Devoir de Programmation
UE Ouverture

ARNOULD Yann, AYED Fatima, KONÉ Daba

Table des matières

1. Fonctions

1.1 Structure de données

Nous avons défini une structure de données permettant de manipuler les polynômes de manière simple et efficace. Cette structure se compose de deux types principaux : les monômes et les polynômes. Un monôme est représenté par un couple d'entiers, tandis qu'un polynôme est une liste de ces monômes.

1.2 Canonique

1.2.1 Implementation

Nous définissons plusieurs fonctions internes dans notre implémentation de canonique pour assurer que le polynôme renvoyé soit conforme à la forme canonique.

remove_double
Prend le polynôme donné à canonique et additionne les coefficients des monômes de même degré, garantissant ainsi qu'il n'y ait qu'un seul coefficient pour chaque degré de la variable formelle. Pour ce faire, elle se sert de la fonction auxiliaire aux.

aux
Prend un monôme (coefficient, degré) et introduit une hashtable ‘table’ qui permettra d'additionner les monômes de même degré efficacement. La clé de cette table est le degré de chaque monôme. Ainsi, nous stockons chaque coefficient dans la liste indexée par le degré de son monôme. Si un degré n'a pas encore de liste associée, nous créons une nouvelle liste pour y stocker le coefficient. Sinon, nous ajoutons le coefficient dans la liste correspondante.

Notre hashtable est créée avec une taille égale à celle du polynômes p donné à la fonction canonique, afin de s’assurer que chaque monôme puisse être stocké dans une liste. Dans le pire des cas, chaque monôme a un degré différent et il n’y aura aucune collision.
Il est important de noter que, grâce à la manière dont la table de hachage est implémentée en OCaml, sa taille n’est pas dépendante de la clé de hachage. Ainsi, peu importe le degré, la table pourra toujours mapper la clé à un index valide.

La fonction aux est appliquée à chaque monôme en utilisant List.iter, qui parcourt le polynôme p.

Après que tous les monômes ont été ajoutés à notre table de hachage, List.fold_left additionnera tous les coefficients d’une même liste et renverra tous les monômes sous la

forme (somme des coefficients, degré) à la fonction Hashtbl.fold. Cette dernière utilise un accumulateur acc pour stocker la liste des monômes sans doublons.

Ensuite, nous définissons une fonction sort_by_degree qui utilise List.sort pour trier les monômes par ordre croissant des degrés, et une fonction remove_null qui utilise List.filter pour enlever tout monôme avec un coefficient nul.

Notre fonction canonique renvoie ainsi un polynôme sous forme de liste de monômes sans doublons, triée par ordre croissant et sans monôme null. Nous avons ainsi respecté toutes les contraintes imposées par l'énoncé.

1.2.2 Complexité

La complexité de la fonction canonique dépend de la complexité des différentes fonctions internes qui la composent :

remove_double
La création et initialisation de la Hashtable à une complexité de O(1). Le parcours de la liste p s'effectue une seule fois avec List.iter, la complexité est donc en O(n) avec n la taille de la liste.
Ensuite, l'insertion dans la table de hachage comprend plusieurs opérations : Hashtbl.find recherche les degrés pour chaque monôme avec une complexité moyenne de O(1) (que la recherche réussisse ou échoue, d'où l'intérêt d'utiliser une table de hachage pour éliminer les doublons). De même, Hashtbl.replace a une complexité de O(1). Ainsi, la complexité de l'insertion est en O(1).

Il convient de noter que la complexité au pire cas de l'insertion dans une table de hachage est de O(n), c'est-à-dire lorsque tous les coefficients sont stockés dans la même liste.

Hashtbl.fold parcourt les cases de la hashtable. En l'absence de collisions (c'est-à-dire si tous les degrés sont différents), la complexité est en O(n). Autrement, la complexité est négligeable.

List.fold_left additionne tous les coefficients d'une même liste. Son pire cas serait que tous les monômes aient le même degré, ce qui donnerait une complexité de O(n). Autrement, ce n'est pas une opération très coûteuse.

Il y a différentes manières d'interpréter le "pire cas" et le "cas moyen" pour la complexité de notre fonction remove_double. Nous choisissons les suivantes :

Ici, la complexité en pire cas serait:

- tous les degrés sont différents et donc toutes les cases de la hashtable sont occupées. Hashtbl.fold a donc une complexité de O(n).
- Cela implique qu'il n'y a aucune collision, donc la complexité de l'insertion dans la hashtable est de O(1).

- Nous pouvons négliger le coût de List.fold_left car dans notre pire cas, les coefficients n'ont pas tous le même degré.

Ainsi, notre pire cas correspond à appliquer la fonction remove_double alors qu'il n'y a aucun coefficient à additionner, avec une complexité de O(n).

De plus, la complexité en cas moyen serait:

- Il y a des monômes avec le même degré dans notre polynôme, donc Hashtbl.fold ne parcourt pas toutes les cases de la table. Nous pouvons négliger son coût.
- Il y a des collisions, mais nous ne considérons pas une complexité d'insertion en O(n) (signifiant que tous les coefficients ont le même degré). Nous estimons que les collisions sont bien réparties dans la table de hachage.
- Ici aussi nous pouvons négliger le coût de List.fold_left.

Notre cas moyen correspond donc à appliquer la fonction remove_double sur un polynôme avec plusieurs coefficients à additionner pour différents degrés, donnant une complexité de O(1).

sort_by_degree

List.sort est basé sur l'algorithme de tri fusion, avec une complexité de O(n'log(n')). Il est important de noter que List.sort est appliquée sur la liste de monôme sans doublons. Ainsi, ici, le n n'est pas la taille de notre polynôme : nous avons donc n' <= n.

Ici, la complexité en pire cas serait :

- tous les polynômes ont des degrés différents, il y a donc n polynômes à trier (n' = n). La taille reste celle du polynôme en entrée, et la complexité est de O(nlog(n)).

De plus, la complexité en cas moyen serait :

- Il y a eu plusieurs collision, nous avons donc n' < n et la complexité est de O(n'log(n')).

remove null

List.filter parcourt la liste sans doublons et vérifie si les coefficients sont nuls. Sa complexité et donc de O(n') avec n' <= n.

1.2.2.1 Conclusion: Complexité de la fonction Canonique

Complexité total au pire cas : O(n) + (O(nlog(n)) + O(n) = O(nlog(n))
Complexité total au cas moyen : O(1) + O(n'log(n')) = O(n'log(n'))

Ainsi, nous obtenons une meilleure complexité dans le cas moyen car n' est strictement inférieur à n.

1.2.3 Assert

Nous avons choisi nos assertions de manière judicieuse afin de vérifier que toutes les contraintes sont respectées :

1. l'assert sur poly1 permet de vérifier que le tri s'effectue bien dans l'ordre croissant
2. l'assert sur poly2 permet de cibler l'addition des monômes de même degré
3. l'assert sur poly3 vérifie que les monômes nuls sont bien retirés de la liste
4. l'assert sur poly5 vérifie que le monôme nul est retiré de la liste même s'il est le seul élément, et qu'une liste vide est renvoyée
5. l'assert sur poly6 vérifie que notre fonction renvoie bien une liste vide si elle reçoit un polynôme vide en entrée

1.3 Poly_add

1.3.1 Implementation

Nous implémentons deux fonctions : poly_add et poly_add_canonique.

