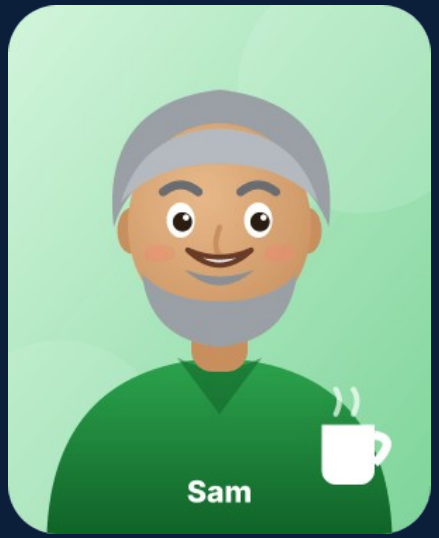
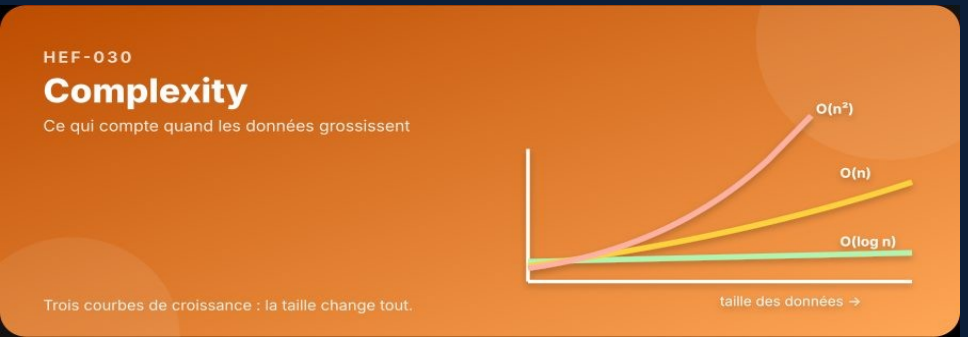
HEF-030 · MISSION 03 / 10

Boucles et compositions

Lire un programme et en déduire son coût, ligne après ligne.





Badge : Analyste de code

70 min · débutant

L'histoire

Kodi hérite d'un script de 40 lignes qui construit le rapport mensuel de HashBox. Il est lent. Où est le problème ?

« Ne cherche pas au hasard, dit Sam. Lis chaque bloc et donne-lui un coût. Puis combine. »



Tu apprends les règles pour calculer la complexité d'un programme à partir de sa structure.

Ta mission

À la fin, je serai capable de…

1. calculer le coût d'une suite d'instructions (addition)
2. calculer le coût de boucles imbriquées (multiplication)
3. reconnaître les boucles qui divisent par deux
4. repérer un coût caché dans un appel de fonction
5. lire un programme et indiquer son goulot d'étranglement

Défi d'échauffement

Défi d'échauffement

Une cuisine prépare un repas pour n invités. Étape 1 : éplucher les légumes (1 minute par invité). Étape 2 : mettre la table (1 minute par invité). Étape 3 : pour **chaque invité**, goûter son plat avec **chaque autre invité** (1 minute par paire). Quel temps total pour 10 invités ? Pour 100 ? Quelle étape domine quand n grandit ?

Le concept

Règle 1 : les instructions en séquence s'additionnent

```
def exemple(liste):
    total = sum(liste)          # O(n)
    maxi = max(liste)           # O(n)
    return total + maxi         # O(1)
```

Coût : O(n) + O(n) + O(1) = **O(n)**. On garde le terme dominant.

Règle 2 : les boucles imbriquées se multiplient

```
def paires(liste):
    for a in liste:             # n tours
        for b in liste:         # n tours
            pass                # O(1)
```

Le concept

Règle 2 : les boucles imbriquées se multiplient (suite)

n × n = **O(n²)**.

