De très grands entiers

Cette épreuve a pour objet la réalisation d'une bibliothèque d'entiers de précision arbitraire. On se place dans le cadre du langage Python et on se propose donc de construire une alternative aux entiers natifs de Python, dont on rappelle qu'ils sont déjà de précision arbitraire.

Préliminaires

Python. Les entiers natifs de Python, de type `int`, sont de précision arbitraire. Si `n` et `m` sont deux entiers Python positifs ou nuls, alors `n << m` est l'entier $\mathtt{n} \times 2^{\mathtt{m}}$ et `n >> m` est l'entier $\lfloor \mathtt{n}/2^{\mathtt{m}} \rfloor$. L'entier `n & m` (resp. `n | m` et `n ^ m`) est le ET logique (resp. le OU logique et le OU exclusif logique) des entiers `n` et `m`, c'est-à-dire que le i-ième bit de `n & m` (resp. `n | m` et `n ^ m`) est obtenu en faisant le ET (resp. OU et OU exclusif) des i-ièmes bits de `n` et `m`. L'addition de deux entiers `n` et `m` a un coût en $O(\max(\log \mathtt{n}, \log \mathtt{m}))$. La représentation d'un entier `n` occupe un espace $O(\log \mathtt{n})$.

Dans le langage Python, toute valeur est représentée par un objet, dont l'identité peut être obtenue avec la fonction `id`. Il s'agit là d'un entier, garanti unique et constant pendant toute la durée de vie de cet objet. Par ailleurs, on peut déterminer si deux objets `x` et `y` sont identiquement les mêmes avec le booléen `x is y`, ou au contraire distincts avec `x is not y`.

Le langage Python fournit nativement une structure de *dictionnaire*. On crée un nouveau dictionnaire, vide, avec `{}`. Si `d` est un dictionnaire et `k` une clé, on teste la présence d'une valeur associée à la clé `k` avec `k in d` et, le cas échéant, on récupère la valeur associée avec `d[k]`. On associe une valeur `v` à la clé `k` avec `d[k] = v` (et toute valeur précédemment associée à `k`, le cas échéant, est écrasée). L'ajout se fait en place. De même, le langage Python fournit nativement une structure d'*ensemble*. On crée un nouvel ensemble, vide, avec `set()`. Si `s` est un ensemble, on teste la présence d'un élément `x` dans `s` avec `x in s` et on ajoute l'élément `x` à `s` avec `s.add(x)`. L'ajout se fait en place.

En interne, un dictionnaire ou un ensemble est réalisé par une *table de hachage*, sur la base des méthodes `__hash__` et `__eq__` fournies par la classe des clés (pour un dictionnaire) ou des éléments (pour un ensemble). Ces deux méthodes se doivent d'être *cohérentes*, c'est-à-dire

$$\text{pour tous } x \text{ et } y, \text{ si } x.\texttt{\_\_eq\_\_}(y) = \texttt{True} \text{ alors } x.\texttt{\_\_hash\_\_}() = y.\texttt{\_\_hash\_\_}(). \qquad (1)$$

Le langage Python fournit nativement une structure de *tableau redimensionnable*, appelé « liste ». On crée une liste vide avec `[]`. Si `t` est une liste, sa longueur est donnée par `len(t)`. Pour un entier `k` tel que $0 \leq \mathtt{k} < \mathtt{len(t)}$, on accède au `k`-ième élément de `t` avec `t[k]` et on le modifie avec `t[k] = v`. Ces deux opérations se font en temps constant. On étend la liste `t` avec un nouvel élément `v`, à la position `len(t)`, avec `t.append(v)` (et `len(t)` est incrémenté). Inversement, l'opération `t.pop()` supprime et renvoie le dernier élément de la liste `t`. En pratique, on considérera que les deux opérations `append` et `pop` se font également en temps constant.

Pour ouvrir un fichier texte `file` en lecture, on peut utiliser `open(file, 'r')`. On obtient alors un objet, avec notamment une méthode `readlines()` qui renvoie une liste contenant toutes les lignes du fichier comme autant de chaînes de caractères. Par ailleurs, si `s` est une chaîne de caractères, `s.split()` renvoie la liste de tous les mots non vides de `s`, dans l'ordre, les mots étant séparés par des caractères blancs (espaces et retours chariot). Ainsi, `"a␣bb␣c\n".split()` renvoie la liste `["a", "bb", "c"]`.

