TP (S2) 6 — Représentation des nombres en machine

Objectifs

- Connaître et savoir manipuler les formats de représentions des entiers et des flottants.
- Connaître les limitations inhérentes à la représentation des nombres en machine.

1. REPRÉSENTATION DES ENTIERS

Exercice 1 [Sol 1]

1) 1.1) Déterminer à la main quel entier positif a pour écriture binaire $(1011001)_2$.

1.2) Soit L une liste dont tous les éléments sont soit 0 soit 1 :

$$L = [c_{N-1}, c_{N-2}, \dots, c_1, c_0].$$

Écrire une fonction `bin2nat(L : list) -> int` renvoyant l'entier positif d'écriture binaire $(c_{N-1}c_{N-2}\dots c_1c_0)_2$.
*La fonction utilisera l'algorithme de Horner, comme vu en cours pour minimiser le nombre de calculs, et vérifiera par une commande `assert` que `L` est bien une **liste** ne contenant que 0 **ou** 1.*

1.3) Utiliser cette fonction pour savoir quel entier positif a pour écriture binaire $(100110111001)_2$.

1.4) Proposer une version récursive de `bin2nat`.

2) 2.1) Déterminer à la main qu'elle est l'écriture binaire sur 10 bits de l'entier positif 100.

2.2) Écrire une fonction `nat2bin(n,N)`, où N est un entier naturel non nul et n est un entier positif représentable sur N bits, renvoyant la liste correspondant à l'écriture binaire de n. *La fonction vérifiera par une commande `assert` que* N *est un **entier naturel non nul** et que* n *est un **entier** représentable **sur** N **bits**.*

2.3) Utiliser cette fonction pour savoir quelle est l'écriture binaire sur 16 bits de l'entier naturel 12345 ?

2.4) Proposer une version récursive de `nat2bin`.

Exercice 2 [Sol 2]

1) On souhaite dans cette question mettre oeuvre l'algorithme usuel de calcul d'une somme de deux entiers positifs directement sur leurs écritures binaires.

1.1) Poser à la main l'addition binaire des deux nombres $(0110)_2$ et $(0111)_2$.

1.2) Écrire une fonction `SB3(b1 : int, b2 : int, b3 : int)-> tuple`, où b_1, b_2, b_3 sont tous les trois dans $\{0,1\}$, et renvoyant le tuple (c_1, c_0) tel que $b_1 + b_2 + b_3 = (c_1c_0)_2$.

Par exemple `SB3(1,0,1)` doit renvoyer la liste $(1,0)$ car $1+0+1 = 2 = (10)_2$.
Attention ! Le code de la fonction `SB3` ne contiendra aucun calcul d'addition, mais devra utiliser uniquement un test sur les différents cas possibles pour les valeurs de b_1, b_2, b_3, sous la forme :

```
if b1 == 1 and b2 == 0 and b3 == 0:
    return (0, 1)
elif ...
```

1.3) Écrire une fonction `addBin(L1 : list,L2 : list)-> list`, où `L1` et `L2` sont deux listes binaires **de la même taille** correspondant à l'écriture binaire de deux entiers naturels n_1 et n_2 (sur un même nombre N de bits), et renvoyant la liste correspondant à l'écriture binaire de la somme $n_1 + n_2$, toujours sur le même nombre N de bits. *Le calcul pourra donc ne pas être exact (si la dernière retenue n'est pas égale à* 0*).*

1.4) Écrire une fonction `addNat(n1 : int,n2 : int, N:int)-> int`, où $n1$ et $n2$ sont deux entiers naturels, et renvoyant l'entier naturel $n1+n2$, calculé en passant par leurs écritures binaires sur N bits(la fonction convertira donc d'abord $n1$ et $n2$ en binaire sur N bits, les additionnera en binaire, puis reconvertira le résultat en décimal ; elle utilisera bien sûr les fonctions de l'exercice 1).

1.5) Tester la fonction précédente en choisissant certains calculs dont les résultats seront exacts, et d'autres qui ne seront vrais que modulo 2^N.

2) Écrire une fonction `mulBin(L1 : list,L2 : list)-> list` effectuant, sur le même principe, cette fois une multiplication en binaire, et écrire la fonction `mulNat(n1 : int, n2 : int, N : int)-> int` correspondante.

3) Écrire une fonction `sousBin(L1 : list,L2 : list)-> list` effectuant, toujours sur le même principe, une soustraction en binaire, et écrire la fonction `sousNat(n1 : int, n2 : int, N : int)-> int` correspondante. *(on supposera que l'entier codé par `L1` est supérieur ou égal à celui codé par `L2`)*

Exercice 3 [Sol 3] Écrire une fonction **récursive** `infBinNat(L1 : list, L2 : list) -> bool`, où `L1` et `L2` sont deux listes binaires de même taille représentant les entiers positifs n_1 et n_2 respectivement, renvoyant **`True`** si $n1 \leqslant n2$ et **`False`** sinon. *L'unique test python autorisé ici pour le code est le test == entre deux bits.*

1.1. Les entiers signés

Exercice 4 [Sol 4]

1) On choisit de coder les entiers relatifs sur 10 bits. *Vous pouvez bien sûr vous servir des fonctions `nat2bin` et `bin2nat` en guise de calculatrice.*

1.1) Quel intervalle d'entiers cela permet-il de coder ?

1.2) Quel entier relatif est codé par 0101010101 ?

1.3) Quel entier relatif est codé par 1010101010 ?

1.4) Quel est le codage de 123 ?

1.5) Quel est le codage de -123 ?