Nous avons choisi de faire deux implémentations différentes pour des raisons d'optimisation. En effet, nous avons besoin de deux versions différentes par rapport à poly_prod (la question suivante).
poly_add sera utilisée plus tard dans poly_prod et nous avons remarqué que poly_prod est optimal lorsque nous lui donnons notre version de poly_add qui ne fait pas appel à canonique.
Or dans l'énoncé, poly_add se doit de renvoyer un polynôme sous forme canonique, donc nous avons défini une fonction poly_add_canonique afin de répondre à la question.
Nous aurions pu implémenter poly_add_canonique en ajoutant canonique à l'intérieur d'une fonction auxiliaire qui reprend poly_add mais cela serait de la copie de code inutile.
Nous avons donc décidé de tout simplement appeler canonique sur poly_add.

Nous implémentons la fonction poly_add avec du pattern-matching. On compare les monômes des 2 listes de polynôme p1 et p2 un à un et on les ajoute en tête de liste puis on appelle récursivement poly_add :

- Si p1 et p2 sont vide, nous renvoyons l'accumulateur
- Si une des deux listes est vide, disons p1, nous renvoyons p2
- Si le monôme x de p1 n'as pas le même degré que celui de p2 (z) :
    - Dans le cas où le degres de x est plus petit que celui de z, nous l'ajoutons en tête de liste puis effectuons l'appel récursif sur le reste de p1 et sur p2 (sans enlever z).
      Nous suivons la même logique si z est le monôme de plus petit degré.

  Ceci est possible car nos listes sont triées et il n'y a pas de doublons. Ainsi, si le degré de x est plus petit que le degré de z, nous savons que le coefficient de x ne pourra s'additionner avec aucun autre coefficient dans p2 car tous les monômes qui suivront z auront des degrés plus grands que celui de x.

- Si x et z on le même degré, nous les additionnons et les ajoutons en tête de liste puis appelons poly_add sur le reste des deux listes. Les polynômes n'ayant pas de doublons, nous savons que nous n'aurons aucun autre monôme avec le même degré.

poly_add_canonique
Nous appelons canonique sur le polynôme résultant de l'appel à poly_add. Ainsi, notre fonction renvoie la somme de deux polynômes canoniques sous forme canonique.

1.3.2 Complexité

poly_add
Parcours les deux polynômes. Supposons que nous avons un polynôme p1 de taille n et un polynôme p2 de taille t, la complexité est en O(n+t).

canonique
Comme vu précédemment, la complexité en pire cas de canonique est de O(nlog(n)). Or ici dans le pire cas la taille de la liste renvoyée par poly_add sera de n+t si aucun monôme ne s'additionne. Donc la complexité est de O((n+t)log(n+t)).

1.3.2.1 Conclusion: Complexité de la fonction poly_add

Complexité total au pire cas: $O(n+t) + O((n+t)\log(n+t)). = O((n+t)\log(n+t))$.

1.3.3 Assert

Nos assertions ont été réalisées avec les mêmes polynômes utilisés pour les assertions de la fonction canonique, ce qui nous permet de couvrir tous les cas à prendre en compte.

1.4 Poly_prod

1.4.1 Implementation

Nous implémentons deux versions de poly_prod : poly_prod qui ne fait pas appel à canonique et poly_prod_canonique qui renvoie le produit des polynômes sous forme canonique.

poly_prod_canonique nous permet de répondre aux contraintes de l'énoncé tandis que poly_prod est l'implémentation la plus efficace pour la partie expérimentation de notre projet.

Nous implémentons la fonction poly_prod qui prends deux polynômes p1 et p2 avec une fonction interne monome_prod qui prend un monôme de p1 et le multiplie avec tous les monômes de p2 (les coefficient sont multipliés tandis que les degrés sont additionné) en utilisant List.map.
Ensuite List.fold_left nous permet de répéter la même opération sur tous les monômes de p1 afin de calculer le produit total en les accumulant dans acc sous forme de somme en appellant poly_add.

poly_prod_canonique suit la même logique que que poly_add_canonique.

Nous tenons à appuyer sur le fait que nous avons implémenter des versions de poly_add et poly_prod sans appel à canonique car avec nos anciennes versions où canonique était appelé à la fin de l'exécution de nos fonction, ce qui ajoutait du temps d'exécution additionnel et rendait notre partie expérimentations moins optimale.

1.4.2 Complexité

La complexité de cette fonction dépend de celle de monome_prod et canonique.

monome_prod et List.fold_left

monome_prod parcours p2 donc la complexité au pire cas est de O(t) avec t la taille de p2.
Ensuite, List.fold_left applique la fonction monome_prod à chaque monôme de p1 qui est de taille n, la complexité totale à la fin de tous les appels à monome_prod est donc de O(n*t).

poly_add
Comme vu auparavant, poly_add a une complexité de O((m+v)log(m+v)) avec m la taille de l'accumulateur et v la taille du résultat de monome_prod.
Ceci s'applique pour chaque appel de monome_prod (un appel de monome_prod est de complexité O(t)) qui lui même est appelé n fois par List.fold_left (O(n)) donc en tout c'est une complexité de O(n*t*(m+v)log(m+v)).

De plus, à la fin de tous les appels à poly_add, la fonction canonique est appelée sur le polynôme calculé (disons qu'il a une taille q), la complexité est de O(q*log(q)).

1.4.2.1 Conclusion: Complexité de la fonction poly_add

Complexité total au pire cas: O(n*t*(m+v)log(m+v)) +O(qlog(q)).

Supposons que dans le pire cas, tous les monômes aient des degrés différents. La taille du résultat passé à canonique (notée q) sera donc n*t (taille de p1 * taille de p2).

Nous avons donc comme complexité totale O(n*t*(m+v)log(m+v)) +O(n*t*log(n*t))= O(n*t(m+v)[log(m+v)+ log(n*t)]).
log(n*t) domine car m+v sont des valeurs intermédiaires, dans le pire des cas n et t sont supérieures.

Nous obtenons une complexité finale de O(n*t*log(n*t)).

1.4.3 Assert

Nos assertions ont été réalisées avec les mêmes polynômes utilisés pour les assertions de la fonction canonique, ce qui nous permet de couvrir tous les cas à prendre en compte. Il est ici important de vérifier que nous éliminons correctement les monômes nuls à chaque mise à jour de l'accumulateur, afin d'éviter des multiplications inutiles.

1.5 Structure de données pour les arbres

1.5.1 Implementation

Nous avons implanté une structure de données simple à utiliser en définissant un seul type expression. Ce type peut être composé d'un entier (Int), d'une puissance (Pow, qui est composé d'un char et d'un int), ainsi que des constructeurs Plus et Mult. Le constructeur Plus prend une liste d'expressions, tandis que Mult prend également une liste d'expressions.

1.6 Expression de la Figure 1 du sujet

let figure1 = Plus [Mult [Int 123; Pow ('x',1)]; Int 42; Pow ('x',3)];;

Vous pouvez notamment la retrouver dans le code source.

1.7 arb2poly

1.7.1 Implementation

Nous pouvons remarquer que notre implémentation de arb2oly repose sur la définition de quatre fonctions internes. La raison pour laquelle nous avons autant de fonctions internes est que la grammaire de l'énoncé impose plusieurs contraintes à respecter, que nous détaillerons tout au long de ce paragraphe.

testPow

Cette fonction permet de vérifier que notre couple (char, int) est bien valide. Nous n'avons qu'une seule variable formelle, x, et notre entier doit être strictement positif. Ainsi, si le couple qui compose le constructeur Pow ne respecte pas ces deux conditions, une exception est levée.

mult2poly

Cette fonction récursive permet de vérifier que le constructeur Mult respecte toutes les contraintes énoncées par la grammaire et permet ainsi de construire des monômes sous la forme (coefficient, degré).

