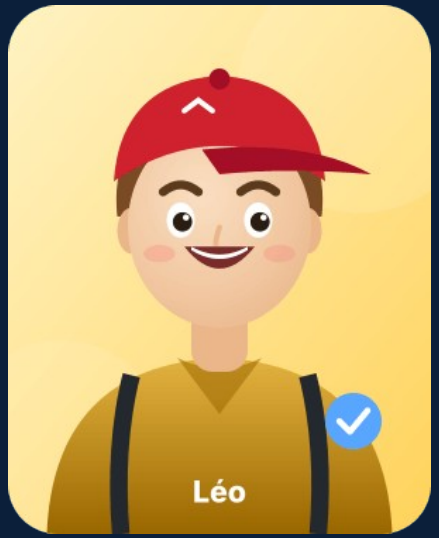
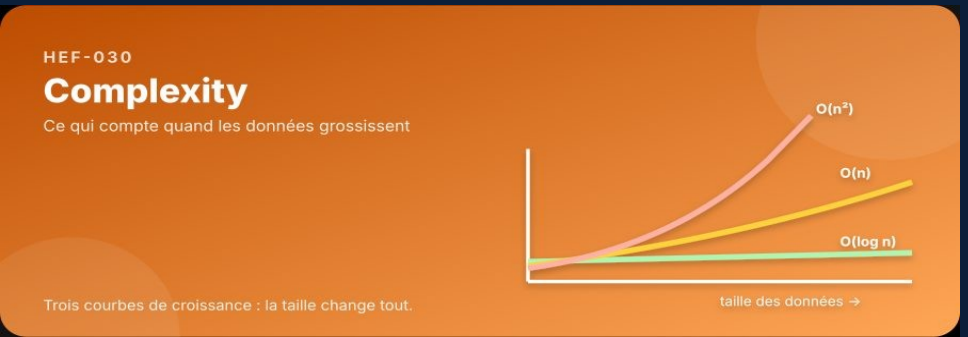
HEF-030 · MISSION 04 / 10

Récursion et récurrences

Pour mesurer un algorithme qui s'appelle lui-même, on compte ses appels.





Badge : Calculateur de profondeur

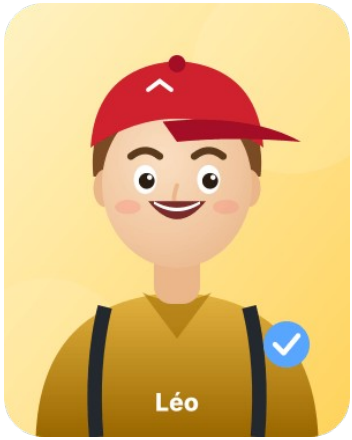
75 min · débutant

L'histoire

Kodi a écrit une fonction récursive pour calculer des suites de Fibonacci pour un bonus de fidélité. Pour n = 30, ça va. Pour n = 50, le serveur ne répond plus.

« Ta fonction recalcule les mêmes choses des milliards de fois, dit Sam. Dessine l'arbre de ses appels. »

Tu apprends à analyser le coût d'un algorithme récursif et à repérer les calculs répétés.



Ta mission

À la fin, je serai capable de…

1. dessiner l'arbre des appels d'une fonction récursive
2. compter le nombre d'appels
3. écrire et résoudre une récurrence simple
4. expliquer pourquoi Fibonacci naïf est exponentiel
5. corriger avec la mémoïsation

Défi d'échauffement

Défi d'échauffement

Une rumeur démarre : chaque personne qui l'entend la répète à **2 autres personnes** au tour suivant. Au tour 0, 1 personne la connaît. Combien de nouvelles personnes l'apprennent au tour 1, 2, 3, 10, 20 ? Que remarques-tu sur la vitesse de croissance ?

Le concept

L'arbre des appels

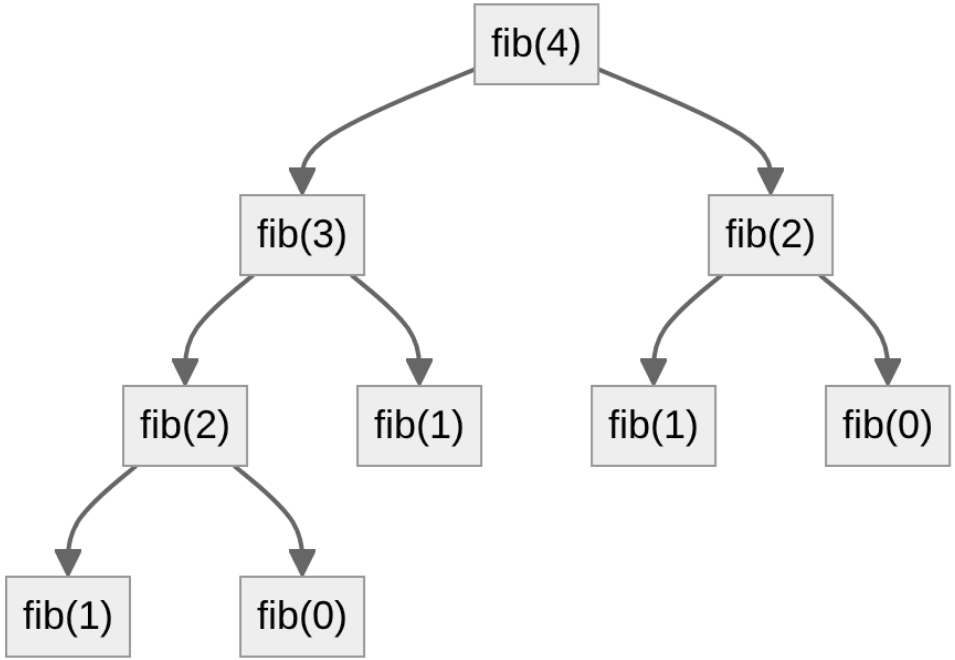
Chaque appel récursif est un nœud ; ses appels internes sont ses enfants. Le coût total est lié au nombre de nœuds.

```
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)
```