2) **2.1)** En utilisant la fonction `bin2nat` codée en exercice 1, écrire une fonction `bin2relat(L : list)->int` renvoyant l'entier relatif d'écriture binaire stockée dans L (on utilisera ici la méthode « naïve » sans complément à 2). Contrôler en retrouvant les résultats de 1.2) et 1.3).

2.2) En utilisant la fonction `nat2bin` codée en exercice 1, écrire une fonction `rel2bin(n : int, N : int)->list`, où $N \in \mathbb{N}^*$ et $n \in [\![-2^{N-1}; 2^{N-1}-1]\!]$, renvoyant la liste L correspondant à l'écriture binaire de n sur N bits (on utilisera la méthode « naïve » sans complément à 2). Contrôler en retrouvant les résultats 1.4) et 1.5).

2.3) En utilisant les fonction `addBin` et `mulBin` codées en exercice 2, écrire les fonctions : `addRelat(n1 : int,n2 : int,N : int)->int` et `mulRelat(n1 : int,n2 : int,N : int)->int`, où $N \in \mathbb{N}^*$ et `n1,n2` sont dans $[\![-2^{N-1}; 2^{N-1}-1]\!]$, et renvoyant respectivement les entiers relatifs `n1+n2` et `n1×n2`, toujours dans $[\![-2^{N-1}; 2^{N-1}-1]\!]$ (ici encore, le résultat ne sera a priori correct que modulo 2^N).

3) Écrire de nouvelles versions des fonctions `rel2bin` et `bin2relat`, en utilisant cette fois la technique du complément à 2.

Exercice 5 [Sol 5] En utilisant la fonction `infBinNat` écrite en exercice 3, écrire une fonction `infBinRelat(L1 : list, L2 : list) -> bool`, où `L1` et `L2` sont deux listes binaires de même taille représentant les entiers signés n_1 et n_2 respectivement, renvoyant **`True`** si $n1 \leqslant n2$ et **`False`** sinon.

Exercice 6 [Sol 6] Si L est une liste binaire représentant un entier relatif n, on vérifie facilement que son complément à 2 (inverser tous les bits et ajouter 1 au résultat) permet d'obtenir la liste binaire représentant l'entier $-n$.

1) Écrire une fonction `moins(L : list)->list` effectuant ce traitement.

2) Écrire une fonction `valeurAbsolue(L : list)->list` renvoyant la liste binaire correspondant à la valeur absolue de l'entier relatif codé par `L`.

Exercice 7 [Sol 7]

1) Écrire une fonction **récursive** `pgcd2(a ; int,b : int)->int` où a et b sont deux entiers naturels, qui calcule et renvoie le pgcd de a et b en n'utilisant que des multiplications par 2, des divisions par 2 et des soustractions.
On s'appuiera sur les propriétés suivantes (que vous pourrez démontrer à titre d'exercice de Mathématiques) :

- $\text{pgcd}(a,b) = 2 \times \text{pgcd}\left(\frac{a}{2}, \frac{b}{2}\right)$ si a et b sont tous les deux pairs ;
- $\text{pgcd}(a,b) = \text{pgcd}\left(a, \frac{b}{2}\right)$ si a est impair et b est pair ;
- $\text{pgcd}(a,b) = \text{pgcd}(a-b,b)$ si a et b sont tous les deux impairs.

2) Sur le même principe, écrire une fonction **récursive** `pgcdBinNat(L1 : list,L2 : list)->list` où L1 et L1 sont deux listes binaires représentant des entiers **naturels** a et b respectivement, sur le même nombre de bits, et renvoyant une liste binaire correspondant au pgcd de a et b, toujours sur le même nombre de bits **tous les calculs étant faits directement sur les listes binaires**.

3) Écrire une fonction `pgcdBinRelat(L1 : list,L2 : list)->list` pour laquelle les listes `L1` et `L2` représentent cette fois des entiers relatifs.

2. REPRÉSENTATION DES RÉELS

Exercice 8 **Conversion en binaire** [Sol 8] On désire réaliser la conversion entre l'écriture des nombres réels en base décimale et la représentation binaire en virgule flottante.

1) On se restreint ici à des mantisses de taille $m = 4$ (exposant quelconque).

1.1) Déterminer les réels positifs dont les représentations binaires sont :

(a) E = 2, M = 1.1010 ; (b) E = −3, M = 1.1100

1.2) Donner la représentation binaire des réels suivants :

(a) $x_2 = 9.0$; (b) $x_4 = \frac{5}{32}$; (c) $x_5 = 6$; (d) $x_6 = 6.125$

2) On souhaite écrire une fonction `dec2binR(x, m)` qui prend en argument un réel x, un entier m et qui renvoie un tuple (s, E, B) où s vaut ± 1, E est un entier relatif et B une chaîne de caractères constituée de 0 et de 1, de la forme $b_1 b_2 \dots b_m$ et codant les décimales binaires de la mantisse M. On rappelle qu'on a alors $x = (-1)^s 2^{\text{E}} \text{M} = (-1)^s 2^{\text{E}}(1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \dots + b_m \times 2^{-m})$. On a par construction $1 \leqslant \text{M} < 2$, et nous avons donné dans le cours les algorithmes suivants :

Calcul de E et M

Données : Un réel x

Résultat : La mantisse de x
M ← $|x|$, E ← 0.
- Tant que M < 1 faire : M ← 2M et E ← E − 1.
- Tant que M ⩾ 2 faire : M ← M/2 et E ← E + 1.

renvoyer E, M.

Calcul de la représentation binaire de la mantisse

Données : Un réel M ∈ [1, 2[

Résultat : La liste des $(b_i)_{1 \leqslant i \leqslant m}$
b = [] et y = M − 1
pour i allant de 1 à m faire :
- Si $y \geqslant 0.5$, $b \leftarrow b + [1]$, $y \leftarrow 2y - 1$.
- Sinon, $b \leftarrow b + [0]$, $y \leftarrow 2y$.

renvoyer b.

