Concours Blanc - MP2I - Corrigé

I. Polynômes en OCaml

I. 1. Structure de données et évaluation de polynômes

Exercice 1 (Structure de données et évaluation de polynômes).
Un polynôme $P \in \mathbb{R}[X]$ peut être efficacement représenté par une liste OCaml.
Par exemple, le polynôme $X^5 + 0.5X^3 + X + 1$ peut être représenté par la liste `[1.;1.;0.;0.5;0.;1.]`

1. Écrire la liste représentant le polynôme $X^7 + 3.2X^3 + X + 2$.
2. A quelle structure de donnée abstraite correspond le type `list` OCaml ? Quelle est l'implémentation concrète sous-jacente ? Quelle différence avec le type `list` du langage Python ?
3. Écrire une fonction `deg: 'a list -> int` qui renvoie le degré d'un polynôme. On essaiera de proposer une version récursive terminale.
4. Écrire une fonction d'**exponentiation rapide récursive** `pow: float -> int -> float`. On essaiera de proposer une version récursive terminale.
5. Écrire une fonction récursive OCaml naïve `polynom_naive_eval: float list -> float -> float` qui évalue un polynôme $P(X)$ en $X = u$ pour une valeur u donnée. On essaiera de proposer une version récursive terminale.
6. Donner un ordre de grandeur asymptotique de la complexité de `polynom_naive_eval`. On justifiera proprement le calcul de complexité. On pourra utiliser une ordre de grandeur asymptotique démontré dans le cours sans démonstration.
7. On peut réduire cette complexité en utilisant le schéma de Hörner qui évalue un polynôme en utilisant la formule suivante :

$$P(u) = a_0 + x \times (a_1 + x \times (a_2 + x \times \ldots (a_{n-1} + x \times a_n)))$$

et les $(a_k)_{0 \leq k \leq n}$ sont les coefficients du polynôme P :

$$P = a_0 + a_1 X + \cdots + a_n X^n$$

Écrire une fonction `polynom_horner_eval` ayant le même prototype que la fonction `polynom_naive_eval` et implémentant le schéma de Hörner. On essaiera de proposer une version récursive terminale.

8. Donner un ordre de grandeur asymptotique de la complexité de `polynom_horner_eval`. On justifiera le calcul de complexité.

Corrigé de l'exercice 1. *[Retour à l'énoncé]*

1. La liste OCaml représentant le polynôme $X^7+3.2X^3+X+2$ est `[2.;1.;0.;3.2;0.;0.;0.;1]`
2. Le type polymorphe `list` du langage OCaml implémente une structure de données abstraite de pile immuable. L'implémentation sous-jacente est une implémentation par maillons chaînés. Cette implémentation est différente de celle des listes Python, qui sont implémentées à l'aide tableaux redimensionnables et autorisent davantage de primitives de manipulations.
3. Voici une version récursive terminale de la fonction `deg` :

```
# let deg p =
  let rec aux p acc =
  match p with
  |[] -> acc
  |a0::pp -> aux pp (acc+1)
  in
  match p with
  | [] -> failwith "empty polynom!"
  |_ -> aux p (-1);;
val deg : 'a list -> int = <fun>
```

4. Voici une version récursive terminale de la fonction `pow` :

```
# let pow x n =
  let rec aux x n acc =
    match n with
    | 0 -> acc
    | n when  (n mod 2 = 0) -> aux (x *. x) (n/2) acc
    |_ -> aux (x *. x) (n/2) (x *. acc)
  in
  if (n < 0) then
    failwith "negative exponant"
  else
    aux x n 1.;;
val pow : float -> int -> float = <fun>
# pow 2. 8;;
- : float = 256.
```

5. Voici une version récursive terminale de la fonction `polynom_naive_eval` :

```
(* evaluation naive d'un polynome utilisant l'exponentiation rapide *)
(* version recursive terminale *)
let polynom_naive_eval p x =
  let rec aux p x k acc =
    match (k,p) with
    |(k, _) when (k < 0) -> failwith "problem"
    |(_, []) -> acc
    |(k, ak::pp) -> aux pp x (k+1) (acc +. ak *. (pow x k))
  in
  aux p x 0 0.;;
val polynom_naive_eval : float list -> float -> float = <fun>
# let p = [1.; 1.; 0.; 0.5; 0.; 1.];;
val p : float list = [1.; 1.; 0.; 0.5; 0.; 1.]
# polynom_naive_eval p 1.;;
- : float = 3.5
# polynom_naive_eval p 2.;;
- : float = 39.
```

6. La complexité de l'exponentiation rapide `pow` est $T(n) \in \Theta(\log_2(n))$ où $n \in \mathbb{N}$ est l'exposant choisi en entrée (algorithme dichotomique). La complexité $T(p)$ de `polynom_naive_eval` ne dépend que du degré p du polynôme à évaluer. Notons $T(k)$ la complexité pour l'évaluation du monôme $a_k\,X^k$. On a la relation de récurrence suivante, d'après le code :

$$\begin{array}{lll} T(p+1) & = 0 & \text{(liste coef. vide, fin eval.)} \\ T(k) & = \underbrace{T(k+1)}_{\text{appel récursif éval. } a_{k+1}\,X^{k+1}} + \underbrace{\log_2(k)}_{\text{coût } \texttt{pow} \text{ pour } X^k} + \underbrace{2}_{\text{1 mult.+1 add.}} & \text{pour } 1 \leq k \leq p \\ T(0) & = 2 & \end{array}$$

Le coût est donc en :

$$T(p) \in \Theta(\sum_{k=1}^{p} \log_2(k)) \in \Theta(p\log_2(p))$$

et on a utilisé l'ordre de grandeur asymptotique vu dans le cours. Même en utilisant une exponentiation rapide, l'évaluation naïve est coûteuse (complexité linéarithmétique en le degré du polynôme évalué) !

