Problème 1. Systèmes de numération

On s'intéresse dans ce problème à la représentation des nombres entiers sur un ordinateur. Dans la première partie, on s'intéresse aux entiers de *petite taille* représentés en binaire et l'utilisation des opérateurs logiques (« et », « ou », « non ») pour faire des opérations arithmétiques. Dans la seconde partie, on s'intéresse aux grands entiers, qui sont représentés comme des entiers en une base β ($\beta = 2^{64}$) sur les ordinateurs modernes.

1.1 Représentation binaire des entiers

Question 1.1. Opérations sur les bits On représente les bits par les chaînes de caractères `'0'` et `'1'`. On considère que le bit `'0'` représente Faux et `'1'` représente Vrai, et on définit les opérateurs logiques $\wedge$ (« et »), $\vee$ (« ou ») et $\neg$ (« non ») en fonction de cette convention : `'1'`$\wedge$`'0'` vaut `'0'`, `'1'`$\vee$`'0'` vaut `'1'`, $\neg$`'1'` vaut `'0'`, etc. *Dans cette question, on peut supposer que les entrées sont bien des caractères `'0'` ou `'1'`, sans le vérifier.*

(a) Écrire la fonction `non(x)` qui prend en entrée un caractère x (`'0'` ou `'1'`) et renvoie le caractère qui représente $\neg x$. **Il est interdit d'utiliser les opérateurs logiques `and`, `or` et `not`.**

(b) Écrire la fonction `et(x,y)` qui calcule $x \wedge y$. **Il est interdit d'utiliser les opérateurs logiques `and`, `or` et `not`.**

Dans la suite, on supposera qu'on a également écrit une fonction `ou(x,y)` qui calcule `x\vee y`.
On définit l'opérateur « ou exclusif », noté $\otimes$, qui renvoie Vrai si ses deux entrées sont distinctes.

(c) Écrire la fonction `xor(x,y)` qui calcule $x \otimes y$, en faisant appel à `et`, `ou` et `non`, et **sans utiliser le branchement conditionnel** (`if: ... else: ...`). Combien d'opérateurs élémentaires ($\wedge$, $\vee$, $\neg$) la fonction `xor` utilise-t-elle ?

Question 1.2. Addition de bits On veut faire l'addition avec retenue de deux bits : étant donnés deux bits x, y et une *retenue entrante* r_e (qui est un bit également), on souhaite calculer $x + y + r_e$ (en voyant x, y et r_e comme des entiers de $\{0, 1\}$).

(a) Montrer que l'entier $x + y + r_e$ est représentable sur deux bits.

Soit add2(x, y, r_e) la fonction qui prend en entrée x, y et r_e, et renvoie le couple (z, r_s) tel que $x + y + r_e$ s'écrive $r_s z$ en binaire (r_s est la *retenue sortante*).

(b) Compléter la fonction `add2` ci-dessous (à l'aide des fonctions de la question précédente).

```
def add2(x, y, re):
    z = ...
    rs = ...
    return z, rs
```

(c) Compter le nombre total d'opérateurs élémentaires ($\wedge$, $\vee$, $\neg$) utilisés dans `add2`. *Si votre fonction utilise `xor`, il faut compter les opérateurs contenus dans `xor`.*

Question 1.3. Entiers de taille fixe On représente un entier par un mot de k bits exactement. Par exemple pour $k = 4$, on représente l'entier 5 par `'0101'`.

(a) Donner la représentation de 3 et de 15 sur 4 bits.

(b) Quels sont les entiers représentables sur k bits exactement ?

(c) Compléter la fonction `sup` ci-dessous qui renvoie `True` si l'entier représenté par `x` est strictement supérieur à l'entier représenté par `y`, et `False` sinon. *On suppose que `len(x) == len(y)`.*

```
def sup(x,y):
    k = len(x)
    for i in range(k):
        if x[i] != y[i]:
```

```
            if x[i] == ...:
                return True
            return False
    return ...
```

Question 1.4. Addition d'entiers de taille fixe On définit la fonction d'addition avec retenue de deux entiers de k bits x et y par $\text{add}(x,y,r) = (x+y+r) \bmod 2^k$ où $r \in \{0,1\}$ est le retenue.

