Corrigé du contrôle

Exercice 1 Représentation machine des entiers relatifs

1. Le codage en complément à deux sur 8 bits permet de représenter les entiers compris entre $-2^7 = -128$ et $2^7 - 1 = 127$.
2. Le premier bit indique le signe donc 00001101 est la représentation de $(1101)_2 = 13$ et 11001101 la représentation de $(11001101)_2 - 2^8 = 205 - 256 = -51$.
3. 60 est positif donc il est représenté par sa décomposition binaire c'est à dire 00111100.
   -127 est négatif donc est représenté par la décomposition binaire de $2^8 - 127 = 129$ c'est à dire 10000001.
4. Rappelons que d'après le cours 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.

   Ceci conduit à la fonction suivante :

```python
def oppose(n):
    x = ''
    for c in n:
        if c == '0':
            x = x + '1'
        else:
            x = x + '0'
    y, r = '', 1
    for c in reversed(x):
        if int(c) + r == 2:
            y, r = '0' + y, 1
        elif int(c) + r == 1:
            y, r = '1' + y, 0
        else:
            y, r = '0' + y, 0
    return y[-8:]
```

   la variable r représente la retenue qui se propage lors de l'addition.

Exercice 2 Analyse d'un algorithme

1. Les deux variables u et v prennent successivement les valeurs suivantes :

u	v
60	35
30	35
15	35
10	15
5	15
5	5

   puis la fonction retourne l'entier 5.

2. Notons u_n et v_n les valeurs prises par les variables u et v après n passages par la boucle conditionnelle.
   Commençons par prouver les invariants suivants :

   tant que u_n et v_n sont définis on a : $u_n > 0$, $v_n > 0$, u_n est impair et $\operatorname{pgcd}(u_n, v_n) = \operatorname{pgcd}(p, q)$.

   – C'est vrai pour $n = 0$ puisque $u_0 = p$ et $v_0 = q$.
   – Si $n \geqslant 0$, supposons le résultat acquis au rang n et montrons-le au rang $n+1$.
     Notons déjà que si u_{n+1} et v_{n+1} sont définis, alors $u_n \neq v_n$.
     Trois cas sont possibles :
     – si u_n est pair alors $u_n = 2u_{n+1}$ et $v_{n+1} = v_n > 0$. Par hypothèse de récurrence, $u_n \geqslant 2$ et $v_n \geqslant 1$ donc $u_{n+1} > 0$ et $v_{n+1} > 0$. De plus v_n est impair donc v_{n+1} aussi et enfin,

$$\operatorname{pgcd}(u_{n+1}, v_{n+1}) = \operatorname{pgcd}(2u_{n+1}, v_{n+1}) \text{ (car } v_{n+1} \text{ est impair) } = \operatorname{pgcd}(u_n, v_n)\,;$$

- si u_n est impair et $u_n > v_n$ alors $u_n - v_n = 2u_{n+1}$ et $v_{n+1} = v_n$. Puisque u_n et v_n sont impairs, $u_n - v_n \geqslant 2$ donc $u_{n+1} > 0$, $v_{n+1} > 0$ et v_{n+1} est impair. Enfin,

$$\text{pgcd}(u_{n+1}, v_{n+1}) = \text{pgcd}(2u_{n+1}, v_{n+1}) = \text{pgcd}(u_n - v_n, v_n) = \text{pgcd}(u_n, v_n)\,;$$

- si u_n est impair et $u_n < v_n$ alors $v_n - u_n = 2u_{n+1}$ et $u_n = v_{n+1}$. Comme au cas précédent, $v_n - u_n \geqslant 2$ donc $u_{n+1} > 0$, $v_{n+1} > 0$, v_{n+1} est impair et :

$$\text{pgcd}(u_{n+1}, v_{n+1}) = \text{pgcd}(2u_{n+1}, v_{n+1}) = \text{pgcd}(v_n - u_n, u_n) = \text{pgcd}(u_n, v_n).$$

Montrons maintenant que tant que u_{n+1} et v_{n+1} sont définis on a $u_{n+1} + v_{n+1} < u_n + v_n$.

