Mathis Degryse <mathis.degryse@proton.me>

Problème 2022
De très grands entiers

Partie I. Principe

Question 1
On a $773 = 3 \times 2^{2^3} + 5 = \langle 3, 3, 5 \rangle$. De plus, $5 = \langle 1, 1, 1 \rangle$ et $3 = \langle 1, 0, 1 \rangle$. On détermine ainsi un IDD représentant 773, en **Fig. 1**.

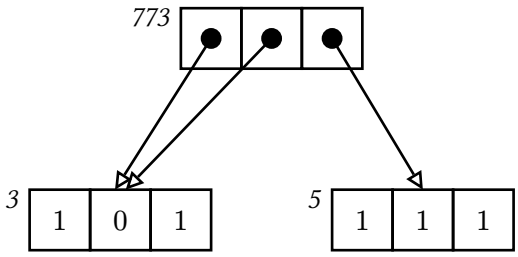

Fig. 1. – IDD représentant l'entier 773

Question 2
On a $b(1) = \langle b(0), b(0), b(0) \rangle = \langle 1, 1, 1 \rangle = 1 \times 2^{2^1} + 1 = 5$.

De même, on calcule $b(2) = 5 \times 2^{2^5} + 5 = 2^{34} + 2^{32} + 2^2 + 2^0$.

Question 3
Commençons par montrer par récurrence sur $n \in \mathbb{N}$ que $s(b(n)) \leq n$.

- Initialisation : pour $n = 0$, on a $s(b(0)) = s(1) = 0 \leq 0$
- Hérédité : Soit $n \in \mathbb{N}$. Supposons que $s(b(n)) \leq n$. On a $b(n+1) = \langle b(n), b(n), b(n) \rangle$. Ainsi, si on considère le graphe B_n représentant l'entier $b(n)$, qui possède moins de n sommets, alors le graphe en **Fig. 2** représente $b(n+1)$ et possède moins de $n+1$ sommets.
  On a donc $s(b(n+1)) \leq n+1$
- Finalement, pour tout $n \in \mathbb{N}$, $s(b(n)) \leq n$

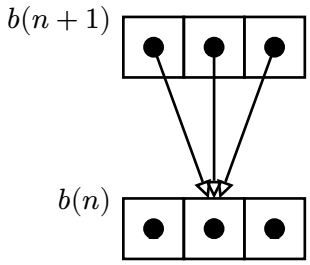

Fig. 2. – IDD représentant l'entier b(n+1)

Montrons maintenant, par récurrence sur $n \in \mathbb{N}$ la propriété suivante:

- $P(n)$: Pour tout $x \in \mathbb{N}$, si $x > b(n)$, alors $s(x) > n$
- Initialisation : Pour $n = 0$, si $x > 1$, alors $x = \langle h, p, l \rangle$ pour certains $h, p, l \in \mathbb{N}$. Ainsi, l'IDD représentant x utilise au moins 1 sommet et donc $s(x) > 0$
- Hérédité : Soit $n \in \mathbb{N}$, supposons $P(n)$. Soit $x > b(n+1)$, il existe $h, p, l \in \mathbb{N}$ tel que $x = \langle h, p, l \rangle$. On sait que l'un des trois entiers h, p, l est plus grand que $b(n)$. En effet, supposons par l'absurde que $h, p, l \leq b(n)$ alors on a $x = \langle h, p, l \rangle \leq \langle b(n), b(n), b(n) \rangle = b(n+1)$. Ce qui

contredit l'hypothèse faite sur x.
On en déduit que l'IDD représentant x utilise au moins 1 sommet ainsi que ceux d'un entier supérieur à $b(n)$. Or d'après $P(n)$, l'IDD représentant cet entier possède au moins $n+1$ sommets. Finalement, $s(x) > n+1$.

- On en déduit que pour tout $n \in \mathbb{N}$, $P(n)$ est vraie

De ces deux propriétés, on obtient le résultat voulu, i.e. pour tout $n \in \mathbb{N}$, $b(n)$ est le plus grand entier i tel que $s(i) \leq n$.

