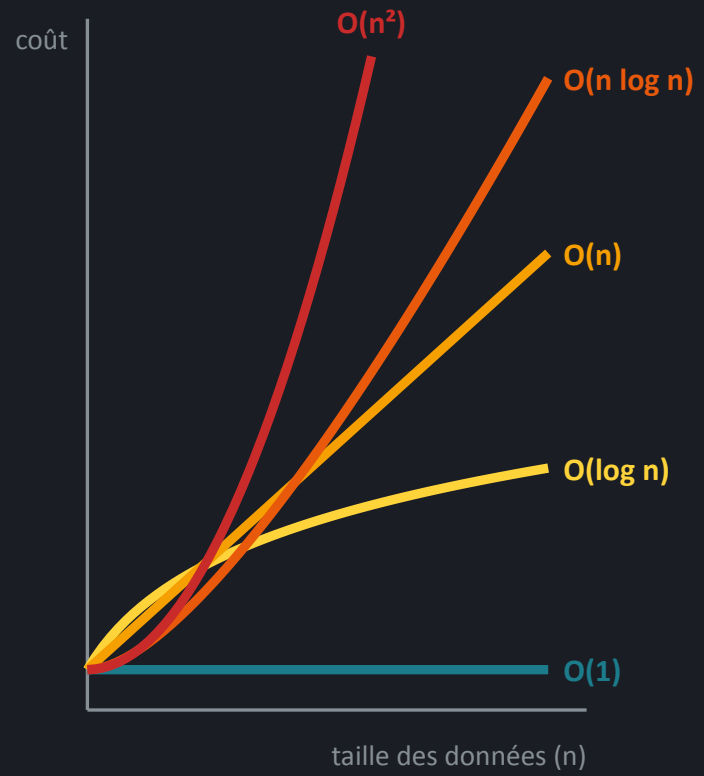
RESSOURCE R5.A.04

Qualité Algorithmique

Évaluer, mesurer et améliorer la qualité d'un code



Maxime Menault · Année universitaire 2026–2027

xfab UPEC UNIVERSITÉ PARIS-EST CRÉTEIL IUT SENART FONTAINEBLEAU

La ressource R5.A.04

Trois compétences visées à la fin du module



ÉVALUER

Savoir évaluer la qualité algorithmique d'un code

Repérer ce qui rend un code lent, gourmand ou difficile à maintenir.



OUTILLER

Savoir utiliser des outils d'analyse de code

Profilers, analyseurs, outils de revue : les choisir et interpréter leurs résultats.



MESURER

Savoir se servir de métriques

Temps, mémoire, complexité : objectiver plutôt que deviner.

Organisation du cours

Une séance type



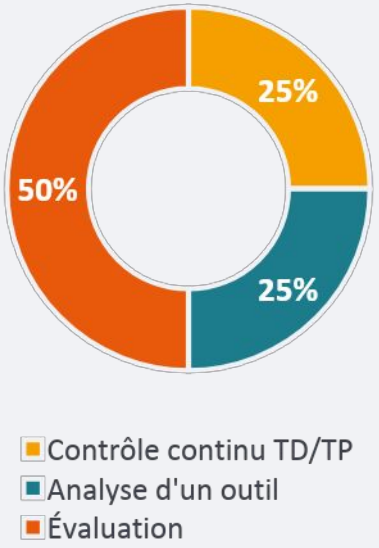
4 heures au total

Sessions de 4 heures
Cours magistral puis mise en pratique

Pause de 15 minutes
Entre le CM et le TD / TP

TD et / ou TP
Mise en pratique immédiate du CM

Notation



Les quatre parties du cours

Aujourd'hui

1 — 2 — 3 — 4

Profiling de code

Mesurer le comportement réel d'un programme à l'exécution

Complexité cyclomatique

Mesurer la complexité du flot de contrôle d'une fonction

Complexité algorithmique

Estimer le coût d'un algorithme selon la taille des données

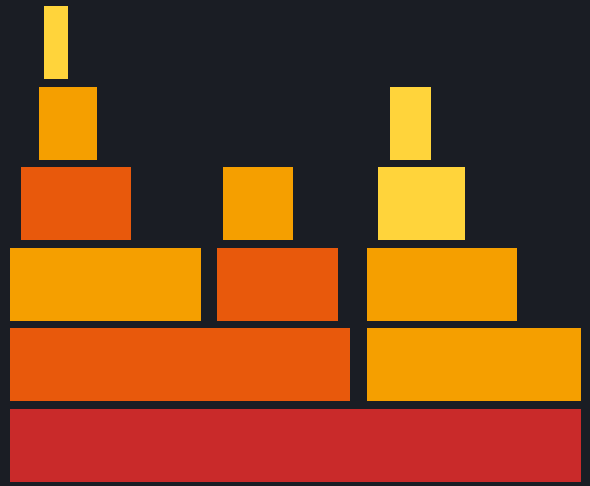
Revue de code

Évaluer la qualité d'un code à plusieurs regards

01

Profiling de code

Observer un programme pendant qu'il s'exécute



Bibliographie

CppCon 2021

The Basics of Profiling
Mathieu Ropert · Tech Lead Hearts of Iron IV, Paradox Development Studio

KDAB

Profiling & Debugging
David Faure · Senior Software Developer, KDAB France

linux.conf.au 2017

Python Profiling and Performance Tuning in Production
Joe Gordon · Python Tech Lead, Pinterest

Strange Loop 2019

Performance Matters
Emery Berger · Researcher, Microsoft & Professor, University of Massachusetts Amherst

LISA13

Blazing Performance with Flame Graphs
Brendan Gregg · Performance Engineer (OpenAI, Intel, Netflix, Oracle, Sun)

Qu'est-ce que le profiling ?

