Contrôle d'informatique

Durée : 2 heures

Ce contrôle est constitué de quatre exercices indépendants.

Exercice 1 Représentation machine des entiers relatifs

Dans cet exercice, on considère des entiers relatifs stockés sur des mots de 1 octet, c'est-à-dire sur 8 bits.

a) Quels sont les entiers relatifs que l'on peut représenter par le codage en complément à deux ?

b) Donner la représentation décimale des entiers signés suivants, codés en complément à deux : `00110101` et `10110101`.

c) Donner le codage en complément à deux des entiers signés suivants : 97 et -34.

d) Rappeler l'algorithme utilisé pour calculer la représentation de l'opposé d'un entier codé en complément à deux. Quel entier obtient-on si on applique cet algorithme à -128 ?

e) Soit x le nombre représenté par `10000010` et y celui représenté par `10101011`. Soit z le nombre obtenu en additionnant (sur 8 bits) ces deux représentations. Que valent x, y et z ? A-t-on $z = x + y$? Si non, précisez la relation liant $x + y$ et z.

Exercice 2 Exponentiation binaire

Dans cet exercice on s'intéresse au calcul de x^n lorsque x et n sont des entiers non nuls en cherchant à minimiser le nombre de multiplications utilisées (et en s'interdisant bien entendu l'usage de l'opérateur `**`).

a) Rédiger en Python une première solution utilisant $n-1$ multiplications. On définira une fonction `power1(x, n)` prenant deux arguments x et n et renvoyant la valeur de x^n.

On considère maintenant l'algorithme suivant, rédigé en Python :

```python
def power2(x, n):
    u = n
    y = x
    z = 1
    while u > 0:
        if u % 2 == 1:
            z = z * y
        y = y * y
        u = u // 2
    return z
```

b) Cet algorithme réalise l'itération de trois suites (u_k), (y_k) et (z_k). Précisez les valeurs initiales et les relations de récurrence qui régissent chacune de ces trois suites.

c) Exprimer y_k en fonction de x et de k.

d) On considère la décomposition en base 2 de l'entier n, que l'on écrit : $n = (b_p b_{p-1} \cdots b_1 b_0)_2$ avec $b_i \in \{0, 1\}$ et $b_p = 1$. Comment s'écrit la décomposition en base 2 de $\lfloor n/2 \rfloor$? En déduire l'expression de u_k en fonction de la décomposition binaire de n.

e) Justifier maintenant la terminaison de cet algorithme, puis prouver que cette fonction renvoie la valeur de x^n.

f) Donner un encadrement du nombre de multiplications utilisées par cet algorithme pour calculer x^n. On exprimera cet encadrement à l'aide de l'entier p.
Précisez enfin pour quels entiers n la borne inférieure (respectivement supérieure) de cet encadrement est atteinte.

Exercice 3 Codage de Fibonacci

On rappelle que la suite de Fibonacci est définie par les conditions initiales $f_0 = 0$, $f_1 = 1$ et la relation de récurrence $f_{n+2} = f_{n+1} + f_n$.

a) Rédiger en Python une fonction nommée `pgf` prenant en argument un entier $n \geqslant 1$ et renvoyant le plus grand terme f_k de la suite de Fibonacci vérifiant $f_k \leqslant n$.

Tout entier $n \geqslant 1$ peut être décomposé en somme de termes distincts de la suite de Fibonacci. Par exemple, $50 = 34 + 8 + 5 + 3 = f_9 + f_6 + f_5 + f_4$. Cependant, pour que cette décomposition soit unique on doit ajouter la contrainte suivante : on n'utilise pas f_0 et f_1 et on s'interdit d'avoir dans la décomposition de n deux termes *consécutifs* de la suite de Fibonacci. Par exemple, la décomposition précédente de 50 ne convient pas, mais $50 = 43 + 13 + 3 = f_9 + f_7 + f_4$ convient. Ce résultat constitue le théorème de Zeckendorf.

b) Montrer l'existence d'une telle décomposition pour tout entier $n \geqslant 1$. *Nous admettrons son unicité.*

Le *codage* de Fibonacci est une représentation des entiers naturels non nuls fondée sur cette décomposition.

Si $n = \sum_{i=0}^{k-1} d_i f_{i+2}$ vérifie les conditions :

$$\forall i \in [\![0, k-1]\!],\ d_i \in \{0,1\}, \qquad d_{k-1} = 1, \qquad \forall i \in [\![0, k-1]\!],\ d_i d_{i+1} = 0$$

alors l'entier n sera représenté par la chaîne de caractères $d_0 d_1 d_2 \cdots d_{k-1}$.
Par exemple, l'entier 50 est représenté par la chaîne de caractères `'00100101'` puisque

$$50 = 0.f_2 + 0.f_3 + 1.f_4 + 0.f_5 + 0.f_6 + 1.f_7 + 0.f_8 + 1.f_9.$$

c) Rédiger en Python une fonction nommée `decode` qui prend en paramètre un code de Fibonacci et qui renvoie l'entier n représenté par ce code. Par exemple, `decode('00100101')` devra retourner `50`.

d) Rédiger en Python la fonction inverse nommée `code` : celle-ci prend en paramètre un entier strictement positif et retourne le code de Fibonacci de ce nombre. Par exemple, `code(50)` retournera la chaîne de caractères `'00100101'`.

