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`

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.

Q1.3

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.

Q1.6

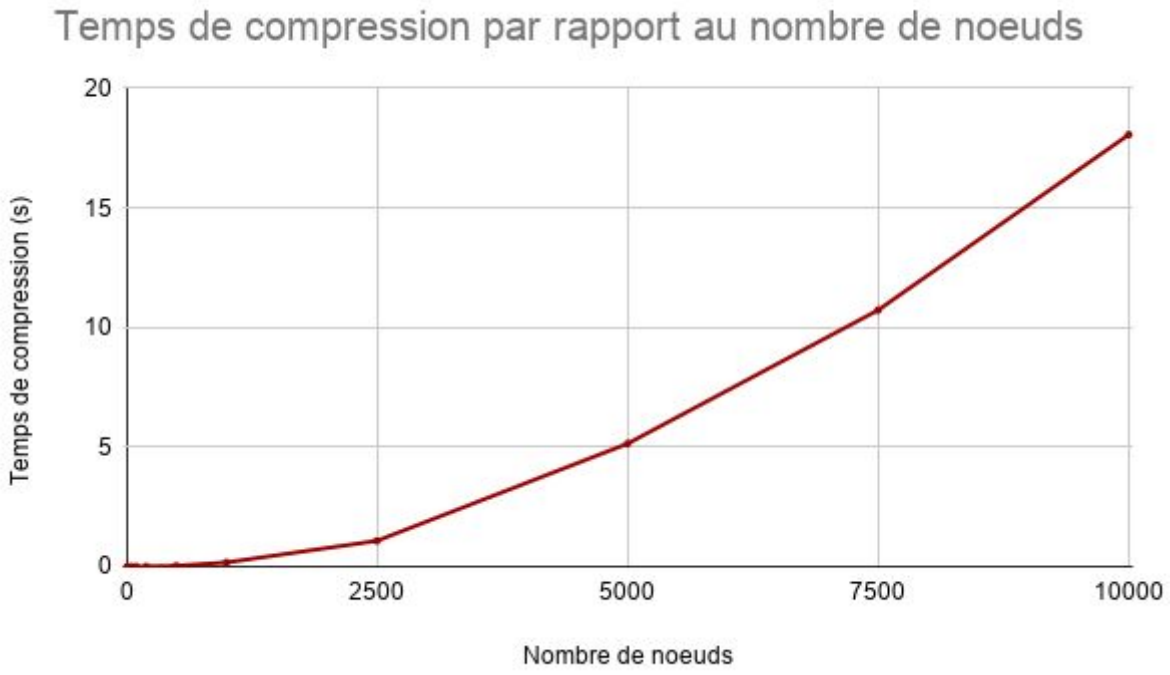
La complexité de la fonction `gen_permutation2` en nombre d'appels au générateur de nombre aléatoires est en **O(n²)** au pire cas. On peut justifier cette complexité par le fait qu'on appelle la fonction `intercale` de manière récursive sur chaque partie de la liste. Cette fonction dans le pire cas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

Exercice 2

Q2.10

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



Le temps de compression est plutôt élevé avec une tendance carrée, mais l'efficacité de la compression est surtout importante dans ses résultats en termes d'espace mémoire et d'efficacité de recherche.

Q2.11

La complexité en pire cas de la recherche dans un ABR compressé (fonction `recherche_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 `recherche_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 :
```
match cible with
   | Noeud_comp {taille_c = _ ; label_c = _ ; gauche_c = gc ;
droite_c = _} ->
       parcours_labels gc labels (i + 1)
   | _ -> false
```

