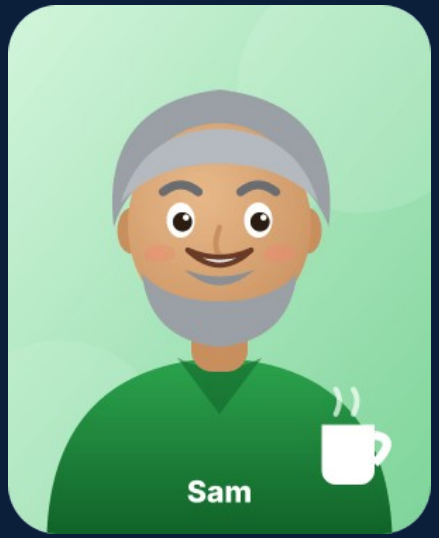
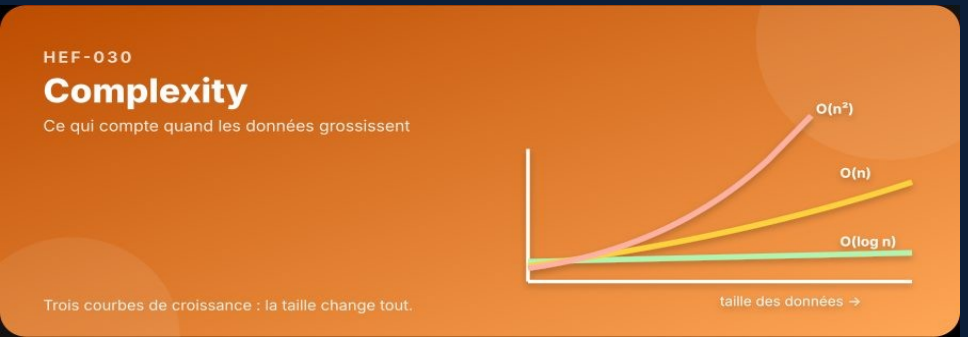
HEF-030 · MISSION 07 / 10

Espace mémoire et compromis temps-mémoire

Aller plus vite en utilisant plus de mémoire, ou économiser la mémoire en acceptant d'aller plus lentement.





Badge : Équilibriste

70 min · débutant

L'histoire

Le serveur de HashBox plante : « MemoryError ». Après avoir ajouté un cache géant pour accélérer les pages, Kodi a consommé toute la mémoire.



« Le temps et la mémoire sont deux ressources, dit Sam. Quand tu en gagnes sur l'une, tu en perds souvent sur l'autre. Il faut décider en connaissance de cause. »

Tu apprends à mesurer la complexité en espace et à arbitrer entre temps et mémoire.

Ta mission

À la fin, je serai capable de…

1. définir la complexité en espace
2. distinguer mémoire de l'entrée et mémoire supplémentaire
3. identifier des compromis temps-mémoire classiques
4. mesurer la mémoire d'une structure avec `sys.getsizeof` et `tracemalloc`
5. décider si un cache est une bonne idée

Défi d'échauffement

Défi d'échauffement

Tu organises une fête et tu dois te souvenir de 100 allergies d'invités. Option A : tu les notes toutes sur une feuille (coûte du papier, mais lecture instantanée). Option B : tu les redemandes à chaque invité (aucun papier, mais beaucoup de temps). Dans quelles situations choisirais-tu A, ou B ?

Le concept

La complexité en espace

C'est la quantité de **mémoire supplémentaire** utilisée en fonction de n, sans compter l'entrée elle-même.

Algorithme	Mémoire supplémentaire
Somme d'une liste	O(1) (une variable)
Copie de liste	O(n)
Tri par insertion	O(1) (sur place)
Tri fusion	O(n)
Récursion de profondeur p	O(p) (la pile d'appels)
Fibonacci avec mémo	O(n)

Le concept

Compromis classiques

On gagne du temps en…	…au prix de la mémoire
gardant un **ensemble** pour tester la présence	O(n) mémoire
mémoïsant les résultats	O(nombre de sous-problèmes)
construisant un **index**	taille de l'index
précalculant une table	taille de la table

Dans l'autre sens, on économise de la mémoire en **recalculant** à la demande, en **compressant**, ou en traitant les données par **flux** (un morceau à la fois).

