Sorbonne Université

Projet STL
Manipulation des Structures Linéaires et Arborescentes en Fonctionnel

Auteur :
Selma BELKADI
Kheireddine OUNNOUGHI
Ryan TAUCH

Référent :
Antoine Genitrini
Vincent Botbol

8 décembre 2024

Résumé

Ce projet consiste à manipuler et analyser deux modèles de structures de données : linéaires et arborescentes, en utilisant le langage de programmation fonctionnelle OCaml. La première partie du travail se concentre sur la représentation, l'ajout et la multiplication de polynômes sous forme linéaire. La deuxième partie porte sur la définition et la manipulation d'expressions arborescentes représentant des opérations sur des polynômes, suivie de leur transformation en polynômes canoniques. L'objectif est d'étudier les performances des différentes stratégies proposées pour chaque opération, avec une analyse de la complexité et une évaluation expérimentale des temps de calcul.

Mots-clés : OCaml, structures de données linéaires, structures arborescentes, polynômes, multiplication de polynômes, analyse de complexité, programmation fonctionnelle.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Présentation du projet

Le but de ce devoir est de manipuler deux modèles de structure de données : l'une sous forme linéaire et l'autre arborescente, et d'implémenter des algorithmes permettant de réaliser diverses opérations (addition et multiplication) sur ces structures, afin de comparer les performances de chaque solution proposée.

1.1 Sujet

Ce projet traite la manipulation de structures de données en OCaml, en utilisant le paradigme fonctionnel. Il vise à comparer deux modèles principaux :

— Les structures linéaires, représentées par des listes triées pour manipuler des polynômes sous forme canonique.

— Les structures arborescentes, utilisées pour représenter des expressions arborescentes selon une grammaire spécifique.

L'objectif est de développer et d'implémenter des algorithmes pour effectuer des opérations telles que l'addition et la multiplication sur ces structures. Une analyse expérimentale des performances permettra d'évaluer l'efficacité des solutions proposées.

1.2 Problématique soulevée

Dans le cadre de ce projet, plusieurs défis ont été identifiés. L'objectif principal est d'évaluer les performances et l'efficacité des structures de données et des algorithmes développés pour la manipulation de polynômes, notamment :

— Comment représenter efficacement les polynômes sous différentes structures (linéaire et arborescente) tout en facilitant les opérations comme l'addition et la multiplication ?

— Comment implémenter des méthodes efficaces pour ces opérations ?

1.3 Hypothèse de solution

Pour résoudre les problématiques identifiées, plusieurs hypothèses ont été formulées sur l'efficacité et la pertinence des structures et fonctions utilisées. Ces hypothèses incluent :

— Une implémentation de la représentation linéaire des polynômes simplifie les opérations d'addition et de multiplication grâce à une structure itérative.

— La fonction `canonique` simplifie les polynômes, réduisant les doublons et regroupant les termes similaires.

Chapitre 2

Polynôme sous forme linéaire

2.1 Introduction

Dans cette partie, nous considérons la représentation suivante des polynômes en la variable formelle x. Le monôme $c \cdot x^d$ est donné par le couple $(c, d) \in \mathbb{Z} \times \mathbb{N}$. Un polynôme est une liste de monômes. d est donné par le couple (c, d) ∈ Z × N. Un polynôme est une liste de monômes.

2.2 Structure de données : Linéaire

Voici la représentation en OCaml de cette structure de données :

```
type monome = Mono of int * int;;
type polynome = monome list;;
```

— le type `monome` représente un couple (c, d), où c est le coefficient et d est le degré.
— Le type `polynome` est une liste de `monome`, permettant de stocker et manipuler des polynômes sous forme linéaire.

2.3 Implémentation de `canonique`

2.3.1 Principe

Étapes clés de la fonction `canonique` :

1. **Regroupement des monômes :** Parcourir la liste des monômes pour identifier ceux qui ont le même degré
2. **Somme des coefficients :** Additionner les coefficients de chaque groupe de monômes ayant le même degré.
3. **Élimination des monômes nuls :** Supprimer les monômes dont le coefficient est nul.
4. **Tri des monômes :** Ordonner la liste des monômes par degré croissant.

2.3.2 Code

```
let canonique (p:polynome) : polynome =
  (* aux : take as argument a polynome sorted by degree
  and combine the monomes of same degree *)
  let rec aux p =
    match p with
    | [] -> []
    | [Mono (coeff, degre)] -> if coeff = 0 then [] else [Mono (coeff, degre)]
    | Mono (coeff1, degre1) :: Mono (coeff2, degre2) :: t ->
      if degre1 = degre2 then
        (* if sum of coeff is null -> remove both *)
        if (coeff1+coeff2) = 0 then aux t
        (* else combine the monomes *)
        else aux (Mono(coeff1 + coeff2, degre1) :: t)

      (* remove monome if coeff = null *)
      else if coeff1 = 0 then aux (Mono(coeff2, degre2) :: t)
      else if coeff2 = 0 then aux (Mono(coeff1, degre1) :: t)
      (* nothing to change *)
      else Mono (coeff1, degre1) :: aux (Mono (coeff2, degre2) :: t)
  in
  aux (List.sort (fun (Mono (_, degre1)) (Mono (_, degre2)) -> compare degre1 degre2) p)
```

2.3.3 Analyse de complexité

2.3.3.1 Mesure de complexité

Pour analyser la complexité, nous avons choisi comme mesure le nombre de monômes dans le polynôme (la taille de la liste qui est donnée comme paramètre), noté n. Cette mesure est pertinente car toutes les étapes de la fonction dépendent de ce nombre.

2.3.3.2 Analyse des étapes

Dans la fonction, chaque paire de monômes est comparée avec des conditions (`if`) pour vérifier :
— Si les degrés sont identiques (pour les combiner).
— Si le coefficient est nul (pour les supprimer).
Ces vérifications sont effectuées en temps constant $O(1)$ pour chaque paire.

Le tri des monômes par degrés est réalisé avec `List.sort`, qui implémente le tri fusion (`merge sort`) [1]

2.3.3.3 Pire cas

Le pire cas se produit lorsque le polynôme est déjà sous sa forme canonique et trié. Dans ce cas :
— Chaque comparaison se fait sans modification $\longrightarrow O(n)$
— Le tri est effectué dans la pire configuration (liste déjà triée) $\longrightarrow O(n \log n)$
La complexité est dominé par le tri, ce qui donne $O(n \log n)$ comme complexité au pire cas.

2.3.3.4 Conclusion

Malgré le fait que le pire cas puisse subvenir théoriquement, il reste rare en pratique. Donc, la complexité globale de `canonique` reste $O(n)$.