7. Voici une version récursive terminale de la fonction `polynom_horner_eval` :

```
(* evaluation d'un polynome avec la methode de Horner: complexité linéaire! *)
(* et récursive terminale!*)
let polynom_horner_eval p x =
   let rec aux prev x acc =
     match prev with
     |[] -> acc
     |ak::pp -> aux pp x (ak +. x *. acc)
   in
   aux (List.rev p) x 0.;;
val polynom_horner_eval : float list -> float -> float = <fun>
# polynom_horner_eval p 1.;;
- : float = 3.5
# polynom_horner_eval p 2.;;
- : float = 39.
```

8. En notant toujours $p+1$ le nombre de coefficients du polynôme de degré p donné en entrée, on a :

$$T(p+1) = T(p) + 2\,, \quad T(0) = 1$$

et en explicitant la récurrence :

$$T(p) \in \Theta(p)$$

L'algorithme d'évaluation de Horner est donc linéaire en le degré p du polynôme. C'est beaucoup mieux que l'évaluation naïve.

I. 2. Opérations sur les polynômes

Exercice 2 (Opération d'addition et multiplication naïve).

1. Écrire une fonction `polynom_add: float list -> float list -> float list` réalisant la somme de deux polynômes.

2. Nous nous intéressons à présent à l'opération de multiplication de deux polynômes

$$P = a_0 + a_1 X + \cdots + a_n X^p \qquad Q = b_0 + b_1 X + \cdots + b_n X^q$$

a. On note $p = \deg(P)$ et $q = \deg(Q)$. Montrer l'existence de $(p+q+1)$ réels c_k tels que le produit $P \times Q$ puisse s'écrire :

$$P \times Q = \sum_{k=0}^{p+q} c_k X^k$$

On donnera une définition explicite de ces coefficients c_k. *Indication : vérifiez bien la validité des indices ! Prenez le temps.*

b. Pour cette fonction, et uniquement pour cette fonction, on considère que les polynômes sont stockés dans des tableaux.
Par exemple, le polynôme $X^5 + 0.5X^3 + X + 1$ peut être représenté par le tableau `[|1.;1.;0.;0.5;0.;1.|]`.
Écrire une fonction OCaml **itérative**

`polynom_naive_mult: float array -> float array -> float array`

qui implémente cet algorithme naïf de multiplication de polynômes à l'aide de la formule de multiplication montrée ci-dessus. On pourra utiliser les fonctions polymorphes `min` et `max` disponibles par défaut en OCaml. *Indication : attention, vérifiez bien que les indices utilisés dans les différents tableaux sont valides ! Revenez à la formule théorique de la question précédente si nécessaire.*

c. Donner un ordre de grandeur asymptotique de la complexité temporelle de `polynom_naive_mult`. On justifiera proprement le calcul de complexité.

Corrigé de l'exercice 2.

[Retour à l'énoncé]

1. Voici une version récursive terminale de la fonction `polynom_add` :

```
# (* addition de deux polynomes: version récursive terminale *)
(* complexité temporelle linéaire en max(degP, degQ)*)
let polynom_add p q =
  let rec aux p q acc =
    match (p, q) with
    |([],[]) -> acc
    |([], bk::qq) -> aux [] qq (bk::acc)
    |(ak::pp, []) -> aux pp [] (ak::acc)
    |(ak::pp, bk::qq) -> aux pp qq ( (ak+.bk)::acc )
  in
  List.rev (aux p q []);;
val polynom_add : float list -> float list -> float list = <fun>
# let p = [1.;1.;1.];;
val p : float list = [1.; 1.; 1.]
# let q = [0.;1.;0.;1.];;
val q : float list = [0.; 1.; 0.; 1.]
# polynom_add p q;;
- : float list = [1.; 2.; 1.; 1.]
```

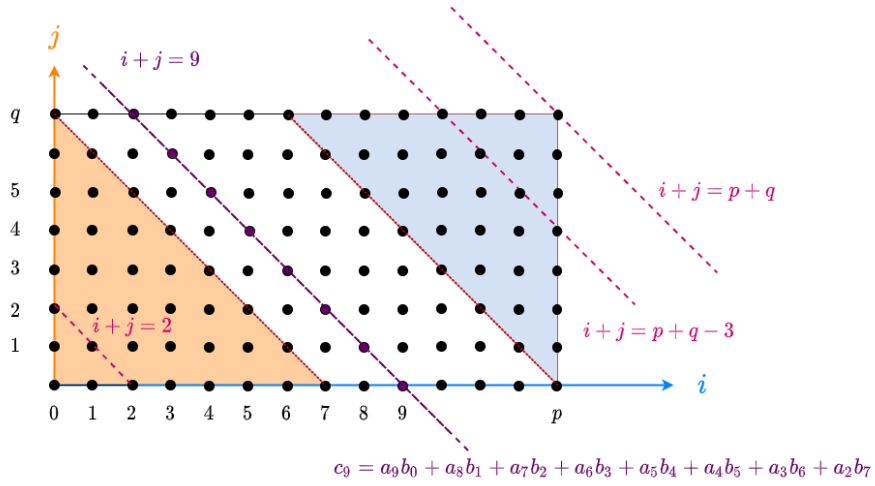
2. a. Le produit de P et de Q forme un nouveau polynôme de degré $p+q$:

$$P \times Q = \sum_{k=0}^{p+q} c_k \, X^k$$

Ce polynôme a $p+q+1$ coefficients qui s'écrivent :

$$\forall k \in [\![0;\, p+q]\!]\,, \quad c_k = \sum_{\substack{i \in [\![0,\,p]\!] \\ j \in [\![0,\,q]\!] \\ \text{tels que } i+j=k}} a_i \, b_j$$

Essayons d'expliciter cette somme, et faisons un dessin ! Comme la multiplication est commutative, on peut, sans perte de généralité, se placer dans le cas où $p \geq q$, quitte à échanger les rôles de P et Q.



