# Rapport du projet d'ouverture

UE OUV | M1 STL | 2020-2021

Hugo Guerrier 3970839 | Emilie Siau 3700323

## Exercice 1

### Q1.1

`extraction_alea (l: 'a list) (p: 'a list) : 'a list * 'a list`

Cette fonction permet de tirer dans la liste `l` un élément aléatoire et de l'insérer à la fin `p` de puis de retourner les deux listes en résultat

Ici nous nous sommes posé la question de si l'on utilisait un :

```
match l with [] -> (l, p) | _ -> ...
```

ou un

```
let len = List.length l in if len = 0 ...
```

… pour s'occuper du cas où l'argument `l` est vide.

Nous avons opté pour le `if` car nous avons besoin de la longueur de `l` à deux autres reprises dans la fonction.

### Q1.2

`gen_permutation (n: int) : int list`

Dans cette fonction nous définissons deux fonctions récursives internes :

- `gen_liste` : Fonction qui permet de générer une liste d'entiers de 1 à n
- `melange` : Cette fonction itère sur une liste et va utiliser la fonction de Q1.1 pour la mélanger

On utilise donc ces deux fonctions pour générer une liste d'entiers entre 1 et n dans un ordre aléatoire.

### Q1.3

La complexité de la fonction `gen_permutation` en nombre d'appels au générateur d'entiers aléatoires est en **O(n)**, n étant l'entier passé en entrée.

On peut justifier ce choix par le fait que le générateur est appelé une fois à chaque passage dans la fonction `extraction_alea`, or la fonction `melange` utilise n fois cette fonction.

Ensuite nous pouvons dire que la complexité de la fonction `gen_permutation` en nombre d'appels au filtrage par motif est en **O(n)**.

Cette réponse vient du fait que la fonction récursive `melange` est la seule à utiliser le *match with*, et elle est appelée une fois sur chaque élément de la liste triée de taille n.

## Q1.4

`intercale (l1: 'a list) (l2: 'a list) : 'a list`

Pour réaliser cette fonction qui permet de mélanger deux listes, nous avons utilisé une fonction interne récursive `intercale_rec (l1: 'a list) (l2: 'a list) (n1: int) (n2: int)` permettant de prendre en argument les tailles des listes pour éviter de les recalculer à chaque appel récursif.
La fonction récursive `intercale_rec` matche donc les listes l1 et l2 :

- Si l1 est la liste vide, on renvoie l2
- Si l2 est la liste vide, on renvoie l1
- Sinon, on découpe l1 en sa tête h1 et sa queue t1, et l2 en sa tête h2 et sa queue t2. On tire ensuite un entier au hasard dans l'intervalle [0, n1+n2[ :
    - Si ce nombre est compris dans [0, n1[ , on construit la liste composée de la tête h1 et la queue est construite en appel récursif avec le reste t1 de l1, avec l2, avec n1 - 1 (car on a réduit l1 donc on diminue sa taille), et avec n2.
    - Si ce nombre est compris dans [n1, n2[ , on construit la liste composée de la tête h1 et la queue est construite en appel récursif avec l1, avec le reste t2 de l2, avec n1, et avec n2 - 1 (car on a réduit l2 donc on diminue sa taille).

## Q1.5

`gen_permutation2 (p: int) (q: int) : int list`

Cette fonction récursive génère une liste d'entiers positifs compris dans l'intervalle [p,q], permutés aléatoirement.

- On vérifie que p et q ne sont pas négatifs, sinon on renvoie la liste vide.
- Si p est supérieur à q, on renvoie la liste vide.
- Si p est égal à q, on renvoie la liste composée de p seulement.
- Enfin, autrement, on calcule le nombre au milieu de l'intervalle [p, q] (leur moyenne entière), et on appelle *intercale* (définie en Q1.4) avec en premier argument la liste créée avec `gen_permutation2` allant de p jusqu'à la moyenne entière, et avec en deuxième argument la liste créée avec `gen_permutation2` allant de la moyenne entière +1 jusqu'à q. Cela permet de mélanger les listes créées à chaque appel récursif et donc d'avoir un meilleur mélange aléatoire, en dépit d'une complexité plus élevée.