Partie II. Représentation en Python

Question 4

Les arguments `hi, p, lo` sont des adresses vers les IDD correspondants. C'est pourquoi les opérations d'assignation correspondantes sont en $O(1)$.
Il reste alors les opérations `t in IDD.table` et `IDD.table[t] = i`. On peut considérer que ces deux opérations sont en temps constant, car le dictionnaire `table` est implémenté par une table de hachage dynamique dont les clés sont de type `Triple`. Le coût moyen des opérations (recherche, assignation) est constant, ce qui est garantit par les méthodes `__hash__`, `__eq__` de la classe `Triple`, qui sont en temps constants.

Question 5

Les définition des fonctions `__eq__`, `__hash__` permettent de garantir l'unicité en mémoire des IDDs, à chaque fois qu'on veut créer un nouvel IDD qui existe déjà en mémoire, ce dernier est renvoyé depuis le dictionnaire `table`.
Elles garantissent de plus la propriété (1), i.e. si `i.__eq__(j)` alors `i.__hash__() == j.__hash__()`.
Pour que l'ajout et la recherche dans un dictionnaire dont les clés sont de type `IDD` se fasse en temps constant, il suffit que les méthodes `__eq__` et `__hash__` soient en temps constant, ce qui est bien le cas ici.

Question 6

Voici un code Python correspondant à la fonction b de **Question I-2**

```python
def big(n : int) -> IDD:
  if n == 0:
    return one
  big_prev = big(n-1)
  return IDD.create(big_prev, big_prev, big_prev)
```

Noter qu'il est très important de ne calculer qu'une seule fois `big(n-1)`, afin de garantir la complexité linéaire en `n`.

Question 7

```python
def size(i : IDD) -> int:
  vu = {zero, one}
  def parcours(j : IDD):
    if not(j in vu):
      vu.add(j)
      parcours(j.hi)
```

```
7         parcours(j.p)
8         parcours(j.lo)
9     parcours(i)
10    return len(vu) - 2
```

Cette fonction effectue un parcours en profondeur des sommets de l'IDD `i`. Le corps de la fonction `parcours` n'est exécuté qu'une seule fois par sommet, ce qui est garanti par le dictionnaire `vu`. Ainsi, si il y a n sommets, on a au plus $3n$ appels à `parcours`, qui sont chacun en temps constants, d'après la **Question II-5**.
On renvoie finalement le nombre de sommets croisés (-2 pour zero et one qui sont décomptés). `size(i)` renvoie donc bien la taille de l'IDD `i` en temps linéaire en la taille de celui-ci.

Question 8

- Commençons par montrer que pour tout IDD $\langle h, p, l\rangle$, on a $\mathbf{pop}(\langle h, p, l\rangle) = \mathbf{pop}(h) + \mathbf{pop}(l)$

  En effet, soit $h = \sum_{i\in I_h} 2^i$ et $l = \sum_{i\in I_l} 2^i$. On a alors

  $\langle h, p, l\rangle = h \times 2^{2^p} + l = \sum_{i\in I_h} 2^{i+2^p} + \sum_{k\in I_l} 2^k$

  Or comme $l < 2^{2^p}$, tous les éléments de I_l sont inférieurs à 2^p et donc, si on pose $I = \{i + 2^p \mid i \in I_h\} \cup I_l$, alors on a

  $\langle h, p, l\rangle = \sum_{i\in I} 2^i$.

