Devoir surveillé n°1

MPSI Lycée Clemenceau : option informatique

Lundi 25 mars 2024

Vous avez 3 heures.

Il sera tenu compte de la présentation et de la rédaction. Les programmes (ou fonctions) devront être commentés et le plus souvent justifiés (terminaison et correction).
Lire l'annexe pour les fonctions prédéfinies et autorisées pour chaque exercice.

Exercice 1 :

Soit $n \in \mathbb{N}^*$, on calcule le produit $prod(n)$ de ses chiffres en base 10, puis le produit des chiffres de $prod(n)$ dans son écriture en base 10 et on recommence l'application de $prod$ jusqu'à obtenir un chiffre entre 0 et 9. Le nombre minimal de fois où on applique $prod$ pour transformer n en un chiffre compris entre 0 et 9 est appelé persistance de n.

1) Donner, en la justifiant, la persistance de 9, puis de 97, puis de 9575.

2) Ecrire une fonction récursive `prod` qui renvoie le produit des chiffres d'un entier écrit en base 10.

3) Ecrire une fonction récursive `persi` qui renvoie la persistance d'un entier naturel.

4) Ecrire une fonction `persi5` qui renvoie le plus petit entier naturel de persistance 5. On proposera une version itérative et une version récursive.

Exercice 2 : approximation de réels

A) Racine carrée entière

On s'intéresse au problème suivant : étant donné un entier $n \geq 1$, on souhaite déterminer un entier positif s_n de sorte que s_n soit "proche" de $\sqrt{n}$. Nous étudierons deux approches pour résoudre ce problème (*autres que celui vu, mais pas par tous, en td*).

1) Algorithme naïf

Dans cette question on cherche à déterminer le plus grand entier s_n tel que $s_n^2 \leq n$. Pour cela on passe en revue les entiers $1, 2, 3, ..$ jusqu'à trouver s_n.

(a) Ecrire une fonction isqrt_naif prenant n en argument et renvoyant la valeur de s_n en utilisant le principe décrit ci-dessus.

(b) Déterminer la complexité de votre programme.

2) Une méthode dichotomique

Dans cette question, on reprend le problème de la question précédente. Afin d'obtenir une méthode plus efficace, on considère le programme suivant :

```
# let isqrt n = let x =ref 1 and y = ref n in
        while (!y - !x)>1 do
            let z = (!x + !y)/2 in
            if z*z >n then y := z else x := z
        done;
        x;;
```

Si $k \geq 1$ on note x_k et y_k les valeurs des variables x et y à la fin de la $k-$ième itération de la boucle `while`. On convient que $x_0 = 1$ et $y_0 = n$.

(a) Expliciter les exécutions du programme ci-dessus sur l'entier 15. (On donnera en particulier les valeurs successives des variables x, y et z).

(b) Démontrer la terminaison du programme.

(c) Justifier la correction du programme.

(d) Réécrire `isqrt` en utilisant la récursivité.

B) Moyenne arithméticogéométrique

3) Soit a et b deux réels tels que $0 < a < b$. Soient (u_n) et (v_n) les suites définies par

$$u_0 = a,\ v_0 = b, \quad \forall n \in \mathbb{N},\ u_{n+1} = \sqrt{u_n v_n},\ v_{n+1} = \frac{v_n + u_n}{2}$$

On admet que ces suites sont adjacentes et on note ℓ leur limite commune appelée moyenne arithméticogéométrique de a et b.

Proposer une fonction `calcul_de_l : float * float * float -> float` qui reçoit en entrée un triplet de réels strictement positifs a, b, p et qui renvoie une valeur approchée de ℓ à p prés.

4) Soit x un réel. On définit trois suites $(s_n)_{n\in\mathbb{N}}$, $(t_n)_{n\in\mathbb{N}}$ et $(c_n)_{n\in\mathbb{N}}$ par :

$$s_0 = \frac{1}{2}\left(x - \frac{1}{x}\right),\ c_0 = \frac{1}{2}\left(x + \frac{1}{x}\right), \quad \forall n \in \mathbb{N}\ c_{n+1} = \sqrt{\frac{1+c_n}{2}},\ s_{n+1} = \frac{s_n}{c_{n+1}},\ t_n = \frac{s_n}{c_n}$$

On admet que, pour tout $n \in \mathbb{N}$, $t_n \leqslant \ln(x) \leqslant s_n$ et que $(s_n)_{n\in\mathbb{N}}$ et $(t_n)_{n\in\mathbb{N}}$ convergent vers $\ln(x)$.

