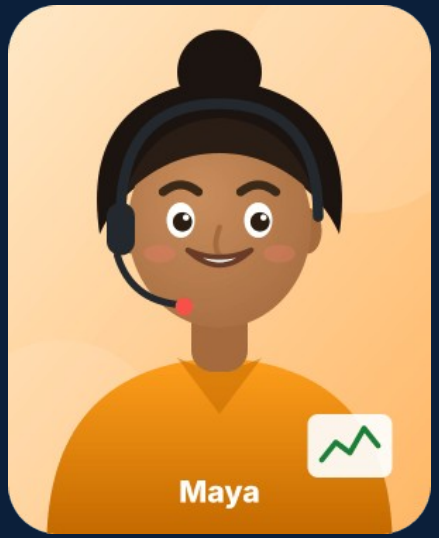
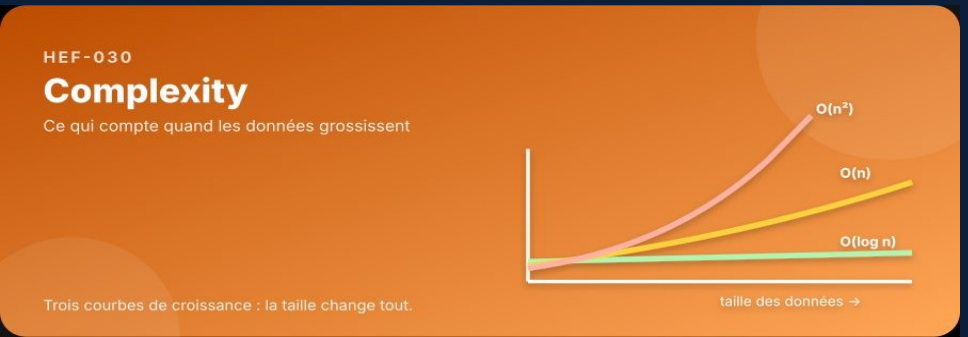
HEF-030 · MISSION 06 / 10

Recherche et tri : le grand écart

Pourquoi un bon tri est un million de fois plus rapide qu'un mauvais, sur les grandes données.



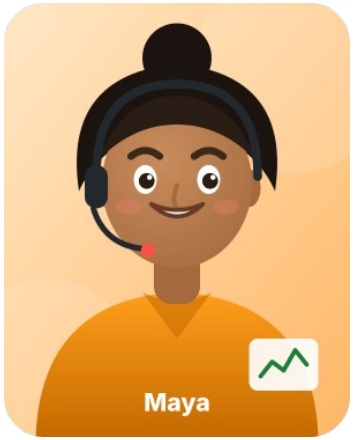


Badge : Comparateur de tris

75 min · débutant

L'histoire

Pour son rapport annuel, HashBox doit trier **2 millions de lignes**. Kodi utilise son tri par insertion de la dernière fois. Après 20 minutes, rien ne s'est passé.



> *« Ton tri est correct, dit Sam, mais il est en O(n²). Pour 2 millions de lignes, ça fait quatre mille milliards d'opérations. »*

Tu compares précisément les algorithmes de recherche et de tri vus plus tôt, et tu comprends pourquoi `sorted` est si rapide.

Ta mission

À la fin, je serai capable de…

1. donner la complexité des recherches linéaire et binaire
2. donner la complexité des tris simples et du tri fusion
3. expliquer pourquoi trier ne peut pas être plus rapide que O(n log n) par comparaisons
4. mesurer expérimentalement la différence
5. choisir entre trier et ne pas trier selon le besoin

Défi d'échauffement

Défi d'échauffement

Tu dois trier un jeu de **1 000 cartes** à la main. Méthode A : tu les ranges une par une à sa place dans la pile déjà triée. Méthode B : tu coupes en deux paquets, tu les tries chacun puis tu les fusionnes. Combien de comparaisons vas-tu faire environ pour chaque méthode ? Laquelle te paraît plus rapide quand on passe à 1 000 000 de cartes ?