Finalement, $\mathbf{pop}(\langle h, p, l\rangle) = |I| = |I_h| + |I_l| = \mathbf{pop}(h) + \mathbf{pop}(l)$

- Montrons maintenant que la fonction `pop` correspond à la fonction $\mathbf{pop}$. On considère l'invariant suivant : « Pour tout IDD `i` ∈ `memo`, `memo[i]` = $\mathbf{pop}(i)$ »
  On cherche maintenant à montrer par induction que `compute` préserve cet invariant :
  - Si `i` ∈ `memo`, alors `compute(i)` préserve évidemment l'invariant.
  - Sinon, on suppose que `compute(i.hi)` et `compute(i.lo)` préservent l'invariant. Ainsi, ces appels ont bien renvoyé `memo[i.hi]` et `memo[i.lo]`. Donc la valeur `v` est égale à $\mathbf{pop}(i.hi) + \mathbf{pop}(i.lo) = \mathbf{pop}(i)$. Cette valeur est ensuite stockée dans `memo`, et l'invariant est bien préservé.

Comme `compute(i)` renvoie `memo[i]`, on a bien $\mathbf{pop}(i) =$ `pop(i)`.

Question 9

La fonction `compute` ne calcule la valeur correspondante à un IDD qu'une seule fois. On obtient un résultat similaire à `parcours`. La fonction `compute` est appelé au plus $2 \times s(i)$ fois, mais cette fois-ci, la complexité d'un appel seul n'est pas constante car on additionne des entiers dont la taille est arbitraire.
On peut montrer facilement par induction que le nombre de 1 dans l'écriture binaire de i est inférieure à $2^{s(i)}$.
Or l'addition a une complexité logarithmique en la taille binaire des entiers. Comme $\log\left(2^{s(i)}\right) = s(i)$, la complexité en temps de `pop(i)` est quadratique en $s(i)$.

On peut donc calculer `pop(big(100))` très rapidement.

Partie III. Conversions

Question 10

Montrons que `to_int(i)` renvoie bien l'entier correspondant à l'IDD `i`, par induction sur `i`.

- Le bon fonctionnement pour `zero` et `one` est aisé à vérifier.
- Soit $i = \langle h, p, l \rangle$. On suppose que `to_int` fonctionne sur h, l, p, c'est à dire que `to_int(i.hi)` $= h$, etc... Commençons par remarquer que, `1 << to_int(i.p)` vaut 2^p. Ainsi, `to_int(i.hi) << (1 << to_int(i.p))` vaut $h \times 2^{2^p}$. Maintenant, comme nous l'avons démontré à la **Question II-8**, les écritures binaires de $h \times 2^{2^p}$ et celles de l sont disjointes. Donc la somme de ces deux éléments est égale à leur `OU` bit à bit. Finalement, `to_int(i)` renvoie bien $l + h \times 2^{2^p}$.

Question 11

Admettons qu'on veuille calculer `to_int(big(100))`, dont l'exécution fait au plus 100 appels récursifs imbriqués. Les entiers considérés seront de l'ordre de $2^{2^{100}}$, ce qui n'est pas (en tant qu'entier) représentable par un ordinateur.
Un overflow sera causé bien avant que la pile d'appels ne déborde.

Question 12

l'idée est de d'abord déterminer p. Pour cela, on maintient la variable `power`, qui vaut `2^(2^p)`. On cherche le plus grand `p` vérifiant `power <= n`. On cherche alors cette valeur en incrémentant `p`, ce qui, pour la valeur de `power`, revient à la mettre au carré.
Une fois qu'on a `p`, il est aisé de déterminer `h,l`

```python
def of_int(n):
    if n == 0:
        return zero
    if n == 1:
        return one
    p = 0
    power = 2
    while power * power <= n:
        power *= power
        p += 1
    h = n // power
    l = n % power
    return IDD.create(of_int(h), of_int(p), of_int(l))
```

Question 13

On peut considérer le contre-exemple suivant

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

qui correspond à l'entier 14. Il peut aussi être représenté par :

```
1  2 1 0 1
```

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

Question 14

Un algorithme répondant au problème peut être :

- on maintient un dictionnaire `memo`, tel que `memo[i]` est le numéro de ligne correspondant à l'IDD `i`. Initialement, `zero,one` sont dans le dictionnaire, associées respectivement aux lignes 0 et 1.
- on maintient un entier `id`, qui est le premier identifiant de ligne non utilisé (initialisé à 2).
- on parcourt les IDDs comme suit:
  - Si `i` est dans `memo` ne rien faire
  - sinon, on veut afficher la ligne correspondant à `i`, mais avant, il faut le faire pour les successeurs de `i`. On parcourt donc avec un appel récursif `i.hi, i.p, i.lo`.
    Ensuite, on incrémente `id` et on affiche la ligne grâce à .

