Devoir : Le compte est bon

UE Informatique, M1 Mathématiques et applications,
EADS, Université d'Aix-Marseille *

2024–2025

Ce devoir met en œuvre la structure d'arbre pour la représentation d'expressions arithmétiques, sur lesquelles on applique une résolution de problème par force brute.

1 Instructions

Ce devoir *fait partie de votre formation* : aller au bout devrait vous permettre à la fois de consolider certaines notions abordées dans le cours, et d'exercer vos compétences en programmation.

La section 3, fixe des tâches que votre programme doit permettre de réaliser. Vous êtes libres de choisir votre style de programmation (utiliser des fonctions, des classes avec ou sans héritage, des modules, …) et le sujet ne prescrit pas d'organisation de votre code, ni de structures de données pour la représentation des différents objets considérés. Les indications sont données en supposant que vous utilisez Python, mais vous pouvez utiliser un langage de programmation de votre choix, sous réserve de validation préalable.

Comme précedemment, vous devez documenter à la fois la manière d'utiliser votre programme pour résoudre les taches, et expliquer les choix techniques que vous avez faits. Ces explications et compléments peuvent être fournis dans les commentaires ou la documentation intégrée de votre programme, ou dans un fichier joint (soit un simple fichier texte type `README.md`, soit un document PDF).

1.1 Évaluation

Vous vous assurez d'obtenir la moyenne en produisant un code qui résout la première tâche (recherche d'une solution) même s'il plante ou rate des solutions dans certains cas atypiques, tant qu'il est exempt d'erreur grossière ou flagrante (code refusé par l'interpréteur ou le compilateur), avec une interface clairement définie (par exemple, quelle fonction appeler, ou quelle classe instancier, et avec quels paramètres pour la tâche à réaliser), et en indiquant ce qui fonctionne et ce qui ne fonctionne pas, ou que vous n'avez pas réussi à traiter.

Si de plus votre code fonctionne sans erreur (sans soulever d'exception, sans générer de boucle infinie) et produit systématiquement des résultats corrects pour la première tâche, vous obtenez au moins 12.

Si votre code résout correctement le premier objectif de la deuxième tâche (recherche de toutes les solutions), quitte à produire beaucoup de doublons vous obtenez au moins 14. De plus,

en produisant des solutions sans doublons modulo associativité et commutativité, vous obtenez au moins 16.

Avec un programme correct, efficace, bien structuré, qui répond à tous les objectifs, entièrement documenté, et avec une justification de vos principaux choix techniques : vous obtenez la note maximale.

2 Contexte

On considère une grammaire très minimaliste d'expressions arithmétiques, qui n'utilise que les entiers et les deux opérations binaires commutatives : addition (`+`) et multiplication (`×`). Une telle expression est donc un arbre binaire, dont chaque nœud interne est étiqueté par une opération, et chaque feuille est étiquetée par un entier naturel.

On veut résoudre une énigme du type *Le compte est bon* : étant donnés des entiers et une valeur cible, on veut savoir s'il existe une expression utilisant uniquement les nombres donnés et qui s'évalue en la valeur cible. Pour fixer les idées, on dira qu'une *entrée* est une liste d'entiers naturels (un mot sur $\mathbf{N}$) $(n_1, \dots, n_k)$, une *cible* est un entier naturel c, un *problème* est donné par une entrée et une cible, et une *solution* du problème est une expression arithmétique dont les feuilles sont exactement les entrées (il y a donc exactement autant de feuilles que d'entiers dans l'entrée) et dont la valeur est égale à la cible. Par exemple :

— avec l'entrée $(1, 2, 5, 10)$, on peut atteindre la cible $53 = (5 \times 10) + (1 + 2)$, ou encore $62 = ((5 + 1) \times 10) + 2$;

— avec l'entrée $(1, 1)$, les seules cibles atteignables sont $1 = 1 \times 1$ et $2 = 1 + 1$.

3 Tâches

3.1 Recherche d'une solution

Votre programme doit fournir un moyen (par exemple une fonction `résout(entrées,cible)`) pour, étant données des `entrées` (fournies par un itérable) et une `cible` (un `int`) :

— obtenir une solution `S` lorsque le problème en admet une (et alors `print(S)` doit afficher quelque chose de lisible, qui représente explicitement l'expression solution) ;

— détecter l'absence de solution sinon (par exemple en obtenant une valeur « poubelle » comme `None`, ou en soulevant une exception).

Par exemple, avec les entrées `[1,2,3]` et la cible `9` on doit obtenir une solution dont l'affichage ressemble à `(1+2)×3`. Et si la cible est `10`, votre programme doit retourner une valeur spéciale ou soulever une exception.

La recherche d'une solution peut se faire par force brute : on essaie toutes les possibilités jusqu'à trouver une solution (et on échoue si on a tout essayé sans trouver de solution).

3.2 Liste de toutes les solutions

On voudrait en fait trouver toutes les solutions possibles à un problème. Votre programme doit donc fournir un moyen (par exemple une fonction `solutions(entrées,cible)`) pour construire l'ensemble (éventuellement vide), représenté par un itérable de votre choix, des solutions d'un problème.

De plus, on voudrait éviter de répéter les solutions qui sont évidemment les mêmes. Plus précisément, on ne veut pas distinguer des solutions qui ne diffèrent que par associativité et

commutativité des opérations [1] Par exemple, avec les entrées `[2,2,3]` et la cible `12`, on voudrait obtenir seulement deux solutions, par exemple `3×2×2` et `3×(2+2)`, plutôt que la liste de toutes les manières équivalentes de parenthéser et d'ordonner ces expressions (`(3×2)×2`, `2×(3×2)`, `(2+2)×3`, *etc.*).

Deux suggestions pour vous aider dans cette tâche (ces suggestions sont optionnelles : vous pouvez aussi trouver d'autres solutions) :

— pour gérer l'associativité on peut remplacer les opérations binaires par des opérations d'arité quelconque, c'est-à-dire des nœuds avec un nombre quelconque de sous-arbres (et on assure que si la racine d'un arbre est `+`, ses sous-arbres sont des feuilles ou des `×`, et inversement) ;

— pour gérer la commutativité on peut définir une relation d'ordre total sur les expressions (et on assure que les sous-arbres d'un nœud apparaissent toujours dans l'ordre croissant).

1. On ignore volontairement la distributivité car elle change le nombre de feuilles.