Devoir surveillé n° 3 — sur machine — 2h45

L'utilisation d'Internet, le partage d'informations avec autrui et la consultation de tout document sont interdits pendant toute la durée de l'épreuve.

Ce devoir est à réaliser à l'aide d'une machine. Vous utiliserez le nom d'utilisateur `ds3-<login>` où `<login>` est votre nom d'utilisateur du laboratoire d'informatique (tout en minuscules avec les tirets). Vous écrirez vos programmes CAML exclusivement dans le fichier `ds3-<login>.ml` qui se trouve à la racine de votre répertoire, sans en changer le nom ni l'emplacement. Vous n'avez pas à recopier vos programmes sur votre copie. Vous pouvez commenter et expliquer vos fonctions directement sous la forme de commentaires en CAML.

> Les questions comportant le symbole ⊛ sont à rédiger au propre sur votre copie.

Votre programme devra respecter scrupuleusement les noms et les types indiqués (ou un type plus général ou équivalent bien entendu) car il sera en partie corrigé automatiquement. Le code fourni devra compiler sans erreurs ni aucun message d'avertissement. Vous n'utiliserez que des caractères ASCII pour les noms de variables. Il est impératif de veiller au strict respect de ces consignes.

La clarté et la simplicité du code et l'utilisation judicieuse de commentaires seront pris en compte dans l'évaluation. On prendra bien soin de tester toutes les fonctions sur des cas triviaux, puis un peu plus complexes.

Lorsque l'on pose des hypothèses sur les arguments (par exemple qu'un argument est un entier naturel), il n'est pas nécessaire dans vos fonctions de le vérifier ni de prévoir le comportement si l'hypothèse est violée (par exemple si l'argument est négatif). On pourra cependant lever des exceptions (`failwith "Argument non valable"` par exemple) si on le souhaite.

Les cinq parties sont indépendantes, à l'exception de la partie IV qui utilise la question 1. Cependant, les parties sont triées par ordre de rentabilité des points et il est donc fortement recommandé de les traiter dans l'ordre.

I Questions de cours

QUESTION 1 Écrire une fonction `tri_fusion : 'a list -> 'a list` qui implémente le tri fusion sur des listes.

QUESTION 2 Écrire une fonction `min_vect_index : 'a vect -> int` qui renvoie l'indice de la première position du minimum d'un tableau.

QUESTION 3 (*5-10 lignes*) ⊛ Qu'est-ce que la programmation dynamique ?

II Exercices

EXERCICE 1 *Nombre de Hamming*

Un nombre de Hamming est un entier $n \in \mathbb{N}^*$ qui ne comporte que[1] des 2, 3 ou 5 dans sa décomposition en facteurs premiers. Écrire une fonction `hamming : int -> bool` qui détermine si un entier donné est un entier de Hamming. Par exemple 421875 est un entier de Hamming mais 345243 ne l'est pas.

EXERCICE 2 *Préfixes d'une liste*

Écrire une fonction `prefixes : 'a list -> 'a list list` qui renvoie la liste de tous les préfixes non vides d'une liste donnée. Par exemple :

1. Éventuellement aucun.

```
prefixes [3; 2; 5; 1];;
- : int list list = [[3]; [3; 2]; [3; 2; 5]; [3; 2; 5; 1]]
```

EXERCICE 3 *Suppression des doublons*

On souhaite écrire une fonction `supprimer_doublons` qui supprime les éventuels doublons d'une liste.

II.1) Écrire une fonction `appartient : 'a -> 'a list -> bool` qui vérifie si un élément appartient à une liste.

II.2) ⊛ Quelle est sa complexité ?

II.3) Écrire une fonction `supprimer_doublons_gauche : 'a list -> 'a list` dans laquelle seule la dernière occurrence de chaque élément est conservée. Par exemple :

```
supprimer_doublons_gauche [1; 2; 3; 1; 4; 3; 1];;
- : int list = [2; 4; 3; 1]
```

II.4) ⊛ Quelle est sa complexité ?

II.5) Écrire une fonction `supprimer_doublons_droite : 'a list -> 'a list` dans laquelle seule la première occurrence de chaque élément est conservée. Par exemple :

```
supprimer_doublons_droite [1; 2; 3; 1; 4; 3; 1];;
- : int list = [1; 2; 3; 4]
```

II.6) ⊛ Quelle est sa complexité ?

III Multiplication de polynômes

Dans ce problème, on se propose d'étudier un algorithme de multiplication de deux polynômes plus efficace que la version naïve que nous avions déjà rencontrée en DM.