(a) Combien de bits faut-il pour représenter le résultat de $\text{add}(x,y,r)$?

(b) Compléter la fonction suivante qui implante la fonction add, où la retenue `r` est représentée par un booléen.

```
def add(x, y, r):
    k = len(x)
    if len(y) != k: raise ValueError("x et y doivent avoir la même taille")

    re = ... if r else ...
    z = ''
    for i in reversed(range(k)): # i va de k-1 à 0
        ..., ... = add2(..., ..., ...)
        z = zi + z
        re = ...

    return z
```

(c) En ne considérant que les opérateurs logiques ($\wedge$, $\vee$, $\neg$) comme opérations élémentaires, calculer la complexité exacte de la fonction `add`. *Remarque : on demande le nombre précis d'opérations, pas l'asymptotique.*

Question 1.5. Représentation d'entiers relatifs en *complément à la base* On considère un mot w de k bits et on note x l'entier représenté par w dans la représentation classique des nombres positifs. Alors si le premier bit de w est `0`, il représente l'entier x et si le premier bit est `1`, w représente $x - 2^k$.
Par exemple, le mot `'1010'` représente l'entier 10 en représentation classique, donc l'entier $10 - 2^4 = -6$ en complément à la base. Le mot `'0001'` représente l'entier 1 dans les deux représentations.

(a) Quel est l'entier représenté par le mot `'1000'` en complément à la base sur 4 bits ?

(b) Donner la représentation de -3 et 2 en complément à la base sur 4 bits.

(c) Combien d'entiers sont représentables sur k bits en complément à la base ? Donner le plus petit et le plus grand de ces entiers.

Question 1.6. Soustraction en complément à la base L'intérêt de cette représentation est que l'algorithme d'addition classique fonctionne encore (*admis*).

(a) Montrer que si `u` est un mot de k bits et `v` le mot obtenu en inversant chaque bit de `u` (chaque `0` devient `1` et chaque `1` devient `0`), alors `add(u,v,False) = '1...1'`.

(b) En déduire que le mot `w` qui représente l'opposé de `u` est obtenu en inversant tous les bits de `u` et en ajoutant `0...01` au résultat.

(c) Écrire une fonction `oppose` qui calcule l'opposé en représentation en complément à la base. *La fonction `oppose` devra utiliser les opérateurs logiques de la première question.*

(d) En déduire une fonction `sub(x,y)` qui calcule l'opération $x - y$ en complément à la base 2.

1.2 Représentation des grands entiers

Dans cette partie, on s'intéresse à la représentation de grands entiers (positifs). Pour cela, un entier N sera représentée par une liste de mots de taille k : on peut voir cette représentation comme l'écriture de N en base $\beta = 2^k$. On appelle une telle représentation un *entier multi-précision*.

Les mots de taille k utilisés pour les entiers multi-précision sont donc des représentations d'entiers positifs (et on n'utilise pas de complément à la base).

Question 1.7. Découpe

(a) Écrire une fonction `decoupe` qui prend en entrée un mot binaire w de taille n et un entier k et renvoie l'entier multi-précision correspondant, c'est-à-dire une liste de mots de taille k telle que si w représente l'entier N en base 2, la liste renvoyée représente N en base 2^k. *Par exemple,* `decoupe('110111001001O1', 4)` *renvoie* `['0011','0111','0010','0101']`.

(b) Écrire une fonction `assemble` qui effectue l'opération inverse.

(c) Exprimer la taille de la liste renvoyée par `decoupe` en fonction de n et k, puis en fonction de N et k.

