ALGORITHMIQUE · FICHE PRATIQUE

Cheatsheet : Appels récursifs et cas de base

Domaine, progression, appels en attente et comparaison avec une boucle.

Par Ali ZAINOUL · Cours gratuit

Une fonction récursive s'appelle elle-même. Pour la comprendre, il faut annoncer le domaine des entrées, le cas traité sans nouvel appel et la manière de se rapprocher de ce cas. Les appels en attente ne disparaissent pas : chacun doit encore recevoir un résultat.

Nous calculons la somme des entiers de 1 à n pour de petits entiers non négatifs. Pour n = 0, la somme sans terme vaut zéro. Pour n = 3, le résultat attendu est 1 + 2 + 3 = 6.

1. Domaine, cas de base et progression

Le domaine précise les entrées autorisées. Le cas de base donne immédiatement un résultat sans nouvel appel. Le cas récursif utilise une entrée plus simple. La progression explique pourquoi les appels finiront par atteindre le cas de base.

Ici, pour n positif, sum_to(n) vaut n + sum_to(n - 1). L'argument reste entier et diminue de 1 jusqu'à zéro. Écrire un cas n == 0 ne suffit pas si l'argument évolue dans la mauvaise direction ou si le domaine n'est pas respecté.

Les trois questions avant le code

Question	Réponse pour la somme
Quelles entrées ?	Petits entiers non négatifs
Quel cas sans appel ?	n == 0 : retourner 0
Quel appel plus simple ?	sum_to(n - 1) pour n positif
Pourquoi l'atteindre ?	Un entier positif diminué de 1 rejoint zéro

2. Définir puis appeler la fonction

La définition ne calcule pas encore sum_to(3) : l'appel situé après la définition demande cette exécution. Chaque appel possède son propre paramètre n. Python transmet la valeur avec return ; MATLAB affecte le nom de sortie result.

Les exemples supposent un entier non négatif et n'effectuent pas une validation complète des types. Une fraction n'appartient pas au contrat. Les environnements limitent aussi la profondeur des appels et la représentation numérique ; ne déduisez pas une capacité illimitée de ce petit exemple.

La somme récursive jusqu’à 3

En MATLAB, enregistrez d’abord sum_to.m puis le script use_sum_to.m dans le même dossier.

Pseudo-code

```
// example.pseudo
// sum_to
FUNCTION sum_to(n)
    IF n == 0 THEN
        RETURN 0
    END IF
    RETURN n + sum_to(n - 1)
END FUNCTION
PRINT sum_to(3)
```

Python

```python
# sum_to.py
def sum_to(n):
    if n == 0:
        return 0
    return n + sum_to(n - 1)

result = sum_to(3)
print(result)
```

Le `if` traite le cas de base. Sinon, le second `return` attend le résultat de l’appel plus petit avant d’effectuer l’addition.

MATLAB

```
% sum_to.m
function result = sum_to(n)
    if n == 0
        result = 0;
        return;
    end
    result = n + sum_to(n - 1);
end
```

`return` termine seulement l'appel courant. La valeur renvoyée est celle affectée à result, pas une expression placée après `return`.

MATLAB - appel

```
% use_sum_to.m
result = sum_to(3);
disp(result);
```

> **Résultat expliqué.** Les deux langages renvoient 6 pour l'appel initial. Le retour du cas de base vaut 0, mais les appels qui l'attendent doivent encore terminer leurs additions.

3. Déplier les appels, puis remonter les résultats

La pile d'appels représente les appels commencés qui n'ont pas terminé. Dans ce parcours séquentiel, le dernier appel commencé peut finir avant ceux qui l'attendent. La descente construit des calculs en attente ; la remontée les complète.

Pour l'appel initial, 3 + sum_to(2) n'est pas encore une addition de deux nombres connus. L'appel pour 2 doit d'abord produire son résultat. Deux paramètres appelés n dans deux appels différents ne sont pas une seule variable globale réécrite.

Trace complète de sum_to(3)

Événement	Calcul ou résultat
Appel pour 3	Attendre 3 + sum_to(2)
Appel pour 2	Attendre 2 + sum_to(1)
Appel pour 1	Attendre 1 + sum_to(0)
Appel pour 0	Cas de base : résultat 0
Retour pour 1	1 + 0 = 1
Retour pour 2	2 + 1 = 3
Retour pour 3	3 + 3 = 6

4. Comparer avec une version itérative

L'itération répète un bloc à l'aide d'une boucle. Elle peut calculer ici le même résultat en conservant un total partiel. L'équivalence recherchée porte sur le résultat pour le même domaine, pas sur l'identité des traces.

La boucle suivante accumule 1, puis 2, puis 3. Elle évite les appels récursifs en attente de la version précédente. Cela ne prouve pas qu'une boucle convient à tous les problèmes : la structure du problème doit guider le choix.

La même somme avec un accumulateur

Pour n = 0, aucune valeur n’est ajoutée et le total reste zéro.

Pseudo-code

```
// example.pseudo
// sum_to_iterative
n <- 3
total <- 0
FOR value FROM 1 TO n INCLUSIVE DO
    total <- total + value
END FOR
PRINT total
```

Python

```python
# sum_to_iterative.py
n = 3
total = 0
for value in range(1, n + 1):
    total = total + value
print(total)
```

MATLAB

```
% sum_to_iterative.m
n = 3;
total = 0;
for value = 1:n
    total = total + value;
end
disp(total);
```

> **Résultat expliqué.** Le résultat vaut encore 6 ; les totaux partiels de la boucle sont 1, 3 et 6.

Erreur à éviter. Un cas de base inaccessible ne garantit pas l'arrêt. Ne testez pas une récursion potentiellement infinie sur une grande entrée pour essayer de comprendre son raisonnement.

▼ Sans exécuter de code, donnez les résultats successivement renvoyés par sum_to(2). Que se passe-t-il si le cas de base est supprimé ?

Les appels descendent jusqu'à 0 ; les résultats remontent dans l'ordre 0, 1, 3. Sans cas de base, l'argument continuerait de diminuer et le programme échouerait sur une limite de l'environnement au lieu de respecter le contrat.

Télécharger cette fiche en PDF · **Toutes les fiches par thème**

Version web : https://www.clearcode.fr/courses/programming/algorithmic/cheatsheet/functions/recursion