Proposer un algorithme `calcul_de_ln : float -> float -> float` qui reçoit en entrée deux réels x et $p > 0$ et qui renvoie une valeur approchée de $\ln(x)$ à p près.

Exercice 3

Le but de l'exercice est d'engendrer l'ensemble $\mathscr{P}(E)$ des parties d'un ensemble fini E donné.

Les algorithmes étudiés peuvent s'appliquer à tout type d'ensemble, mais pour leur programmation explicite, on se limitera au cas d'ensembles d'entiers, dont les éléments seront stockés dans un tableau (ou vecteur)

Pour caractériser un sous-ensemble A d'un tel ensemble E, on convient d'utiliser un *tableau de présence* noté `p` de la façon suivante : pour tout `k` entre 1 et n, `E.(k)` est élément de A si et seulement si `p.(k)=1`.

Exemple, si $n = 4$ et si le tableau `E` est `[|3;5;7;9;|]`, alors $E = \{3, 5, 7, 9\}$ et :
— `p=[|0;1;0;0|]` correspond à la partie $\{5\}$
— `p=[|1;0;0;1|]` correspond à la partie $\{3, 9\}$, etc.

Pour les fonctions et procédures demandées, le candidat pourra choisir entre version itérative ou récursive lorsque l'énoncé ne précise rien.

Si une version récursive est requise, la fonction ou procédure écrite pourra être elle-même récursive, ou bien faire appel à une fonction ou procédure auxiliaire récursive.

1) *Affichage d'une partie* : écrire deux versions (itérative **et** récursive) d'une procédure

```
# affpartie E p
```

qui affiche à l'écran la partie de l'ensemble E (de cardinal n, stocké dans le tableau `E`) caractérisée par le tableau de présence `p`. Les valeurs seront affichées entre accolades, séparées par des espaces.

Par exemple, des paramètres correspondant à la partie $\{3, 9\}$ devront conduire à l'affichage de

```
{ 3 9 }
```

2) *Génération de $\mathcal{P}(E)$ – version itérative* : pour cette première méthode, on utilise la propriété suivante, **que l'on ne demande pas de démontrer**. Lorsqu'un entier j varie de 0 à $2^n - 1$, son écriture en base 2 fournit tous les tableaux de présence correspondant aux 2^n parties d'un ensemble à n éléments (sous réserve bien sûr de considérer les écritures comportant exactement n chiffres, avec éventuellement des 0 pour les chiffres de poids le plus élevé). Par exemple, pour $n = 3$, les écritures en base 2 des entiers de 0 à 7 sont : 000, 001, 010, 011, 100, 101, 110, 111.

a) Écrire une fonction

```
# deuxpuis n
```

renvoyant 2^n.

b) On convient ici que les valeurs (0 ou 1) stockées dans un tableau `p` de présence correspondent aux chiffres de l'écriture en base 2 d'un entier j, **rangés par ordre de poids croissant avec l'indice**.

Par exemple, `p=[|1;1;1;0;0;1;0|]` correspond à $j = 39 = 2^0 + 2^1 + 2^2 + 2^5$ et `p=[|0;0;0;1;0;1|]` correspond à $j = 40 = 2^3 + 2^5$.

Écrire deux versions (itérative **et** récursive) d'une procédure d'en-tête

```
# ajoute1 p
```

recevant un tableau `p` contenant — selon le principe ci-dessus — les chiffres de l'écriture en base 2 d'un entier j et modifiant `p` pour qu'il représente de même l'entier $j+1$.

On justifiera la méthode utilisée pour effectuer l'addition.

c) Déduire des questions précédentes une procédure itérative

```
# AffEnsPartiesI E
```

qui affiche à l'écran successivement toutes les parties de l'ensemble E (stocké dans le tableau `E`).

3) *Génération de $\mathcal{P}(E)$ – version récursive* : on peut engendrer récursivement tous les tableaux de présence correspondant aux différentes parties de l'ensemble donné E (et afficher la partie A associée au tableau `p` dès qu'il est rempli !). Écrire selon ce principe une fonction récursive ou utilisant une fonction auxiliaire récursive

```
# affenspartiesr E
```

qui affiche à l'écran successivement toutes les parties de l'ensemble E .

Cette question est indépendante des précédentes.

Exercice 4 : manipulation de listes

1) Ecrire une fonction calculant la somme des éléments (entiers) d'une liste.

2) Ecrire une fonction `sublist : 'a list -> int -> int -> 'a list` renvoyant la liste formée des éléments dont la position est comprise, au sens large, entre les valeurs spécifiées.

