Représentation des nombres

L'ordinateur manipule, au plus bas niveau l'équivalent de deux types d'informations, soit des 1 soit des 0 : les **bits**.
Comment alors représenter un nombre quelconque, manipulé de manière classique sous sa forme décimale, en machine ?
Le système décimal peut être remplacé par le système binaire (ou octal ou hexadécimal).

Rappel : La représentation d'un nombre dans un système en base b, pour toute valeur entière positive de b, est positionnelle : le mot $a_n...a_1$ a pour valeur $a_1 \times b^0 + ... + a_n \times b^n = \sum_{i=0}^{n} a_i b^i$.
On appelle bit **de poids fort** celui qui est *à gauche* du mot, et doit être multiplié par la plus grande puissance de b. Le bit **de poids faible** est celui des unités, à multiplier par b^0.

Représentation des entiers positifs sur des mots de taille fixe

---

Conversion décimal - binaire

Comment convertir en binaire un nombre sous forme décimale ?
Deux algorithmes permettent de le faire, l'un calculant d'abord les bits de poids forts, l'autre d'abord les bits de poids faible.

Algorithme 1

- Tant que le nombre est non nul
  - si le nombre est pair, ajouter 0 en fin de liste, sinon ajouter 1 puis diviser (en division entière) le nombre par 2
- Renverser puis retourner la liste obtenue

Algorithme 2

- Tant que le nombre est non nul
  - Enlever la plus grande puissance de 2 possible. Ajouter un 1 à l'indice correspondant dans la liste à renvoyer.
- Renverser puis retourner la liste obtenue

Pour l'implémentation de cet algorithme, il faut donc préparer en amont un tableau contenant les puissances de 2 (et l'utiliser avec intelligence ! : comment trouver la plus grande puissance de 2 possible sans gaspiller de ressources ?).

Système hexadécimal

Le système hexadécimal est également un système positionnel, mais en base 16. Pour compenser le fait que notre système décimal ne comporte que 10 chiffres, on y adjoint les 6 premières lettres de l'alphabet. Ainsi A est l'encodage hexadécimal de 10 ; $2F3$ est celui de $2 \times 16^2 + 11 \times 16 + 3$.

Pour passer du système binaire au système hexadécimal, on remarque qu'on peut grouper les chiffres d'un nombre en binaire par paquet de 4, en partant du bit de poids faible.

Travaux pratiques

Système décimal, binaire, hexadécimal

1. Énumérer les nombres de 0 à 19 en binaire.
2. Combien peut-on représenter de nombres avec un tel système restreint à 5 bits ? 8 bits ? n bits ?
3. De combien de bits a-t-on besoin pour représenter en binaire les nombres de 0 à 7 ? de 0 à 15 ? de 0 à 36 ? de 0 à n ?
4. Répondre au mêmes questions en considérant non plus le système binaire mais le système hexadécimal.
5. TP - implémenter l'addition et la soustraction avec propagation de retenue. *On pourra considérer les nombres binaires sous forme de listes de* 0 *et de* 1*, avec bit de poids faible à la fin.*
6. TP - Créer deux fonctions de conversion du décimal vers le binaire, et les fonctions réciproques. On considérera un nombre binaire sous la forme d'une liste de 0 et de 1, où le terme de plus haut degré est le premier de la liste.
7. TP - Créer une fonction de conversion binaire - hexadécimal, et la fonction réciproque.
8. *Avec des portes "ou","et", "non", construire les circuits permettant l'addition et la soustraction de deux bits, puis de nombres stockés sur n bits.*

Retenir : On peut représenter nombres en binaire sur n bits.
Lorsque $n = 5$, cela correspond aux nombres de à .
Lorsque $n = 7$, cela correspond aux nombres de à .
Lorsque $n = 8$, cela correspond aux nombres de à .
Lorsque $n = 12$, cela correspond aux nombres de à .

Inversement, pour représenter les nombres de 0 à n, on a besoin de bits.
Ainsi, pour représenter les nombres de 0 à 135, on a besoin de bits.