Rappelons que Mult doit représenter un monôme (coef, degré) donc le but est de construire un couple (coef, degré) à partir du constructeur Mult.

1. Mult doit être un produit d'au moins deux expressions, ce qui est vérifié en testant la longueur de la liste qui le composent car il prend une liste d'expressions.

Ensuite, le pattern matching permet de vérifier pour chaque éléments de la liste composant Mult:

1. Si l'élément est un Int, on le récupère en construisant un couple dont il deviendra le premier élément puis on appelle la fonction récursivement sur le reste de la liste et sur ce couple. Si un couple est à déjà été construit, on l'additionne au coefficient déjà existant.
2. Si l'élément est une puissance, on appelle la fonction testPow et si une exception n'est pas levée, on récupère le degré de la puissance (valeur) qui deviendra la puissance du couple (on l'additionne à la puissance du couple déjà existant s'il y avait d'autres variables formelle 'x' dans notre liste). On appelle ensuite la fonction récursivement sur notre couple et le reste de la liste de Mult.
3. Si l'élément est un Plus, on appelle poly_pod(plus2poly) dessu car un Plus dans un Mult revient à appliquer la distributivité de la multiplication. On l'appelle sur plus2poly afin d'éviter toute multiplication inutile.
   On appelle ensuite récursivement mult2poly.
4. Si l'élément est un Mult, on lève une exception car Mult ne peut pas avoir de Mult comme opérateur principal des expressions qui le compose.

plus2poly

Cette fonction récursive permet de vérifier si le constructeur Plus respecte toutes les contraintes énoncées par la grammaire et ainsi de construire un polynôme qui est une somme de plusieurs monôme (somme de constructeur Mult, Pow, Int).
Rappelons que Plus prend une liste d'expressions.

1. Effectue la même vérification que Mult pour la taille de sa liste.
2. Si l'élément est un Int, on le récupère, créer une liste composé d'un seul couple (int, 0) ou l'additionne au coefficient du couple déjà existant puis on effectue l'appelle récursif en concaténant notre liste avec.
3. Si l'élément est un Mult, c'est la même idée mais on appelle mult2poly pour gérer les éléments composant Mult.
4. En cas d'un élément Plus, on lève une exception car il ne peut pas avoir soi même comme opérateur principal de ses expressions.

aux

Dans notre expression, il n'y a en réalité qu'un seul constructeur qui englobe tous les autres. Si l'expression est un Int, cela correspond à un monôme de degré 0. Pour un Pow, nous

obtenons un monôme avec un coefficient égal à 1 et un degré non nul. Il s'agit donc de monômes et non de polynômes.

De même, pour le constructeur Mult, qui représente plusieurs expressions à multiplier, le résultat sera un seul monôme, le produit des différentes expressions. Cette fonction permet de traiter ces cas et de vérifier qu'ils respectent bien les contraintes imposées.

Cependant, pour le constructeur Plus, le résultat sera un polynôme, puisque l'addition de plusieurs monômes donne bien un polynôme. Ainsi, cette fonction sert de porte d'entrée pour traiter l'expression donnée en argument à notre fonction arb2poly.

canonique
Nous souhaitons que le polynôme construit à partir de notre expression soit sous forme canonique, c'est pourquoi nous appelons la fonction canonique pour garantir cette propriété.

1.7.1 Assert

Nos premiers asserts (cas nominaux) nous permettent de couvrir tous les cas de base. Nous avons également développé une fonction assert_exception_arb2poly afin de lever une exception lorsque les contraintes essentielles de la grammaire ne sont pas respectées, telles que :

- Lorsque Mult et Plus sont appelés avec moins de 2 éléments dans leur liste.
- Lorsque Mult est appelé avec un Mult comme opérateur principale de ses expressions
- Lorsque Plus est appelé avec un Plus comme opérateur principale de ses expressions

1.8 extraction_alea

1.7.1 Implementation

La fonction extraction_alea est implémentée en générant aléatoirement un entier entre 1 et la taille de la liste l, en utilisant la fonction random qui fait appel à Random.int.

Ensuite, une fonction auxiliaire utilise le pattern matching sur la liste l et l'indice i. Si l'indice i correspond au chiffre généré aléatoirement, l'élément de la liste est ignoré et la fonction continue sur la queue de la liste. Sinon, l'élément est ajouté à la liste résultante et le parcours de la liste continue.
Cette fonction auxiliaire est ensuite appelée sur la liste l et l'indice 0 pour parcourir la liste, ajoutant l'élément correspondant (de l'indice généré aléatoirement) à la liste p, après qu'il ait été retiré de la liste l grâce à List.nth.

Nous n'avons pas implémenté d'asserts pour cette fonction, un simple print étant suffisant pour vérifier son bon fonctionnement.

1.9 gen_permutation

1.9.1 Implementation

Notre fonction est implémentée à l'aide de deux fonctions internes :

gen_liste
Cette fonction génère récursivement une liste d'entiers allant de 1 à n.

aux
Cette fonction appelle extraction_alea sur la liste l jusqu'à ce qu'elle soit vide. Les éléments extraits sont ajoutés à la liste p, qui est initialement vide.

Nous n'effectuons pas d'asserts non plus pour cette fonction.

1.10 abr

1.10.1 Implementation

Pour implémenter notre fonction abr, nous avons défini un type a tree = Empty | Node of 'a * 'a tree * 'a tree.

Ainsi, notre arbre est composé de nœuds, chaque nœud étant construit avec un élément de type 'a, un arbre gauche et un arbre droit. Nous utiliserons le constructeur Node pour construire les nœuds de l'arbre.

Nous implémentons ensuite une fonction interne insertion qui, grâce au pattern matching, permet d'insérer une valeur n dans un arbre a.

insertion
Si l'arbre est vide, nous créons un nouveau nœud avec la valeur n et deux sous-arbres vides.
Si l'arbre n'est pas vide, trois cas se présentent :

- x=n , on lève une exception car les doublons ne sont pas autorisés
- x>n on effectue l'appel récursif sur l'arbre droit jusqu'à trouver un sous arbre vide
- x<n on effectue l'appel récursif sur l'arbre gauche jusqu'à trouver un sous arbre vide

Ensuite, nous utilisons List.fold_left pour appliquer notre fonction insertion à tous les éléments de la liste l donnée en entrée à la fonction abr.

Enfin, nous réalisons deux asserts pour vérifier qu'aucun doublon n'est présent et que le résultat respecte bien les contraintes d'un arbre binaire de recherche (abr).

1.11 etiquetage

La fonction étiquetage est implémentée à l'aide de pattern matching sur l'arbre abr d'entrée, de type int tree. Il s'agit d'une fonction récursive qui répond à toutes les contraintes de l'énoncé de manière assez simple.

L'implémentation fonctionne comme suit :

- La fonction est appelée récursivement sur les sous-arbres gauche et droit pour traiter tous les nœuds de l'arbre.
- Random.float et Random.int sont utilisés pour générer aléatoirement les valeurs nécessaires à l'étiquetage des nœuds.

1.12 gen_arb

1.11.1 Implementation

Notre implémentation fait appel à plusieurs pattern matching, notamment dans la fonction auxiliaire aux.

