UE d'Ouverture **Devoir de Programmation**

Par FANG Julien et LIM Floria

3 Présentation

Question 1.1

Voici la structure de données et les primitives nécessaires à notre liste d'entiers 64 bits :

```
module ListeInt64 =

struct

  type t = int64 list (*liste de Int64*)

  let vide() = [] (*liste vide*)

  (*ajoute x à la fin de la liste*)
  let rec insertQueue x l =
    match l with
    | [] -> [x]
    | h::t -> h::(insertQueue x t)

  (*ajoute x au début de la liste*)
  let insertTete x l = x::l

  (*supprime le premier élément de la liste*)
  let removeTete l =
    match l with
    | [] -> []
    | h::t -> t

  (*retourne le premier élément de la liste*)
  let getTete l =
    match l with
    | [] -> []
    | h::t -> h

end ;;
```

Question 1.2

Afin d'obtenir la décomposition d'un grand entier, nous avons tout d'abord implémenté une fonction auxiliaire nommée decompositionUnEntier qui prend un entier x strictement inférieur à 2^{64} et nous renvoie la liste de booléens représentant sa décomposition en base 2.
Afin de parcourir les bits d'un entier x, nous récupérons le bit de poids le plus faible à l'aide d'un AND logique, et vérifions s'il est égal à 0 ou à 1. Nous faisons ensuite un décalage à droite de 1 bit sur l'entier x afin de retirer le bit de poids le plus faible et parcourir les bits restants.

```
if (Int64.logand x 1L = 0L) then (*on recupère le bit de poids faible*)
  false :: (decompositionUnEntier (Int64.shift_right_logical x 1)) (*on décale à droite de 1 bit*)
else
  true :: (decompositionUnEntier (Int64.shift_right_logical x 1)) (*on décale à droite de 1 bit*)
```

Comme les Int64 sont des entiers signés, il nous a fallu traiter le cas où nous avons un entier négatif. Pour cela, nous avons tout d'abord vérifié si l'entier x était inférieur à 0. Si c'est le cas, alors nous savons que le 64ème bit de x est à 1, et donc nous ajoutons un true à la fin de la décomposition obtenue lors du parcours en ayant complété au préalable cette dernière sur 63 bits grâce à la fonction completion de la Q1.3.

```
if (signe < 0) then (completion (aux x) 63)@[true] (*si x est négatif, alors il s'agit d'un entier sur 64 bits*)
else (aux x)
```

Ainsi, en appelant cette fonction sur chacun des éléments de la liste du grand entier, nous obtenons la décomposition demandée. Nous ré-utilisons également la fonction completion pour obtenir une liste de booléens sur 64 bits pour les entiers de la liste du grand entier, excepté le dernier. Cela est nécessaire pour délimiter correctement chaque élément du grand entier dans la liste de booléens résultante.

Question 1.3

Pour la fonction completion, il suffit de parcourir la liste en décrémentant n jusqu'à qu'il soit égal à 0. Si on arrive à la fin de la liste et que n n'a pas encore atteint 0, on ajoute n false en début de liste.

Question 1.4

La fonction composition prend une liste de booléens et renvoie le grand entier correspondant.
Nous passons par une fonction auxiliaire qui aura en paramètre une liste de booléen (l), une variable désignant un élément de la liste du grand entier (tmp), ainsi que la liste résultante (res).
Nous disposons également d'un compteur qui désigne la position du bit auquel on se trouve (c), ainsi qu'un second compteur égal à la valeur de ce dernier (cpt). Ainsi, cpt vaut 2^c si le bit à la position c est à 1.

```
(*compose une liste de booléens en un grand entier*)
let composition l =
  let rec aux l cpt tmp c res =
    match l with
    | [] -> if (tmp <> 0L) then (ListeInt64.insertQueue tmp res) else res
    | h::t ->
      if (c=64) then (aux l 1L 0L 0 (ListeInt64.insertQueue tmp res))
      else
        if ((c=63) && h) then let tmp_neg=(Int64.neg cpt) in (aux t (Int64.mul cpt 2L) (Int64.add tmp tmp_neg) (c+1) res)
        else if h then (aux t (Int64.mul cpt 2L) (Int64.add cpt tmp) (c+1) res)
        else (aux t (Int64.mul cpt 2L) tmp (c+1) res)
  in aux l 1L 0L 0 (ListeInt64.vide())
```

Comme nous savons que chaque entier de la liste se limite à 2^{64}, nous parcourons la liste de booléens et nous arrêtons à chaque fois que c=64. A ce moment-là, on ajoute l'entier tmp à la queue de la liste du grand entier puis nous réinitialisons tmp ainsi que les compteurs.
Aussi, nous vérifions si nous tombons sur un true lorsque c=63. Si c'est le cas, alors cpt est un entier négatif, et nous le convertissons alors avec Int64.neg avant de continuer.

Question 1.5

La fonction table est la composition de nos fonctions completion et decompositionUnEntier.