Le concept

Flux et générateurs

Plutôt que de charger un fichier de 10 Go en mémoire, on le lit **ligne par ligne**. En Python, un **générateur** produit les valeurs une à une.

```
def carres_liste(n):
    return [i * i for i in range(n)]       # n valeurs en mémoire

def carres_generateur(n):
    for i in range(n):
        yield i * i                         # une valeur à la fois

print(sum(carres_generateur(1000)))        # même résultat, mémoire O(1)
```

Le concept

Mesurer la mémoire

```
import sys
print(sys.getsizeof([0] * 1000))     # taille de la liste elle-même
print(sys.getsizeof(set(range(1000))))
```

`tracemalloc` suit la mémoire utilisée par un bloc de code :

```
import tracemalloc

tracemalloc.start()
donnees = [i for i in range(100000)]
courant, pic = tracemalloc.get_traced_memory()
tracemalloc.stop()
print(courant // 1024, "Ko")
```

Le concept

Mesurer la mémoire (suite)

🧠 ***Astuce*** *: avant d'ajouter un cache, demande « combien d'éléments au maximum, et combien d'octets chacun ? »*

Exemple pas à pas

Problème : comparer la mémoire d'une liste et d'un générateur pour 1 million de valeurs.

```
import sys

liste = [i for i in range(1_000_000)]
gen = (i for i in range(1_000_000))

print("liste     :", sys.getsizeof(liste) // 1024, "Ko")
print("générateur:", sys.getsizeof(gen), "octets")
print(sum(liste) == sum(i for i in range(1_000_000)))   # même résultat
```

Observation : la liste occupe plusieurs mégaoctets (8 octets par référence au minimum, sans compter les entiers eux-mêmes) ; le générateur quelques centaines d'octets.

Le compromis : le générateur ne permet de lire les valeurs qu'**une seule fois, dans l'ordre**. La liste permet l'accès par indice et plusieurs parcours.

Exemple pas à pas (suite)

Cache avec borne : on limite la taille du cache pour contrôler la mémoire.

```
from functools import lru_cache

@lru_cache(maxsize=128)
def carre_couteux(n):
    return n * n

for i in range(1000):
    carre_couteux(i % 200)
print(carre_couteux.cache_info())
```

À toi, avec aide

Écris deux versions de `moyenne_fichier(lignes)` où `lignes` est un itérable de textes contenant chacun un nombre :

1. une version qui construit d'abord la liste de tous les nombres ;
2. une version qui calcule la moyenne en un seul passage, avec 2 variables.

Compare la mémoire avec `tracemalloc` pour 200 000 lignes.

À toi, avec aide · réponse

Indices progressifs

Indice 1

La version flux garde seulement `somme` et `compte`.

Indice 2

Utilise `(str(i) for i in range(200000))` comme source.

Indice 3

Compare le pic mémoire `get_traced_memory()[1]`.

À toi, avec aide · réponse

Réponse

```
import tracemalloc

def moyenne_liste(lignes):
    nombres = [float(l) for l in lignes]
    return sum(nombres) / len(nombres)

def moyenne_flux(lignes):
    somme, compte = 0.0, 0
    for l in lignes:
        somme += float(l)
        compte += 1
    return somme / compte

def pic(f):
    tracemalloc.start()
    f((str(i) for i in range(200000)))
    _, p = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return p // 1024
```

À toi, avec aide · réponse

Réponse (suite)

```
print("liste:", pic(moyenne_liste), "Ko ; flux:", pic(moyenne_flux), "Ko")
```

La version flux est en O(1) mémoire ; la liste en O(n).

À toi, en solo

Table de précalcul. Tu dois répondre très vite à 100 000 questions « combien y a-t-il d'entiers premiers inférieurs ou égaux à x ? » pour x ≤ 100 000.

