2021/2022

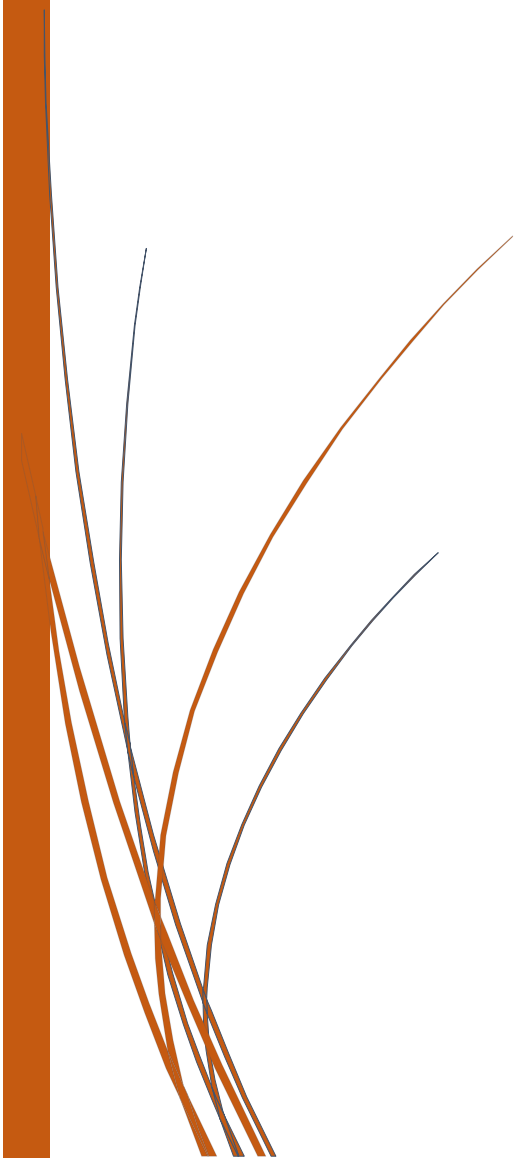
Devoir de Programmation

UE Ouverture

Sous la supervision de :

Mr. GENITRINI Antoine

Mr. CHAILLOUX Emmanuel



AMNACHE Miassa

BENAKCHA Mebarek Raid

1 Polynôme sous forme linéaire

Présentation :

Dans le cadre de l'évaluation finale du module ouverture, il nous a été demandé de manipuler deux modèles de structure de données : l'une sous forme linéaire et l'autre arborescente. Des expérimentations sur l'efficacité d'évolution des structures arborescentes ainsi que de leur transformation sous forme linéaire concluront notre travail.

Un polynôme est une expression de la forme $c_0X^0 + c_1X^1+c_2X^2+... +c_nX^n$. Les c_i sont les coefficients du polynôme, on considère qu'ils appartiennent à Z. Les termes cX^d sont appelés des monômes, c'est le coefficient et d est le degré du monôme. Un polynôme est donc une somme de monômes. Ici **un polynôme est une liste de monômes**.

Polynôme sous forme linéaire :

Définition du type monôme

2X est représenté par (2, 1)
$4X^2$ est représenté par (4, 2)
x^3 est représenté par (1, 3)
123 est représenté par (123, 0)
0 est représenté par (0, 0)

1.1 Définition linéaire du type polynôme

On peut représenter un monôme par un enregistrement qui contient deux champs : « c : le coefficient, d : degré »

```
type monome = { c : int; d : int; };; (** Definition du type monome *)
type polynome = monome list;; (** Definition linéaire du type polynome *)
```

