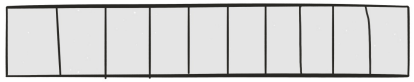
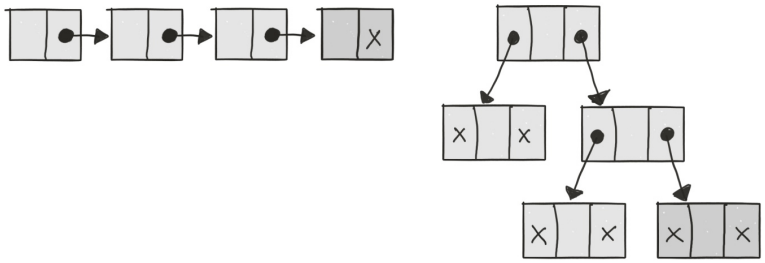
Algorithmique et programmation

14 octobre 2016

David Pichardie

Les données

- Elles peuvent représenter des nombres dans des formats divers…
  - entiers de capacités limités,
  - un sous-ensemble des nombres réels : les *flottants*
- Mais aussi des données structurées
  - tableaux 
  - structures chaînées 

Représentations des nombres entiers

```
$ ./fact 30
(0, 1)
(1, 1)
(2, 2)
(3, 6)
(4, 24)
(5, 120)
(6, 720)
(7, 5040)
(8, 40320)
(9, 362880)
(10, 3628800)
(11, 39916800)
(12, 479001600)
(13, 6227020800)
(14, 87178291200)
(15, 1307674368000)
(16, 20922789888000)
(17, 355687428096000)
(18, 6402373705728000)
(19, 121645100408832000)
(20, 2432902008176640000)
(21, -4249290049419214848)
(22, -1250660718674968576)
(23, 8128291617894825984)
(24, -7835185981329244160)
(25, 7034535277573963776)
(26, -1569523520172457984)
(27, -5483646897237262336)
(28, -5968160532966932480)
(29, -7055958792655077376)
```

Arithmétique bornée !

```
$ ocaml fact.ml 30
(0, 1)
(1, 1)
(2, 2)
(3, 6)
(4, 24)
(5, 120)
(6, 720)
(7, 5040)
(8, 40320)
(9, 362880)
(10, 3628800)
(11, 39916800)
(12, 479001600)
(13, 6227020800)
(14, 87178291200)
(15, 1307674368000)
(16, 20922789888000)
(17, 355687428096000)
(18, 6402373705728000)
(19, 121645100408832000)
(20, 2432902008176640000)
(21, -4249290049419214848)
(22, -1250660718674968576)
(23, -1095080418959949824)
(24, 1388186055525531648)
(25, -2188836759280812032)
(26, -1569523520172457984)
(27, 3739725139617513472)
(28, 3255211503887843328)
(29, 2167413244199698432)
```

Arithmétique bornée !

```
$ python fact.py 30
(0, 1)
(1, 1)
(2, 2)
(3, 6)
(4, 24)
(5, 120)
(6, 720)
(7, 5040)
(8, 40320)
(9, 362880)
(10, 3628800)
(11, 39916800)
(12, 479001600)
(13, 6227020800)
(14, 87178291200)
(15, 1307674368000)
(16, 20922789888000)
(17, 355687428096000)
(18, 6402373705728000)
(19, 121645100408832000)
(20, 2432902008176640000)
(21, 51090942171709440000L)
(22, 1124000727777607680000L)
(23, 25852016738884976640000L)
(24, 620448401733239439360000L)
(25, 15511210043330985984000000L)
(26, 403291461126605635584000000L)
(27, 10888869450418352160768000000L)
(28, 304888344611713860501504000000L)
(29, 8841761993739701954543616000000L)
```

Arithmétique non-bornée !

