Correction du DS 2

Julien Reichert

Exercice 1

Pour rappel, l'écriture en virgule flottante sur 32 bits utilise un bit de signe, puis 8 bits d'exposant et finalement 23 bits de mantisse.

La partie entière du résultat est de 5, on écrit par ailleurs $\frac{100}{17} = 5 + \frac{15}{17}$. Pour la suite, il est bien plus judicieux de n'utiliser que des fractions.

$$\begin{aligned}
2\times \tfrac{15}{17} &= 1 + \tfrac{13}{17}\\
2\times \tfrac{13}{17} &= 1 + \tfrac{9}{17}\\
2\times \tfrac{9}{17} &= 1 + \tfrac{1}{17}\\
2\times \tfrac{1}{17} &= 0 + \tfrac{2}{17}\\
2\times \tfrac{2}{17} &= 0 + \tfrac{4}{17}\\
2\times \tfrac{4}{17} &= 0 + \tfrac{8}{17}\\
2\times \tfrac{8}{17} &= 0 + \tfrac{16}{17}\\
2\times \tfrac{16}{17} &= 1 + \tfrac{15}{17}
\end{aligned}$$

Ici, la période a été trouvée. Elle était de taille 8, ce qui est cohérent avec la prévision qu'il s'agisse d'un diviseur de 16 (conséquence du petit théorème de Fermat). L'écriture du nombre $\frac{100}{17}$ en binaire est donc $\overline{101,1110000111100001\ldots}^2$, ce qui donne en écriture scientifique binaire $\overline{1,0111100001111000 01\ldots}^2 \times 2^2$.

L'exposant est 2, représenté sur 8 bits comme $2 + 2^{8-1} - 1$, soit $\overline{10000001}^2$. La mantisse commence par le 0 suivant immédiatement la virgule (le 1 étant implicite), elle est donc de $\overline{01111000011110000111100}$ et elle est arrondie par défaut.

La représentation finale est alors 0 10000001 01111000011110000111100.

Exercice 2

Une première version, très propre, sera à maîtriser et à savoir refaire à terme.

```
def century(annee):
    numero = (annee+99)//100
    if 3 < numero < 21:
        return str(numero) + "th"
    suffixes = ["th", "st", "nd", "rd"]
    indice = numero % 10
    if indice > 3:
        indice = 0
    return str(numero) + suffixes[indice]
```

Cette autre version n'est pas esthétique, mais elle est plus compréhensible et intuitive quand on débute.

```
def century(annee):
    numero = (annee+99)//100
    numerostr = str(numero)
    if numero in [11, 12, 13]:
        return numerostr + "th"
    elif numero % 10 == 1:
        return numerostr + "st"
    elif numero % 10 == 2:
        return numerostr + "nd"
    elif numero % 10 == 3:
        return numerostr + "rd"
    else:
        return numerostr + "st"
```

Exercice 3

Une version en une ligne est possible et son écriture « normale » en deux lignes devrait être compréhensible à ce niveau.

```
def filtre(l):
    neg = []
    pos = []
    for x in l:
        if x < 0:
            neg.append(x)
        elif x > 0:
            pos.append(x)
    return neg + [0] * (len(l)-len(neg)-len(pos)) + pos
```

Exercice 4

Pour le coup, on va laisser de côté la vérité mathématique derrière le problème.

```
def transforme(c1, c2):
    if c1 == c2:
        return c1
    l = ["R", "V", "B"]:
    l.remove(c1)
    l.remove(c2)
    return l[0]

def fusion(l):
    while len(l) > 1:
        ll = []
        for i in range(len(l)-1):
            ll.append(transforme(l[i], l[i+1]))
        l = ll
    return l[0]
```

La fonction `transforme` agit en temps constant, et la fonction fusion fait `n-1` tours de la boucle while sur une liste de taille `n`, chacun appelant `k-1` fois la fonction `transforme`, après deux accès à un élément de la liste, où `k` est la taille de la liste au moment de la boucle, pour un total en $\mathcal{O}(\mathtt{n}^2)$.

Exercice 5

Il est évident que $\bar{p}(k)$ diminue (strictement tant que c'est non nul) au fur et à mesure que k augmente, puisqu'un nombre strictement inférieur à 365 apparaît dans le produit au numérateur et une division par 365 s'effectue en plus. Du coup la liste en entrée est croissante, et une dichotomie n'est fiable que si elle agit sur une liste monotone.

La première fonction n'est pas correcte, car la valeur fin ne doit pas être exclue quand `l[mil] >= .5`. Sinon, pour la liste `[0., .6, 1.]` la réponse sera l'indice `0`, qui n'est pas celui où le seuil est dépassé. Les deux autres fonctions sont correctes en cas de terminaison car l'invariant « `l[max(deb-1, 0)]` $<$ `.5` $\leq$ `l[fin]` » est préservé (ici on rédige les preuves pour l'initialisation et l'hérédité, et on traite du `return` intempestif de la deuxième fonction), et cet invariant implique que `fin` est le premier indice de dépassement de la valeur `.5` à la sortie de la boucle, quand `deb` est égal à `fin`.

Par suite, le problème de la deuxième fonction, c'est que pour avoir `deb` égal à `fin` dans l'appel de la fonction sur une liste ne contenant pas exactement `.5`, il faut ramener l'une des deux variables à `mil` (seule modification qu'elles subissent) alors que `mil` vaut l'autre, c'est-à-dire quand les deux sont espacées d'un et donc par définition `mil` vaut `deb`. Malheureusement on peut être dans le cas où `l[deb]` est inférieur strictement à `.5` et donc c'est `deb` qui est ramené à `mil`, le tour de boucle étant alors inutile. Finalement avec la même liste `[0., .6, 1.]` on observe une boucle infinie. A contrario, la valeur `fin - deb` décroît strictement à chaque tour de boucle dans les deux autres fonctions (évident pour la première, vrai aussi pour la deuxième car `mil` est inférieur strict à `fin` en tant que moyenne arrondie vers le bas avec une valeur strictement inférieure), il s'agit donc d'un variant qui garantit la terminaison des fonctions.

La quatrième fonction a été bien commentée, tous ces commentaires sont vrais, la fonction termine (la taille de la liste considérée à chaque étape est notre variant) et elle est correcte. Cependant, la dichotomie ayant pour but d'avoir un nombre de comparaisons logarithmique en la taille de la liste, recopier la moitié des éléments au premier tour de boucle et de nouveau la moitié de la moitié au suivant, et ainsi de suite, occasionne un coût final linéaire en la taille de la liste, ce qui ruine l'intérêt de l'algorithme.

```
def seuil(n):
    p0 = 1
    k = 1
    while p0 > .5:
        p0 *= (n-k)/n
        k += 1
    return k
```