1.2 Représentation canonique du polynôme

Une **fonction canonique** prend en entrée un polynôme sous forme linéaire et renvoie sa représentation canonique (s'il n'existe pas plusieurs monômes de même degré et si (c, d) est un élément de la liste alors $c \neq 0$. Enfin le polynôme valant 0 est représenté par une liste vide).
Pour réaliser ça, on a défini trois fonctions auxiliaires : une qui assure le tri (monComp), la suppression des doublons (remDup) et la suppression des monômes dont le coefficient est à 0 (remZero).
List.sort monComp Tri croissant par rapport au degré du monôme.
remDup Fonction de suppression des duplications des monômes du même degré.
remZero Suppression des monômes dont le coefficient = 0.

```
let monComp a b = a.d - b.d;; (** Tri croissant par rapport au degré du monôme *)
let pol = List.sort monComp [{c=2;d=3}; {c=9;d=4}; {c=2;d=3}; {c=9;d=5}; {c=9;d=1}];;
let rec remDup pol = match pol with (** Fonction de suppression des duplications des monômes du même degré *)
        [] -> []
        |[{c;d}] -> [{c;d}]
        |a::b::rest -> if a.d = b.d then {c=a.c+b.c; d=a.d}::remDup rest else a::remDup (b::remDup rest);;
let rec remZero pol = match pol with (** Suppression des monômes dont le coéfficient = 0 *)
          [] -> []
          |a::rest -> if a.c = 0 then remZero rest else a::remZero rest;;

let canonique pol = let (pol:polynome) = List.sort monComp pol in let pol = remDup pol in remZero pol;; (** La fonction 'canonique' *)
```

1 Polynôme sous forme linéaire

Analyse de la complexité :

Toutes les fonctions auxiliaires utilisées pour définir la fonction **`canonique`** se basent sur la comparaison dans leurs traitements. On fait l'analyse de la complexité de la fonction selon le nombre total de comparaisons qu'elle fait.
1. **`List.sort`** : Méthode prédéfinie dans OCaml, qui trie une liste selon une fonction passée en paramètre, cette dernière indique le critère de tri. ***monComp a b = a.d – b.d*** est la fonction passée, qui précise de trier les monômes dans l'ordre croissant de leurs degrés. L'implantation de cette méthode utilise le Merge Sort, donc la complexité est en ***O(n log n)***. (https://ocaml.org/api/List.html#VALsort).
2. **`remDup`** : Une fonction qui parcourt toute la liste de monômes une fois, pour chaque deux monômes successifs du même degré, elle fait l'addition de leurs coefficients. ***O(n)***
3. **`remZero`** : Une fonction qui parcourt toute la liste de monômes une fois et supprime chaque monôme de coefficient 0. ***O(n)***

En général, la complexité temporelle de la fonction **`canonique`** est de **O(n log n)**.

1.3 Addition des polynômes :

La fonction **`poly_add`** prend en entrée deux polynômes canoniques et renvoie leur somme sous forme canonique.

```
let  poly_add pol1 pol2 =
let rec poly_add_aux pol1 pol2 =
match pol1, pol2 with (** Addition de deux polynomes canoniques *)
            [],[] -> []
            |pol1, [] -> pol1
            |[], pol2 -> pol2
            |a1::rest1, a2::rest2 -> if a1.d = a2.d then ( {c=a1.c + a2.c; d=a1.d}::poly_add_aux rest1 rest2 )else if a1.d < a2.d then
            a1::(poly_add_aux rest1 (a2::rest2)) else a2::(poly_add_aux (a1::rest1) rest2)
in remZero(poly_add_aux pol1 pol2);;
```

Analyse de la complexité :

Pour faire l'addition de deux polynômes canoniques, la fonction **`poly_add`** prend la tête de chaque liste a1 et a2 (qui représentent le monôme du degré minimum) et les compare, s'ils sont du même degré, elle additionne leurs coefficients, et fait le même traitement récursivement pour les deux restes des deux listes. Sinon, elle prend le monôme du degré minimum et appelle récursivement **`poly_add`** pour le reste de cette liste avec l'autre liste. Le temps d'exécution change en fonction de nombre de comparaisons entre les monômes.

$$poly\_add(pol1,\quad pol2) = \begin{cases} additioner\ les\ coefficients::poly\_add(rest1, rest2) : si\ a1.d = a2.d \\ a_i::poly\_add(rest_i, pol_j) : si\ a_i.d < a_j.d \end{cases}$$

Le nombre de comparaisons maximum serait : t_1+t_2 tel que t_i est la taille de la liste pol_i . D'où ***O(t_1+t_2)***

1 Polynôme sous forme linéaire

1.4 Produit des polynômes :

La fonction **`poly_prod`** prend en entrée deux polynômes canoniques et renvoie leur produit sous forme canonique.

```
let rec poly_prod pol1 pol2 = match pol1, pol2 with (** Produit de deux polynomes cannoniques *)
          [],[] -> []
          |pol1, [] -> []
          |[],pol2 -> []
          |a1::rest1, a2::rest2 ->poly_add (poly_add [{c=a1.c*a2.c;d=a1.d+a2.d}] (poly_prod [a1] rest2)) (poly_prod rest1 (a2::rest2))
          ;;
```

Analyse de la complexité :

Considérons l'algorithme naïf ;
Cette structure est à la fois de type "somme" et de type "produit" et se définit de façon récursive.