Le concept

Recherche

Algorithme	Condition	Pire cas
Linéaire	aucune	O(n)
Binaire	liste triée	O(log n)
Dictionnaire / ensemble	clés hachables	O(1) en moyenne

Tris simples

Tri	Pire cas	Meilleur cas	Stable ?	Remarque
Sélection	$O(n^2)$	$O(n^2)$	non	toujours $n^2/2$ comparaisons
Insertion	$O(n^2)$	O(n)	oui	excellent sur données presque triées

Le concept

Tris simples (suite)

Tri	Pire cas	Meilleur cas	Stable ?	Remarque
Bulles	$O(n^2)$	$O(n)$	oui	rarement utile

Tris efficaces

Tri	Pire cas	Mémoire en plus	Stable ?
Fusion	$O(n \log n)$	$O(n)$	oui
Rapide (quicksort)	$O(n^2)$ rare, $O(n \log n)$ en moyenne	$O(\log n)$	non
Tas	$O(n \log n)$	$O(1)$	non
`sorted` de Python (Timsort)	$O(n \log n)$	$O(n)$	oui

Timsort combine fusion et insertion et profite des parties déjà triées : sur des données presque triées, il approche $O(n)$.

Le concept

Pourquoi O(n log n) est une barrière

Pour trier par comparaisons, il faut distinguer parmi n! ordres possibles. Chaque comparaison a 2 réponses, donc il faut au moins $\log_2(n!) \approx n \log_2 n$ comparaisons. **Aucun tri par comparaisons ne peut faire mieux en général.**

(Des tris spéciaux comme le tri par comptage dépassent cette barrière, mais seulement si les valeurs sont des petits entiers.)

Trier pour chercher : à quelles conditions ?

Situation	Choix
Une seule recherche	recherche linéaire (trier coûte plus)
Beaucoup de recherches sur les mêmes données	trier une fois puis recherche binaire, ou ensemble

Le concept

Trier pour chercher : à quelles conditions ? (suite)

Situation	Choix
Données qui arrivent en continu	structure incrémentale (tas, BST, `bisect.insort`)

Si on fait k recherches : linéaire = k × n ; trier + binaire = n log n + k log n. Au-delà d'environ log n recherches, trier est rentable.

Exemple pas à pas

Problème : mesurer l'écart entre le tri par insertion et `sorted`.

```
import random, time

def tri_insertion(liste):
    liste = list(liste)
    for i in range(1, len(liste)):
        courant = liste[i]
        j = i - 1
        while j >= 0 and liste[j] > courant:
            liste[j + 1] = liste[j]
            j -= 1
        liste[j + 1] = courant
    return liste

for n in (500, 1000, 2000):
    donnees = [random.random() for _ in range(n)]
    t = time.perf_counter(); a = tri_insertion(donnees); t1 = time.perf_counter()
```

Exemple pas à pas (suite)

```
    - t
     t = time.perf_counter(); b = sorted(donnees); t2 = time.perf_counter() - t
     assert a == b
     print(n, round(t1, 4), round(t2, 5))
```

Observation : quand n double, l'insertion devient environ 4 fois plus lente ; `sorted` à peine plus du double. Pour n = 2 millions, l'insertion prendrait des heures.

Cas presque trié : l'insertion devient rapide.

```
import time

def tri_insertion(liste):
    liste = list(liste)
    for i in range(1, len(liste)):
        courant = liste[i]
        j = i - 1
```

Exemple pas à pas (suite)

```
        while j >= 0 and liste[j] > courant:
            liste[j + 1] = liste[j]
            j -= 1
        liste[j + 1] = courant
    return liste

donnees = list(range(3000))
donnees[10], donnees[2990] = donnees[2990], donnees[10]
t = time.perf_counter(); tri_insertion(donnees); print(round(time.perf_counter() -
    t, 4))
```

À toi, avec aide

Implémente le **tri par comptage** pour des entiers entre 0 et 100 : compte combien de fois chaque valeur apparaît, puis reconstruis la liste triée.

1. Quel est son coût en fonction de n (nombre d'éléments) et k (taille de la plage) ?
2. Pourquoi ne contredit-il pas la barrière n log n ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Crée une liste `comptes` de taille 101, remplie de zéros.

Indice 2

Parcours les données pour incrémenter `comptes[x]`.

Indice 3

Parcours `comptes` et recrée la liste en répétant chaque valeur.