## Q1.6

La complexité de la fonction `gen_permutation2` en nombre d'appels au générateur de nombre aléatoires est en **O(n²)** au pire cas. On peut justifier cette complexité par le fait qu'on appelle la fonction `intercale` de manière récursive sur chaque partie de la liste. Cette fonction dans le pire cas appelle n fois le générateur de nombre aléatoires.

La complexité de la fonction `gen_permutation2` en nombre d'appels à la structure `match with` est en **O(n²)** pour les mêmes raisons.

## Q1.7

Rappel : Un arbre binaire de recherche est un arbre trié optimisé pour la recherche dans lequel chaque nœud a une étiquette, un fils gauche dont l'étiquette est strictement inférieur ou est une feuille et un fils droit dont l'étiquette est strictement supérieure ou est une feuille. On considère qu'on ne peut pas insérer plusieurs fois le même entier dans le même ABR

```
(* Définition du type pour construire un ABR *)
type abr =
  | Feuille
  | Noeud of contenu

and contenu = {
  label : int ;
  gauche : abr ;
  droite : abr ;
}
```

On définit donc un type *abr* qui nous permet de représenter un arbre binaire de recherche de manière simple, un `abr` est :

- Soit une feuille, et donc un arbre vide
- Soit un noeud avec une étiquette, un fils gauche et un fils droit qui sont tous deux aussi des abr

On peut donc définir n'importe quel ABR de manière récursive.

Une meilleure manière de faire aurait été la suivante :

```
(* Définition du type pour construire un ABR *)
type abr =
  | Feuille
  | Noeud of {
    label : int ;
    gauche : abr ;
    droite : abr ;
  }
```

Cette version prendra en effet deux mots mémoire de moins car le contenu est défini directement dans Noeud et ne fait pas référence à un autre type.
Nous n'avons pas fait ce changement car il aurait fallu revoir et modifier nos `match with` qui se basaient sur `contenu` dans toutes nos fonctions et nous n'avons pas pris le temps pour effectuer ces changements.

`inserer_abr (noeud: abr) (arbre: abr) : abr`
Cette fonction permet simplement d'ajouter correctement le noeud dans l'*abr* et de retourner l'*abr* résultant de cette insertion.

`creer_abr (lst: int list) : abr`
Enfin, cette fonction permet de créer un *ABR* à partir d'une liste d'entiers en parcourant la liste et en ajoutant chaque élément dans l'*abr* résultat à l'aide de la fonction `inserer_abr`

et d'une fonction récursive interne qui se charge de parcourir la liste. Il aurait pu être intéressant dans ce cas d'utiliser la fonction `List.iter` pour parcourir les éléments de la liste.

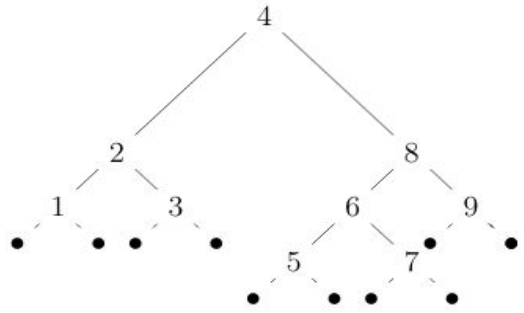
# Exercice 2

## Q2.8

```
let rec str_struct_abr (arbre: abr) : string =
  match arbre with
  | Feuille -> ""
  | Noeud n ->
    "(" ^ (str_struct_abr n.gauche) ^ ")" ^ (str_struct_abr n.droite)
```

Définition de la fonction permettant d'extraire la structure d'un arbre sous forme de chaîne de caractères en ignorant ses étiquettes. Cette fonction est très utile pour comparer la structure pure de tous les sous-arbres d'un ABR.