Représentations des nombres entiers

- Représentation binaire sur *n* bits

$$x = b_{n-1} \cdots b_0$$

- Interprétation non-signée

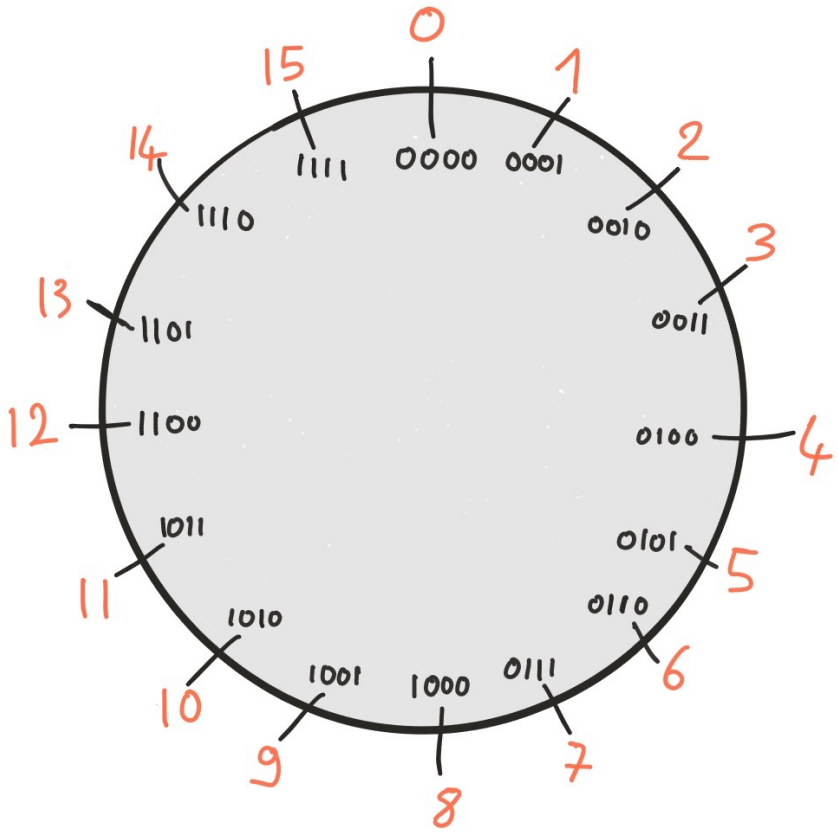
$$U(x) = \sum_{k=0}^{n-1} b_k 2^k$$

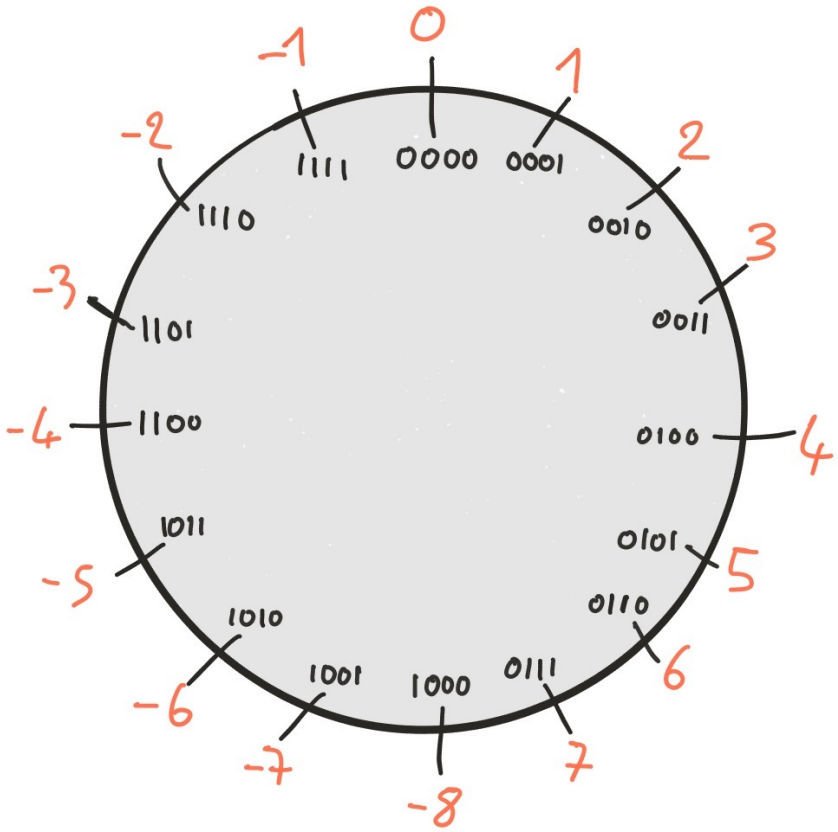
- Interprétation signée

$$S(x) = -b_{n-1} 2^{n-1} + \sum_{k=0}^{n-2} b_k 2^k$$

Représentation des nombres entiers

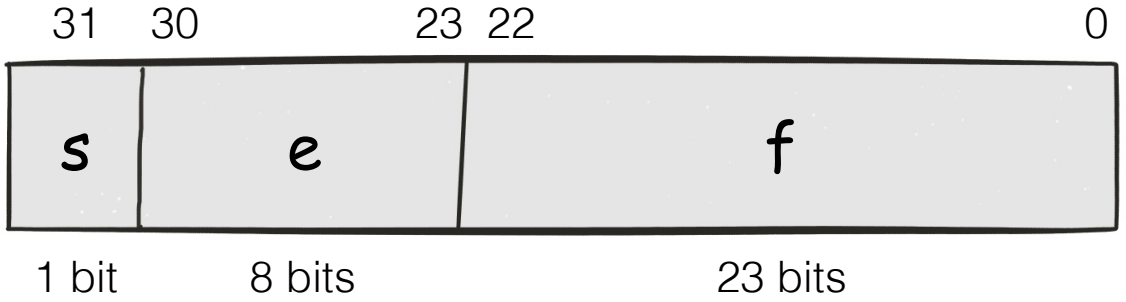
n=4





Les nombres flottants

$x =$



$$F(x) = (-1)^s \times M \times 2^{e-127}$$

$$M = \begin{cases} 1, f_{23} f_{22} \cdots f_0 & e \neq 0, e \neq 255 \\ 0, f_{23} f_{22} \cdots f_0 & e = 0 \end{cases}$$

$$F(x) = -\infty, +\infty \text{ ou Nan si } e = 255$$

Les nombres flottants
vus d'avion

- Quand les calculs réelles ne tombent pas sur un flottant, il faut arrondir vers un flottant proche
    - cela peut occasionner des pertes de précision importantes

Flottants : quizz

Qu'est-ce que ce programme va afficher ?

```
x = 0.0
for i in range(10):
    x = x + 0.1
if x == 1.0:
    print x, ' est egal a 1.0'
else:
    print x, ' n\'est pas egal a 1.0'
```

```
1.0  n'est pas egal a 1.0
```

Quelle représentation machine pour 0.1 ?

- Rappel

$$F(x) = (-1)^s \times M \times 2^{e-127}$$

$$M = \begin{cases} 1, f_{23} f_{22} \cdots f_0 & e \neq 0, e \neq 255 \\ 0, f_{23} f_{22} \cdots f_0 & e = 0 \end{cases}$$

- il n'est pas possible de représenter ce nombre de manière exacte !

Explications

- Lors des additions successives, c'est le représentant de 0.1 qui est ajouté
- Le résultat final n'est pas égal à 1.0
- … est n'est pas non plus le représentant flottant de 1.0 !
- L'affichage flottant de Python se permet de faire un arrondi et affiche 1.0 au lieu de 0.99999999999…

Flottants : moralité

- Vigilance sur le cumul d'erreurs d'arrondis
- Ne jamais faire de tests d'égalités sur les flottants
- Ne pas prendre l'affichage des flottants pour « argent comptant »

Entiers/flottants : les conversions de types

- Conversions explicites
  - flottants vers entiers

```
float(2353)
```

```
2353.0
```

  - entiers vers flottants

```
int(3.14)
```

```
3
```

perte de précision !