Question 1.6

Afin d'implémenter la fonction genAlea, nous avons tout d'abord codé une fonction auxiliaire genererAleatoire. Cette fonction génère tout d'abord un entier aléatoire sur 63 bits.

```
let random = (Random.int64 Int64.max_int) in
```

Afin de générer un entier sur 64 bits, nous appliquons avec une probabilité de 50% la négation sur l'entier obtenu. Ensuite, afin d'obtenir un entier sur exactement n bits au maximum, on réalise un décalage à droite de (64-n) bits.

```
if Random.bool () then
  Int64.shift_right_logical random (64-n)
else
  Int64.shift_right_logical (Int64.neg random) (64-n)
```

Ainsi pour genAlea, nous calculons tout d'abord l = n/64 qui est égal au nombre d'éléments à 64 bits qu'aura la liste du grand entier. Nous générons ainsi l entiers aléatoire sur 64 bits au maximum, puis un dernier entier sur n - l*64 bits au maximum.

```
(*génère un grand entier aleatoire sur n bits au maximum*)
let genAlea n =
  if (n<=64) then [genererAleatoire n]
  else
    let l = n/64 in (*nombre d'éléments sur 64 bits*)
    let m = n - l*64 in (*le reste est sur m bits*)
    let rec aux i =
      if (i=0) then []
      else (genererAleatoire(64))::(aux (i-1))
    in
  let reste = (genererAleatoire m) in
  if (reste <> 0L) then (aux l)@[reste] (*si le reste est à 0, on ne l'ajoute pas*)
  else (aux l)
```

2 Arbre de décision

Question 2.7

Voici notre structure de données permettant d'encoder des arbres binaires de décision :

```
(*structure de donnée permettant d'encoder des arbres binaires de décision*)
type arbre_ref =
| Feuille of bool (*feuille*)
| Noeud of int * arbre_ref ref * arbre_ref ref (*Noeud (profondeur, fils gauche, fils droit)*)
```

Notre structure utilise des références car il sera nécessaire de modifier les pointeurs des nœuds pour la suite du projet. Chaque nœud contient une étiquette désignant sa profondeur.

Question 2.8

Afin de réaliser la fonction cons_arbre, nous avons tout d'abord implémenter 2 fonctions auxiliaires :

- (split l) permettant de couper une liste l en deux
- (puissanceSup n) calculant la plus petite puissance de 2 supérieure à n

Notre fonction cons_arbre split récursivement la liste de booléens donnée en argument, puis les rassemble deux à deux pour former un nœud de l'arbre.
Afin d'obtenir un arbre parfait, nous complétons la liste de booléens au préalable afin que sa longueur (équivalente au nombre de feuilles de l'arbre) soit égale à une puissance de 2. En effet, nous souhaitons que toutes les feuilles soient à la même profondeur, et que chaque nœud de l'arbre possède deux feuilles.

```
(*construit un arbre binaire de décision à partir d'une liste de booléens*)
let cons_arbre l =
  let length = List.length l in
  let rec aux l n =
    match l with
    | [] -> ref (Feuille false) (*arbre vide*)
    | h :: [] -> ref (Feuille h) (*feuille*)
    | h :: t ->
        let (l1, l2) = split l in (*on sépare la liste en deux*)
        ref (Noeud(n, aux l1 (n + 1), aux l2 (n + 1))) (*crée un noeud avec les deux listes en faisant un appel récursif*)
  in aux (completion l (puissanceSup length)) 1 (*on complète la liste l pour que sa longueur soit une puissance de 2*)
```

Question 2.9

Pour construire la liste des étiquettes des feuilles du sous-arbre enraciné en N, il suffit de faire un parcours préfixe du sous-arbre et de concaténer ses feuilles.

3 Compression de l'arbre de décision et ZDD

Question 3.10

Voici notre structure de données ListeDejaVus :

```
module ListeDejaVus =

struct
  type t = (int64 list * arbre_ref) list ref (*liste de couple (grand entier, pointeur vers un noeud)*)

  let vide () = ref []

  (*ajoute x à la tête de la liste*)
  let insertTete x l =
    l := x :: !l

  (*recherche un grand entier dans la liste*)
  let rec recherche x l =
    match !l with
    | [] -> None
    | (n, a) :: t ->
      if n = x then Some (n, a)  (*si on trouve le grand entier, on renvoie le couple*)
      else recherche x (ref t)  (*sinon on continue la recherche*)

end
```

Chaque élément de la liste est un couple de (grand entier, pointeur vers un nœud). Nous avons défini nos éléments comme des références afin qu'on puisse mettre à jour en temps réel la liste à n'importe quel moment. Nous avons implémenté dans la structure une fonction pour insérer un élément (un couple), ainsi qu'une fonction pour rechercher un grand entier dans la liste.

Question 3.11

Lors de l'implémentation, afin de pouvoir remplacer le pointeur vers N depuis son parent, il nous a été nécessaire de mémoriser le parent du nœud lors du parcours, ainsi que s'il s'agissait de l'enfant de gauche ou de droite.

Afin de vérifier si la deuxième moitié de la liste ne contenait que des valeurs false, nous avons fait appel à notre fonction split définie précédemment à la Q2.8 pour récupérer la deuxième moitié de la liste.