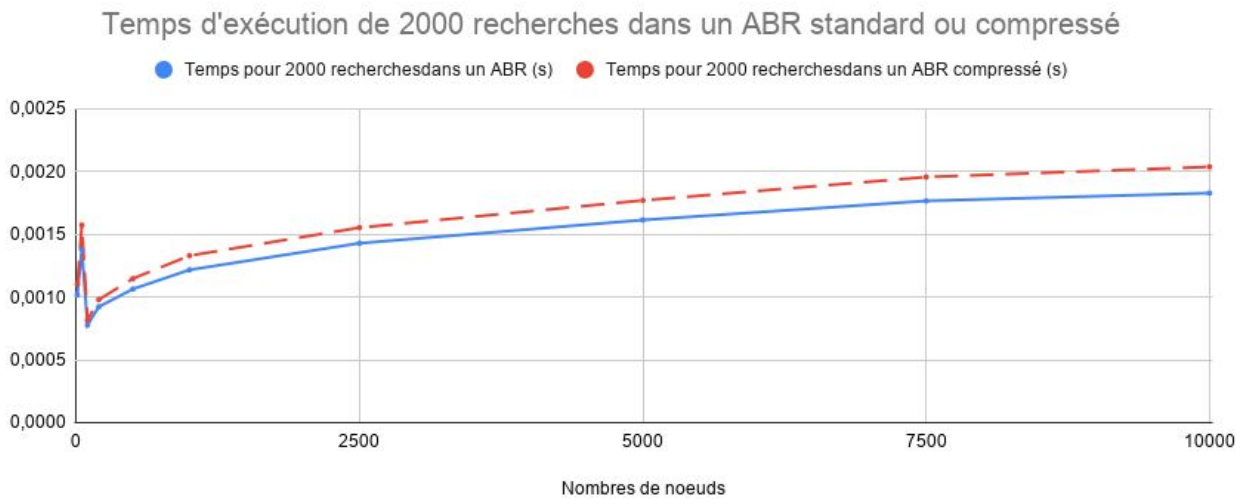
Après relecture :
```
match cible with
   | Noeud_comp noeud_cible ->
       parcours_labels noeud_cible.gauche_c labels (i + 1)
   | _ -> false
```

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))**.

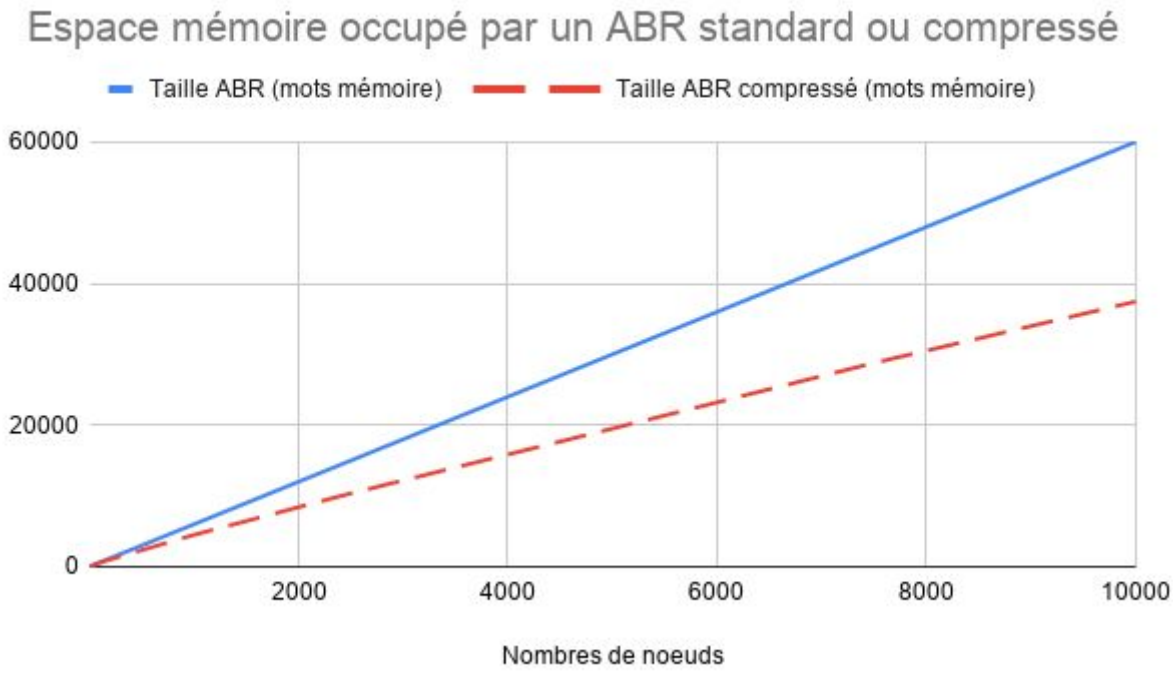
Exercice 3

Q3.14



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.

Q3.15



Nous pouvons observer sur ce graphique que la diminution de prise d'espace mémoire est de près d'un tiers, ce qui est non négligeable sur ce genre de données gloutonnes en taille.

Explications supplémentaires

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 `recherche_valeur_comp`.

Visuels de `a2` et `a2_comp` :