La structure est représentée par une chaîne de caractères composée du dictionnaire {'(' , ')'} (ensemble de parenthèses). On met une parenthèse ouvrante dès que l'on a un fils gauche, on explore récursivement ce même fils gauche, puis on ferme la parenthèse et on explore récursivement le fils droit.
Par exemple, l'arbre suivant aura pour représentation de structure "((())())((())())()" :




## Q2.9

`prefixe (arbre: abr) : int array`

Cette fonction sert simplement à récupérer toutes les étiquettes de l'ABR exploré dans l'ordre préfixe. Cette fonction ainsi que `str_struct_abr` seront utiles plus tard dans le projet.

## Q2.10

```
(* Définition du type pour construire un ABR compressé *)
type abr_comp =
  | Feuille_comp
  | Noeud_comp of contenu_comp
  | Pointeur_comp of pointeur_comp

and contenu_comp = {
  taille_c : int ;
  label_c : int ;
  gauche_c : abr_comp ;
  droite_c : abr_comp ;
}

and pointeur_comp = {
  cible_c : abr_comp ;
  labels_c : int array ;
}
```

Pour représenter un ABR compressé, il fallait définir un nouveau type pour qu'il puisse supporter d'être :

- Soit une feuille, donc le bout d'un arbre
- Soit un noeud, comme dans le type `abr` mais avec la taille de l'arbre en plus (pratique pour la recherche dans un pointeur qu'on détaillera plus tard)
- Soit un pointeur qui contient donc la cible qu'il pointe ainsi qu'un tableau des étiquettes présentent dans l'arbre

Ce type est presque identique au type précédemment défini, mais il permet de représenter au mieux un arbre compressé.

Comme pour le type `abr`, nous aurions une meilleure version possible prenant moins d'espace mémoire :

```
(* Définition du type pour construire un ABR compressé *)
type abr_comp =
  | Feuille_comp
  | Noeud_comp of {
    taille_c : int ;
    label_c : int ;
    gauche_c : abr_comp ;
    droite_c : abr_comp ;
  }
  | Pointeur_comp of {
    cible_c : abr_comp ;
    labels_c : int array ;
  }
```

Cependant pour les mêmes raisons que pour `abr`, nous n'avons pas modifié notre type.

Pour réaliser la fonction de compression d'un ABR, il nous a fallu créer moultes fonctions intermédiaires pour éviter le plus d'erreurs possible :

`chercher_structure (arbre: abr) (structure: string) : abr`
Cette fonction sert à chercher dans un arbre binaire de recherche de manière préfixe, si un nœud correspond à une certaine structure donnée à l'aide de la fonction `str_struct_abr`. Si aucun nœud n'est trouvé, la fonction renvoie simplement une feuille.