2.1) Écrire la fonction demandée en utilisant les résultats précédents.

2.2) La mantisse de 0.1 est 4-périodique, mais que constate-t-on si on affiche le résultat de `dec2binR(0.1,60)` ? Expliquer. Comment pourrait-on contourner ce problème ?

Exercice 9 **Norme IEEE754** [Sol 9] On cherche dans cet exercice à déterminer numériquement les valeurs des nombres de bits e et m utilisés respectivement pour le codage de l'exposant E et de la mantisse M pour les réels.

1) Pour $x = 1.0 \times 2^{\text{E}}$ et $\text{E} \in \mathbb{N}$, et pour E suffisamment grand, le test `x+1 == x` renvoie la valeur `True`.

1.1) Justifier ce comportement.

1.2) Écrire une fonction `mantisse()`, basée sur cette propriété, permettant de déterminer l'entier m.

1.3) Quel serait le comportement de ce dernier algorithme avec comme valeur de départ $x = 2^{\text{E}}$?

2) On rappelle que l'exposant E a pour valeur maximale $\text{E}_{\max} = 2^{e-1} - 1$ pour les nombres normalisés, la valeur $\text{E} = 2^{e-1}$ étant réservée au codage de l'infini (noté `inf`).

2.1) Que donnent la série d'instructions suivantes : *(à tester dans la console)*

```
>>> x = 1.0*2**1023
>>> x = x*2
>>> x
>>> x = x/2
>>> x
```

2.2) Écrire une fonction `exposant`, basée sur les propriétés de la valeur `inf` vis à vis de la division, qui permette de déterminer l'entier e. *On s'interdira d'utiliser un test d'égalité à* `float('inf')`.

Exercice 10 **Comparaison à zéro** [Sol 10] On donne les deux scripts suivants :

```
def test1():
    n = 10
    k = 0
    while n != 0 and k < 50:
        n -= 1
        k += 1
        print(n)
```

```
def test2():
    n = 1.
    k = 0
    while n != 0 and k < 50:
        n -= 0.1
        k += 1
        print(n)
```

1) Prédire le comportement de chacun des scripts et expliquer les différences de comportement.

2) Que dire du test de comparaison à zéro lorsqu'on manipule des flottants ?

3) Modifier le second algorithme pour qu'il termine quand la variable n est le plus près de zéro possible en valeur absolue.

Exercice 11 **Associativité** [Sol 11]

1) Pour $x = 1.0 \times 2^{53}$ ($\approx 1.0 \times 10^{16}$), calculer $-x + (x+1)$ et $(-x+x)+1$. Expliquer le résultat. Quelle propriété usuelle de l'addition a-t-on perdu numériquement ?

2) Évaluer de même $\dfrac{(y+x)-x}{y}$ et $\dfrac{y+(x-x)}{y}$, et ce pour $x = 1.0 \times 10^{n}$, $y = 10^{-n}$, et $n = 6, n = 7, n = 8, n = 9$.

Exercice 12 **Test d'égalité** [Sol 12]

1) Que donne le test `0.1+0.05 == 0.15` ?

2) On cherche à expliquer le résultat suivant, et on pose $x = 0.1$, $y = 0.05$, $z = 0.15$. On note par ailleurs x_r, y_r et z_r les réels représentés associés aux réels x, y et z.

2.1) Donner les exposant et la mantisse (en binaire) des représentations de x, y et z, en se limitant à 4 bits pour les mantisses.

2.2) Calculer la somme binaire de x_r et y_r. Cela explique-t-il le résultat du test ?

3. APPLICATIONS

Exercice 13 **Résolution d'une équation du second degré** [Sol 13] On évalue numériquement les solutions x_1, x_2 de l'équation : $ax^2 + bx + c = 0$, par application des formules théoriques. On se limite au cas où $\Delta = b^2 - 4ac > 0$, pour lequel on a :

$$x_1 = \frac{-b+\sqrt{\Delta}}{2a}, \quad x_2 = \frac{-b-\sqrt{\Delta}}{2a}, \quad (1)$$

1) Écrire la fonction `racines(a,b,c)` qui renvoie x_1 et x_2 en appliquant (1).

2) On considère l'équation : $\varepsilon x^2 - \dfrac{1}{\varepsilon}x + \varepsilon = 0$. Pour $\varepsilon < 1/\sqrt{2}$, on a $\Delta > 0$. Pour $\varepsilon \to 0$, on montre que les racines x_1, x_2 vérifient : $x_1 \approx \dfrac{1}{\varepsilon^2}$ et $x_2 \approx \varepsilon^2$.

2.1) Évaluer les solutions de l'équation à l'aide de la fonction `racines(a,b,c)` , pour $\varepsilon = 2^{-10}, \varepsilon = 2^{-11}, \cdots$ et comparer aux valeurs obtenues par l'approximation précédente. Expliquer.

2.2) Pour contourner ce problème, on peut se contenter de calculer x_1 par (1), puis d'utiliser la relation sur le produit des racines : $x_1 x_2 = \dfrac{c}{a}$.
Écrire une fonction `racines2(a, b, c)` qui renvoie les racines calculées par cette méthode, et l'utiliser pour $\varepsilon = 2^{-10}, \varepsilon = 2^{-11}, \cdots$. Comparer aux résultats attendus par l'approximation. Expliquer.

3) On considère l'équation $x^2 - \frac{1}{\varepsilon}x + 1 = 0$. Pour $\varepsilon < \frac{1}{2}$, on a bien $\Delta > 0$.

