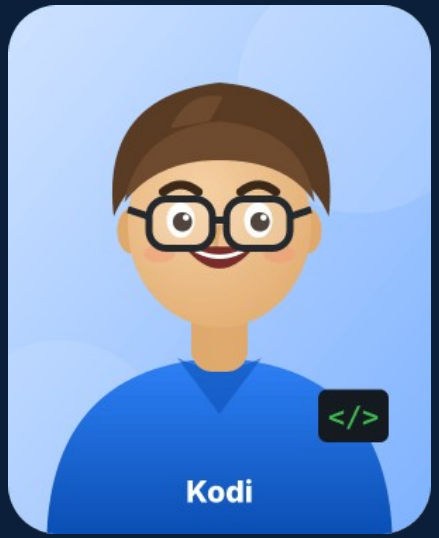
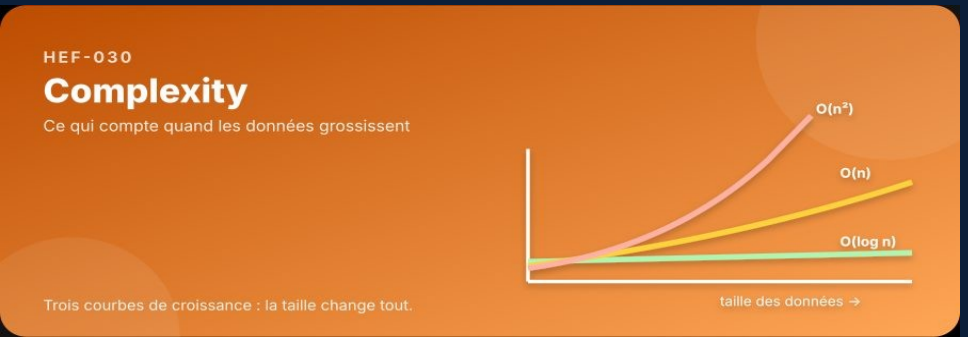
HEF-030 · MISSION 09 / 10

Profilage : trouver où le temps passe

Avant d'optimiser, trouve la ligne qui coûte vraiment. L'intuition se trompe souvent.





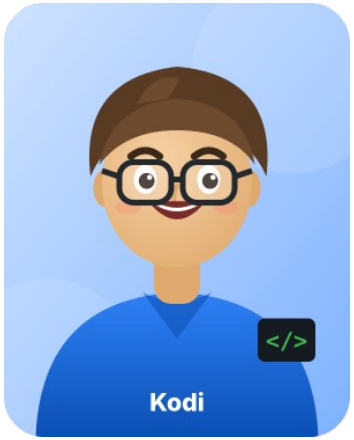
Badge : Détective de performance

70 min · débutant

L'histoire

L'histoire

La génération du rapport mensuel prend 40 secondes. Kodi a une intuition : « C'est sûrement l'accès à la base de données. » Il passe deux jours à l'optimiser. Le gain : 1 seconde.



> *« Tu as optimisé au hasard, dit Sam. Un profileur t'aurait montré en trente secondes que 35 secondes se passent dans une fonction de mise en forme. »*

Tu apprends à utiliser un profileur pour viser juste.

Ta mission

À la fin, je serai capable de…

1. expliquer pourquoi l'intuition se trompe sur les lenteurs
2. utiliser `cProfile` pour classer les fonctions par temps
3. lire un rapport de profilage (appels, temps cumulé, temps propre)
4. identifier le goulot d'étranglement
5. appliquer la règle d'Amdahl de façon intuitive

Défi d'échauffement

Défi d'échauffement

Un trajet de 60 minutes se découpe ainsi : 5 min à pied, 10 min de métro, 40 min de bus, 5 min d'attente. Si tu fais deux fois plus vite **le métro**, de combien de minutes gagnes-tu ? Et si tu fais deux fois plus vite **le bus** ? Quelle partie faut-il améliorer en priorité ?

Le concept

Le principe

Un **profileur** observe un programme pendant qu'il s'exécute et mesure combien de temps chaque fonction consomme.

`cProfile` en Python

```
import cProfile, pstats, io

def travail():
    return sum(i * i for i in range(200000))

profil = cProfile.Profile()
profil.enable()
travail()
profil.disable()
```

Le concept

`cProfile` en Python (suite)

```
sortie = io.StringIO()
pstats.Stats(profil, stream=sortie).sort_stats("cumulative").print_stats(5)
print(sortie.getvalue()[:600])
```

Lire le rapport

Colonne	Signification
`ncalls`	nombre d'appels
`tottime`	temps passé **dans la fonction elle-même** (sans ses sous-appels)
`cumtime`	temps total **y compris** les sous-appels
`percall`	temps moyen par appel

- Une fonction avec un grand `tottime` est un **goulot** : son code lui-même est lent.