Profiler, c'est mesurer un programme pendant son exécution, sur un scénario donné.

- Analyse dynamique d'un programme
- Mesure de données à l'exécution
- Mesures obtenues avec des outils dédiés : les profilers
- Résultat : un profil (rapport, graphe d'appels, flame graph…)
- Un profil n'est valable que pour les données et la machine utilisées



Analyse statique

On lit le code sans l'exécuter.
Ex. : linters, complexité cyclomatique



Analyse dynamique

On exécute le code sur des données réelles.
Ex. : profilers, debuggers, sanitizers

Que mesure-t-on ?

Temps

CPU : calcul effectif. Réel (wall-clock) : attentes incluses.

Appels de fonctions

Nombre d'appels, temps propre (self) vs cumulé (total).

Mémoire

Allocations, pic de consommation, fuites.

Matériel

Défauts de cache, branches mal prédites, instructions par cycle.

Entrées / sorties

Disque, réseau, base de données : souvent le vrai goulot.

Concurrence

Attente de verrous, threads inactifs, contention.

Deux grandes techniques

Instrumentation

Le code est modifié pour enregistrer chaque appel.

+ **Exact : nombre d'appels, chemins suivis**
– **Surcoût important**
– **Fausse le temps des petites fonctions**

Ex. : gprof (-pg), Callgrind, Tracy

Échantillonnage

Le programme est interrompu à intervalle régulier pour relever la pile d'appels.

+ **Faible surcoût, utilisable en production**
– **Statistique : rate les événements courts**

Ex. : perf, Intel VTune, py-spy

Effet de sonde : le profiler ajoute son propre coût et fausse la mesure.

gprof : appels comptés par instrumentation, temps estimé par échantillonnage.

Exemples de profilers

gprof

C / C++ CPU

Profil plat + graphe d'appels. Compiler avec -pg.

hprof

Java CPU / tas

Agent JVM historique, retiré au JDK 9 (⟶ JFR).

Intel VTune

C / C++, Python… CPU / matériel

Hotspots, threads, microarchitecture du CPU.

Heaptrack

C / C++ Mémoire

Trace chaque allocation : pics, fuites, temporaires.

Valgrind massif

C / C++ Mémoire

Évolution du tas dans le temps. Très lent.

Coz

C / C++ Causal

Quel gain global si cette ligne allait plus vite ?

Et aussi : perf, Callgrind + KCachegrind, py-spy, cProfile, Tracy, Superluminal…

gprof : comment ça marche ?

Code source → **`gcc -pg`** → **Exécution** → **`gmon.out`** → **`gprof`**

Code source	`gcc -pg`	Exécution	`gmon.out`	`gprof`
Votre programme C, sans modification.	Ajoute un appel à mcount à l'entrée de chaque fonction.	Compte les appels et relève l'instruction en cours 100 fois par seconde.	Écrit à la fin normale du programme.	Croise binaire et mesures : profil plat et graphe d'appels.



Exact

Le nombre d'appels et les relations appelant → appelé, grâce à l'instrumentation.



Estimé

Les temps : un échantillon vaut 0,01 s. Une fonction courte peut afficher 0,00 s.

Un exemple fil rouge

```c
int est_premier(int n) {
    if (n < 2) return 0;
    for (int d = 2; d < n; d++)
        if (n % d == 0) return 0;
    return 1;
}
int somme_chiffres(int n) {
    int s = 0;
    for (; n > 0; n /= 10) s += n % 10;
    return s;
}
int compter_premiers(int limite) {
    int total = 0;
    for (int i = 0; i < limite; i++)
        if (est_premier(i)) total++;
    return total;
}
int premiers_chanceux(int limite) {
    int total = 0;
    for (int i = 0; i < limite; i++)
        if (est_premier(somme_chiffres(i)) && est_premier(i))
            total++;
    return total;
}
```

Ce que fait le programme

`est_premier` : teste tous les diviseurs de 2 à n − 1.

`compter_premiers` : compte les premiers sous 60 000.