Aussi, nous avons traité les feuilles comme des nœuds. En effet, nous ne gardons qu'une seule feuille True et qu'une seule feuille False après la compression. Tous les nœuds ayant pour enfant des feuilles pointeront vers ces deux derniers. Notre ListeDejaVus contiendra donc les couples ([0], pointeur vers la feuille False) et ([1], pointeur vers la feuille True).

```
let n = (composition liste_bool) in (*on calcule le grand entier associé au noeud*)
let couple = (ListeDejaVus.recherche n l) in (*on cherche si le grand entier est présent dans la ListeDejaVus*)
(match couple with
  | None -> (*le grand entier n'est pas dans la ListeDejaVus, on l'ajoute*)
    (ListeDejaVus.insertTete (n, abr) l);
    (aux g l abr true);
    (aux d l abr false);
  | Some (_, a) -> (*le grand entier est dans la ListeDejaVus, on change le pointeur vers le noeud depuis le parent*)
    match !parent with
    | Noeud (pp, gg, dd) ->
      if depuisGauche then parent := Noeud(pp, a, dd)
      else parent := Noeud(pp, gg, a);
    | _ -> ())
```

Question 3.12

Nous avons tout d'abord implémenté les fonctions dot_noeud et dot_link, l'un créant les nœuds du graphe et l'autre les arêtes les reliant. Chaque nœud du graphe possède un identifiant unique permettant de les différencier. L'argument isTrait de notre fonction dot_link nous indique si l'arête doit être en pointillé ou non.

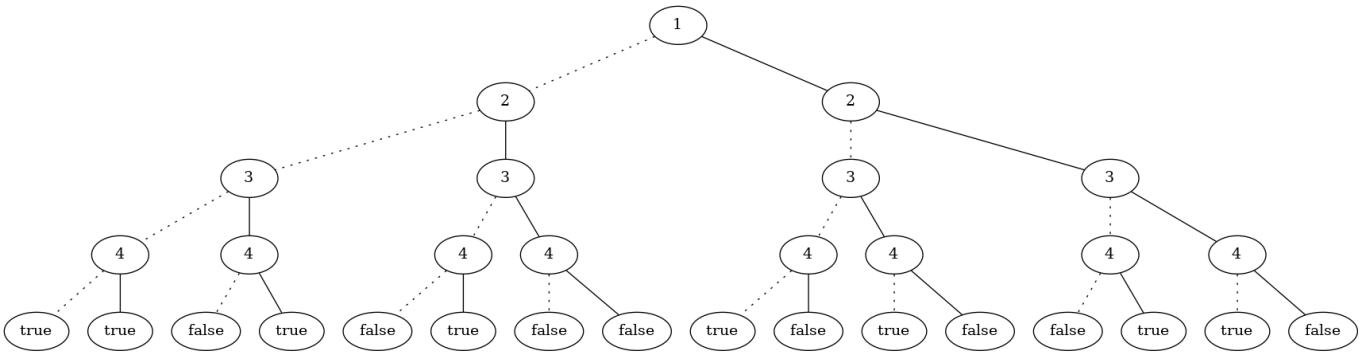
```
(*création d'un noeud*)
let dot_noeud f id label =
  fprintf f "  %d [label=\"%s\"];\n" id label

(*création d'une arête entre deux noeuds*)
let dot_link f id_parent id_enfant isTrait =
  let style = if isTrait then "solid" else "dotted" in
  fprintf f "  %d -- %d [style=\"%s\"];\n" id_parent id_enfant style
```

Lors du parcours de notre arbre, nous disposons d'un compteur afin d'obtenir un identifiant différent pour chaque nœud de notre arbre. Ce compteur est une référence vers un entier, car nous voulons que la valeur de ce dernier soit à jour à n'importe quel moment durant notre parcours.
Aussi, comme un même nœud peut être pointé par plusieurs nœuds différents, il nous a été nécessaire de mémoriser les nœuds que l'on a déjà parcouru pour éviter les doublons d'un même nœud dans notre graphe. Pour cela, nous avons utilisé une liste dans laquelle nous avons stocké les pointeurs des nœuds déjà parcourus ainsi que l'identifiant qui leur correspond. Les éléments de notre liste sont donc des couples de (pointeur vers un nœud, identifiant).
Si on tombe sur un nœud déjà présent dans la liste, cela signifie que le nœud et ses fils ont déjà été créés et nous n'avons rien à faire. Sinon, on crée le nœud grâce à dot_noeud puis nous parcourrons récursivement les fils gauche et droit. Les arêtes sont ajoutées à la fin, lorsque tous les nœuds ont été créés.

```
| Noeud (profondeur, gauche, droite) ->
  if not (List.exists (fun (_, a) -> a == abr) !parcourus) then (*le noeud n'a pas encore été créé*)
    begin
      dot_noeud f !cpt (string_of_int profondeur); (*on créé le noeud*)
      parcourus := (!cpt, abr)::(!parcourus); (*on ajoute le noeud à la liste des noeuds déjà parcourus*)
      let id = !cpt in (*on retient l'identifiant du noeud pour la création des arêtes*)
      cpt := !cpt + 1;
      aux gauche parcourus cpt; (*on crée tous les noeuds du sous-arbres gauche*)
      aux droite parcourus cpt; (*on crée tous les noeuds du sous-arbres droite*)
      let id_gauche = (*on recherche l'identifiant de l'enfant gauche*)
        match (List.find_opt (fun (_, a) -> a == gauche) !parcourus) with
      | Some (c, _) -> c
      | None -> -1
      in
      let id_droite = (*on recherche l'identifiant de l'enfant droit*)
        match (List.find_opt (fun (_, a) -> a == droite) !parcourus) with
        | Some (c, _) -> c
        | None -> -1
      in (*creation des arêtes*)
      dot_link f id id_gauche false;
      dot_link f id id_droite true;
    end
```