À toi, avec aide · réponse

Réponse

```
def tri_comptage(liste, maximum=100):
    comptes = [0] * (maximum + 1)
    for x in liste:
        comptes[x] += 1
    resultat = []
    for valeur, c in enumerate(comptes):
        resultat.extend([valeur] * c)
    return resultat

print(tri_comptage([5, 3, 99, 3, 0, 5]))   # [0, 3, 3, 5, 5, 99]
```

Coût : **O(n + k)**. Il ne compare jamais deux éléments entre eux : la barrière n log n ne s'applique qu'aux tris par comparaisons. En contrepartie, il exige des entiers dans une petite plage connue.

À toi, en solo

Seuil de rentabilité. Tu dois faire k recherches sur n = 100 000 éléments non triés.

1. Écris le coût linéaire total et le coût « trier une fois + recherches binaires ».
2. À partir de combien de recherches k trier devient-il rentable ? (utilise `math.log2`)
3. Vérifie avec un petit programme qui mesure les temps pour k = 1, 10, 100, 1 000.

À toi, en solo · réponse

Pistes de correction

```
import math, random, time, bisect

n = 100_000
donnees = [random.randint(0, 10**9) for _ in range(n)]
cibles = [random.randint(0, 10**9) for _ in range(1000)]

def lineaire(k):
    t = time.perf_counter()
    for c in cibles[:k]:
        c in donnees
    return time.perf_counter() - t

def tri_puis_binaire(k):
    t = time.perf_counter()
    tri = sorted(donnees)
    for c in cibles[:k]:
        i = bisect.bisect_left(tri, c)
        i < len(tri) and tri[i] == c
    return time.perf_counter() - t
```

À toi, en solo · réponse

Pistes de correction (suite)

```
for k in (1, 10, 100, 1000):
    print(k, round(lineaire(k), 4), round(tri_puis_binaire(k), 4))

print("théorie : rentable dès que k × n > n log n, soit k >", round(math.log2(n)))
```

Rentable dès que k dépasse environ $\log_2(n) \approx 17$ recherches, en théorie. En pratique `in` sur une liste étant très optimisé en C, le seuil réel est un peu plus grand.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Utiliser un tri O(n²) sur de grandes données	Programme qui ne finit pas	Utiliser `sorted()`
Trier pour une seule recherche	Plus coûteux que chercher	Compter le nombre de recherches
Croire que `sorted` est lent	Il est écrit en C et très optimisé	L'utiliser en priorité
Oublier la stabilité	Ordre des égaux perdu	Choisir un tri stable si nécessaire

Mini-boss

Mini-boss

Bataille des tris. Implémente `tri_insertion`, `tri_fusion` et utilise `sorted`.

1. Génère des données de trois types : aléatoires, déjà triées, triées à l'envers.
2. Mesure les trois tris pour n = 500, 1 000, 2 000 sur chaque type de données.
3. Remplis un tableau 3 × 3 × 3.
4. Écris un paragraphe : quel tri choisir dans quel cas ? Où l'insertion gagne-t-elle ?

Mini-boss · réponse

Pistes de correction

```
import random, time

def tri_insertion(liste):
    liste = list(liste)
    for i in range(1, len(liste)):
        courant = liste[i]
        j = i - 1
        while j >= 0 and liste[j] > courant:
            liste[j + 1] = liste[j]
            j -= 1
        liste[j + 1] = courant
    return liste

def tri_fusion(liste):
    if len(liste) <= 1:
        return list(liste)
    m = len(liste) // 2
    a, b = tri_fusion(liste[:m]), tri_fusion(liste[m:])
    r, i, j = [], 0, 0
```

Mini-boss · réponse

Pistes de correction (suite)

```
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            r.append(a[i]); i += 1
        else:
            r.append(b[j]); j += 1
    return r + a[i:] + b[j:]

def mesurer(f, d):
    t = time.perf_counter(); f(d); return time.perf_counter() - t

for n in (500, 1000):
    jeux = {"aléatoire": [random.random() for _ in range(n)],
            "trié": list(range(n)), "inversé": list(range(n, 0, -1))}
    for nom, d in jeux.items():
        print(n, nom, *(round(mesurer(f, d), 5) for f in (tri_insertion, tri_fusio
   n, sorted)))
```

Mini-boss · réponse

Pistes de correction (suite)

Résultats typiques : l'insertion est la meilleure sur données déjà triées (O(n)), la pire sur données inversées (O(n²)). La fusion est stable en O(n log n) partout. `sorted` gagne presque toujours.

