Devoir d'été

8 juin 2024

INFORMATIQUE MP2I -> MPI

DURÉE DE L'ÉPREUVE: **4 heures**

L'usage de la calculatrice et de tout dispositif électronique est interdit.

Ce sujet comporte huit pages numérotées de 1/8 à 8/8

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il est amené à prendre.

Tournez la page S.V.P.

Vue d'ensemble du sujet

Ce sujet est composé de 2 parties indépendantes, utilisant les langage de programmation C et OCaml. La première partie est composée de 4 sous-parties reliées entre elles. La seconde partie est composée de 3 sous-parties indépendantes.

Les différentes parties peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

— La partie I s'intéresse à la multiplication rapide de polynômes par la transformée de Fourier rapide (FFT) et passe par l'implémentation de nombres complexes, utilisant les nombres flottants dont on étudie un algorithme améliorant la précision, ainsi que diverses fonctions sur les polynômes.

— La partie II traite de la logique et propose trois sous-parties indépendants, l'une résolvant un problème de logique à l'aide de la logique des propositions, la deuxième traite de la logique du premier ordre et la dernière étudie des formules logiques n'étant formées qu'avec le constructeur ternaire conditionnel.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus. En OCaml, les noms des arguments des fonctions sont donnés avec des indications de type. Il n'est pas nécessaire de recopier ces indications de type sur votre copie.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle.

1 Multiplications de polynômes

1.1 Implémentation des nombres complexes (C) – 14 pts

Dans cette partie, on s'intéresse à l'implémentation de fonctions de bases pour manipuler les nombres complexes en C. On définit le type structuré suivant pour les représenter :

```
struct complexe_s {
  double re;
  double im;
};
typedef struct complexe_s complexe_t;
```

☐ 1 – Écrire une fonction `complexe_t complexe(double x, double y)` qui prend en entrée deux nombres réels x et y et qui revoie le nombre complexe $x + \mathrm{i}y$.

☐ 2 – Écrire une fonction `complexe_t prod(complexe_t c1, complexe_t c2)` qui prend en entrée deux nombres complexes c_1 et c_2 et qui renvoie leur produit $c_1 c_2$.

☐ 3 – En mathématiques, l'ensemble de Mandelbrot est une fractale définie comme l'ensemble des points c du plan complexe pour lesquels la suite de nombres complexes définie par récurrence par :

$$\begin{cases} z_0 = 0 \\ z_{n+1} = z_n^2 + c \end{cases}$$

est bornée.

Écrire une fonction non-récursive `complexe_t mandelbrot(complexe_t c, int n)` qui calcule z_n à l'aide d'une boucle.

☐ 4 – S'il existe n tel que $|z_n| > 2$, alors la suite diverge[1] et c n'est pas dans l'ensemble de Mandelbrot. On suppose qu'on a définit une variable globale :

```
int M = 1000000;
```

Écrire une fonction `bool in_mandelbrot(complexe_t c)` qui renvoie `false` s'il existe $n \in [\![0, M-1]\!]$ tel que $|z_n| > 2$ et `true` sinon. Donner la complexité de votre algorithme dans le pire des cas en fonction de M.

☐ 5 – On note $\omega_N = \mathrm{e}^{2\mathrm{i}\pi/N}$. L'ensemble des racines N-ièmes de l'unité est donc $\left\{\omega_N^k \;\middle|\; k \in [\![0, N-1]\!]\right\}$.
Écrire une fonction `complexe_t expo(complexe_t w, int k)` qui prend en entrée ω_N et k et qui calcule ω_N^k. La complexité doit être $O(\log k)$ dans le pire des cas.

☐ 6 – Écrire une fonction `complexe_t racine(complexe_t w)` qui prend en entrée ω_N et qui calcule ω_{2N}. La correction de cette fonction doit être justifiée.
On pourra utiliser la fonction `sqrt`, et s'intéresser entre autre aux modules de ω_N et ω_{2N}.

