Devoir surveillé n° 1 — 3 h

La calculatrice n'est pas autorisée. En revanche, l'aide-mémoire CAML l'est, et on pourra utiliser toute fonction ou opérateur s'y trouvant.

Les programmes seront rédigés clairement et proprement, en couleur, et en mettant en valeur l'indentation. On veillera à choisir des noms de variables formant des mots complets et explicites. Il est indispensable de commencer par un brouillon avant de recopier la version finale au propre. *Dans toute question qui demande d'implémenter une fonction, vous veillerez à introduire votre programme par quelques lignes explicatives et à préciser le type de vos fonctions.*

Attention : ceci n'est pas rappelé à chaque question, mais tous ces éléments sont pris en compte dans le barème de notation.

I Trois exercices

Exercice 1 : Prévoir le résultat

I.1) Prévoir la réponse de CAML pour chacune des phrases dans la séquence suivante :

```
let a = function b -> (b 4) / 5;;
let b a = function c -> a (function d -> c + d) + c;;
let c = b a;;
let d = c 11;;
let e = a b;;
```

I.2) Vous engagez-vous[1] à toujours utiliser des noms de variables explicites ?

Exercice 2 : Tri insertion

I.3) Écrire une fonction `insere : 'a -> 'a list -> 'a list` prenant en argument un objet et une liste triée par ordre croissant et renvoyant la liste obtenue en insérant l'objet dans la liste en conservant son caractère trié. Par exemple, `insere 5 [2; 4; 7; 8; 9]` renvoie `[2; 4; 5; 7; 8; 9]`.

I.4) Écrire une fonction `tri_insertion : 'a list -> 'a list` triant la liste : lorsque c'est encore possible, il suffit d'insérer la tête dans la queue qui aura été triée récursivement.

I.5) Quelle est la complexité temporelle dans le meilleur et le pire des cas ? Préciser la forme de listes pour lesquelles ces cas sont atteints.

Exercice 3 : Écriture en base b

Rappel 1. *Soit $n \in \mathbb{N}$ et $b \in \mathbb{N}^* \setminus \{1\}$. Il existe un unique polynôme $P = \sum_{k=0}^{d} a_k X^k$ à coefficients dans $[\![0; b-1]\!]$ tel que $n = P(b)$. La suite des coefficients $(a_k)_{k \in [\![0;d]\!]}$ est appelée* décomposition *de n dans la base b.*

Dans cet exercice, on prend la liste des coefficients dans l'ordre *décroissant des indices*, ce que l'on appelle *représentation* d'un entier en base b. Par exemple, la représentation de 2017 en base 10 est $[2; 0; 1; 7]$.

I.6) Donner la représentation de 42 en base 3.

I.7) Quel est l'entier ayant pour représentation $[2; 1; 11]$ en base 16 ?

I.8) Écrire une fonction `ecriture_base` qui calcule la représentation d'un entier naturel n en base b. La complexité devra être linéaire en la taille de la représentation, ce qui sera justifié. On suppose que $b \geq 2$, il est inutile de le vérifier. Par exemple :

```
ecriture_base 87 2;;
- : int list = [1; 0; 1; 0; 1; 1; 1]
```

On pourra utiliser la fonction `rev` ou chercher une fonction auxiliaire récursive terminale.

I.9) Écrire la réciproque `lecture_base`. La complexité devra encore être linéaire et justifiée. On supposera que la liste en premier argument est bien une représentation valable en base b, il n'est donc pas utile de le vérifier. Par exemple :

```
lecture_base [1; 2; 0; 1] 3;;
- : int = 46
```

1. Répondre par oui ou par non, sachant que seule la réponse positive rapporte des points, mais qu'elle engage alors son auteur pour toute l'année.

II Représentation d'ensembles avec des intervalles

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 peut être intéressant de manipuler des intervalles au lieu de valeurs singulières. Cela permet également de manipuler des ensembles infinis (e.g. $\mathbb{Q}$ ou $\mathbb{R}$) sous la forme d'un ensemble fini d'intervalles.

L'objectif de ce problème est d'implémenter des ensemble d'entiers à base de listes triées d'intervalles. Nous nous limiterons à des intervalles fermés de $\mathbb{R}$ dont les bornes sont des entiers. Un intervalle est représenté en CAML par une paire d'entiers (son minimum en premier et son maximum en second), c'est-à-dire un objet de type `int * int`. Sauf mention du contraire, 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 à la valeur représentant le maximum. *Il n'est donc pas utile de vérifier cette propriété dans vos programmes, qui peuvent donc le supposer*. Deux intervalles sont *disjoints* si leur intersection est vide.

