TP2 : récursivité

AP3 Algorithmique et programmation — L2 mathématiques

29 septembre 2016

Pour ce TP, on aura besoin de mesurer le *temps d'exécution* de certains programmes. Pour cela, on va utiliser la fonction `clock` de la bibliothèque `time` de `python` qui renvoie le *temps processeur*[1] en secondes. Pour comprendre son fonctionnement, faites tourner le petit programme suivant (n'hésitez pas faire des modifications et à en observer les conséquences) :

```
def waste_time(): #a pure waste of time
  i = 0
  while(i<10000000): i += 1
  return None

import time
start = time.clock() #get a starting point
waste_time()
print (time.clock() - start) #print the delay from start
```

Exercice 1 (Suite de Fibonacci)**.** La suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ se définit de la façon suivante :

$$F_0 = 0,\ F_1 = 1 \quad \text{et} \quad \forall n \geq 0,\ F_{n+2} = F_{n+1} + F_n$$

(a) **Algorithme naïf**

i. En exploitant directement la définition **récursive** de F_n, écrire une fonction `fib1(n)` prenant en paramètre un entier n et renvoyant F_n.

ii. Dans le calcul de F_n, combien de fois cet algorithme calcule-t-il F_1 ? Et F_k, pour $k < n$?

iii. Mesurer le temps d'exécution de `fib1(18)` puis de `fib1(30)`.

(b) **Algorithme linéaire**

i. Écrire une fonction `fib2(n)` prenant en paramètre un entier n et renvoyant F_n en ne calculant chaque F_k, pour $k < n$ qu'une seule fois. Indice : on pourra construire récursivement une liste contenant $(F_1, F_2, \dots, F_n)$.

ii. Combien d'itérations cet algorithme utilise-t-il ?

iii. Mesurer le temps d'exécution de `fib2(18)` puis de `fib2(30)`.

1. C'est plus ou moins le temps effectif de calcul par le processeur depuis le début du programme.

(c) **Algorithme logarithmique** (*bonus*)
La suite de Fibonacci a la propriété suivante : pour tout $k \geq 1$

$$F_{2k} = (2F_{k-1} + F_k)F_k$$
$$F_{2k+1} = F_{k+1}^2 + F_k^2$$

Écrire une fonction `fib3(n)` prenant en paramètre un entier n et renvoyant F_n, dont l'exécution est plus rapide que l'algorithme linéaire. N'hésitez pas à tester les deux derniers algorithmes sur des valeurs très grandes ($n = 1000$), le type `long` en `python` n'ayant pas de taille maximale.

Remarque. La méthode naïve sur cette nouvelle formule fait trop d'appels récursifs. Écrire une fonction `fib3aux(n)` prenant en paramètre un entier n et renvoyant le tuple (F_{n-1}, F_n) qui fait un unique appel récursif.

Exercice 2 (Coefficients binomiaux)**.**

(a) Écrire une fonction `fact(n)` prenant en paramètre un entier n et renvoyant $n\,!$.

(b) En déduire une fonction `binom1(n,p)` prenant en paramètres deux entiers n et p et renvoyant le coefficient binomial $\binom{n}{p} = \frac{n!}{(n-p)!p!}$. Mesurer son temps d'exécution sur quelques exemples.

(c) Écrire une fonction `binom2(n,p)` prenant en paramètre n et p, deux entiers et renvoyant le coefficient binomial $\binom{n}{p}$ en utilisant la formule de Pascal :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

Comparer son temps d'exécution avec celui de `binom1` sur les exemples précédents.

(d) Écrire une fonction `pascal_triangle(n)` prenant en paramètre un entier n et affichant les valeurs de tous les $\binom{i}{j}$, $j \leq i \leq n$ sous la forme d'un triangle de Pascal. Par exemple, `pascal_triangle(6)` devra afficher :

```
0:      1
1:      1       1
2:      1       2       1
3:      1       3       3       1
4:      1       4       6       4       1
5:      1       5       10      10      5       1
6:      1       6       15      20      15      6       1
```

Exercice 3 (Décomposition dans une base)**.**

(a) Écrire un algorithme qui décompose en binaire un nombre écrit en décimal.

(b) Même question pour le changement de base de décimal vers hexadécimal.

Exercice 4 (PGCD)**.**

(a) Écrire une fonction qui prend en entrée deux nombres a et b dans $\mathbb{N}$ et renvoie le pgcd de a et de b.