Correction

1. Énumérer les nombres de 0 à 19 en binaire : $0, 1, 10, 11, 100, 101, 110, 111....10010, 10011$
2. Combien peut-on représenter de nombres avec un tel système restreint à 5 bits ? 8 bits ? n bits ? 2^n
3. De combien de bits a-t-on besoin pour représenter en binaire les nombres de 0 à 7 ? de 0 à 15 ? de 0 à 36 ? de 0 à n ? 3, 4, 6, partie entière supérieur du log en base 2 de n.
4. Répondre au mêmes questions en considérant non plus le système binaire mais le système hexadécimal : En hexadécimal : 19 s'écrit 13. On notera 19_{10} et 13_{16} pour indiquer la base du système utilisé. Pour représenter un nombre, on a besoin de quatre fois moins de symboles qu'en binaire, soit la partie entière supérieure du log en base 16.
5. TP - implémenter l'addition et la soustraction avec propagation de retenue.

```
# rajoute des 0 à la première liste jusqu'à ce que
#  sa taille soit égale à celle de la seconde
def  complete(l1,l2) :
    l = len(l2)-len(l1)
    rajout = [0 for i in range(l)]
    return rajout+l1

def add (l1,l2):
    if len(l1)<len(l2):
        l1 = complete(l1,l2)
    else:
        l2 = complete(l2,l1)
    res = [i for i in l1]
    retenue = 0
    for i in range (1,len (l1)+1):
        (a,b) = l1[-i],l2[-i]
        res[-i]= (a+b+retenue) % 2
        retenue = (a+b+retenue) // 2
    return res


def sub(l1,l2):# on suppose que l1 est de longueur >= l2
    res = [i for i in l1]
    retenue = 0
    for i in range (1,len (l1)+1):
            (a,b) = l1[-i],l2[-i]
            res[-i]= (a-b-retenue) % 2
            retenue = (b+retenue+1-a) // 2
    return res
```

6. TP - Créer deux fonctions de conversion du décimal vers le binaire, et les fonctions réciproques. On considérera un nombre binaire sous la forme d'une liste de 0 et de 1, où le terme de plus haut degré est le premier de la liste.

```
def conversion1(n):
    l = []
```

```
    while n>0:
         l.append(n-2*(n//2))
         n = n//2
    l.reverse()
    return l

#renvoie la partie entière inférieur du log en base 2 de n
def log2Inf(n):
    a = 0
    b = 1
    while b<=n :
        a+=1
        b*=2
    return a-1

#cree un tableau des puissances de 2
def puiss2(k):
    l = []
    a = 1
    for i in range(k):
        l.append(a)
        a*=2
    return l

def conversion2(n):
    indMax = log2Inf(n)+1
    #on prépare le tableau des puissances de 2 qui peuvent servir
    tabPuiss = puiss2(indMax)
    res = [0 for i in range(indMax)]
    ind = -1
    while (n>0):
        while (n-tabPuiss[ind]<0):
            ind-=1# par construction, pas d'erreur "out of bounds"
        res[ind]=1
        n-=tabPuiss[ind]
    res.reverse()
    return res
```

7. TP - Créer une fonction de conversion binaire - hexadécimal, et la fonction réciproque.

```
def conversionB(l):
    b = 0 #b va exprimer l en décimal
    l.reverse() # on travaille des poids faibles aux poids forts
    ind = 0
    puiss = 1
    while ind<len(l):
        b+=l[ind]*puiss
        puiss*=2
        ind+=1
    # b est la version décimale du nombre
    conv = [0,1,2,3,4,5,6,7,8,9,'A','B','C','D','E','F']
    return (conv[b])
```

```
def binToHexa(l):
    res = []
    while l!= []:
        bloc = l[-4:]
        l = l[:-4]
        res.append(conversionB(bloc))
    res.reverse()
    return res
```

Travaux pratiques

Représentation des entiers

Représentation des entiers signés sur des mots de taille fixe

Si on veut pouvoir représenter des nombres relatifs, le système précédent n'est plus suffisant. Comment alors représenter -1 ? -3 ?
Supposons qu'on utilise 4 bits. On peut représenter à l'aide de ces 4 bits $2^4 = 16$ nombres.

Représentation signe-magnitude

Le premier bit est un **bit de signe** (1 pour un nombre négatif, 0 pour un nombre positif), et le reste est la représentation en binaire de la partie numérique du nombre.

- La représentation sur 4 bits de 4 est : ; celle de -4 est :
- La représentation de -12 sur 5 bits est :
- Le nombre 0 est représenté par :
- Ajouter 1 à la représentation (sur 4 bits) de 3 donne : c'est-à-dire :
- Ajouter 1 à la représentation (sur 4 bits) de -3 donne : c'est-à-dire :
- Sommer 5 et -5 avec cette représentation.

Qu'en conclure ?