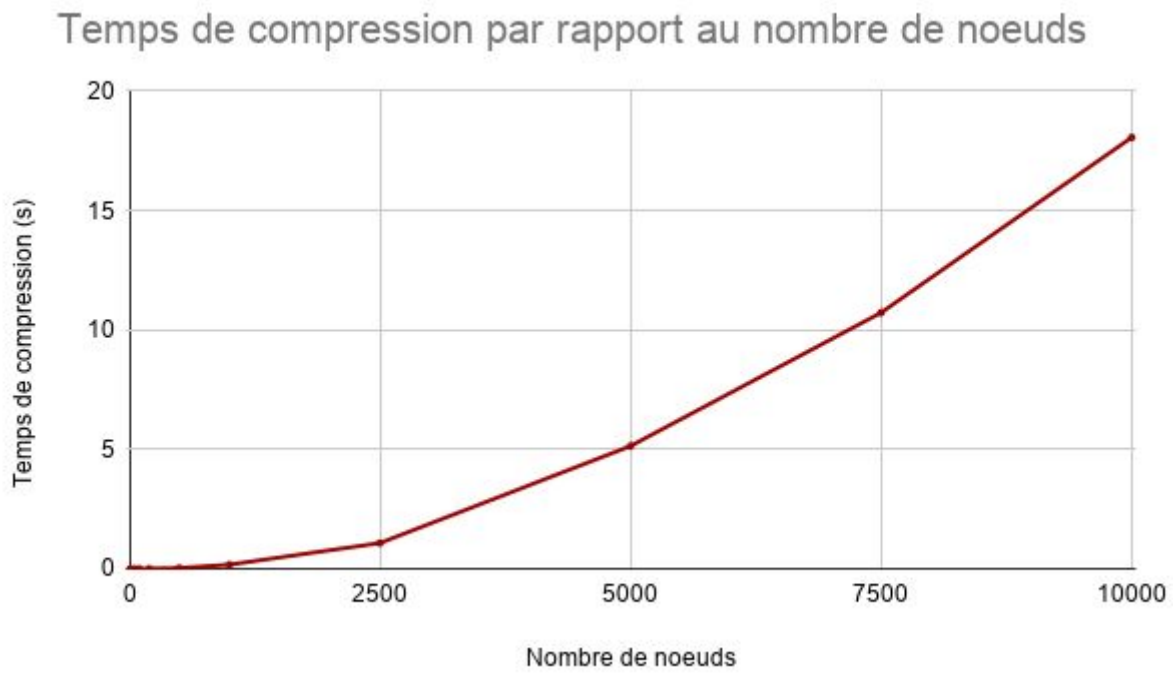
`chercher_label_comp (arbre: abr_comp) (lab: int): abr_comp`
On définit cette fonction pour permettre de trouver dans un ABR compressé un nœud avec l'étiquette donnée en ignorant les pointeurs. On utilise cette fonction pour retrouver dans un *abr_comp* la cible d'un pointeur.

`inserer_abr_comp (noeud: abr_comp) (arbre: abr_comp) : abr_comp`
Cette fonction est aussi très importante dans la construction d'un ABR compressé, elle permet d'insérer le nœud dans l'arbre et de retourner le résultat de cette insertion. Pour insérer un pointeur, il suffit de suivre le même schéma que pour un nœud classique, mais en considérant la première valeur dans les étiquettes du pointeur comme celle de la racine du sous-arbre.

`compresser_abr (src: abr) : abr_comp`
Cette fonction est la fonction "bout de chaîne" qui permet donc de compresser un ABR. Elle utilise une fonction récursive interne `parcourir_arbre (noeud: abr) (compresse: abr_comp) : abr_comp` permettant de garder l'arbre compressé en construction pendant qu'elle parcourt l'arbre source.
Elle se décompose en plusieurs étapes :

- On parcourt l'arbre source en ordre préfixe
- Pour chaque nœud on regarde s' il existe une structure identique dans l'arbre source avant lui même en ordre préfixe à l'aide de la fonction `chercher_structure` .
- Si on trouve qu'il n'y a **pas** de noeud de **la même structure** avant (le label du noeud trouvé est égal à celui de notre noeud donné) dans l'arbre, on insère alors un nouveau noeud compressé avec un taille de base de 1 avec la fonction `inserer_abr_comp` puis on appelle la fonction de compression sur le fils droit, avec le résultat du fils gauche, ce qui correspond à un parcours préfixe.
- En revanche si on **trouve** un noeud de **la même structure** avant celui que l'on est en train de traiter (le label du noeud trouvé est différent de celui de notre noeud donné), cela veut dire que l'on peut faire un pointeur vers le noeud qu'on a trouvé, on recherche donc le noeud à pointer avec la fonction `chercher_label_comp` et on insère le pointeur en récupérant la liste des étiquettes avec la fonctions `prefixe` .

Voici un diagramme de l'efficacité de notre algorithme de compression :




Le temps de compression est en **O(n²)** ce qui est relativement long et peut prendre beaucoup de temps pour des grosses données, mais l'efficacité de la compression est surtout importante dans ses résultats en termes d'espace mémoire et d'efficacité de recherche. Il serait néanmoins intéressant d'optimiser la fonction de compression pour réduire sa complexité.