Complexité. Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. On considérera que toutes les opérations élémentaires de Python (affectation, comparaison avec `is`, accès à un élément de liste, etc.) s'effectuent en temps constant. La complexité sera exprimée sous la forme $O(f(n, m))$ où n et m sont les tailles des arguments de la fonction, et f une expression simple. Les calculs de complexité seront justifiés succinctement. Lorsqu'une question de programmation précise qu'une complexité est attendue, sauf demande explicite, il n'est alors pas nécessaire de justifier que la fonction écrite vérifie cette contrainte. Pour les calculs d'espace, on considérera qu'un pointeur occupe un espace constant.

Dépendances. Ce sujet contient plusieurs parties. Chaque partie utilise des définitions et des résultats des parties précédentes. Les questions restent néanmoins indépendantes, au sens où toute question peut être traitée en admettant les résultats énoncés dans les questions précédentes.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Partie I. Principe

Pour représenter de grands entiers, on exploite l'observation suivante : tout entier naturel s'écrit de manière unique

— soit comme l'entier 0 ;
— soit comme l'entier 1 ;
— soit comme $h \times 2^{2^p} + \ell$ avec $0 < h < 2^{2^p}$ et $0 \leq \ell < 2^{2^p}$.

Dans ce dernier cas, on note $\langle h, p, \ell \rangle$ ce triplet. Ainsi, l'entier 42 s'écrit $\langle 2, 2, 10 \rangle$ car $42 = 2 \times 2^{2^2} + 10 = 2 \times 16 + 10$. De même, l'entier 10 s'écrit $\langle 2, 1, 2 \rangle$ car $10 = 2 \times 2^{2^1} + 2 = 2 \times 4 + 2$ et l'entier 2 s'écrit $\langle 1, 0, 0 \rangle$ car $2 = 1 \times 2^{2^0} + 0 = 1 \times 2 + 0$.

À cette décomposition, on rajoute l'idée qu'un même triplet peut être construit de manière unique en mémoire. On a alors une représentation sous forme d'un graphe où les sommets sont des triplets $\langle h, p, \ell \rangle$, avec h, p et ℓ étant 0, 1 ou une référence à un autre sommet. Une telle représentation des entiers est baptisée IDD (pour *Integer Dichotomy Diagrams*). La figure 1 illustre l'IDD représentant l'entier 42.

Question 1. Dessiner l'IDD correspondant à l'entier 773.

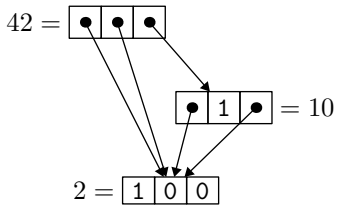

FIGURE 1 – Représentation de l'entier 42 par un IDD.

Nombres énormes. Pour $n \in \mathbb{N}$, on définit le *nombre énorme* $b(n)$ de la manière suivante :

$$\begin{array}{rcl} b(0) & \stackrel{\text{def}}{=} & 1 \\ b(n+1) & \stackrel{\text{def}}{=} & \langle b(n), b(n), b(n) \rangle. \end{array} \tag{2}$$

Question 2. Donner la valeur de $b(1)$ en base 10. Donner la valeur de $b(2)$ en base 2.

Taille d'un entier. On définit la *taille* d'un entier n, notée $\mathbf{s}(n)$, comme le nombre de sommets distincts dans l'IDD qui le représente, les entiers 0 et 1 n'étant pas comptés comme des sommets. Ainsi, $\mathbf{s}(42) = 3$ au regard de la figure 1. En particulier, $\mathbf{s}(0) = \mathbf{s}(1) = 0$.

Question 3. Montrer que $b(n)$ est le plus grand entier i tel que $\mathbf{s}(i) \leq n$.

Partie II. Représentation en Python

On s'intéresse maintenant à la construction des IDD dans le langage Python. Un IDD est représenté par un objet de la classe `IDD`, dont le code est donné dans la figure 2. Cet objet possède trois champs, `hi`, `p` et `lo`, qui sont eux-mêmes trois objets de la classe `IDD`, et il représente l'entier $\langle \texttt{hi}, \texttt{p}, \texttt{lo} \rangle$. Pour assurer l'unicité en mémoire d'un même entier, on utilise la technique du *hash-consing* : une table globale, `table` (ligne 4), contient tous les IDD déjà construits. Il s'agit d'un dictionnaire dont les clés sont des objets de la classe `Triple` (lignes 28–41) et dont les valeurs sont des objets de la classe `IDD`. Pour construire un `IDD`, on se sert de la fonction `IDD.create` (ligne 11). Elle consulte la table pour déterminer si l'objet a déjà été construit (ligne 15). Le cas échéant, elle le renvoie (ligne 16). Sinon, l'objet est construit, ajouté à la table et renvoyé (lignes 18–20).