Pour chaque monôme de la première liste, la fonction poly_prod multiplie son coefficient avec le coefficient de chaque monôme de l'autre liste et additionne leurs degrés, puis elle fait appel à poly_add pour additionner le résultat total. Le produit de deux polynômes peut être représenté par l'équation suivante

$$pol1 \times pol2 = \sum_{i=0} \sum_{j=0} (c_i \times c_j) X^{d_i + d_j}$$

La complexité temporelle en fonction de nombre de multiplications est donc : ***O(n^2)***

2 Expression arborescente

1.5 Définition de structure arborescente :

```
type e =   (** Structure arborescente d'une expression *)
    Int of int
  | Chap of  int
  | Plus of  plusChild list
  | Prod of  prodChild list
and plusChild =
    Int2 of int
  | Chap2 of int
  | Prod2 of  prodChild list
and prodChild =
    Int3 of int
  | Chap3 of int
  | Plus2 of plusChild list;;
```

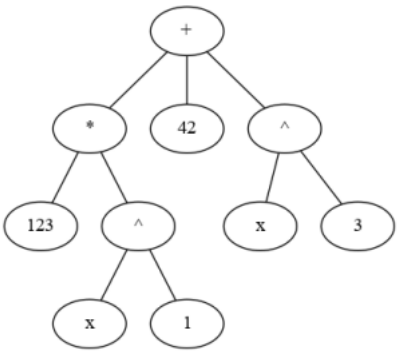
Le type ***e*** correspond à l'expression générale : E = int | E∧ | E+ | E∗

- Int, Int2, Int3 sont les constructeurs des nœuds qui représentent des valeurs entières.
- Chap, Chap2, Chap3 sont les constructeurs des nœuds qui représentent les degrés des monômes.
- Plus, Plus2 sont les constructeurs des nœuds qui représentent l'opération d'addition. Ils sont de type **plusChild** list.
- Prod, Prod2 sont les constructeurs des nœuds qui représentent l'opération de la multiplication. Ils sont de type **prodChild** list.

Le type **plusChild** (resp **prodChild**) représente ce que **E+** (resp **E∗**) peut accepter comme expression tel que : Int2, Chap2, Prod2 sont les constructeurs des nœuds des sous-arbres dont la racine est E+ et Int3, Chap3, Plus2 sont les constructeurs des nœuds des sous-arbres dont la racine est E∗ .

Pour assurer que **plusChild** appelle **prodChild** et que **prodChild** appelle **plusChild** nous les avons définis en récursion mutuelle (mot clé **AND**).

1.6 Définition d'une expression équivalente à l'arbre ci-dessous :




```
let arbreEq = Plus([Prod2([Int3(123);Chap3(1)]);Int2(42);Chap2(3)]);;
```

2 Expression arborescente

1.7 Conversion d'un arbre en un polynôme :

La fonction arb2poly prend en entrée une expression arborescente et la transforme en un polynôme canonique.

Avant de définir la fonction arb2poly il a fallût définir les fonctions auxiliaires arb2poly_plus et arb2poly_prod qui servent à transformer les sous-arbres dont la racine est "+" et "*" respectivement.

```
let rec arb2poly_plus a = match a with
            | []->[]
            |Int2 (x:int)::rest -> poly_add ([{c=x;d=0}]) (arb2poly_plus rest)
            |Chap2(x:int)::rest -> poly_add ([{c=1;d=x}]) (arb2poly_plus rest)
            |(Prod2(b::rest))::rest2 ->poly_add (poly_prod (arb2poly_prod [b]) (arb2poly_prod
rest)) (arb2poly_plus rest2) and
     arb2poly_prod a = match a with
            | [] -> [{c=1;d=0}]
            |Int3 (x:int)::rest -> poly_prod ([{c=x;d=0}]) (arb2poly_prod rest)
            |Chap3(x:int)::rest -> poly_prod ([{c=1;d=x}]) (arb2poly_prod rest)
            |Plus2(b::rest)::rest2 ->poly_prod (poly_add (arb2poly_plus [b]) (arb2poly_plus
rest))(arb2poly_prod rest2);;
```

De manière récursive, **arb2poly_plus** (resp **arb2poly_prod**) convertit la liste dont la tête est :

- **Int2(x)::rest** (resp **Int3(x)::rest**) en une addition (resp multiplication) entre le polynôme réduit à un seul monôme de degré 0 et de coefficient x avec la transformation du reste (**arb2poly_plus**/**arb2poly_prod** rest).