1.2 Sur la précision des calculs (C) – 14 pts

Un nombre réel x est représenté en machine en base 2 par un flottant qui a un signe s, une mantisse m et un exposant e tel que $x = s \times m \times 2^e$. Dans la norme IEEE 754, en convention normalisé la partie entière de la mantisse est 1 qui est un bit caché. En simple précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 23 bits et l'exposant sur 8 bits. En double précision, le signe est codé sur 1 bit, la partie décimale de la mantisse sur 52 bits et l'exposant sur 11 bits. Dans cet exercice, on observe le résultat de calculs obtenus par un programme. On pourra utiliser la fonction de signature : `double pow(double v, double p)` qui calcule v^p.
☐ 7 – On définit les valeurs suivantes :

```
    double a=2e65;
    double b=-2e65;
    double c=1.0;
```

Donner et Expliquer les valeurs prises par `(a+b)+c` et `a+(b+c)`.

☐ 8 – Compte tenu des approximations faites lors du codage, on peut trouver plusieurs nombres x tels que $1 + x = 1$ après un calcul fait par la machine. Le plus petit nombre représentable exactement en machine et supérieur à 1 s'écrit $1 + \epsilon$, avec ϵ un réel appelé ϵ machine. On admet que ϵ s'écrit sous la forme 2^{-n} avec n un entier naturel strictement positif. Écrire une fonction de signature `double epsilon()` qui renvoie la valeur de n.

☐ 9 – On définit une liste chaînée de nombres à l'aide d'une structure `nb` comportant un double et un pointeur vers une structure `nb` définie ci-dessous

```
struct nb {double x; struct nb* suivant;};
```

Écrire une fonction de signature `double somme(struct nb* tab)` qui calcule la somme des éléments de la liste `tab`.

[1] bonus s'il reste du temps : le démontrer

L'algorithme suivant permet d'augmenter la précision du calcul lors du calcul de la somme d'une liste d'éléments (i.e., un ensemble d'éléments possédant éventuellement des doublons, pas nécessairement une liste chaînée).

Entrée : Une liste l de réels de taille au moins 2.
Sortie : La somme des réels contenus dans la liste l.
tant que *la liste l contient strictement plus d'un élément*
  Calculer la somme $s = x + y$ des deux plus petits éléments x et y de l
  Supprimer x et y de l
  Insérer s dans l
renvoyer *l'unique élément de l*

☐ 10 – Montrer la terminaison de cet algorithme à l'aide d'un variant.

☐ 11 – Montrer la correction partielle de cet algorithme à l'aide d'un invariant.

☐ 12 – Nommer une structure de donnée abstraite adaptée pour appliquer cet algorithme, ainsi qu'une structure de donnée concrète permettant de l'implémenter.
Donner la complexité des opérations élémentaires pour cette structure de données, et en déduire la complexité de l'algorithme calculant la somme des éléments.

1.3 Opérations élémentaires sur les polynômes (C) – 12 pts

Dans cette partie, on implémente les polynômes à coefficient dans $\mathbb{C}$ en C par des tableaux à l'aide du type structuré suivant :

```
struct polynome_s {
  int deg;
  complexe_t *co;
};
typedef struct polynome_s polynome_t;
```

La case `co[i]` contient le coefficient du monôme en X^i.
Remarquons que le nombre de coefficients d'un polynôme de degré d est $d+1$, on a donc pas besoin de stocker la taille du tableau `co`.
☐ 13 – Écrire une fonction `polynome_t *nouv_poly(int d)` qui prend en entrée un entier d et qui renvoie un pointeur vers un nouveau polynôme de degré d dont tous les coefficients valent 0.

☐ 14 – Écrire une fonction `void libere_poly(polynome_t *P)` qui permet de libérer la mémoire utilisée par un polynôme P. On ne demande pas de vérifier qu'un pointeur est non-nul avant de libérer la mémoire qu'il désigne.

☐ 15 – Écrire une fonction `complexe_t eval(polynome_t *P, complexe_t x)` qui prend en argument un polynôme P et un nombre complexe x et qui calcule $P(x)$ en utilisant au plus $d+1$ multiplications de nombres complexes.