La classe `Triple` (lignes 28–41) représente un triplet de trois `IDD`, avec trois composantes `hi`, `p` et `lo`. La classe `Triple` est munie d'une fonction de hachage (lignes 35–36) et d'une fonction d'égalité (lignes 38-41), qui sont appelées lorsque le dictionnaire `table` est utilisé.

Enfin, on construit deux objets particuliers, `zero` et `one`, pour représenter les entiers 0 et 1 (lignes 43–44). Pour faciliter le code de certaines fonctions, il est pratique de définir les trois composantes de `zero` et `one` en termes de ces mêmes objets (lignes 45–46).

Tournez la page S.V.P.

```
1  class IDD:
2      """cet objet a 3 champs: hi,p,lo, trois autres IDD"""
3
4      table = {}
5
6      def __init__(self, hi, p, lo):
7          self.hi = hi
8          self.p = p
9          self.lo = lo
10
11     def create(hi, p, lo):
12         if hi is zero:
13             return lo
14         t = Triple(hi, p, lo)
15         if t in IDD.table:
16             return IDD.table[t]
17         else:
18             i = IDD(hi, p, lo)
19             IDD.table[t] = i
20             return i
21
22     def __hash__(self):
23         return id(self)
24
25     def __eq__(self, other):
26         return self is other
27
28 class Triple:
29     """cette classe sert uniquement de clé dans table"""
30     def __init__(self, hi, p, lo):
31         self.hi = hi
32         self.p = p
33         self.lo = lo
34
35     def __hash__(self):
36         return 31*(31*id(self.hi) + id(self.p)) + id(self.lo)
37
38     def __eq__(self, other):
39         assert isinstance(other, Triple)
40         return self.hi is other.hi and self.p is other.p and \
41                self.lo is other.lo
42
43 zero = IDD(None, None, None)
44 one = IDD(None, None, None)
45 zero.hi, zero.p, zero.lo = zero, zero, zero
46 one.hi, one.p, one.lo = zero, zero, one
```

FIGURE 2 – Construction des IDD en Python.

Par la suite, le constructeur de la classe `IDD` ne sera jamais utilisé. On utilisera exclusivement la fonction `IDD.create`. De plus, on suppose que seule la fonction `IDD.create` accède au dictionnaire stocké dans `table`. Ainsi, pour construire une valeur de type `IDD` représentant l'entier 42, on peut écrire le code suivant :

```
i2  = IDD.create(one, zero, zero)
i10 = IDD.create(i2, one, i2)
i42 = IDD.create(i2, i2, i10)
```

Dans tout le reste de ce sujet, on utilise la variable `n` pour désigner un entier Python de type `int` et les variables `i` et `j` pour désigner des entiers IDD de type `IDD`.

Question 4. Expliquer pourquoi il est légitime de considérer que la fonction `IDD.create` s'exécute en temps constant.

Question 5. Justifier la définition des méthodes `__hash__` (lignes 22–23) et `__eq__` (lignes 25–26) de la classe `IDD`. Expliquer pourquoi on peut considérer que les opérations d'ajout et de recherche dans un ensemble d'éléments de type `IDD` ou dans un dictionnaire dont les clés sont de type `IDD` s'exécutent en temps constant.

Question 6. Donner le code Python d'une fonction `big(n: int) -> IDD` correspondant à la fonction b (équation (2) page 3). La complexité en temps doit être en $O(\texttt{n})$, mais il n'est pas demandé de la justifier.

Question 7. Donner le code Python d'une fonction `size(i: IDD) -> int` qui renvoie la taille de l'entier `i`, c'est-à-dire $\texttt{s}(\texttt{i})$. On rappelle que les objets `zero` et `one` ne doivent pas être décomptés. La complexité en temps doit être $O(\texttt{s}(\texttt{i}))$. On justifiera la complexité.

Poids de Hamming. Le poids de Hamming d'un entier naturel n, noté $\mathbf{pop}(n)$ (pour *population count*), est le nombre de chiffres 1 dans l'écriture de n en base 2. Ainsi, $\mathbf{pop}(42) = 3$ car $42 = 101010_2$. La figure 3 contient une fonction `pop` qui renvoie le poids de Hamming d'un IDD.

