Corrigé du contrôle d'informatique

Exercice 1 Représentation machine des entiers relatifs

a) Le codage en complément à deux sur 8 bits permet de représenter les entiers compris entre $-2^7 = -128$ et $2^7 - 1 = 127$.

b) Le premier bit indique le signe donc `00110101` est la représentation de $(110101)_2 = 53$ et `10110101` la représentation de $(10110101)_2 - 2^8 = 181 - 256 = -75$.

c) 97 est positif donc il est représenté par sa décomposition binaire c'est à dire `01100001`.
-34 est négatif donc est représenté par la décomposition binaire de $2^8 - 34 = 222$ c'est à dire `11011110`.

d) On obtient la représentation de l'opposé de x en procédant ainsi :

(i) remplacer dans la représentation de x tous les bits égaux à 0 par des 1 et réciproquement ;
(ii) additionner 1 au résultat obtenu ;
(iii) ne garder que les 8 derniers bits.

L'entier -128 est représenté par la décomposition binaire de $2^8 - 128 = 128$, soit `10000000`. Appliqué à cette représentation l'algorithme ci-dessus renvoie `01111111 + 1 = 10000000`, autrement dit la représentation de -128, qui se trouve être son propre opposé (*ce qui est vrai modulo 256*).

e) D'après le bit de signe x et y sont des nombres négatifs ; plus exactement $x = -126$ et $y = -85$.
L'addition de leurs représentations occupe 9 bits : `100101101` ; le neuvième bit est tronqué donc z est représenté par `00101101` et $z = 45$.
On a bien entendu $x + y \neq z$ puisque $x + y \notin [\![-128, 127]\!]$, mais $z = x + y + 256$.

Exercice 2 Exponentiation binaire

a) On définit la fonction :

```python
def power1(x, n):
    y = x
    for k in range(n-1):
        y = y * x
    return y
```

b) Cette fonction réalise l'itération des trois suites définies par les valeurs initiales $u_0 = n$, $y_0 = x$, $z_0 = 1$ et les relations de récurrence :

$$z_{k+1} = \begin{cases} z_k & \text{si } u_k \text{ est pair} \\ y_k z_k & \text{si } u_k \text{ est impair} \end{cases} \qquad y_{k+1} = y_k^2 \qquad u_{k+1} = \lfloor u_k/2 \rfloor.$$

c) On en déduit bien évidemment que $y_k = x^{2^k}$.

d) Lorsque $n = (b_p b_{p-1} \cdots b_1 b_0)_2$ on a $\lfloor n/2 \rfloor = (b_p b_{p-1} \cdots b_1)_2$. De ceci il résulte que $u_k = (b_p b_{p-1} \cdots b_k)_2$.

e) Ainsi, $u_p = (b_p)_2 = 1 \neq 0$ et $u_{p+1} = 0$ donc l'algorithme se termine (la condition de la boucle conditionnelle cesse d'être vérifiée) et renvoie la valeur de z_{p+1}.
Sachant que $u_k \bmod 2 = b_k$, la relation de récurrence qui régit l'évolution de la suite (z_k) peut aussi s'écrire : $z_{k+1} = z_k \times y_k^{b_k} = z_k \times x^{b_k 2^k}$. Ainsi,

$$z_{p+1} = z_0 \prod_{k=0}^{p} x^{b_k 2^k} = x^{\sum_{k=0}^{p} b_k 2^k} = x^n$$

ce qui justifie la validité de cet algorithme.

f) Notons $c(n)$ le nombre de multiplications réalisées par cet algorithme. La boucle conditionnelle est réalisée $p+1$ fois. Durant cette exécution, l'opération `y = y * y` est réalisée à chaque fois, et l'opération `z = z * y` à chaque fois que $b_k = 1$.
Le nombre de valeurs de b_k égales à 1 est compris entre 1 et $p+1$ donc $p + 2 \leqslant c(n) \leqslant 2(p+1)$.
On a $c(n) = p + 2$ lorsque seul b_p est égal à 1. On a alors $n = (100 \cdots 000)_2 = 2^p$.
On a $c(n) = 2(p+1)$ lorsque tous les b_k sont égaux à 1. On a alors $n = (111 \cdots 111)_2 = 2^{p+1} - 1$.

Exercice 3 Codage de Fibonacci

a) Il suffit d'itérer deux suites $u_i = f_i$ et $v_i = f_{i+1}$ jusqu'à obtenir la condition d'arrêt $u_i \leqslant n < v_i$.

```
def pgf(n):
    u, v = 0, 1
    while v <= n:
        u, v = v, u + v
    return u
```

b) Montrons par récurrence sur $n \geqslant 1$ que n peut être décomposé en sommes de termes distincts et non consécutifs de la suite de Fibonacci.