Complément à deux

Une autre représentation est celle dite du **complément à deux**, qui consiste à représenter l'opposé de $a \geq 0$ par son complément à 2^k (le résultat de la différence entre 2^k et a).
Par exemple, sur 5 bits, on cherche à représenter -7. La représentation de 7 en binaire est 00111. La représentation de -7 dans ce système est donnée par le résultat de la différence en binaire $100000 - 00111 = 11001$.
Le calcul du complément à deux se fait également par l'algorithme suivant :

- Changer chaque 0 en 1 et réciproquement ;
- ajouter 1 au résultat obtenu

Exercices

1. Faire le tableau des nombres représentés en complément à deux sur 5 bits.
2. Montrer que l'algorithme proposé permet effectivement de calculer le complément à deux
3. Montrer que l'arithmétique sur les nombres en compléments à deux est indépendante du signe des nombres considérés
4. TP - Écrire une fonction qui calcule le complément à deux
5. TP - En déduire une fonction de soustraction, utilisant la fonction d'addition déjà écrite.

Correction du TP

2. Montrer que l'algorithme proposé permet effectivement de calculer le complément à deux : Soit a un nombre sous forme binaire sur k bits, et b le nombre obtenu de celui-là en inversant les 0 et les 1. Alors $a + b = 11....11$ (avec le même nombre de bits k). Quand on ajoute 1, on trouve, sur $k + 1$ bits : 100...00, soit l'expression en binaire de 2^k. Le nombre obtenu par l'opération $b + 1$ est donc bien le complément à deux de a sur k bits.
3. Montrer que l'arithmétique sur les nombres en compléments à deux est indépendante du signe des nombres considérés :
   - Si a et b représentent des nombres positifs, l'addition se fait classiquement ;
   - Si a est positif et b négatif, $a + b = a + 2^k - (-b) = 2^k + a - b$. Les représentations de a et b étant sur k bits se font modulo 2^k ;
   - Soient a et b. Alors $(a - b) + b = a$

```
def comp2(Lm):
    Ln = [i for i in Lm]
    for i in range (len(Ln)):
        Ln[i]=1-Ln[i]
    curseur = len(Ln)-1
    while(curseur >=0 and Ln[curseur]==1):
        Ln[curseur]=0
        curseur-=1
    if (curseur>=0):
        Ln[curseur]=1
    return Ln
```

   - Dans le cas de la soustraction, on procède de la même manière. On en déduit un moyen de soustraire des nombres à 'laide de l'addition et du complément à deux.
4. TP - Écrire une fonction qui calcule le complément à deux

```
#Calcule le complément à deux d'un nombre
# entree : nombre représenté sous forme de liste de bits l
#sortie : complément à deux sous forme de liste, même nombre de bits
def complADeux(l):
    res = [(1-i) for i in l]
    res = add(res,[1])
    return res

print(complADeux([1,0,0,1,1,0,0,1,0,0,0]))
```

5. On en déduit une fonction très simple qui effectue la différence de deux nombres binaires sous forme de listes de bits.

```
def sub(a,b):
    return add(a, complADeux(b))
```

Entiers multiprécision de Python

Un nombre peut être représenté en machine sur un nombre fini n de bits, mais dans ce cas, le nombre de nombres représentés est au plus :
Ces entiers sont appelés **petits entiers** ou **entiers machines** : les opérations les concernant sont programmés par des instructions du langage de la machine et directement effectuée par des circuits électroniques, dont le temps de calcul est indépendant de la taille.
Le problème peut être contourné en considérant qu'un nombre entier $\sum_{i=0}^{n} a_i b^i$est représenté par une liste d'entiers (ou de bits) a_i, et la limite devient celle des listes : on peut alors représenter un nombre avec plusieurs centaines de milliers de chiffres (ou de bits).
Dans ce cas, on choisira de placer l'unité ou bit de poids faible en tête de liste, simplifiant les opérations arithmétiques et l'accès au chiffre a_i.
Python dispose d'une bibliothèque de "grands entiers" ou **entiers multiprécision** appelée **GMP** :
Das ce cadre, le temps d'une opération arithmétique n'est plus constant et dépend de la taille des nombres en entrée : il devient difficile d'évaluer la complexité des opérations arithmétiques sur ces entiers.
Les exercices qui suivent sont à adapter à votre vitesse. Si besoin, faites-les en base 10 *et non dans le cas général de la base b*