Si la boucle interne ne parcourt pas tout (`for j in range(i)`), la somme 0 + 1 + … + (n−1) = n(n−1)/2, qui reste **O(n²)**.

Règle 3 : les boucles qui divisent ou multiplient

```
n = 1000
i = 1
tours = 0
while i < n:
    i *= 2           # i vaut 1, 2, 4, 8, ..., on s'arrête après log₂(n) tours
    tours += 1
print(tours)         # 10
```

Coût **O(log n)**.

Le concept

Règle 4 : les appels de fonctions ont un coût

Une instruction `x in liste` semble simple, mais elle parcourt la liste : **O(n)**. À l'intérieur d'une boucle de n tours, elle donne **O(n²)**.

```
def a_doublon_cache(liste):
    for x in liste:                  # n tours
        if liste.count(x) > 1:       # count parcourt la liste : O(n)
            return True
    return False                     # au total O(n²)
```

Règle 5 : les conditions

Pour un `if/else`, on prend le coût de la **branche la plus coûteuse** (pire cas).

Le concept

Coûts à connaître en Python

Opération	Coût
`liste[i]`, `liste.append(x)`	O(1)
`x in liste`, `liste.index(x)`, `liste.count(x)`	O(n)
`liste.insert(0, x)`, `liste.pop(0)`	O(n)
`x in ensemble`, `dict[cle]`	O(1) en moyenne
`sorted(liste)`	O(n log n)
`liste[a:b]` (copie)	O(b − a)

🧠 ***Astuce*** *: repère les boucles, puis repère les opérations cachées qui parcourent une collection.*

Exemple pas à pas

Problème : analyser ce script.

```
def rapport(membres, cours):
    # membres : liste de n noms ; cours : liste de m couples (nom, titre)
    resultat = []
    for nom in membres:                          # n tours
        titres = []
        for (n_c, titre) in cours:               # m tours
            if n_c == nom:
                titres.append(titre)
        resultat.append((nom, titres))
    return resultat
```

Coût : n × m comparaisons → **O(n × m)**.

Amélioration : indexer les cours par nom en une seule passe.

Exemple pas à pas

Exemple pas à pas (suite)

```
def rapport_rapide(membres, cours):
    index = {}
    for nom, titre in cours:                    # m tours
        index.setdefault(nom, []).append(titre)
    return [(nom, index.get(nom, [])) for nom in membres]   # n tours

membres = ["Ada", "Bob"]
cours = [("Ada", "HEF-000"), ("Ada", "HEF-010"), ("Bob", "HEF-000")]
print(rapport_rapide(membres, cours))
```

Coût : **O(n + m)**.

Avec n = m = 100 000 : 10^{10} opérations contre 2×10^5. **La première version peut prendre plusieurs minutes, la seconde une fraction de seconde.**

À toi, avec aide

À toi, avec aide

Donne le coût de chaque fonction et la ligne responsable :

```
def f1(liste):
    return [x * 2 for x in liste]

def f2(liste):
    resultat = []
    for x in liste:
        if x not in resultat:
            resultat.append(x)
    return resultat

def f3(liste):
    liste.sort()
    return liste[len(liste) // 2]
```

À toi, avec aide · réponse

Indices progressifs

Indice 1

f1 : une seule boucle implicite.

Indice 2

f2 : `x not in resultat` parcourt `resultat`, qui grandit.

Indice 3

f3 : le tri domine.

À toi, avec aide · réponse

Réponse

- `f1` : O(n).
- `f2` : O(n^2) dans le pire cas (sans doublon, `resultat` atteint n éléments et chaque test le parcourt).
- `f3` : O(n log n) (le tri), la lecture au milieu est O(1).