☐ 16 – Écrire une fonction `polynome_t *prod_poly(polynome_t *P1, polynome_t *P2)` qui calcule le produit des polynômes P_1 et P_2 de degrés respectifs d_1 et d_2 en temps $O(d_1 d_2)$.

☐ 17 – Expliquer brièvement comment on pourrait calculer un polynôme P de degré d à partir de $d+1$ couples $(x_i, P(x_i))$ deux à deux distincts.

1.4 Multiplication rapide via la transformée de Fourier rapide (C) – 16 pts

Dans cette partie, on considère 2 représentations différentes des polynômes :

Par une liste de coefficients de type `type coeffs = Complex.t list`, le coefficient de tête étant le coefficient constant ;

Par un tableau de valeurs de type `type valeurs = Complex.t array` de taille N contenant les $P(\omega_N^i)$, $i \in [\![0, N-1]\!]$, où N est une puissance de 2.

On suppose disposer de toutes les fonctions et constantes définies dans le module `Complex` via `open Complex`. On en rappelle ici quelques unes :

le nombre 0 **:** `zero : t`;

le nombre 1 **:** `one : t`;

le nombre i **:** `i : t`;

la négation unaire : `neg : t -> t`;

l'addition : `add : t -> t -> t`;

la soustraction : `sub : t -> t -> t`;

la multiplication : `mul : t -> t -> t`;

la puissance : `pow : t -> t -> t`;

le nombre $re^{i\theta}$ **:** `polar : float -> float -> t`.

On suppose également que l'on a à notre disposition toutes les fonctions que l'on a implémenté précédemment en C :

`.t -> Complex.t` etc.

☐ 18 – Écrire une fonction `prod_poly : valeurs -> valeurs -> valeurs` calculant le produit de deux polynômes p_1 et p_2 donnés par des tableaux de valeurs de même taille n. Donner sa complexité en fonction de n.

☐ 19 – Expliquer pourquoi passer naïvement par des tableaux de valeurs à l'aide de la fonction `eval` ne permet pas d'être plus efficace que la multiplication naïve des polynômes p_1 et p_2 de degrés respectifs d_1 et d_2 en $O(d_1 d_2)$.

Afin de rendre plus efficace la conversion de liste de coefficients à tableau de valeurs, on utilise une stratégie de type diviser pour régner : Si on note p_0 le polynôme obtenu en ne gardant que les éléments d'indice pair de la liste des coefficients de p et p_1 celui obtenu en ne gardant que les éléments d'indice impair, on a les relations

$$p(X) = p_0(X^2) + Xp_1(X^2), \qquad p(-X) = p_0(X^2) - Xp_1(X^2)$$

☐ 20 – Écrire une fonction `split : 'a list -> 'a list * 'a list` qui prend en entrée une liste ℓ et qui renvoie deux listes ℓ_0 et ℓ_1 contenant respectivement les éléments aux indices pairs dans ℓ et ceux aux indices impairs. La complexité doit être en $O(|\ell|)$ dans le pire des cas.

Si on choisit judicieusement les valeurs de X auxquelles on évalue le polynôme, on peut avoir à chaque fois un nombre ω et son opposé $-\omega$, dont les calculs utilisent tous deux les mêmes valeurs $p_0(\omega^2)$ et $p_1(\omega^2)$ que l'on peut réutiliser. On décide d'évaluer le polynôme en les racines N-ièmes de l'unité, où N est une puissance de 2.
☐ 21 – Montrer que $-\omega_{2N}^k = \omega_{2N}^{k+N}$.

☐ 22 – Écrire une fonction récursive `fft : coeffs -> Complex.t -> valeurs` qui prend en entrée un polynôme p de degré $N-1$ sous forme de liste de coefficients, ainsi que la valeur ω_N, avec N une puissance de 2.

☐ 23 – Donner une relation de récurrence vérifiée par la complexité de la fonction `fft` et en déduire sa complexité (on pourra citer un algorithme connu dont la complexité vérifie la même relation).