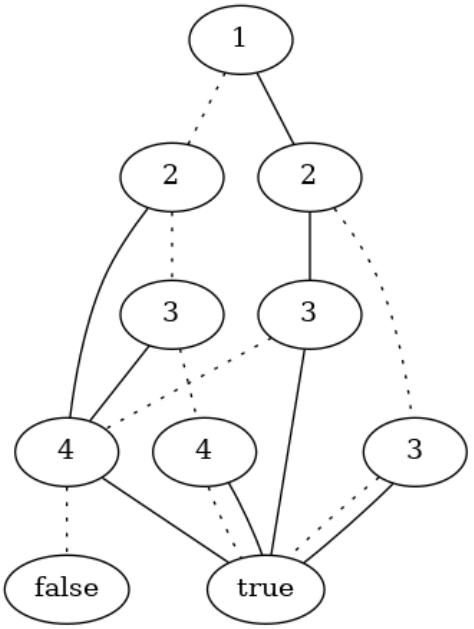
De plus, afin que notre fonction dot puisse fonctionner à la fois sur un arbre compressé et non compressé, il a fallu que l'on compare les références des nœuds avec l'opérateur '=='. En effet, avec seulement l'opérateur '=', deux nœuds différents ayant les mêmes fils étaient considérés comme identiques, et cela nous a posé problème pour les arbres non compressés.

Question 3.13



Graphe dot construit sur le grand entier [25899]

Question 3.14



Graphe dot construit sur le grand entier [25899] après compression

4 Compression avec historique stocké dans une structure arborescente

Question 4.15

Chaque nœud de notre ArbreDejaVus contient une étiquette qui est un pointeur vers un nœud de l'arbre ZDD.

```
type arbreDV =
| Noeud of arbre_ref ref option * arbreDV ref * arbreDV ref (*Noeud (pointeur vers un noeud, fils gauche, fils droit)*)
| Feuille (* Feuille *)
```

Nous avons implémenté dans la structure les fonctions d'insertion et de recherche d'un pointeur. Ces deux fonctions parcourent une liste de booléens et l'arbre de recherche simultanément. Lorsqu'on tombe sur la valeur true dans la liste, alors on parcourt le fils droit de l'arbre de recherche, sinon on parcourt le fils gauche.
Lors de l'insertion d'un pointeur, si on tombe sur une feuille, alors on crée un nouveau nœud d'étiquette vide, et on continue le parcours. Une fois la liste de booléens parcourue, nous ajoutons le pointeur dans l'étiquette du nœud auquel on se trouve dans l'arbre de recherche.

Question 4.16

Voici l'algorithme élémentaire de compression que nous avons adapté pour utiliser l'arbre de recherche ArbreDejaVus :
— Soit G l'arbre de décision qui sera compressé petit à petit. Soit un arbre de recherche ArbreDejaVus vide.
— En parcourant G via un parcours suffixe, étant donné N le nœud en cours de visite :
- — Calculer la liste_feuilles l associées à N (le nombre d'éléments qu'elle contient est une puissance de 2).
- — Si la deuxième moitié de la liste ne contient que des valeurs false alors remplacer le pointeur vers N (depuis son parent) vers un pointeur vers l'enfant gauche de N
- — Si la recherche du parcours l renvoie un pointeur, alors remplacer le pointeur vers N (depuis son parent) par un pointeur vers le pointeur trouvé ;
- — Sinon ajouter à ArbreDejaVus un pointeur vers N avec le parcours l.

Question 4.17

Nous avons adapté le code de la compression avec une ListeDejaVus pour un ArbreDejaVus. Le code est similaire à ce dernier, mais nous n'avons pas besoin de calculer le grand entier associé au nœud parcouru. Seule la liste_feuilles est nécessaire.

Question 4.18

Nous obtenons le même graphe à partir du fichier dot construit sur le grand entier [25899] avec un ArbreDejaVus.

5 Analyse de complexité

Question 5.19

— Complexité temporelle de compressionParListe :
Soit n le nombre de nœuds.
On fait un parcours suffixe pour chaque noeud de l'arbre, et à chaque noeud, on peut appeler les fonctions suivantes une fois :

- liste_feuilles : en O(m), avec m le nombre de feuilles du nœud. Donc $m = 2^{log(n+1)-1}$.
- composition : en O(m) car on parcourt toute la liste_feuilles
- insertTete : en O(1)
- split : en O(m) car on coupe la liste_feuilles
- recherche : dans le pire des cas, on parcourt toute la ListeDejaVus. Donc en O(k), avec k le nombre d'éléments de la liste.

On sait que k <= n car on insère un élément dans la liste lorsqu'on tombe sur un nœud correspondant à un grand entier encore jamais rencontré.
Avec le parcours de chaque nœud et en additionnant l'ensemble de ces fonctions, on obtient au pire cas O(n*(3m+k)) avec $m = 2^{log(n+1)-1}$ et k <= n, donc O(n²).

