TP3 : premier pas en algorithmique

AP3 Algorithmique et programmation — L2 mathématiques

6 octobre 2016

Exercice 1 (Arrêt d'algorithme ?)**.** Écrire une fonction `fun(i,n)` qui prend en argument deux nombres i et n et qui rend la liste des i premières itérations de la fonction :

$$f : n \mapsto \begin{cases} 1 & \text{si } n = 1 \\ f(n/2) & \text{si } n \text{ est pair} \\ f(3n+1) & \text{sinon} \end{cases}$$

Par exemple, `fun(4,3)` retourne `[3,10,5,16]`. Calculer les valeurs de $f(n)$ pour $n \in [0, 20]$. Que constate-t-on ?

Exercice 2 (Primalité)**.** Le but de cet exercice est d'écrire un algorithme efficace générant les nombres premiers de l'intervalle $[0, n]$ pour $n > 0$ donné. On utilise pour cela le crible d'Eratosthène : pour chaque $2 \leq k \leq n$, on sait certainement que tous les multiples de k ne sont pas premiers, et on peut donc les éliminer. En itérant ce processus, on obtient à la fin une liste des nombres premiers inférieurs à n.

Écrire une fonction `sieve(n)` qui implémente le crible d'Eratosthène. Au choix, on pourra coder une fonction récursive ou itérative. Estimer la complexité de l'algorithme. Indice : on pourra utiliser la formule suivante

$$\sum_{p \text{ premier}, p \leq n} \frac{1}{p} = \mathcal{O}(\ln n \ln n)$$

Remarque. Cet algorithme sert à générer **tous** les nombres premiers inférieurs à une borne donnée. Ce n'est pas du tout un algorithme efficace pour tester si un **unique** nombre est premier en général.

Exercice 3 (Nombres de Perrin)**.**

(i) Écrire une fonction `perrin(n)` calculant *rapidement* la fonction :

$$\text{per}(n) = \begin{cases} \text{per}(0) = 3 \\ \text{per}(1) = 0 \\ \text{per}(2) = 2 \\ \text{per}(n) = \text{per}(n-2) + \text{per}(n-3) & \text{sinon} \end{cases}$$

(ii) La conjecture de Perrin (1899) dit que n est premier si et seulement si n divise per(n). Vérifiez algorithmiquement cette conjecture.

À rendre pour le 20 octobre 2016

Exercice 4 (Jeu des allumettes)**.** Le jeu des allumettes est un jeu à deux joueurs. Un nombre $n \geq 1$ d'allumettes sont disposées sur une table. Chacun des joueurs, appelés `un` et `deux`, doit 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`).

Exercice 5 (Exponentielle modulaire)**.** En se basant sur l'observation suivante :

$$\forall e \in \mathbb{N}, \forall a \in \mathbb{N},\ a^e = \begin{cases} a^k \times a^k & \text{si } e = 2k \\ a \times a^k \times a^k & \text{si } e = 2k+1 \end{cases}$$

Écrire une fonction `exp_mod(a,e,n)` qui renvoie $a^e \pmod n$.

Exercice 6 (Test de Fermat)**.** Rappelons que le petit théorème de Fermat assure que si p est premier, alors $\forall a \in [1, p-1],\ a^{p-1} = 1 \pmod p$. Cela nous donne un moyen bien pratique de vérifier si un nombre n est composé (c'est-à-dire non premier) : on tire au hasard $a \in [1, n-1]$ et calcule $a^{n-1} \mod n$; si l'on obtient pas 1, c'est que le nombre n est composé (par la contraposée du théorème de de Fermat) ; sinon, c'est qu'il est possiblement premier et on dit que n *passe le test de Fermat*. En répétant ce test plusieurs fois, on obtient un algorithme qui sait à coup sûr quand un nombre est composé et qui sait probablement quand un nombre est premier.

Écrire une fonction `fermat_test(n,k)` qui répète k fois le test de Fermat exposé ci-dessus sur le nombre n et retourne `True` si n est composé et `False` si n passe les k tests de Fermat.

Indice : n'oubliez pas que vous avez codé une fonction `exp_mod`.

Exercice 7 (Pseudo premiers)**.** Le test de Fermat est assez fiable en pratique mais il souffre d'un gros souci : certains nombres, bien que rares, passent le test de Fermat pour quasiment toutes les valeurs de a ! Plus précisément, il existe une infinité de nombres n composé tel que pour tout $a \in [1, n-1]$ avec pgcd$(a, n) = 1$ on a $a^{n-1} = 1 \pmod{n}$. C'est-à-dire que vous auriez beau réitérer le test de Fermat, si vous n'avez pas la chance de tomber sur un a non premier à n, la fonction `fermat_test` répondra toujours `False`, vous menant ainsi à croire que n est très probablement premier... On appelle un tel n pseudo premier.

Écrire un programme qui trouve les 20 premiers nombres pseudo premiers.