En remarquant que le passage d'une représentation à l'autre correspond en réalité à la multiplication par une matrice de Vandermonde, on peut remarquer que l'opération réciproque est en réalité très similaire, et qu'on peut appliquer la même méthode avec une autre valeur de départ (à la place de ω_N) pour obtenir la liste des coefficients, à un facteur multiplicatif $2N$ près. Cette technique est encore aujourd'hui l'état-de-l'art pour la multiplication de grands polynômes.

2 Un peu de logique

2.1 À la recherche de l'île Maya – 12 pts

Dans cet exercice, vous introduirez les variables propositionnelles qui vous semblent nécessaires.

Un voyageur navigue dans un archipel à la recherche de l'île Maya. Les îles de cet archipel sont occupées par d'étranges habitants : les Pires, qui mentent systématiquement ; et les Purs, qui disent toujours la vérité. Sur chaque île, le voyageur rencontre deux habitants, A et B qui acceptent de lui parler.

Sur la première île, A et B tiennent les propos suivants :

A : B est un Pur et nous sommes sur l'île Maya.

B : A est un Pire et nous sommes sur l'île Maya.

☐ 24 – Modéliser la situation à l'aide d'une formule F_1 du calcul propositionnel.

☐ 25 – Mettre F_1 sous forme normale conjonctive sans utiliser de table de vérité.

☐ 26 – Appliquer l'algorithme de Quine à la CNF obtenue à la question précédente et en déduire si le voyageur est arrivé sur l'île Maya et, si possible, la nature de A et B.

Sur la deuxième île, A et B tiennent les propos suivants :

A : Nous sommes deux Pires, et nous sommes sur l'île Maya.

B : L'un de nous au moins est un Pire, et nous ne sommes pas sur l'île Maya.

☐ 27 – Modéliser la situation à l'aide d'une formule F_2 du calcul propositionnel.

☐ 28 – Établir la table de vérité de F_2 et en déduire une formule équivalente à F_2 sous forme normale disjonctive.

☐ 29 – Déterminer si le voyageur est arrivé sur l'île Maya.

2.2 Vocabulaire autour de la logique du premier ordre – 12 pts

Dans cette partie, x, y, z sont des symboles de variables. Soit F la formule du premier ordre définie par :

$$F = (\forall x.\exists y.f(g(x,y),a,z)) \wedge (\forall z.f(x,g(x,Q(a)),z))$$

☐ 30 – Pour chaque symbole dans F, dire si c'est un symbole de fonction ou un symbole de relation et donner son arité.

☐ 31 – Déterminer l'ensemble des termes et des formules atomiques de F.

☐ 32 – Pour chaque occurence de variable dans F, indiquer si elle est libre ou liée ; dans le cas où elle est liée, préciser par quel quantificateur.

☐ 33 – Effectuer la substitution $F^{\{z:=Q(x)\}}$.

2.3 Formules ordonnées (OCaml) – 20 pts

Dans ce problème, on considère des formules de logique propositionnelle sur n variables notées X_i, avec $0 \leq i < n$. Une valeur de vérité est un élément de $\mathbb{B} = \{V, F\}$. Une valuation est une fonction v de $\{0, 1, \ldots, n-1\}$ dans $\mathbb{B}$, qui assigne une valeur de vérité à chacune des variables. On note $\mathcal{V}_v(f)$ la valeur de vérité de la formule f pour la valuation v.

On choisit de représenter nos formules en OCaml avec le type suivant.

```
type formula =
  | True
  | False
  | If of int * formula * formula
```

La valeur de vérité d'une telle formule est définie de la manière suivante :

$$\begin{aligned} \mathcal{V}_v(\texttt{True}) &= V \\ \mathcal{V}_v(\texttt{False}) &= F \\ \mathcal{V}_v(\texttt{If}(i, f, g)) &= \text{si } v(i) = V \text{ alors } \mathcal{V}_v(f) \text{ sinon } \mathcal{V}_v(g) \end{aligned}$$

On ajoute par ailleurs la contrainte que toute formule est *ordonnée*, au sens où si elle est de la forme `If(`i, f, g`)`, alors les formules f et g ne font intervenir que des variables *strictement plus grandes que* i. Ainsi,

```
If(0, If(1, False, True), If(2, True, False))
```

est une formule ordonnée, mais

```
If(0, If(1, False, True), If(0, True, False))
```

n'en est pas une.