1. Version sans précalcul : compter à chaque question.
2. Version avec précalcul : une table construite une fois.
3. Compare le temps et la mémoire. Quand le précalcul est-il justifié ?

À toi, en solo · réponse

Pistes de correction

```
import time, random

N = 100_000

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

def sans_precalcul(x):
    return sum(1 for k in range(2, x + 1) if est_premier(k))

# crible d'Ératosthène + table cumulée
crible = [True] * (N + 1)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
crible[0] = crible[1] = False
for i in range(2, int(N ** 0.5) + 1):
    if crible[i]:
        for j in range(i * i, N + 1, i):
            crible[j] = False
cumul, c = [0] * (N + 1), 0
for k in range(N + 1):
    c += crible[k]
    cumul[k] = c

assert all(sans_precalcul(x) == cumul[x] for x in (10, 100, 1000))
t = time.perf_counter()
for _ in range(100000):
    cumul[random.randint(0, N)]
print(round(time.perf_counter() - t, 4), "s pour 100 000 questions ; table de", N
   + 1, "entiers")
```

Pistes de correction (suite)

Le précalcul coûte O(N log log N) une fois et O(N) mémoire, puis chaque réponse est O(1). Justifié dès qu'il y a beaucoup de questions sur une plage bornée.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Cache sans limite	La mémoire grossit jusqu'au plantage	Borner la taille (`lru_cache(maxsize=…)`)
Charger tout en mémoire	`MemoryError` sur les gros fichiers	Lire par flux
Ignorer la pile d'appels	Récursion profonde = mémoire et `RecursionError`	Utiliser une boucle
Optimiser la mémoire sans besoin	Code plus complexe pour rien	Mesurer d'abord

Mini-boss

Mini-boss

Le cache de HashBox. La page de profil appelle `calculer_stats(membre_id)` (coûteux : simulé par un calcul de 5 ms). 50 000 membres, mais 80 % des requêtes concernent 500 membres populaires.

1. Écris la fonction sans cache et mesure 2 000 requêtes générées selon cette répartition.
2. Ajoute `lru_cache` avec `maxsize` = 100, 500, 5 000.
3. Mesure temps total et mémoire pour chaque taille.
4. Recommande une taille de cache avec une justification chiffrée.

Mini-boss · réponse

Pistes de correction

```
import random, time
from functools import lru_cache

random.seed(1)
populaires = list(range(500))
requetes = [random.choice(populaires) if random.random() < 0.8 else random.randint
   (500, 49999) for _ in range(2000)]

def calculer_stats(i):
    time.sleep(0.0005)         # simule 0,5 ms (réduit pour que l'exercice reste c
   ourt)
    return i * 2

def executer(f):
    t = time.perf_counter()
    for r in requetes:
        f(r)
    return time.perf_counter() - t
```

Mini-boss · réponse

Pistes de correction (suite)

```
print("sans cache :", round(executer(calculer_stats), 2), "s")
for taille in (100, 500, 5000):
    cache = lru_cache(maxsize=taille)(calculer_stats)
    print(taille, round(executer(cache), 2), "s", cache.cache_info().hits, "succès
   ")
```

Recommandation attendue : ~500 à 1 000 entrées capturent presque tous les gains ; 5 000 consomme plus de mémoire pour un gain marginal.

Quiz éclair

Quiz éclair

1. La complexité en espace mesure :

A. la mémoire supplémentaire utilisée
B. le temps d'exécution
C. le nombre de lignes

2. Un générateur permet de :

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

3. Un cache sans limite risque de :

A. ralentir le processeur

Quiz éclair

Quiz éclair (suite)

B. effacer les données
C. consommer toute la mémoire

4. 🔁 *Révision mission 06* : Le tri fusion utilise une mémoire supplémentaire de :

A. O(n)
B. O(1)
C. O(n²)

5. 🔁 *Révision mission 05* : `x in ensemble` coûte en moyenne :

A. O(n)
B. O(1)
C. O(n²)

Réponses

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

Je réfléchis

1. Dans ta vie, où as-tu « précalculé » quelque chose pour gagner du temps plus tard ?
2. Quel serait un bon critère pour décider qu'un cache vaut la mémoire qu'il coûte ?