Question 15

On maintient un dictionnaire associant à chaque identifiant de ligne l'IDD associé. On peut alors lire chaque ligne et construire un à un les IDD associés. Il suffit finalement de renvoyer le dernier IDD, c'est à dire celui correspondant au dernier identifiant croisé.

```python
def parser(file : str) -> IDD:
    id = 0
    f = open(file, 'r')
    memo = {0 : zero, 1 : one}
    for line in f:
        id, h, p, l = list(map(int,line.split(" ")))
        i = IDD.create(memo[h], memo[p], memo[l])
        memo[id] = i
    return memo[id]
```

Partie IV. Arithmétique

Question 16

Soit $i = \langle h_1, p_1, l_1 \rangle, j = \langle h_2, p_2, l_2 \rangle$ deux IDDs. Remarquons que :

- Si $p_1 > p_2$ alors $i > j$ et inversement.
- Si $p_1 = p_2$ alors si $h_1 > h_2$, on a $i > j$, et inversement.
- Si $h_1 = h_2$ et $p_1 = p_2$, alors on a de même avec l_1, l_2

```python
def compare(i : IDD, j : IDD) -> int:
    if i is j:
      return 0
    if i is zero:
      return -1
    if j is zero:
      return 1
    cp = compare(i.p, j.p)
    if cp != 0:
```

```
10        return cp
11      ch = compare(i.hi, j.hi)
12      if ch != 0:
13        return ch
14      return compare(i.lo, j.lo)
```

Question 17

Commençons par analyser les appels de chaque fonctions.
`xp(i)` fait un appel à `xp(i.p)` et `pred(i)`. Et `pred(i)` fait un appel à `pred(i.lo)` ou `xp(i.p)` et `pred(i.hi)`

Considérons alors l'ensemble $\mathbb{N} \times \{0, 1\}$, qui est bien fondé pour l'ordre lexicographique. On considère le variant suivant : aux appels `xp(i)` on associe la valeur $(i, 1)$ et aux appels `pred(i)` on associe la valeur $(i, 0)$ (on parle de i à la fois comme l'IDD et l'entier qu'il représente). On peut alors vérifier que `xp(i)` et `pred(i)` ne font que des appels récursifs faisant diminuer notre variant, car `i.p, i.lo, i.hi` sont tous inférieurs à `i`. On en déduit que `xp` et `pred` terminent.

Question 18

L'idée est d' « essayer » d'incrémenter `i.lo`. Si jamais `i.lo` atteint sa borne, i.e. `2^(2^(i.p))`, alors on le remplace par `zero` et on incrémente `i.hi`. Si lui même atteint cette même limite, on le met à 0 et on incrémente `i.p`

```python
def succ(i : IDD) -> IDD:
  if i is zero:
    return one
  limit = xp(i.p)
  if i.lo is limit:
    if i.hi is limit:
      return IDD.create(one, succ(i.p), zero)
    return IDD.create(succ(i.hi), i.p, zero)
  return IDD.create(i.hi, i.p, succ(i.lo))
```