Pour $\varepsilon \to 0$, on montre que les racines vérifient : $x_1 \approx \frac{1}{\varepsilon}$ et $x_2 \approx \varepsilon$.

3.1) Rappeler la limite du plus petit réel positif normalisé représentable.

3.2) Résoudre l'équation pour des valeurs de ε proches mais supérieures à cette limite. Expliquer les résultats observés.

Exercice 14 **Polynomes de Rump** [Sol 14] Les polynômes de Rump (polynômes en x et fractions rationnelles en y) ont pour expression : $R(x,y) = \frac{1335}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$.

Nous allons comparer les résultats en suivant deux modes d'évaluation de l'expression.

1) **1.1)** Écrire la fonction `rump1(x:int, y:int)->float` qui renvoie la valeur de $R(x,y)$ en utilisant l'expression ci-dessus.

1.2) Noter le résultat de `rump1(77617,33096)`.

2) On peut réécrire $R(x,y)$ sous la forme :

$$\frac{1}{4}(22y^8 - 4y^6x^2 + 1335y^6 - 484y^4x^2 + 44y^2x^4 - 8x^2) + \frac{x}{2y}.$$

2.1) Écrire une fonction `F(x:int, y:int)->int` qui calcule et renvoie l'expression entre parenthèses.

2.2) En déduire une fonction `rump2(x:int, y:int)->float)` qui utilise $F(x,y)$.

2.3) Comparer `rump2(77617,33096)` avec `rump1(77617,33096)`. Lequel de ces deux résultats vous paraît le plus fiable ? Expliquer.

3) On veut visualiser les écarts $|\text{rump1}(x,y) - \text{rump2}(x,y)|$ pour N valeurs entières de x et y choisies au hasard dans un intervalle de la forme $[-d;d]$ (N et d seront choisies par l'utilisateur). On fera appel à la bibliothèque `matplotlib` à l'aide l'instruction `import matplotlib.pyplot as plt`, et à la fonction `randint` du module `random`.

Écrire la fonction `test(N:int, d:int)->None` qui construit aléatoirement une liste de N couples $(x_i, y_i) \in [-d;d]^2$, puis qui représente sur un graphique les entiers i de 1 à N en abscisses, et en ordonnées les écarts $|\text{rump1}(x_i,y_i) - \text{rump2}(x_i,y_i)|$.

Exercice 15 **Suite convergente ou divergente ?** [Sol 15] Soit $q = \frac{1-\sqrt{5}}{2}$. On définit deux suites u et v en posant :

$$u_0 = 1,\ v_0 = 1,\ u_1 = q \text{ et pour tout } n,\ u_{n+2} = u_{n+1} + u_n \text{ et } v_{n+1} = qv_n.$$

1) **1.1)** Écrire la fonction `comparer(n: int) -> None` qui affiche à l'écran, pour k allant de 0 à n, la valeur de u_k et à côté, celle de v_k.

1.2) À l'aide de cette fonction faire une conjecture sur le comportement asymptotique des deux suites.

2) **2.1)** Déterminer l'expression de u_n et celle de v_n en fonction de n. Que remarquez-vous ?

2.2) Expliquer les résultats obtenus avec la fonction `comparer`.

SOLUTIONS DES EXERCICES

Solution 1

1) 1.1) Il s'agit de 89.

1.2)
```
def bin2nat(L : list) -> int :
    assert type(L) == list
    for e in L :
        assert e == 0 or e == 1
    n = 0
    for e in L :
        n = 2*n+e
    return n
```

1.3) Il s'agit de 2489.

```
>>> bin2nat([1,0,0,1,1,0,1,1,1,0,0,1])
2489
```

1.4)
```
def bin2nat(L : list) -> int :
    assert type(L) == list
    for e in L :
        assert e == 0 or e == 1
    if len(L) == 0:
        return 0
    else:
        return 2*bin2nat(L[:-1]) + L[-1]
```

```
>>> bin2nat([1,0,0,1,1,0,1,1,1,0,0,1])
2489
```

2) 2.1) Il s'agit de $[0,0,0,1,1,0,0,1,0,0]$.

2.2)
```
def nat2bin(n : int, N : int)-> list :
    assert type(N) == int
    assert N >= 1
    assert type(n) == int
    assert 0 <= n <= 2**N-1
    L = [0]*N
    i = N-1
    while n != 0:
        L[i] = n%2
        n = n//2
        i -= 1
    return L
```

2.3) Il s'agit de :

```
>>> nat2bin(12345, 16)
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1]
```

2.4) En cas terminal, bien mettre `N == 1` sinon le `assert` va poser soucis.

```
def nat2bin(n : int, N : int)-> list :
    assert type(N) == int
    assert N >= 1
    assert type(n) == int
    assert 0 <= n <= 2**N-1
    if N == 1:
        return [n%2]
    else:
        return nat2bin(n//2, N-1) + [n%2]
```

2.5) Il s'agit de :

```
>>> nat2bin(12345, 16)
[0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1]
```

Solution 2

1) 1.1) On trouve $(1101)_2$.