1. D'après la documentation officielle d'OCaml

2.4 Implémentation de `poly_add`

2.4.1 Principe

Étapes clés de la fonction `poly_add` :

1. **Cas de base :** Si l'un des polynômes est vide, la fonction renvoie l'autre, car il n'y a rien a ajouter
2. **Comparaison des degrés :** Si les 2 monômes ont le même degré, alors :
   — Si la somme de leurs coefficients est nulle, ils sont supprimés.
   — Sinon, nous additionnons les 2 coefficients.
   — Si le degré du premier monôme est plus petit, ce monôme est ajouté tel quel.
3. **Cas général :** Si les degrés des 2 monômes sont différents, la fonction ajoute le monôme dont le degré est le plus petit et continue avec les reste des monômes.

2.4.2 Code

```
let rec poly_add (p1: polynome) (p2: polynome) : polynome =
  let rec aux p1 p2 =
    match p1 with
    | [] -> p2
    | Mono(coeff1, degre1) :: t1 ->
      match p2 with
      | [] -> p1
      | Mono(coeff2, degre2) :: t2 ->
        if degre1 = degre2 then
          if (coeff1 + coeff2) = 0 then aux t1 t2
          else Mono(coeff1 + coeff2, degre1) :: aux t1 t2
        else if degre1 < degre2 then
          Mono(coeff1, degre1) :: aux t1 p2
        else
          Mono(coeff2, degre2) :: aux p1 t2
  in aux p1 p2;;
```

2.4.3 Analyse de complexité

2.4.3.1 Mesure de complexité

La mesure de complexité choisie pour analyser la fonction `poly_add` est le nombre de comparaison entre les polynômes.

2.4.3.2 Analyse des étapes

La fonction `poly_add` a 3 étapes principales :

1. Comparaison des termes $\longrightarrow O(1)$.
2. Addition des coefficients $\longrightarrow O(1)$.
3. Insertion dans la liste $\longrightarrow O(1)$.

La complexité de la fonction `poly_add` dans ce cas est $O(n+m)$, où n et m représentent respectivement le nombre de termes dans les deux polynômes. Cela veut dire que la fonction parcourt chaque terme des deux polynomes.

2.4.3.3 Pire cas

En effet, le pire cas se produit lorsque chaque élément du polynôme ont des degrés différents. On va donc parcourir entièrement les deux polynomes. La complexité en pire cas est donc en O(n+m).

2.4.3.4 Conclusion

En résumé, la complexité de la fonction `poly_add` dépend du nombre de termes dans les polynômes. Dans la plupart des cas, elle est de l'ordre de la taille des deux polynômes, soit $O(n+m)$.

2.5 Implémentation de `poly_prod`

2.5.1 Principe

La fonction `poly_prod` multiplie deux polynômes en suivant ces étapes :

1. **Multiplication des monômes :** La fonction `aux` multiplie chaque monôme de `p1` avec tous les monômes de `p2`.
2. **Addition des produits :** La fonction `aux2` ajoute les produits de chaque monôme de `p1` avec tous les monômes de `p2`.
3. **Simplification :** Enfin, la fonction `canonique` simplifie les termes résultants en combinant les monômes de même degré et supprimant les termes nuls.

2.5.2 Code

```
let poly_prod (p1:polynome) (p2:polynome) : polynome =
  let rec aux (Mono (coeff1, degre1)) p =
    match p with
    | [] -> []
    | Mono (coeff2, degre2) :: t ->
      Mono (coeff1 * coeff2, degre1 + degre2) :: aux (Mono (coeff1, degre1)) t
  in
  let rec aux2 p1 p2 =
    match p1 with
    | [] -> []
    | h :: t -> poly_add (aux h p2) (aux2 t p2)
  in canonique (aux2 p1 p2);;
```

2.5.3 Analyse de complexité

2.5.3.1 Mesure de complexité

La mesure de complexité utilisée est le nombre d'opérations de monômes entre les deux polynômes, selon la taille des polynomes n et m. Cette mesure est pertinente car toutes les étapes de la fonction dépendent de la taille des polynômes.

2.5.3.2 Analyse des étapes

1. **Multiplication des monômes :** La fonction `aux` multiplie chaque monôme de `p1` (de taille n) avec chaque monôme de `p2` (de taille m). La complexité de cette opération est $O(n \times m)$, car chaque monôme de $p1$ est multiplié par chaque monôme de $p2$.
2. **Addition des produits :** La fonction `aux2` parcourt tous les monômes de `p1`(de taille n) et de `p2` (de taille m) et applique la multiplication et fait l'addition du résultat des produits.
3. **Simplification :** La fonction `canonique` simplifie les termes après la multiplication.

En cas général, la complexité est de $O(n \times m)$.

2.5.3.3 Pire cas

La pire configuration possible serait que chaque produit donne un monôme distinct, ce qui signifie qu'aucune simplification ne peut se faire. Dans ce cas, tous les monômes devront être simplifié via l'appel de `canonique`.

2.5.3.4 Conclusion

En conclusion, la complexité de la multiplication des polynômes est généralement $O(n \times m)$, où n et m sont les tailles des deux polynômes.

Chapitre 3

Expression arborescente

3.1 Introduction

Dans cette partie, nous allons définir des expressions arborescentes qui représentent des opérations simples sur des polynômes en la variable x. Voici la grammaire qu'elles vérifient :

$$
\begin{aligned}
E &= \text{int} \mid E^{\wedge} \mid E^{+} \mid E^{*} \\
E^{\wedge} &= x\text{^int}^{+} \\
E^{+} &= (E \setminus E^{+}) + (E \setminus E^{+}) + \dots \\
E^{*} &= (E \setminus E^{*}) \cdot (E \setminus E^{*}) \cdot \dots
\end{aligned}
$$

Nous remarquons en particulier :
— **int** représente un entier, x la variable formelle, et $\textbf{int}^{+}$ un entier strictement positif ;
— Les expressions de la sous-famille $E^{\wedge}$ représentent les puissances de la variable x ;
— Les expressions de la sous-famille E^{+} sont des sommes d'au moins deux expressions dont les opérateurs principaux ne sont pas des sommes ;
— Les expressions de la sous-famille E^{*} sont des produits d'au moins deux expressions dont les opérateurs principaux ne sont pas des produits.

3.2 Structure de données : Arborescente

Voici la représentation en OCaml de cette structure de données :

```
type expr =
  | Int of int
  | Var of string
  | Pow of expr list
  | Add of expr list
  | Mul of expr list;;
```

Dans ce code, nous définissons un type appelé `expr` pour représenter les noeuds d'un arbre d'expressions mathématiques. Chaque noeud peut être de différents types, comme suit :
— `Int` : Un nombre entier. Par exemple, `Int(3)` représente le nombre 3.
— `Var` : Une variable. Par exemple, `Var("x")` représente la variable x.
— `Pow` : Une puissance.. Par exemple, `Pow([Var("x"), Int(2)])` représente x^2.
— `Add` : Une addition. Par exemple, `Add([Int(3), Int(4)])` représente $3 + 4$.
— `Mul` : Une multiplication. Par exemple, `Mul([Int(3), Var("x")])` représente $3 * x$.