Question 19

- (3) : On a $1 = 0 + 2^0$, et $0 < 2^0$, donc $R(1) = (0, 0)$
- (4) : On considère $\langle h, p, l\rangle$ et $(m, i) = R(h)$. On se place dans le cas où $m \neq 0$, i.e. $h = 2^i + m$. On a alors $\langle h, p, l\rangle = 2^{2^p+i} + m \times 2^{2^p} + l$. Le bit de poids fort de ce nombre est bien entendu $2^p + i$, c'est à dire $I(i, p)$, et le reste est $m \times 2^{2^p} + l$, c'est à dire $\langle m, p, l\rangle$
  On a finalement $R(\langle h, p, l\rangle) = (\langle m, p, l\rangle, I(i, p))$.
- (5) : On considère $\langle h, p, l\rangle$ et $(m, i) = R(h)$. On se place dans le cas où $m = 0$, i.e. $h = 2^i$. On a alors $\langle h, p, l\rangle = 2^{2^p+i} + l$. Le bit de poids fort de ce nombre est bien entendu $2^p + i$, c'est à dire $I(i, p)$.
  On a finalement $R(\langle h, p, l\rangle) = (l, I(i, p))$.
- (6) - (7) - (8) : il suffit, comme pour (3), de calculer les expressions.
- (9) : On considère $I(\langle h, p, l\rangle, i) = \langle h, p, l\rangle + 2^i$. Soit $(e, j) = R(i)$, supposons dans ce cas qu'on ait $j > p$.
  On a donc $i = e + 2^j$, et donc

$$\begin{aligned} I(\langle h,p,l\rangle,i) &= h\times 2^{2^p}+l+2^e\times 2^{2^j} \\ &= I(0,e)\times 2^{2^j}+(h\times 2^{2^p}+l) \\ &= \langle I(0,e),j,\langle h,p,l\rangle\rangle \end{aligned}$$

- (10) : Reprenons la même situation, avec cette fois $j \leq p$. Or par définition de I, on a $\langle h,p,l\rangle < 2^i = 2^e \times 2^{2^j}$
  Si jamais on avait $j < p$, on aurait $\langle h,p,l\rangle < 2^e \times 2^{2^p}$, avec $e < 2^p$, ce qui est impossible. On a donc $j = p$

  On a cette fois :

$$\begin{aligned} I(\langle h,p,l\rangle,i) &= h\times 2^{2^p}+l+2^e\times 2^{2^j} \\ &= (h+2^e)\times 2^{2^p}+l \\ &= I(h,e)\times 2^{2^j}+l \\ &= \langle I(h,e),j,l\rangle \end{aligned}$$

Question 20

Il suffit de remplacer `i` par `zero` dans la fonction `imsb`

```python
def power2(i : IDD) -> IDD:
  return imsb(zero, i)
```

Question 21

De même, on peut simplement utiliser `rmsb`. Il faut juste faire attention au cas où `i` est `zero`, car dans ce cas `rmsb` ne peut pas être appelé.

```python
def binary_length(i : IDD) -> IDD:
  if i is zero:
    return one
  (m,j) = rmsb(i)
  return succ(j)
```

Question 22

L'idée est la suivante :
L'écriture binaire de $\langle h,p,l\rangle$ est celle de h, suivie de celle de l, complétée avec des 0 entre les deux. Il suffit alors de déterminer combien de 0 il faut écrire avant d'afficher l. Pour cela, on calcule la longueur de l, disons `right_length`, ainsi que le nombre de bits à écrire après h, disons `total_length`. Ce premier peut se calculer avec `binary_length`, et le deuxième est simplement 2^p, qu'on peut calculer grâce à `power2`.
On peut alors utiliser `compare` et `pred` pour itérer autant de fois que nécessaire.

```python
def print2(i : IDD):
  if i is zero:
    print(0, end = "")
  elif i is one:
    print(1, end = "")
  else:
```

```
7       print2(i.hi)
8       total_length = power2(i.p)
9       right_length = binary_length(i.lo)
10      while compare(total_length, right_length):
11        total_length = pred(total_length)
12        print(0, end = "")
13      print2(i.lo)
14
15  class And:
16    memo_cp = {}
17    memo_and = {}
18
19    def compute(s : IDD, t : IDD) -> IDD:
20      if (s,t) in And.memo_and:
21        return And.memo_and[(s,t)]
22      sol = zero
23      if s is t or s is zero:
24        sol = s
25      elif And.compare(s, t) > 0:
26        sol = And.compute(t, s)
27      elif And.compare(t.p, s.p) > 0:
28        sol = And.compute(s, t.lo)
29      else:
30        sol = IDD.create(And.compute(s.hi, t.hi), s.p, And.compute(s.lo, t.lo))
31      And.memo_and[(s,t)] = sol
32      return sol
33
34    def compare(i : IDD, j : IDD) -> int:
35      if (i,j) in And.memo_cp:
36        return And.memo_cp[(i,j)]
37      sol = 0
38      if i is j:
39        sol = 0
40      elif i is zero:
41        sol = -1
42      elif j is zero:
43        sol = 1
44      else:
45        cp = And.compare(i.p, j.p)
46        ch = And.compare(i.hi, j.hi)
47        if cp != 0:
48          sol = cp
49        elif ch != 0:
50          sol = ch
```

```
51        else:
52          sol = And.compare(i.lo, j.lo)
53      And.memo_cp[(i, j)] = sol
54      return sol
```