3) Ecrire une fonction `minmaxlist : 'a list -> 'a * 'a` qui renvoie le couple formé par le plus petit et plus grand élément d'une liste.

4) Ecrire une fonction `min2list : 'a list -> 'a * 'a` qui retourne le couple des deux plus petits éléments.

5) Ecrire une fonction `listplateau : 'a list -> int * int * 'a` identifiant le plus long plateau formé d'éléments consécutifs identiques dans une liste quelconque (donc pas forcément triée !). Le résultat est le triple `(d; l; x)` formé par la position `d` du début du plateau, par sa longueur `l` et par la valeur `x` de l'élément ainsi répété.
Par exemple :

```
listplateau ["e"; "x"; "x"; "x"; "c"; "c"; "x"; "c"; "c"; "c"; "c"; "u"; "c"];;
- : int * int * string = 7, 4, "c"
```

Ecrire une fonction `ocsmin : 'a list -> 'a * int = <fun>` qui renvoie le couple formé par l'élément minimum d'une liste et le nombre de fois où il apparaît. Cette fonction ne devra parcourir qu'une seule fois la liste.

Exercice 5 : Représentation d'ensembles avec des intervalles (CCP 2015)

De nombreux algorithmes reposent sur la manipulation d'ensembles d'éléments ordonnés. Lorsque ces ensembles contiennent de nombreux éléments adjacents (aucune valeur n'existe entre les deux éléments), il est plus performant en terme de temps de calcul et d'occupation mémoire de manipuler des intervalles au lieu de valeurs singulières. Cela permet également de manipuler des ensembles infinis sous la forme d'un ensemble fini d'intervalles contenant un nombre de valeurs infinies (i.e. dans $\mathbb{Q}$ ou $\mathbb{R}$).

L'objectif de ce problème est de comparer deux implantations différentes d'un ensemble d'entiers, la première à base de listes triées d'intervalles, et la seconde à base d'arbres binaires d'intervalles.

Nous nous limiterons à des intervalles fermés de $\mathbb{R}$ dont les bornes sont des entiers $[min, max]$. L'extension à des intervalles ouverts et aux valeurs $-\infty$ et $+\infty$ ne pose pas de problème majeur.

I Préliminaires : Représentation des intervalles

I.1 Définitions

Définition 1) (Intervalles disjoints) Deux intervalles sont disjoints si leur intersection est vide.

Définition 2) (Fusion d'intervalles) La fusion de deux intervalles a comme minimum le plus petit des minima des deux intervalles, et comme maximum le plus grand des maxima des deux intervalles. Cette opération correspond à l'union des intervalles si ceux-ci ne sont pas disjoints.

I.2 Représentation en CaML

Nous ferons l'hypothèse que les couples qui représentent des intervalles sont bien formés, c'est-à-dire que la valeur représentant le minimum est inférieure ou égale a la valeur représentant le maximum.

Un intervalle est représenté par le type `intervalle` équivalent à une paire de `int` (son minimum en premier et son maximum en second).

```
type intervalle = int * int;;
```

Nous allons maintenant définir plusieurs opérations sur les intervalles.

2.1.3 Opérations sur la structure d'intervalle

La première opération est le test si deux intervalles sont disjoints.

Question 1) Écrire en CaML une fonction `disjoints` de type `intervalle -> intervalle -> bool` telle que l'appel (`disjoints i1 i2`) renvoie la valeur `true` si les intervalles `i1` et `i2` sont disjoints, et la valeur `false` sinon.

La seconde opération est la fusion de deux intervalles.

Question 2) Écrire en CaML une fonction `fusion` de type `intervalle -> intervalle -> intervalle` telle que l'appel (`fusion i1 i2`) renvoie un intervalle correspondant à la fusion de `i1` et `i2`.

2.2 Réalisation à base d'une liste triée d'intervalles

La réalisation la plus simple d'un ensemble de valeurs en utilisant des intervalles consiste à utiliser une liste d'intervalles. Pour réduire la complexité amortie de certaines opérations, nous utiliserons une liste triée d'intervalles d'entiers.

2.2.1 Définitions

Les algorithmes manipuleront des listes d'intervalles respectant certaines contraintes que nous appellerons liste bien formée.

Définition 3) (Liste bien formée d'intervalles) Une liste bien formée d'intervalles est une liste d'intervalles qui respecte les contraintes suivantes :

- les intervalles sont bien formés,
- les intervalles sont disjoints deux à deux,
- la liste est triée selon la relation d'ordre suivante : un intervalle i_1 est strictement plus petit qu'un intervalle i_2 si et seulement si le maximum de i_1 est strictement plus petit que le minimum de i_2.