`premiers_chanceux` : premiers dont la somme des chiffres est aussi première.

```
gcc -O0 -g -pg exemple.c \
    -o exemple
./exemple
gprof -b ./exemple gmon.out
```

Lire le profil plat

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.49     0.49   136828     0.00     0.00  est_premier
  0.00      0.49     0.00    60000     0.00     0.00  somme_chiffres
  0.00      0.49     0.00        1     0.00   214.87  compter_premiers
  0.00      0.49     0.00        1     0.00   275.13  premiers_chanceux
```

`% time`	Part du temps total passée dans la fonction elle-même	**`cumulative`**	Somme des temps propres des lignes au-dessus
`self seconds`	**Temps propre : la colonne à regarder en premier**	**`self ms/call`**	Temps propre moyen par appel
`calls`	Nombre d'appels, compté exactement	**`total ms/call`**	Par appel, fonctions appelées comprises



`somme_chiffres` : 60 000 appels, 0,00 s. Aucun échantillon ne l'a touchée, elle n'est pas gratuite pour autant.

Lire le graphe d'appels

```
index % time    self  children    called     name
                0.00    0.28       1/1           main [2]
[3]     56.1    0.00    0.28       1         premiers_chanceux [3]
                0.28    0.00   76828/136828       est_premier [1]
                0.00    0.00   60000/60000        somme_chiffres [5]
```

← Appelant (parent)
← Fonction étudiée
← Appelées (enfants)

self

Temps propre de la fonction, ou de l'enfant sur sa ligne.

children

Temps passé dans les fonctions qu'elle appelle.

called

76828/136828 : appels depuis ce parent / appels totaux.

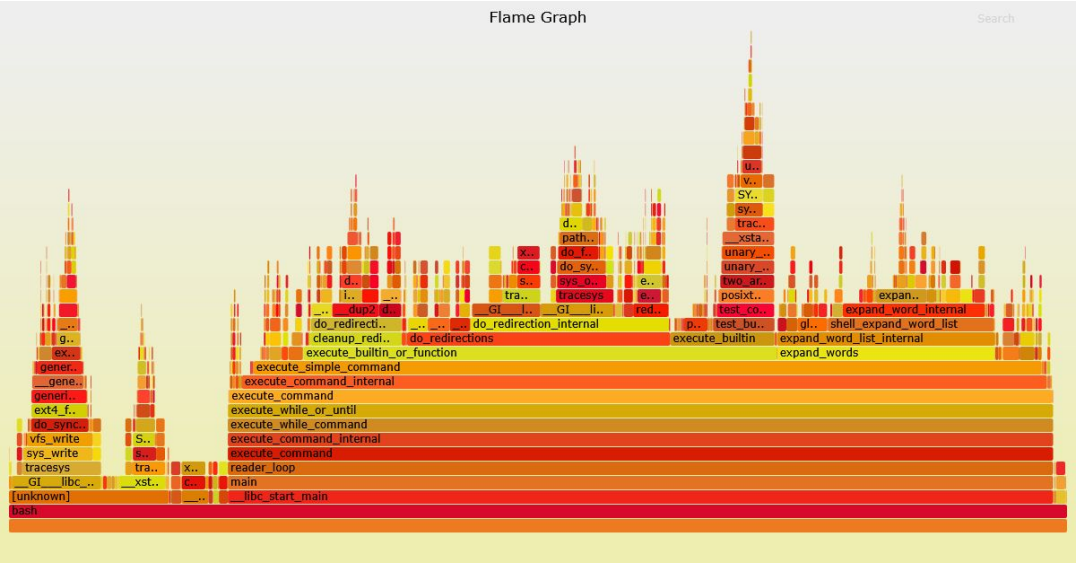
[n] renvoie au bloc de cette fonction : on navigue de fonction en fonction.

<spontaneous> appelant inconnu de gprof (ex. : main, appelée par la libc).

Méthode de lecture d'un profil

1	**Trier par self seconds**	*Quelle fonction consomme le plus de temps propre ?*
2	**Regarder calls**	*Appelée trop souvent (algorithme) ou trop lente à chaque appel (implémentation) ?*
3	**Remonter le graphe d'appels**	*Qui l'appelle, combien de fois ? Ces appels sont-ils tous utiles ?*
4	**Formuler une hypothèse**	*Corriger une seule chose à la fois, puis re-profiler pour la confirmer.*

Autre profiling : les flame graphs



Largeur
Part des échantillons : plus c'est large, plus c'est coûteux.

Hauteur
Profondeur de la pile : chaque étage appelle celui du dessus.

Axe horizontal
Trié alphabétiquement, ce n'est pas une frise temporelle.

Que chercher ?
Les plateaux larges en haut de pile : le temps propre.

Objectifs du profiling



Mesurer les performances
Établir une référence chiffrée (baseline).



Valider une optimisation
Comparer avant / après, chiffres à l'appui.



Identifier les problèmes
Localiser les points chauds et les goulots d'étranglement.



Détecter les régressions
Suivre les mesures d'une version à l'autre (CI).



Prioriser l'effort
Peu de code concentre souvent l'essentiel du coût.



Comprendre le système réel
L'intuition sur « ce qui est lent » se trompe souvent.

Pourquoi mesurer avant d'optimiser ?

Accélérer ×3 une fonction qui pèse…

10 % → **≈ 7 %**

du temps total — de temps gagné

80 % → **60 %**

du temps total — de temps gagné !

Loi d'Amdahl : `accélération globale = 1 / ((1 − p) + p / s)`

p : part du temps optimisée · s : facteur d'accélération

« Premature optimization is the root of all evil » — Donald Knuth, 1974

Comment bien profiler son code

01

Créer un scénario reproductible

Mêmes données, même machine, même build. Plusieurs exécutions.

02

Savoir quoi mesurer

Temps, mémoire, latence, débit ? Partir du symptôme observé.

03

Choisir le bon profiler

Selon la question : CPU ou mémoire, instrumentation ou échantillonnage.

04

Se donner un objectif

Un critère chiffré, ex. : moins de 200 ms par requête.

Mesurer → **Analyser** → **Modifier** → **Re-mesurer** 

Pièges courants

Profiler un build Debug

Mesurer le binaire réellement livré : Release avec symboles (-O2 -g).

Ignorer le bruit

Fréquence CPU, cache froid, autres processus : répéter les mesures.

Des données irréalistes

Avec 10 éléments, un algorithme en $O(n^2)$ passe inaperçu.

Les effets de disposition

Ordre d'édition des liens, taille de l'environnement… peuvent changer les performances.

Gains et limites du profiling



Gains potentiels

- Respect du cahier des charges du projet
- Meilleur retour d'expérience du client
- Moins de coûts d'infrastructure et d'énergie
- Meilleure compréhension du code



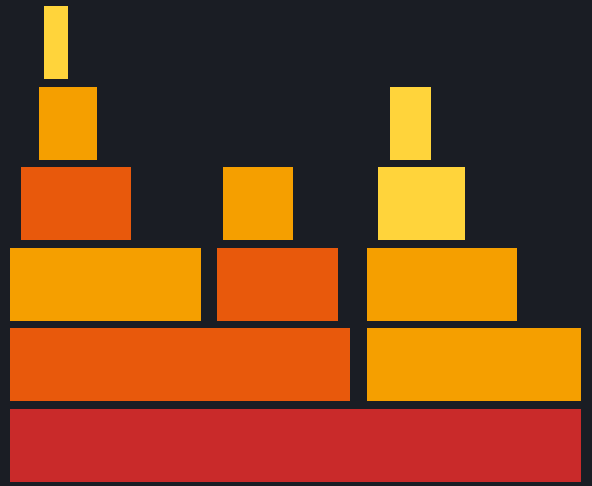
Limites

- Dégradation d'autres métriques (ex. mémoire contre temps) : ignorable ou bloquante selon le contexte
- Lisibilité et maintenabilité parfois sacrifiées
- Résultats liés à la machine, aux données, au compilateur
- Rendements décroissants : chaque gain coûte plus cher

TD

Utilisation de gprof

Profiler un programme C de bout en bout



TD — gprof