## Q2.11

Cette fonction permet de rechercher la présence d'une valeur dans un `abr_comp`. Elle renvoie *true* si la valeur existe parmi les labels de `l'abr_comp`, sinon elle renvoie `false`.

```
262  let rec recherche_valeur_comp (arbre: abr_comp) (valeur: int) : bool =
263    match arbre with
264    | Feuille_comp -> false
265    | Noeud_comp {taille_c = _ ; label_c ; gauche_c ; droite_c} ->
266
267      if label_c = valeur then true
268      else if valeur > label_c then
269        recherche_valeur_comp droite_c valeur
270      else
271        recherche_valeur_comp gauche_c valeur
272
273    | Pointeur_comp {cible_c ; labels_c} ->
```

Tout d'abord il faut regarder à quoi ressemble l'ABR compressé *arbre.*

- Si c'est une **feuille** on n'a pas besoin de chercher plus loin : la valeur ne sera pas présente dedans.
- Si c'est un **noeud**, il faut vérifier son label et le comparer avec la valeur. S'ils sont égaux, on a trouvé la valeur et on renvoie true. Par définition, un ABR est trié avec systématiquement, les valeurs inférieures au label à sa gauche, et les valeurs supérieures à sa droite. Ainsi, si la valeur est inférieure au label il faut aller visiter

récursivement le fils gauche, et si la valeur est supérieure il faut aller visiter récursivement le fils droit.

- Si c'est un **pointeur**, c'est plus complexe. On détaille la suite de la fonction ci-dessous :

```
| Pointeur_comp {cible_c ; labels_c} ->

  (* Il faut parcourir intelligemment son tableau *)
  let rec parcours_labels (cible: abr_comp) (labels: int array) (i: int) : bool = (
    if i >= Array.length labels then false

    else if valeur = labels.(i) then true

    else if valeur < labels.(i) then (

      (* on veut aller à gauche *)
      match cible with
        | Noeud_comp noeud_cible ->
          parcours_labels noeud_cible.gauche_c labels (i + 1)
        | _ -> false

    ) else (
```

Un pointeur est composé d'une cible et d'un tableau de labels. Pour trouver la valeur, il faut parcourir le tableau de labels, mais pas de 0 jusqu'à sa longueur, puisqu'on doit faire une recherche efficace et que le tableau est trié en ordre préfixe.
On définit ainsi une fonction interne récursive chargée de parcourir le tableau, qui prend en argument une cible, un tableau d'entiers et un entier qui servira de curseur dans le tableau :

- Pour le cas **terminal** où i dépasse du tableau, c'est qu'on a pas trouvé notre valeur ; on renvoie false.
- Si la valeur est **égale** à celle contenue dans le tableau à la position i, c'est qu'on a trouvé la valeur ; on renvoie true.
- Si la valeur est **inférieure** à celle contenue dans le tableau à la position i, c'est qu'il faut aller visiter le fils gauche. Dans un tableau en ordre préfixe ([noeud, fils_gauche, fils_droit]), il faut donc avancer à la case suivante pour continuer notre parcours. Il faut donc appeler récursivement la fonction en prenant pour cible le fils gauche de la cible, et avancer i de 1.
- Si la valeur est **supérieure** à celle contenue dans le tableau à la position i, c'est qu'il faut aller visiter le fils droit. Nous détaillons cette partie ci-dessous :

```
    ) else (

      (* On veut aller à droite : nécessite saut dans le tableau *)
      match cible with

        (* Précision de la cible (forcément un noeud) pour avoir accès ses fils *)
        | Noeud_comp noeud_cible -> (

          (* Calcul de la taille du fils gauche pour aller au fils droit dans le tableau *)
          match noeud_cible.gauche_c with
          | Noeud_comp gauche_cible ->
            parcours_labels noeud_cible.droite_c labels (i + 1 + gauche_cible.taille_c)
          | Pointeur_comp gauche_cible ->
            parcours_labels noeud_cible.droite_c labels (i + 1 + (Array.length gauche_cible.labels_c))
          | Feuille_comp ->
            parcours_labels noeud_cible.droite_c labels (i + 1)

        )

        | _ -> false

    )
  ) in parcours_labels cible_c labels_c 0
```