- Si u_n est pair, $u_{n+1} + v_{n+1} = \dfrac{u_n}{2} + v_n$ et puisque $u_n > 0$, $u_{n+1} + v_{n+1} < u_n + v_n$.
- Si u_n est impair, $u_{n+1} + v_{n+1} = \dfrac{u_n + v_n}{2}$ et puisque $u_n + v_n > 0$, $u_{n+1} + v_{n+1} < u_n + v_n$.

Si l'algorithme ne se terminait pas, u_n et v_n seraient définies pour tout $n \in \mathbb{N}$ et la suite $(u_n + v_n)_{n\in\mathbb{N}}$ serait une suite d'entiers strictement décroissante d'entiers positifs, ce qui ne se peut. Ceci justifie la terminaison de la fonction, qui retourne le pgcd de p et q d'après l'invariant prouvé ci-dessus.

3. La fonction pgcd s'appuie sur les relations suivantes :

$$pgcd(2a, 2b+1) = \text{pgcd}(a, 2b+1), \qquad \text{pgcd}(2a+1, 2b+1) = \begin{cases} \text{pgcd}(a-b, 2b+1) & \text{si } a > b \\ \text{pgcd}(b-a, 2a+1) & \text{si } a < b \end{cases}$$

Pour qu'elle puisse s'appliquer dans le cas général, on lui adjoint les relations :

$$\text{pgcd}(2a+1, 2b) = \text{pgcd}(2a+1, b) \quad \text{et} \quad \text{pgcd}(2a, 2b) = 2\,\text{pgcd}(a, b)$$

La dernière relation nécessite d'utiliser un accumulateur pour garder en mémoire la puissance de 2 par laquelle multiplier le résultat final.

```python
def pgcd(p, q):
    u, v = p, q
    d = 1
    while u != v:
        if v % 2 == 1 and u % 2 == 0:
            u = u // 2
        elif u % 2 == 1 and v % 2 == 0:
            v = v // 2
        elif u % 2 == 0 and v % 2 == 0:
            d *= 2
            u, v = u // 2, v // 2
        elif u > v:
            u = (u - v) // 2
        else:
            v = (v - u) // 2
    return d * u
```

Exercice 3

1. La version naïve de l'exponentiation prend la forme suivante :

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

2. Nous allons maintenir trois invariants :

$$u_i = x^{2^i}, \quad v_i = x^{a_{i-1}2^{i-1} + \cdots + a_0}, \quad w_i = (a_p \cdots a_i)_2.$$

Les valeurs initiales sont $u_0 = x$, $v_0 = 1$, $w_0 = n$ et on dispose des relations de récurrence :

$$u_{i+1} = u_i^2, \qquad v_{i+1} = \begin{cases} u_i v_i & \text{si } w_i \text{ est impair} \\ v_i & \text{sinon} \end{cases}, \qquad w_{i+1} = \lfloor w_i/2 \rfloor$$

On choisit pour condition d'arrêt $w_i = 0$ ce qui correspond à $i = p+1$ et $v_{p+1} = x^n$.
Ceci conduit à l'algorithme suivant :

```
def puissance(x, n):
    u, v, w = x, 1, n
    while w > 0:
        if w % 2 == 1:
            v *= u
        u = u * u
        w = w // 2
    return v
```

3. Le nombre de multiplications de u avec lui-même est égal à p ; le nombre de produits de u par v est égal au nombre de termes valant 1 parmi $a_0, \dots, a_p$; il y en a entre 1 et $p+1$ donc le nombre total de multiplications effectuées est compris entre $p+1$ et $2p+1$.
   La borne inférieure $p+1$ est atteinte lorsque tous les a_i (à l'exception de a_p) sont nuls, c'est à dire lorsque $n = (1000\cdots00)_2 = 2^p$.
   La borne supérieure $2p+1$ est atteinte lorsque tous les a_i valent 1, c'est à dire lorsque $n = (111\cdots11)_2 = 2^{p+1} - 1$.

Exercice 4

1. 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
```

2. 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 $S \setminus \{f_k\}$ est strictement inférieure à f_{k-1} donc la somme des termes de S est strictement inférieure à $f_k + f_{k-1} = f_{k+1}$.

3. Le codage 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
```

4. 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 $n - f_k$.

```python
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
```

5. 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 :

```python
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
```