Question 8. Montrer que la fonction `pop` est correcte. *Indication : Proposer un invariant pour le dictionnaire* `memo` *et montrer que la fonction* `compute` *le préserve.*

Question 9. Donner et justifier la complexité de la fonction `pop`, en fonction de $\texttt{s}(\texttt{i})$. Peut-on calculer `pop(big(100))`, qui vaut 2^{100}, en un temps raisonnable ?

Tournez la page S.V.P.

```
def pop(i: IDD) -> int:
    """nombre de 1 dans la représentation en base 2 de i"""
    memo = {}
    memo[zero] = 0
    memo[one] = 1
    def compute(i):
        if i in memo:
            return memo[i]
        else:
            v = compute(i.hi) + compute(i.lo)
            memo[i] = v
            return v
    return compute(i)
```

FIGURE 3 – Fonction `pop`.

```
def to_int(i: IDD) -> int:
    """convertit un IDD en un entier Python"""
    if i is zero:
        return 0
    elif i is one:
        return 1
    else:
        return to_int(i.hi) << (1 << to_int(i.p)) | to_int(i.lo)
```

FIGURE 4 – Fonction `to_int`.

Partie III. Conversions

Dans cette partie, on étudie différentes opérations de conversions vers et depuis d'autres formats de représentation.

Avec le type `int`. La figure 4 contient le code d'une fonction `to_int` qui convertit un IDD vers le type `int` de Python, en supposant que la mémoire est suffisamment grande pour stocker le résultat.

Question 10. Montrer que la fonction `to_int` est correcte.

Question 11. La fonction récursive `to_int` est-elle susceptible de faire déborder la pile d'appels de Python (par défaut limitée à 1000 appels imbriqués) ?

Question 12. Donner le code d'une fonction Python `of_int(n: int) -> IDD` qui convertit un entier Python en IDD.

Sérialisation. Pour écrire un IDD $n \geq 2$ dans un fichier, on se propose d'utiliser le format texte suivant. Le fichier contient exactement $\mathbf{s}(n)$ lignes et chaque ligne est de la forme

```
i j k l
```

où `i`, `j`, `k` et `l` sont quatre entiers, avec $2 \leq \texttt{i} \leq \mathbf{s}(n)+1$ et $0 \leq \texttt{j}, \texttt{k}, \texttt{l} < \texttt{i}$. L'entier `i` numérote la ligne, à partir de 2. Cette ligne définit un nouvel IDD, numéroté `i`, comme valant $\langle j, k, \ell \rangle$ où j, k et ℓ sont les trois IDD respectivement numérotés `j`, `k` et `l`. Les numéros 0 et 1 font référence aux IDD 0 et 1. Le fichier représente l'IDD défini par la dernière ligne. Ainsi, l'IDD représentant l'entier 42 peut être sérialisé par les trois lignes suivantes :

```
2 1 0 0
3 2 1 2
4 2 2 3
```

Ces trois lignes correspondent aux trois IDD de la figure 1.

Question 13. Cette représentation est-elle unique ? Si oui, justifier. Sinon, donner deux fichiers définissant le même entier.

Question 14. Décrire un algorithme pour réaliser cette sérialisation, c'est-à-dire pour imprimer successivement les lignes du fichier qui représente un IDD n donné. Donner sa complexité. On ne demande pas d'écrire le code Python.

Question 15. Donner le code Python d'une fonction `parser(file: str) -> IDD` qui reconstruit l'IDD décrit par le contenu du fichier `file`. On suppose que ce fichier contient la sérialisation d'un IDD, *i.e.*, on ne demande pas de vérifier la bonne formation de ce fichier. La complexité en temps doit être proportionnelle à la taille du fichier, mais il n'est pas demandé de la justifier.

Partie IV. Arithmétique

Question 16. Donner le code Python d'une fonction `compare(i: IDD, j: IDD) -> int` qui compare deux IDD. Elle renvoie l'entier -1 si $\texttt{i} < \texttt{j}$, l'entier 0 si $\texttt{i} = \texttt{j}$ et l'entier 1 si $\texttt{i} > \texttt{j}$.

Incrémentation et décrémentation. La figure 5 contient le code de deux fonctions `xp` et `pred`. La fonction `xp` calcule $2^{2^{\texttt{i}}} - 1$ pour un argument $\texttt{i} \geq 0$. La fonction `pred` calcule $\texttt{i} - 1$ pour un argument $\texttt{i} > 0$.

