Contrôle d'informatique

Durée : 2 heures

Exercice 1 Représentation machine des entiers relatifs

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

1. Quels sont les entiers relatifs que l'on peut représenter par le codage en complément à deux ?
2. Donner la représentation décimale des entiers signés suivants, codés en complément à deux : 00001101 et 11001101.
3. Donner le codage en complément à deux des entiers signés suivants : 60 et -127.
4. On suppose les entiers représentés en complément à deux par une chaîne de caractères de longueur 8 constituée des caractères `'0'` et `'1'`. Par exemple, les entiers évoqués à la question 2 sont représentés par les chaînes de caractères `'00001101'` et `'11001101'`.

   Rédiger une fonction qui prend en paramètre la représentation d'un entier n et retourne celle de l'entier $-n$.

   Indication. Si s est une chaîne de caractères, l'énumération `for c in s:` énumère les caractères de s de la gauche vers la droite tandis que `for c in reversed(s):` énumère ces mêmes caractères de la droite vers la gauche.

Exercice 2 pgcd binaire

On considère deux entiers p et q strictement positifs et la fonction :

```python
def pgcd(p, q):
    u, v = p, q
    while u != v:
        if u % 2 == 0:
            u = u // 2
        elif u > v:
            u = (u - v) // 2
        else:
            u, v = (v - u) // 2, u
    return u
```

1. Appliquer cette fonction aux entiers $p = 60$ et $q = 35$ en détaillant les calculs, c'est à dire en précisant les différentes valeurs prises par les variables u et v.
2. Prouver que lorsque q est un entier *impair* cette fonction calcule le pgcd de p et de q.
3. Compléter cette fonction pour qu'elle puisse s'appliquer à deux entiers naturels non nuls *quelconques* (vous réécrirez complètement le code de la fonction modifiée sur votre copie).

Exercice 3 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 `**`).

1. Rédiger en Python une première solution utilisant $n-1$ multiplications (on définira une fonction prenant deux arguments x et n et retournant la valeur de x^n).

Le calcul peut être accéléré si on procède de la façon suivante :

– on décompose l'entier n en base 2 : $n = (a_p a_{p-1} \cdots a_1 a_0)_2$ avec $a_i \in \{0, 1\}$ et $a_p = 1$;

– on effectue le produit des x^{2^i} correspondant aux valeurs de a_i qui valent 1.

2. Rédiger une deuxième fonction fondée sur le principe ci-dessus pour calculer x^n.
3. 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 pour quels entiers la borne inférieure (respectivement supérieure) de cet encadrement est atteinte.

Exercice 4 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 $f_{n+2} = f_{n+1} + f_n$.

1. 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.

2. 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

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

3. 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`.
4. 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'`.
5. **Application à l'exponentiation rapide.**
   (a) Si $k \geqslant 2$, montrer que le calcul de x^{f_k} peut être réalisé à l'aide de $k-2$ multiplications.
   (b) En déduire une fonction `puissance` qui calcule x^n lorsque n est représenté par son code de Fibonacci. Par exemple, `puissance(2, '00100101')` devra retourner `1125899906842624` (c'est la valeur de 2^{50}).