Les a_i et b_j qui interviennent dans le calcul d'un coefficient c_k, avec k fixé, sont ceux dont les indices (i, j) sont tels que $i + j = k$ (il sont situés sur les lignes obliques dessinées sur le schéma) mais qui respectent également les bornes $0 \leq i \leq p$ et $0 \leq j \leq q$ c'est-à-dire les indices qui correspondent à des points existants, à l'intérieur du rectangle. On peut donc réécrire de manière plus explicite :

$$c_k = \sum_{i=\max(0,k-q)}^{\min(k,p)} a_i b_{k-i}$$

On vérifie bien que :

— Pour l'indice i : $0 \leq \max(0, k-q) \leq i \leq \min(k, p)$ donc $i \in [\![0, p]\!]$

— Pour l'indice j :

$$\begin{aligned} & 0 \leq \max(0, k-q) \leq i \leq \min(k, p) \\ \Rightarrow \quad & k - q \leq \leq i \leq k \\ \Rightarrow \quad & -k \leq -i \leq q - k \\ \Rightarrow \quad & -k + k \leq k - i \leq q - k + k \\ \Rightarrow \quad & 0 \leq j = k - i \leq q \end{aligned}$$

On remarque également que cette formule fonctionne bien dans tous les cas car le changement de variable $i \leftrightarrow j$, $p \leftrightarrow q$ aboutit à la même somme ! Nous sommes rassurés.

b. Voici une version itérative de la fonction `polynom_naive_mult` :

```
(* complexité: Theta(p*q) --> quadratique!!*)
let polynom_naive_mult p q =
  let degp = deg p in
  let degq = deg q in
  let pp = Array.of_list p in
  let qq = Array.of_list q in
  let deg_mult = degp+degq in
  let res = Array.make (deg_mult+1) 0. in
  let cnt = ref 0 in
  for k=0 to deg_mult do
    Printf.printf "\nk=%d: " k;
    let sum = ref 0. in
    for i = (max 0 (k-degq)) to (min k degp) do
      Printf.printf "(%d,%d)" i (k-i);
      sum := !sum +. pp.(i) *. qq.(k-i);
      cnt := !cnt + 1
    done;
    res.(k) <- !sum (* on stocke le coefficient ck *)
  done;
  Printf.printf "\nNombre d'operations =%d\n" !cnt;
  res;;
val polynom_naive_mult : float list -> float list -> float array = <fun>
# let p = [1.;1.;1.];;
val p : float list = [1.; 1.; 1.]
# let q = [0.;1.;0.;1.; 7.1; 3.];;
val q : float list = [0.; 1.; 0.; 1.; 7.1; 3.]
# polynom_naive_mult p q;;

k=0: (0,0)
k=1: (0,1)(1,0)
k=2: (0,2)(1,1)(2,0)
k=3: (0,3)(1,2)(2,1)
k=4: (0,4)(1,3)(2,2)
k=5: (0,5)(1,4)(2,3)
k=6: (1,5)(2,4)
k=7: (2,5)
Nombre d'operations =18
- : float array = [|0.; 1.; 1.; 2.; 8.1; 11.1; 10.1; 3.|]
```

c. La complexité dépend bien sûr des degrés p et q des deux polynômes à multiplier. Il s'agit d'une fonction itérative : la complexité se calcule directement, en sommant les opérations élémentaires effectuées dans les deux boucles imbriquées. On ne s'intéresse qu'aux multiplications par exemple. Il y a une multiplication par tour boucle :

$$T(p, q) = \sum_{k=0}^{p+q} \sum_{i=\max(0,k-q)}^{\min(k,p)} 1$$

Plaçons nous dans le cas $p \geq q$.

Si $0 \leq k < q$ (zone triangulaire orange), alors l'indice i de la somme varie de 0 à k et donc le calcul du coefficient c_k requiert de l'ordre de $k+1$ opérations élémentaires

Si $q \leq k \leq p$ (zone parallélogramme blanche), alors l'indice i de la somme varie de $k-q$ à k et donc le calcul du coefficient c_k requiert de l'ordre de $q+1$ opérations élémentaires

Si $p < k \leq p+q$ (zone triangulaire bleue), alors l'indice i de la somme varie de $k-q$ à p et donc le calcul du coefficient c_k requiert de l'ordre de $p+q-k+1$ opérations élémentaires

On a donc :

$$\begin{aligned}
T(p,q) &= \sum_{k=0}^{q-1}(k+1) + \sum_{k=q}^{p}(q+1) + \sum_{k=p+1}^{p+q}(p+q+1-k) \\
&= \sum_{k=1}^{q} k + (q+1)\times(p-q+1) + \sum_{k=1}^{q} k \\
&= \frac{q(q+1)}{2} + (q+1)\times(p-q) + \frac{q(q+1)}{2} = q(q+1) + (q+1)\times(p-q) \\
&= \cancel{q^2} + q + pq - \cancel{q^2} + p - q = pq + p \in \underset{p,q\to+\infty}{\Theta}(pq)
\end{aligned}$$

On en déduit que $T(p,q) \in \Theta(p \times q)$, l'algorithme est quadratique.

Le cas alternatif $q \geq p$ se traite de manière tout à fait symétrique et on montre de même que $T(p,q) \underset{p,q\to+\infty}{\Theta}(pq)$

Exercice 3 (Multiplication DPR).

Cette partie propose de mettre en œuvre le paradigme *diviser pour régner* (DPR) pour créer un nouvel algorithme de multiplication de deux polynômes.

1. On pose $m \in \mathbb{N}$. Justifier l'existence de deux polynômes P_1 et P_2 tels que :

$$P = X^m P_1 + P_2$$

Pour $m = \left\lceil \frac{\deg(P)}{2} \right\rceil$, quels sont les degrés respectifs de P_1 et P_2 ?

2. Écrire une fonction récursive `polynom_decomp: float list -> int -> float list*float list` qui prend un polynôme P et un entier m en entrée et renvoie la décomposition sous la forme d'un couple (P_1, P_2). On pourra utiliser les fonctions `fst` et `snd`, de prototype `'a*'a -> 'a` qui renvoient respectivement la première composante et la deuxième composante d'un couple. On essaiera de proposer une version récursive terminale.

3. Exprimer le produit $P \times Q$ en utilisant la décomposition précédente sur P et Q pour un $m \in \mathbb{N}$ fixé.

4. Écrire en OCaml une fonction récursive

```
polynom_offset: float list -> int -> float list
```

qui prend en entrée un polynôme P et un entier m et qui renvoie le polynôme $X^m \times P$. On essaiera de proposer une version récursive terminale.

5. Écrire une fonction récursive

```
polynom_dpr_mult: float list -> float list -> float list
```

multipliant deux polynômes en utilisant la formule trouvée avec $m = \left\lceil \frac{\max(\deg P, \deg Q)}{2} \right\rceil$. On pourra utiliser les fonctions précédemment mentionnées. **On ne demande pas ici une fonction récursive terminale.**

6. Prouver rigoureusement la terminaison de l'algorithme dans le cas où les deux polynômes P et Q sont de même degré, puis adapter pour le cas général.