III.1 Préliminaire

III.1) Écrire une fonction `subvect : 'a vect -> int -> int -> 'a vect` telle que `subvect tableau debut longueur` renvoie un sous-tableau de `tableau` à partir de la position `debut` et de longueur `longueur`. On supposera que les arguments désignent bien un sous-tableau valable (éventuellement vide).

III.2 Opérations de base sur les polynômes

On s'intéresse dans cette section à des polynômes à coefficients entiers que l'on représentera à l'aide de tableaux[2]. Un polynôme $P = \sum_{k=0}^{n-1} a_k X^k \in \mathbb{Z}[X]$ est ainsi représenté par un vecteur `poly` de type `int vect` de taille n, l'entier `poly.(k)` représentant le coefficient a_k. Ainsi, le polynôme X^2+1 peut être représenté — par exemple — par le tableau `[|1; 0; 1; 0|]`. Il y a unicité de la représentation si a_{n-1} est non nul, c'est-à-dire si $n-1$ est le degré du polynôme. Dans ce cas, on dit que la représentation est *adaptée*. La représentation adaptée de X^2+1 est `[|1; 0; 1|]`.

III.2) ⊛ Quelle représentation adaptée choisir pour le polynôme nul ?

III.3) Écrire une fonction `degre : int vect -> int` qui calcule le degré d'un polynôme dont la représentation n'est pas nécessairement adaptée.

2. Contrairement au DM n° 1 dans lequel on considérait des polynômes à coefficients flottants représentés par des listes.

III.4) Écrire une fonction `reduire_repr : int vect -> int vect` qui transforme la représentation éventuellement non adaptée d'un polynôme en sa représentation adaptée.

III.5) Écrire une fonction `agrandir_repr : int vect -> int -> int vect` telle que `agrandir poly m` renvoie une représentation de `poly` de taille `m`. Cette fonction supposera que le degré du polynôme est inférieur ou égal à $m-1$.

Les complexités seront évaluées en fonction de la taille des représentations des polynômes, qui n'est pas nécessairement leur degré plus un si la représentation n'est pas adaptée.

III.3 Multiplication naïve

III.6) Écrire une fonction `somme : int vect -> int vect -> int vect` qui calcule la somme de deux polynômes quelconques. On ne suppose pas les représentations adaptées.

III.7) ⊛ Quel est le nombre d'additions d'entiers nécessaires, si les deux polynômes ont une représentation de même taille n, en fonction de cette taille ?

III.8) Écrire une fonction `multiplication_naive : int vect -> int vect -> int vect` qui calcule le produit de deux polynômes quelconques. On ne suppose pas les représentation adaptées.

III.9) ⊛ Quel est le nombre nécessaire d'additions d'entiers d'une part et de multiplications d'autre part, si les deux polynômes ont une représentation de même taille n, en fonction de cette taille ?

III.4 Méthode de Karatsuba

Par la suite on va chercher à réduire le nombre de multiplications entières mises en jeu pour multiplier deux polynômes.

Soient P et Q deux polynômes dont la représentation a une taille $n = 2^k$ avec $k \in \mathbb{N}$. Si $n \geq 2$, on pose $m = \frac{n}{2}$ et on écrit $P = X^m P_1 + P_2$ et $Q = X^m Q_1 + Q_2$, la représentation de chacun des quatre polynômes P_1, P_2, Q_1 et Q_2 étant de taille m.

III.10) ⊛ Que vaut PQ en fonction de P_1, P_2, Q_1 et Q_2 ?

III.11) ⊛ Comment s'appelle le paradigme (la technique) que nous sommes en train de mettre en place ? Justifier rapidement.

III.12) ⊛ Montrer que le nombre de multiplications $M(n)$ vérifie la relation de récurrence $M(n) = 4M(n/2) + \Theta(n)$.

III.13) ⊛ Conclure.

III.14) ⊛ On pose $R_1 = P_1Q_1$, $R_2 = (P_1 + P_2)(Q_1 + Q_2)$ et $R_3 = P_2Q_2$. Exprimer $P_1Q_2 + P_2Q_1$ en fonction de R_1, R_2 et R_3.

III.15) ⊛ En déduire une expression de PQ qui ne nécessite que 3 appels récursifs au lieu de 4.

III.16) ⊛ Quelle est alors la relation de récurrence vérifiée par $M(n)$?

III.17) ⊛ Montrer que $M(n) = \Theta(n^c)$ ou c est une constante que vous expliciterez. Vérifier que $c \approx 1.58$.

III.18) Écrire une fonction `karatsuba : int vect -> int vect -> int vect` qui implémente le produit de deux polynômes dont les deux représentations sont de taille $n = 2^k$ avec $k \in \mathbb{N}$ en utilisant la méthode décrite ci-dessus.

III.19) ⊛ Quelle est sa complexité spatiale ?