☐ 34 – Écrire une fonction `check: int -> formula -> bool` qui prend en arguments un entier n et une formule f et qui détermine si d'une part f est bien une formule ordonnée et si d'autre part f est limitée aux variables X_i avec $0 \leq i < n$. La complexité doit être linéaire en la taille de la formule. On ne demande pas de justifier la complexité.

☐ 35 – Donner une formule ordonnée pour $n = 3$ variables qui est vraie si et seulement si les trois variables ont la même valeur de vérité.

Tautologies. Afin de décider si une formule est une tautologie, on se donne le type OCaml `result` suivant :

```
type assignment = bool array
type result = Tautology | Refutation of assignment
```

Le type `assignment` correspond à une valuation. Une valeur de type `assignment` est un tableau `a` de taille n, où `a.(`i`)` donne la valeur de la variable X_i.
☐ 36 – Écrire une fonction `decide: int -> formula -> result` qui prend en arguments un entier n et une formule f, supposée ordonnée et sur n variables, et qui renvoie

— `Tautology` si f est une tautologie ;

— `Refutation` v sinon, avec $\mathcal{V}_v(f) = F$.

On s'efforcera de proposer quelque chose de plus efficace que le test systématique de toutes les valuations possibles, en faisant intervenir le caractère ordonné de la formule.

Construire des formules arbitraires. Pour montrer que toute formule booléenne classique admet une formule ordonnée équivalente, il suffit de se donner des fonctions sur les formules ordonnées qui correspondent aux connecteurs logiques usuels, telles que la négation, la conjonction, la disjonction, etc.
☐ 37 – Écrire une fonction `mk_not: formula -> formula` qui reçoit en argument une formule ordonnée f et qui renvoie une formule ordonnée correspondant à la négation $\neg f$, c'est-à-dire une formule ordonnée g telle que, pour toute valuation v, on a $\mathcal{V}_v(g) = \neg\mathcal{V}_v(f)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille de f.

☐ 38 – Écrire une fonction `mk_or: formula -> formula -> formula` qui reçoit en arguments deux formules ordonnées f et g et qui renvoie une formule ordonnée correspondant à la disjonction $f \vee g$, c'est-à-dire une formule ordonnée h telle que, pour toute valuation v, on a $\mathcal{V}_v(h) = \mathcal{V}_v(f) \vee \mathcal{V}_v(g)$. Donner, sans la justifier, la complexité de votre code en fonction de la taille des formules f et g.

Poisson d'avril. Le poisson d'avril est une espèce extrêmement rare qui n'a jamais été permis d'observer dans la nature. Néanmoins, les ichtyologues ont permis de déterminer les faits suivants :

(F_1) tout poisson d'avril qui ne nage pas en mer chaude a des rayures rouges;

(F_2) tout poisson d'avril a des nageoires bleues ou n'a pas de rayures rouges;

(F_3) les poissons d'avril qui vivent dans le corail ne mangent pas de crevettes;

(F_4) un poisson d'avril mange des crevettes si et seulement s'il nage en mer chaude;

(F_5) tout poisson d'avril qui a des nageoires bleues nage en mer chaude et vit dans le corail;

(F_6) tout poisson d'avril qui ne nage pas en mer chaude a des nageoires bleues.

On souhaite mettre en application le code OCaml développé plus haut pour démontrer qu'il n'existe pas de poisson d'avril.
☐ 39 – Expliquer comment construire une formule ordonnée de type `formula` qui est une tautologie si et seulement si les poissons d'avril n'existent pas.

Fin de l'épreuve