DS MPSI ITC : Entiers et ensembles

Solution. □

Durée 2h
Aucun appareil électronique n'est autorisé.
Les téléphones doivent être rangés dans les sacs et les sacs placés au fond de la salle.
Répondre sur les feuilles A4 fournies uniquement. Ne pas corner, ne pas agrafer.

Sauf mention du contraire, l'usage des dictionnaires et de toutes les méthodes sur les liste est autorisé.

1 Nombres

Tous les nombres sont écrits en représentation big-endian. Le bit le plus à gauche est donc le bit de poids fort. Les nombres en base 2 sont écrits sous la forme d'une séquence de bits comme

$$b_n b_{n-1} \dots b_1 b_0$$

On numérote dans ce devoir les bits à partir du bit 0 (donc du bit de poids faible).

Le i-ème bit b_i est le coefficient de 2^i dans la représentation binaire du nombre. Ainsi la séquence précédente correspond au nombre :

$$\sum_{i=0}^{n} b_i 2^i.$$

1.1 Codage binaire

Les questions ci-dessous concernent principalement la possibilité d'agir au niveau des bits pour les objets `Python`. Consulter l'appendice 29 pour se renseigner sur les opérateurs bitwise en `Python`.

Question 1.

Exprimer `21 & 36` en base 10.

Solution. $21 \to 10101_2$ et $36 \to 100100_2$ donc

$$\begin{array}{r} 010101 \\ \wedge\, 100100 \\ \hline 000100 \end{array}$$

soit 4 en base 10. □

Question 2.

Exprimer `45 | 26` en base 10.

Solution. $45 \to 101101_2$ et $26 \to 011010_2$ donc

$$\begin{array}{r} 101101 \\ \vee\, 011010 \\ \hline 111111 \end{array}$$

soit 63 en base 10. □

Question 3.

Exprimer `45 ^ 26` en base 10.

Solution. En reprenant les écritures précédentes :

$$\begin{array}{r} 101101 \\ \oplus\, 011010 \\ \hline 110111 \end{array}$$

soit 55 en base 10. □

Question 4.

Exprimer `~ 18` en base 10.

Solution. On utilise la relation $\sim n = -(n+1)$.

$$\sim 18 = -(18+1) = -19.$$

□

Question 5.

Exprimer `23 << 3` en base 10.

Solution. $23 \to 10111_2$ donc

$$10111 \ll 3 = 10111000_2.$$

Or $10111000_2 = 184$. □

Question 6.

Exprimer `-37 >> 2` en base 10.

Solution. On fait un décalage de deux bits à droite.

-37 a un complément à 2 « infini » valant $\overline{\ldots 1111011011}$. On décale de 2 rangs. On obtient $\overline{\ldots 1111110110}$. Ceci a la même valeur que $\overline{10110}^5$. En passant à la base 10 :

$$-2^5 + 2^4 + 2^2 + 2^1 = -32 + 16 + 4 + 2 = -32 + 22 = -10.$$

□

Question 7.

Expliquer pourquoi si `a` et `b` sont deux entiers `Python`, la complexité d'une opération comme `a & b` est logarithmique en $\max(a, b)$.

Solution. Le nombre de bits de a est logarithmique en z ; idem pour b.

Une opération bitwise effectue une boucle sur les bits : il y a un nombre logarithmique (en le max des valeurs) de passages dans la boucle. □

Question 8.

En utilisant uniquement les opérateurs bitwise de `Python`, écrire une fonction `digit(n:int,i:int)->int` qui renvoie le i-ième bit de $n \geq 0$ dans son codage en base 2.

```
print(digit(13,3)) # 1
print(digit(13,1)) # 0
```

Solution. Code

```
def digit(n,i):
    return (n & (1 << i)) >> i

def digit(n,i):
    return (n >> i) & 1
```

□

Question 9.