- **Chap2(x)::rest** (resp **Chap3(x)::rest**) en une addition (resp multiplication) entre le polynôme réduit à un seul monôme de degré x et de coefficient 1 avec la transformation du reste (**arb2poly_plus**/**arb2poly_prod** rest).

- **Prod2(b::rest)::rest2** (resp **Plus2(b::rest)::rest2)** en une addition (resp une multiplication) entre le produit (resp la somme) de la transformation de la tête 'b' et la transformation du reste '**rest**' et la transformation du reste de l'addition (resp la multiplication) '**rest2**'.

Une fois, les fonctions auxiliaires sont prêtes, on définit la fonction arb2poly qui fonctionne de la même façon que **arb2poly_plus** et **arb2poly_prod** tout en les utilisant.

```
let rec arb2poly a = match a with (** Transformer un arbre en polynome *)
            | Plus([]) -> []
            |Prod([]) -> []
            |Int (x:int) -> [{c=x;d=0}]
            |Chap(x:int) -> [{c=1;d=x}]
            |Plus(b::rest) -> poly_add (arb2poly_plus [b]) (arb2poly_plus rest)
            |Prod(b::rest) -> poly_prod (arb2poly_prod [b]) (arb2poly_prod rest);;
```

2 Expression arborescente

1.8 La fonction **extraction_alea** :

```
let extraction_alea (a:int list) (b:int list) = match a,b with (** Extraction aléatoire *)
      | [],b -> ([],b)
      |a,b -> (
          let l = List.length a in let r = (Random.int (l) + 1 ) in (
              let rec extract_aux lst i= match lst with
              [] -> []
              |a::rest -> if (i=1) then rest else a::(extract_aux rest (i-1))
              in ((extract_aux a r),((List.nth a (r-1))::b))
          )
      ) )
```

La définition de la fonction **extraction_alea** est assez simple, le code représente une traduction directe de l'énoncé en langage OCaml.

1.9 La fonction gen_permutation :

```
2     let gen_permutation n = ( (** Génération d'une liste d'entiers d'ordre aléatoire *)
3
4           let rec unfold_right f init =( (** fonction pour aider à générer une liste
    croissante d'entiers *)
5       match f init with
6       | None -> []
7       | Some (x, next) -> x :: unfold_right f next)
8        in let range n =
9       (let irange x = if x > n then None else Some (x, x + 1) in
10      unfold_right irange 1)
11      in let l = (range n) in (
12          let rec gen (x,y)=
13
14              match (x,y) with
15              |([],p) -> ([],p)
16              |(x,y) -> gen (extraction_alea x y)
17          in (gen (range (n),[]))
18      ));;
```

La fonction **gen_permutation** fait son traitement en deux étapes :

-Génération d'une liste croissante de 1 à n (appel à la fonction **range**).

-Appel à **extraction_alea** sur le résultat de la fonction **range**. (Utilisation de la fonction ***gen*** récursivement).

2 Expression arborescente

1.10 Construction d'ABR à partir d'une liste d'entiers :

Afin de pouvoir construire un ABR, nous devons d'abord définir le type arbre qui va représenter sa structure :

```
    type arbre = Feuille
                   | Noeud of int*arbre*arbre
                   | Noeud2 of string*arbre*arbre;;
```

Les types de nœuds possibles d'un arbre sont :

Feuille (qui représente un nœud sans valeur et sans fils)

Nœud (qui représente un nœud avec une valeur entière et qui a deux fils de type arbre).

