# Correction du DS 3

Informatique pour tous, première année

Julien Reichert

## Exercice 1

On travaillera sur $\frac{1}{37}$ en tant que fraction (valeur décimale : $0,027027\ldots$), malheureusement le cycle est de taille 36, soit le pire possible (en tant que diviseur de $37-1$). On sait qu'en binaire notre nombre commence par $0,\ldots$ puisqu'il s'agit de la partie entière en décimal aussi.

Procédons donc aux multiplications par 2 successives.

$$
\begin{array}{llll}
2\times \frac{1}{37} = \mathbf{0} + \frac{2}{37} & \to & 2\times \frac{2}{37} = \mathbf{0} + \frac{4}{37} & \to \\
2\times \frac{4}{37} = \mathbf{0} + \frac{8}{37} & \to & 2\times \frac{8}{37} = \mathbf{0} + \frac{16}{37} & \to \\
2\times \frac{16}{37} = \mathbf{0} + \frac{32}{37} & \to & 2\times \frac{32}{37} = \mathbf{1} + \frac{27}{37} & \to \\
2\times \frac{27}{37} = \mathbf{1} + \frac{17}{37} & \to & 2\times \frac{17}{37} = \mathbf{0} + \frac{34}{37} & \to \\
2\times \frac{34}{37} = \mathbf{1} + \frac{31}{37} & \to & 2\times \frac{31}{37} = \mathbf{1} + \frac{25}{37} & \to \\
2\times \frac{25}{37} = \mathbf{1} + \frac{13}{37} & \to & 2\times \frac{13}{37} = \mathbf{0} + \frac{26}{37} & \to \\
2\times \frac{26}{37} = \mathbf{1} + \frac{15}{37} & \to & 2\times \frac{15}{37} = \mathbf{0} + \frac{30}{37} & \to \\
2\times \frac{30}{37} = \mathbf{1} + \frac{23}{37} & \to & 2\times \frac{23}{37} = \mathbf{1} + \frac{9}{37} & \to \\
2\times \frac{7}{37} = \mathbf{0} + \frac{18}{37} & \to & 2\times \frac{14}{37} = \mathbf{0} + \frac{36}{37} & \to \\
2\times \frac{28}{37} = \mathbf{1} + \frac{35}{37} & \to & 2\times \frac{19}{37} = \mathbf{1} + \frac{33}{37} & \to \\
2\times \frac{33}{37} = \mathbf{1} + \frac{29}{37} & \to & 2\times \frac{29}{37} = \mathbf{1} + \frac{21}{37} & \to \\
2\times \frac{21}{37} = \mathbf{1} + \frac{5}{37} & \to & 2\times \frac{5}{37} = \mathbf{0} + \frac{10}{37} & \to \\
2\times \frac{10}{37} = \mathbf{0} + \frac{20}{37} & \to & 2\times \frac{20}{37} = \mathbf{1} + \frac{3}{37} & \to \\
2\times \frac{3}{37} = \mathbf{0} + \frac{6}{37} & \to & 2\times \frac{6}{37} = \mathbf{0} + \frac{12}{37} & \to \\
2\times \frac{12}{37} = \mathbf{0} + \frac{24}{37} & & &
\end{array}
$$

À ce stade, on a vingt-trois bits depuis le premier 1 et la mantisse est pleine. On note cependant que le bit suivant serait un 1 car $\frac{24}{37} \geq \frac{1}{2}$.

On note qu'à partir du moment où on a trouvé $\frac{36}{37}$, on sait que la moitié du cycle est atteinte et que le reste est symétrique.

L'exposant est $-6$. Il se représente sur 8 bits comme $-6 + 2^{8-1} - 1$, soit $\overline{01111001}^2$. L'écriture de la mantisse en virgule flottante omettant le 1 avant la virgule, on a donc les bits 10111010110011111001001 car, le bit suivant étant un 1 suivi ultérieurement de quelques 1 (seulement une infinité…), l'arrondi se fait par excès.

La représentation finale est alors 0 01111001 10111010110011111001001.