Écrire la fonction `binary(n:int):list[int]` qui donne le codage de $n \geq 0$ en base 2 sous forme d'une liste de 0 et 1 en notation big-endian. Toutes les méthodes sont autorisées.

```
print(binary(13), binary(0)) # [1, 1, 0, 1] [0]
```

Solution. Code

```
def binary(n):
    if n == 0:
        return [0]
    bits = []
    while n:
        bits.append(n & 1)    # extrait le bit de poids faible
        n >>= 1               # décale vers la droite
    return bits[::-1]         # big-endian

def binary(n):#V2
    if n==0:
        return [0]
    res=[]#2 opérations élémentaires
    while n!=0:#floor(log_2(n)) +1  passages
        res.append(n%2)# 2 opérations
        n = n//2# 2 opérations
    res = res[::-1]# 0(log_2(n)+1) opérations
    return res# 1 opération
```

□

1.2 Complément à 2

On demande de détailler les calculs.

Question 10.

Écrire -27 en complément à 2

1. sur 6 bits
2. sur 10 bits.

Solution. $-27 + 2^6 = 37$ et le binaire de 37 est 100101. Donc $\overline{100101}^6$.
Pour la seconde question on réalise une extension de format en dupliquant le bit de poids fort : $\overline{1111100101}^{10}$. □

Question 11.

Donner en base 10 le nombre dont le complément à 2 sur 5 bits est $\overline{11001}^5$.

Solution. -7 □

Question 12.

Donner -1 en complément à deux sur 64 bits.

Solution. 64 fois 1 □

Question 13.

Écrire en base 10 le nombre m exprimable en complément à 2 sur N bits et dont l'opposé n'est pas exprimable dans ce format.

Solution. -2^{N-1} □

Question 14.

Écrire le nombre m de la question précédente en complément à deux sur N bits.

Solution. 1 suivi de $N-1$ zéros. □

On veut calculer l'opposé d'un nombre x dont on connaît le codage en complément à 2 sur N bits si $x \neq m$: on conserve tous les bits depuis le bit de poids faible jusqu'au premier 1 inclus, et on inverse tous les bits strictement à gauche de ce premier 1.

Question 15.

Soit un nombre x sont le complément à 2 sur N bits s'écrit

$$b_{N-1}\dots b_{k+1}1\underbrace{0\dots0}_{k \text{ zéros}}.$$

1. Écrire le nombre y obtenu en appliquant la méthode décrite plus haut.
2. Montrer que $x+y=0$.

Solution. On note $\bar{b}$ l'opposé du bit b, c'est à dire $1-b$.

$$\begin{array}{rcccccccc} & b_{N-1} & \dots & b_{k+1} & 1 & \underbrace{0} & \dots & 0 \\ & & & & & k \text{ zéros} & & \\ + & \overline{b_{N-1}} & \dots & \overline{b_{k+1}} & 1 & \underbrace{1} & \dots & 1 \\ & & & & & k \text{ uns} & & \\ \hline & 0 & \dots & 0 & 0 & 0 & \dots & 0 \end{array}$$

à partir de la colonne $k+1$ et jusqu'à la colonne $N-1$, il y a propagation de la retenue puisque $(1)+b_{k+i}+\overline{b_{k+i}}=10$. □

2 Ensembles

2.1 Implémentation par des listes

Dans cette sous-section, un ensemble est une liste d'objets supposée sans répétition.

Question 16.

Ecrire la fonction
`check(E:list)->bool` qui indique par un booléen si la liste E peut être considérée comme un ensemble.

```
print(check([3, 1, 4, 2]))
print(check([1, 2, 1, 3]))
```

```
True
False
```

Solution. Code

```
def check(E: list) -> bool:
    vus = {}                # dictionnaire des éléments déjà rencontrés
    for x in E:
        if x in vus:        # doublon détecté
            return False
        vus[x] = True       # on mémorise x
    return True
```

□

Dans toute la suite, on suppose que les listes passées en arguments sont sans doublon et représentent donc bien des ensembles.

Question 17.

Ecrire la fonction `is_empty(E:list)->bool` qui indique si E est l'ensemble vide.

```
print(is_empty([]))
print(is_empty([1]))
```

```
True
False
```

Solution. Code

```
def is_empty(E: list) -> bool:
    return len(E) == 0
```

□

Question 18.

