Algorithmique des structures de données arborescentes
Feuille d'exercices 2

1 Preuves de propriétés par récurrence

Exercice 2.1 On note $h(t)$ la hauteur et $n(t)$ le nombre de nœuds d'un arbre binaire non vide t. Montrer par récurrence sur un paramètre bien choisi que :

a) tout arbre binaire non vide a au moins une feuille.

Note. Cette question est corrigée dans le polycopié et le diaporama. Il est conseillé de lire et comprendre la rédaction.

b) Montrer précisément les trois propriétés de l'exercice 1.13 de la feuille 1.

c) Montrer précisément la propriété de l'exercice 1.14 de la feuille 1.

Exercice 2.2 La suite de Fibonacci est une suite d'entiers définie par

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_{n+2} = f_{n+1} + f_n \text{ pour } n \geqslant 0. \end{cases}$$

1. Montrer par récurrence que pour $n \geqslant 1$, on a $f_n \geqslant \frac{1}{2}\sqrt{2}^n$.

   Attention au cas de base : il n'est pas suffisant de montrer l'inégalité seulement pour $n = 1$.

On considère l'algorithme suivant (écrit en langage OCaml) pour calculer f_n :

```
let rec fib n =
if n <= 0 then 0
else if n = 1 then 1
else fib(n-1) + fib(n-2)
```

2. Écrire l'arbre d'appels récursifs sur l'appel `fib 5`.
3. On note $c(n)$ le nombre d'appels effectués lorsqu'on évalue `fib n` (y compris le premier appel sur l'argument n). Établir une relation de récurrence vérifiée par la suite $c(n)$.
4. Déduire des questions 1 et 3 une minoration du nombre d'appels à `fib` effectués par `fib n`.
5. Écrire une version de l'algorithme de complexité $O(n)$, où n est la *valeur* de l'argument. On considère que toutes les opérations arithmétiques prennent un temps $O(1)$.

Le produit de matrices 2×2 est défini par la formule :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} aa' + bc' & ab' + bd' \\ ca' + dc' & cb' + dd' \end{pmatrix}$$

6. Combien d'opérations arithmétiques suffisent-elles pour calculer ce produit ?

Soit F la matrice $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$.

7. Montrer par récurrence sur n que pour tout $n \geqslant 1$, la puissance $n^{\text{ème}}$ de F est

$$F^n = \begin{pmatrix} f_{n+1} & f_n \\ f_n & f_{n-1} \end{pmatrix}$$

8. On rappelle les formules permettant de calculer rapidement une puissance (exponentiation rapide) :

$$F^n = \begin{cases} (F^2)^{n/2} & \text{si } n \text{ est pair,} \\ F \cdot F^{n-1} & \text{si } n \text{ est impair.} \end{cases} \tag{2.1}$$

   En utilisant (2.1), donner un algorithme effectuant $O(\log(n))$ opérations arithmétiques pour calculer f_n.

9. Pourquoi peut-on dire que cet algorithme est de complexité *linéaire*, et non *logarithmique* ?

2 TD machine

Exercice 2.3 1. Écrire en OCaml la fonction `fib` de l'exercice 2.

2. Jusqu'à quelle valeur de n cet algorithme permet-il de calculer f_n en moins de 1mn ?

3. La fonction `Sys.time()` retourne le temps CPU utilisé par le programme en cours (voir la documentation). En utilisant cette fonction, écrire une fonction `timer` de type `int -> float` qui calcule le temps processeur consommé sur l'appel à `fib n` .

   Rappel la soustraction de flottants est `-.` .

4. Calculer `timer(`$n+1$`)/.timer` n pour n valant 20, 30, 40. Comparer avec le résultat théorique de l'exercice 2.

5. Écrire la version en $O(n)$ et vérifier qu'elle permet de calculer f_{100}.

6. Écrire la version en $O(\log n)$. **Note.** Pour calculer des valeurs de f_n pour de grands entiers n, il faut utiliser une bibliothèque comme `Num` ou `Zarith` .

Dans les exercices suivants, on utilise le type `'a tree` suivant :

```
1   type 'a tree = Empty | Bin of 'a * 'a tree * 'a tree
```

Exercice 2.4 1. Écrire une fonction `size` de type `'a tree -> int` calculant la taille d'un arbre.

2. Écrire une fonction `height` de type `'a tree -> int` qui calcule la hauteur d'un arbre.

3. Écrire une fonction `at_depth` de type `'a tree -> int -> 'a list` qui renvoie la liste des étiquettes des nœuds se trouvant à profondeur donnée dans un arbre, lus de gauche à droite.

Exercice 2.5 Écrire une fonction `are_equal` de type `'a tree -> 'a tree -> bool` qui teste si deux arbres sont égaux. **Note.** L'opérateur `=` d'OCaml répond à la question, mais on demande de le ré-écrire.

Exercice 2.6 Écrire une fonction `mem` de type `'a -> 'a tree -> bool` qui teste si une valeur apparaît comme étiquette dans un arbre.

Exercice 2.7 Écrire une fonction `is_full` de type `'a tree -> bool` testant si un arbre binaire est plein.

Exercice 2.8 1. Écrire une fonction `is_perfect` de type `'a tree -> bool` testant si un arbre est parfait.

2. Évaluer la complexité de la fonction écrite.

3. Si la fonction n'est pas de complexité linéaire, en écrire une version linéaire.

Exercice 2.9 Écrire une fonction `is_quasi_perfect` de type `'a tree -> bool` testant si un arbre binaire est quasi-parfait.