3.3 Exemple : Représentation d'un arbre

Dans cet exemple, nous montrons comment un polynôme peut être représenté par notre structure implémentée. Chaque nœud de l'arbre représente un opérateur ou un opérande, et les fils correspondent aux sous-expressions qui composent le polynôme. Nous allons illustrer cette représentation arborescente à l'aide d'une liste d'expressions.

La figure 3.1 ci-dessous représente un arbre qui illuste un polynôme. Les nœuds de l'arbre sont des opérations (telles que la multiplication et la puissance), et les valeurs des nœuds représentent des entiers ou des variables.

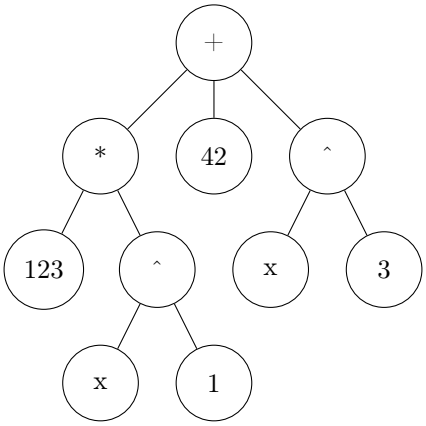

FIGURE 3.1 – Représentation du polynôme $123 \cdot x + 42 + x^3$ sous forme d'arbre

Cet arbre peut être représenté sous forme d'une liste d'expressions comme suit :

```
let expr_tree =
  Add [
    Mul [Int 123; Pow [Var "x";Int 1]];
    Int 42;
    Pow [Var "x";Int 3]
  ];;
```

La structure utilisée pour représenter l'arbre dans cet exemple se base sur une liste imbriquée d'expressions. Chaque élément de la liste représente soit un opérateur, soit un opérande. Cette hiérarchie permet de montrer l'organisation des opérations.

0. L'arbre 3.1 est extrait du sujet du devoir (arbre de gauche dans la Figure 1).

3.4 Implémentation de `arb2poly`

3.4.1 Principe

1. `Int n` :
   — Si c'est un noeud entier `n`, il est converti en un monôme constant `Mono(n, 0)`.
2. `Var x` :
   — Si c'est un noeud variable `x`, il est converti en un monôme `Mono(1, 1)`.
3. `Pow [base; Int n]` :
   — Si c'est un noeud puissance où la base est une expression et l'exposant est un noeud entier `n`, le résultat est un monôme `Mono(1, n)`.
4. `Add children` :
   — Si c'est un noeud addition de plusieurs sous-arbres (`children`), la fonction :
      — Parcourt chaque noeud enfant pour le convertir en monôme.
      — La combinaison se fait avec une fonction canonique pour simplifier le résultat.
5. `Mul children` :
   — Si c'est un noeud multiplication de plusieurs sous-arbres (`children`), la fonction :
      — Parcourt chaque noeud enfant pour le convertir en monôme.
      — Multiplie chaque terme à l'aide de la fonction `poly_prod`.
6. `_ (cas par défaut)` :
   — Renvoie une liste vide.

3.4.2 Code

```
let rec arb2poly expr_tree =
  match expr_tree with
  | Int n -> [Mono(n, 0)]
  | Var x -> [Mono(1, 1)]
  | Pow [base; Int n] -> if base = Var "x" then [Mono(1, n)] else failwith "only x as var"
  | Add children ->
      let rec add_polys kids acc =
        match kids with
        | [] -> acc
        | h :: t ->
            let poly_hd = arb2poly h in
            add_polys t (poly_hd @ acc)
      in
      canonique (add_polys children [])
  | Mul children ->
      let rec prod_polys kids =
        match kids with
        | [] -> [Mono(1, 0)]
        | h :: t ->
            let poly_hd = arb2poly h in
            poly_prod poly_hd (prod_polys t)
      in
      prod_polys children
  | _ -> [];;
```

Chapitre 4

Synthèse d'expressions arborescentes

4.1 Introduction

Cette partie permet de construire un générateur d'expressions arborescentes. La première étape consiste à construire la structure arborescente, la deuxième étape s'attache à étiqueter la structure suivant des règles peu contraintes, et enfin la troisième partie consiste à modifier localement l'arbre étiqueté afin qu'il satisfasse la grammaire décrite dans la section précédente.

4.2 Implémentation `extraction_alea`

4.2.1 Principe

1. Tirer un index aléatoire dans la liste de monômes.
2. Récuperer l'élément à cet index.
3. Supprimer cet élément de la liste de monômes.
4. Ajouter cet élément à la liste de retour.

4.2.2 Code

```
let extraction_alea list_L list_P =
  let rec remove_element n l =
    match l with
    | [] -> []
    | h::t when n = 0 -> t
    | h::t -> h :: remove_element (n - 1) t
  in
  if List.length list_L = 0
    then ([], list_P)
  else
    let random_int l = Random.int (List.length l) in
    let r = random_int list_L in
    let elt = List.nth list_L r in
    (remove_element r list_L, elt::list_P);;
```

4.3 Implémentation `gen_permutation`

4.3.1 Principe

1. Générer une liste d'entiers allant de 1 à n avec un ordre croissant.
2. Tant que la liste initiale n'est pas vide.
   — Extraire un élément aléatoire de la liste initiale.
   — Ajouter cet élément à l'autre liste.
3. Répéter jusqu'à ce que tous les éléments soient transférés.
4. Retourner la liste finale contenant des permutations.

4.3.2 Code

```
let gen_permutation n =
  let rec generate_list upperBound count = match count with
    | _ when count = upperBound -> [count]
    | _ -> count :: generate_list upperBound (count+1)
  in
  let rec fill_list (l, p) =
    if List.length l = 0 then p
    else fill_list (extraction_alea l p)
  in fill_list (generate_list n 1,[]);;
```

4.4 Implémentation `abr`

4.4.1 Principe

Le type `'a btree` a été d'abord définit pour représenter les arbres binaires. La fonction `abr` permet d'ajouter un élément dans l'arbre. Voici les étapes :

1. Si l'arbre est vide `(Empty)`, alors un nouveau noeud est créé.
2. Si l'abre n'est pas vide, l'élément est comparé à la racine :
   — Si l'élément est plus petit que la racine, alors il est inséré dans le sous-arbre gauche.
   — Si l'élément est plus grand ou égal à la racine, alors il est inséré dans le sous-arbre droit.

