Récursivité

Exercice 1 *Calcul des termes d'une suite définie par récurrence*

On définit la suite $(u_n)_{n\in\mathbb{N}}$ par $u_0 = u_1 = u_2 = 0$ et pour tout $n \geqslant 3$,

$$u_n = u_{n-1} + u_{n-2} \cdot u_{n-3} + 1.$$

On considère la fonction suivante :

```
def f(n):
  if n <= 2:
    return 0
  else:
    return f(n-1)+f(n-2)*f(n-3)+1
```

1. Montrer que la fonction `f(n)` termine sur tout entier naturel n et retourne u_n.
2. On note a_n le nombre d'appels de la fonction `f` quand on appelle `f(n)` (on compte l'appel principal fait à `f(n)`, et tous les autres appels faits au cours du processus récursif). Ainsi, $a_0 = a_1 = a_2 = 1$. Dessiner l'arbre des appels récursifs quand on lance `f(3)`, et faire de même pour `f(4)`. Que valent a_3 et a_4 ?
3. Exprimer a_n en fonction a_{n-1}, a_{n-2} et a_{n-3} pour $n \geqslant 3$.
4. Montrer une borne inférieure exponentielle sur a_n. En déduire une borne inférieure sur le temps d'exécution de cet algorithme.
5. Proposer un algorithme plus efficace pour calculer u_n et donner sa complexité.

Exercice 2 *Exponentiation rapide*

Le but de cet exercice est de voir une méthode rapide de calcul d'une puissance.

1. Écrire un algorithme récursif qui étant donné un nombre a et un entier naturel n, calcule a^n en se basant sur le fait que pour $n \geqslant 1$, $a^n = a \cdot a^{n-1}$. Combien fait-on de multiplications ?
2. Si q et r sont les quotient et reste de la division euclidienne de n par 2, on a
$$a^n = a^{2q+r} = (a^2)^q \cdot a^r.$$
Écrire un algorithme récursif calculant l'exponentiation basée la remarque ci-dessus permettant de faire moins de multiplications qu'à la question précédente.
3. Donner une borne sur le nombre de multiplications effectuées par cet algorithme.
4. Soit $m \geqslant 2$ fixé. Si q et r sont les quotient et reste de la division euclidienne de n par m, on a
$$a^n = a^{mq+r} = (a^m)^q \cdot a^r.$$
En déduire un nouvel algorithme récursif calculant l'exponentiation. Comparer à l'algorithme de la question 2.

Exercice 3 *La suite de Fibonacci*

La suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ est définie par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \text{ pour } n \geqslant 0 \end{cases}$$

On a vu en cours que l'algorithme récursif découlant directement de la traduction de la relation de récurrence mène à un algorithme faisant un nombre exponentiel d'additions. Nous avons vu ensuite qu'il est facile de calculer F_n avec un nombre linéaire d'additions. Le but de cette partie est de voir comment calculer F_n avec encore moins d'opérations arithmétiques.

La méthode est basée sur la remarque suivante : pour $n > 0$, on a la relation suivante :

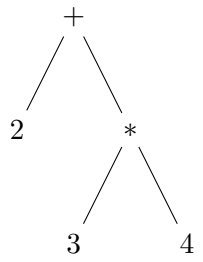
$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}.$$

1. En utilisant la relation ci-dessus, exprimer $\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$ en fonction de $\begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$.
2. Adapter l'algorithme d'exponentiation rapide pour calculer la puissance n d'une matrice M de taille 2×2. (On supposera qu'on dispose d'une fonction calculant le produit de deux matrices de tailles 2×2.)
3. En déduire un nouvel algorithme calculant F_n.
4. Combien d'opérations arithmétiques fait ce nouvel algorithme ?

Exercice 4 *Évaluation d'une formule arithmétique*

On considère des formules arithmétiques complètement parenthésées utilisant les entiers et les opérations $+$ et $*$ (ces opérations prennent exactement deux entrées).

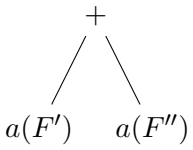
Par exemple $(2 + (3 * 4))$ est une expression arithmétique. Il est naturel d'associer à une expression arithmétique bien parenthésée un arbre. Par exemple l'arbre associé à l'expression arithmétique $(2 + (3 * 4))$ est :



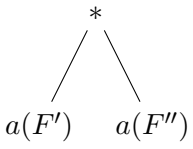
Définissons plus formellement l'arbre $a(F)$ associé à une expression arithmétique F. L'arbre associé à une expression arithmétique est défini récursivement de la façon suivante :

— si F est un entier, $a(F)$ est l'arbre réduit à la feuille F ;

— si F est de la forme $(F' + F'')$, $a(F)$ est l'arbre :



— si F est de la forme $(F' * F'')$, $a(F)$ est l'arbre :



Une expression arithmétique s'évalue de manière naturelle en un nombre entier en effectuant les opérations arithmétiques : ainsi, l'expression arithmétique $(2 + (3 * 4))$ s'évalue en 14.

1. Dessiner l'arbre correspondant à la formule arithmétique $(2 + ((1 + 4) * 3))$.

On considère maintenant des algorithmes prenant en entrée des arbres codant des expressions arithmétiques. On suppose qu'on a les fonctions élémentaires suivantes sur les arbres :

— la fonction `estFeuille(A)` qui renvoie `True` si l'arbre A est réduit à une feuille et `False` sinon ;

— la fonction `racine(A)` qui retourne l'entier étiquetant l'unique noeud de A si A est une feuille, l'opération `'+'` ou `'*'` étiquetant la racine sinon ;

— les fonctions `filsGauche(A)` et `filsDroit(A)` retournant les sous-arbres gauche et droit de A.

2. Écrire un algorithme `evaluation(A)` qui évalue une formule arithmétique.
3. Démontrer que votre algorithme est correct.
4. Déterminer la complexité dans le pire des cas de l'algorithme `evaluation(A)`. On comptera ici un coût $O(1)$ pour une opération arithmétique sur les entiers.