```
# 1. Compiler avec instrumentation
gcc -O0 -pg analyseur.c -o analyseur
# 2. Exécuter → produit gmon.out
time ./analyseur
# 3. Analyser
gprof ./analyseur gmon.out > profil.txt
```

Profil plat : temps par fonction (self / cumulé)

Graphe d'appels : qui appelle qui, combien de fois

gmon.out est écrasé
Chaque exécution le remplace : renommez-le (gmon_avant.out…).

Compilez en -O0
En -O2, le compilateur inline et optimise : le profil change.

gprof ne voit pas la libc
Le temps passé dans strlen, qsort, malloc… n'est pas compté.

TD — Consignes

Forkez le dépôt, puis clonez votre fork
https://grond.iut-fbleau.fr/menault/2026_TD1_DEV51_Qualite_Algo

1. **Mesurer**
   Compiler sans -pg, lancer avec time, noter le temps et l'empreinte.
2. **Profiler**
   Recompiler en -O0 -pg, lancer, puis gprof → profil_avant.txt.
3. **Analyser**
   Profil plat et graphe d'appels : où passe le temps, et pourquoi ?
4. **Proposer**
   Lister les pistes d'amélioration, classées par gain attendu.
5. **Optimiser**
   Un commit par optimisation. L'empreinte doit rester identique.
6. **Valider**
   Re-profiler (profil_apres.txt) et chiffrer le gain obtenu.

Compte rendu noté (contrôle continu) : `COMPTE_RENDU.md` à la racine de votre fork
Temps avant / après, analyse du profil, pistes proposées, optimisations réalisées et gains mesurés.

À retenir

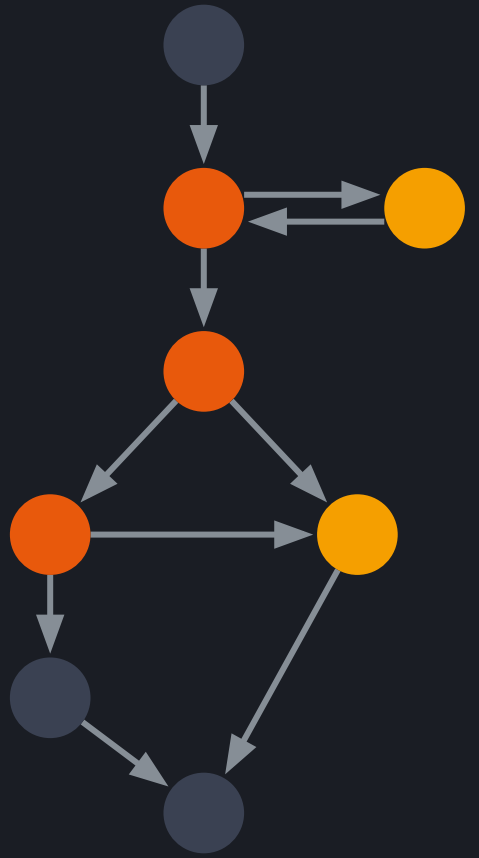
Prochaine partie : complexité cyclomatique

1. **Mesurer avant d'optimiser**
2. **Un scénario reproductible et un objectif chiffré**
3. **Le bon outil pour la bonne question**
4. **Toute optimisation a un coût**

02

Complexité cyclomatique

Mesurer la complexité d'un code sans l'exécuter



La complexité cyclomatique

Le nombre de chemins linéairement indépendants dans le graphe de flot de contrôle d'une fonction.

- Proposée par **Thomas McCabe** en 1976
- Calculée sur le **code source**, sans l'exécuter
- Une valeur **par fonction** (ou méthode)
- Mesure la complexité **structurelle** : combien de branches un lecteur doit suivre, combien de cas un testeur doit couvrir

$$M = E - N + 2P$$

- **E** nombre d'arêtes (transitions)
- **N** nombre de nœuds (blocs d'instructions)
- **P** composantes connexes (1 pour une fonction)

En pratique : **M = décisions + 1**

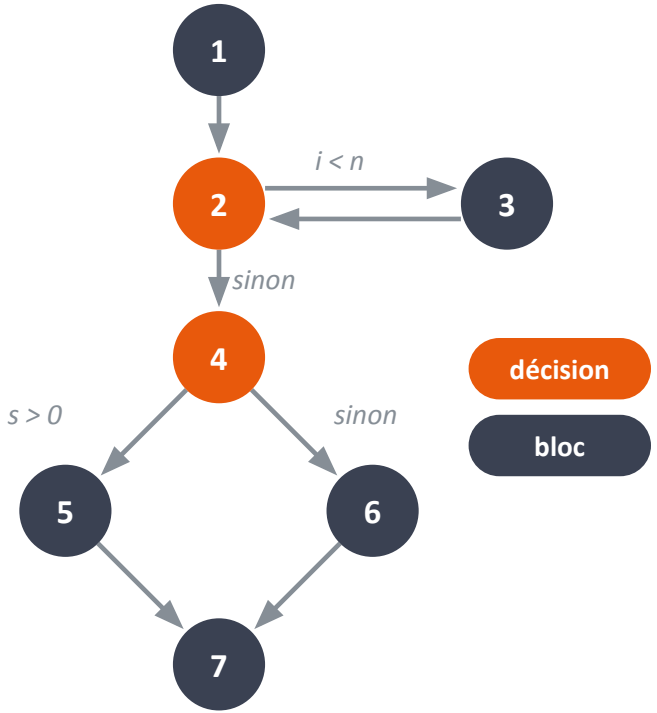
Du code au graphe de flot de contrôle