4.4.2 Code

```
type 'a btree =
  | Empty
  | Node of 'a btree * 'a * 'a btree;;
```

```
let abr list =
  let rec insertABR elt tree =
    match tree with
    | Empty -> Node(Empty, elt, Empty)
    | Node (left, root, right) when elt < root -> Node(insertABR elt left, root, right)
    | Node (left, root, right) -> Node(left, root, insertABR elt right)
  in
  let rec buildABR l tree =
    match l with
    | [] -> tree
    | h::t -> buildABR t (insertABR h tree)
  in buildABR list Empty;;
```

Chapitre 5

Expérimentations

5.1 Génération n ABR de taille 20

Dans cette section, nous décrivons le processus de génération des arbres binaires ainsi que leur transformation en polynômes à l'aide des fonctions développées. Un arbre binaire.

Dans cette section, nous simplifions l'exemple pour dessiner un arbre binaire de taille 12. L'arbre binaire que nous allons utiliser représente un polynôme, où chaque nœud est une opération ou une valeur, en suivant la structure d'un arbre binaire.

Voici l'arbre binaire de taille 12 :

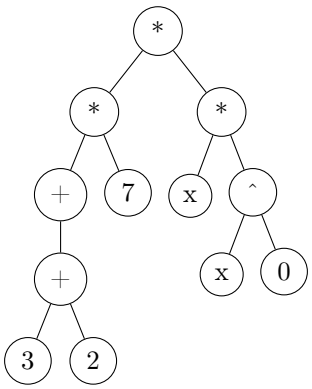

FIGURE 5.1 – Arbre binaire représentant un polynôme de taille 12

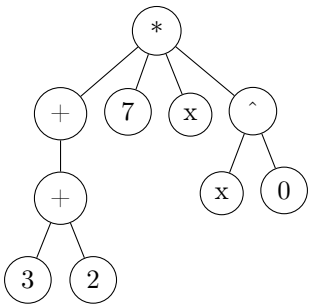

FIGURE 5.2 – Transformation de l'arbre

5.2 Stratégies d'addition

5.2.1 Méthode 1 : naif

La **méthode naïve** consiste à additionner les polynômes un par un. Pour cela, nous commençons par transformer chaque arbre de la liste en un polynôme à l'aide de la fonction `arb2poly` (décrite dans la section 4.4). Cette étape crée une nouvelle liste où chaque élément est un polynôme de l'arbre d'origine. Ensuite, nous

additionnons tous les polynômes de cette liste, de façon séquentielle : ajouter le premier polynôme avec le deuxième, puis le résultat avec le troisième, et ainsi de suite, jusqu'à ce qu'il ne reste qu'un seul polynôme. Cette approche est simple à implémenter, mais elle peut être inefficace lorsque la liste de polynômes est longue, car chaque addition est effectuée individuellement, ce qui augmente le temps de calcul global.

Voici un aperçu de la fonction associée à cette méthode, `add_n_tree_to_poly_naive` :

```
let add_n_tree_to_poly_naive l =
  let rec list2polyList treeList =
    match treeList with
    | [] -> []
    | h::t -> (arb2poly h) :: (list2polyList t)
  in
  let rec sumPolyList treeList =
    match treeList with
    | [] -> []
    | [h] -> h
    | h::t -> poly_add h (sumPolyList t)
  in sumPolyList (list2polyList l);;
```

5.2.2 Méthode 2 : fusion

La **méthode fusion** additionne plusieurs arbres polynômiaux d'une façon plus concise. Tous les arbres de la liste sont regroupés dans un seul nœud `Add`. Ensuite, la fonction `arb2poly` (décrite dans la section 4.4) transforme cet arbre en un polynôme. Cette fonction parcourt l'arbre et applique des règles simples : Pour les additions, elle convertit les sous-arbres en polynômes et les combine en simplifiant les termes.

Voici un aperçu de la fonction associée à cette méthode, `add_n_tree_to_poly_fusion` :

```
let add_n_tree_to_poly_fusion l = arb2poly (Add l);;
```

5.2.3 Méthode 3 : diviser pour régner

Cette méthode combine une liste d'arbres en un polynôme en utilisant une approche **divide and conquer** (diviser pour régner). Elle divise la liste d'arbres en deux sous-listes à chaque étape avec la fonction `split`. Si la liste est vide, elle renvoie une liste vide. Si elle contient un seul arbre, elle le convertit en polynôme avec `arb2poly` (décrite dans la section 4.4). Sinon, elle calcule les polynômes des deux moitiés récursivement, puis les additionne avec `poly_add` (décrite dans la section 2.4).

Voici un aperçu de la fonction associée à cette méthode, `add_n_tree_to_poly_divide` :

```
let add_n_tree_to_poly_divide l =
  let rec divide_and_conquer trees =
    let rec split lst =
      let rec aux l1 l2 count =
        match l2 with
        | [] -> (List.rev l1, [])
        | h :: t ->
            if count = 0 then (List.rev l1, l2)
            else aux (h :: l1) t (count - 1)
      in
      aux [] lst (List.length lst / 2)
    in
    match trees with
    | [] -> []
    | [h] -> arb2poly h
    | _ -> let (left, right) = split trees
       in poly_add (divide_and_conquer left) (divide_and_conquer right)
  in divide_and_conquer l;;
```

5.3 Stratégies de multiplication

5.3.1 Méthode 1 : naïve

La méthode de multiplication naïve pour le calcul du produit de polynômes représente une approche simple. Cette méthode commence par convertir chaque arbre en une liste de polynômes, grâce à la fonction `list2polyList`. La multiplication se fait de manière successive, ce qui entraîne une complexité importante.

```
let prod_n_tree_to_poly_naive l =
    let rec list2polyList treeList =
        match treeList with
            | [] -> []
            | h::t -> (arb2poly h) :: (list2polyList t)
    in
        let rec prodPolyList treeList =
            match treeList with
                | [] -> []
                | [h] -> h
                | h::t -> poly_prod h (prodPolyList t)
in prodPolyList (list2polyList l);;
```

5.3.2 Méthode 2 : fusion

La **méthode fusion** combine les expressions des arbres en les multipliant. Tous les arbres de la liste sont d'abord réunis sous un seul nœud `Mul`. Ensuite, la fonction `arb2poly` (décrite dans la section 4.4) est utilisée pour convertir cet arbre en un polynôme. La fonction parcourt chaque sous-arbre et applique les règles correspondantes pour effectuer la multiplication des polynômes.

Voici un aperçu de la fonction associée à cette méthode, `prod_n_tree_to_poly_fusion` :

```
let prod_n_tree_to_poly_fusion l = arb2poly (Mul l);;
```

5.3.3 Méthode 3 : diviser pour régner naïf

Cette méthode a pour idée principale de diviser la liste des arbres en sous-listes plus petites et de calculer le produit des polynômes pour chaque sous-liste, en combinant les résultats à la fin.