Écrire la fonction `intersection(E:list,F:list)->list`.

```
print(intersection([1,2,3],[2,3,4]))
```

```
[2, 3]
```

Solution. Code

```
#version avec dictionnaire
def intersection(E: list, F: list) -> list:
    vus = {}
    for x in F:
        vus[x] = True
    I = []
    for x in E:
        if x in vus:
            I.append(x)
    return I

#version sans dictionnaire
def intersection(E,F):
    r = []
    for e in E:
       for k in F:
           if e==k:
              r.append(e)
    return r
```

□

Question 19.

Quelle est la complexité en fonction des cardinaux n et m de E et F d'un appel `intersection(E,F)` ?

Solution. Cela dépend du code.

Dans la version sans dictionnaire, la double boucle imbriquée engendre une complexité en $O(n(m+1))$ (le $+1$ au cas où E et F seraient vides mais j'accepte aussi $O(nm)$).

Dans la version avec dictoionnaire, le coût amort d'un ajout étant en $O(1)$, la complexité est un $O(n+m)$ puisqu'on a deux boucles qui se suivent sans être imbriquées. □

Question 20.

Ecrire la fonction `union(E:list,F:list)->list`.

```
print(union([1,2,3],[2,3,4]))
```

```
[1, 2, 3, 4]
```

Solution. Code

```
def union(E: list, F: list) -> list:
    U = []
    vus = {}
    for x in E:
        U.append(x)
        vus[x] = True
    for x in F:
        if x not in vus:
            U.append(x)
    return U
```

□

Question 21.

Ecrire la fonction `setminus(E:list,F:list)->list` représentant $E \setminus F$.

```
print(setminus([1,2,3,4],[2,4]))
```

```
[1, 3]
```

Solution. Code

```
def setminus(E: list, F: list) -> list:
    vus = {}
    for x in F:
        vus[x] = True
    D = []
    for x in E:
        if x not in vus:
            D.append(x)
    return D
```

□

Question 22.

Ecrire la fonction `sym_diff(E:list,F:list)->list` représentant la différence symétrique.

```
print(sym_diff([1,2,3],[2,4]))
```

```
[1, 3, 4]
```

Solution. Code

```
def sym_diff(E: list, F: list) -> list:
    vusE = {}
    vusF = {}
    for x in E:
        vusE[x] = True
    for x in F:
        vusF[x] = True

    S = []

```

```
    for x in E:
        if x not in vusF:
            S.append(x)

    for x in F:
        if x not in vusE:
            S.append(x)

    return S

#V2
def sym_diff(E,F):
    return setminus(union(E,F),intersection(E,F))
```

□

Question 23.

On suppose dans cette question que les ensembles E et F sont représentés par des listes triées par ordre croissant.
Ecrire une fonction `intersection(E:list,F:list)->list` qui calcule $E \cap F$ avec une complexité linéaire en $|E|+|F|$, sans utiliser de dictionnaire.

```
print(intersection([1,3,5,7],[2,3,6,7,8]))
```

```
[3, 7]
```

Solution. Code (principe analogue à la fusion du tri fusion)

```
def intersection(E: list, F: list) -> list:
    i = 0
    j = 0
    I = []

    while i < len(E) and j < len(F):
        if E[i] == F[j]:
            I.append(E[i])
            i += 1
            j += 1
        elif E[i] < F[j]:
            i += 1
        else:
            j += 1

    return I
```

À chaque itération, au moins un des indices i ou j est incrémenté. Chacun parcourt sa liste une seule fois, d'où une complexité

$$\mathcal{O}(|E| + |F|).$$

□

2.2 Implémentation par un entier

On code désormais un ensemble fini d'entiers positifs E par un entier e dont le bit de rang k vaut 1 si et seulement si $k \in E$. Par exemple, si $E = \{0, 7, 12\}$ alors $e = 4225$ puisque

$$2^{12} + 2^7 + 2^0 = 4225.$$

Si on préfère, le codage binaire de 4225 étant 1000010000001_2, les positions des bits à 1 sont exactement les nombres présents dans l'ensemble.