Partie V. Opérations logiques et applications

Question 23

La seule équation à retravailler pour le `OU` est la 4e équation.

$$
\begin{aligned}
0 \vee t &= t \\
s \vee s &= s \\
s \vee t &= t \vee s \quad \text{si } s > t \\
s \vee t &= \langle t.hi, t.p, s \vee t.lo \rangle \quad \text{si } s.p < t.p \\
s \vee t &= \langle s.hi \vee t.hi, s.p, s.lo \vee t.lo \rangle \quad \text{sinon}
\end{aligned}
$$

de même,

$$
\begin{aligned}
0 \oplus t &= t \\
s \oplus s &= 0 \\
s \oplus t &= t \oplus s \quad \text{si } s > t \\
s \oplus t &= \langle t.hi, t.p, s \oplus t.lo \rangle \quad \text{si } s.p < t.p \\
s \oplus t &= \langle s.hi \oplus t.hi, s.p, s.lo \oplus t.lo \rangle \quad \text{sinon}
\end{aligned}
$$

Question 24

On considère les deux nombres dont l'écriture binaires sont:

- $A_p = \underbrace{10101010...1010}_{2^p \text{ bits}}$
- $B_p = \underbrace{01010101...0101}_{2^p \text{ bits}}$

On définit par exemple A_p par récurrence comme :

- $A_1 = 2$
- $A_{p+1} = \langle A_p, p, A_p \rangle = A_p \times 2^{2^p} + A_p$

Un appel à `log_and(A(p), B(p))` ferait lui même deux appels (d'après la règle (15)) à `log_and(A(p-1), B(p-1))`

On en déduit par récurrence immédiate que la complexité en temps de ce calcul est exponentiel en p. Pour conclure, il faut alors montrer que $s(A_p)$ est polynomial en p. On peut alors montrer par induction que A_p a au plus p sommets plus les sommets des idds représentant les entiers de 0 à p. Ce qui permet de conclure, car le même raisonnement tient pour B_p.

Question 25

Il suffit d'appliquer une mémoïsation en suivant les règles données par l'énoncé :

```python
def log_and(s : IDD, t : IDD) -> IDD:
  return And.compute(s, t)
```

Lorsqu'on fait un appel à `log_and(s,t)`, les appels récursifs auront toujours des arguments qui sont des sommets des IDDs `s` et `t`. On en déduit qu'il y a $s(t) \times s(s)$ paire d'arguments appelées. En dehors

des appels récursifs, un appel à `log_and` est en $O(s(t) + s(s))$, à cause des appels à `compare`. Comme les appels sont mémoïsés, il suffit de compter combien de fois chaque paire est l'argument d'un appel à `log_and`. Comme chaque appel en génère au plus 2, et qu'il y a $s(t) \times s(s)$ paires d'IDDs, on sait qu'il y aura au plus $2 \cdot s(t) \cdot s(s)$ appels à `log_and`.

On a finalement une complexité de $O(s(t) \cdot s(s) \cdot (s(t) + s(s)))$, et ce à cause des appels à `compare`. De plus, on utilise un dictionnaire en variable globale, ce qui n'est pas très propre. On peut donc créer une classe qui sert à calculer `log_and` et `compare` en mémoïsant les résultats.

