Feuille d'exercices n°7 - Étude de fonctions récursives

Notions abordées

- méthode pour étudier la complexité d'une fonction récursive
- importance de limiter les appels récursifs
- méthode pour montrer la correction d'une fonction récursive
- arbre des appels récursifs, étapes de calcul

Étude de complexité

Exercice 1 Suite de Fibonacci

On considère les deux fonctions suivantes codées en Ocaml.

```
let rec fibo1 (n:int) : int =
(*hyp : nn >= 0
calcule le terme de rang n de la suite de Fibonnacci*)
        if n = 0 then 0
        else if n = 1 then 1
        else fibo1 (n-1) + fibo1 (n-2)


let fibo2 (n:int) : int =
(*hyp : nn >= 0
calcule le terme de rang n de la suite de Fibonnacci*)
        let rec aux (a:int) (b:int) (nn:int) : int =
        (* hyp : nn >= 0
        calcule le n-ième terme de la suite définie par
        u_0=a, u_1=b et pr k\in N, u_(k+2)=u_(k+1)+u_k *)
                if nn =0 then a
                else if nn=1 then b
                else aux b (a+b) (nn-1)
        in aux 0 1 n
```

Question 1 ✍
Identifier les fonctions récursives et pour chacune dire si elle est récursive terminale.

Question 2 ✍
Tracer l'arbre des appels récursifs réalisés lors de l'appel `fibo1 4`. Même question pour `fibo2 4`.

Question 3 ✍
Donner l'évolution de l'écriture de `fibo1 4` lors de son évaluation. Même question pour `fibo2 4`.

Question 4 ✍
Pour $n \in \mathbb{N}$, on note u_n (resp. v_n) le nombre d'appels récursifs réalisé lors de l'appel `fibo1 n`. (resp. `fibo2 n`). Donner une définition récursive de u (resp. de v). Décrire le comportement asymptotique de u (resp. de v) à l'aide d'un Θ.

Question 5 ✍
Conclure sur l'efficacité de ces fonctions. Pour une fonctions récursive, quel est le lien entre la complexité et le fait qu'elle soit récursive terminale ?

Exercice 2 Multiplication

On considère les trois fonctions suivantes codées en Ocaml.

```
let mult1 (a:int) (b:int) : int =
(* hyp  a>=0, calcule a fois b*)
        let rec aux (aa:int) (res:int) : int =
                if aa = 0 then res
                else aux (aa-1) (res+b)
        in aux a 0

let rec mult2 (a:int) (b:int) : int =
(* hyp  a>=0, calcule a fois b*)
        if a = 0 then 0
        else if a mod 2 = 0
                then (mult2 (a/2) b) + (mult2 (a/2) b)
                else b + (mult2 (a/2) b) + (mult2 (a/2) b)

let rec mult3 (a:int) (b:int) : int =
(* hyp  a>=0, calcule a fois b*)
        if a = 0 then 0
        else let r = mult2 (a/2) b in
                if a mod 2 = 0
                then r+r
                else b+r+r
```

Question 1 ✍
Identifier les fonctions récursives et pour chacune dire si elle est récursive terminale.

Question 2 ✍
Tracer l'arbre des appels récursifs réalisés lors de l'appel `mult1 3 4`. Même question pour `mult2 3 4` et `mult3 3 4`.

Question 3 ✍
Donner l'évolution de l'écriture de `mult1 3 4` lors de son évaluation. Même question pour `mult2 3 4` et `mult3 3 4`.

Question 4 ✍
On remarque que les nombres d'opérations élémentaires et d'appels récursifs effectués lors d'un appel

à `mult1`, `mult2` ou `mult3` ne dépendent pas de la valeur du deuxième argument. On note donc, pour $n \in \mathbb{N}$, u_n (resp. v_n et w_n) le nombre d'appels récursifs réalisé lors d'un appel de la forme `mult1 n b`. (resp. `mult2 n b` et `mult3 n b`) pour $b \in \mathbb{Z}$. Donner une définition récursive de u (resp. de v et w). Décrire le comportement asymptotique de u (resp. de v et w) à l'aide d'un Θ.

Question 5 ✍
Conclure sur l'efficacité de ces fonctions. Pour une fonctions récursive, quel est le lien entre la complexité et le fait qu'elle soit récursive terminale ?

Question 6 ✍ ou 🖳
Proposer une version récursive terminale de la multiplication à partir de la fonction de meilleure complexité parmi `mult1`, `mult2` et `mult3`.

Preuve de correction

Exercice 3 Factorielle

Question 1 ✍
Justifier que la fonction `fact1` termine. Même question pour `fact2`.

Question 2 ✍
Démontrer que `fact1` calcule bien la factorielle de l'entier qu'elle prend en argument pourvu qu'il soit positif. Même question pour `fact2`

Exercice 4 Mystère

On considère la fonction suivante codée en Ocaml.

```
let rec mystere (k:int) (n:int) : int =
(* hyp  k<=n, calcule .... *)
        if (k = n || k = 0) then 1
        else mystere (k-1) (n-1) + mystere (k) (n-1)
```

Question 1 ✍
Justifier que les appels récursifs à la fonction `mystere` sont valides, c'est-à-dire que la fonction est appelée sur des arguments qui vérifient les hypothèses.

Question 2 ✍
Justifier que la fonction `mystere` termine.

Question 3 ✍
Donner l'évolution de l'écriture de `mystere 2 4` lors de son évaluation.

Question 4 ✍
Que calcule `mystere` ? Donner une description (brève et précise).

Question 5 ✍
Démontrer que `mystere` est correcte.

Question 6 ✍

Tracer l'arbre des appels récursifs réalisés lors de l'appel `mystere 2 4`. Que remarque-t-on ? Comment améliorer la complexité de cette fonction ?