Pour `fib(4)` :

Le concept

L'arbre des appels (suite)



Le concept

L'arbre des appels (suite)

Description du schéma : `fib(4)` appelle `fib(3)` et `fib(2)`. `fib(3)` appelle à son tour `fib(2)` et `fib(1)`. On voit que `fib(2)` est calculé deux fois : les mêmes calculs sont répétés.
Le nombre d'appels est environ **$1{,}6^n$** : **exponentiel**.

Les récurrences

Une **récurrence** décrit le coût d'un algorithme récursif en fonction du coût sur des problèmes plus petits.

Algorithme	Récurrence	Résultat
Parcours récursif d'une liste (retire 1 élément)	$T(n) = T(n-1) + 1$	$O(n)$
Recherche binaire	$T(n) = T(n/2) + 1$	$O(\log n)$
Tri fusion	$T(n) = 2\,T(n/2) + n$	$O(n \log n)$

Le concept

Les récurrences (suite)

Algorithme	Récurrence	Résultat
Fibonacci naïf	$T(n) = T(n-1) + T(n-2) + 1$	$O(1{,}6^n)$

Comment résoudre ?

Méthode de l'arbre :

1. dessiner les niveaux de l'arbre des appels ;
2. calculer le travail à chaque niveau ;
3. additionner les niveaux.

Tri fusion : à chaque niveau, on traite n éléments au total ; il y a $\log_2 n$ niveaux ; donc n log n.

Le concept

La mémoïsation

Si les mêmes sous-problèmes reviennent, **on mémorise leurs résultats** dans un dictionnaire.

```
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n < 2:
        return n
    if n not in memo:
        memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
    return memo[n]
```

Chaque valeur n'est calculée qu'une fois : **O(n)**. On a échangé de la mémoire contre du temps.

Exemple pas à pas

Exemple pas à pas

Problème : mesurer le nombre d'appels de `fib` et de `fib_memo`.

```
appels = 0

def fib(n):
    global appels
    appels += 1
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

for n in (10, 15, 20, 25):
    appels = 0
    fib(n)
    print(n, appels)
# 10 177
# 15 1973
# 20 21891
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# 25 242785
```

Observation : quand n augmente de 5, le nombre d'appels est multiplié par environ 11 : croissance **exponentielle**.

```
def fib_memo(n, memo=None, compteur=None):
    if memo is None:
        memo, compteur = {}, [0]
    compteur[0] += 1
    if n < 2:
        return n
    if n not in memo:
        memo[n] = fib_memo(n - 1, memo, compteur) + fib_memo(n - 2, memo, compteur
   )
    return memo[n]

def appels_memo(n):
```

Exemple pas à pas (suite)

```
    memo, compteur = {}, [0]
    fib_memo(n, memo, compteur)
    return compteur[0]

print([appels_memo(n) for n in (10, 15, 20, 25)])   # [19, 29, 39, 49]
```

Avec mémoïsation : environ 2n appels. Pour n = 25 : 49 appels contre 242 785.

À toi, avec aide

À toi, avec aide

Voici une fonction récursive :

```
def somme(liste):
    if not liste:
        return 0
    return liste[0] + somme(liste[1:])
```

1. Écris la récurrence du **nombre d'appels** T(n).
2. Quel est le nombre d'appels pour n = 5 ?
3. Remarque cachée : `liste[1:]` copie la liste. Quel est le coût total, copies incluses ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Chaque appel en fait un autre sur une liste plus courte de 1.

Indice 2

$T(0) = 1$, $T(n) = 1 + T(n-1)$.

Indice 3

La copie à l'étape k coûte environ $n-k$ éléments.

Réponse

1. $T(n) = T(n-1) + 1$, donc $T(n) = n + 1$ appels.
2. $n = 5$: 6 appels (dont l'appel sur la liste vide).
3. Les copies coûtent $(n-1) + (n-2) + \dots + 1 \approx n^2/2$: le coût total est **$O(n^2)$**, alors qu'une boucle simple est $O(n)$. La récursion n'est donc pas gratuite.