- Conversions implicites

```
1.2 + 3
```

```
3.2
```

python 2.7

```
1 / 4
```

```
0
```

```
1 / 4.
```

```
0.25
```

```
1 // 4
```

```
0
```

```
1 // 4.
```

```
0.0
```

Conversions de types (numériques) : moralité

- Certains opérateurs utilisent le même symbole mais changent de comportement en fonction du type des arguments
- Le type d'une valeur peut être changé explicitement (si son contenu le permet)
- Dans certains langages, les types peuvent être convertis implicitement : élégant, mais parfois déroutant !

Récapitulatif

- Les entiers machines ont une capacité limité. Python fait le choix de proposer une précision arbitraire, quitte à devoir occuper plus de mémoire.

- Les nombres réels sont généralement représentés par un sous-ensemble des rationnels : les flottants. Les pertes de précision par rapport aux opérations mathématiques réelles peuvent être très grandes.

Flot de contrôle

Flot de contrôle

Les boucles

Boucles *for*

compter jusqu'à trois…

```
for i in [1 ,2 , 3]:
    print i
```

en sortie de boucle, i vaut 3

```
for i in range(3):
    print i+1
```

range(n) construit la liste [0, .., n -1]

```
for i in range(1,4):
    print i
```

range(a,n) construit la liste [a, .., n -1]

```
for i in range(2,7,2):
    print i/2
```

range(a,n,k) construit la liste [a, .., n -1], mais ne garde que les multiples de k

Exercice

Écrire une fonction `index(t,e)` qui renvoie la position de l'élément `e` dans le tableau `t`.
(sans utiliser d'instruction `return` dans la boucle principale)

Exercice

Écrire une fonction `index(t,e)` qui renvoie la (première) position de l'élément `e` dans le tableau `t`, ou `-1` si l'élément `e` n'est pas présent dans `t`.
(sans utiliser d'instruction `return` dans la boucle principale)

Solution #0

```
def find(e,t):
    for i in range(len(t)):
        if t[i]==e:
            return i
    return -1
```

l'instruction `return` interrompt la fonction, et par la même occasion, la boucle

mais on veut parfois seulement interrompre la boucle et pas toute la fonction…

Solution #1

```
def find(e,t):
    for i in range(len(t)):
        if t[i]==e:
            break
    if t[i]==e:
        return i
    else:
        return -1
```

l'instruction `break` interrompt la boucle

il faut comprendre par quelle porte nous sommes sortis de la boucle

Solution #2

```
def find(e,t):
    i = 0
    while (i<len(t) and t[i]!=e):
        i += 1
    if i < len(t):
        return i
    else:
        return -1
```

La boucle `while` généralise la boucle `for`

Attention aux débordements de tableaux !

l'opérateur `and` est paresseux : le deuxième argument n'est pas évalué si le premier renvoie `False`

Exercice

Écrire une fonction `enter_str_gt4()` qui demande à l'utilisateur une entrée interactive (avec la fonction `raw_input(msg)`) et renvoie le résultat si la chaîne correspondante possède au moins 4 caractères. La fonction ne termine pas tant que l'utilisateur n'a pas saisie une chaîne adaptée.

Solution

```
def enter_str_gt4():
    while True:
        v = raw_input('Entrer un texte d\'au moins 4 caractères : ')
        if len(v) >= 4:
            break
        print 'texte trop court, essaye encore !'
    return v

res = enter_str_gt4()
print 'résultat =', res
```

boucle infinie

unique sortie de la boucle

Moralité

- Privilégier si possible les boucles `for`, mais éviter cependant de trop vous appuyer sur la valeur finale de la variable d'itération (peut varier en fonction des langages)
- En l'absence de construction `repeat/until`, utiliser une boucle `while` infinie + `break`.