1. Représenter 12345678 et 9999999999999876543210 **en base** 10 sous forme de somme, avec des puissances de 10 et dans le cadre courant (sous forme de liste) ;
2. Écrire une fonction qui convertit un nombre sous forme décimale en le même nombre sous forme de liste **en base b**(*exercice rapide, proche de ce qu'on a fait avec le système binaire*), ainsi que la fonction réciproque. Ces fonctions pourront servir pour tester plus aisément les fonctions d'additions et de produit qui suivent.
3. En ce qui concerne la fonction qui convertit une liste en enteir sous forme décimale : est-elle optimale ? Pouvez-vous réduire le nombre de calculs effectués ? (*Attention aux exponentiations !)*
4. Écrire une fonction `multiplication` qui calcule le produit de deux nombres a et b sous la forme décrite ci-dessus. On pourra disposer des fonctions suivantes :
   - `addition(L,M)` qui additionne deux entiers sous formes de liste
   - `additions(L)` qui prend une liste d'entiers (sous formes de liste) et renvoie la somme sous forme de liste
   - `etapeMultiplication(L,b,decalage)` qui renvoie sous forme de liste le produit de l'entier (liste) `L` par le chiffre `b`, après décalage de `b` crans.
     Par exemple `etapeMultiplication([1,2,4],2,3)` renvoie `[2,4,8,0,0,0]`

Multiplication rapide de Karatsuba

Cette section vise à écrire une fonction de multiplication plus rapide (moins naïve) que celle à laquelle nous sommes arrivés précédemment.

Il s'agit d'utiliser l'égalité $(a \times 10^k + b)(c \times 10 + d) = ac \times 10^{2k} + (ad + bc) \times 10^k + bd$. Pour calculer le membre de droite, on a besoin de faire le calcul de quatre produits ac, bd, ad, bc. Mais on peut regrouper les calculs sous la forme : $(a \times 10^k + b)(c \times 10^k + d) = ac \times 10^{2k} + (ac + bd - (a-b)(c-d)) \times 10^k + bd$, et pour calculer les produits dont on a besoin, on peut procéder récursivement. (*Remarque : on peut également s'intéresser à* $(a+b)(c+d) - ac - bd$)

L'approche consiste à diviser **récursivement** le problème en sous-problèmes plus petits avant de combiner leurs solution pour construire la solution au problème d'origine : c'est un algorithme

de type **diviser pour régner**.

La multiplication par la base de numération correspond à un simple décalage ; les additions sont de complexité temporelle plus petite que les les multiplications, d'où le gain de temps en passant de 4 produits à 3 pour de très grands nombres.

Avant de coder l'algorithme, procédez à l'application de l'algorithme de Karatsuba sur de "petits" exemples (nombres de quatre chiffres). Combien de multiplication utilisez-vous ? Combien en aurait-il fallu avec la multiplication naïve ?

```
##conversions entiers/liste aller et retour
def conversion(n):
    A = []
    while(n>0):
        A.append(n%10)
        n = n//10
    A.reverse()
    return A

#Conversion d'une liste vers un entier. Pourquoi faire dans ce sens? Combien d'opérations so
def noisrevnoc(L):
    n = 0
    for i in range(len(L)):
        n*=10
        n+=L[i]
    return n


##  Additions sur des entiers sous forme de liste

#met les deux listes à la même taille
def complete(L,M) :
    decal = len(L)-len(M)
    if decal<0:
        L = [0 for i in range(-decal)]+L
    return L

# addition de deux entiers sous forme de liste
def addition(L,M):
    (L,M) = (complete(L,M), complete(M,L))
    A = [i for i in L]
    retenue = 0
    A.reverse()
    M.reverse()
    for j in range(len(M)):
        res = A[j]+M[j]+retenue
        A[j]=res%10
        retenue = res//10
    if retenue!=0:
        A.append(retenue)
    A.reverse()
    return A

#additions de plusieurs entiers sous forme de liste
def additions(L):
    R = []
    for element in L:
        R = addition(R,element)
    return R


## def etapeMultiplication(L,b,decalage):
    #L est sous forme de liste, b est un chiffre
    #on copie L dans une liste A
```

```
    A = [i for i in L]
    A.reverse()
    retenue = 0
    for i in range(len(A)):
        res = A[i]*b+retenue
        A[i]=res%10
        retenue = res//10
    if retenue!=0:
        A.append(retenue)
    A.reverse()
    for i in range(decalage):
        A.append(0)
    return A

def multiplication(L,M):
    res = []
    for i in range(len(M)):
        pl = etapeMultiplication(L,M[-i-1],i)
        res.append(etapeMultiplication(L,M[-i-1],i))
    return additions(res)
```