7. Pour cette question, on suppose que les deux polynômes à multiplier sont de même degré. Donner un ordre de grandeur asymptotique de la complexité temporelle de `polynom_dpr_mult`. On justifiera le calcul de complexité.

8. Commenter.

Corrigé de l'exercice 3.

[Retour à l'énoncé]

1. L'anneau $\mathbb{R}[X]$ est un anneau euclidien.
P_2 est le reste de la division euclidienne du polynôme P par X^m, P_1 le quotient. Le reste doit vérifier :

$$\deg(P_2) < \left\lceil \frac{n}{2} \right\rceil \Rightarrow \deg(P_2) \leq \left\lfloor \frac{n}{2} \right\rfloor$$

Pour conserver le degré entre les deux expressions du polynôme, on doit avoir :

$$\deg P_2 + \deg X^m = \deg(P) \Rightarrow \deg(P_2) = \deg(P) - \left\lceil \frac{\deg(P)}{2} \right\rceil = \left\lfloor \frac{\deg(P)}{2} \right\rfloor$$

2. Voici une version récursive terminale de la fonction `polynom_decomp` :

```
(* division euclidienne d'un polynome par X^m P = P1*X^m + P2 --> renvoie (P2,P1)
*)
(* complexité linéaire *)
let polynom_decomp p m =
  let rec aux p m k acc =
    match (p, k) with
    |([],_) -> acc
    |(ak::pp, k) when k >= m -> aux pp m (k+1) (ak::(fst acc), snd acc)
    |(ak::pp, k) when k < m -> aux pp m (k+1) (fst acc, ak::(snd acc))
    |_ -> failwith "pb"
  in
  let (p1rev, p2rev) = aux p m 0 ([], []) in
  (List.rev p1rev, List.rev p2rev);;
val polynom_decomp : 'a list -> int -> 'a list * 'a list = <fun>
# let p = [1.;1.;1.;1.;2.;4.];;
val p : float list = [1.; 1.; 1.; 1.; 1.; 2.; 4.]
# polynom_decomp p 2;;
- : float list * float list = ([1.; 1.; 1.; 2.; 4.], [1.; 1.])
```

3. On pose $m = \left\lceil \frac{\max(\deg(P), \deg(Q))}{2} \right\rceil$. Alors :

$$P \times Q = (P_1 X^m + P_2) \times (Q_1 X^m + Q_2) = P_1 Q_1 X^{2m} + (P_1 Q_2 + P_2 Q_1) X^m + P_2 Q_2$$

4. Voici une version récursive terminale de la fonction `polynom_offset` :

```
(* multiplication d'un polynome par X^m --> décalage du stockage de ses coeff de 
 zéros vers la droite *)
let polynom_offset p m =
  let rec aux p m k acc =
    match (p,k) with
    |(_, k) when k < m -> aux p m (k+1) (0.::acc)
    |([],_) -> acc
    |(a::pp, k)  ->  aux pp m (k+1) (a::acc)
  in
  List.rev (aux p m  0 []);;
val polynom_offset : float list -> int -> float list = <fun>
# polynom_offset [1.; 2.; 3.] 2;;
- : float list = [0.; 0.; 1.; 2.; 3.]
```

5. Voici une version récursive (non terminale !) de la fonction `polynom_dpr_mult` :

```
(* Multiplication DPR naive *)
let rec polynom_dpr_mult p q =
  match (p,q) with
  |([],[]) -> []
  |(a0::[], _) -> List.map (fun x -> a0 *.x) q
  |(_, b0::[]) -> List.map (fun x -> b0 *.x) p
  |_ ->
  let degp = deg p in
  let degq = deg q in
  let n = if (degp > degq) then degp else degq in
  let m = if (n mod 2 = 0) then (n/2) else (n/2)+1 in
  let (p1,p2) = decomp p m and
      (q1,q2) = decomp q m in

  let p1q2 = polynom_dpr_mult p1 q2 and
      p2q1 = polynom_dpr_mult p2 q1 and
      p1q1 = polynom_dpr_mult p1 q1 and
      p2q2 = polynom_dpr_mult p2 q2 in

  let r = (polynom_add p1q2 p2q1) in
  let tmp = polynom_add p2q2 (polynom_offset r m) in (* tmp = P2Q2 + X^m*(P1Q2+Q1
2) *)
  polynom_add tmp (polynom_offset p1q1 (2*m));;
val polynom_dpr_mult : float list -> float list -> float list = <fun>
# let p = [1.;2.;3.];;
val p : float list = [1.; 2.; 3.]
# let q = [0.;1.;4.;5.];;
val q : float list = [0.; 1.; 4.; 5.]
# polynom_dpr_mult p q;;
- : float list = [0.; 1.; 6.; 16.; 22.; 15.]
```

6. On note $\mathcal{A} = \mathbb{R}[X] \times \mathbb{R}[X]$ l'espace des paramètres de la fonction `polynom_dpr_mult` et on note Φ la fonction suivante, qui va nous permettre de plonger cet espace des paramètres dans un ensemble ordonné, pour nous permettre de faire une preuve par induction :

$$\begin{array}{lcl} \mathcal{A} & \to & (\mathbb{N}\times\mathbb{N}, \preceq_{lex}) \\ (P,\,Q) & \mapsto & (\deg(P),\,\deg(Q)) \end{array}$$

Nous allons montrer par induction la terminaison de la fonction.

$$\mathcal{P}(\deg(P),\,\deg(Q)) : \text{l'appel } \texttt{polynom\_dpr\_mult p q} \text{ se termine}$$

Cas de base : $\mathbb{N}\times\mathbb{N}$ possède un unique élément minimal pour l'ordre lexicographique, qui est $(0,0)$. On note $\mathcal{M}$ les couples $(P,Q) \in \mathcal{A}$ tels que $\Phi(P,Q) = (0,0)$. Ce sont les couples (a_0, b_0) (couples de polynômes constants). D'après le code, ces cas sont traités et se terminent.