Rappel 2. *Les trois programmes suivants sont identiques.*

```
let premier (a, b) =
  a
;;
```

```
let premier paire =
  let a, b = paire in
  a
;;
```

```
let premier paire =
  match paire with
  | (a, b) -> a
;;
```

II.1) Quel est le type des trois programmes précédents ?

II.2) Écrire en CAML une fonction `disjoints` de type `(int * int) -> (int * int) -> bool` telle que l'appel `disjoints inter1 inter2` renvoie la valeur `true` si les intervalles `inter1` et `inter2` sont disjoints, et la valeur `false` sinon. Une réponse n'utilisant pas d'expression conditionnelle (`if ... then ... else`) ni de garde `when`, mais uniquement les opérations booléennes rapporte un point de plus.

Définition 1 (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.*

II.3) À quelle condition cette opération correspond-elle à l'union des deux intervalles ?

II.4) Écrire en CaML une fonction `fusion` de type `(int * int) -> (int * int) -> (int * int)` telle que l'appel `fusion inter1 inter2` renvoie un intervalle correspondant à la fusion de `inter1` et `inter2`.

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 de certaines opérations, nous utiliserons une liste triée d'intervalles d'entiers. Les algorithmes manipuleront des listes d'intervalles respectant certaines contraintes que nous appellerons *liste bien formée*.

Définition 2. *Une liste bien formée d'intervalles est une liste d'intervalles qui respecte les contraintes suivantes :*

(a) les intervalles sont bien formés ;

(b) les intervalles sont disjoints deux à deux ;

(c) 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.

II.5) Justifier que l'on peut remplacer, dans (c), la relation d'ordre proposée par l'ordre lexicographique sur les couples représentant les intervalles. C'est l'ordre naturel en CAML : si `inter1` et `inter2` sont deux intervalles de type `(int * int)`, on peut donc écrire `inter1 < inter2`.

II.6) Montrer qu'une liste strictement croissante d'intervalles bien formés est une liste bien formée d'intervalles.

Une liste (triée) d'intervalles est représentée en CAML par le type `(int * int) list`.

II.7) Écrire en CAML une fonction `appartenir` de type `int -> (int * int) list -> bool` telle que l'appel `appartenir valeur liste` sur une liste bien formée d'intervalles `liste` renvoie la valeur `true` si la valeur `valeur` appartient à l'un des intervalles contenus dans la liste `liste` et la valeur `false` sinon. L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. Quelle est sa complexité ?

II.8) Écrire en CAML une fonction `ajouter` de type `(int * int) -> (int * int) list -> (int * int) list` telle que l'appel `ajouter intervalle liste` sur une liste bien formée d'intervalles `liste` renvoie une liste bien formée d'intervalles contenant les intervalles contenus dans la liste qui sont disjoints de l'intervalle `intervalle`, ainsi que :

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

L'algorithme utilisé ne devra parcourir qu'une seule fois la liste.

II.9) Calculer une estimation de la complexité dans les meilleur et pire cas de la fonction `ajouter` en fonction de la longueur de la liste `liste`. On donnera, en justifiant, des exemples de valeurs des paramètres `intervalle` et `liste` de la fonction `ajouter` qui correspondent aux meilleur et pire cas en nombre d'appels récursifs effectués.

II.10) Écrire en CaML une fonction `verifier` de type `(int * int) list -> bool` telle que l'appel `verifier liste` sur une liste de paires d'entiers `liste` renvoie la valeur `true` si la liste `liste` respecte les contraintes d'une liste bien formée d'intervalles, et la valeur `false` sinon. Quelle est sa complexité ?

L'algorithme utilisé ne devra parcourir qu'une seule fois la liste. *Attention*, dans cette question, et seulement dans celle-là, on ne suppose pas *a priori* que les intervalles sont bien formés.

III Autour du tri rapide

Une séquence s de taille n de valeurs v_i avec $\{1 \leq i \leq n\}$ est notée $(v_n, \ldots, v_1)$. Sa taille n est notée $|s|$. Une même valeur v peut figurer plusieurs fois dans s, nous noterons $|s|_v = \text{card}(\{i \in \mathbb{N} / \ v_i = v\})$ le cardinal de v dans s, c'est-à-dire le nombre de fois que v figure dans s. Le symbole de Kronecker, noté δ, est tel que δ_{v_1,v_2} vaut 1 si $v_1 = v_2$ et 0 sinon.