Cas de base
Le pattern matching sur l'expression e donnée en entrée à gen_abr et en dehors de la fonction auxiliaire aux permet de traiter les cas de base tels que Int et Pow, qui sont renvoyés tels quels.
Pour les constructeurs Plus et Mult, le pattern matching agit comme une porte d'entrée pour traiter leurs éléments. Nous les traitons en appelant la fonction auxiliaire aux sur la liste qui les compose, ainsi que sur les caractères 'M' ou 'P' utilisés comme identifiants.

aux
La fonction aux prend une liste d'expressions (elle est d'abord appelée sur la liste qui compose Mult ou Plus) et effectue un pattern matching sur les différents constructeurs qu'elle contient :

- Cas Int ou Pow : Ces constructeurs sont renvoyés tels quels, en les ajoutant en tête de liste à l'appel récursif. Ils sont donc ajoutés à la liste de résultats.
- Cas Plus :
    - Si le caractère n est égal à 'P', cela signifie que nous sommes dans le contexte d'une somme, donc nous appelons aux sur les éléments de Plus tout en conservant 'P' comme caractère. Le résultat de cet appel est ensuite

concaténé avec l’appel récursif effectué sur le reste des éléments de la liste, en conservant 'P' comme identifiant.
    - Si n n’est pas égal à 'P' (indiquant que nous venons d’un Mult), nous appliquons la même logique mais cette fois en enveloppant le premier appel à aux dans le constructeur Plus.

- Cas Mult :On applique la même logique si on rencontre le constructeur Mult.

Nous avons ajouté un assert pour nous assurer que le fonctionnement de la fonction est correct.

2. Expérimentations

Nous avons choisi de récupérer les données sous forme de fichier .txt. Toutes les spécifications et les détails concernant l'écriture de ce fichier sont expliqués dans le paragraphe intitulé "Remarques importantes concernant l'écriture du fichier".

2.13 : exper_gen_abrs (n:int) (taille:int)

gen_permutations (n:int)

La fonction exper_gen_abrs s'appuie sur la fonction auxiliaire gen_permutations, qui génère de manière uniforme des listes d'entiers aléatoires entre 1 et la taille spécifiée, à l'aide de gen_permutation. Ces listes sont ensuite utilisées par exper_gen_abrs, qui se charge de les transformer en arbres binaires de recherche (abr), puis en expressions valides selon la grammaire définie dans la section 1.2. Cette transformation est effectuée à l'aide de List.map, qui applique successivement les fonctions abr, etiquetage, et gen_arb à chaque liste générée.

- Liste de listes d'entiers → Liste d'ABR
- (abr) → Liste d'ABR
- (etiquetage) → Liste d'expressions non valides par la grammaire
- (gen_arb) → Liste d'expressions valides selon la grammaire

exper_gen_abrs_20 (n:int) (pas:int) max(int)

Afin de pouvoir représenter les temps de calcule de la fonction exper_gen_abrs pour n variant de 100 à 1000 avec un pas de 100 comme l'exigeait l'énoncé, nous avons développé la fonction exper_gen_abrs_20 qui va exécuter 10 fois exper_gen_abrs avec n variant de 100 à 1000 avec un pas de 100 et qui va renvoyer la liste des listes d'expressions renvoyer par les différents appels à exper_gen_abrs tout en écrivant la durée de l'appel à exper_gen_abrs et l'itération n pour laquelle il a été calculé dans un fichier exper_gen_abrs_20.txt sous le format n:durée.

time_execution

Avant de parler des questions 2.14 et 2.15, il faut aborder la fonction time_execution qui sera utilisée plus tard dans les 2 questions suivantes.

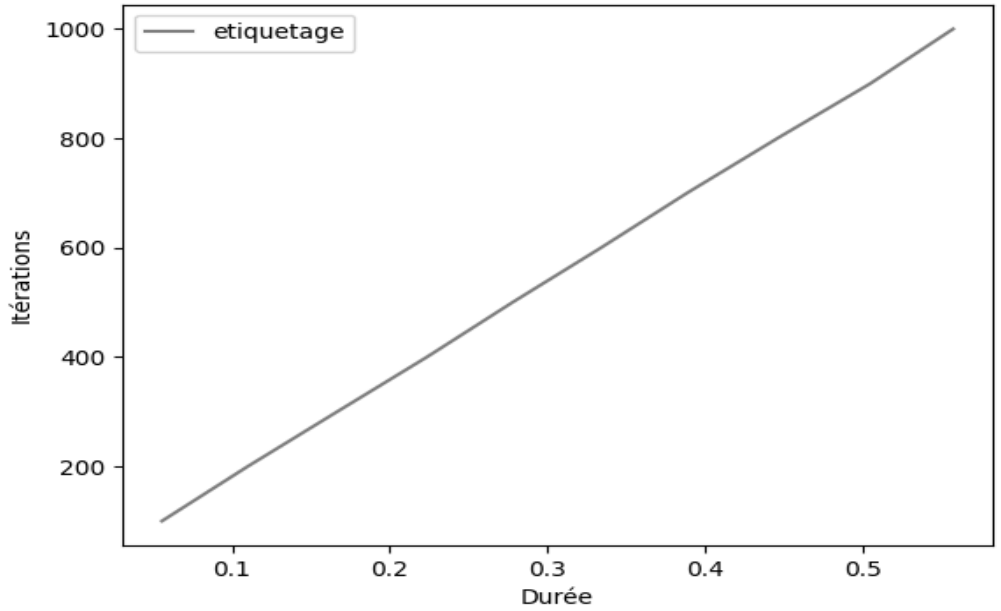
La fonction time_execution prend en argument une fonction (qui va renvoyer une liste de polynôme), son argument unique (ici une liste de polynôme), un nom de fichier et un booléen is_end.

Son rôle est d'exécuter la fonction passée en paramètre, de mesurer la durée d'exécution de cette fonction, puis de récupérer la taille de la liste renvoyée par cette fonction. Ces informations sont ensuite écrites dans le fichier spécifié. Si is_end vaut true, un saut de ligne est ajouté après chaque enregistrement. Enfin, la fonction retourne le résultat de l'exécution de la fonction passée en argument.

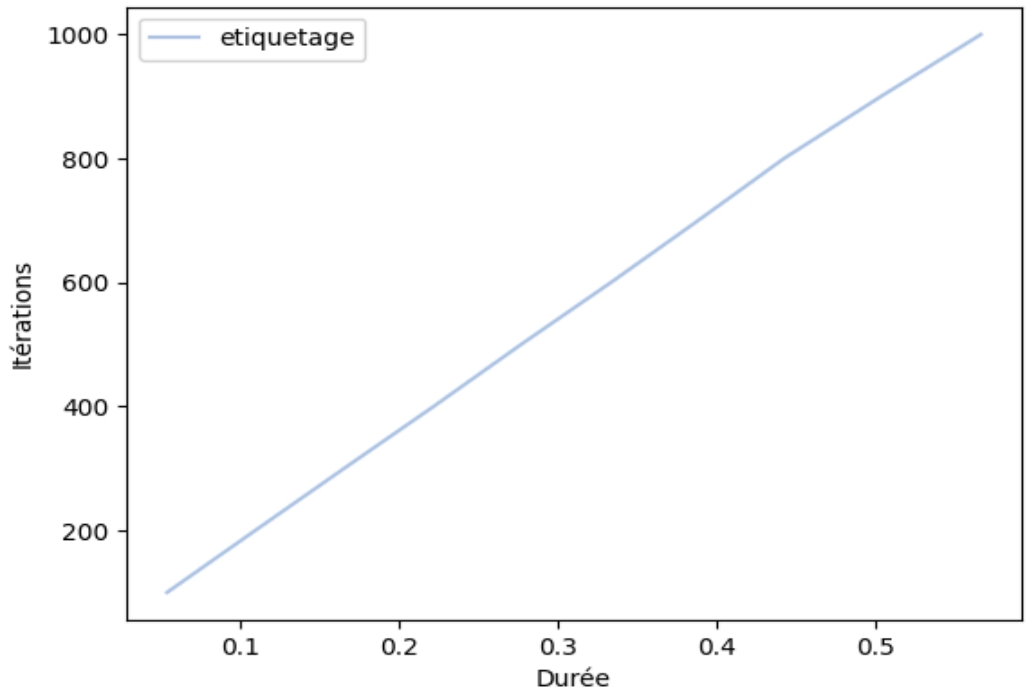
Graphe du temps d’exécution de exper_gen_abrs_20

