XU Marc 28705948
Huynh Yok Yann 28707630

Compte rendu Projet d'Ouverture

Partie 1 :

Notre représentation des grands entiers traités dans ce projet est :

```
module BigInt =
struct
type integer = Int64.t list (* liste d'entiers int64 *)

let get_tete (bigint : integer) : Int64.t = List.hd bigint

let inserer_queue (nelem : Int64.t) (bigint : integer) : integer =
bigint @ [nelem]

let supprimer_tete (bigint : integer) : integer = List.tl bigint
end
```

On prend une liste d'entiers Int64 qui sont des entiers codés sur 64 bits exactement. Les primitives get_tete, inserer_queue, supprimer_tete sont faciles à implémenter puisqu'on manipule une simple liste.

```
decomposition (x : BigInt.integer) : bool list
```

Cette fonction consiste à faire la représentation binaire d'un grand entier.
On passe par une fonction auxiliaire qui va construire notre liste de booléen en parcourant notre liste (x, qui représente notre grand entier)
Si on tombe sur un 0L (0 mais en Int64) alors on doit rajouter 64 bits à false (puisque le bit de poids faible est à gauche)
Sinon, on doit faire la décomposition binaire du nombre avec la fonction :

```
rec convert_to_binary (n : Int64.t) (bits : bool list) : bool list
```

```
completion (l : bool list) (n : int) : bool list
```

Cette fonction permet de tronquer ou de remplir notre liste de booléen.
Si n est inférieur à la taille de l, alors on tronque pour avoir les n premiers booléen.
Sinon on rajoute la différence avec des booléen false.

```
composition (l : bool list) : BigInt.integer
```

Cette fonction fait l'opposé de decomposition.
On a deux cas :

- une liste avec juste 1 false qui représente notre 0L, on avait mis une liste vide à la base mais on s'est rendu compte qu'après pendant l'implémentation de nos fonctions

pour définir les graphes .dot, cela créait des problèmes avec les ID avec notre algorithme de compression

- une liste de booléen qui représente notre entier, on va parcourir la liste par groupe de 64 booléens et chaque groupe représente un entier Int64 qu'on ajoute à notre liste qui représente notre grand entier.

```
table (x : int) (n : int) : bool list
```

Cette fonction est un simple appel successif de completion n sur la decomposition d'un entier x.

```
genAlea (n : int) : BigInt.integer
```

Génère un grand nombre aléatoire.

Partie 2 :

Notre structure de donnée est la suivante :

```
type decision_tree =
| Leaf of bool
| Node of int * decision_tree * decision_tree
```

Un arbre de décision est soit une feuille étiquetée par un booléen, soit un nœud composé d'un entier et deux deux autres arbres de décisions.

```
cons_arbre (tab : bool list) : decision_tree
```

Construit un arbre de décision à partir d'une table de vérité tab.

```
liste_feuilles (noeud : decision_tree) : bool list
```

À partir d'un arbre de décision, donne sa table de vérité sous une liste de booléen

Partie 3 :

Notre structure de données est la suivante :

```
type listeDejaVus = (BigInt.integer * decision_tree ) list
```

Elle consiste à une liste de couples de grands entiers et d'arbres de décisions qui représente un nœud d'un graphe.

```
compressionParListe (dt : decision_tree) (ldv : listeDejaVus) :
decision_tree * listeDejaVus
```

On calcule le nombre de feuilles dans le nœud dt, et si on l'a déjà traité (càd qu'on le trouve dans ldv) alors on renvoie le couple de l'arbre de décision correspondant. Sinon on continue la compression :
On regarde l'arbre dt et si c'est une feuille, alors on le renvoie tel quel.
Sinon c'est un nœud et on compresse les fils, et on donne au fils droit la listedejavus résultante de la compression du fils gauche. Pour compresser ces fils, on applique la règle-z, Si le fils droit est une feuille à false, alors on compresse le noeud avec le fils gauche, c'est-à-dire qu'on redirige toutes les arêtes vers N vers le fils gauche.

```
dot (tree : decision_tree) (ldv : listeDejaVus) (filename: string)
```

Notre fonction dot se divise

A FINIR !