— Complexité spatiale de compressionParListe :
Soit n le nombre de nœuds et h la hauteur de l'arbre. On a donc $h = \lceil log_2(n) \rceil = log(n+1)$ car l'arbre est parfait.
Dans le pire des cas, l'arbre n'a quasiment pas été modifié et on stocke tous les grands entiers correspondant à chaque nœud. Les feuilles sont dans tous les cas obligatoirement compressées, donc on aura au minimum $2^{h-1} - 2$ nœuds en moins. Au final, le ZDD aura une taille en :
$O(n - 2^{h-1} + 2) = O(n - 2^{log(n+1)-1}) = O(n - \frac{n+1}{2}) = O(\frac{n}{2})$
En supposant que le stockage d'un grand entier est négligeable par rapport à la taille du ZDD, on a au final une complexité spatiale au pire cas en $O(\frac{n}{2})$.

— Complexité temporelle de compressionParArbre :
Soit n le nombre de nœuds.
On fait un parcours suffixe pour chaque noeud de l'arbre, et à chaque noeud, on peut appeler les fonctions suivantes une fois :

- liste_feuilles : en O(m), avec m le nombre de feuilles du nœud. Donc $m = 2^{log(n+1)-1}$.
- recherche : en O(m) car on parcourt toute la liste_feuilles
- inserer : en O(m) car on parcourt toute la liste_feuilles
- split : en O(m) car on coupe la liste_feuilles

Avec le parcours de chaque nœud et en additionnant l'ensemble de ces fonctions, on obtient au pire cas O(n*(4m)) avec $m = 2^{log(n+1)-1}$, donc O(n²).

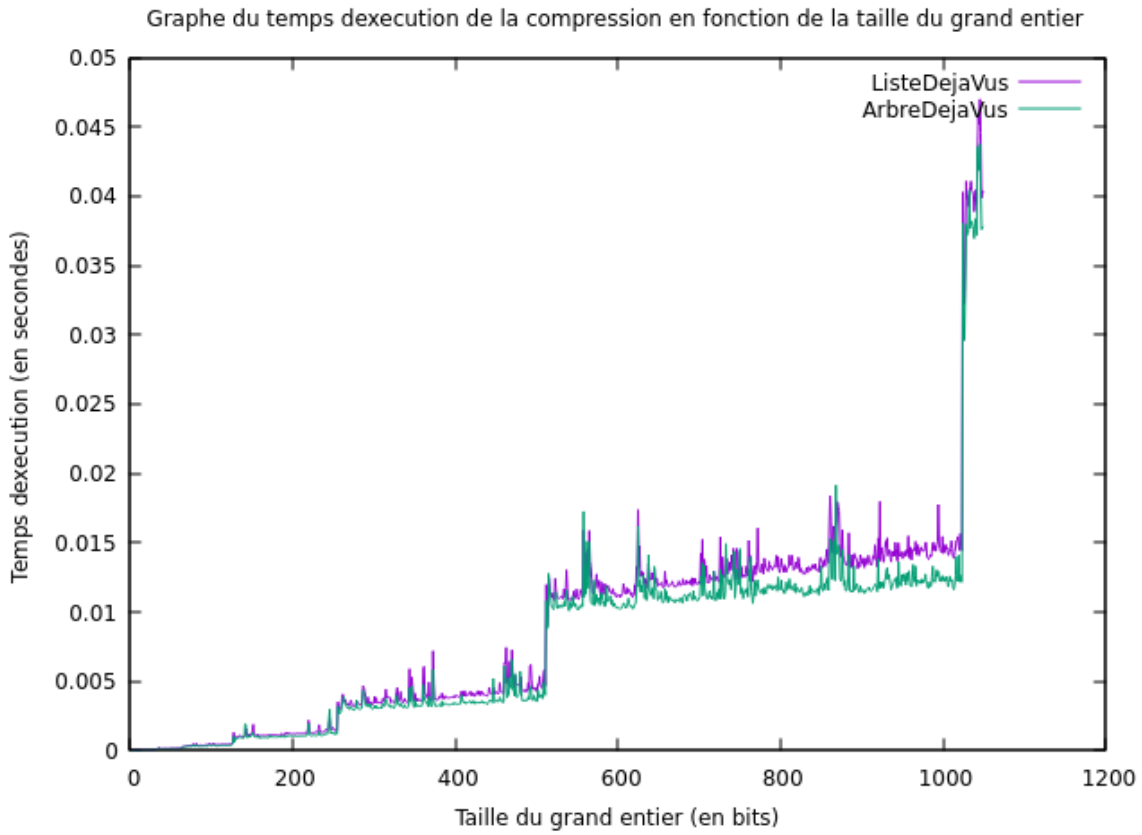
— Complexité spatiale de compressionParArbre :
Dans le pire des cas, la complexité spatiale du ZDD est de $O(\frac{n}{2})$ comme pour compressionParListe, car l'arbre compressé obtenu est le même. En plus de cela, la taille de notre ABR est de l'ordre du nombre de feuilles de l'arbre, c'est-à-dire en $O(2^{log(n+1)-1})$.