Version de `f2` en O(n) :

```
def f2_rapide(liste):
    vus, resultat = set(), []
    for x in liste:
        if x not in vus:
            vus.add(x)
            resultat.append(x)
    return resultat

print(f2_rapide([3, 1, 3, 2, 1]))   # [3, 1, 2]
```

À toi, en solo

Donne le Big O et propose une version plus rapide si possible :

```
def commun(a, b):                # a et b : listes de taille n
    resultat = []
    for x in a:
        if x in b:
            resultat.append(x)
    return resultat

def plus_proches(liste):         # trouve les deux valeurs les plus proches
    meilleur = None
    for i in range(len(liste)):
        for j in range(i + 1, len(liste)):
            d = abs(liste[i] - liste[j])
            if meilleur is None or d < meilleur:
                meilleur = d
    return meilleur
```

À toi, en solo · réponse

Pistes de correction

- `commun` : O(n²) car `x in b` parcourt `b`. Version rapide : `bset = set(b)` puis `x in bset` : **O(n)**.
- `plus_proches` : O(n²). Version rapide : trier, puis comparer les voisins : **O(n log n)**.

```
def commun_rapide(a, b):
    bset = set(b)
    return [x for x in a if x in bset]

def plus_proches_rapide(liste):
    s = sorted(liste)
    return min(s[i + 1] - s[i] for i in range(len(s) - 1))

print(commun_rapide([1, 2, 3, 4], [3, 4, 5]))   # [3, 4]
print(plus_proches_rapide([10, 3, 8, 5, 4]))    # 1
```

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Ignorer un coût caché	`x in liste` semble O(1)	Connaître le coût des opérations de base
Additionner au lieu de multiplier	Boucles imbriquées sous-estimées	Boucle dans boucle → multiplier
Compter les constantes	Perte de temps	Garder le terme dominant
Oublier les tranches `liste[a:b]`	Chaque tranche copie	Compter la copie

Mini-boss

Mini-boss

Voici un script lent :

```
def classement(scores):                 # scores : liste de (nom, valeur), n éléme
   nts
    resultat = []
    for nom, valeur in scores:
        rang = 1
        for autre_nom, autre_valeur in scores:
            if autre_valeur > valeur:
                rang += 1
        resultat.append((nom, rang))
    return resultat
```

1. Donne son Big O.
2. Écris une version en O(n log n).

Mini-boss

Mini-boss (suite)

3. Vérifie que les deux donnent les mêmes rangs (gère les égalités : même rang) sur 300 listes aléatoires.
4. Mesure les temps pour n = 500, 1 000, 2 000.

Mini-boss · réponse

Pistes de correction

```
import random, time

def classement_lent(scores):
    return [(nom, 1 + sum(1 for _, v in scores if v > valeur)) for nom, valeur in
   scores]

def classement_rapide(scores):
    valeurs = sorted((v for _, v in scores), reverse=True)
    premier_rang = {}
    for i, v in enumerate(valeurs, 1):
        premier_rang.setdefault(v, i)
    return [(nom, premier_rang[valeur]) for nom, valeur in scores]

for _ in range(300):
    s = [(str(i), random.randint(0, 10)) for i in range(random.randint(0, 15))]
    assert classement_lent(s) == classement_rapide(s)

for n in (500, 1000, 2000):
    s = [(str(i), random.randint(0, 10**6)) for i in range(n)]
```

Mini-boss · réponse

Pistes de correction (suite)

```
    t = time.perf_counter(); classement_lent(s); a = time.perf_counter() - t
    t = time.perf_counter(); classement_rapide(s); b = time.perf_counter() - t
    print(n, round(a, 4), round(b, 4))
```

Lent : $O(n^2)$. Rapide : $O(n \log n)$ (tri) + $O(n)$.