Moyenne des graphes :

Ce graphique représente la moyenne de cinq essais réalisés avec la fonction exper_gen_abrs_20. Ces cinq essais, que l'on retrouve dans le code source, ont permis de générer les données utilisées pour ce graphique.



Ce graphique est le résultat d'une utilisation unique de la fonction exper_gen_abrs_20. Les autres logs sont retrouvables dans le dossier export.



On peut noter que le temps de calcul pour exper_gen_abrs_20, avec des valeurs de n variant de 100 à 1000, se situe dans l'ordre des dixièmes de seconde.

2.14 exp_somme

Afin de répondre à la question 2.14, nous avons mis en place trois stratégies différentes, basées sur poly_add ainsi que sur poly_add_canonique, qui applique la fonction canonique sur le résultat renvoyé par poly_add. Ces trois stratégies sont les suivantes :

2.14.1 Stratégie naïve récursive : exper_somme1

Simple implémentation d'une fonction récursive qui effectue la somme des polynômes contenus dans la liste l donné en argument en appelant poly_add_canonique.

2.14.2 Stratégie avec List.fold_left :exper_somme2

Simple implémentation d'une fonction qui utilise List.fold_left qui appelle la fonction poly_add_canonique sur chaque polynôme contenu dans la liste l donné en argument.

2.14.3 Stratégie naïve itérative : exper_somme3

- Initialise result qui est une variable mutable, liste vide qui contiendra le résultat des additions au fur et à mesure du déroulement de l'algorithme.
- Initialise remaining comme variable mutable à la liste l de polynômes donné en argument

- tant que la liste de polynôme l n’est pas vide (remaining), on utilise du pattern matching sur remaining et on additionne la tête de sa liste avec la liste result en appelant poly_add_canonique .

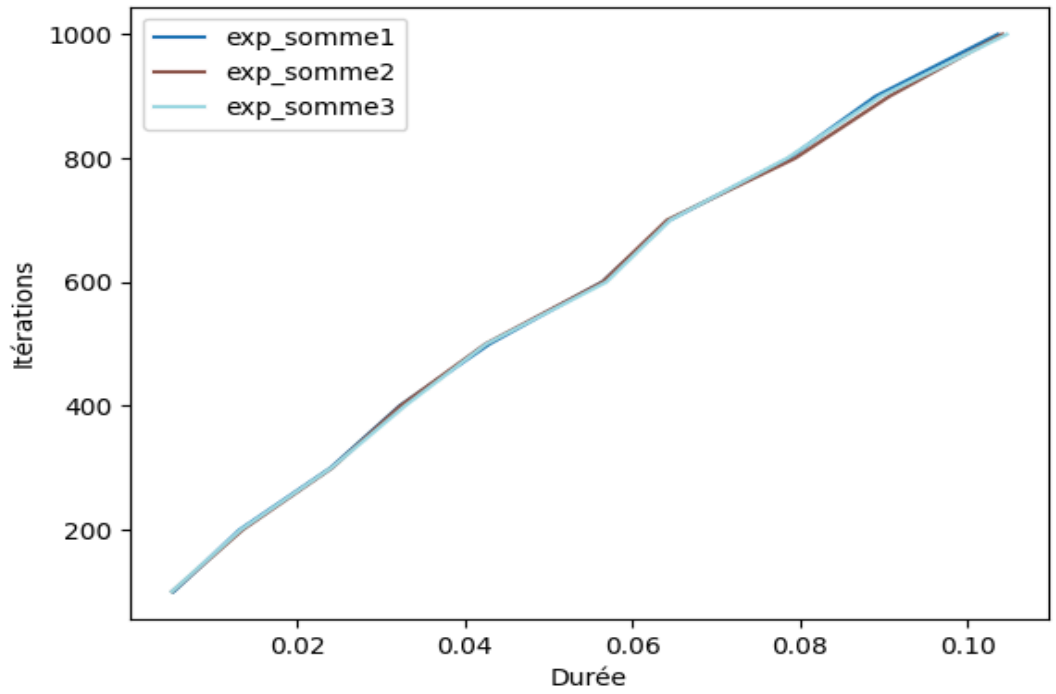
exp_somme

La fonction exp_somme prend en paramètre une liste de listes de polynômes, obtenue via la fonction exper_gen_abrs_20, qui génère les expressions à convertir en polynômes. Pour chaque élément de cette liste, exp_somme applique les stratégies de somme de polynômes présentées précédemment. Elle utilise également la fonction time_execution pour comparer les temps d’exécution des différentes stratégies. De plus, la fonction vérifie que les stratégies retournent les mêmes résultats et s'assure que le polynôme retourné n'est pas vide, afin d'éviter d'éventuelles erreurs.

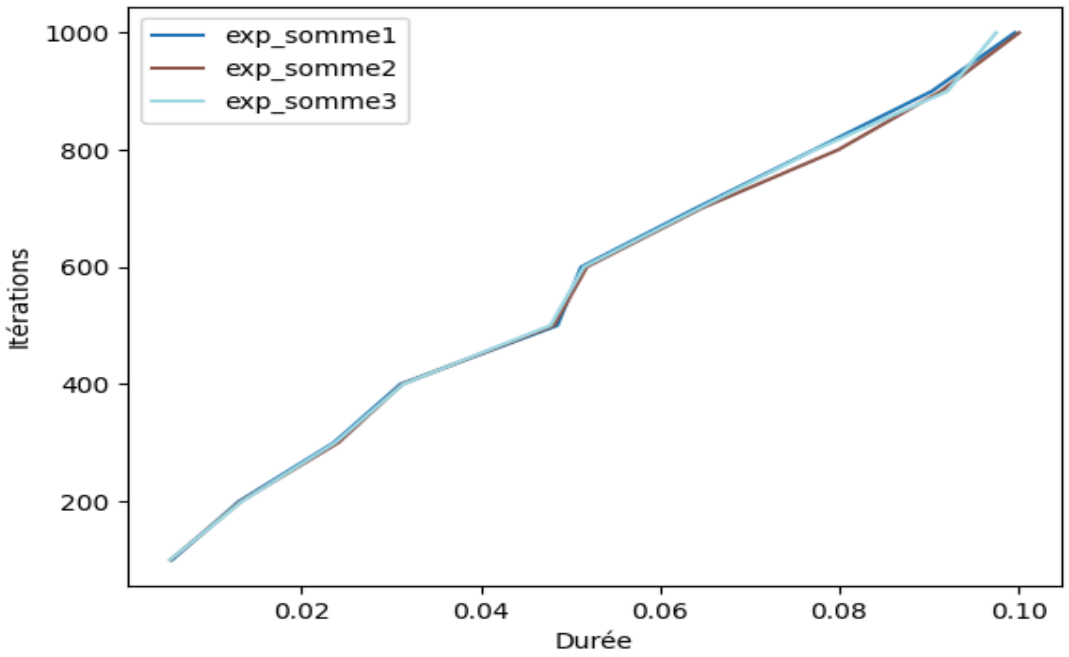
Graphe du temps d’exécution de exp_somme

Moyenne des graphes :

Ce graphe représente la moyenne de cinq graphes, chacun obtenu à l'issue de cinq appels distincts à la fonction exp_somme. Comme pour les autres graphes, vous pouvez les retrouver dans le code source.