- C'est clair pour $n = 1$ puisque $n = f_2$.
- Si $n \geqslant 2$, supposons le résultat acquis jusqu'au rang $n-1$ et considérons le plus grand terme f_k de la suite de Fibonacci vérifiant la condition $f_k \leqslant n$.
  On a $f_k \leqslant n < f_{k+1}$ donc $0 \leqslant n - f_k < f_{k-1}$. Si $n = f_k$ le résultat est acquis au rang n ; sinon on a $1 \leqslant n - f_k < f_{k-1}$ et par hypothèse de récurrence $n - f_k$ peut être décomposé en somme de termes distincts et non consécutifs de la suite de Fibonacci. De plus, l'encadrement précédent montre que f_{k-1} ne peut faire partie de cette décomposition donc le résultat est bien acquis pour $n = f_k + (n - f_k)$.

Remarque. La justification de l'unicité de cette décomposition (non demandée) consiste à prouver le lemme suivant :

> *la somme de tout ensemble de termes de la suite de* Fibonacci *distincts et non consécutifs et dont le plus grand élément est f_k est strictement inférieure à f_{k+1}.*

Ceci se prouve par récurrence sur k :

- c'est bien le cas pour $k = 0$;
- Si $k > 1$, supposons le résultat acquis jusqu'au rang $k-1$ et considérons un tel ensemble S. Par hypothèse de récurrence la somme des termes de $\mathrm{S} \setminus \{f_k\}$ est strictement inférieure à f_{k-1} donc la somme des termes de S est strictement inférieure à $f_k + f_{k-1} = f_{k+1}$.

c) Le décodage est simple : on parcours la suite de Fibonacci en additionnant les termes de la suite associés aux caractères `'1'` de la représentation :

```
def decode(s):
    u, v = 1, 1
    x = 0
    for c in s:
        if c == '1':
            x += v
        u, v = v, u + v
    return x
```

d) Pour le codage on s'inspire de la démarche établie à la question 2 : on détermine le plus grand terme f_k de la suite de Fibonacci qui soit inférieur ou égal à n puis on décompose $n - f_k$.

```
def code(n):
    u, v = 1, 1
    while v <= n:
        u, v = v, u + v
    s = '1'
    x = n - u
    while u > 1:
        u, v = v - u, u
        if u <= x:
            s = '1' + s
            x = x - u
        else:
            s = '0' + s
    return s
```

e) la relation $x^{f_k} = x^{f_{k-1}} \cdot x^{f_{k-2}}$ montre que le couple $(x^{f_k}, x^{f_{k-1}})$ peut être calculé à partir du couple $(x^{f_{k-1}}, x^{f_{k-2}})$ à l'aide d'une seule multiplication. Sachant que le calcul de $(x^{f_2}, x^{f_1}) = (x, x)$ ne nécessite aucune multiplication, on prouve alors par récurrence que le calcul de $(x^{f_k}, x^{f_{k-1}})$ ne nécessite que $k - 2$ multiplications.

Si n est représenté par la chaîne de caractères $d_0 d_1 d_2 \cdots d_{k-1}$ le calcul de x^n consiste à effectuer le produit des $x^{f_{i+2}}$ correspondant aux valeurs de d_i qui valent 1 :

```
def puissance(x, s):
    u, v = x, x
    y = 1
    for c in s:
        if c == '1':
            y *= v
        u, v = v, u * v
    return y
```

Exercice 4 Décomposition sur la base factorielle

a) Pour décoder l'expression d'un nombre décomposé sur la base factorielle, il suffit de maintenir les invariants : $u_k = k!$ et $v_k = a_k k! + \cdots + a_1 1! + a_0$. Ceux-ci se définissent par les relations :

$$u_0 = 1, \quad v_0 = a_0 \qquad \text{et} \qquad u_{k+1} = (k+1)u_k, \quad v_{k+1} = v_k + a_{k+1} u_{k+1}.$$

```
def decode(s):
    u, v = 1, s[0]
    for k in range(1, len(s)):
        u = u * k
        v = v + u * s[k]
    return v
```

b) Puisque $a_0 \in [\![0, 1[\![$ on a nécessairement $a_0 = 0$.
Pour tout $i \geqslant 2$, $i!$ est divisible par 2 donc $k - a_1! - a_0 = k - a_1$ est pair. On a donc $a_1 = k \bmod 2$.
Pour tout $i \geqslant 3$, $i!$ est divisible par 6 donc $\frac{1}{2!}(k - a_2 2! - a_1 1! - a_0)$ est divisible par 3. On a donc $a_2 = \frac{1}{2!}(k - a_0 - a_1 1!) \bmod 3$.
Plus généralement, $\frac{1}{i!}\Big(k - \sum_{j=0}^{i-1} a_j j!\Big) \equiv a_i \bmod (i+1)$ donc $a_i = \frac{1}{i!}\Big(k - \sum_{j=0}^{i-1} a_j j!\Big) \bmod (i+1)$.

c) Ceci conduit à la fonction :

```
def code(n, k):
    a = [0]
    for i in range(1, n):
        a.append(k % (i+1))
        k = k // (i+1)
    return a
```

d) Il suffit de suivre la description de l'énoncé :

```
def permutation(n, k):
    a = code(n, k)
    ell = [i for i in range(n)]
    sigma = []
    for i in range(n):
        sigma.append(ell[a[n-i-1]])
        del ell[a[n-i-1]]
    return sigma
```