Noeud2 (qui représente un nœud qui porte l'opérateur « + », « * », « ^ » et qui a deux fils de type arbre)

Ensuite, pour construire l'ABR à partir d'une liste d'entier, nous devons définir une fonction récursive **`inserer`** qui s'occupe de l'insertion des valeurs lues à partir de la liste dans cet ABR initialement réduit à une feuille. Si la valeur est inférieure à celle de la racine, elle va être insérer dans le sous arbre gauche (appel récursif de **`inserer`** sur la valeur l avec le sous-arbre gauche), sinon elle va être insérer dans le sous-arbre droit (de la même manière que le fils gauche).

```
let abr l =
let l = List.rev l in
    let rec abr_aux l =

        let rec inserer v = function
            | Feuille -> Noeud (v, Feuille, Feuille)
            | Noeud (r, fg ,fd) ->
                if v < r then Noeud ( r,inserer v fg, fd)
                else Noeud (r, fg,  inserer v fd)
        in match l with
            [] -> Feuille
            | t::q -> inserer t (abr_aux q) in abr_aux l
            ;;
```

Remarque : la fonction **`inserer`** fait l'insertion de sorte que l'ABR final aura comme racine la dernière valeur de la liste, donc on a utilisé **`List.rev`** pour inverser la liste et pour que la racine de l'ABR soit la tête de la liste.

2 Expression arborescente

1.11 La fonction etiquetage :

Comme pour la fonction **`extraction_alea`**, la définition de la fonction `etiquetage` nécessite seulement la traduction directe de l'énoncé en code.

```
let etiquetage a =
    let rec etiquetage_aux = function
    | Noeud(v,Feuille,Feuille) -> (if (v mod 2 = 1) then (let vr = ((Random.int 401) - 200) in
Noeud2("*",Noeud(vr,Feuille,Feuille),Noeud2("x",Feuille,Feuille)))
                                                    else
(Noeud2("^",Noeud2("x",Feuille,Feuille),Noeud((Random.int 101), Feuille,Feuille))))
    | Noeud(l, fg,fd) -> let p = Random.int 100 in if (p<75) then Noeud2("+",etiquetage_aux
fg, etiquetage_aux fd) else Noeud2("*",etiquetage_aux fg, etiquetage_aux fd)
    |Feuille -> let p = Random.int 100 in (if (p<50) then Noeud(((Random.int 401) -
200),Feuille,Feuille) else Noeud2("x",Feuille,Feuille)) in etiquetage_aux a;;
```

La seule remarque à noter est que la méthode '**`Random.int`** x' génère un entier aléatoire entre 0 et x exclusif soit [0 ; x[. Donc, il fallait trouver x tel que l'entier à générer soit $-200 \leq i \leq 200$

Les étapes suivies :

$$-200 \leq i \leq 200$$
$$0 \leq i + 200 \leq 400$$
$$0 \leq i + 200 < 401$$

D'où, x = 401. Et pour revenir à l'intervalle : $-200 \leq i \leq 200$, il suffit de soustraire 200 de la valeur retournée par **`Random.int`** `401`.

1.12 Génération de l'arbre qui respecte la grammaire :

La fonction à définir ***gen_arb*** prend en argument la structure retournée par etiquetage. Cette dernière ne respecte pas nécessairement la grammaire des expressions arborescentes des polynômes :

$$E = int \mid E_{\wedge} \mid E_{+} \mid E_{*}$$
$$E_{\wedge} = x \ ^{\wedge} \ int^{+}$$
$$E_{+} = (E \setminus E_{+}) + (E \setminus E_{+}) + \ldots$$
$$E_{*} = (E \setminus E_{*}) * (E \setminus E_{*}) * \ldots$$

Dans le résultat de la fonction **`etiquetage`**, on pourra avoir des expressions E+ qui ont des sous-expressions directes de type E+, de même pour E*. Donc le traitement principal réside dans la fusion de manière récursive les sous-expressions de l'expression E+ avec les sous-expressions des sous-expressions directes E+ (garder une seule expression E+ dont ses nœuds sont l'union des sous-arbres E+ directs) et de même pour E*.

De la même façon qu'on a défini **`arb2poly`** avec ses auxiliaires **`arb2poly_plus`** et **`arb2poly_prod`**, on a besoin de définir **`gen_arb_plus`**, qui va transformer les sous-arbres de racine « + » en une liste de type **plusChild** list, respectivement prodChild list avec « * ». Mais la construction d'une liste plusChild list et **prodChild** list nécessite d'abord la fusion des sous-arbres directs des sous-expressions de même type E+ ou E*, d'où on a défini les fonctions **`traite`** et **`traite2`** pour s'en occuper. Reste juste à mentionner que la fusion de deux listes ne se fait pas de manière native en OCaml, donc il nous a fallu définir la fonction **`merge`** qui fait la fusion.

2 Expression arborescente

Une fois on a défini toutes ces fonctions auxiliaires, on définit **gen_arb** qui s'en sert pour générer une expression qui respecte la grammaire de la section 1.2 à partir d'un arbre retourné par **etiquetage**.

