TP 6 : écriture en base deux et exponentiation rapide

1 Ecriture en base deux

On va écrire un algorithme itératif (boucle `while`) permettant d'obtenir l'écriture en base deux d'un nombre entier n.

Deux méthodes sont possibles.

(M1) Des poids forts vers les poids faibles : Mettons qu'on veuille savoir comment s'écrit 54 en base deux. On commence par se demander entre quelles puissances de deux il est : ici 32 et 64. On sait alors quel est le premier chiffre (en partant de la gauche, celui qu'on appelle le chiffre de poids fort) de son écriture en base deux : comme $32 = 2^5$, $54 = 2^5 + (54 - 32) = 2^5 + 22$. On fait le même travail pour 22, etc.

(M2) Des poids faibles vers les poids forts : L'idée de départ est que si $n = b_r 2^r + \cdots + b_1 2 + b_0$ alors en particulier $n = 2B + b_0$ où B est un entier et $b_0 \in \{0, 1\}$. Donc b_0 est le *reste* de la division euclidienne de n par 2. Donc on sait calculer b_0, le dernier chiffre de l'écriture de n en base 2 (le chiffre de poids le plus faible). On a bien sûr $b_0 =$ si n est pair et $b_0 =$ si n est impair. Pour 54 ce chiffre fait donc .

On peut ensuite appliquer la même méthode au quotient B de cette division euclidienne. A partir de l'écriture de départ, $B = b_r 2^{r-1} + \cdots b_2.2 + b_1$ et ceci permet d'obtenir b_1 etc...

Illustration sur un exemple :

$$\begin{aligned} 24 &= 2 \times 12 + 0 \to b_0 = 0, \\ 12 &= 2 \times 6 + 0 \to b_1 = 0, \\ 6 &= 2 \times 3 + 0 \to b_2 = 0, \\ 3 &= 2 \times 1 + 1 \to b_3 = 1, \\ 1 &= 2 \times 0 + 1 \to b_4 = 1. \end{aligned}$$

Question 1 : Implémenter un algorithme suivant la méthode (M2).

On notera `base2PoidsFaibles` la fonction correspondante, prenant un entier naturel `n` en argument, et renvoyant la liste $[b_r, b_{r-1}, \ldots, b_0]$ de ses chiffres en base deux (dans cet ordre !).

Remarque : penser aux cas particuliers.

Question 2 : Ecrire une fonction `puissance2inf` qui pour chaque entier $n \geq 1$ renvoie le plus grand entier r tel que $2^r \leq n$ et renvoie 0 si $n = 0$.

Question 3 : En déduire l'écriture d'un algorithme suivant la méthode (M1) (plus compliquée).

2 L'algorithme d'exponentiation rapide

Le but de cet algorithme est de calculer la puissance N-ième x^N d'un nombre x qui peut être un entier ou un flottant, avec un minimum d'opérations.

L'intérêt de cet algo. ne se limite pas aux entiers. Si on l'introduit ici, c'est qu'ensuite on l'appliquera aussi dans les anneaux de congruences, où il est encore plus efficace, et qu'on s'en servira pour des problèmes d'arithmétiques...

a) L'idée essentielle : Si $N = a_0 + a_1 2 \cdots + a_n 2^n$, écriture en base deux, avec $a_i \in \{0, 1\}$ alors :

$$x^N = x^{a_0} (x^2)^{a_1} \ldots (x^{2^n})^{a_n}.$$

Ensuite deux points de vue possibles, cela dépend si on connait déjà l'écriture en base 2 de N ou pas.

b) **1ère méthode (des poids faibles vers les poids forts)**

En lisant le développement en base deux de droite à gauche si $N = (a_n \dots a_0)_{[2]}$ *: c'est le plus souvent comme cela que l'algorithme est présenté, car dans ce cas, on peut aussi obtenir les chiffres successifs du développement en base* 2 *à chaque étape par l'algorithme des poids faibles déjà vu plus haut : pas besoin d'avoir fait le calcul du dév. en base 2 avant !*

Pour mettre en oeuvre l'algorithme on utilise trois variables qu'on va appeler `res` et `aux` comme résultat et auxiliaire et la variable `N` qui au départ contient comme valeur l'exposant N et qui va permettre à chaque étape de calculer le chiffre du développement en base 2 de N.