1.2)
```
def SB3(b1 : int, b2 : int, b3 : int) -> tuple :
    if b1 == 0 and b2 == 0 and b3 == 0 :
        return (0,0)
    elif b1 == 0 and b2 == 0 and b3 == 1:
        return (0,1)
    elif b1 == 0 and b2 == 1 and b3 == 0:
        return (0,1)
    elif b1 == 1 and b2 == 0 and b3 == 0:
        return (0,1)
    elif b1 == 1 and b2 == 1 and b3 == 0:
        return (1,0)
    elif b1 == 1 and b2 == 0 and b3 == 1:
        return (1,0)
```

```
    elif b1 == 0 and b2 == 1 and b3 == 1:
        return (1,0)
    else :
        return (1,1)
```

1.3)

```
def addBin(L1 : list, L2 : list) -> list :
    N = len(L1)
    S = [0]*N
    r = 0
    for k in range(N) :
        r, S[N-1-k] = SB3(L1[N-1-k], L2[N-1-k], r)
    return S
```

1.4)

```
def addNat(n1 : int, n2 : int, N : int) -> int :
    return bin2nat(addBin(nat2bin(n1,N),nat2bin(n2,N)))
```

1.5)

```
>>> 12+23
35
>>> addNat(12, 23, 10)
35
>>> (12+23)% 2**5
3
>>> addNat(12, 23, 5) # dépassement de capacité : vraie \
↪ modulo 2**N
3
```

2)

```
def mulBin(L1 : list, L2 : list) -> list :
    N = len(L1)
    P = [0]*N
    # Dans le schéma de multiplication : L1 = liste du dessus, \
    ↪ L2 = liste du dessous
    for k in range(N):
        if L2[N-1-k] == 1:
            P = addBin(P, L1)
        L1 = L1[1:]+[0] # on oublie le bit de droite de L1 (on \
        ↪ doit coder sur N bits uniquement), et on met un \
        ↪ zéro à la place.
        # Remarque : L1 n'est pas modifiée en dehors de cette \
        ↪ fonction (slicing = variable locale dans la fonction)
    return P

def mulNat(n1 : int, n2 : int, N : int) -> int :
    return bin2nat(mulBin(nat2bin(n1,N), nat2bin(n2,N)))
```

```
>>> 12*23
276
>>> mulNat(12, 23, 10)
276
>>> (12*23)% 2**5
20
>>> mulNat(12, 23, 5) # dépassement de capacité : vraie modulo \
↪ 2**N
20
```

3)

```
def sousBin(L1 : list, L2 : list) -> list :
    N = len(L1)
    S = [0]*N
    S[N-1] = 1
    i = N-1
    for k in range(N) :
        if L1[i] == L2[i]:
            S[i] = 0
        elif L1[i] == 1 and L2[i] == 0:
            S[i] = 1
        else :
            S[i] = 1
            L = addBin(L2[:i],nat2bin(1,i))
            L2 = L+L2[i:]
        i -= 1
    return S

def sousNat(n1 : int, n2 : int, N : int) -> int :
    return bin2nat(sousBin(nat2bin(n1,N), nat2bin(n2,N)))
```

Solution 3

```
def infBinNat(L1 : list, L2 : list) -> bool :
    if len(L1) == 0:
        return True
    elif L1[0] == 0 and L2[0] == 1:
        return True
    elif L1[0] == 1 and L2[0] == 0:
        return False
```

```
    else :
        return infBin(L1[1:], L2[1:])
```

Solution 4

1) 1.1) $[\![-2^9; 2^9-1]\!] = [\![-512; 511]\!]$.

1.2) Comme le bit de poids fort est 0, l'entier est positif et codé directement : il s'agit de 341.

1.3) Comme le bit de poids fort est 1, l'entier est négatif, et on retranche donc $2^{10} = 1024$ à 682 : il s'agit de -342.

1.4) Comme 123 est positif, il est codé directement : 0001111011.

1.5) Comme -123 est négatif, on ajoute 1024, ce qui donne 901, que l'on code 1110000101.
Rq : pour les entiers négatifs, on peut bien sûr aussi utiliser les compléments à 2.

2) 2.1)

```
def bin2relat(L : list)->int :
    N = len(L)
    n = bin2nat(L)
    if L[0] == 0 :
        return n
    else :
        return n-2**N
```

```
>>> bin2relat([0,1,0,1,0,1,0,1,0,1])
341
>>> bin2relat([1,0,1,0,1,0,1,0,1,0])
-342
```

2.2)

```
def rel2bin(n : int, N : int)->list :
    if n >= 0:
        L = nat2bin(n, N)
    else :
        L = nat2bin(n+2**N, N)
    return L
```

```
>>> rel2bin(123, 8)
[0, 1, 1, 1, 1, 0, 1, 1]
>>> rel2bin(-123, 8)
[1, 0, 0, 0, 0, 1, 0, 1]
```

2.3)

```
def addRelat(n1 : int,n2 : int,N : int)->int :
    return bin2relat(addBin(rel2bin(n1,N),rel2bin(n2,N)))
```

```
>>> addRelat(3, -4, 10)
-1
>>> addRelat(-3, -4, 10)
-7
>>> addRelat(-3, 20, 10)
17
```

```
def mulRelat(n1 : int,n2 : int,N : int)->int :
    return bin2relat(mulBin(rel2bin(n1,N,rel2bin(n2,N))))
```

3)

```
def rel2bin(n : int,N : int)->list :
    if n >= 0:
        return nat2bin(n,N)
    else :
        L = nat2bin(-n,N)
        CL = [1-e for e in L]
        return addBin(CL,nat2bin(1,N))

def bin2relat(L : list)-> int :
    N = len(L)
    if L[0] == 0:
        return bin2nat(L)
    else :
        CL = [1-e for e in L]
        return -bin2nat(addBin(CL,nat2bin(1,N)))
```

Solution 5

```
def infBinRelat(L1 : list, L2 : list) -> bool :
    if L1[0] == L2[0] :
        return infBinNat(L1,L2)
    elif L1[0] == 0 and L2[0] == 1:
        return False
    elif L1[0] == 1 and L2[0] == 0:
        return True
```