```
int signe_somme(int *t, int n) {          0
    int s = 0;                            1
    for (int i = 0; i < n; i++)           2
        s += t[i];                        3
    if (s > 0)                            4
        return 1;                         5
    return -1;                            6
}                                         7
```

Les numéros à droite correspondent aux nœuds du graphe.

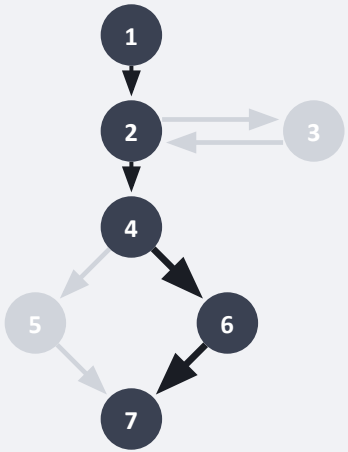
N = 7 nœuds · E = 8 arêtes · P = 1

M = 8 − 7 + 2 × 1 = 3



Les chemins indépendants

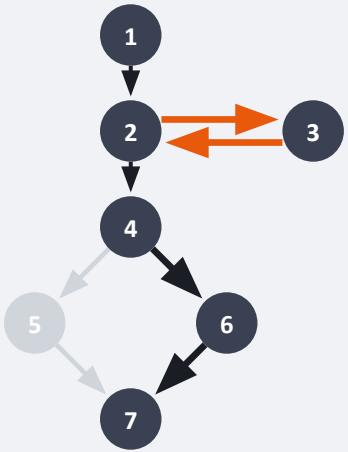
Chemin 1



1 → 2 → 4 → 6 → 7

Test : **t vide (n = 0)** ⟶ −1

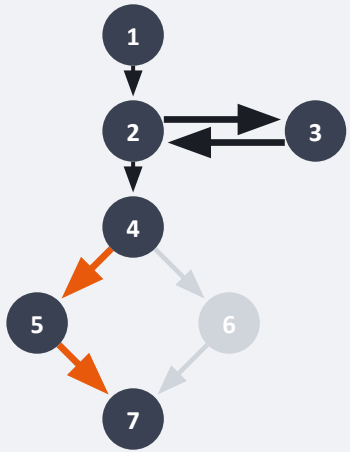
Chemin 2



1 → 2 → 3 → 2 → 4 → 6 → 7

Test : **t = {−3}** ⟶ −1

Chemin 3



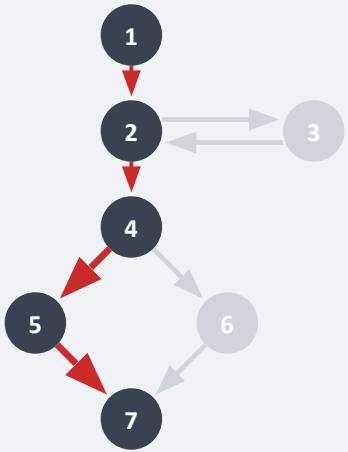
1 → 2 → 3 → 2 → 4 → 5 → 7

Test : **t = {4}** ⟶ 1

Chaque chemin ajoute au moins une arête **nouvelle** : 3 chemins indépendants, donc M = 3 tests.

Et le chemin 1 → 2 → 4 → 5 → 7 ?

Chemin 4 ?



1 → 2 → 4 → 5 → 7

Pas compté dans M

1. Il n'apporte aucune arête nouvelle

En comptant les arêtes : **chemin 4 = chemin 1 + chemin 3 − chemin 2**

	1→2	2→3	3→2	2→4	4→5	4→6	5→7	6→7
Chemin 1	1			1		1		1
+ Chemin 3	1	1	1	1	1		1	
− Chemin 2	−1	−1	−1	−1		−1		−1
= Chemin 4	**1**			**1**	**1**		**1**	

2. Il est impossible à l'exécution

Sans entrer dans la boucle, n ≤ 0 donc s = 0 : s > 0 est forcément faux, le nœud 5 est inatteignable.

Le graphe ignore les valeurs des variables : M compte des chemins du graphe, pas des exécutions.

Une base contient exactement **M = 3** chemins : tout autre chemin se déduit de ceux-ci.

Règles de construction du graphe

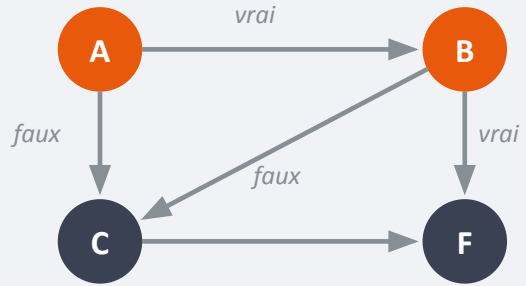
1. **Un nœud = un bloc sans branchement**
   Les instructions qui s'enchaînent toujours forment un seul nœud.

2. **Une décision = un nœud à deux sorties**
   if, for, while, case… Une boucle ajoute une arête de retour.

3. **Une seule sortie**
   Tous les return mènent au même nœud de fin.

4. **&& et || : une décision chacun**
   Évaluation paresseuse : la condition suivante n'est pas toujours évaluée.