Ce graphique est le résultat d'une utilisation unique de la fonction exp_somme.



On observe que les temps de calcul de la fonction exp_somme sont nettement plus rapides que ceux de exper_gen_abrs_20, qui se situent dans l'ordre de la centaine à la dizaine de secondes.

2.15 exp_produit

Dans cette section, nous avons mis en place quatre stratégies différentes pour le calcul du produit de polynômes. Ces stratégies reposent sur la fonction poly_prod_canonique, qui applique la forme canonique sur le résultat renvoyé par poly_prod. Voici un aperçu des stratégies :

2.15.1 Stratégie naïve récursive 1 : exper_produit1

Cette stratégie implémente une fonction récursive simple qui effectue le produit des polynômes contenus dans la liste l donnée en argument, en appelant poly_prod_canonique.

2.15.2 Stratégie naïve récursive 2 : exper_produit2

Cette stratégie est similaire à la précédente, mais avec une légère différence : après avoir effectué le produit des polynômes en appelant poly_prod, nous appliquons ensuite canonique sur le résultat pour obtenir un polynôme canonique.

2.15.3 Stratégie naïve itérative :exper_produit3

Cette stratégie est identique à la stratégie naïve itérative utilisée pour la somme, mais elle appelle poly_prod_canonique pour obtenir le résultat canonique du produit des polynômes.

2.15.4 Stratégie diviser pour régner : exper_produit4

Cette stratégie repose sur l'algorithme "diviser pour régner". Elle est implémentée à l'aide d'une fonction interne récursive divide_and_conquer, qui prend en paramètre une liste de polynômes, un entier début et un entier longueur. Voici le fonctionnement détaillé :

- Prend une liste de polynômes, un entier début et un entier longueur
- Si la longueur est à 0 alors la liste est vide, on renvoie la liste vide
- Si la longueur est à 1 alors il n'y a qu'un polynôme, on le renvoie
- On divise la longueur l par 2 est on la stock dans la variable milieu
- On appelle récursivement divide_and_conquer sur p1 qui est la première moitié de la liste (qui sera récursivement la moitié de la moitié ect).
- De même pour p2 sur l'autre moitié de la liste
- On finit par appeler canonique pour avoir un polynôme résultat sous la forme canonique
- Le premier appel est fait sur la liste l ,0 pour l'indice de début et List.length pour la longueur

exp_produit

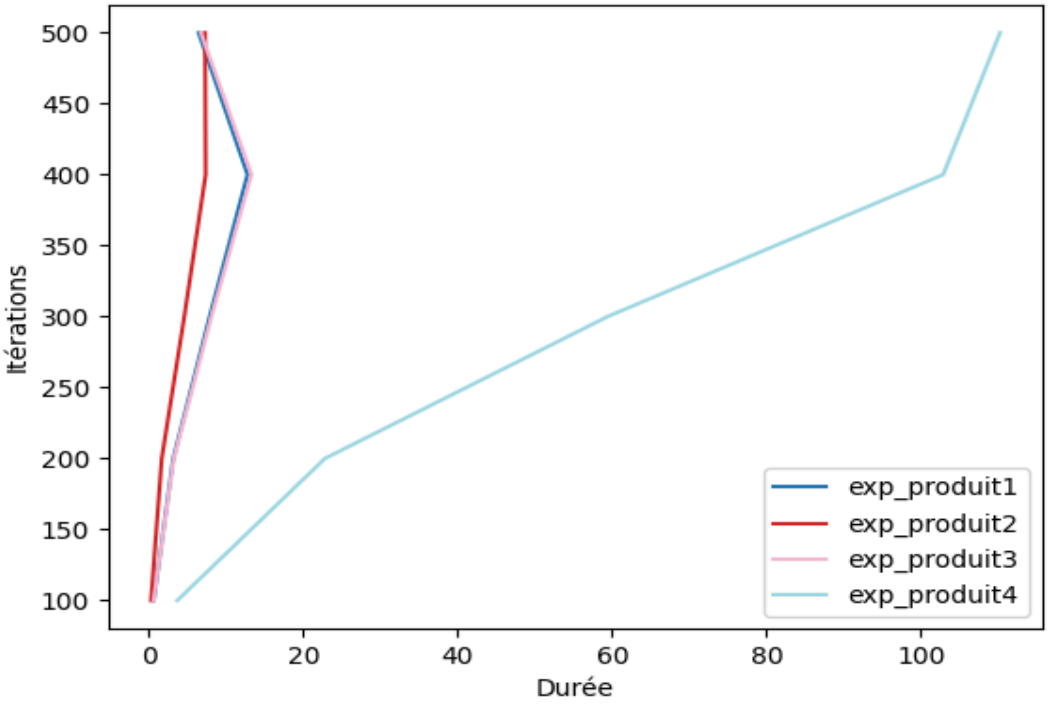
La fonction exp_produit fonctionne de manière similaire à la fonction exp_somme. Pour chaque élément de la liste de listes de polynômes en paramètre, elle utilise la fonction time_execution pour mesurer le temps d'exécution de chaque stratégie évoquée précédemment. Cela permet de comparer les temps d'exécution en fonction de la taille des éléments de la liste donnée en paramètre.

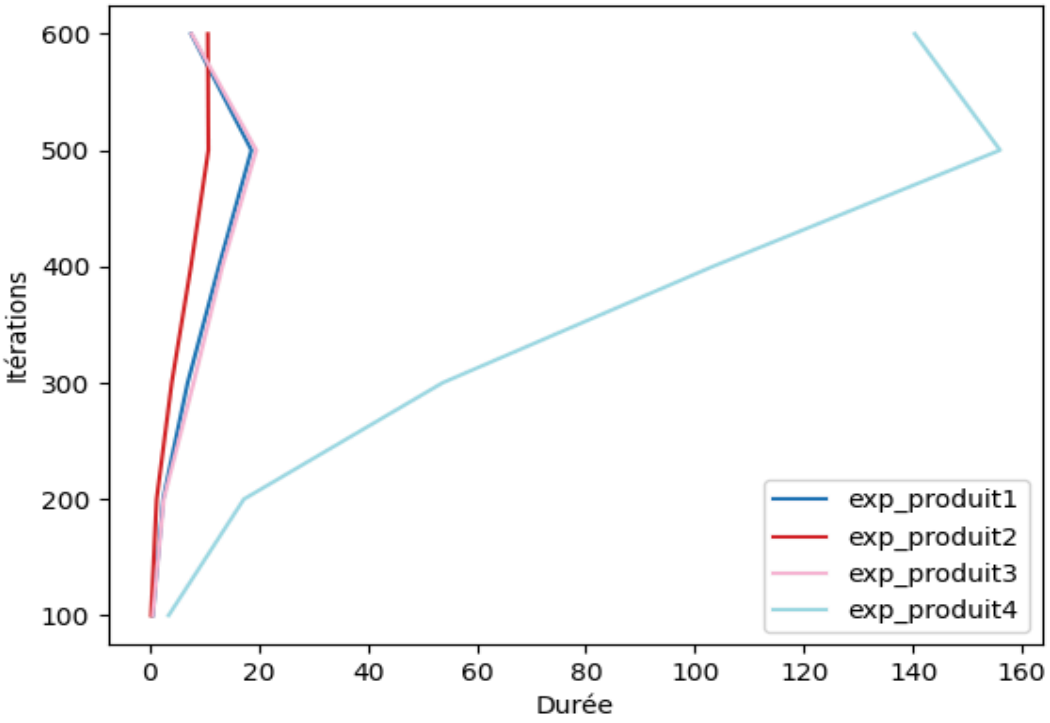
Cependant, en raison de problèmes liés à la taille limite des entiers en OCaml, nous rencontrons des difficultés pour représenter les temps de calcul lorsque n varie de 100 à 1000 avec un pas de 100. Ces limitations entraînent des difficultés dans la représentation des résultats et la génération des graphes de performance.