Quiz éclair

1. Un tri par comparaisons ne peut pas faire mieux en général que :

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

2. Le tri par insertion est très efficace quand les données sont :

A. aléatoires
B. inversées
C. presque triées

3. Trier avant de chercher est rentable quand :

A. on fait beaucoup de recherches

Quiz éclair

Quiz éclair (suite)

B. on fait une seule recherche
C. les données changent sans cesse

4. 🔁 *Révision mission 05* : `liste.pop(0)` coûte :

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

5. 🔁 *Révision mission 04* : La mémoïsation consiste à :

A. effacer la mémoire
B. supprimer la récursion
C. mémoriser les résultats déjà calculés

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quand tu ranges des documents, utilises-tu plutôt un tri « par insertion » ou « par fusion » ?
2. Pourquoi connaître la barrière n log n évite-t-il de chercher un tri miracle ?

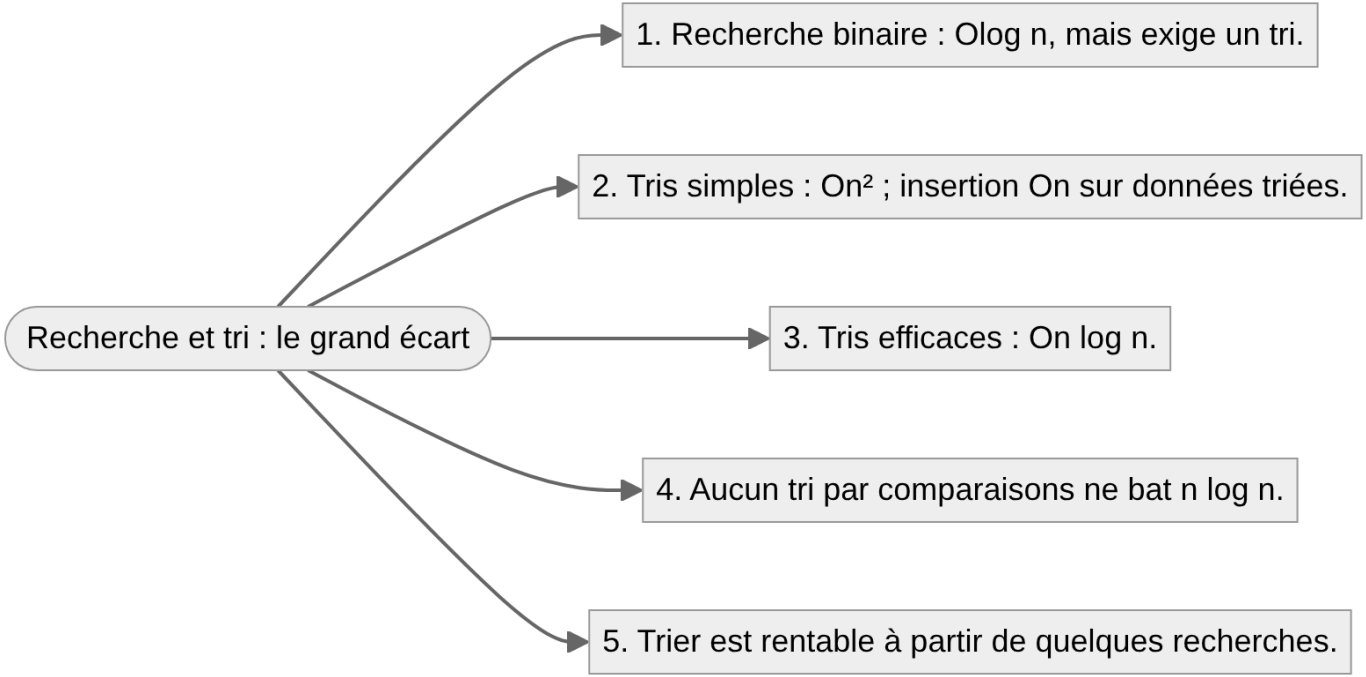
À retenir

À retenir

1. Recherche binaire : O(log n), mais exige un tri.
2. Tris simples : $O(n^2)$; insertion O(n) sur données triées.
3. Tris efficaces : O(n log n).
4. Aucun tri par comparaisons ne bat n log n.
5. Trier est rentable à partir de quelques recherches.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Comparateur de tris

Prochaine mission : Espace mémoire et compromis temps-mémoire