Donc pour n le nombre de nœuds, la complexité spatiale au pire cas est en $O(\frac{n}{2} + \frac{n+1}{2})$, c'est-à-dire en $O(\frac{3n+1}{4})$.

6 Etude expérimentale

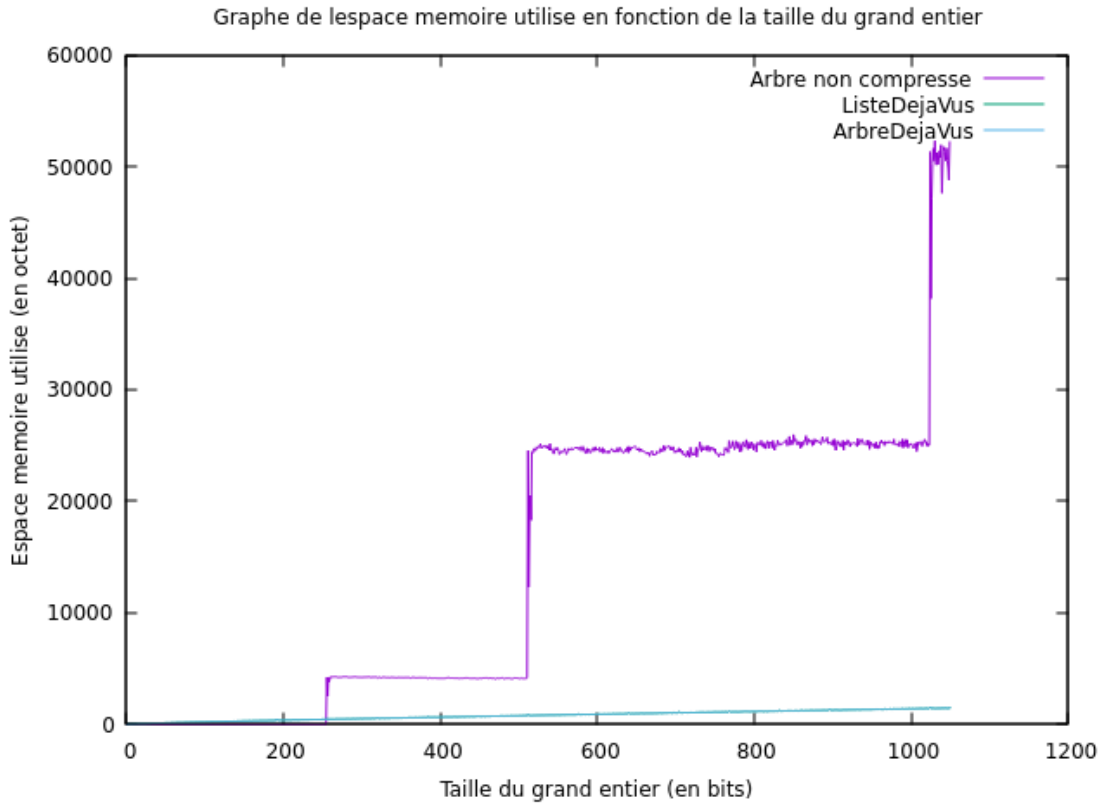
Question 6.20

Les valeurs calculées pour la vitesse d'exécution seront enregistrées dans notre fichier "graphes_temps.txt". Il calcule des grands entiers allant de 1 à 1049 bits puis effectue une compression par ListeDejaVus puis par ArbreDejaVus à 10 reprises afin d'avoir une moyenne. On utilise Sys.time() afin d'obtenir le temps d'exécution pour compressionParListe et compressionParArbre.