Solution 6

1)
```python
def moins(L : list)->list :
    N = len(L)
    CL = [1-e for e in L]
    return addBin(CL,nat2bin(1,N))
```

2)
```python
def valeurAbsolue(L : list)->list :
    if L[0] == 0:
        return L
    else :
        return moins(L)
```

Solution 7

1)
```python
def pgcd2(a : int,b : int)-> int :
    if a < b :
        return pgcd2(b,a)
    elif b == 0 :
        return a
    elif a%2 == 0 and b%2 == 0 :
        return 2*pgcd2(a//2,b//2)
    elif b%2 == 0:
        return pgcd2(a,b//2)
    elif a%2 == 0:
        return pgcd2(a//2,b)
    else :
        return pgcd2(a-b,b)
```

```python
def pgcdBinNat(L1 : list, L2 : list)->list :
    if infBinNat(L1,L2) and L1 != L2 :
        return pgcdBinNat(L2,L1)
    elif L2 == [0]*len(L2) :
        return L1
    elif L1[-1] == 0 and L2[-1] == 0 :
        R = pgcdBinNat(L1[:-1],L2[:-1])
        R.append(0)
        return R
    elif L2[-1] == 0:
        L2 = [0]+L2[:-1]
        return pgcdBinNat(L1,L2)
    elif L1[-1] == 0 :
        L1 = [0]+L1[:-1]
        return pgcdBinNat(L1,L2)
    else :
        return pgcdBinNat(sousBin(L1,L2,L2))
```

3)
```python
def pgcdBinRelat(L1 : list, L2 : list)->list :
    return pgcdBinNat(valeurAbsolue(L1,valeurAbsolue(L2)))
```

Solution 8

1) **1.1)** (a) $x = 2^2(1+2^{-1}+2^{-3}) = 6.5$, (b) $x = 2^{-3}(1+2^{-1}+2^{-2}) = 0.21875$

1.2) (a) $9 = 8+1 = 2^3+1 = 2^3(1+2^{-3})$. On a donc E = 3 et M = 0010; (b) $\frac{5}{32} = \frac{4}{32}+\frac{1}{32} = \frac{1}{8}+\frac{1}{32} = 2^{-3}+2^{-5} = 2^{-3}(1+2^{-2})$, d'où E = −3, M = 0100; (c) $6 = 2^2(1+2^{-1})$, d'où E = 2, M = 1000; (d) $6.125 = 2^2(1+2^{-1}+2^{-5})$, on a donc E = 2, M = 1000 (pas de bit b_5). Les deux derniers nombres ont même représentation sous 4 bits.

2) **2.1)** On peut proposer la fonction suivante :

```python
def dec2binR(x: float, m: int)->(int,int,str):
    #codage du signe
    if x >= 0:
        s = 1
    else:
        s = -1
    #codage de l'exposant
    E = 0
    M = abs(x)
    while M >=2:
        E += 1
        M = M/2
    while M<1 :
        E -= 1
        M = 2*M
    #codage de la mantisse
    B = ''
    y = M-1
    for i in range(1, m+1):
        y *= 2
        if y >= 1:
```

```
            B = B+'1'
            y -= 1
        else:
            B = B+'0'
    return s, E, B
```

2.2) On constate des zéros à partir de b_{53} et non plus la période attendue. En réalité la fonction ne fait pas les calculs pour 0.1, mais pour le réel qui représente 0.1 en machine! Sa mantisse étant codée sur 52 bits, c'est comme si ceux qui suivent étaient tous nuls ... On pourrait contourner ce problème en programmant le codage binaire des rationnels a/b, mais sans calculer a/b :

```
def rat2bin(a,b,m):
    bits=''
    E=0
    # calcul du signe
    s = 1
    if a < 0:
        s = -s
        a = -a
    if b < 0:
        s = -s
        b = -b
    #calcul de E
    while a >= 2*b:
        E += 1
        b *= 2
    while a<b:
        E -= 1
        a *= 2
    # calcul du numérateur de y = M-1
    a -= b
    # calcul de b_1, ..., b_m
    for i in range(1, m+1):
        a *= 2
        if a >= b:
            bits += '1'
            a -= b #b_i=1
        else :
            bits += '0' #b_i=0
    return s, E, bits
```

Solution 9

1) 1.1) On a numériquement $x+1=x$ quand la représentation de $x+1$ « absorbe » 1 (c'est-à-dire quand elle est égale à celle de x), on a $x+1=2^E+1=2^E\times(1+2^{-E})$ qui sera représenté par 2^E si $E > m$, car dans ce cas la décimale sur 2^{-E} est absorbée.

1.2) Pour déterminer ce nombre, on peut donc proposer :

```
def mantisse()->int:
    m = 1
    x = 2.0
    while x + 1 != x:
        x *= 2
        m += 1
    return m-1
```

```
>>> mantisse()
52
```

1.3) Avec $x=2^E$, on travaille sur des valeurs exactes, et on boucle de manière infinie en Python.

2) 2.1) On obtient `inf` comme valeur renvoyée dans les deux cas.

```
>>> x = 1.0*2**1023
>>> x = x*2
>>> x
inf
>>> x = x/2
>>> x
inf
```

2.2) On peut utiliser le fait que pour $x=1.0\times 2^{E_{max}}$, on a $(2\times x)/2$ renvoie `inf`, différent de x.

```
from math import log
def exposant():
    p = 1
    x = 2.0
    while x/2 != x:
        x *= 2
        p += 1
    return log(p, 2) + 1 #nb de bits pour coder p=2^{e-1}
```

```
>>> exposant()
11.0
```

Solution 10

1) Le premier s'arrête au bout de dix itérations, car la variable n prend exactement la valeur zéro. Pour le deuxième, ce n'est pas le cas, et c'est le second test qui permet d'arrêter la boucle.