Induction : On fixe $(P,Q) \in \mathcal{A}\setminus\mathcal{M}$. Si l'un ou l'autre des polynômes est de degré 0, l'algorithme termine (cf cas de base du code). On suppose donc dans la suite que $\deg(P) \geq 1$ et $\deg Q \geq 1$.

Hypothèse d'induction : on suppose que, pour tout couple de polynôme $(U,\,V)$ tel que $\Phi(U,V) \prec_{lex} \Phi(P,Q)$, l'appel `polynom_dpr_mult u v` termine.

Prouvons que l'appel `polynom_dpr_mult p q` termine.

L'appel `polynom_dpr_mult p q` engendre 4 appels récursifs (donc en nombre fini) :

— L'appel `polynom_dpr_mult p1 q1` calcule le produit $P_1 \times Q_1$. Il s'effectue bien sur un couple (P_1,Q_1) tel que $\Phi(P_1,Q_1) \prec_{lex} \Phi(P,Q)$ car, avec $m = \left\lceil \dfrac{\max(\deg(P),\deg(Q))}{2} \right\rceil$, par construction $\deg(P_1) \leq \left\lfloor \dfrac{\deg(P)}{2} \right\rfloor < \deg(P)$ d'après la première question, et la

dernière inégalité stricte est valide car on a supposé que $\deg(P) \geq 1$. Il se termine donc.

— L'appel `polynom_dpr_mult p2 q2` calcule le produit $P_2 \times Q_2$. Il s'effectue bien sur un couple (P_2, Q_2) tel que $\Phi(P_2, Q_2) \prec_{lex} \Phi(P, Q)$ car, avec $m = \left\lceil \frac{\max(\deg(P), \deg(Q))}{2} \right\rceil$, par construction $\deg(P_2) \leq \left\lfloor \frac{\deg(P)}{2} \right\rfloor < \deg(P)$ d'après la première question, et la dernière inégalité stricte est valide car on a supposé que $\deg(P) \geq 1$. Il se termine donc.

— L'appel `polynom_dpr_mult p1 q2` calcule le produit $P_1 \times Q_2$. Il s'effectue bien sur un couple (P_1, Q_2) tel que $\Phi(P_1, Q_2) \prec_{lex} \Phi(P, Q)$ pour les mêmes raisons que ci-dessus. Il se termine donc.

— L'appel `polynom_dpr_mult p2 q1` calcule le produit $P_2 \times Q_1$. Il s'effectue bien sur un couple (P_2, Q_1) tel que $\Phi(P_2, Q_1) \prec_{lex} \Phi(P, Q)$ pour les mêmes raisons que ci-dessus. Il se termine donc.

Ainsi, les 4 appels récursifs se terminent par hypothèse d'induction et donc l'appel `polynom_dpr_mult p q` se termine.

Conclusion : Par principe d'induction, l'algorithme se termine pour toute entrée $(P, Q) \in \mathcal{A}$.

7. Comme on suppose que les deux polynômes sont de même degré n, la complexité temporelle T de la fonction ne s'écrit plus en fonction des deux paramètres p et q correspondant aux degrés des deux polynômes, mais en fonction du seul paramètre $n = p = q$. Elle vérifie la relation de récurrence suivante :

$$T(n) = 4T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + f(n)$$

et bien sûr $T(0) = 1$ (la multiplication de deux polynômes de degré 0... coûte une multiplication de flottants). Il peut arriver que les appels récursif soient sur des polynômes de degré en réalité strictement inférieur à $\left\lfloor \frac{n}{2} \right\rfloor$. Il peut en effet arriver que le reste de la division euclidienne ait un degré bien inférieur à $\left\lfloor \frac{n}{2} \right\rfloor$. L'estimation sera de toute façon une domination.
La fonction f correspond au coût de la recomposition, qui fait appel aux fonctions `polynom_add` et `polynom_offset`, toutes les deux linéaires par rapport à n. On a donc $f(n) \in \Theta(n)$.
On montre, dans le cas où n est une puissance de 2, en réinjectant (faites-le !) que :

$$T(n) \in O(n^2)$$

Il s'agit du cas du théorème maître avec $a = 4$, $b = 2$ et $\beta = 1$. On a $\log_b(a) = \log_2(4) = 2 > \beta$. Donc la complexité est en $O(n^{\log_b(a)}) = O(n^2)$.

Tout ce travail... et nous n'avons rien gagné du tout par rapport à la multiplication naïve codée plus haut. Pire : on a perdu en complexité spatiale car la fonction est récursive non terminale et les blocs d'activation s'empilent !

Exercice 4 (Algorithme de Karatsuba).

Cette question présente l'*algorithme de Karatsuba*.

1. On reprend les décompositions de P et Q vues précédemment. Montrer que le produit $P \times Q$ peut encore se mettre sous la forme équivalente suivante :

$$P \times Q = X^{2m} R_1 + X^m (R_2 - R_1 - R_3) + R_3$$

avec $R_1 = P_1 Q_1$, $R_2 = (P_1 + P_2)(Q_1 + Q_2)$ et $R_3 = P_2 Q_2$.

2. Écrire une fonction `karatsuba` utilisant cette nouvelle expression, en supposant que l'on dispose, en plus de toutes les fonctions déjà mentionnées, d'une fonction `polynom_sub: float list-> float list -> float list` qui permet de soustraire deux polynômes. **On ne demande pas ici une fonction récursive terminale.**

3. Donner un ordre de grandeur asymptotique théorique de la complexité temporelle de la fonction `karatsuba` dans le cas où l'on multiplie deux polynômes de même degré. On justifiera le calcul de complexité. Comparer avec les deux algorithmes de multiplication proposés précédemment.

Corrigé de l'exercice 4.