```
return (a % 4 == 0
        && a % 100 != 0)
        || a % 400 == 0;
```



A a % 4 == 0 **B** a % 100 != 0
C a % 400 == 0 (renvoyé) **F** fin

N = 4 · E = 5 → M = 5 − 4 + 2 = 3

En pratique : compter les décisions

M = nombre de points de décision + 1

Chaque mot-clé ci-dessous ajoute 1

`if` +1	`else if / elif` +1	`for` +1	`while` +1	`do … while` +1
`case` +1	`catch / except` +1	`&&  ||` +1	`and  or` +1	`a ? b : c` +1

Ne comptent pas `else` `default` `return` `break` `finally`

⚠ Selon l'outil, case, && et || peuvent être comptés ou non : comparez des valeurs issues du même outil.

Outils de calcul

Lizard

C, Java, Python… | CLI

lizard src/ : la complexité de chaque fonction.

pmccabe

C / C++ | CLI

Outil historique sous Linux, sortie compacte.

Radon

Python | CLI

radon cc : une note de A à F par fonction.

flake8 + mccabe

Python | Linter

--max-complexity 10 : alerte au-delà du seuil.

PMD / Checkstyle

Java | Linter

Règles de complexité intégrées au build.

SonarQube

Multi-langages | Plateforme

Tableaux de bord et seuils bloquants en CI.

Et aussi : ESLint (règle complexity), extensions d'IDE comme CodeMetrics…

Profiling et complexité cyclomatique

	Profiling	Complexité cyclomatique
Type d'analyse	Dynamique : on exécute le code	Statique : on lit le code
Nature de la mesure	Concrète : secondes, octets, appels	Abstraite : un nombre de chemins
Prérequis	Un programme qui tourne et un scénario	Le code source, même incomplet
Dépend des données ?	Oui	Non
Objectif	La performance	La lisibilité, la testabilité
Quand ?	Sur un programme existant, ciblé	Dès l'écriture, en revue, en CI



Complémentaires : une fonction très lente peut avoir M = 1, et une fonction à M = 30 peut être instantanée.

Exemple simple — C

