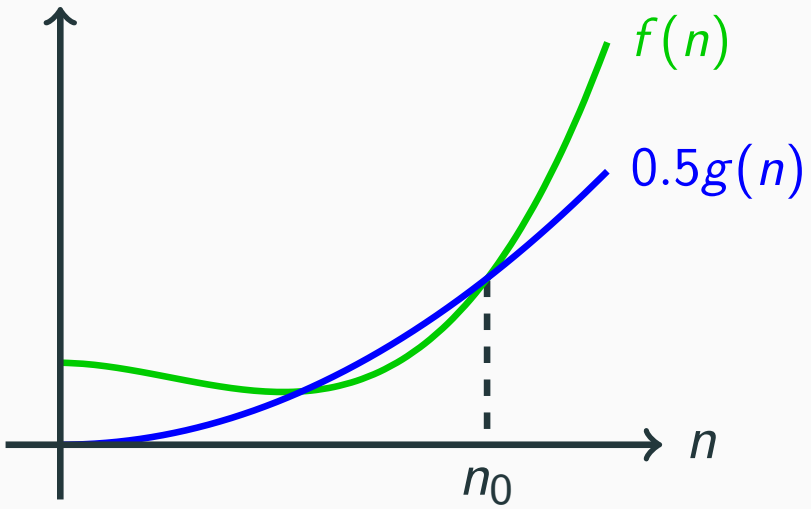
Définition de Ω

On dit que $f \in \Omega(g)$ si f ne croît pas plus lent que g.

- $2n \in \Omega(n)$, $n^2 \in \Omega(n)$
- g est une borne inférieur de f

Formellement : Soient f et g deux fonctions à variable réelle n.

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



Définition de Θ

On dit que $f \in \Theta(g)$ si f et g ont une croissance similaire.

- $2n \in \Theta(n)$, $n^2 \in \Theta(n^2)$
- g est une borne serrée de f

$n \notin \Theta(n^2)$, $n^2 \notin \Theta(n)$

Définition de Θ

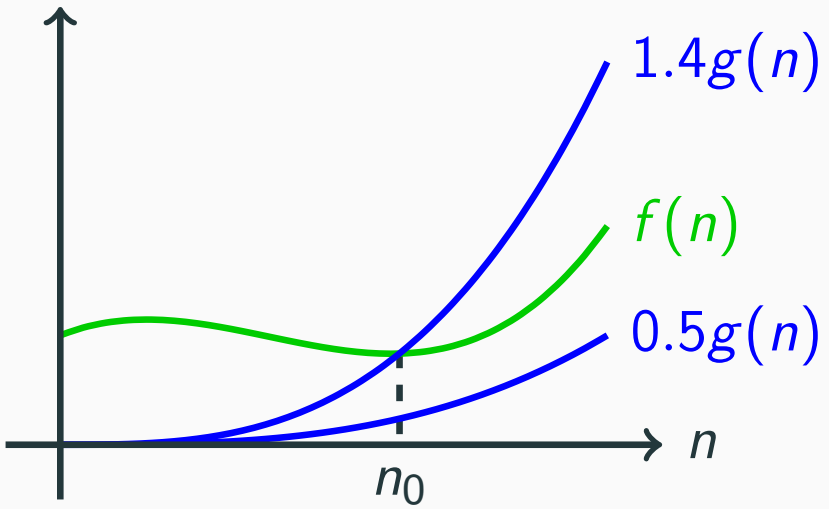
On dit que $f \in \Theta(g)$ si f et g ont une croissance similaire.

- $2n \in \Theta(n)$, $n^2 \in \Theta(n^2)$
- g est une borne serrée de f

$\Theta(f)$: complexité de f.
$f \in \Theta(n^2) \rightarrow f \notin \Theta(n)$

Formellement : Soient f et g deux fonctions à variable réelle n.

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



Exemples

1. $f_1(n) = 3n + 4$ $\Theta(n)$

a. $O(n)$? ✓ $3n+4 \leq 4n$ $n \geq 4$

b. $O(n \log n)$? ✓ $3n+4 \leq 4n \cdot \log n$ $n \geq 4$

c. $O(n^2)$? ✓ $3n+4 \leq 4n^2$ $n \geq 4$

d. $\Theta(n^2)$? ✗

e. $\Theta(n)$? ✓ $3n \leq f_1(n) \leq 4n$ $n \geq 4$

f. $\Omega(n^2)$? ✗

Exemples

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

a. $O(n)$?
b. $O(n \log n)$?
c. $O(n^2)$?
d. $\Theta(n^2)$?
e. $\Theta(n)$?
f. $\Omega(n^2)$?

Exemples

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

a. $O(n)$?
b. $O(n \log n)$?
c. $O(n^2)$?
d. $\Theta(n^2)$?
e. $\Theta(n)$?
f. $\Omega(n^2)$?

Exemples

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

a. $O(n)$?
b. $O(n \log n)$?
c. $O(n^2)$?
d. $\Theta(n^2)$?
e. $\Theta(n)$?
f. $\Omega(n^2)$?