ENSEIRB-Matmeca
2021-2022

1ère année - Filière Info
PG104

TD n°3 - Récursivité terminale

Exercice 1: Récursivité terminale

1. Écrire une fonction `plus` récursive terminale, qui appelée avec deux entiers a et b comme arguments, effectue la somme de a et b en utilisant l'idée que pour ajouter b, on additionne "b fois" la valeur 1.
2. Écrire une fonction `produit` récursive terminale, qui appelée avec deux entiers a et b comme arguments, effectue le produit de a par b en utilisant l'idée que $a \times b$ revient à faire $a + a + \ldots + a + a$ (b fois).
   Remarque : Le fonction `produit` est plus difficile à implémenter que la fonction `plus`. Est-il envisageable d'écrire une version récursive terminale en n'utilisant que deux paramètres ? Sinon, proposer une autre solution.

Exercice 2: Suite de Fibonacci

Dans cet exercice, on se propose de calculer les valeurs de la suite de Fibonacci, définie par la récurrence classique :

$$\begin{cases} \mathsf{fibo}(1) & = 1 \\ \mathsf{fibo}(2) & = 1 \\ \mathsf{fibo}(n) & = \mathsf{fibo}(n-1) + \mathsf{fibo}(n-2) \quad \text{lorsque } n >= 3 \end{cases}$$

On propose de comparer deux méthodes différentes :

— Méthode 1 : utiliser une récurrence double, à savoir une fonction réalisant deux appels récursifs à l'intérieur de son code.

— Méthode 2 : utiliser une fonction annexe pour réaliser le calcul avec une simple récurrence (cette fonction annexe effectue le calcul d'une suite de Fibonacci généralisée).

1. Écrire la fonction correspondant à la première méthode.
   Quelle valeur maximale de la suite peut-on calculer en un temps raisonnable ?
   Quel est l'inconvénient majeur de cette méthode ?
2. Écrire la fonction `fiboGen` qui prend 3 arguments `n`, `a` et `b` et calculant le n-ème nombre de la suite de Fibonacci commençant avec les valeurs a et b.

$$\begin{cases} \mathsf{fiboGen}(1,a,b) & = a \\ \mathsf{fiboGen}(2,a,b) & = b \\ \mathsf{fiboGen}(n,a,b) & = \mathsf{fiboGen}(n-1,a,b) + \mathsf{fiboGen}(n-2,a,b) \quad \text{lorsque } n >= 3 \end{cases}$$

3. Inventer une relation de récurrence simple sur `fiboGen`. Ecrire la fonction associée.
   Quelle propriété possède-t'elle ?

4. Discuter / estimer / comparer les complexités des deux méthodes.

Exercice 3: Trampolines

Certains compilateurs, comme la plupart des moteurs d'exécution de Javascript actuels, ne sont pas capables d'optimiser les appels récursifs terminaux. Néanmoins, il existe une technique portant le sympathique nom de *trampoline*, qui permet d'utiliser des fonctions récursives terminales sans souffrir des limites de la pile. Considérons pour commencer une fonction récursive terminale simple calculant la factorielle sur les grands nombres :

```
function fact(n, p) {
    if (n<=1n)
     return p;
    else
     return fact(n-1n, n*p);
}
```

1. Quelle est la valeur maximale du premier paramètre que l'on peut passer à cette fonction avant de recevoir un message d'erreur ?
2. Quelles sont les différences entre les expressions `fact(10n**4n, 1n)` et `() ⇒ fact(10n**4n, 1n)` ?

La technique précédente est une technique dite de *contrôle de l'évaluation*. En encapsulant un calcul dans une fonction, on peut choisir le moment où le calcul sera effectué. Dans le cadre de cet exercice, la transformation consistant à encapsuler une valeur dans un niveau de fonction sera appelé *congeler une valeur*. Les valeurs congelées deviennent des fonctions, et l'opération de *dégel* consiste simplement à appliquer la fonction (sans paramètres).

3. Écrire une nouvelle version de la fonction `fact` nommée `frozenFact` en remplaçant chaque appel récursif pas la congélation de l'appel récursif.
4. Que renvoie un appel de fonction comme `frozenFact(4, 1)` ? Comment faire pour récupérer le résultat effectif du calcul ?

La fonction ainsi construite ne fait pas le calcul elle-même, puisque elle renvoie une fonction, qui ne prend pas de paramètre. Néanmoins, une propriété importante ici est que l'appel récursif congelé n'est pas empilé sur la pile d'appel : c'est une fonction anonyme renvoyée par la fonction.

5. Comment tester en Javascript si une valeur donnée est une fonction ?
6. Écrire une fonction `trampoline` qui décongèle une valeur congelée, jusqu'à extraire la valeur à l'intérieur.
7. Pourquoi cette fonction ne *peut pas* être écrite comme une fonction récursive ?
8. Écrire une fonction `factTotal` qui peut calculer des valeurs de la factorielle pour des nombres arbitrairement grands.
9. Appliquer la même méthode pour la fonction `fibo` de l'exercice précédent.

Exercice 4: Subset Sum

Considérons le problème suivant, correspondant à une simplification du jeu de chiffres "Le compte est bon" (cf. https://fr.wikipedia.org/wiki/Des_chiffres_et_des_lettres#Le_Compte_est_Bon) si vous ne

connaissez pas ce monument de la culture française). Étant donné un ensemble de nombre positifs S (qui peuvent se répéter), ainsi qu'une valeur objectif v, est-il possible de trouver un sous-ensemble de S dont la somme vaut exactement v ?

Par exemple, si $S = \{1, 3, 3, 7\}$ et $v = 10$, alors il est possible d'obtenir $v = 3 + 7$ comme la somme de deux éléments de S. Par contre, il est impossible d'obtenir la valeur 9.

1. Poser le problème comme un problème récursif, en identifiant les sous-problèmes de même nature.
2. Écrire une fonction récursive qui, étant donné un ensemble S et une valeur objectif v, renvoie un booléen disant si cette valeur est atteignable ou pas. On peut supposer que l'ensemble S est ordonné de manière décroissante.

Une manière d'optimiser les calculs faits dans ces fonctions consiste à les mémoriser quelque part une fois qu'ils sont faits. Pour cela, on propose de construire un dictionnaire stockant les valeurs :

```
const memo = {};
```

Le dictionnaire en question va avoir pour clés les paires (S, v), et pour valeur les résultats de la fonction. Comme on ne peut utiliser un tableau comme une clé, il faut ruser et les transformer en chaîne de caractères :

```
const set = [7,3];
const v   = 10;
const key = `${v}[${set}]`;
memo[key] = true;
memo; // → { '10[7,3]': true }
```

▷ La transformation qui consiste à créer des clés dans un tableau sous la forme de chaînes de caractère a un coût non nul (d'autant plus qu'il est fait à chaque appel), qui peut éventuellement dépasser celui du calcul lui-même. Ici, on fait le choix d'une solution courte et simple. En pratique, on préférera *hacher* les clés, soit à la main, soit en utilisant une table de hachage comme `Map`.

3. Écrire une nouvelle version de votre fonction qui :
   - avant de commencer un calcul, vérifie s'il n'est pas déja dans `memo` ;
   - au moment de renvoyer un résultat, le stocke dans le dictionnaire `memo`.
4. Utiliser une fermeture pour enfermer le dictionnaire `memo` et le rendre inaccessible à l'extérieur de votre fonction.
5. (Bonus) Renvoyer le sous-ensemble dont la somme vaut v lorsqu'il existe. On pourra envisager une version sans optimisation par dictionnaire et une avec.