La fonction `divide_and_conquer` prend en entrée une liste d'arbres et deux indices, `left` et `right`, qui indiquent les limites de la partie de la liste.

Si cette partie a qu'un seul arbre, alors la fonction applique `arb2poly` pour obtenir son polynôme équivalent. Sinon la liste est divisé en 2 et la fonction est appelée de manière récursive sur les deux sous-listes. Les éléments des sous-listes sont multipliés avec `poly_prod`.

```
let prod_n_tree_to_poly_divide l =
  let rec divide_and_conquer lst left right =
    if right - left <= 1 then
      arb2poly (List.nth lst left)
    else
      let mid = (left + right) / 2 in
      poly_prod
        (divide_and_conquer lst left mid)
        (divide_and_conquer lst mid right)
  in
  divide_and_conquer l 0 (List.length l)
;;
```

5.3.4 Méthode 4 : algorithme de Karatsuba

La fonction `mult_karatsuba` utilise une méthode optimisé de multiplication de polynômes appelés *Karatsuba* [2]. Cette méthode consiste à diviser les polynômes en deux parties, pour réduire le nombre d'opération nécessaires pour faire le produit.

5.3.4.1 Principe

Soit deux polynômes $A(x)$ et $B(x)$ de degré n :

$$A(x) = A_1 x^{n/2} + A_0$$
$$B(x) = B_1 x^{n/2} + B_0$$

où A_1, A_0, ainsi que B_1 et B_0 sont des polynômes de degré $n/2$.
L'algorithme de *Karatsuba* utilise la formule suivante pour calculer le produit :

$$A(x) \times B(x) = A_1 B_1 x^n + (A_1 + A_0)(B_1 + B_0)x^{n/2} + A_0 B_0$$

Au lieu de calculer directement les produits $A_1 \times B_1$, $A_0 \times B_0$ et $(A_1 + A_0)(B_1 + B_0)$, l'algorithme de *Karatsuba* permet de réduire le nombre d'opérations en calculant que 3 produits :

$$P_1 = A_1 \times B_1$$
$$P_2 = A_0 \times B_0$$
$$P_3 = (A_1 + A_0) \times (B_1 + B_0)$$

Cela permet de réduire le nombre de multiplications de quatre à trois, ce qui améliore la complexité de l'algorithme.

5.3.4.2 Exemple de déroulement

Soit deux polynômes $A(x)$ et $B(x)$ de degré $n = 3$:

$$A(x) = 4x^3 + 3x^2 + 2x + 1$$
$$B(x) = x^3 + 2x^2 + 3x + 4$$

La multiplication classique des polynômes donne :

$$A(x) \times B(x) = (4x^3 + 3x^2 + 2x + 1)(x^3 + 2x^2 + 3x + 4)$$

En développant chaque terme :

$$4x^3 \times (x^3 + 2x^2 + 3x + 4) = 4x^6 + 8x^5 + 12x^4 + 16x^3$$

$$3x^2 \times (x^3 + 2x^2 + 3x + 4) = 3x^5 + 6x^4 + 9x^3 + 12x^2$$

$$2x \times (x^3 + 2x^2 + 3x + 4) = 2x^4 + 4x^3 + 6x^2 + 8x$$

$$1 \times (x^3 + 2x^2 + 3x + 4) = x^3 + 2x^2 + 3x + 4$$

En additionnant tous les termes, on obtient :

$$A(x) \times B(x) = 4x^6 + 11x^5 + 20x^4 + 30x^3 + 20x^2 + 11x + 4$$

Avec la méthode de *Karatsuba*, les polynômes sont divisés en deux parties :

$$D_1(x) = 4x + 3 \quad D_0(x) = 2x + 1$$
$$E_1(x) = x + 2 \quad E_0(x) = 3x + 4$$

Puis nous calculons 3 produits intermédiaires :

$$D_1E_1 = 4x^2 + 11x + 6$$
$$D_0E_0 = 6x^2 + 11x + 4$$
$$(D_1 + D_0)(E_1 + E_0) = (6x + 4)(4x + 6) = 24x^2 + 52x + 24$$

Cela donne :

$$AB = (4x^2 + 11x + 6) \cdot x^4 + \big((24x^2 + 52x + 24) - (4x^2 + 11x + 6) - (6x^2 + 11x + 4)\big)\, x^2 + 6x^2 + 11x + 4$$

Et en simplifiant :

$$AB = 4x^6 + 11x^5 + 20x^4 + 30x^3 + 20x^2 + 11x + 4$$

Voici la fonction `mult_karatsuba` :

```
let rec mult_karatsuba poly1 poly2 =
  if(degre poly1 = 0 || degre poly2 = 0 ) then
    poly_prod poly1 poly2
  else
    let k =
      let max_val = max (degre poly1) (degre poly2) in
    if max_val mod 2 = 0 then max_val else max_val+1 in
    Printf.printf"k val %d \n"k;
    let (a0, a1) = split_poly poly1 (k/2) in
    let (b0, b1) = split_poly poly2 (k/2) in
    let c1 = mult_karatsuba a0 b0 in (* a0*b0 *)
    let c2 = mult_karatsuba a1 b1 in (* a1*b1 *)
    let c3 = somme_poly a1 a0 in (* a1 + a0 *)
    let c4 = somme_poly b1 b0 in (* b1 + b0 *)
    let u = mult_karatsuba c3 c4 in (* (a1 + a0)* (b1 + b0) *)
    let c5 = diff_poly (diff_poly u c2) c1 in
    somme_poly (somme_poly (multExpo c2 (k)) (multExpo c5 (k/2))) c1
```

5.3.5 Méthode 6 : transformation de Fourier rapide (FFT)

La méthode `mult_FFT` utilise la Transformée de Fourier Rapide (FFT) [1] pour multiplier deux polynômes. Cette méthode permet de passer du domaine des coefficients à un domaine fréquentiel, où le calcul du produit de deux polynômes devient beaucoup plus rapide.

L'idée de base est de transformer les polynômes en vecteurs de coefficients, puis d'appliquer la FFT pour obtenir leur représentation en domaine fréquentiel. Une fois les polynômes transformés, on effectue leur produit dans ce domaine, puis on utilise l'inverse de la FFT pour revenir aux coefficients du produit du polynôme.

5.3.5.1 Principe

Pour deux polynômes $A(x)$ et $B(x)$, de degrés respectifs $n_1 - 1$ et $n_2 - 1$, leur produit $C(x) = A(x) \times B(x)$ est un polynôme de degré $n - 1$, avec $n = n_1 + n_2 - 1$. La FFT réduit la complexité du calcul de ce produit à $O(n \log n)$.