On commence par créer notre classe

```python
class And:
  memo_cp = {}
  memo_and = {}
```

Puis on crée deux méthodes calculant `compare` et `log_and` mais en utilisant nos deux dictionnaires.

```python
    def compute(s : IDD, t : IDD) -> IDD:
      if (s,t) in And.memo_and:
        return And.memo_and[(s,t)]
      sol = zero
      if s is t or s is zero:
        sol = s
      elif And.compare(s, t) > 0:
        sol = And.compute(t, s)
      elif And.compare(t.p, s.p) > 0:
        sol = And.compute(s, t.lo)
      else:
        sol = IDD.create(And.compute(s.hi, t.hi), s.p, And.compute(s.lo, t.lo))
      And.memo_and[(s,t)] = sol
      return sol
```

```python
    def compare(i : IDD, j : IDD) -> int:
      if (i,j) in And.memo_cp:
        return And.memo_cp[(i,j)]
      sol = 0
      if i is j:
        sol = 0
      elif i is zero:
        sol = -1
      elif j is zero:
        sol = 1
      else:
        cp = And.compare(i.p, j.p)
        ch = And.compare(i.hi, j.hi)
        if cp != 0:
          sol = cp
```

```
16        elif ch != 0:
17          sol = ch
18        else:
19          sol = And.compare(i.lo, j.lo)
20      And.memo_cp[(i, j)] = sol
21      return sol
```

Et enfin, il ne nous reste qu'à définir `log_and`.

```python
def log_and(s : IDD, t : IDD) -> IDD:
  return And.compute(s, t)
```

On montre avec un raisonnement similaire à ce qui précède, que la complexité totale est en $O(s(t) \cdot s(s))$.

Question 26

Il suffit de remarquer que $s - t = s\Delta(s \wedge t)$, où Δ est l'opérateur de différence symétrique, ce qui correspond bien à la fonction `log_xor`

```python
def difference(s : IDD, t : IDD) -> IDD:
  return log_xor(s, log_and(s, t))
```

Question 27

On peut calculer l'intersection t de $\{n\}$ avec l'IDD s représentant l'ensemble, on a alors $t =$ `zero` si et seulement si $n \in s$.

```python
def mem(n : int, s : IDD) -> bool :
  t = log_and(s, power2(of_int(n)))
  return not(t is zero)
```

Question 28

On a $s \subseteq t \iff s \vee t = t$

```python
def subset(s : IDD, t : IDD) -> bool :
  return log_or(s, t) is t
```

Question 29

Soit n un IDD, sa représentation en mémoire est l'ensemble des sommets contenu dans l'IDD. En effet, grâce au dictionnaire utilisé dans la classe IDD, chaque sommet n'est stocké qu'au plus une fois. Chacun de ces sommets contient juste trois structures (qui sont des adresses, de taille constante donc). La taille totale de cette structure est donc linéaire en le nombre de sommets de l'IDD, c'est à dire en $O(s(n))$

Question 30

On utilise une taille constante pour chaque sommet (plus précisément 3 adresses). Le nombre de sommets $s(n)$ est majoré par $\mathbf{pop}(n) \times (n.p + 1)$. Or comme il y a K entiers dans l'ensemble S considéré, on a $\mathbf{pop}(n) = K$. De même, comme chaque entier est codé sur W bits au plus, ils sont

inférieurs à 2^W. Si n représente l'ensemble S, alors $n \leq \sum_{i=0}^{2^W-1} 2^i \leq 2^{2^W}$. Ainsi on peut en déduire que l'IDD n vérifie $n.p \leq W$.

Finalement, la mémoire utilisée est en $O(K \times W)$

Si on utilise une liste d'entiers pour représenter un ensemble de K entiers de W bits, alors l'espace nécessaire est en $O(K \times W)$.

Question 31

Montrons par récurrence sur n que les ensembles $2, 3, ..., n$ utilisent une taille mémoire inférieure à $C \times n$ où C est une constante correspondant à la mémoire utilisée par un objet de la classe `IDD`.