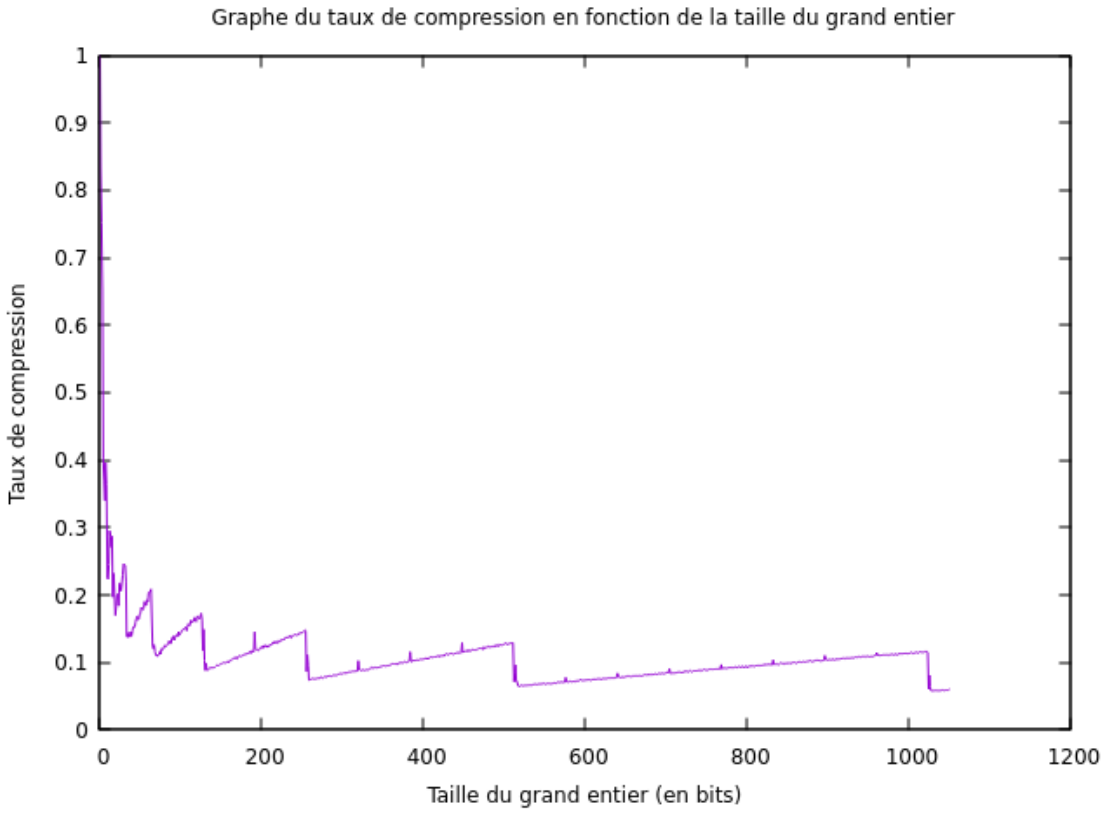
Nous pouvons remarquer que le temps d'exécution augmente dès qu'on passe à la puissance de 2 supérieure. Cela est logique car la hauteur de l'arbre est augmentée de 1 lorsque c'est le cas. Aussi, la compression avec une ListeDejaVus est légèrement plus rapide que celle avec un ArbreDejaVus.

Nous utilisons le module Gc pour calculer l'espace mémoire utilisé. Nous calculons la mémoire à l'aide de Gc.live_word pour obtenir le nombre de mot (octets) alloué dans la mémoire qui sont en cours d'utilisation. Le nombre d'octet est obtenu en soustrayant le nombre d'octets avant la création ou la compression de l'arbre par le nombre d'octets après.
De plus, on utilise Gc.full_major() pour faire appelle au garbage collector afin qu'il effectue une collecte sur les objets non référencés dans la mémoire.
Tout comme pour la vitesse d'exécution, nous travaillons sur des grands entiers allant jusqu'à 1049 bits, tout en effectuant les calculs à 10 reprises pour obtenir une moyenne.



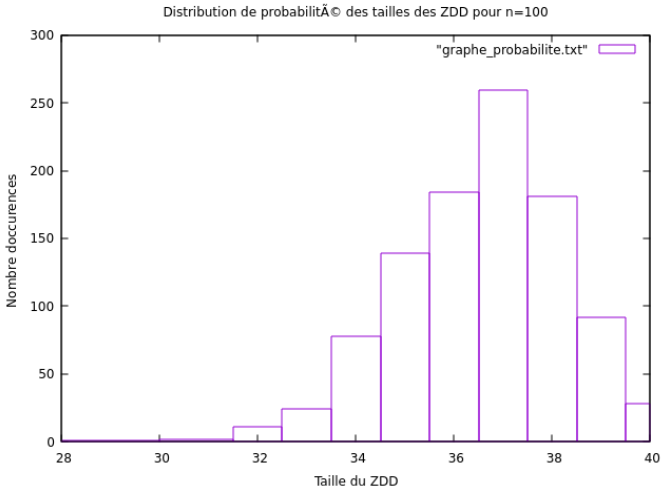
L'espace mémoire utilisé par un arbre non compressé augmente drastiquement dès qu'on passe à la puissance de 2 supérieure, tandis qu'elle est linéaire pour un arbre compressé.

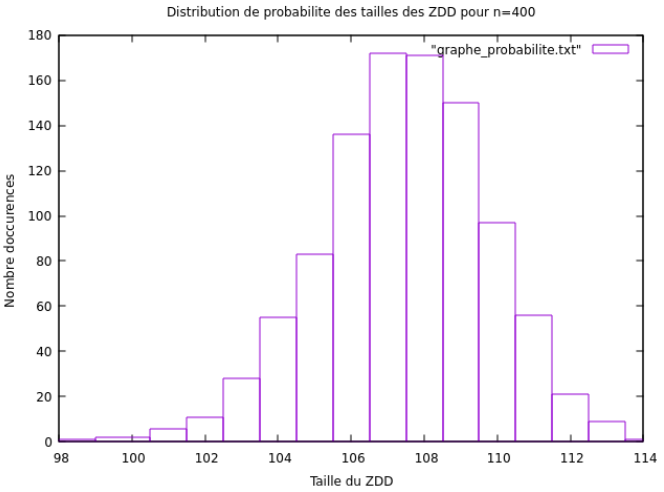
Pour obtenir le taux de compression, nous avons créé une fonction nommée nb_noeud nous permettant d'obtenir le nombre de nœuds d'un arbre de décision. Nous obtenons le taux en divisant le nombre de nœuds après la compression par le nombre de nœuds avant la compression. Nous réalisons simplement une compression par ListeDejaVus car nous obtenions le même arbre compressé dans les deux cas.