De manière pas complètement formelle, à l'étape i :

- On stocke dans `aux` la valeur de x^{2^i}, obtenue comme carré de $x^{2^{i-1}}$ obtenu à l'étape précédente.
- En même temps, on divise à chaque étape N par 2 pour connaître le i-ème chiffre a_i de son écriture en base 2. i.e. on itère `N =N//2` . Le a_i est le reste de la division euclidienne de `N` par 2.
- Enfin toujours à l'étape i , *si* $a_i = 1$, on multiplie `res` par x^{2^i} c'est-à-dire la valeur de `aux` à l'étape i, sinon on ne change pas `res`.

Exercice à faire : implémenter cette idée d'algorithme en PYTHON.

Rappel l'algo. de dév. en base deux des poids faibles vers les poids forts illustré sur un exple :

$$\begin{aligned} 24 &= 2\times 12+0 \to a_0 = 0,\\ 12 &= 2\times 6+0 \to a_1 = 0,\\ 6 &= 2\times 3+0 \to a_2 = 0,\\ 3 &= 2\times 1+1 \to a_3 = 1,\\ 1 &= 2\times 0+1 \to a_4 = 1. \end{aligned}$$

c) **2ème méthode (des poids forts vers les poids faibles)**

En lisant l'écriture en base 2 *de gauche à droite :* cette méthode évite l'utilisation d'une variable auxiliaire, mais nécessite d'avoir calculé le développement en base 2 de N : ce qui n'est pas un gros problème pratique en PYTHON, mais jouerait sur les questions de *coût*.

L'idée astucieuse, qui est utile dans d'autres contextes (algorithme de Hörner), est d'écrire :

$$x^N = ((\dots(x^{a_n})^2 x^{a_{n-1}})^2 \dots)x^{a_1})^2 x^{a_0}$$

Par exemple avec 24 qui s'écrit 11000 en base 2 : $x^{24} = (((x)^2.x)^2)^2)^2)$.

On lit donc l'écriture en base 2 *de* N *de gauche à droite, on part de* `res:= 1` *et à chaque étape s'il y a un* 1 *dans le développement de* N *en base* 2 *on remplace* `res` *par* `(res*res)*x`*, sinon on prend juste le carré i.e. on remplace* `res` *par* `(res*res)`*.*

Exercice à faire : implémenter cette idée d'algorithme en PYTHON

d) Montrer que si N s'écrit avec $n+1$ chiffres en base 2, alors cet algorithme permet de calculer x^N avec au plus $2n$ multiplications.

e) Comparer la vitesse de vos fonctions avec celle de la fonction `pow(x,N)` de PYTHON qui utilise ce même algorithme d'exponentiation rapide.

Suite du T.P. 6

3 Une version du jeu de Nim

Une règle du jeu de Nim est la suivante. Des allumettes sont rangées en tas. Le nombre de tas et le nombre d'allumettes dans chaque tas est arbitraire. Il y a deux joueurs, A et B. Le premier joueur A prend un nombre quelconque d'allumettes (mais au moins une) dans UN tas ; il peut ne prendre qu'une seule allumette, ou autant qu'il souhaite, mais il ne doit toucher qu'un seul tas. Le joueur B joue ensuite selon les mêmes règles et les joueurs jouent à tour de rôle. Le joueur qui prend la dernière allumette gagne la partie.

Définition : Nous appellerons *configuration gagnante* une configuration telle que si un joueur J (A ou B) peut y parvenir par son coup, alors quelle que soit la façon dont l'autre joueur K (resp. B ou A) joue après, ce second joueur perdra.

Représentation du jeu : On choisit de se donner l'état du jeu comme une liste où chaque entrée représente l'effectif d'un tas. Par exemple `[4,5,2]` signifie qu'il y a trois tas, un avec `4` allumettes, un autre avec `5` et un autre avec `2` (bien sûr, l'ordre des tas ne compte pas). Un telle liste sera appelée une *configuration du jeu*.

Exemple de configuration gagnante : Avec cette convention, on comprend que la configuration `[2,2]` est une configuration gagnante. En effet, si A laisse cette configuration à B, alors B n'a que deux possibilités. Si B prend une allumette d'un tas, alors A en prend une de l'autre tas et on arrive à `[1,1]` pour B et clairement A va gagner. Si B prend deux allumettes, alors A prend les deux du tas restant. Dans tous les cas A gagne.