```c
int carre(int x) {
    return x * x;
}

int valeur_absolue(int x) {
    if (x < 0)                    +1
        return -x;
    return x;
}
```

`carre`

M = 1

Aucune décision : un seul chemin.

`valeur_absolue`

M = 2

1 décision (if) + 1.

Trivial à lire et à tester

Exemple moyen — Java

```java
public static String mention(double note) {
    if (note < 0 || note > 20)                          +2
        throw new IllegalArgumentException("Note invalide");
    if (note >= 16)                                     +1
        return "Très bien";
    else if (note >= 14)                                +1
        return "Bien";
    else if (note >= 12)                                +1
        return "Assez bien";
    else if (note >= 10)                                +1
        return "Passable";
    return "Ajourné";
}
```

mention

M = 7

5 if / else if, 1 opérateur ||, + 1.

Acceptable, mais 7 cas de test

Exemple complexe — Python

```
def frais_livraison(commande, client):
    if not commande.articles:                                  +1
        return 0
    poids = 0
    for article in commande.articles:                          +1
        if article.dematerialise:                              +1
            continue
        poids += article.poids * article.quantite
    if client.premium and commande.total >= 50:                +2
        return 0
    if commande.pays == "FR":                                  +1
        frais = 4.99 if poids < 2 else 7.99                    +1
    elif commande.pays in ("BE", "LU", "DE"):                  +1
        frais = 9.99 if poids < 2 else 14.99                   +1
    else:
        if poids > 30:                                         +1
            raise ValueError("Colis trop lourd")
        frais = 19.99 + 2 * max(0, poids - 5)
    if commande.express:                                       +1
        frais *= 1.5 if commande.pays == "FR" else 2           +1
    return round(frais, 2)
```

`frais_livraison`

M = 13

6 if / elif, 1 for, 3 ternaires, 1 and, + 1.

Au-delà de 10 : à découper

Réduire la complexité

```
static final double[] SEUILS = {16, 14, 12, 10};
static final String[] MENTIONS =
    {"Très bien", "Bien", "Assez bien", "Passable"};

static String mention(double note) {                    M=3
    verifier(note);
    for (int i = 0; i < SEUILS.length; i++)
        if (note >= SEUILS[i])
            return MENTIONS[i];
    return "Ajourné";
}
static void verifier(double note) {                     M=3
    if (note < 0 || note > 20)
        throw new IllegalArgumentException(...);
}
```

Extraire des fonctions
Une responsabilité par fonction.

Remplacer les if en cascade
Par un tableau ou un dictionnaire.

Sortir tôt
Traiter les cas d'erreur en premier.

Utiliser le polymorphisme
À la place des switch sur un type.

Gains de la complexité cyclomatique



Simplicité de lecture

Moins de branches à suivre : le code se comprend d'une traite.



Simplicité de test

M cas de test couvrent une base de chemins : moins de chemins, moins de tests.



Maintenabilité

Repérer les fonctions à risque pour cibler revue et refactorisation.



Automatisation

Calcul instantané, sans exécution : seuil vérifié en CI, suivi dans le temps.

Objectif : des fonctions courtes, à une seule responsabilité, faciles à tester.

Limites de la complexité cyclomatique

Fondement scientifique discuté

Études partagées. M suit de près le nombre de lignes : que mesure-t-elle de plus ?

Pas forcément adaptée

Un switch de 30 cas est lisible, mais vaut M = 31.

Aveugle à l'imbrication

5 if successifs ou 5 if imbriqués : même M, lisibilité très différente.

Dépend de l'outil

&&, ||, case : comptés ou non. Valeurs non comparables d'un outil à l'autre.

Quelle limite appliquer ?

1 – 10	11 – 20	21 – 50	> 50
Simple	Modéré	Complexe	Très risqué

Convention largement reprise, pas réellement de consensus globale

Seuils par défaut

McCabe (1976)	**10**
PMD, Checkstyle	**10**
Lizard	**15**
Radon	**A à F**

Pas de valeur universelle

→ Fixer le seuil en équipe

→ L'appliquer d'abord au code nouveau ou modifié

→ Justifier les exceptions en revue de code

Pour aller plus loin : la complexité NPath

Nombre de **chemins d'exécution possibles** (Nejmeh, 1988) : les décisions successives **se multiplient** au lieu de s'additionner.