Quiz éclair

Quiz éclair

1. Deux boucles imbriquées de n tours donnent :

A. O(2n)
B. O(n^2)
C. O(n)

2. `x in liste` coûte :

A. O(1)
B. O(log n)
C. O(n)

3. Un `while i < n: i *= 2` fait environ :

A. $\log_2$ n tours

Quiz éclair

Quiz éclair (suite)

B. n tours

C. n^2 tours

4. 🔁 *Révision mission 02* : $O(n^2)$ signifie que si n est multiplié par 10, le travail est multiplié par :

A. 10

B. environ 100

C. 1 000

5. 🔁 *Révision mission 01* : On compte les opérations plutôt que les secondes parce que :

A. les opérations sont plus longues

B. c'est obligatoire

C. les secondes dépendent de la machine

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle « opération cachée » t'a le plus surpris·e ?
2. Dans quel projet de ta vie un coût caché te ralentit-il ?

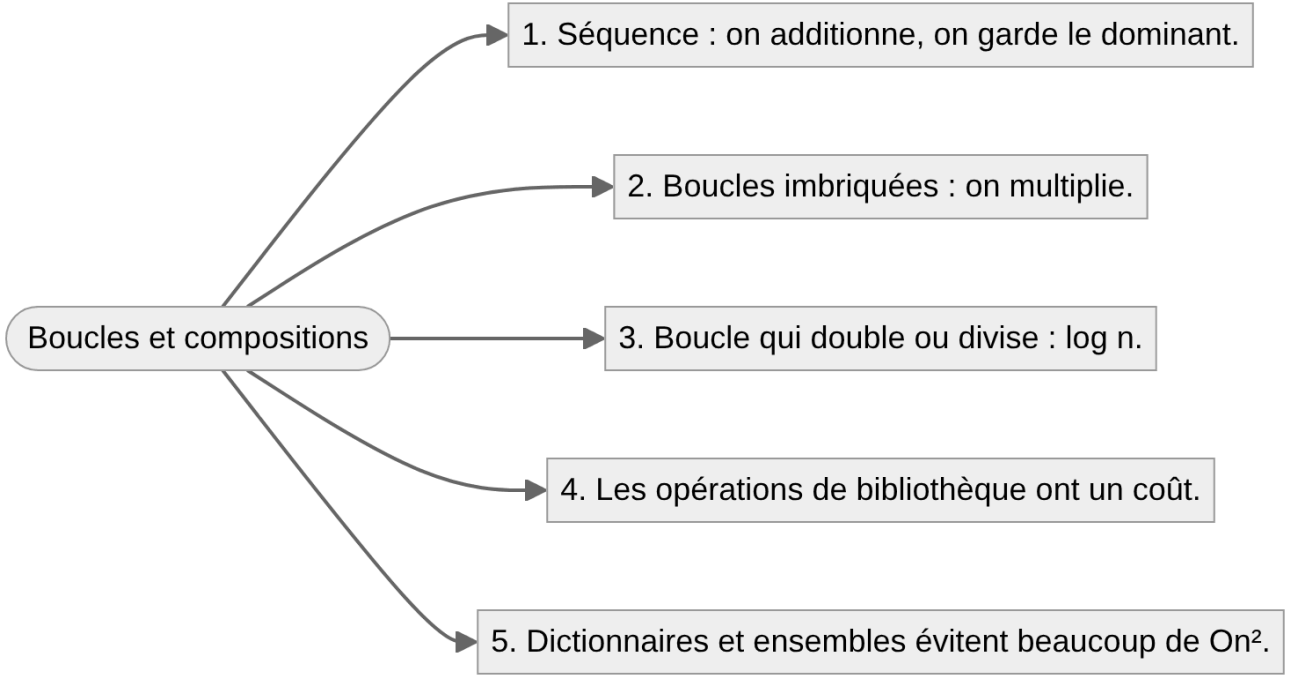
À retenir

À retenir

1. Séquence : on additionne, on garde le dominant.
2. Boucles imbriquées : on multiplie.
3. Boucle qui double ou divise : log n.
4. Les opérations de bibliothèque ont un coût.
5. Dictionnaires et ensembles évitent beaucoup de $O(n^2)$.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Analyste de code

Prochaine mission : Récursion et récurrences

