Concours Blanc - MP2I

Durée 3 heures - Documents et calculatrice interdits

Ce sujet comporte 7 pages. Il est composé de deux problèmes : le premier sera traité en langage OCaml, le second en langage C.

Avertissement : Il est rappelé qu'une copie doit être rendue lisible, compréhensible et propre, l'objectif étant de montrer que l'on a compris ou trouvé, et de bien se faire comprendre du correcteur.
En particulier :

- Sa rédaction doit permettre de lire et de comprendre un raisonnement ou un programme sans décryptage. Elle ne doit en aucun cas être énigmatique.
- Elle doit être rédigée en français, avec sa grammaire et son orthographe. Les abréviations abusives doivent être évitées.
- Une conclusion doit clairement apparaître en réponse à une question et être lisible.
- On prendra soin d'indiquer son nom sur chaque copie et de les numéroter.

La note finale tiendra compte de la qualité de la rédaction.

Vous numéroterez chaque copie double, en prenant soin de mettre votre nom sur chaque copie.

Les stylos effaçables et le crayon à papier sont interdits.

I. Polynômes en OCaml

Toutes les fonctions de ce problème seront écrites en langage OCaml.

Sauf mention explicite d'une indication contraire, **toutes les fonctions seront implémentées de manière récursive, en utilisant des structures de données immuables.**

Pour toutes les fonctions récursives, **on essaiera, dans la mesure du possible, de proposer une implémentation récursive terminale**.
Cependant, une version non terminale correcte sera toujours mieux notée qu'une version terminale incorrecte.
En cas de doute sur la version terminale, je vous invite à écrire une version non terminale puis, à proposer en dessous votre tentative de version terminale, pour sécuriser l'obtention de points.

Les seules fonctions du module `List` qui sont autorisées sont les fonctions `List.rev` et `List.map`. **Aucune autre fonction du module `List` et aucun autre module ne sont autorisés.** Si la fonction `List.rev` est utilisée, elle devra l'être avec beaucoup de précaution pour ne pas détériorer la complexité des fonctions récursives.

Cet exercice présente plusieurs algorithmes de calcul du produit $P \times Q$ de deux polynômes P et Q de $\mathbb{R}[X]$, dont l'*algorithme de Karatsuba.*

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.

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 :

$$P = a_0 + a_1X + \cdots + a_nX^p \qquad Q = b_0 + b_1X + \cdots + b_nX^q$$

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.

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 de $\mathbb{R}[X]$ tels que $\deg(P_2) < \deg(X^m)$ et :
$$P = X^m P_1 + P_2$$
Pour $m = \left\lceil \frac{\deg(P)}{2} \right\rceil$, que vaut le degré de P_1 ?

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.

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.

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.

II. Extrait concours Mines-Ponts

Nous supposons définies deux constantes entières `INT_MIN` et `INT_MAX` qui désignent respectivement le plus petit entier et le plus grand entier représentables par le type `int` en machine. Elles sont représentées par $-\infty$ et ∞ dans les schémas.

Nous introduisons une structure maillon constituée de deux champs par la déclaration suivante.

```c
struct maillon {
    int donnee;
    struct maillon *suivant;
};
typedef struct maillon maillon_t;
```

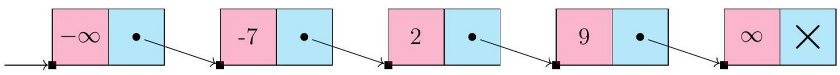
Dans l'ensemble de cette partie, nous réalisons le type abstrait EnsembleEntiers à l'aide d'une liste de maillons simplement chaînés. Nous supposons que tous les entiers insérés, supprimés ou recherchés sont strictement compris entre `INT_MIN` et `INT_MAX`. Nous maintenons les trois invariants suivants :

— Pour tous maillons m et m' consécutifs dans la liste chaînée, de champs `donnee` respectifs u et u', on a l'inégalité $u < u'$. Autrement dit, la liste est triée et ne contient pas de doublons.

— La liste est encadrée par deux maillons sentinelles ayant `INT_MIN` comme champ `donnee` en tête de liste et `INT_MAX` en fin de liste.

— Pour toute valeur entière u contenue dans l'ensemble, il existe un maillon accessible depuis le maillon sentinelle de tête ayant u comme champ `donnee`.

Par exemple, l'ensemble $\{2, -7, 9\}$ est représenté par la liste chaînée dessinée en figure 1. Dans cette figure, la croix représente la valeur d'adresse `NULL`.



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 :

```
bool insere_errone(maillon_t *t, int v) {
    maillon_t *p = localise(&t, v);
    maillon_t *n = malloc(sizeof(maillon_t));
    n->suivant = p->suivant;
    n->donnee = v;
    p->suivant = n;
    return true;
}
```

   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. Discerner le ou les tests de la question 3 manqués par la fonction `insere_errone` puis 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;
}
```

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.

```
int random_elt(maillon_t *t) {
    maillon_t *c = t->suivant;
    int ret = c->donnee;
    if (c->donnee == INT_MAX) {
        assert(false);
    }
    int z = 2;
    while ((c->suivant)->donnee != INT_MAX) {
        c = c->suivant;
        if (random()%z) {
            ret = c->donnee;
        }
    }
    return ret;
}
```

   (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`.