Flot de contrôle

Les fonctions

Bonnes pratiques

- Un programme trop gros devient difficile à comprendre et à débuguer
- Il faut le découper en petites **fonctions** élémentaires
  - chaque fonction aura une *spécification* claire
  - chaque fonction sera testée individuellement (*test unitaire)*

Arguments

passage par position

```
def print_name(first,last,reverse):
    if reverse:
        print last, first
    else:
        print first, last

print_name('David','Pichardie',False)
print_name('David','Pichardie',True)
```

Arguments

passage par nom

```
def print_name(first,last,reverse):
    if reverse:
        print last, first
    else:
        print first, last

print_name(reverse=False,
           first='David',
           last='Pichardie')
```

Arguments

valeurs par défaut

```
def print_name(first,last,reverse=False):
    if reverse:
        print last, first
    else:
        print first, last

print_name('David','Pichardie')
```

Ordre d'évaluation

```
def print_sum(i,j):
    print (i+j)
    return i+j

def print2():
    print 2
    return 2

def print1():
    print 1
    return 1

v = print_sum(print1(),print2())

print 'v =', v
```

1re évaluation

2e évaluation

3e évaluation

Variables globales

sans l'annotation `global`, la variable `count` serait locale !

```
def fib(n):
    global count
    count = count +1
    if n ==0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

count = 0
print "fib(10) =", fib(10)
print "#appel fib =", count
```

Exercice

- Ecrire un programme python **`tobin.py`** qui transforme son entrée entière en une représentation binaire sur la ligne de commande
  exemple : `%python tobin.py 11`
  `1 0 1 1`
- Consignes
  - ne pas utiliser de structure auxiliaire type tableau
  - proposer une version avec récursivité, puis sans
- Remarque : pour afficher un caractère sans revenir à la ligne, utiliser la commande `print` *`chaine`*`,`

Solution

version avec récursivité

```
import sys

def tobin(n):
    if n<=1: print(n),
    else:
        tobin(n//2)
        print(n % 2),


tobin(int(sys.argv[1]))
print "" #pour revenir a la ligne
```

Solution

version sans récursivité

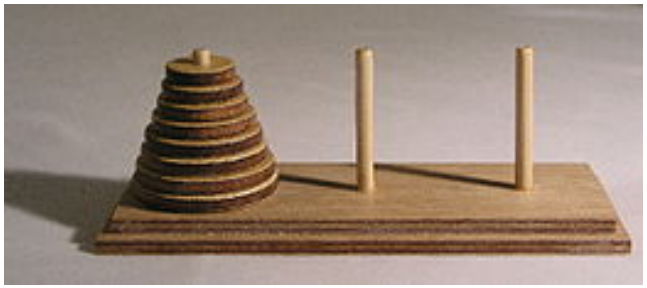
```
import sys

def tobin(n):
    # on calcule la plus grande puissance de 2 inférieur ou
    # egale a n pour trouver le bit de poid fort
    v = 1
    while v <= n//2 :
        v *= 2 # equivalent a v = v*2

    # puis on parcours les bits de n en partant du plus fort
    while v > 0:
        if n >= v: # v=2**k et le k[eme] bit de n vaut 1
            print '1',
            n = n-v
        else: # v=2**k et le k[eme] bit de n vaut 0
            print '0',
        v = v//2


tobin(int(sys.argv[1]))
print "" #pour revenir a la ligne
```

Exercice : tours de Hanoï



- On dispose de 3 piquets et de N disques sur ces piquets
- On ne peut déplacer plus d'un disque à la fois
- On ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide
- Au départ, tous les disques sont sur le premier piquet
- Comment déplacer les disques pour positionner tous les disques sur le troisième piquet ?