Le concept

Lire le rapport (suite)

- Une fonction avec un grand `cumtime` mais un petit `tottime` délègue : il faut descendre dans ses sous-appels.

La loi d'Amdahl (version intuitive)

Si une partie représente une fraction **p** du temps et que tu la rends **s** fois plus rapide, le gain global est limité :

$$\text{temps après} = (1 - p) + p / s$$

Partie optimisée	Part du temps (p)	Accélération (s)	Temps total après
Métro	10/60 ≈ 17 %	2	54 min
Bus	40/60 ≈ 67 %	2	40 min

Le concept

La loi d'Amdahl (version intuitive) (suite)

Optimise ce qui pèse le plus.

Méthode

1. **Mesurer** (profil) → trouver le goulot.
2. **Comprendre** pourquoi il est lent (complexité, structure).
3. **Corriger** une seule chose.
4. **Re-mesurer** pour vérifier le gain.
5. **Recommencer** tant que le gain compte.

🧠 ***Astuce*** *: « Premature optimization is the root of all evil » (Donald Knuth). Optimise ce que les mesures désignent, pas ce que tu crois.*

Exemple pas à pas

Problème : un rapport est lent. Qui est coupable ?

```
import cProfile, pstats, io

def charger(n):
    return [(i, f"membre{i}") for i in range(n)]

def formater(lignes):
    sortie = ""
    for i, nom in lignes:
        sortie += f"{i};{nom}\n"          # concaténation répétée
    return sortie

def statistiques(lignes):
    return len(lignes)

def rapport(n):
    lignes = charger(n)
```

Exemple pas à pas (suite)

```
    texte = formater(lignes)
    return statistiques(lignes), len(texte)

p = cProfile.Profile()
p.enable()
rapport(60000)
p.disable()

s = io.StringIO()
pstats.Stats(p, stream=s).sort_stats("tottime").print_stats(4)
print("\n".join(s.getvalue().splitlines()[:12]))
```

Lecture du résultat : `formater` pèse le plus lourd en temps propre. Correction (une seule chose) :

```
def formater_rapide(lignes):
    return "".join(f"{i};{nom}\n" for i, nom in lignes)
```

Exemple pas à pas (suite)

Re-mesure : on vérifie le gain avec `timeit`.

```
import timeit

lignes = [(i, f"membre{i}") for i in range(60000)]

def formater(lignes):
    sortie = ""
    for i, nom in lignes:
        sortie += f"{i};{nom}\n"
    return sortie

def formater_rapide(lignes):
    return "".join(f"{i};{nom}\n" for i, nom in lignes)

assert formater(lignes) == formater_rapide(lignes)
print(round(min(timeit.repeat(lambda: formater(lignes), number=3, repeat=3)), 4),
      round(min(timeit.repeat(lambda: formater_rapide(lignes), number=3, repeat=3)
```

Exemple pas à pas (suite)

```
    ), 4))
```

À toi, avec aide

Profile ce code et identifie le goulot :

```
def est_premier(n):
    for d in range(2, n):
        if n % d == 0:
            return False
    return n > 1

def compter_premiers(limite):
    return sum(1 for n in range(limite) if est_premier(n))
```

Utilise `cProfile` sur `compter_premiers(5000)`. Quelle fonction pèse le plus ? Propose une correction (une seule modification) et mesure le gain.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Active le profileur autour de l'appel.

Indice 2

Trie par `tottime`.

Indice 3

Il suffit de tester les diviseurs jusqu'à la racine carrée de n.

À toi, avec aide · réponse

Réponse

```
import cProfile, pstats, io, time

def est_premier(n):
    for d in range(2, n):
        if n % d == 0:
            return False
    return n > 1

def est_premier_rapide(n):
    if n < 2:
        return False
    d = 2
    while d * d <= n:
        if n % d == 0:
            return False
        d += 1
    return True

def compter(f, limite):
```

À toi, avec aide · réponse

Réponse (suite)

```
    return sum(1 for n in range(limite) if f(n))

p = cProfile.Profile(); p.enable()
compter(est_premier, 3000)
p.disable()
s = io.StringIO()
pstats.Stats(p, stream=s).sort_stats("tottime").print_stats(3)
print("\n".join(s.getvalue().splitlines()[:10]))

assert compter(est_premier, 3000) == compter(est_premier_rapide, 3000)
for f in (est_premier, est_premier_rapide):
    t = time.perf_counter(); compter(f, 3000); print(f.__name__, round(time.perf_c
   ounter() - t, 4))
```

Le goulot est `est_premier` (la boucle jusqu'à n). Avec la borne √n, le coût passe de O(n) à O(√n) par test.