Graphe du temps d'exécution de exp_produit

En raison de ces problèmes (probablement dus à la limite de taille des entiers, ce qui provoque des listes vides dans exp_produit), nous ne sommes pas en mesure de représenter un graphe pour des valeurs de n allant de 100 à 1000. De plus, étant donné que les appels à exp_produit ne s'arrêtent pas nécessairement à la même itération, nous n'affichons pas une moyenne des graphes, mais plutôt deux graphes issus de différentes itérations.

Comme pour tous les autres graphes présentés précédemment, et ceux qui seront ajoutés par la suite, tous les résultats graphiques sont disponibles dans le code source.





On remarque que à l'instar des 2 autres exp_produit est bien plus lent que exp_somme, d'un facteur 1000 (au minimum vu qu'on a pas les valeurs pour n=1000).

2.16 exper_gen_abr_15

Notre fonction exper_gen_abr_15 qui réalise bien ce que l'énoncé avait demandé à un détail près (le détail étant que pour réaliser 15 abr avec comme dernier abr un abr de taille 2**13 il a fallu qu'on commence avec une puissance de moins -1 qu'on traitera comme une puissance de 0).

Elle se reposera aussi sur 2 fonctions internes pour produire son résultat. Ces deux fonctions internes sont nommées power_positif et aux.

power_positif

Cette fonction va tout simplement calculer la valeur de son paramètre i à la puissance n pour n positif (si n est négatif il sera traité comme une puissance 0).

aux

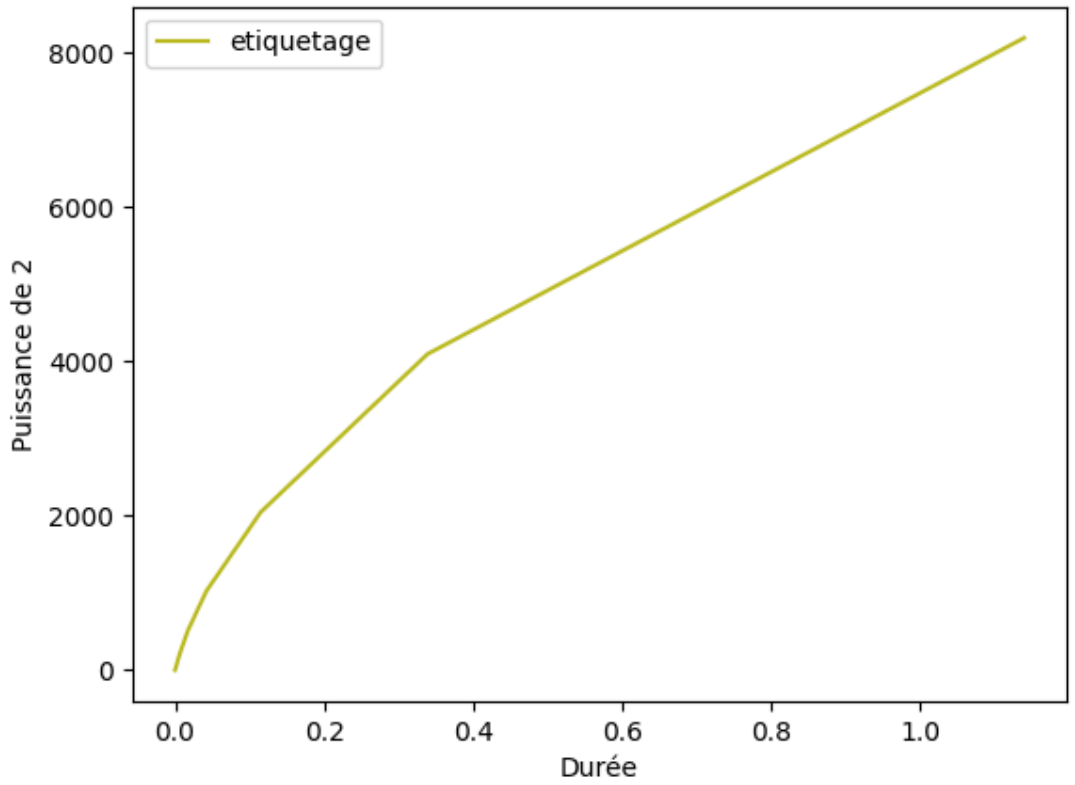
Cette fonction qui prend en paramètres des valeurs entières n et max (on suppose n<max) va utiliser exper_gen_abrs pour obtenir un abr avec comme taille la puissance de 2 calculée avec power_positif.

Ensuite, elle fera une récursion de n à max en incrémentant n de 1 en concaténant les listes obtenu de exper_gen_abrs, tout en écrivant dans un fichier exper_gen_abr_15.txt la durée de l'exécution de exper_gen_abrs à chaque récursion. Le résultat est ainsi une liste d'expression de taille 1,1,1,2,4,8...2**13.

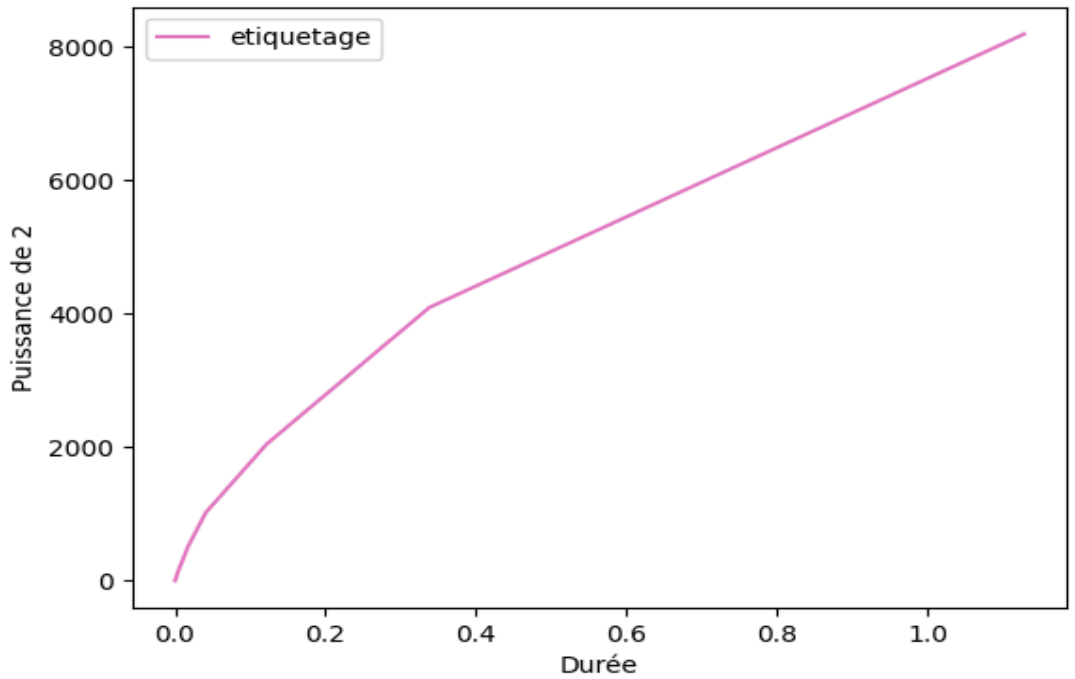
Graphe du temps d'exécution de exper_gen_abr_15