À toi, en solo

Compte le nombre d'appels pour calculer le **nombre de chemins** dans une grille n × n, en ne bougeant que vers la droite ou le bas :

```
def chemins(i, j):
    if i == 0 or j == 0:
        return 1
    return chemins(i - 1, j) + chemins(i, j - 1)
```

1. Mesure le nombre d'appels pour `chemins(n, n)` avec n = 4, 6, 8, 10.
2. Ajoute la mémoïsation (dictionnaire indexé par `(i, j)`) et compare.
3. Quel est le Big O avec mémoïsation ?

À toi, en solo · réponse

Pistes de correction

```
appels = 0

def chemins(i, j):
    global appels
    appels += 1
    if i == 0 or j == 0:
        return 1
    return chemins(i - 1, j) + chemins(i, j - 1)

def chemins_memo(i, j, memo, compteur):
    compteur[0] += 1
    if i == 0 or j == 0:
        return 1
    if (i, j) not in memo:
        memo[(i, j)] = chemins_memo(i - 1, j, memo, compteur) + chemins_memo(i, j
   - 1, memo, compteur)
    return memo[(i, j)]

for n in (4, 6, 8, 10):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    appels = 0
    r = chemins(n, n)
    c = [0]
    assert chemins_memo(n, n, {}, c) == r
    print(n, r, appels, c[0])
```

Sans mémoïsation, le nombre d'appels est de l'ordre du résultat lui-même (très grand). Avec mémoïsation : **O(n²)** cases, chacune calculée une fois.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Sous-problèmes répétés	Coût exponentiel caché	Dessiner l'arbre des appels, mémoïser si besoin
Copier à chaque appel	`liste[1:]` rend $O(n^2)$	Passer des indices au lieu de copier
Mémo partagé par erreur	Argument par défaut `memo={}` conservé entre appels	Créer le dictionnaire dans la fonction
Profondeur de récursion	`RecursionError` pour n grand	Utiliser une boucle ou augmenter la limite avec prudence

Mini-boss

Mini-boss

Rendre la monnaie (nombre minimal de pièces). Pièces `[1, 3, 4]`, montant 6 : la réponse optimale est 2 (3 + 3).

1. Écris la version récursive naïve `min_pieces(montant)` : essaie chaque pièce.
2. Compte les appels pour montant = 10, 15, 20, 25.
3. Ajoute la mémoïsation et compte à nouveau.
4. Écris la récurrence et le Big O avec et sans mémoïsation.

Mini-boss · réponse

Pistes de correction

```
appels = 0

def min_pieces(m, pieces=(1, 3, 4)):
    global appels
    appels += 1
    if m == 0:
        return 0
    meilleur = float("inf")
    for p in pieces:
        if p <= m:
            meilleur = min(meilleur, 1 + min_pieces(m - p, pieces))
    return meilleur

def min_pieces_memo(m, memo, compteur, pieces=(1, 3, 4)):
    compteur[0] += 1
    if m == 0:
        return 0
    if m not in memo:
        meilleur = float("inf")
```

Mini-boss · réponse

Pistes de correction (suite)

```
        for p in pieces:
            if p <= m:
                meilleur = min(meilleur, 1 + min_pieces_memo(m - p, memo, compteur
   , pieces))
        memo[m] = meilleur
    return memo[m]

for m in (10, 15, 20):
    appels = 0
    r = min_pieces(m)
    c = [0]
    assert min_pieces_memo(m, {}, c) == r
    print(m, r, appels, c[0])
```

Récurrence : T(m) = T(m−1) + T(m−3) + T(m−4) + 1 → exponentiel. Avec mémoïsation : chaque montant calculé une fois, avec k pièces : **O(m × k)**.

Quiz éclair

1. Pourquoi `fib(n)` naïf est-il très lent ?

A. il recalcule les mêmes sous-problèmes
B. il utilise trop de variables
C. il est écrit en Python

2. La mémoïsation consiste à :

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

3. La récurrence T(n) = T(n/2) + 1 correspond à :

A. le tri fusion

Quiz éclair

Quiz éclair (suite)

B. Fibonacci
C. la recherche binaire

4. 🔁 *Révision mission 03* : Deux boucles imbriquées de n tours donnent :

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

5. 🔁 *Révision mission 02* : $O(n \log n)$ est :

A. plus rapide que $O(n)$
B. plus lent que $O(n)$ et plus rapide que $O(n^2)$
C. identique à $O(n^2)$

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. As-tu déjà refait plusieurs fois un même calcul sans t'en rendre compte ? Dans quel contexte ?
2. Quel est l'avantage d'un schéma des appels par rapport à une formule seule ?

À retenir

À retenir

1. L'arbre des appels montre le travail total.
2. Une récurrence exprime le coût d'un algorithme récursif.
3. Sous-problèmes répétés → coût exponentiel.
4. La mémoïsation transforme l'exponentiel en polynomial.
5. La récursion a aussi un coût caché (copies, pile).

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Calculateur de profondeur

Prochaine mission : Le coût des structures de données