Le code de **gen_arb** :

```
let rec merge list1 list2 =
    match list1, list2 with
        | [], _ -> list2
        | hd :: tl, _ -> hd :: merge tl list2 ;;
let rec gen_arb a = match a with
    | Noeud(v,_,_) -> Int(v)
    | Noeud2("+",g,d) -> Plus(merge (traite g) (traite d))
    | Noeud2("*",g,d)->Prod(merge (traite2 g) (traite2 d))
    | Noeud2("^",g,Noeud(v,_,_)) -> Chap(v)
    | Noeud2("x",_,_) -> Chap(1)

    and  gen_arb_plus a = match a with
    | Noeud(v,_,_) -> Int2(v)
    | Noeud2("^",_,Noeud(v,_,_)) -> Chap2(v)
    | Noeud2("x",_,_) -> Chap2(1)
    | Noeud2("*",g,d) -> Prod2(merge (traite2 g) (traite2 d))
    and  gen_arb_prod a = match a with
    | Noeud(v,_,_) -> Int3(v)
    | Noeud2("^",_,Noeud(v,_,_)) -> Chap3(v)
    | Noeud2("x",_,_) -> Chap3(1)
    | Noeud2("+",g,d) -> Plus2(merge (traite g) (traite d))

    and traite a = match a with
    | Noeud2("+",g,d) -> merge (traite g) (traite d)
    | a -> (gen_arb_plus a)::[]
    and traite2 a = match a with
    | Noeud2("*",g,d) -> merge (traite2 g) (traite2 d)
    | a -> (gen_arb_prod a)::[]
;;
```

3 Expérimentations

2.13 Génération de n arbres de taille 20 :

On a défini la fonction **repArbre** qui prend en paramètre le nombre d'arbres de taille 20 à générer. Elle fait un appel récursif à une méthode auxiliaire **helper** qui va à chaque itération de 1 à n, générer un arbre de taille 20, puis l'ajouter dans la liste des arbres de même taille, pour retourner cette dernière à la fin.

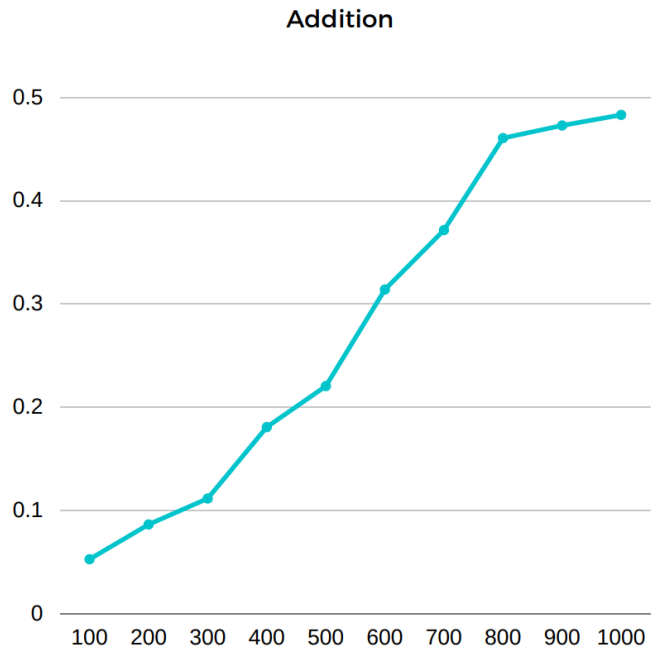
```
let repArbre num =
     let rec helper = fun n acc ->
          if n > 0
          then helper (n-1) ((gen_arb (etiquetage (abr (getList(gen_permutation 20))))) ::
acc)
          else acc
     in
          helper num []
;;
```

2.14 Somme des polynômes issus à partir de ces arbres :

Pour faire la somme des polynômes issus à partir des arbres générés dans la question précédente, on a opté pour la fonction **sumArbre**, qui fait appel à **poly_add** en lui passant la transformation de l'arbre en tête de la liste en polynôme, avec la transformation récursive du reste de cette liste en polynômes (en faisant appel à **arb2poly**).

```
let rec sumArbre liste = match liste with
    | [] -> []
    | [a] -> arb2poly a
    | a::rest -> poly_add (arb2poly a) (sumArbre rest)
```