Question 17. Montrer que le calcul de `xp(i)` ou de `pred(i)` termine toujours.

Question 18. En utilisant la fonction `xp`, donner le code Python d'une fonction `succ(i: IDD) -> IDD` qui calcule $\texttt{i} + 1$.

Tournez la page S.V.P.

```
1  def xp(i: IDD) -> IDD:
2      """2^(2^i)-1"""
3      if i is zero:
4          return one
5      else:
6          p = pred(i)
7          j = xp(p)
8          return IDD.create(j, p, j)
9
10 def pred(i: IDD) -> IDD:
11     """prédécesseur i-1, pour i>0"""
12     assert i is not zero
13     if i is one:
14         return zero
15     elif i.lo is not zero:
16         return IDD.create(i.hi, i.p, pred(i.lo))
17     elif i.hi is one:
18         return xp(i.p)
19     else:
20         return IDD.create(pred(i.hi), i.p, xp(i.p))
```

FIGURE 5 – Fonctions `xp` et `pred`.

Ajout/retrait du bit de poids fort. Pour $n > 0$, on pose $R(n) = (m, i)$ avec $n = m + 2^i$ et $0 \leq m < 2^i$. Dit autrement, i est le bit de poids fort de l'entier n. Inversement, on pose $I(m, i) = m + 2^i$ pour $0 \leq m < 2^i$. Pour calculer $R(n)$ et $I(m, i)$ sur les IDD, on se donne les équations suivantes :

$$
\begin{aligned}
R(1) &= (0,0) && && (3)\\
R(\langle h,p,\ell\rangle) &= (\langle m,p,\ell\rangle,\, I(i,p)) && \text{si } m \neq 0 && (4)\\
&= (\ell,\, I(i,p)) && \text{sinon} && (5)\\
&\quad \text{avec } (m,i) = R(h) && && \\
I(0,0) &= 1 && && (6)\\
I(0,1) &= \langle 1,0,0\rangle && && (7)\\
I(1,1) &= \langle 1,0,1\rangle && && (8)\\
I(\langle h,p,\ell\rangle, i) &= \langle I(0,e), j, \langle h,p,\ell\rangle\rangle && \text{si } j > p && (9)\\
&= \langle I(h,e), j, \ell\rangle && \text{sinon} && (10)\\
&\quad \text{avec } (e,j) = R(i) && &&
\end{aligned}
$$

Question 19. Justifier les équations (3)–(10).

Dans la suite, on suppose avoir écrit deux fonctions Python réalisant ces calculs, sous la forme suivante :

```
def rmsb(i: IDD) -> Tuple[IDD, IDD]:
    """extrait le bit 1 de point fort de i>0, c'est-à-dire
       renvoie (m, j) avec i = m + 2^j et 0 <= m < 2^j"""
    ...
def imsb(i: IDD, j: IDD) -> IDD:
    """renvoie i + 2^j pour 0 <= i < 2^j"""
    ....
```

Question 20. Donner le code Python d'une fonction `power2(i: IDD) -> IDD` qui calcule $2^{\mathtt{i}}$.

Question 21. Donner le code Python d'une fonction `binary_length(i: IDD) -> IDD` qui calcule le nombre de chiffres dans l'écriture en base 2 de l'entier `i`.

Question 22. Donner le code d'une fonction Python `print2(i: IDD)` qui imprime l'entier `i` en base 2. Ainsi, `print2(of_int(42))` doit afficher

```
101010
```

On suppose que le nombre de chiffres est suffisamment raisonnable pour que l'impression ait une chance de terminer.

Autres opérations arithmétiques. Il est également possible de définir les opérations d'addition, de soustraction, de multiplication ou encore de division sur les IDD, mais cela dépasserait le cadre de ce sujet.

Partie V. Opérations logiques et applications

La structure des IDD est parfaitement adaptée à la définition d'opérations logiques (ET, OU, OU exclusif) sur l'écriture en binaire des entiers correspondants. Ainsi, on peut définir le ET logique de deux IDD s et t, noté $s \wedge t$, avec les cinq équations suivantes :

$$0 \wedge t = 0 \tag{11}$$
$$s \wedge s = s \tag{12}$$
$$s \wedge t = t \wedge s \quad \text{si } s > t \tag{13}$$
$$s \wedge t = s \wedge t.\mathtt{lo} \quad \text{si } s.\mathtt{p} < t.\mathtt{p} \tag{14}$$
$$s \wedge t = \langle s.\mathtt{hi} \wedge t.\mathtt{hi}, s.\mathtt{p}, s.\mathtt{lo} \wedge t.\mathtt{lo} \rangle \quad \text{sinon} \tag{15}$$

Question 23. Donner des équations comparables pour la définition de l'opération OU (notée $s \vee t$) et OU exclusif (notée $s \oplus t$).