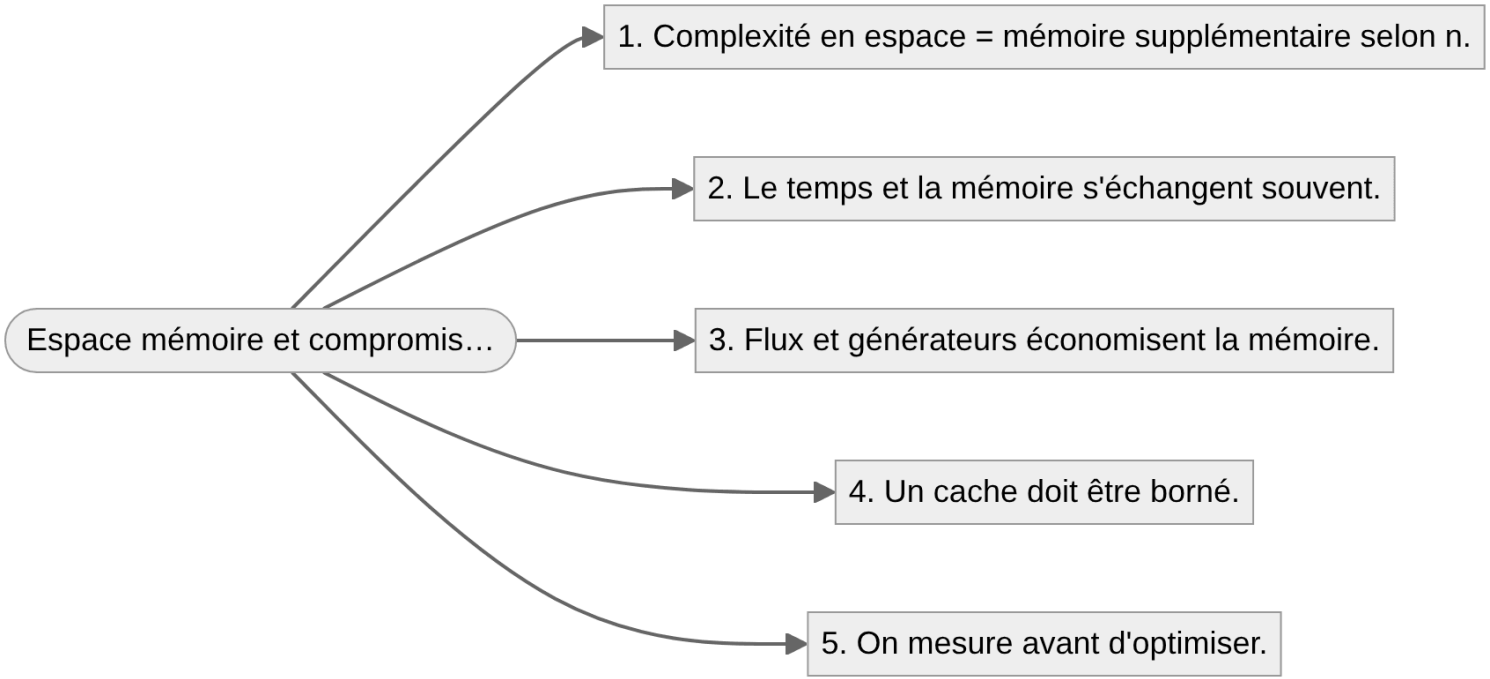
À retenir

À retenir

1. Complexité en espace = mémoire supplémentaire selon n.
2. Le temps et la mémoire s'échangent souvent.
3. Flux et générateurs économisent la mémoire.
4. Un cache doit être borné.
5. On mesure avant d'optimiser.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Équilibriste

Prochaine mission : Mesurer pour de vrai