[Retour à l'énoncé]

1. Il suffit de calculer $R_2 - R_1 - R_3$ en développant le produit $R_2 = (P_1 + P_2)(Q_1 + Q_2)$ et de vérifier qu'il vaut bien $P_1 Q_2 + P_2 Q_1$. Allons-y :

$$R_2 - R_1 - R_3 = (P_1+P_2)(Q_1+Q_2) - P_1Q_1 - P_2Q_2 = \cancel{P_1Q_1} + P_1Q_2 + P_2Q_1 + \cancel{P_2Q_2} - \cancel{P_1Q_1} - \cancel{P_2Q_2} = P_1Q_2 + P_2Q_1$$

2. Voici une version récursive (non terminale !) de la fonction `karatsuba` :

```
(* Algo de multiplication de Karatsuba *)
let rec karatsuba p q =
  match (p,q) with
  |([],[]) -> []
  |(a0::[], _) -> List.map (fun x -> a0 *.x) q
  |(_, b0::[]) -> List.map (fun x -> b0 *.x) p
  |_ ->
  let degp = deg p in
  let degq = deg q in
  let n = if (degp > degq) then degp else degq in
  let m = if (n mod 2 = 0) then (n/2) else (n/2)+1 in
  let (p1,p2) = decomp p m and
      (q1,q2) = decomp q m in

  let r2 = karatsuba (polynom_add p1 p2) (polynom_add q1 q2) and
      r1 = karatsuba p1 q1 and
      r3 = karatsuba p2 q2 in

  let tmp = polynom_sub (polynom_sub r2 r1) r3 in
  let tmp2 = polynom_add r3 (polynom_offset tmp m) in
  polynom_add tmp2 (polynom_offset r1 (2*m));;
val karatsuba : float list -> float list -> float list = <fun>
# let p = [1.;2.;3.];;
val p : float list = [1.; 2.; 3.]
# let q = [0.;1.;4.;5.];;
val q : float list = [0.; 1.; 4.; 5.]
# karatsuba p q;;
- : float list = [0.; 1.; 6.; 16.; 22.; 15.]
```

3. On retrouve la même relation de récurrence que précédemment, mais il n'y a cette fois ci que 3 appels récursifs (3 multiplications, une pour R_1, une pour R_2, une pour R_3) au lieu de 4 :

$$T(n) = 3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + f(n)$$

et comme `polynom_sub` est elle aussi linéaire (c'est quasiment la même fonction que `polynom_add`), on a toujours $f(n) \in \Theta(n)$. On montre pour le cas où n est une puissance de 2, par réinjection (faites le !), que $T(n) \in O(n^{\log_2(3)})$. On a cette fois-ci gagné quelque chose car $\log_2(3) \approx 1.58 < 2$.

Cela peut sembler anecdotique mais représente en réalité un gain appréciable pour des polynômes de degré élevé.

Par exemple, pour multiplier deux polynômes de degré 20, $20^2 = 400$ alors que $20^{\log_2(3)} \approx 115$. Pour deux polynômes de degré 100, $100^2 = 10\,000$ alors que $100^{\log_2(3)} \approx 1479$, c'est 8 fois moins long en temps...

II. Extrait concours Mines-Ponts

II. 1. Listes triées simplement chaînées

1. Écrire en C une fonction `maillon_t *init(void)` dont la spécification suit :
   - *Effet* : crée une copie de l'ensemble vide par l'instanciation de deux nouveaux maillons sentinelles chaînés entre eux.
   - *Valeur de retour* : un pointeur vers le maillon de tête.
2. Écrire en C une fonction `maillon_t *localise(maillon_t *t, int v)` dont la spécification suit :
   - *Précondition* : le pointeur `t` désigne le maillon sentinelle de tête d'une liste chaînée.
   - *Postcondition* : en notant u le champ `donnee` du maillon désigné par la valeur de retour et u' celui du maillon successeur, on a les inégalités $u < v \leq u'$.
3. Nous souhaitons écrire une fonction `bool insere(maillon_t *t, int v)` ainsi spécifiée :
   - *Précondition* : le pointeur `t` désigne le maillon sentinelle de tête d'une liste chaînée.
   - *Postcondition* : la liste désignée par le pointeur t contient la valeur entière v ainsi que les autres valeurs précédemment contenues.
   - *Valeur de retour* : le booléen `true` si la liste contient un élément de plus et `false` sinon.

   Présenter sous forme de croquis plusieurs exemples de données d'entrées de la fonction `insere` couvrant l'ensemble des valeurs de retour possibles. Dans chaque cas, on dessinera les états initial et final de la liste à la manière de la figure 1 et on donnera la valeur de retour.
4. Nous proposons le code erroné suivant :

```
6.  bool insere_errone(maillon_t *t, int v) {
7.      maillon_t *p = localise(&t, v);
8.      maillon_t *n = malloc(sizeof(maillon_t));
9.      n->suivant = p->suivant;
10.     n->donnee = v;
11.     p->suivant = n;
12.     return true;
13. }
```

   Le compilateur produit le message d'erreur

```
incompatible pointer types passing 'maillon_t **'
to parameter of type 'maillon_t *'
```

   Expliquer ce message et proposer une première correction.
5. Indiquer le ou les tests de la question 3 manqués par la fonction `insere_errone`. `insere_errone`. Écrire une fonction `insere` correcte respectant scrupuleusement la spécification et en appliquant les principes de la programmation défensive.
6. Écrire en C une fonction `bool supprime(maillon_t *t, int v)` dont la spécification suit :
   - *Précondition* : le pointeur `t` désigne le maillon sentinelle de tête d'une liste chaînée.
   - *Postcondition* : la liste désignée par le pointeur t ne contient pas la valeur entière v mais contient les autres valeurs précédemment contenues.
   - *Valeur de retour* : le booléen `true` si la liste contient un élément de moins et `false` sinon.

7. Calculer la complexité en temps des fonctions `insere` et `supprime`.
8. Un programme C peut stocker des données dans différentes régions de la mémoire. Citer ces régions. Dire dans laquelle ou dans lesquelles de ces régions la valeur entière 717 est inscrite lorsque nous exécutons le programme suivant.

```
int v = 717;

int main(void) {
    maillon_t *t = init();
    insere(t, v);

    return 0;
}
```

Corrigé de l'exercice 5.

