ENSEIRB-Matmeca
2024-2025

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 ?

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 intermédiaire pour réaliser le calcul avec une simple récurrence ; la fonction intermédiaire 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: Contrôle de l'évaluation - 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 deux expressions suivantes :
   (i) `fact(10n**4n, 1n)` et
   (ii) `() ⇒ fact(10n**4n, 1n)` ?

▷ La technique consistant à *encapsuler* un calcul dans une fonction ne prenant aucun paramètre est une technique dite de *contrôle de l'évaluation*. Comme le calcul se fait au sein d'une fonction, il devient possible de choisir le moment où il sera effectué.

Dans 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* (*thaw* en anglais) consiste simplement à appliquer la fonction (toujours sans paramètres).

3. Écrire une nouvelle version de la fonction `fact` nommée `frozenFact` en remplaçant chaque appel récursif par 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 simplement une fonction. 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.
   Pourquoi cette fonction ne *peut pas* être écrite comme une fonction récursive ?

▷ La fonction `trampoline` ainsi construite permet de faire des calculs récursifs en plaçant ses états intermédiaires dans le tas (du point de vue de la mémoire). Elle n'est pas donc contrainte par les limites de taille de la pile.

7. Écrire une fonction récursive `factTotal` qui peut calculer des valeurs de la factorielle pour des nombres arbitrairement grands.
8. Appliquer la même méthode pour la fonction `fibo` de l'exercice précédent.

Exercice 4: Cutting stock

Le problème de la découpe optimale (cf. par exemple `https://en.wikipedia.org/wiki/Cutting_stock_problem`) est un problème combinatoire consistant à se donner une longueur L d'un matériau quelconque pouvant se découper en multiples de l'unité (du tissu, du papier, du chocolat ...), ainsi qu'un tableau associant une valeur *positive* à chaque longueur de ce matériau.

Par exemple, supposons disposer de 8 carrés de chocolat, et que la revente de ce chocolat par nombre de carrés rapporte les sommes suivantes :

Longueur	0	1	2	3	4	5	6
Valeur	0	1	5	8	9	10	17

À ce moment, il est possible de se convaincre que le gain maximal pour 8 unités est de 22. Il est obtenu en vendant d'une part 2 carrés (pour 5) et d'autre part 6 carrés (pour 17).

Comparativement, si les gains étaient les suivants (on ne change que la valeur du gain à l'unité) :

Longueur	0	1	2	3	4	5	6
Valeur	0	3	5	8	9	10	17

Alors il est possible d'obtenir une gain de 24 en vendant chaque carré à l'unité (pour 8×3).

Remarque : pour que le problème soit bien posé, il est nécessaire que le gain pour 0 unités soit égal à 0. Sinon, le gain maximal est évidemment infini.

▷ Les valeurs sont stockées dans un tableau de la manière suivante :

```
const gains = [0, 1, 5, 8, 9, 10, 17];
```

Cela assure que l'on puisse avoir la propriété `gains[`*longueur*`] = `*valeur* pour les longueurs qui sont des indices du tableau. Néanmoins, on peut aussi demander la valeur d'une longueur dépassant la taille du tableau. Pour accéder à cette valeur, on se propose d'utiliser la fonction suivante :

```
// Return the price if any, otherwise 0
function getP(prices, i) { return (i < prices.length) ? prices[i] : 0; }
```

Cela assure qu'on ait la propriété `getP(gains,` *longueur*`) = `*valeur* pour toute *longueur* ≥ 0.

1. Décrire le problème de la découpe optimale comme un problème récursif, en identifiant les sous-problèmes de même nature.
2. Écrire une fonction `maxGen(f, a, b)` qui renvoie le maximum de l'ensemble des valeurs prises par la fonction `f` dans l'intervalle $[a; b[$, et `undefined` si cet intervalle est vide.

```
maxGen((i) ⇒ i*i, -5, 5); // → 25
maxGen((i) ⇒ i*i, 5, -5); // → undefined
```

3. Écrire la fonction `cuttingStock` calculant le gain optimal.

```
cuttingStock([0, 1, 5, 8, 9, 10, 17],8); //→ 22
cuttingStock([0, 3, 5, 8, 9, 10, 17],8); //→ 24
```

4. Proposer une manière de se rendre compte du nombre d'appels récursifs réalisés par la fonction.

Une manière d'optimiser les calculs faits dans la fonction `cuttingStock` consiste à mémoriser les résultats des calculs intermédiaires dans une structure de donnée une fois qu'ils sont faits (technique de *mémoïsation*). Pour cela, on propose de construire un dictionnaire stockant les valeurs :

```
const memo = { 0: 0 }; // i.e the cost for length 0 is 0
```

5. Écrire une fonction qui, à chaque fois qu'elle calcule une découpe optimale pour un prix donné, stocke le résultat dans `memo` avant de le renvoyer
6. Finaliser la fonction de manière à ce que `memo` ne soit pas accessible de l'extérieur de la fonction.