C'est un cas plus compliqué que pour visiter le fils gauche, car pour aller au fils droit, il faut sauter du bon nombre de cases dans le tableau. Ce nombre est la taille du fils gauche.
Pour ce faire, nous avons besoin de regarder le fils gauche pour obtenir sa taille :

- Si c'est un **noeud**, on a accès à sa taille grâce à la structure du type `Noeud_comp` qui possède un champ *taille_c* et on peut donc l'utiliser pour sauter récursivement à *i + 1 + taille_c* cases (on fait +1 pour prendre en compte l'avancée systématique d'une case pour ne pas rester sur le noeud père) en pointant vers le fils droit.
- Si c'est un **pointeur**, on peut obtenir sa taille en prenant la taille de son tableau de labels et donc sauter récursivement à *i + 1 + taille du tableau* cases en pointant vers le fils droit.
- Si c'est une **feuille**, le fils gauche est donc vide : il est de taille 0, donc on saute récursivement à la case suivante (*i + 1 + 0*), qui sera donc le fils droit, en pointant sur le fils droit.

Et finalement, pour lancer cette fonction interne, on l'appelle sur notre cible de départ avec son tableau de labels, en commençant au début du tableau (i = 0).

La complexité en pire cas de la recherche dans un ABR compressé (fonction `rechercher_valeur_comp`) est la recherche d'une valeur qui n'est pas dans l'ABR, et que cet ABR n'est pas équilibré, par exemple avec toutes ses valeurs systématiquement en fils gauche (arbre pouvant être créé à partir d'un tableau d'entiers trié en ordre décroissant). Ainsi, la fonction va parcourir les n noeuds qui composent l'ABR avant de retourner qu'il n'a rien trouvé.
La complexité du pire cas est donc en **O(n)**.
Tous les autres parcours de l'ABR ont cette complexité ou moins.

Nous sommes beaucoup revenus sur la fonction `rechercher_valeur_comp`.
Premièrement car tous les tests avec `a1_comp` ne passaient pas, puis nous avons augmenté nos jeux de tests pour s'assurer que tout fonctionne bien avec des arbres plus complexes comme `a2_comp`. Cela a levé de nouvelles erreurs qui nous ont permis de corriger complètement notre fonction et d'être sûrs de son bon fonctionnement.
Nous utilisions également trop spécifiquement le match, en spécifiant les champs à chaque fois, ce qui était inutile et lourd, surtout syntaxiquement:

*Avant relecture :*

```
match cible with
     | Noeud_comp {taille_c = _ ; label_c = _ ; gauche_c = gc ;
droite_c = _} ->
       parcours_labels gc labels (i + 1)
   | _ -> false
```

*Après relecture :*

```
match cible with
   | Noeud_comp noeud_cible ->
       parcours_labels noeud_cible.gauche_c labels (i + 1)
   | _ -> false
```

## Q2.12 (facultative)

Il est vrai que la complexité la plus intéressante à trouver est celle en moyenne, plus que celle dans le pire cas.
Cette complexité moyenne est en **O(log(n))**.