2) Ne fonctionne pas comme on pourrait l'attendre par une étude formelle (ne tenant pas compte de la représentation)

3) On peut proposer un test avec un epsilon.

```
def test3():
    n = 1.
    eps = 0.001
    k = 0
    while abs(n) > eps and k < 50:
        n -= 0.1
        k += 1
        print(n)
```

Solution 11

1) On trouve 0 et 1 pour le second. En effet, $x+1 = 2^{53}[1+\frac{1}{2^{53}}]$, la mantisse étant codée sur 52 bits, le terme en $\frac{1}{2^{53}}$ n'est pas représenté. La propriété perdue est l'associativité.

2) La deuxième expression vaut toujours 1. Pour la première on somme des réels de valeurs très éloignées, ce qui conduit à des erreurs.

Solution 12

1) Le test renvoie **`False`**.

2) **2.1)** On a (on peut utiliser la fonction `dec2binR`) :

- pour x : E = -4, M = 1.1001
- pour y : E = -5, M = 1.1001
- pour z : E = -3, M = 1.0011

2.2) On a :

$$\begin{aligned} x+y &= 2^{-4}\left(1+\frac{1}{2}+\frac{1}{2^4}\right)+2^{-5}\left(1+\frac{1}{2}+\frac{1}{2^4}\right)\\ &= 2^{-4}\left(1+\frac{1}{2}+\frac{1}{2^4}+\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^5}\right)\\ &= 2^{-4}\left(2+\frac{1}{2^2}+\frac{1}{2^4}+\frac{1}{2^5}\right)\\ &= 2^{-3}\left(1+\frac{1}{2^3}+\frac{1}{2^5}+\frac{1}{2^6}\right). \end{aligned}$$

Le résultat est donc représenté avec E = -3 et M = 1.0010 qui n'est pas la représentation de 0.15.

Solution 13

1)

```
from math import sqrt
def racines(a, b, c):
    d = b**2-4*a*c
    if d >= 0:
        return (-b+sqrt(d)/(2*a), (-b-sqrt(d))/(2*a)) #x1 et x2
    else:
        return None
```

2) **2.1)**

```
>>> eps = 2**(-10)
>>> a, b, c = eps, -1/eps, eps
>>> racines(a, b, c)
(525311.9999990463, 9.5367431640625e-07)
>>> eps = 2**(-11)
>>> a, b, c = eps, -1/eps, eps
>>> racines(a, b, c)
(2099199.9999997616, 2.384185791015625e-07)
>>> eps = 2**(-12)
>>> a, b, c = eps, -1/eps, eps
>>> racines(a, b, c)
(8392703.99999994, 5.960464477539063e-08)
>>> eps = 2**(-13)
>>> a, b, c = eps, -1/eps, eps
>>> racines(a, b, c)
(33562623.999999985, 1.4901161193847656e-08)
>>> eps = 2**(-14)
>>> a, b, c = eps, -1/eps, eps
>>> racines(a, b, c)
```

```
(134234112.0, 0.0)
```

En posant $\varepsilon = 2^{-p}$, on a :

$$\Delta = \frac{1}{\varepsilon^2} - 4\varepsilon^2 = 2^{2p} - 2^{2-2p} = 2^{2p-1}\left(2 - \frac{1}{2^{4p-3}}\right).$$

Or $2 - \dfrac{1}{2^{4p-3}}$ est arrondi à 2 dès que $\dfrac{1}{2^{4p-3}}$ est arrondi à zéro, donc dès que $4p - 3 \geqslant 53$, et donc dès que $p \geqslant 14$, dès lors Δ est représenté par $\dfrac{1}{\varepsilon^2}$ ce qui entraîne $x_2 = 0$.

2.2)
```
def racines2(a, b, c):
    d = b**2-4*a*c
    if d >= 0:
        x1 = (-b+sqrt(d))/(2*a)
        return (x1, 1/x1)
    else:
        return None
```

```
>>> eps = 2**(-10)
>>> a, b, c = eps, -1/eps, eps
>>> racines2(a, b, c)
(1048575.9999990463, 9.536743164071174e-07)
>>> eps = 2**(-11)
>>> a, b, c = eps, -1/eps, eps
>>> racines2(a, b, c)
(4194303.9999997616, 2.3841857910157605e-07)
>>> eps = 2**(-12)
>>> a, b, c = eps, -1/eps, eps
>>> racines2(a, b, c)
(16777215.99999994, 5.960464477539084e-08)
>>> eps = 2**(-13)
>>> a, b, c = eps, -1/eps, eps
>>> racines2(a, b, c)
(67108863.999999985, 1.490116119384766e-08)
>>> eps = 2**(-14)
>>> a, b, c = eps, -1/eps, eps
>>> racines2(a, b, c)
(268435456.0, 3.725290298461914e-09)
```

x_1 est évalué sans problème, et x_2 s'obtient sans problème par cette méthode.

3) 3.1) On a $x_{\min} = 1.0 \times 2^{-1022} \approx 2,225 \times 10^{-308}$.

3.2) Comme précédemment, en posant $\varepsilon = 2^{-p}$, on a :

$$\Delta = \frac{1}{\varepsilon^2} - 4 = 2^{2p} - 2^2 = 2^{2p-1}(2 - \frac{1}{2^{2p-3}}),$$

or $2 - \dfrac{1}{2^{2p-3}}$ est arrondi à 2 dès que $2p - 3 \geqslant 53$ et donc dès que $p \geqslant 28$, dès lors Δ est représenté par $\dfrac{1}{\varepsilon^2}$ ce qui entraîne $x_2 = 0$, et on a une erreur (bien que le résultat final soit représentable).