On peut montrer qu'une liste triée d'intervalles bien formés, est une liste bien formée d'intervalles.

La définition 3) peut s'exprimer sous la forme d'une propriété $LBFI(e)$ qui indique que $e = \langle v_1, ..., v_n \rangle$ est une liste bien formée d'intervalles (contenant les intervalles v_i) :

$$LBFI(e) \iff \begin{cases} e = \langle\rangle \\ \text{ou } e = \langle u_1 \rangle \\ \text{ou } e = \langle v_1, v_2, ..., v_n \rangle \text{ et } v_1 < v_2 \text{ et } LBFI(e') \text{ avec } e' = \langle v_2, ..., v_n \rangle \end{cases}$$

2.2.2 Représentation en CaML

Une liste triée d'intervalles est représentée par le type `liste` équivalent à une liste d'`intervalle`.

```
type liste = intervalle list;;
```

Nous allons maintenant définir plusieurs opérations sur les listes bien formées d'intervalles et estimer leur complexité.

2.2.3 Opérations sur la structure de liste bien formée d'intervalles

La première opération est l'ajout d'un intervalle dans une liste bien formée d'intervalles.

Question 3) Écrire en CaML une fonction `ajouter` de type `intervalle -> liste -> liste` telle que l'appel (`ajouter i l`) sur une liste bien formée d'intervalles `l` renvoie une liste bien formée d'intervalles contenant les intervalles contenus dans la liste `l` qui sont disjoints de `i`, et :

— soit le résultat de la fusion de `i` et des intervalles contenus dans la liste `l` qui ne sont pas disjoints de `i`,
— soit l'intervalle `i`, si tous les intervalles contenus dans la liste `l` lui sont disjoints.

L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Question 4) Donner des exemples de valeurs des paramètres `i` et `l` de la fonction `ajouter` qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.

Calculer une estimation de la complexité dans les meilleur et pire cas de la fonction `ajouter` en fonction de la longueur de la liste `l`. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.

La deuxième opération est le test d'appartenance d'une valeur dans une liste bien formée d'intervalles.

Question 5) Écrire en CaML une fonction `appartenir` de type `int -> liste -> bool` telle que l'appel (`appartenir v l`) sur une liste bien formée d'intervalles `l` renvoie la valeur `true` si la valeur `v` appartient à un des intervalles contenus dans la liste `l` et la valeur `false` sinon.

L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

La troisième opération est la vérification qu'une liste d'intervalles est bien formée, c'est-à-dire respecte les contraintes de la définition 3) formalisée par la propriété $LBFI(\texttt{e})$ définie précédemment.

Question 6) Écrire en CaML une fonction `verifier` de type `liste -> bool` telle que l'appel (`verifier l`) sur une liste d'intervalles `l` renvoie la valeur `true` si la liste `l` respecte les contraintes d'une liste bien formée d'intervalles, et la valeur `false` sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Annexe

Exercice 1 :

Fonctions utiles : / pour le quotient de la division euclidienne de deux entiers, `mod` pour le reste de la division euclidienne

Exercice 2 :

Rien de particulier

Exercice 3 :

Fonctions utiles :
— / pour le quotient de la division euclidienne de deux entiers, `mod` pour le reste de la division euclidienne
— `print_string` : affiche une chaine de caractère
— `print_sint` : affiche un entier
— `incr` : fonction qui incrémente d'un une référence entière

Pour les tableaux :
— `Array.make n a` : créer un tableau de taille n dont tous les éléments sont égaux à `a`
— `Array.length` : donne la longueur d'un tableau
— `p.(i)` : donne l'élément du tableau d'indice `i` (l'indice commence à 0)
— `p.(i)<- a` : pour mettre la valeur de `a` dans le tableau à l'indice `i`

Exercice 4 :

Aucune fonction sur les listes prédéfinie n'est autorisée à par, si besoin, `List.hd`, `List.tl`, le constructeur `::`.

Les fonctions de deux variables `min` et `max` sont autorisées. Elles sont curryfiées.

Exercice 5 :

Pour avoir le premier et le second élément d'un couple on utilise `fst` et `snd`.

Aucune fonction sur les listes prédéfinie n'est autorisée à par, si besoin, `List.hd`, `List.tl`, le constructeur `::`.

Les fonctions de deux variables `min` et `max` sont autorisées. Elles sont curryfiées.