III.20) Écrire une fonction `multiplication_karastuba : int vect -> int vect -> int vect` qui généralise la fonction précédente à des polynômes quelconques dont la représentation n'est pas nécessairement adaptée. La représentation du polynôme renvoyé comme résultat devra être adaptée.
Indication : il suffit de rajouter ou de supprimer des zéros.

La similitude entre la représentation binaire d'un entier et les polynômes permet d'adapter l'algorithme précédent aux grands entiers (et inversement), la seule différence se trouve dans la gestion de la retenue. On peut utiliser la même idée pour la multiplication de deux matrices (algorithme de Strassen).

Dans les années 1950, Andreï Kolmogorov travaille sur la complexité des opérations arithmétiques et conjecture qu'une multiplication de deux nombres de n chiffres ne peut être réalisée en moins de $O(n^2)$ opérations. À l'époque, aucun algorithme plus rapide que la multiplication standard n'est connu. À l'automne 1960, Kolmogorov organise un séminaire dans lequel il parle de sa conjecture, auquel assiste Anatolii Alexevich Karatsuba. Une semaine plus tard, Karatsuba avait trouvé cet algorithme, qui prouvait que la conjecture était fausse.

L'algorithme Toom-Cook est un raffinement qui consiste à découper les polynômes en r blocs avec $r > 2$. La complexité peut alors passer en $O(n^{1+\epsilon})$ où ϵ est un réel strictement positif arbitraire. L'algorithme de Schönhage-Strassen, qui utilise la transformation de Fourier rapide, permet d'obtenir une complexité de $O(n \log n)$.

IV Nombre d'inversions dans une liste

On appelle *inversion* d'une suite finie d'éléments distincts $(x_1, \dots, x_n)$ tout couple (i, j) tel que $i < j$ mais $x_i > x_j$.

IV.1) ⊛ Quelles sont les inversions de $(2, 3, 1, 5, 4)$?

IV.2) Écrire en CAML une fonction `nb_inversions : 'a list -> int` permettant de calculer le nombre d'inversions d'une liste d'éléments distincts, en testant un à un tous les couples (i, j) avec $i < j$.

IV.3) ⊛ Quelle est sa complexité ?

On se propose d'adopter une stratégie *diviser pour régner* pour résoudre ce problème de manière plus efficace. La stratégie diviser pour régner consiste à diviser la liste en deux. Les inversions sont alors :

- les inversions de chacune des deux sous-listes ;
- plus les inversions qui sont à cheval entre les deux.

IV.4) ⊛ Comment obtient-on les inversions de chacune des deux sous-listes ?

IV.5) ⊛ Quelle doit être la complexité de la recherche des inversions à cheval pour que cette stratégie soit plus efficace que la méthode naïve ?

IV.6) ⊛ Montrer que si les deux sous-listes sont triées, il est possible d'atteindre cette complexité. *Indication : on pourra utiliser une variable auxiliaire contenant la taille d'une des deux sous-listes bien choisie.*

IV.7) Écrire une fonction `inversions_a_cheval : 'a list -> 'a list -> int` qui calcule le nombre d'inversions à cheval entre deux sous-listes supposées triées.

IV.8) Écrire une fonction `nb_inversions_diviser_pour_regner : 'a list -> int` qui calcule le nombre d'inversions en utilisant cette stratégie diviser pour régner. *Indication : adapter le tri fusion. On pourra réutiliser des fonctions écrites précédemment, par exemple dans la question 1 du sujet.*

IV.9) ⊛ Quelle est sa complexité ? Est-ce mieux que l'algorithme naïf ?

V Bonus

Cette partie est très fortement sous-barémée et ne doit être abordée que si vous pensez avoir traité tout ce qui précède avec succès. Ces problèmes sont tirés du fabuleux *Project Euler*[3]. Vous donnerez la réponse directement sur votre copie, en expliquant votre démarche, et vous êtes libres d'utiliser PYTHON ou CAML pour résoudre ces problèmes.

EXERCICE 4 *Projet Euler 004*

⊛ Un entier naturel (écrit en base 10) est un palindrome s'il se lit à l'identique de gauche à droite et de droite à gauche. L'entier $9009 = 91 \times 99$ est le plus grand palindrome à s'écrire comme le produit de deux entiers à deux chiffres. Quel est le plus grand palindrome égal au produit de deux entiers à trois chiffres ?

EXERCICE 5 *Projet Euler 034*

⊛ Le nombre 145 est la somme des factorielles de ses chiffres : $145 = 1! + 4! + 5! = 1 + 24 + 120$. Trouver tous les entiers $n \geq 1$ qui ont cette propriété.

3. https://projecteuler.net