[Retour à l'énoncé]

1. Voici une implémentation de `init` conforme aux spécifications :

```
maillon_t *init(void)
{
  maillon_t *m1 = malloc(sizeof(maillon_t));
  assert(m1 != NULL);

  maillon_t *m2 = malloc(sizeof(maillon_t));
  assert(m2 != NULL);

  m1->donnee= INT_MIN;
  m1->suivant = m2;
  m2->donnee = INT_MAX;
  m2->suivant = NULL;

  return m1;
}
```

2. Voici une implémentation de `localise` conforme aux spécifications. Attention ici, à l'inégalité stricte dans $u < v \leq u'$.

```
maillon_t *localise(maillon_t *t, int v)
{
  assert(t != NULL);
  maillon_t *p = t;
  maillon_t *n = t->suivant;

  int uprim;

  // tant que le suivant n'est pas le
  //maillon sentinelle de fin
  while (n->donnee != INT_MAX)
  {

    uprim = n->donnee;

    if (v <= uprim)
      break;

    p = n;

    assert(n->suivant != NULL);
    n = n->suivant;

  }

  return p;
}
```

3. Si la valeur n'est pas présente, elle sera insérée et le code doit renvoyer `true`. Sinon, il ne faut pas réinsérer la valeur.
4. Le compilateur repère une incohérence syntaxique car la fonction `localise` attend un premier paramètre de type `maillon_t *` mais le code appelle cette fonction en lui donnant comme premier paramètre l'adresse (opérateur &) d'un pointeur de type `maillon_t *`, donc un objet de type `maillon_t **` Il suffit de retirer l'opérateur d'adresse & pour corriger cette erreur.
5. La fonction proposée ne gère pas le cas où la valeur que l'on souhaite insérer est déjà présente dans la liste. Voici une implémentation de `insere` corrigée, et conforme aux spécifications.

```
bool insere(maillon_t *t, int v)
{
  // maillon précédent le nouveau maillon à inserer
  // coût de localisation linéaire (recherche séquentielle *)
  maillon_t *p = localise(t, v);
  if (p->donnee == v)
    return false; // on n'insère pas: pas de doublons dans un ensemble

  // nouveau maillon
  maillon_t *n = malloc(sizeof(maillon_t));
  assert(n != NULL); // gestion d'un eventuel probleme d'allocation

  // côut d'insertion une fois la zone d'insertion localisee constant
  n->donnee = v;
  n->suivant = p->suivant;
  p->suivant = n;

  return true;
}
```

6. Voici une implémentation de `supprime` conforme aux spécifications.

```
bool supprime(maillon_t *t, int v)
{
  // coût de localisation linéaire (recherche séquentielle *)
  maillon_t *p = localise(t, v);

  if (p->suivant->donnee == v)
  {
    // cout de suppression constant une fois le maillon localisé
    maillon_t *s = p->suivant;
    p->suivant = p->suivant->suivant;
    free(s);
    return true;
  }
  else
    return false;
}
```

7. Pour ces deux fonctions `insere` et `supprime`, le coût vient essentiellement de la localisation : localisation de la zone d'insertion pour `insere`, localisation du maillon à supprimer pour `supprime`. Cette localisation nécessite le parcours de la liste en suivant les adresses successives d'un maillon à l'autre. Le coût est en $O(n)$, où n est la taille courante de la liste. Une fois la localisation effectuée, l'opération d'insertion ou de suppression en elle-même s'effectue en temps constant car elle consiste à rebrancher les adresses de manière cohérente par quelques copies d'adresse, auxquels s'ajoutent une copie de valeur dans le champ donnée pour l'insertion, et une libération mémoire pour la suppression. Le coût total est donc en $O(n)$ pour ces deux fonctions.
8. La mémoire virtuelle allouée à un processus est découpée en segments logiques : segment de code, segment de données statiques (`rodata`, `data`, `bss`..), segment de pile, segment de tas notamment et éventuellement d'autres, qui dépendent du système d'exploitation, de l'architecture... Un programme C peut stocker des données dans différentes régions de la mémoire.

Dans le code suivant, la valeur 717 est d'abord stockée dans le segment de données statique. Toutefois, lors du passage de cette valeur en argument de la fonction `insere`, une copie de cette valeur est réalisée dans une zone mémoire située dans le bloc d'activation de l'appel de la fonction `insere`, dans le segment de pile (passage par valeurs). Puis, à l'intérieur de la fonction `insere`, une zone mémoire correspondant à un nouveau maillon est allouée dans le segment du tas, et la valeur 717 est copiée dans la zone mémoire du tas correspond au champ `donnee` de ce maillon. La valeur 717 est donc présente à trois endroits dans la mémoire du processus :

— dans le segment de données statiques, et elle pourra y rester de manière pérenne jusqu'à la fin de l'exécution

— dans le segment de pile, dans le bloc d'activation de la fonction `insere`, mais cette copie de 717 sera détruite à la fin de l'exécution de la fonction `insere`, lorsque la zone mémoire correspondant à ce bloc d'activation sera libérée

— dans le segment du tas, dans la zone mémoire allouée au nouveau maillon. Elle restera stockée là de manière pérenne, jusqu'à ce que la zone mémoire allouée au maillon soit libérée manuellement par le programmeur par un appel à la fonction `free`.

II. 2. Extensions des opérations de base

Nous disposons d'une implémentation alternative du type abstrait EnsembleEntiers par une structure de données d'arbre binaire de recherche.

1. Définir le terme arbre binaire de recherche. Citer une propriété désirable afin que la complexité en temps des trois primitives (insertion, suppression et test d'appartenance) soit logarithmique. Nommer un exemple d'arbre binaire de recherche qui jouit de cette propriété.
2. Décrire, en langue française ou par du pseudo-code, un algorithme aussi efficace que possible qui transforme un ensemble représenté sous forme d'arbre binaire de recherche en un ensemble représenté sous forme d'une liste chaînée avec sentinelles. Donner sa complexité en temps et en espace.
3. Nous souhaitons compléter l'implémentation du type EnsembleEntiers de la section 1.1 par une primitive supplémentaire qui renvoie un élément aléatoirement choisi dans un ensemble non vide. Nous considérons que l'expression `random()%z` engendre un entier aléatoire choisi uniformément entre 0 et $z-1$ et évacuons toute préoccupation relative à la validité de z.

   Dans une première ébauche, nous proposons le code suivant. La fonction `random` fonctionne exactement de la même manière que `rand`. On suppose que son initialisation a été effectuée.

```
20. int random_elt(maillon_t *t) {
21.     maillon_t *c = t->suivant;
22.     int ret = c->donnee;
23.     if (c->donnee == INT_MAX) {
24.         assert(false);
25.     }
26.     int z = 2;
27.     while ((c->suivant)->donnee != INT_MAX) {
28.         c = c->suivant;
29.         if (random()%z) {
30.             ret = c->donnee;
31.         }
32.     }
33.     return ret;
34. }
```

(a) Décrire avec quelle probabilité l'expression `random_elt(t)` renvoie un élément u_i lorsque le pointeur t désigne un ensemble à n éléments dont u_i est le i-ième élément en numérotant à partir de 0.