# Exercice 3

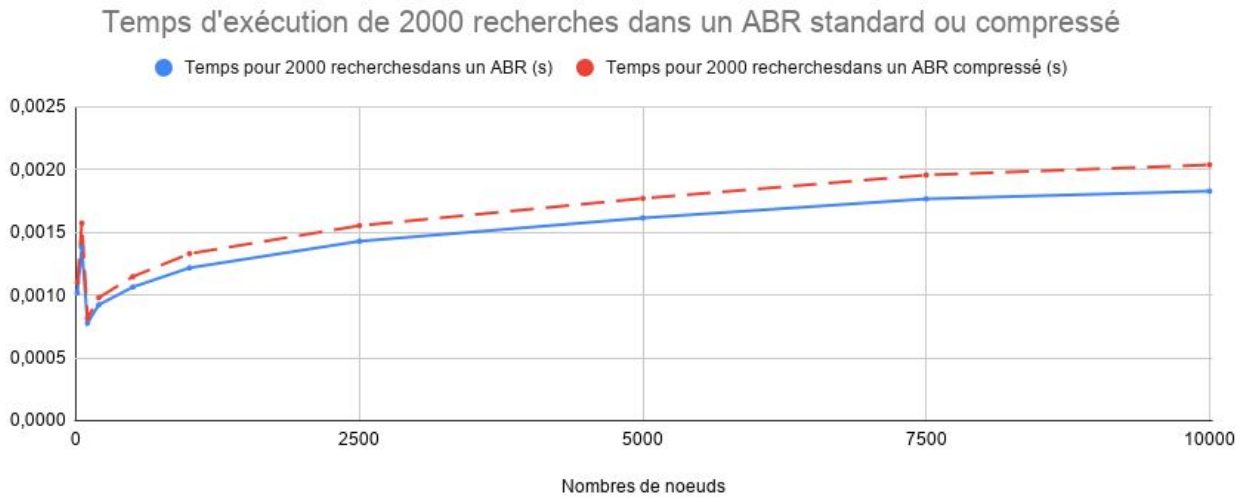
## Q3.14

```
let chrono fonction arg1 arg2 : float =
  let start_time = Sys.time () in
  fonction arg1 arg2 ;
  Sys.time () -. start_time
```

La fonction de test qui permet de mesurer le temps d'exécution de 20000 requêtes de recherche dans un ABR fonctionne de la sorte :

- On retient le temps de départ avec la fonction `Sys.time`
- On exécute 20000 fois la recherche d'une valeur prise au hasard entre 1 et le nombre de noeuds de l'ABR ou de l'ABR compressé.
- On retourne le temps courant (`Sys.time` également) moins le temps de départ.

Voici un diagramme du temps d'exécution de 20000 recherches dans un ABR et dans un ABR compressé en fonction de son nombre de noeuds :

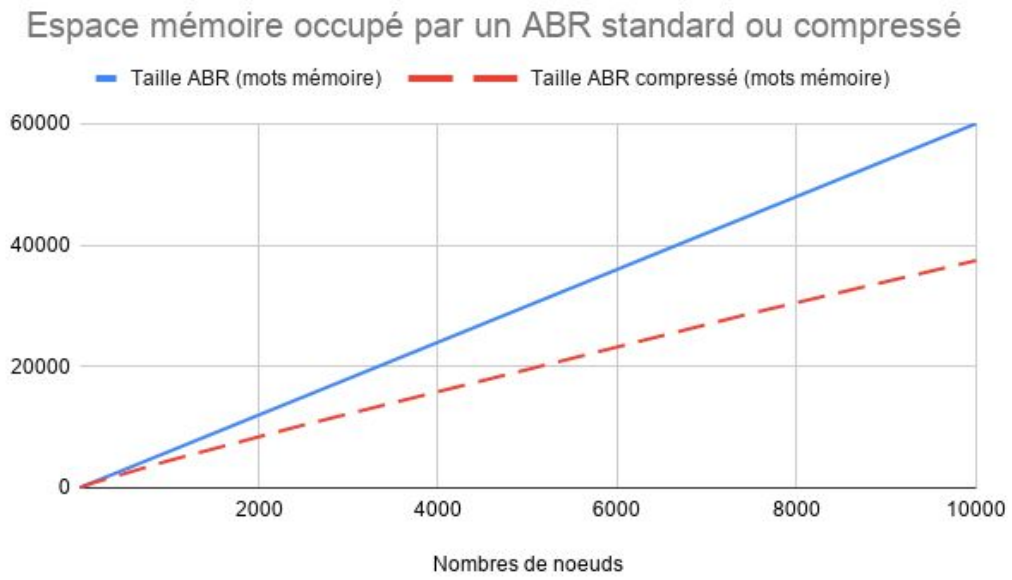



Nous observons une légère augmentation du temps de recherche dans un arbre compressé mais celle-ci reste minime par rapport au gain d'espace mémoire, et sa croissance est linéaire.
On calcule donc que le rapport moyen de "temps de recherche ABR non compressé / temps de recherche ABR compressé" est 0,9135035817.