1. **Transformation en vecteurs de coefficients** :
   Les coefficients des polynômes $A(x)$ et $B(x)$ sont transformés en deux vecteurs de taille N, où N est une puissance de 2 supérieure ou égale à n. Des zéros sont ajoutés pour compléter les vecteurs.
2. **Passage au domaine fréquentiel** :
   La FFT est applqiué au deux vecteurs de coefficients, en les multipliant avec la matrice complexe appelée matrice de Fourier.
   $$\hat{A}[k] = \sum_{j=0}^{N-1} a_j \cdot \omega^{jk}, \quad \text{avec } \omega = e^{2i\pi/N}.$$
3. **Multiplication dans le domaine fréquentiel** :
   Chaque coefficient de $\hat{A}$ est multiplié par le coefficient correspondant de $\hat{B}$, donnant un nouveau vecteur $\hat{C}$.
4. **Retour au domaine temporel** :
   Seuls les n premiers coefficients du vecteur $\hat{C}$ sont gardés.

5.3.5.2 Exemple de déroulement

Soit deux polynômes $A(x)$ et $B(x)$ de degré $n = 2$:

$$A(x) = 3x^2 + 2x + 1$$
$$B(x) = 2x^2 + 4x + 3$$

1. **Vecteurs de coefficients** :
   Les coefficients sont :
   $$A = [1, 2, 3, 0, 0], \quad B = [3, 4, 2, 0, 0].$$
2. **FFT** :
   La FFT produit :
   $$\hat{A} = [6, 1 - 3i, -2, 1 + 3i, 0], \quad \hat{B} = [9, 2 - 4i, -1, 2 + 4i, 0].$$
3. **Multiplication fréquencielle** :
   Le produit $\hat{A} \cdot \hat{B}$ donne :
   $$\hat{C} = [54, -10 + 14i, 2, -10 - 14i, 0].$$
4. **IFFT** :
   Application de la FFT inverse pour repasser au domaine temporel :
   $$C(x) = 6x^4 + 14x^3 + 17x^2 + 10x + 3.$$

Voici le code de la fonction `mult_FFT` :

```
let mult_FFT poly1 poly2 =
  let vec1 = get_coeffs poly1 in
  let vec2 = get_coeffs poly2 in

  let n1 = List.length vec1 in
  let n2 = List.length vec2 in

  let n = nextpowof2 (n1 + n2) in

  let a1 = Array.make n { re = 0.0; im = 0.0 } in
  let b1 = Array.make n { re = 0.0; im = 0.0 } in

  for i = 0 to n1 - 1 do
      a1.(i) <- { re = List.nth vec1 i; im = 0.0 };
  done;

  for i = 0 to n2 - 1 do
      b1.(i) <- { re = List.nth vec2 i; im = 0.0 };
  done;

  let theta = 2.0 *. Float.pi /. float_of_int n in
  let w = { re = cos theta; im = sin theta } in

  let a2 = fft a1 n w in
  let b2 = fft b1 n w in

  let c1 = Array.init n (fun i -> mul a2.(i) b2.(i)) in

  let win = conj w in
  let c2 = fft c1 n win in
  let d = Array.map (fun c -> { re = c.re /. (float_of_int n); im = 0. }) c2 in
  Array.fold_left (fun pol (di, i) ->
    if((Float.round di.re) <> 0.0) then
      Mono(int_of_float (Float.round di.re), i) :: pol
    else
      pol
  ) [] (Array.mapi (fun i di -> (di, i)) d)
;;
```

5.4 Comparaison des stratégies

5.4.1 Durées pour les additions (Pas de 100)

Le tableau 5.1 présente les résulats des performances des 3 méthodes : naïve, fusion et diviser-pour-régner, en fonction de la taille de n des données.
La méthode naïve, bien que facile à implementer, devient rapidement lente lorsque n augmante, ce qui la rend peu efficace. L'addition par fusion est légèrement rapide comparé à la méthode naïve. Cependant, son avantage diminue lorsque n est plus grands. En revanche, l'approche diviser-pour-régner reste la méthode la plus efficace dans presque tous les cas. Elle garde des temps d'exécution faible, même quand n devient très grand.
Ces résultats montrent que, pour des petites instances, toutes les méthodes peuvent convenir. Mais pour des tailles plus importantes, il vaut mieux utiliser des méthodes optimisés, tel que l'approche diviser-pour-régner, car elles permettent de gagner du temps tout en restant fiable.

Taille n	**Addition Naïve**	**Addition Fusion**	**Addition Divide**
100	0.00157s	0.00142s	**0.00128s**
200	0.00295s	0.00308s	**0.00293s**
300	0.00544s	0.00442s	**0.00433s**
400	0.00646s	0.00633s	**0.00580s**
500	0.00894s	0.00828s	**0.00767s**
600	0.00921s	0.00978s	**0.00897s**
700	0.01196s	0.01174s	**0.01068s**
800	0.01375s	0.01380s	**0.01253s**
900	0.01533s	0.01611s	**0.01388s**
1000	0.02291s	0.01963s	**0.01535s**

TABLE 5.1 – Durées des additions pour n avec un pas de 100

5.4.2 Temps des additions

En analysant le graphe en ci-dessous, il est évident que chaque méthode a un comportement différent en fonction de la taille n.
La méthode naïve (représentée en bleu) montre une augmentation progressive du temps avec l'augmentation de n. La méthode fusion (représentée en rouge) a aussi une tendance croissante mais ses temps d'exécutions restent globalement inférieurs comparés aux temps de la méthode naïve. Enfin, l'approche diviser-pour-régner (représentée en vert) est la plus efficace, surtout pour les grandes tailles.

5.4.3 Moyennes pour les additions (Question 19)

En analysant les résultats présentés dans le tableau 5.2, il est clair que l'approche diviser-pour régner offre de meilleurs résultats en terme de temps d'exécution.