- Initialisation : Pour $n = 2$, il suffit d'un espace C, donc l'inégalité est vérifiée.
- Hérédité : Supposons qu'on ait les ensembles $2, 3, ..., n$ avec un espacé mémoire $C \times n$. Alors il suffit de rajouter l'IDD $n + 1$ qui est composé de trois pointeurs vers des IDDs déjà existants, on ne rajoute alors que la mémoire utilisée par ce sommet. La taille utilisée pour représenter les ensembles $2, 3, ..., n + 1$ est donc inférieure à $C \times (n + 1)$

Finalement, l'espace mémoire utilisé pour représenter $2, 3, ..., n$ est en $O(n)$.

Question 32

Soit $S' = \mathcal{P}(\{0, 1, ..., K - 1\})$.

Si on représente les éléments de S' par des IDDs, alors il s'agit des IDDs $0, 1, ..., 2^K - 1$. L'espace mémoire utilisé est alors, d'après la **Question V-31**, en $O(2^K)$.

Si on représente les éléments de S' par des listes, alors il faut une liste par sous-ensemble, dont la taille est celle du sous-ensemble. L'espace utilisé est donc, pour chaque taille $i \in [\![0, K]\!]$ possible de sous-ensemble, le nombre de tels sous-ensembles possibles ($\binom{K}{i}$) multiplié par la taille de ces sous-ensembles ($i \times W$). On a de plus $W = O(\log(K))$

$$\begin{aligned}\sum_{i=0}^{K} \binom{K}{i} \cdot i \cdot W &\leq W \cdot K \cdot \sum_{i=0}^{K} \binom{K}{i} \\ &= W \cdot K \cdot 2^{K+1}\end{aligned}$$

L'espace mémoire utilisé est donc en $O(\log(K) K 2^K)$

On peut en conclure que bien que les listes soient plus compactes pour la représentation d'un ensemble, le partage de la mémoire les rend plus efficaces pour le stockage d'un grand nombre d'ensembles.

Partie VI. Pour aller plus loin

Question 33

Les opérations sur les entiers sur W bits étant très rapides, on peut imaginer une structure différente pour les IDD comme suit:

Tout entier naturel s'écrit de manière unique comme:

- cas 1 : comme un entier dans l'intervalle $[\![0, 2^W - 1]\!]$
- cas 2 : comme $h \times 2^{2^p} + l$ avec $0 < h < 2^{2^p}, 0 \leq l < 2^{2^p}, p \geq \log(W)$

Pour cela, on pourrait rajouter un champ `small` à la classe `IDD`. Lorsque celui-ci est compris entre 0 et $2^W - 1$, il est la valeur que représente l'IDD. Une solution plus propre serait d'utiliser un langage avec un typage plus fort comme ocaml.

Une telle implémentation, qui n'est pas plus compliquée que ce qui a été fait précédemment, permet d'économiser un facteur constant sur la complexité de toutes les opérations.

Il suffit de redéfinir les cas de bases de chaque fonctions. Par exemple, pour `log_and`, si les deux IDDs sont dans le cas 1, on peut utiliser le `ET` logique bit à bit fourni par l'architecture.

Question 34

On peut, pour représenter un entier relatif, utiliser la définition initiale d'IDD pour représenter tous les entiers naturels, puis représenter l'entier négatif $-n$ par le sommet représentant n auquel on ajoute un bit de signe.

On peut alors adapter les définitions de la Partie 4 : Arithmétique, qui sont les seules touchées par ce changement. Par exemple, la fonction `pred`, dans le cas où son argument est négatif, disons `-n`, devrait renvoyer `-(succ(n))`. On devrait adapter de même chaque fonction.

Il faut aussi traiter du cas de `zero`, qui est sémantiquement équivalent à `-zero`. On peut simplement remarquer que les sommets sont non signés, et donc qu'aucun sommet ne pointera jamais vers `-zero`.