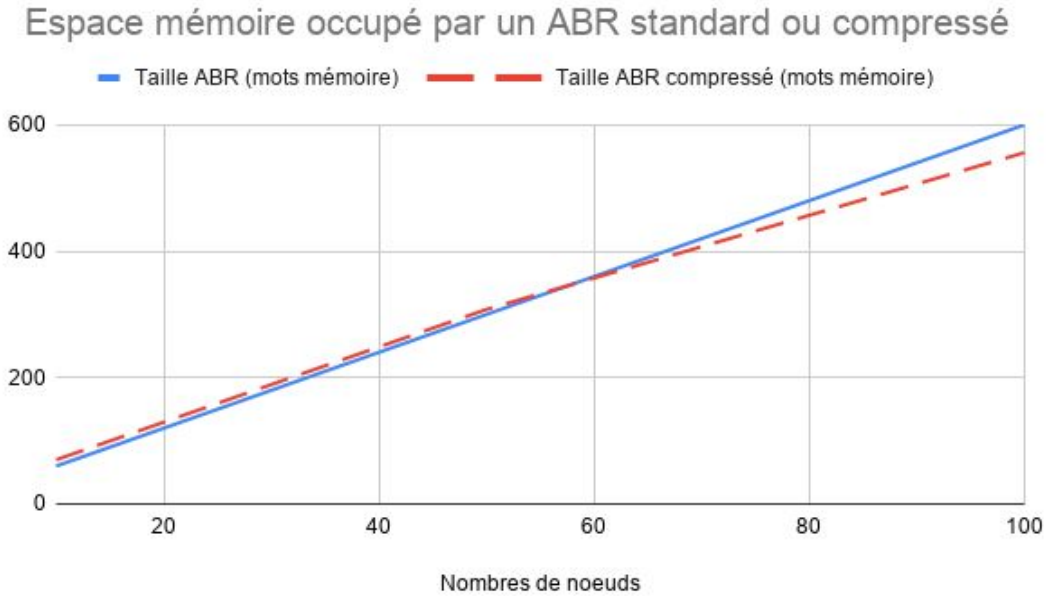
## Q3.15

```
(* Fonction pour récupérer la taille de l'espace mémoire occupé par une structure *)
let sizeof (x: 'a) : int = Obj.reachable_words (Obj.repr x)
```

Cette fonction permet de calculer l'espace mémoire (en mots mémoire) occupé par une structure.




Nous pouvons observer sur ce graphique que pour un ABR donné, sa version compressée prendra en moyenne deux tiers de sa place en mémoire, ce qui est un gain d'espace très intéressant. Pour 10000 noeuds par exemple, le rapport "taille de l'ABR non compressé / taille de l'ABR compressé" est de 1,601879539



Cependant nous pouvons voir qu'entre 1 et 60 nœuds environ, la version compressée d'un ABR prend plus d'espace mémoire que la version non compressée. On peut donc en conclure que la compression n'est intéressante qu'avec des arbres de plus de 60 nœuds.

# Fichier de test

Nous avons créé des jeux de tests plutôt fournis qui nous ont été très utiles, ainsi qu'un makefile pour lancer leur exécution.
Vous pourrez donc lancer les lignes de commande make test et make clean pour constater nos études de tests.

Pour les tests, l'`abr a1` reprend l'ABR plutôt simple du sujet, avec 2 structures réutilisables et des pointeurs courts (hauteurs de 1 ou 2).
L'`abr a2` est un ABR plus complexe, avec 3 structures réutilisables et des pointeurs plus longs (hauteurs de 1, 2 ou 3) et qui permettent d'explorer chacun des cas de la fonction `rechercher_valeur_comp`.

Visuels de `a2` et `a2_comp` :