Question 1.8. Manipulation de grands entiers

(a) Écrire un algorithme `est_superieur` qui prend en entrée deux listes `M` et `N` représentant des entiers multi-précision M et N respectivement, et qui renvoie `True` si $M > N$ et `False` sinon. *On pourra supposer que les listes* `M` *et* `N` *commencent par un mot non nul (différent de* `0...0`*).*

(b) Soit x et y deux entiers représentables sur k bits et $z = \mathrm{add}(x, y, r)$ avec la fonction add définie à la question 1.4. Montrer que

$$z = \begin{cases} x + y + r & \text{si } z \geq x + r, \\ x + y + r - 2^k & \text{sinon.} \end{cases}$$

(c) Écrire une fonction `un` qui prend en entrée un entier k et renvoie la représentation de 1 sur k bits.

(d) Compléter l'algorithme suivant qui calcule la somme de deux grands entiers, représentés comme liste de mots de taille k. *On suppose que* `M` *et* `N` *sont de même taille.*

```
def somme(M, N):
    i = len(N)-1
    S = [''] * (i+1)
    r = False
    while i >= 0:
        S[i] = add(..., ..., ...)
        r = ...
        i -= 1
    if r:
        u = ...
        S = [u] + S
    return S
```

Question 1.9. Représentation récursive Une façon alternative de représenter un grand entier est la représentation récursive suivante :

— un *entier de niveau* 0 est un mot de k bits ;
— pour $\ell > 0$, un *entier de niveau* ℓ est un couple `(M,N)` où `M` et `N` sont des entiers de niveau $\ell - 1$.

Un entier de niveau 0 représente l'entier dont il est l'écriture en base 2. Si `M` et `N` sont deux entiers de niveau $\ell - 1$ représentant respectivement M et N, `(M,N)` représente l'entier $M + 2^{k2^\ell} N$.

(a) Quels sont le plus petit et le plus grand entiers représentables par un entier de niveau ℓ, en fonction de ℓ et de la taille k des entiers de niveau 0 ?

(b) Écrire une fonction `niveau(M)` qui renvoie le niveau d'un entier récursif `M`. *On pourra tester si un entier récursif est de niveau* 0 *grâce au test* `type(M) is str` *qui détermine si* `M` *est de type* `str` *(chaîne de caractère).*

(c) Écrire une fonction `longueur(M)` qui renvoie la longueur des entiers de niveau 0 contenus dans l'entier récursif `M`. *Par exemple,* `longueur((('0010','1101'),('1000','1111')))` *doit renvoyer* 4.

(d) Écrire une fonction `est_nul(M)` qui renvoie `True` si l'entier représenter par `M` vaut 0, et `False` sinon. *On supposera disposer d'une fonction* `nul(w)` *qui teste si un mot de k bit est nul.*

(e) Écrire une fonction `minimal(M)` qui prend en entrée un entier récursif `M` et renvoie l'entier de niveau minimal représentant le même entier que `M`.

(f) Écrire une fonction `aplatit` qui prend en entrée un entier récursif et renvoie sa représentation sous forme d'entier multi-précision.

Question 1.10. Manipulation d'entiers récursifs

(a) Compléter la fonction `est_sup_rec` qui prend en entrée deux entiers de même niveau représentant M et N, et qui renvoie `True` si $M > N$ et `False` sinon.

```
def est_sup_rec(M, N):
    M = minimal(M)
    N = minimal(N)

    if niveau(M) == niveau(N):
        if niveau(M) == 0:
            return ...
        M1, M2 = M
        N1, N2 = N
        if est_sup_rec(M2, N2):
            return True
        elif est_sup_rec(..., ...):
            return False
        elif est_sup_rec(..., ...):
            return True
        return False

    return ...
```

(b) Écrire une fonction `un_rec` qui prend en entrée deux entiers k et ℓ et renvoie l'entier 1 représenté par un entier récursif de niveau ℓ sur k bits.

(c) En s'inspirant de la fonction `somme` pour les entiers multi-précision, écrire une fonction `somme_rec` pour les entiers récursifs.