Question 24. On pourrait écrire une fonction Python `log_and(i: IDD, j: IDD) -> IDD`, récursive, qui suive exactement les équations (11)–(15). Montrer que sa complexité en temps pourrait être exponentielle en les tailles $\mathtt{s}(\mathtt{i})$ et $\mathtt{s}(\mathtt{j})$ de ses arguments.

Tournez la page S.V.P.

Question 25. En utilisant le principe de mémoïsation, donner le code Python d'une fonction `log_and` plus efficace dont la complexité en temps est $O(\mathtt{s}(\mathtt{i}) \times \mathtt{s}(\mathtt{j}))$. On justifiera la complexité. *Indication : On pourra introduire une classe pour des paires d'IDD servant de clés dans un dictionnaire.*

Dans la suite, on suppose qu'on a écrit de la même façon des fonctions `log_or` et `log_xor`, également de complexité en temps $O(\mathtt{s}(\mathtt{i}) \times \mathtt{s}(\mathtt{j}))$.

Application : ensembles d'entiers naturels. Un ensemble fini d'entiers naturels $S \subseteq \mathbb{N}$ peut être représenté par un entier n en posant

$$n = \sum_{i \in S} 2^i. \tag{16}$$

Dit autrement, les chiffres 1 dans la représentation de n en base 2 indiquent les éléments de l'ensemble S. Ainsi, l'ensemble $S = \{1, 4, 5, 8, 9\}$ est représenté par l'entier 818, car $818 = 1100110010_2$.

Un ensemble S est donc naturellement représenté par l'IDD qui encode l'entier n défini par (16). En particulier, les fonctions `log_and`, `log_or` et `log_xor` introduites plus haut calculent respectivement l'intersection, l'union et la disjonction exclusive de deux ensembles.

Question 26. Donner le code Python d'une fonction `difference(s: IDD, t: IDD) -> IDD` qui calcule la différence ensembliste pour des ensembles représentés par des IDD.

Question 27. Donner le code Python d'une fonction `mem(n: int, s: IDD) -> bool` qui détermine si l'entier `n` appartient à l'ensemble représenté par `s`.

Question 28. Donner le code Python d'une fonction `subset(s: IDD, t: IDD) -> bool` qui détermine si $\mathtt{s} \subseteq \mathtt{t}$ pour deux ensembles représentés par des IDD.

Question 29. Justifier, informellement, que l'espace mémoire occupé par la représentation physique d'un IDD n est proportionnel à sa taille $\mathtt{s}(n)$.

Question 30. On admet l'inégalité $\mathtt{s}(n) \leq \mathbf{pop}(n) \times (n.\mathtt{p} + 1)$ pour tout IDD n. Soit S un ensemble de K entiers, qui s'écrivent tous sur au plus W bits. Majorer l'espace utilisé par un IDD qui représente S selon (16), en fonction de K et W. Comparer avec l'espace utilisé par une liste d'entiers Python pour représenter ce même ensemble.

Question 31. Montrer que l'espace *total* occupé par les IDD représentant tous les entiers $2, 3, \ldots, n$ est en $O(n)$.

Question 32. Soit $S = \{0, 1, \ldots, K-1\}$. Comparer l'espace occupé simultanément par tous les sous-ensembles de S, dans les deux cas suivants :

1. les ensembles sont représentés par des IDD ;
2. les ensembles sont représentés par des listes d'entiers.

Partie VI. Pour aller plus loin

Question 33. La structure des IDD ne permet pas de tirer parti des opérations natives fournies par la machine, comme par exemple des opérations arithmétiques ou logiques sur 64 bits fournies par le processeur. Proposer une adaptation de la structure des IDD à même de tirer le meilleur parti d'une arithmétique native sur W bits.

Question 34. Dans ce sujet, nous nous sommes limités à des entiers naturels. Proposer une extension de cette bibliothèque à des entiers relatifs. On ne demande pas d'écrire le code mais on demande d'être précis quant à la représentation choisie et à l'adaptation des différentes opérations.

* *
*