A titre d'exercice vérifiez que `[1,2,3]` est une configuration gagnante.

Codage en binaire et configurations « à somme nulle » :

A chaque configuration de jeu, on va associer sa représentation binaire obtenue comme suit. On écrit l'effectif de chaque tas en binaire en écrivant chaque effectif avec le nombre de chiffres en binaire nécessaire pour représenter le plus grand effectif. Par exemple la configuration `[1,2]` a pour représentation binaire `[01,10]` et `[2,3,6,7]` a pour représentation binaire `[010,011,110,111]`.

Pour chaque configuration c écrite en binaire on écrit les nombres les uns en dessous des autres puis on calcule la somme (modulo 2) des chiffres colonne par colonne par exemple pour `[010,011,110,110]`, on écrit en tableau :

$$\begin{array}{r} 010 \\ 011 \\ 110 \\ 110 \\ \hline 001 \end{array}$$

On note cette somme $S(c)$ et la représentation en tableau ci-dessus sera appelée *tableau de la configuration* où chaque ligne représente un tas.

Pour `c1=[11,10]`, on aura `S(c1)=01`. Pour `c2=[10,10]` on aura `S(c2)=00`.

Pour `c3=[010,011,110,111]`, on aura `S(c3)=000`.

On a alors l'étonnant :

Théorème : une configuration `c` est gagnante si, et seulement si, `S(c)` a *tous ses chiffres égaux à* 0.

Démonstration du théorème : elle est en deux parties.

• Si on modifie un tas d'une configuration à somme nulle, alors la configuration n'est plus à somme nulle. En effet modifier un tas c'est modifier une ligne L_i du tableau ci-dessus.

Dire la configuration est à somme nulle revient à dire que la somme de toutes les autres lignes que la i-ième est égale à la i-ième donc changer la i-ième supprime forcément cette propriété.

• Il est toujours possible, à partir d'une configuration à somme non nulle, de la transformer, en modifiant seulement une ligne, en une configuration à somme nulle.

Question 1 : comment faire cela ?

• **Question 2 :** pourquoi les deux points précédents permettent de conclure pour la démonstration du théorème ?

Question 3 : Travail informatique à faire : Ecrire une fonction `Nim` qui reçoit une liste `L` dont les entrées sont les effectifs initiaux des tas, en nombre quelconque, choisis par l'utilisateur.

La fonction va permettre à l'ordinateur et au joueur de jouer à tour de rôle.

L'ordinateur commence.

- Si la configuration n'est pas à somme nulle, il la transforme par la méthode que vous avez trouvée à la question 2, en une configuration à somme nulle. Puis il demande à l'humain de jouer et à chaque étape l'ordi refabrique une configuration à somme nulle et l'ordi est sûr de gagner.

- Si la configuration est à somme nulle, l'ordinateur enlève juste une allumette et fait ainsi tant qu'il n'hérite pas d'une situation à somme non nulle. Bien sûr si le joueur humain sait calculer tout cela il gagnera mais dès qu'il fait une erreur l'ordi gagnera.

Solution de la Question 1 expliquée sur l'exemple :

11010
10010
00110
00111

01001

On regarde le premier bit de `S(c)` où il y a un `1` : ici c'est le second.

On regarde la première ligne du tableau où ce bit vaut aussi `1`. : ici c'est dans la première.

Dans cette ligne, on va modifier tous les bits qui sont à la même place que les `1` dans `S(c)`, je les écris avec un étoile dessus $L_1 = 1\overset{*}{1}010\overset{*}{0}$.

Et donc on remplace L_1 par $L_1' = 10011$.

L'important, c'est que comme on modifie le premier bit qui vaut `1` le nombre obtenu est inférieur, donc cela correspond bien à *enlever* des allumettes.

Solution de la question 2 : le fait d'être à somme nulle étant un *invariant* de la configuration de jeu laissée à l'humain et le nombre d'allumettes étant strictement décroissant (c'est le variant), va arriver le moment où la configuration laissé à l'humain sera nulle : l'ordi gagne.