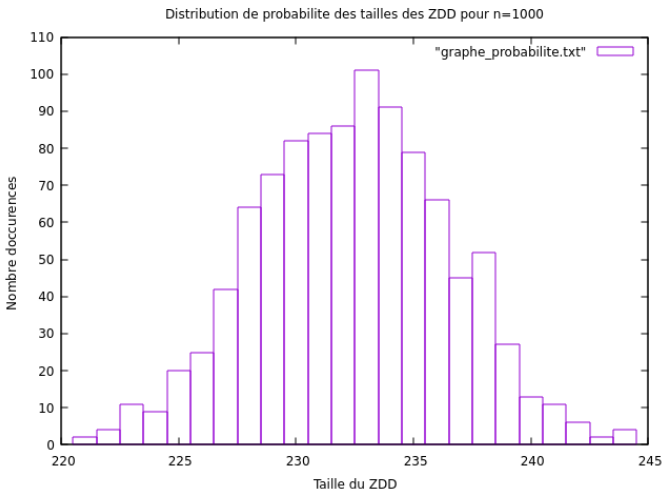


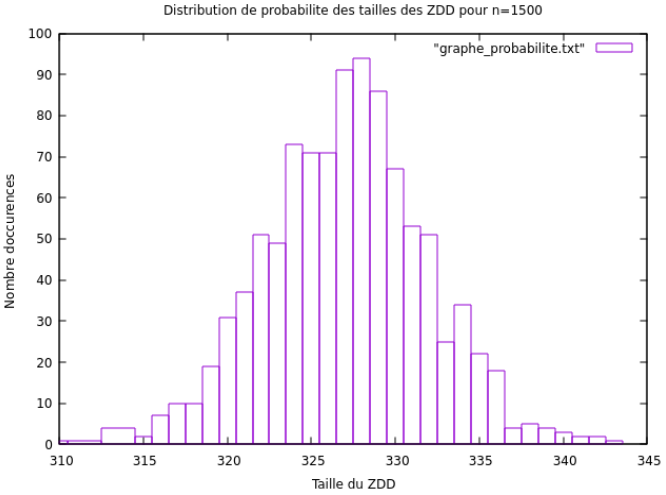
Question 6.21

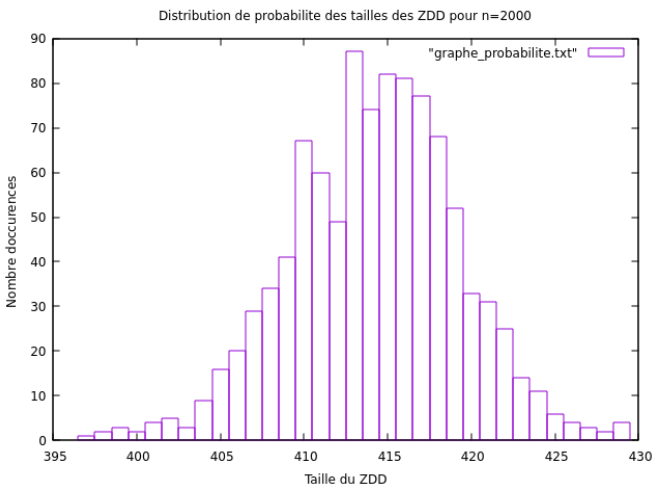
On crée à chaque fois à 1000 reprises un grand entier sur n bits fixé.
En fixant n=100, la taille du ZDD associé est comprise dans l'intervalle [28-40]
En fixant n=400, la taille du ZDD associé est comprise dans l'intervalle [98-114]
En fixant n=1000, la taille du ZDD associé est comprise dans l'intervalle [220-245].
En fixant n=1500, la taille du ZDD associé est comprise dans l'intervalle [310-345].
En fixant n=2000, la taille du ZDD associé est comprise dans l'intervalle [395-430].

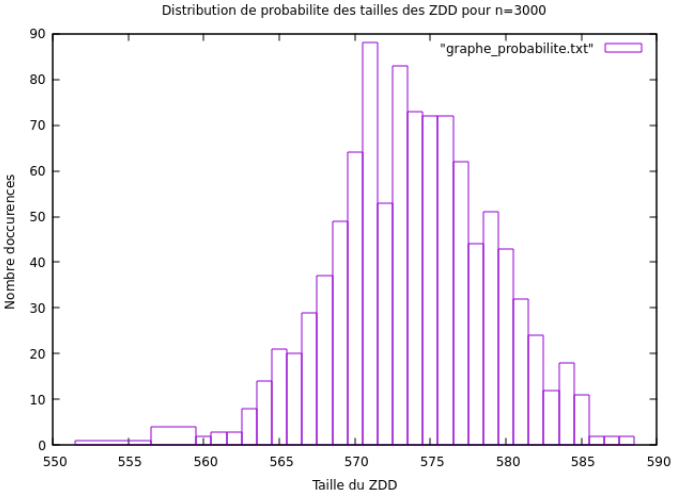












Nous remarquons que la taille du ZDD associée augmente de façon logarithmique par rapport à la taille n. En effet, elle est quasiment égale à la moitié de n pour n=100, tandis qu'elle vaut $\frac{1}{6}$ de la taille pour n=3000. Dans tous les cas, la taille des ZDD est concentrée sur une plage de valeur.