Solution 14

1) 1.1) La fontction `rump1` :

```
def rump1(x: int, y:int) -> float:
    a = 1335/4
    b = 11/2
    return a*y**6 + x**2*(11*x**2*y**2 - y**6 - 121*y**4 - \
    ↪ 2) + b*y**8 + x/(2*y)
```

1.2) `rump1(77617,33096)` renvoie $1.1805916207174113e+21$.

2) 2.1) La fonction `F(x,y)` :

```
def F(x:int, y:int)->int:
    return 22*y**8 - 4*y**6*x**2 + 1335*y**6 - \
    ↪ 484*y**4*x**2 + 44*y**2*x**4 - 8*x**2
```

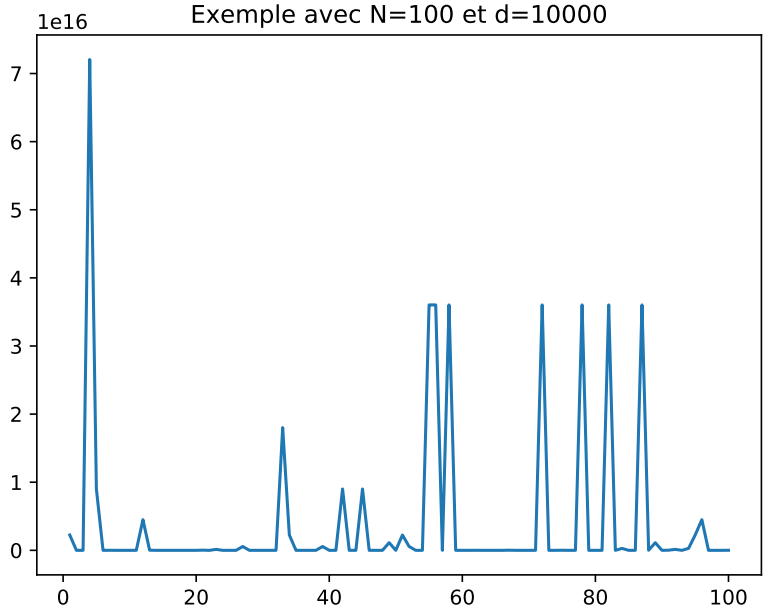
2.2) La fonction `rump2` :

```python
def rump2(x: int, y:int) -> float:
    return (F(x,y)+x*2/y)/4
```

```python
import numpy as np
import matplotlib.pyplot as plt
from random import randint

def test(N:int, d:int)->None :
    val = []
    for _ in range(N):
        p, q = randint(-d,d), randint(-d,d)
        while q == 0:
            q = randint(-d,d)
        val.append((p,q))
    res = [ abs(rump1(e[0],e[1]) - rump2(e[0],e[1])) for e in \
    ↪ val ]
    plt.plot(range(1,N+1),res)
    plt.show()
```

2.3) `rump2(77617,33096)` renvoie -0.8273960599468213, ce qui n'a rien à voir avec le résultat de `rump1`, même si formellement c'est la même expression! Le résultat le plus fiable est évidemment celui de `rump2`, car le calcul de $F(77617, 33096)$ est exact, il donne -8 (c'est un calcul sur des entiers), c'est lorsqu'on ajoute $2 * x/y \approx 4.690415760212715$, qu'il y a une conversion en flottant (suivi d'une division par 4 qui est une puissance de 2), mais cette opération n'engendre pas de grosses erreurs d'approximation.



3) La fonction `test(N,d)` :

Solution 15

1) 1.1) La fonction `compare` :

```python
from math import sqrt
def compare(n: int)-> None:
    u0 = 1 ; q = (1-sqrt(5))/2
    u1 = q
    print(u0, ' v = ', 1)
    print(u1, ' v = ', q)
    v = q
    for k in range(n-2):
        u0, u1 = u1, u0+u1
        v *= q
        print(k+2,' u = ',u1,'  v = ', v)
```

1.2) En exécutant par exemple `compare(500)`, la dernière ligne affichée est :

il semble que la suite u pourrait bien tendre vers $-\infty$, alors que la suite v semble converger vers 0.

2) 2.1) Il est clair que pour tout n, $v_n = q^n$. La suite u vérifie une récurrence linéaire à deux pas, la méthode vue dans le cours de mathématique nous donne qu'il existe deux réels a et b tel que pour tout n, $u_n = a\left(\frac{1+\sqrt{5}}{2}\right)^n + b\left(\frac{1-\sqrt{5}}{2}\right)^n$, les valeurs $u_0 = 1$ et $u_1 = q$ entraînent $a = 0$ et $b = 1$, c'est à dire $u_n = q^n$. On remarque donc que les suites u et v sont égales, ce qui ne semblait pas du tout le cas avec la simulation numérique !

2.2) Lorsqu'on regarde de plus prés les valeurs successives calculées, on s'aperçoit que u_{40} est négatif comme u_{39}, alors que v_{40} est positif (ce qui est normal puisque $q < 0$). À partir de là, les termes calculés de la suite u, sont tous négatifs et se cumulent. Le problème est que dans la machine, la valeur codée de u_1 n'est pas exactement q, mais une valeur approchée $\tilde{q}$, et avec $u_1 = \tilde{q}$, le coefficient a devant $\left(\frac{1+\sqrt{5}}{2}\right)^n = \frac{1}{|q|^n}$ dans l'expression de u_n (coefficient qui vaut théoriquement 0), n'est pas nul, et comme $\frac{1}{|q|^n} \to +\infty$, cela peut expliquer le comportement de u d'un point de vue numérique.