Concernant le nombre $0,42$, on applique la même méthode. On peut aussi agir sur la fraction $\frac{21}{50}$.

$$
\begin{aligned}
2\times 0,42 &= \mathbf{0} + 0,84\\
2\times 0,84 &= \mathbf{1} + 0,68\\
2\times 0,68 &= \mathbf{1} + 0,36\\
2\times 0,36 &= \mathbf{0} + 0,72\\
2\times 0,72 &= \mathbf{1} + 0,44\\
2\times 0,44 &= \mathbf{0} + 0,88\\
2\times 0,88 &= \mathbf{1} + 0,76\\
2\times 0,76 &= \mathbf{1} + 0,52\\
2\times 0,52 &= \mathbf{1} + 0,04\\
2\times 0,04 &= \mathbf{0} + 0,08\\
2\times 0,08 &= \mathbf{0} + 0,16\\
2\times 0,16 &= \mathbf{0} + 0,32
\end{aligned}
$$

Le bit suivant aurait cette fois-ci été un 0.

Ici, le cycle aurait été de taille 20 en démarrant à $0,84$.

L'exposant est $-2$, et avec les mêmes calculs on tombe sur 0 01101 1010111000.

## Exercice 2

```
def somme_chiffres(n):
    somme = 0
    for i in str(abs(n)):
        somme += int(i)
    return somme

def somme_chiffres_alt(n):
    somme, signe = 0, 1
    for i in str(abs(n)):
        somme += signe * int(i)
        signe *= -1
    return somme

def somme_chiffres_fix(n):
    somme = somme_chiffres(n)
    while(somme >= 10):
        somme = somme_chiffres(somme)
    return somme

def div2(n):
    return str(n)[-1] in ['2','4','6','8','0']

def div3(n):
    return somme_chiffres_fix(n) in [0,3,6,9]
```

```
def div5(n):
    return str(n)[-1] in ['5','0']

def div9(n):
    return somme_chiffres_fix(n) in [0,9]

def div10(n):
    return str(n)[-1] == '0'

def div_11(n):
    somme = somme_chiffres_alt(abs(n))
    while(somme > 10 or somme < -10):
        somme = somme_chiffres_alt(abs(somme))
    return somme == 0
```

## Exercice 3

Le programme prend en entrée une chaîne de caractères (ou, avec une utilité moindre, une séquence de chaînes de caractères) et renvoie le symétrique de cette chaîne (ou la chaîne obtenue en collant les chaînes de la séquence dans l'ordre inverse).

La terminaison est gratuite en raison de l'absence de structures de contrôle pouvant poser un problème (pour le moment, cela se limite aux boucles conditionnelles).

La complexité est quadratique en la taille de `s` en considérant comme opération de base l'ajout d'un caractère à la fin d'une chaîne, ce qui est le choix le plus pertinent vu que le temps mis pour l'exécution de la fonction a lui-même plutôt tendance à être quadratique que linéaire. La raison est que `rep = car + rep` est un recopiage de tous les caractères de `car + rep`, et de plus cette expression nécessite autant d'ajouts de caractères à la fin d'une chaîne que `rep` a d'éléments.

La correction est évidente en posant un invariant de boucle classique : après un certain nombre de tours de boucle, une certaine proportion du travail est faite.

## Exercice 4

```
def recherche_dicho(l,x):
    n = len(l)
    if n == 0:
        return False
    ind_deb = 0
    ind_fin = n-1
    while (ind_deb <= ind_fin):
        ind_mil = (ind_deb + ind_fin)//2
        if l[ind_mil] == x:
            return True
        if l[ind_mil] < x:
            ind_deb = ind_mil+1
        else:
            ind_fin = ind_mil-1
    return False
```

La terminaison de la fonction nécessite de prouver la terminaison de la boucle conditionnelle, ce qui se fait au choix avec le variant `ind_fin - ind_deb` (facile) ou avec le variant $\lfloor \log_2(\texttt{ind\_fin - ind\_deb}) \rfloor$ (utile).

La complexité asymptotique se lit par le second variant : c'est au pire sa première valeur, donc le logarithme en base 2 de la taille de la liste.

La correction se fonde sur l'invariant de boucle suivant : à tout moment, si `x` est dans `l`, alors il y est entre `ind_deb` et `ind_fin`. Si on le trouve, c'est bon, sinon si `ind_deb` dépasse `ind_fin`, on peut laisser tomber.