À toi, en solo

Voici un traitement en trois étapes dont tu ne connais pas la répartition des temps. Écris-le, **profile-le**, puis optimise **uniquement** l'étape qui domine. Montre le gain total et vérifie qu'il est cohérent avec la loi d'Amdahl.

```
def etape_a(n): return [i * 2 for i in range(n)]
def etape_b(liste): return [x for x in liste if x in liste[:200]]
def etape_c(liste): return sorted(liste, reverse=True)

def traitement(n):
    a = etape_a(n)
    b = etape_b(a)
    return etape_c(b)
```

À toi, en solo · réponse

Pistes de correction

```
import cProfile, pstats, io, time

def etape_a(n): return [i * 2 for i in range(n)]
def etape_b(liste):
    return [x for x in liste if x in liste[:200]]          # coûteux : copie + rec
   herche linéaire
def etape_b_rapide(liste):
    echantillon = set(liste[:200])
    return [x for x in liste if x in echantillon]
def etape_c(liste): return sorted(liste, reverse=True)

def traitement(n, b=etape_b):
    return etape_c(b(etape_a(n)))

p = cProfile.Profile(); p.enable()
traitement(5000)
p.disable()
s = io.StringIO()
pstats.Stats(p, stream=s).sort_stats("tottime").print_stats(3)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
print("\n".join(s.getvalue().splitlines()[:9]))

assert traitement(5000) == traitement(5000, etape_b_rapide)
for b in (etape_b, etape_b_rapide):
    t = time.perf_counter(); traitement(5000, b); print(b.__name__, round(time.per
   f_counter() - t, 4))
```

`etape_b` domine car `liste[:200]` est recopié à chaque tour et le test `in` est linéaire. En la corrigeant, le gain global est proche de la part de temps qu'elle occupait.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Optimiser à l'intuition	Gain faible, temps perdu	Profiler d'abord
Profiler sur un trop petit cas	Le goulot réel n'apparaît pas	Profiler avec des données réalistes
Changer plusieurs choses à la fois	On ne sait pas ce qui a aidé	Un changement, une mesure
Confondre `tottime` et `cumtime`	On optimise la mauvaise fonction	Lire les deux colonnes

Mini-boss

Mini-boss

Chasse au goulot. Voici un petit programme lent :

```
def compter_mots(texte):
    mots = texte.split()
    resultat = []
    for mot in mots:
        trouve = False
        for entree in resultat:
            if entree[0] == mot:
                entree[1] += 1
                trouve = True
        if not trouve:
            resultat.append([mot, 1])
    return resultat

def top(resultat, k):
    return sorted(resultat, key=lambda e: -e[1])[:k]
```

Mini-boss

Mini-boss (suite)

1. Génère un texte de 20 000 mots (vocabulaire de 2 000 mots).
2. Profile et nomme le goulot.
3. Corrige en une modification, vérifie l'équivalence des résultats.
4. Mesure le gain et compare à ce que prévoit Amdahl.

Mini-boss · réponse

Pistes de correction

```
import random, time

random.seed(0)
vocab = [f"mot{i}" for i in range(2000)]
texte = " ".join(random.choice(vocab) for _ in range(20000))

def compter_mots(texte):
    resultat = []
    for mot in texte.split():
        trouve = False
        for entree in resultat:
            if entree[0] == mot:
                entree[1] += 1
                trouve = True
        if not trouve:
            resultat.append([mot, 1])
    return resultat

def compter_mots_rapide(texte):
```

Mini-boss · réponse

Pistes de correction (suite)

```
    c = {}
    for mot in texte.split():
        c[mot] = c.get(mot, 0) + 1
    return [[m, n] for m, n in c.items()]

a, b = compter_mots(texte), compter_mots_rapide(texte)
assert sorted(map(tuple, a)) == sorted(map(tuple, b))
for f in (compter_mots, compter_mots_rapide):
    t = time.perf_counter(); f(texte); print(f.__name__, round(time.perf_counter()
    - t, 3))
```

Le goulot est la boucle interne qui parcourt `resultat` : O(n × v). Le dictionnaire la remplace par un accès O(1). Le tri final (`top`) est négligeable, donc le gain global est presque égal au gain local.