On rappelle qu'en Python l'opérateur `+` agit sur les chaînes de caractères par concaténation : `'000' + '111'` renvoie la chaîne `'000111'`.

e) **Application à l'exponentiation rapide.**

Si $k \geqslant 2$, montrer que le calcul de x^{f_k} peut être réalisé à l'aide de $k-2$ multiplications.

En déduire une fonction `power` qui calcule x^n lorsque n est représenté par son code de Fibonacci. Par exemple, `power(2, '00100101')` devra retourner `1125899906842624`, c'est-à-dire la valeur de 2^{50}.

Exercice 4 — Décomposition sur la base factorielle

Un entier $n > 0$ étant fixé, on admet que tout entier k pris dans $[\![0, n![\![$ peut s'écrire de manière unique sous la forme :

$$k = a_{n-1}(n-1)! + a_{n-2}(n-2)! + \cdots + a_2 2! + a_1 1! + a_0 \qquad \text{avec pour tout } i,\ a_i \in [\![0, i+1[\![.$$

L'écriture ci-dessus est appelée décomposition sur la base factorielle de l'entier k.
Par exemple, pour $n = 5$ et $k = 85$, on a $k = 3 \times 4! + 2 \times 3! + 0 \times 2! + 1 \times 1! + 0 \times 0!$.

a) rédiger une fonction `decode` qui prend en argument une liste $[a_0, a_1, \ldots, a_{n-1}]$ supposée vérifier $a_i \in [\![0, i+1[\![$ et qui renvoie la valeur de l'entier k représenté par cette décomposition sur la base factorielle. On rappelle qu'il n'existe pas de fonction factorielle prédéfinie en Python.

Par exemple, `decode([0, 1, 0, 2, 3])` devra renvoyer la valeur 85.

b) À quoi est égal a_0 ?

Montrer que $k - a_1$ est pair, et en déduire une expression simple de a_1 en fonction de k (on pourra noter $x \bmod y$ le reste de la division euclidienne de x par y). De même, exprimer a_2 en fonction de k et a_1 puis plus généralement a_i en fonction de k et de l'entier $a_{i-1}(i-1)! + a_{i-2}(i-2)! + \cdots + a_2 2! + a_1 1! + a_0$.

c) En déduire une fonction Python `code(n, k)` qui prend en arguments la taille n et un entier $k \in [\![0, n![\![$ et qui renvoie la décomposition de k sur la base factorielle sous la forme d'un tableau $[a_0, a_1, \ldots, a_{n-1}]$.

d) **Application à la génération de permutations**

Nous savons que les permutations des éléments de $[\![0, n[\![$ sont au nombre de $n!$. Il est donc possible d'établir une bijection entre $[\![0, n![\![$ et l'ensemble de ces permutations. Nous allons pour cela utiliser la décomposition sur la base factorielle.

Une fois k décomposé sur la base factorielle, la permutation σ_k de $[\![0, n]\!]$ représentée par k se calcule comme suit. En premier lieu, on considère la séquence $\mathcal{L} = (0, 1, \ldots, n-1)$ à n éléments. Cette séquence est modifiée au fur et à mesure que les valeurs prises par la permutation σ_k sont calculées.

La première valeur calculée est $\sigma_k(0)$, égale au $(1 + a_{n-1})$-ième élément de $\mathcal{L}$ (c'est-à-dire à a_{n-1}). Une fois $\sigma_k(0)$ calculé, cet entier est retiré de $\mathcal{L}$, qui ne contient plus que $n-1$ entiers.

La seconde valeur calculée est $\sigma_k(1)$, égal au $(1 + a_{n-2})$-ième élément de $\mathcal{L}$. Une fois $\sigma_k(1)$ calculé, cet entier est retiré de $\mathcal{L}$. Le procédé est répété jusqu'au calcul de $\sigma_k(n-1)$ égal à l'unique élément de $\mathcal{L}$ restant.

Par exemple, dans le cas $n = 5$, $k = 85$ on a : $\sigma_{85}(0) = 3$ (car $a_4 = 3$), et $\mathcal{L}$ devient $(0, 1, 2, 4)$. Ensuite $\sigma_{85}(1) = 2$ (car $a_3 = 2$) et $\mathcal{L}$ devient $(0, 1, 4)$. Ensuite $\sigma_{85}(2) = 0$ (car $a_2 = 0$) et $\mathcal{L}$ devient $(1, 4)$. Ensuite $\sigma_{85}(3) = 4$ (car $a_1 = 1$) et pour finir $\sigma_{85}(4) = 1$.

Ecrire la fonction `Permutation(n, k)` qui prend en arguments la taille n et l'entier k de $[\![0, n![\![$ et qui renvoie la permutation σ_k représentée par le tableau des $\sigma_k(i)$ dans l'ordre de i croissants. On rappelle à toutes fins utiles que l'instruction `del t[i]` supprime l'élément de rang `i` de la liste `t`.

Ainsi, `Permutation(5, 85)` devra renvoyer la liste `[3, 2, 0, 4, 1]`.