Question 24.

Écrire la fonction `is_empty(n:int)->bool` qui teste si l'ensemble codé par n est vide.

```
print(is_empty(0))
print(is_empty(4225))
```

```
True
False
```

Solution. Code

```
def is_empty(n: int) -> bool:
    return n == 0
```

□

Question 25.

Écrire la fonction `is_in(e:int,x:int)->bool` qui indique si $x \in E$.

```
# E = {0,7,12} -> 4225
print(is_in(4225, 7))
print(is_in(4225, 9))
```

```
True
False
```

Solution. Code

```
def is_in(e: int, x: int) -> bool:
    return (e & (1 << x)) != 0
```

On décale le bit 1 de x positions vers la gauche. Le masque obtenu vaut 2^x. Si le bit de rang x est présent dans e, alors l'opération `&` est non nulle. □

Question 26.

Écrire la fonction `intersection(a:int,b:int)->int` qui renvoie l'entier codant $A \cap B$.

```
# A = {0,7,12} -> 4225
# B = {0,7,9}  -> 641
print(intersection(4225, 641))
```

```
129
```

Solution. Code

```
def intersection(a: int, b: int) -> int:
    return a & b
```

□

Question 27.

La variables `a` et `b` étant entières positives, quelle est la complexité de l'appel `intersection(a,b)` ?

Solution. Logarithmique en le maximum des deux valeurs. □

Question 28.

Écrire la fonction `union(a:int,b:int)->int` qui renvoie l'entier codant $A \cup B$.

```
# A = {0,7,12} -> 4225
# B = {0,7,9}  -> 641
print(union(4225, 641))
```

```
4737
```

Solution. Code

```
def union(a: int, b: int) -> int:
    return a | b
```

□

Question 29.

Écrire la fonction `sym_diff(a:int,b:int)->int` qui renvoie l'entier codant la différence symétrique $A \triangle B = (A \setminus B) \cup (B \setminus A)$.

```
# A = {0,7,12} -> 4225
# B = {0,7,9}  -> 641
print(sym_diff(4225, 641))
```

```
4608
```

Solution. Code

```
def sym_diff(a: int, b: int) -> int:
    return a ^ b
```

□

Appendice

Les **opérateurs bitwise** (opérateurs au niveau des bits) en Python permettent de manipuler directement l'écriture binaire des entiers.

AND bit à bit : `&`

```
a = 12      # 1100
b = 10      # 1010
print(a & b)
```

```
8
```

On obtient 1000_2. Règle : $1 \& 1 = 1$, sinon 0.

OR bit à bit : `|`

```
print(a | b)
```

```
14
```

On obtient 1110_2. Règle : $0|0 = 0$, sinon 1.

XOR (ou exclusif) : `^`

```
print(a ^ b)
```

```
6
```

On obtient 0110_2. Règle : 1 si les bits sont différents, 0 sinon.

NOT (complément bit à bit) : `~`

```
print(~a)
```

```
-13
```

En Python, les entiers sont en représentation signée (tout est fait pour faire croire à l'utilisateur qu'ils sont stockés sous forme de complément à deux avec taille illimitée). On a la formule :

$$\sim a = -(a+1)$$

Décalage à gauche : <<

```
print(a << 2)
```

```
48
```

$1100_2 \ll 2 = 110000_2$. Cela correspond à :

$$a \ll n = a \times 2^n$$

Décalage à droite : >>

```
print(a >> 2)
```

```
3
```

$1100_2 \gg 2 = 0011_2$.
On fait un décalage de n bits à droite. Dans le cas d'un entier positif, cela correspond à :

$$a \gg n = \left\lfloor \frac{a}{2^n} \right\rfloor$$

avec propagation du bit de signe si l'entier est négatif (le résultat reste négatif).

Attention : on décale vers la doite de n rangs une représentation en CA2.

Divers

```
print((13 >> 3) & 1, (13 >> 1) & 1)
```

```
1 0
```