```
if (a) x++;
if (b) y++;
if (c) z++;
```

M

4

3 + 1

NPath

8

2 × 2 × 2

k conditions successives

k	M = k + 1	NPath = 2^k
3	4	8
10	11	1 024
20	21	1 048 576

Seuil courant (PMD) : 200. Mais une autre métrique s'attaque à l'imbrication…

Ouverture : la complexité cognitive

Proposée par SonarSource (2017) pour mesurer l'**effort de lecture** : l'imbrication coûte plus cher, un switch compte une seule fois.

if successifs

```
if (a) n++;          +1
if (b) n++;          +1
if (c) n++;          +1
```

Cyclomatique **4** — Cognitive **3**

if imbriqués

```
if (a)               +1
  if (b)             +2
    if (c)           +3
      n++;
```

Cyclomatique **4** — Cognitive **6**

switch de 7 cas

```
switch (jour) {      +1
  case 1: …
  …
  case 7: …
```

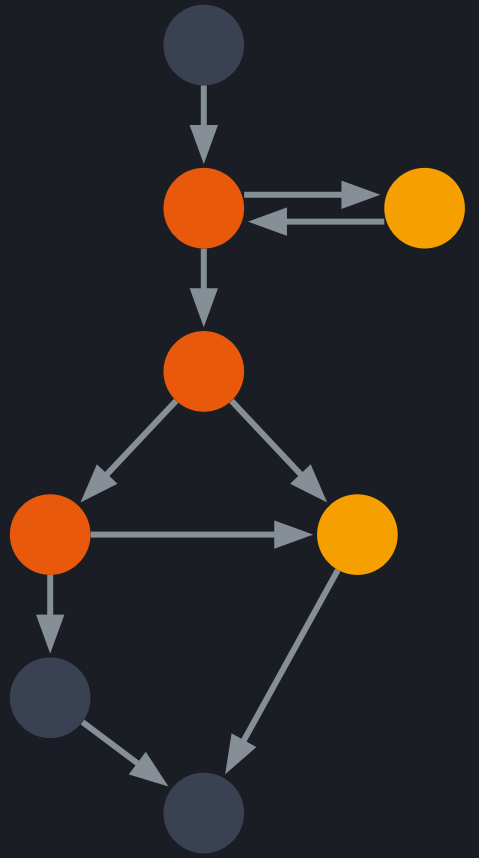
Cyclomatique **8** — Cognitive **1**

Utilisée par **SonarQube** (seuil par défaut : 15), **clang-tidy**, **ESLint** (plugin sonarjs)…

TD

Complexité cyclomatique

Calculer à la main, puis avec un outil



TD — Consignes

Forkez le dépôt du TD, puis clonez votre fork
`https://grond.iut-fbleau.fr/menault/2026_TD2_DEV51_Qualite_Algo`

1. **Dessiner le graphe**
   Pour C2, C4, J2, J6, P2 et P3 : nœuds, arêtes, puis M = E − N + 2P.

2. **Compter les décisions**
   Pour toutes les fonctions : M = décisions + 1.

3. **Classer**
   Simple, modéré, complexe : selon la grille du cours.

4. **Justifier les cas particuliers**
   && et || sans if, switch, compréhension, catch…

5. **Refactoriser**
   Réécrire la fonction la plus complexe, puis recalculer M.

6. **Bonus : outiller**
   Comparer vos valeurs à celles de Lizard.

Compte rendu noté (contrôle continu) : `COMPTE_RENDU.md` à la racine de votre fork
Valeur de M et détail du calcul pour chaque fonction, graphes & refactorisation.

TD — Les codes à analyser

exercices.c

C1	carre	simple
C2	moyenne	simple
C3	est_bissextile	simple
C4	nom_jour	moyen
C5	compter_mots	moyen
C6	date_valide	complexe

Bibliotheque.java

J1	nombreDeLivres	simple
J2	estDisponible	simple
J3	peutEmprunter	complexe
J4	calculerPenalite	complexe
J5	rechercher	complexe
J6	importer	complexe

exercices.py

P1	celsius_vers_fahrenheit	simple
P2	signe	simple
P3	notes_valides	moyen
P4	mention	moyen
P5	analyser_mot_de_passe	complexe



Les fonctions main et les constructeurs ne sont pas à analyser.

TD — Bonus : tester un outil

```
# Installation
python3 -m pip install --user lizard

# Toutes les fonctions des trois fichiers
lizard exercices.c Bibliotheque.java \
       exercices.py

# Seulement celles au-delà de 10
lizard -C 10 -w exercices.c \
       Bibliotheque.java exercices.py

# Variante : un switch compte pour 1
lizard -m exercices.c
```

Vos valeurs correspondent-elles à celles de l'outil ? Expliquez chaque écart.

Avec -m, que devient nom_jour ? Ce choix vous semble-t-il défendable ?

Quel seuil -C retiendriez-vous pour ce projet, et pourquoi ?

PARTIE 2 · COMPLEXITÉ CYCLOMATIQUE

À retenir

Prochaine partie : complexité algorithmique

1. **Une mesure statique, sans exécution**
2. **M = décisions + 1, par fonction**
3. **Un indicateur, pas un verdict**
4. **Un seuil choisi en équipe**