Opération	**Naïve**	**Fusion**	**Divide**
Addition	0.70082s	0.58028s	**0.56653s**

TABLE 5.2 – Temps moyens pour les additions (10 répétitions)

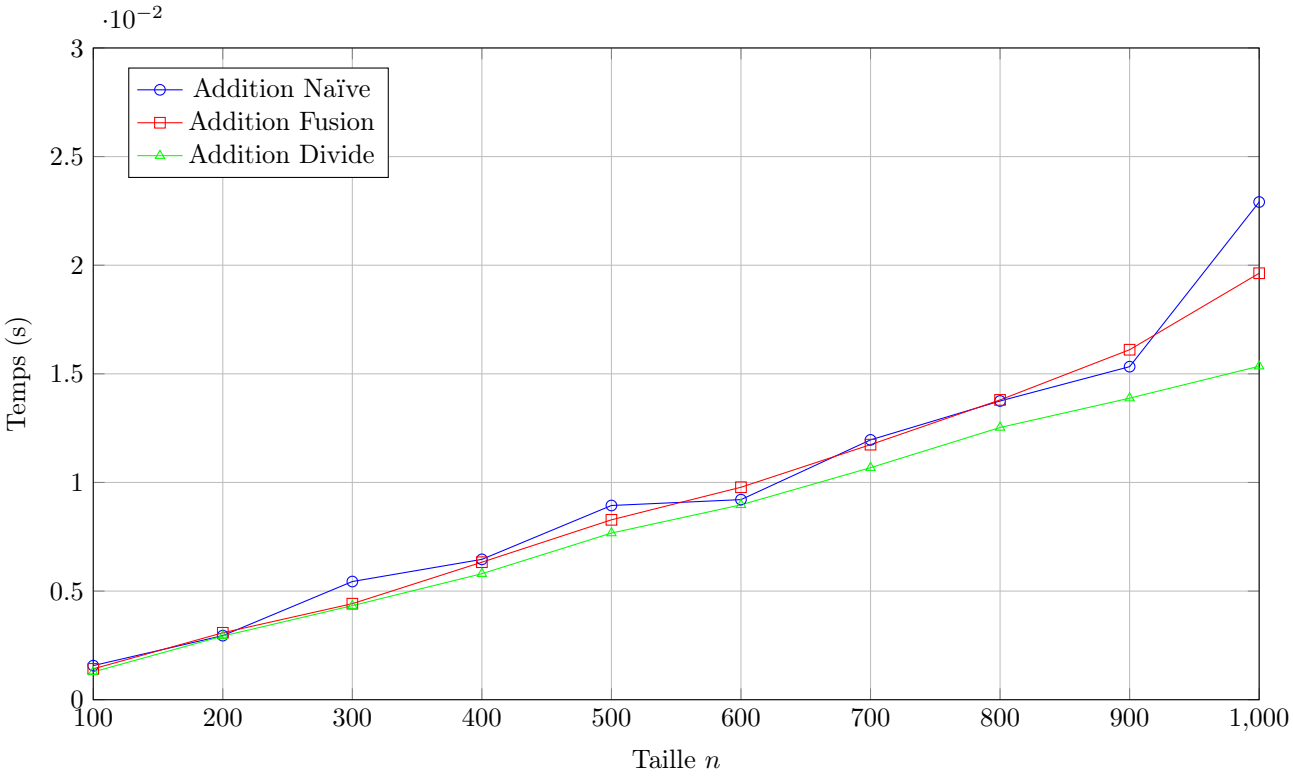

FIGURE 5.3 – Performances des méthodes d'addition

5.4.4 Durées pour les multiplications (Pas de 100)

Taille n	**Produit Naïf**	**Produit Fusion**	**Produit Divide**
100	**0.70s**	0.71s	4.90s
200	4.50s	**4.22s**	30.94s
300	9.72s	**9.53s**	64.63s
400	**9.91s**	10.12s	102.41s
500	25.70s	**24.30s**	203.40s
600	2.21s	**1.98s**	87.16s
700	24.75s	**24.19s**	208.42s
800	**3.86s**	4.48s	212.69s
900	**13.00s**	14.34s	329.39s
1000	**10.99s**	11.96s	240.41s

TABLE 5.3 – Durées des multiplications pour n avec un pas de 100

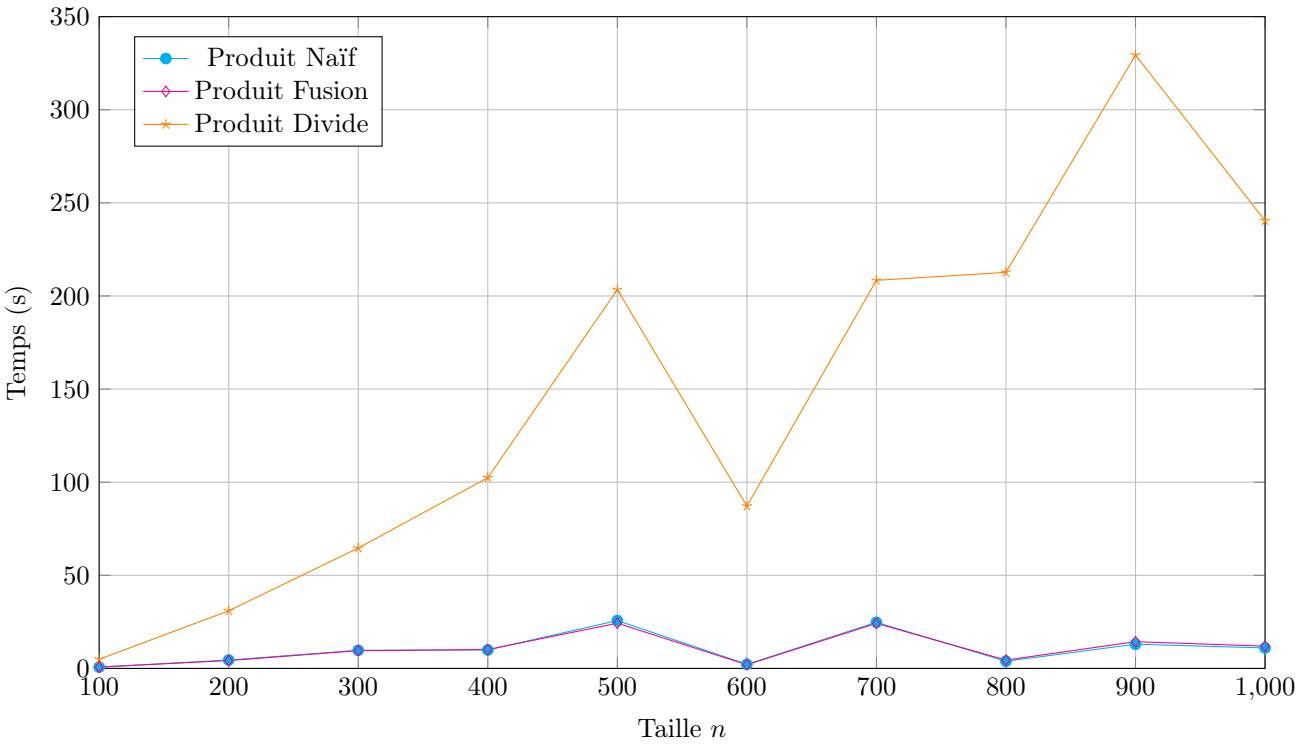

FIGURE 5.4 – Performances des méthodes de multiplication

5.4.5 Durées pour les multiplications (Pas de 20)

Taille n	Produit Naïf	Produit Fusion	Produit Divide
20	0.015914	**0.015704**	0.109297
40	0.040958	**0.040670**	0.222645
60	0.094580	**0.089547**	0.420163
80	**0.164344**	0.164800	1.043825
100	**0.000918**	0.000963	0.441775
120	**0.318272**	0.324676	1.871173
140	**0.001556**	0.001575	0.834098
160	**0.611171**	0.664548	3.930907
180	0.837769	**0.832240**	5.222320
200	1.000413	**0.995982**	6.714298
220	1.192366	**1.187353**	7.575200
240	**1.325375**	1.351544	9.002425
260	1.877804	**1.850912**	11.863939
280	**2.633701**	2.640604	16.958034
300	0.380042	**0.377190**	4.499134

TABLE 5.4 – Durées des opérations de multiplication pour différentes tailles n.