Quiz éclair

Quiz éclair

1. Un profileur sert à :

A. corriger automatiquement le code
B. trouver où le temps est passé
C. réduire la taille du fichier

2. `tottime` désigne :

A. le temps total avec sous-appels
B. le nombre d'appels
C. le temps passé dans la fonction elle-même

3. Selon Amdahl, il faut optimiser en priorité :

A. ce qui pèse le plus dans le temps total

Quiz éclair

Quiz éclair (suite)

B. ce qui est le plus facile
C. ce qui est le plus ancien

4. 🔁 *Révision mission 08* : Si T(2n)/T(n) ≈ 4, la croissance est :

A. linéaire
B. quadratique
C. constante

5. 🔁 *Révision mission 07* : Un générateur permet de :

A. trier plus vite
B. accéder par indice
C. produire des valeurs une à une sans tout stocker

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Quelle optimisation « à l'intuition » as-tu déjà faite ? Qu'a-t-elle donné ?
2. Pourquoi le profilage évite-t-il des discussions sans fin dans une équipe ?

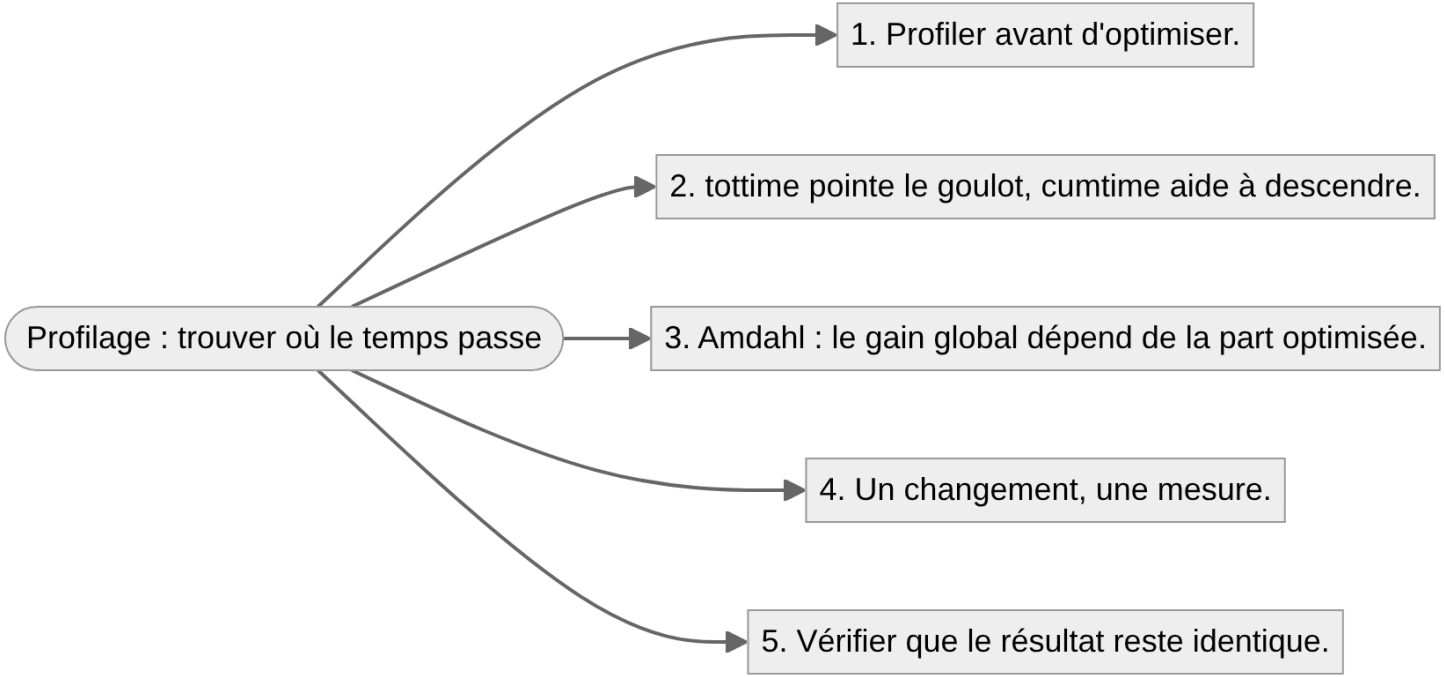
À retenir

À retenir

1. Profiler avant d'optimiser.
2. `tottime` pointe le goulot, `cumtime` aide à descendre.
3. Amdahl : le gain global dépend de la part optimisée.
4. Un changement, une mesure.
5. Vérifier que le résultat reste identique.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Détective de performance

Prochaine mission : Optimiser avec preuves