(b) Modifier, si besoin, la fonction `random_elt` afin qu'elle renvoie un élément distribué selon une loi de probabilité uniforme (i.e. chaque élément de l'ensemble est équiprobable).

Alternativement à ce qui précède et uniquement dans la question suivante, nous envisageons de représenter des ensembles de flottants.

4. Dire quelles difficultés supplémentaires il y aurait à manipuler des listes dont les données sont de type double plutôt que de type `int`.

Corrigé de l'exercice 6. *[Retour à l'énoncé]*

1. La notion d'arbre binaire de recherche (ABR) est définie de manière inductive. Un ABR est :

— soit l'arbre vide,

— soit un arbre étiqueté dont les étiquettes sont choisies dans un ensemble $(E, \preceq)$ totalement ordonné et tel que, pour un nœud $N(\ell, v, r)$, où ℓ désigne le sous-arbre gauche, r le sous-arbre droit et v son étiquette, on a :

— Toutes les étiquettes des nœuds de ℓ sont strictement inférieures à v pour la relation d'ordre $\preceq$

— Toutes les étiquettes des nœuds de r sont strictement supérieures à v pour la relation d'ordre $\preceq$

Comme on mentionne ici les ABR pour implémenter une structure de données d'ensemble, on exclut bien entendu le cas des doublons.

Pour que les trois primitives (insertion, suppression et test d'appartenance) de la structure de donnée d'ensemble s'effectuent avec une complexité temporelle logarithmique, l'ABR utilisé pour l'implémentation concrète doit être équilibré.

Les arbres AVL ou les arbres bicolores (arbres rouge-noir) sont des ABR équilibrés.

2. Il suffit ici de décrire une fonction effectuant le parcours infixe d'un ABR, qui renvoie la liste des étiquettes classées dans l'ordre croissant. Nous l'avons déjà codé en C, il suffit de bien rajouter les deux sentinelles.

Complexité temporelle : il s'agit simplement de parcourir l'ABR, la complexité temporelle est donc proportionnelle au nombre de nœuds de l'arbre est donc linéaire en le nombre d'éléments de l'ensemble.

Complexité spatiale : La complexité dans le segment de tas est constante (pas d'allocation dans le tas autre que les données). Il s'agit essentiellement ici d'étudier la complexité dans le segment de pile. Comme la fonction n'est pas récursive terminale, les blocs d'activation s'empilent. La complexité spatiale dans la pile est dominée par le nombre maximum de blocs empilés qu'il peut y avoir au cours de l'exécution. Celui-ci est égal à la hauteur de l'ABR. Si l'ABR est équilibré, cette complexité spatiale dans la pile est donc logarithmique en le nombre d'éléments de l'ensemble.

3. a. On parcourt toute la liste et la valeur retournée est la dernière valeur paire rencontrée. La fonction a une chance sur 2 (s'il est pair) de renvoyer le dernier élément (le plus grand) de la liste, une chance sur 4 l'avant dernier...

$$\mathbb{P}(X = u_i) = \frac{1}{2^{n-i}},\ 1 \leq i \leq n-1$$

La première valeur n'est renvoyée que si tous les éléments qui suivent sont impairs.

$$\mathbb{P}(X = u_0) = \frac{1}{2^{n-1}}$$

On vérifie bien (bon réflexe à prendre !) que la somme de toutes ces probabilités vaut 1 :

$$\begin{aligned}\sum_{i=0}^{n-1}\mathbb{P}(X=u_i) &= \frac{1}{2^{n-1}}+\sum_{i=1}^{n-1}\frac{1}{2^{n-i}}\\ &= \frac{1}{2^{n-1}}+\sum_{i=1}^{n-1}\frac{1}{2^i} = \frac{1}{2^{n-1}}+\frac{1}{2}\times\sum_{i=0}^{n-2}\frac{1}{2^i}\\ &= \frac{1}{2^{n-1}}+\frac{1}{2}\times\frac{1-\left(\frac{1}{2}\right)^{n-1}}{1-\frac{1}{2}} = \frac{1}{2^{n-1}}+\frac{1-\left(\frac{1}{2}\right)^{n-1}}{2-2\times\frac{1}{2}}\\ &= \frac{1}{2^{n-1}}+\left(1-\left(\frac{1}{2}\right)^{n-1}\right) = 1\end{aligned}$$

b. Voici une implémentation de `random_elt` permettant de renvoyer un élément en respectant un loi de distribution uniforme :

```c
int random_elt(maillon_t *t)
{
  maillon_t *c = t->suivant;

  if (c->donnee == INT_MAX)
    assert(false); // liste vide


  // calcul du nombre n d'éléments de l'ensemble
  int n = 0;
  while ( c->donnee != INT_MAX) {
    c = c->suivant;
    n = n + 1;
  }
  printf("Nombre d'éléments de la liste: %d\n", n);
  // pioche uniforme de l'un de ces éléments (indice de 0 à n-1)
  int idx  = random()%n;

  c = t->suivant;
  for (int i = 0; i < idx; i++)
    c = c->suivant;

  return c->donnee;
}
```

4. On ne peut représenter de manière exacte l'ensemble des réels sur un nombre finis de bits (64 bits en double précision). En particulier, les tests d'égalité et d'inégalité entre nombres flottants, même en double précision, sont problématiques, notamment lorsque l'on cherche à comparer des nombres ayant des ordres de grandeur très différents. A titre d'exemple, en double précision, $1+10^{-18}$ est égal à 1 (la précision relative pour un encore norme IEEE754 en double précision est environ de 10^{-16}) : on n'est donc plus en mesure de différencier, du point de vue numérique, l'encodage de ces deux nombres. Les fonctions d'insertion et de suppression doivent être adaptées, avec des stratégies de normalisation et l'introduction de ε, pour prendre en compte les erreurs d'arrondis et les problématiques de représentation des nombres réels en machine.