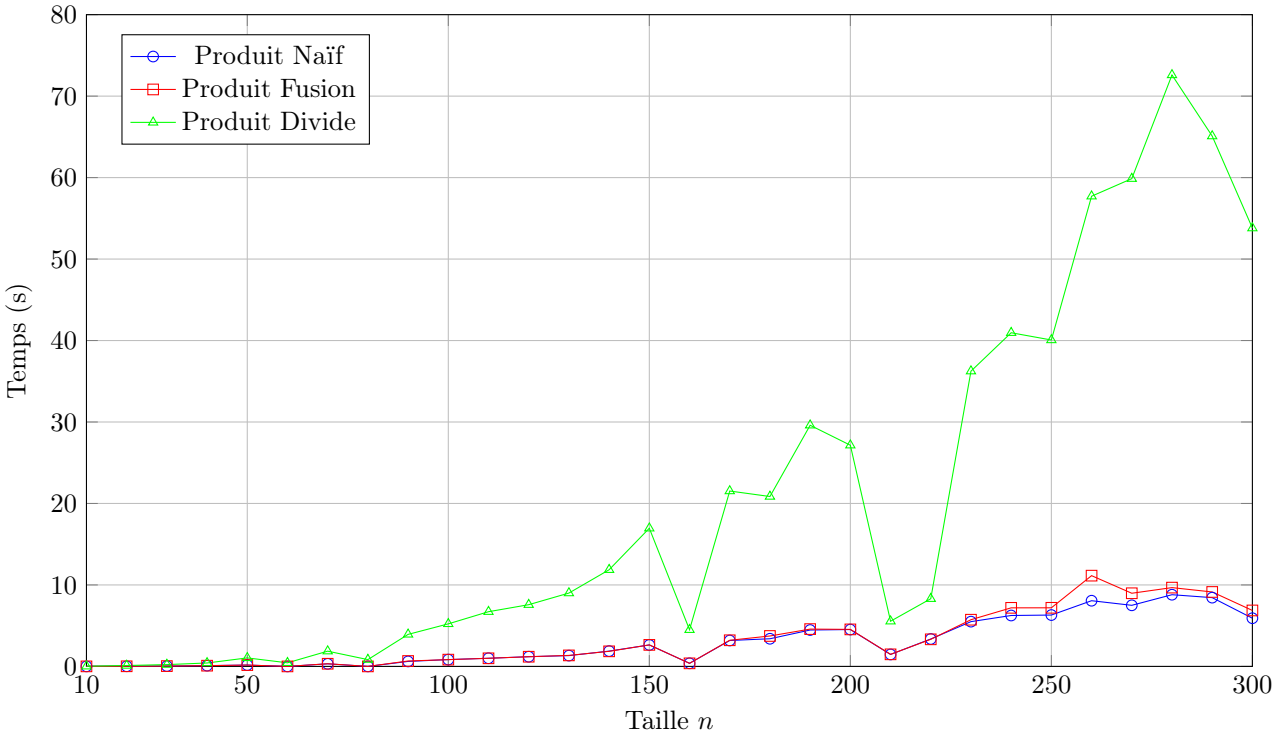

FIGURE 5.5 – Comparaison des durées des opérations de multiplication pour différentes tailles n.

5.4.6 Moyennes pour les multiplications (Question 19)

Opération	**Naïve**	**Fusion**	**Divide**
Produit	4.04708s	**4.00111s**	4.38365s

TABLE 5.5 – Temps moyens pour les multiplications (10 répétitions)

5.4.7 Explications des résultats

5.4.7.1 Comparaison des différentes approches

D'après les résultats obtenus, on remarque que les stratégies naïve et fusion sont similaires en performance et que la stratégie Diviser pour régner présente des temps moins bons. Ce résultat peut paraître étrange à première vue mais si on la compare par exemple à notre approche naïve, on remarque qu'au final, on réalise le même nombre d'opérations mais dans un ordre différent et plus coûteux à mettre en place.

En effet, par exemple pour n = 1000, on va réaliser la multiplication à la suite des 1000 arbres pour l'approche naïve, soit :

$$(a1 * (a2 * (a3 * ... * (a999 * a1000))))$$

Alors qu'avec l'approche Diviser pour régner, les opérations sont organisés sous la forme :

$$((a1 * a2) * (a3 * a4)) * ... * ((a997 * a998) * (a999 * a1000))$$

Au final, on a bien le même nombre d'opérateurs principaux, mais la stragégie Diviser pour régner nécessite une préparation assez coûteuse dont n'a pas besoin notre méthode naïve ce qui peut expliquer cette différence de temps d'exécution.

5.4.7.2 Explication de la tendance de la courbe

Pour le temps des multiplications avec le nombre d'arbres n qui varie, on remarque sur le graphique que les temps d'exécutions fluctuent et q'il ne présente pas une tendance croissante comme on pourrait l'imaginer. A première vue, on pourrait se dire que ces résultats sont erronés mais il y a une raison derrière cela.

En effet, lors de nos expérimentations, nous avons remarqué que pour de grands n, les polynomes retournés par nos stratégies étaient très quasiment toujours nuls et qu'il y avait une corrélation entre un polynome obtenu qui était nul et un temps incohérent (temps d'exécution inférieur par rapport à des n plus petits).

Au début, nous pensions que cela était dû à une limite d'Ocaml, mais après pas mal de recherche, il nous est venu l'idée que cela pouvait être dû à la rencontre d'un polynome nul lors des multiplications. Cela expliquerait le résultat nul à la fin de l'exécution ainsi que les temps obtenus non croissants en fonction de la taille de n.

En effet, lors de la rencontre d'un facteur nul, toutes les prochaines suivantes s'exécutent très rapidement. En outre, il est plus probable de rencontrer un facteur nul avec un grand nombre d'arbres. Cela explique également la grande différence de temps de diviser pour régner avec les deux autres méthodes pour des n très grands puisque la rencontre d'un facteur nul à moins d'impact dans cette approche.

Bibliographie

[1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90) :297–301, 1965.

[2] A. A. Karatsuba. Multiplication of large numbers. *Soviet Physics Doklady*, 7 :595–596, 1962.