Le temps d'exécution de la fonction **sumArbre** en fonction de n allant de 100 jusqu'à 1000 en pas de 100 :



3 Expérimentations

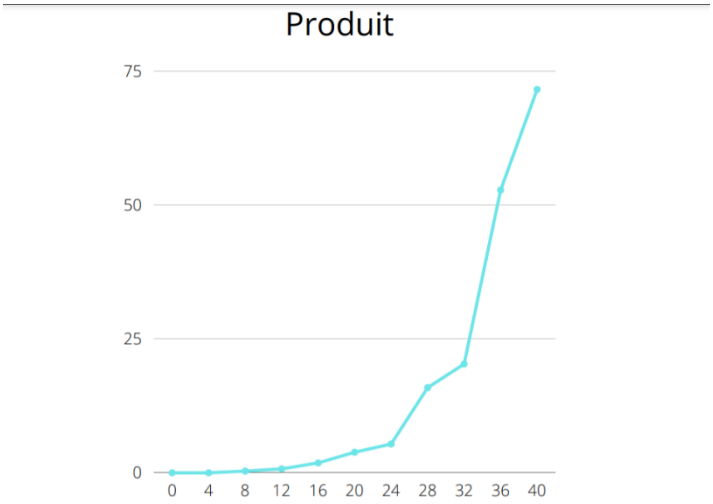
On remarque une croissance linéaire de temps d'exécution par rapport au nombre d'arbre. Cela est normal vu que chaque fonction implantée dans les questions précédentes s'exécute en temps linéaire selon le nombre de On remarque une croissance linéaire de temps d'exécution par rapport au nombre d'arbre. Cela est normal vu que chaque fonction implantée dans les questions précédentes s'exécute en temps linéaire selon le nombre de nœuds de l'arbre qu'elle traite et puis **`poly_add`** qui est en **O(t1 + t2)**. Le pire cas peut s'engendrer si **`etiquetage`** génère un arbre avec que des nœuds « * », ce qui fera appel à **`poly_prod`** lors de l'appel de **`arb2poly_prod`**. Ce cas est très rare vu que etiquetage génère « * » avec probabilité de 25%.

2.15 Produit des polynômes issus à partir de ces arbres :

De même façon qu'on a fait **`sumArbre`**, la fonction **`prodArbre`** fait exactement le même traitement, juste en changeant l'addition avec la multiplication.

```
let rec prodArbre liste = match liste with
    | [] -> [{c=1;d=0}]
    | [a] -> arb2poly a
    | a::rest -> poly_prod (arb2poly a) (prodArbre rest)
```

Comme la fonction **`poly_prod`** s'exécute en temps quadratique, il était impossible de faire le produit de 100 arbres. Donc afin d'obtenir une approximation de la complexité temporelle de cette fonction, on fait le produit de n arbres allant de 4 jusqu'à 40 en pas de 4.



Comme la courbe l'indique, le temps d'exécution croît exponentiellement dès que le nombre d'arbres dépasse 32. Une manière pour améliorer ce résultat est de considérer d'autre technique pour faire la multiplication comme : la méthode de Karatsuba, Fast Fourrier Transformation…

3 Expérimentations

2.16 Génération de 15 ABR de tailles de puissances 2 croissantes :

Une manière de générer une liste de 15 arbres de tailles 2^0,2^0,2^1,2^2,…,2^{13}, c'est de définir une liste ***twoList*** qui contient les tailles de ces arbres, puis, appliquer la méthode **`List.map`** de **`gen_arb`** à l'étiquetage de chaque taille, qui va retourner une liste d'arbres de tailles souhaitées ***abr152***.

```
let twoList = [1;1;2;4;8;16;32;64;128;256;512;1024;2048;4096;8192];;
    let abr152 = List.map gen_arb (List.map etiquetage (List.map abr (List.map getList
(List.map gen_permutation twoList))));;
```

2.17 Somme des arbres :

On a utilisé la même stratégie comme la première partie de l'expérimentation. La fonction qui calcule la somme des polynômes des 15 ABR précédents est :

```
sumArbre abr152;;
```

Le temps d'exécution qu'elle a pris était 150.678879s (< 3 min)

2.18 Produit des arbres :

Idem pour cette partie, on calcule le produit de ces ABR avec **prodArbre**.

```
prodArbre abr152;;
```

Temps d'exécution : 2470.913477s (> 40min)