(b) Améliorer la fonction précédente pour qu'elle renvoie à la fois $\text{pgcd}(a, b)$ et les entiers (dits de Bézout) $u, v \in \mathbb{Z}$ tels que :

$$ua + vb = \text{pgcd}(a, b)$$

Exercice 5 (Théorème de restes chinois)**.** Soient $n_1, \dots, n_k$ des entiers deux à deux premiers entre eux (i.e. tels que $\text{pgcd}(n_i, n_j) = 1$ pour tout couple $i, j \leq k$ avec $i \neq j$). On sait, par le *théorème des restes chinois* que, quelques soient $a_1, \dots, a_k \in \mathbb{N}$, il existe un unique entier $x \leq \prod_{i=1}^{k} n_i$ tel que :

$$\begin{cases} x = a_1 \pmod{n}_1 \\ \quad \vdots \\ x = a_k \pmod{n}_k \end{cases}$$

(a) Écrire une fonction qui prend en entrée une liste de couples d'entiers `(`a_i`,`n_i`)` et qui renvoie l'unique entier x comme ci-dessus.

(b) Écrire un programme qui :

- demande le nombre d'équations,
- puis demande à l'utilisateur les équations une à une sous la forme d'une chaîne du type `x = a % n`.

Exercice 6 (Le jeu des allumettes)**.** Le jeu des allumettes est un jeu à deux joueurs. Un nombre $n \geq 1$ d'allumettes sont disposés sur une table. Chaque joueur, appelés `un` et `deux`, doivent tour à tour enlever de une à trois allumettes. Celui qui prend la dernière allumette a perdu. C'est le joueur `un` qui commence.

On dit qu'un joueur a une *stratégie gagnante* pour n allumettes s'il a une façon de jouer qui lui permet de gagner quelques soient les réponses de l'autre joueur. Si pour chaque n, un des deux joueurs a une stratégie gagnante, on dit que le jeu est *déterminé*.

(a) Réfléchir quelques minutes pour se convaincre que ce jeu est déterminé. Indice : traitez d'abord le cas où il y a $n \leq 4$ allumettes, puis essayez de généraliser.

(b) Montrer que si $n \neq 1 \pmod 4$, alors le joueur `un` a une stratégie gagnante. Montrer que dans le cas où $n = 1 \pmod 4$, c'est le joueur `deux` qui a une stratégie gagnante.

(c) Écrire deux fonctions `un(n)` et `deux(n)` telles que :

- `un(n)` teste si $n \neq 1 \pmod 4$ et si c'est le cas, choisit le nombre r d'allumettes à enlever. Si ce n'est pas le cas, elle choisit un nombre aléatoire r entre 1 et 3. À l'issue, elle appelle la procédure `deux` sur l'entrée $n - r$.
- `deux(n)` demande à l'utilisateur d'entrer un nombre entre 1 et 3 et appelle `un` sur l'entrée $n - r$.

(d) Utiliser les fonctions précédentes pour écrire un programme permettant de réaliser le jeu des allumettes sur un entier n donné entre un joueur "humain" (le joueur `deux`) et l'ordinateur (le joueur `un`).

À rendre pour le 06 octobre 2016

Exercice 7 (Tours de Hanoï)**.** Le jeu des tours de Hanoï consiste en un plateau de 3 bâtons appelés "tours", que l'on note T_1, T_2, T_3, surlequels viennent s'empiler n disques de diamètres tous différents. La position initiale des disques est la suivante : ils sont tous sur la tour T_1 empilés du plus grand (tout en bas) au plus petit (tout en haut). Le but de jeu est de reproduire la même situation en T_3 en suivant les règles suivantes :

- on ne peut déplacer qu'un disque qui est au-dessus d'une des piles,
- on ne peut déplacer qu'un disque à la fois,
- un disque ne peut pas s'empiler sur un disque dont le diamètre est strictement inférieur au sien.

Sur papier, déterminer une stratégie pour résoudre le problème de tours de Hanoï à $n \geq 1$ disques. (N'oubliez pas le titre de ce TP...)

Écrire une fonction `hanoi(start,stop,inter,n)` prenant en paramètres le numéro de la tour de départ `start`, celui de la tour d'arrivée `stop`, celui de la tour intermédiaire `inter`, et le nombre de disques `n` et qui renvoie la liste des coups à effectuer pour résoudre le problème des tours de Hanoï. Un coup sera stocker sous la forme d'un couple (i, j) où i est le numéro de la tour de départ du coup et j celui de la tour d'arrivée.

Exemple : pour résoudre le problème des tours de Hanoï à 2 disques, on appelle

```
hanoi(1,3,2,2)
```

ce qui doit retourner la liste `[(1,2),(1,3),(2,3)]`.