Moyenne des graphes :

Ce graphe est la moyenne de 5 graphes qu'on a réalisé suite à 5 essais de exper_gen_abr_15



et celui là est juste un graphe issue d'une utilisation de exper_gen_abr_15



On remarque malgré le fait que comparer à exper_gen_abrs_20 qui prend une liste de liste d'expression de taille 20 pour n allant de 100 à 1000 exper_gen_abr_15 prend plus de temps car elle a une taille qui est bien supérieure à celle mis dans exper_gen_abrs_20 malgré le fait que le nombre d'arbres ne change pas.

2.17 et 2.18

Ici ce qui reste à faire est de tout simplement appliquer la fonction exp_somme et exp_produit sur la liste d'expression obtenu avec exper_gen_abr_15 transformer au préalable en polynôme avec List.map et abr2poly comme ceci.Vous pouvez retrouver l'exemple dans notre code source.

Problèmes rencontrés lors des expérimentations

2.15 Polynôme canonique issu du produit des n arbres

Malgré toutes les optimisations effectuées, nous n'avons pas réussi à atteindre n=1000 pour le produit des n polynômes, notre fonction s'arrête bien avant ou se termine sans renvoyer de résultat.
Nous allons détailler tout ce que nous avons entrepris afin de trouver une solution.

- Nous avons fait des tests de n=100 à 1000 avec des pas de 100 et ça n'a pas fonctionné, nous avons donc essayé de commencer avec des valeurs plus hautes comme 400 et faire des pas de 1, de 20 pour pouvoir identifier ou l'erreur pourrait se produire mais la fonction ne dépasse jamais 500. De plus c'est assez aléatoire car nous arrivions à atteindre 500 pour plusieurs tentatives mais pas tous.
    - vous pourrez constater dans notre code source un fichier expert_produit_log_echec_500.txt qui montre de manière lisible l'échec du produit à n=500
- Nous avons essayé sur deux ordinateurs différents mais nous rencontrions les mêmes problèmes.

- Nous avons redéfini nos fonctions poly_prod et poly_add plusieurs fois, nous avons fait des tests avec les versions où canonique est appelé et les versions où canonique n'est pas appelé.

- Nous avons essayé plusieurs stratégie: algorithme de karatsuba qui était trop long à implémenter, il fallait modifier une grande partie de notre code source; algorithme diviser pour régner qui est plus lent que nos stratégie naïve, ce qui n'est pas normal. Algorithme récursif,avec accumulateur, mais rien ne fait. Nous avons fini par choisir celles qui étaient les plus optimales sur 500 polynômes.

- L'idée était d'appeler canonique après chaque produit calculé afin de s'assurer qu'il n'y a pas de doublons ni de terme nul ce qui nous permettait de limiter le nombre de multiplications inutiles. Ceci revient à additionner tous les termes généré par la multiplication qui ont le même degrés afin de diminuer la longueur du polynôme résultat qui sera multiplié avec le reste des polynômes dans la liste

- Nous nous heurtons au problème des représentations entières. En Ocaml, un entier ne peux pas dépasser -4611686018427387904/+4611686018427387903
  Nous soupçonnons cette limite comme étant une source potentielle d'erreur.

- Nous avons inséré quelque print au niveau des test de la fonction 2.13 lors de la génération des arbres et nous soupçonnons l'enchaînement de fonctions qui génère nos polynôme de générer au bout d'un certain n un polynôme nul (liste vide) qui sera multiplier avec les autres polynômes et annulera tous notre résultat.
  - Afin de vérifier notre théorie, nous avons inséré le test suivant dans poly_prod "si un terme est nul alors renvoyer l'autre terme" et le test fonctionne pour n= 1000 et nous avons bien un résultat qui s'affiche.
    Nous sommes ainsi sûr que le problème vient d'un terme nul généré quelque part lorsque n avoisine les 500 polynômes.
    Notre test nous permet de localiser le problème or nous ne pouvons pas le garder car le produit d'un terme nul doit renvoyer un terme nul et non le deuxième terme. Si nous gardons ce test dans la fonction poly_prod, elle sera fausse car elle ne respectera pas le principe du produit.
    Nous avons malheureusement découvert ce problème trop tard pour modifier le code source. Néanmoins modifier le code d'étiquetage ou de gen_abr aurait peut être corriger le problème.

Potentiels amélioration

Nous pourrions avoir une différente représentation de notre Big Integer en implantant par exemple cette librairie que nous avons trouvé en ligne : https://github.com/ocaml/Zarith. Par manque de temps, nous n'avons pas été en mesure de l'implémenter.

Remarques importantes concernant l'écriture de fichiers

- Dans le cadre des expérimentations, nous avons opté pour l'enregistrement des données dans des fichiers plutôt que pour l'affichage dans la console. Cette méthode présente plusieurs avantages : elle est plus pratique pour l'analyse des résultats, elle est plus lisible que l'affichage en console, et elle permet de conserver une trace persistante des exécutions.

Dans plusieurs fonctions où il est nécessaire d'écrire dans des fichiers, nous aurions pu envisager d'ouvrir et de fermer les fichiers à l'extérieur de ces fonctions (dans une fonction englobante) et de passer un argument de type out_channel aux fonctions concernées, en utilisant l'instruction Printf.fprintf sur cet argument. Cependant, en OCaml, lorsqu'on appelle Printf.fprintf, les données à écrire sont d'abord mises dans un tampon (buffer), et l'écriture réelle n'a lieu que lorsque le fichier est fermé avec l'instruction close_out file. Nous avons donc jugé plus judicieux d'ouvrir et de fermer le fichier à chaque appel de fonction, ce qui permet d'effectuer des écritures fréquentes et de suivre l'évolution des fonctions (qui peuvent parfois être très longues à exécuter).

- Par défaut, l'ouverture d'un fichier en OCaml a pour effet de créer un nouveau fichier (et donc de remplacer le fichier existant si celui-ci existe). Pour éviter ce comportement, nous utilisons l'instruction open_out_gen [Open_creat; Open_append; Open_text] 0o666 "fichier.txt". Cela permet d'ouvrir un fichier en mode lecture, d'ajouter du contenu sans supprimer ce qui est déjà présent et de garantir l'écriture à la suite du contenu existant, ce qui est indispensable dans le cadre d'ouvertures/fermetures successives.