On considère les deux fonctions suivantes en CAML :

```
let rec partition liste pivot =
  match liste with
  | [] -> ([], pivot, [])
  | tete :: queue ->
     let (petits, milieu, grands) = (partition queue pivot) in
     if (tete <= milieu) then
       (tete :: petits, milieu, grands)
     else
       (petits, milieu, tete :: grands)
;;
```

```
let rec tri liste =
  match liste with
  | [] -> []
  | tete :: queue ->
     let (petits, milieu, grands) = (partition queue tete) in
     tri petits @ milieu :: tri grands
;;
```

III.1) Quel est le type de ces deux fonctions ?

III.2) Soit la constante définie par `let exemple = [3; 1; 4; 2];;`. Détailler soigneusement les étapes du calcul de `tri exemple;;` en précisant, pour chaque appel aux fonctions `partition` et `tri`, la valeur des paramètres et du résultat.

III.3) Expliquer, de manière informelle, ce que font les deux fonctions, en détaillant bien le rôle de chacune des variables. Des illustrations seront appréciées.

III.4) Expliquer en quoi ce tri, dit « tri rapide », illustre le paradigme *diviser pour régner*, en indiquant bien à quoi correspondent les trois étapes.

Soit les entiers e et v et les séquences d'entiers $p = (p_n, \ldots, p_1)$ de taille n, $g = (g_r, \ldots, g_1)$ de taille r et $d = (d_s, \ldots, d_1)$ de taille s, telles que `(g, v, d) = (partition p e)`. On veut montrer que

(a) $e = v$
(b) $\forall i \in [\![1, n]\!],\ |p|_{p_i} = |g|_{p_i} + |d|_{p_i}$
(c) $0 \leq s \leq n$ et $0 \leq r \leq n$
(d) $n = s + r$
(e) $\forall i \in [\![1, r]\!],\ g_i \leq v$
(f) $\forall i \in [\![1, s]\!],\ v < d_i$

On va montrer cette propriété par récurrence sur $n \in \mathbb{N}$, taille de la séquence argument de la fonction `partition`. On note donc H_n la propriété ci-dessus, qui dépend de $n \in \mathbb{N}$.

III.5) Montrer très soigneusement que H_0 est vraie.

Soit $n \in \mathbb{N}$, on suppose que H_n est vraie au rang n et on veut montrer que c'est encore le cas au rang $n+1$.
Soit les entiers e et v', et les séquences d'entiers $p' = (p'_{n+1}, p'_n, \dots, p'_1)$ de taille $n+1$, $g' = (g'_{r'}, \dots, g'_1)$ de taille r' et $d' = (d'_{s'}, \dots, d'_1)$ de taille s', telles que `(g', v', d') = (partition p' e)`. On note alors $p = (p_n, \dots, p_1) = (p'_n, \dots, p'_1)$ la séquence d'entier de taille n, $g = (g_r, \dots, g_1)$ de taille r et $d = (d_s, \dots, d_1)$ de taille s, telles que `(g, v, d) = (partition p e)`.

III.6) Écrire ce que donne l'hypothèse de récurrence.

III.7) On suppose que $p'_{n+1} \leq v$. Montrer très soigneusement qu'alors H_{n+1} est vraie.

III.8) Conclure. On remarquera simplement que le cas $p'_{n+1} > v$ se traite de manière analogue, sans le détailler.

Soit la séquence $p = (p_m, \dots, p_1)$ de taille m, soit la séquence $r = (r_n, \dots, r_1)$ telle que `r = tri p`. On veut montrer de même que

(a) $m = n$
(b) $\forall i \in [\![1, m]\!], |r|_{p_i} = |p|_{p_i}$
(c) $\forall i \in [\![1, n-1]\!],\ r_i \leq r_{i+1}$.

III.9) Montrer très soigneusement ce résultat.

III.10) Montrer que le calcul des fonctions `partition` et `tri` se termine quelles que soient les valeurs de leurs paramètres respectant le type des fonctions.

Pour les calculs de complexité, on comptera comme opération élémentaire uniquement les comparaisons d'éléments. Toutes les autres opérations auront un coût nul. On note $P(n)$, respectivement $T(n)$, la complexité de la fonction `partition`, respectivement `tri`, dans le le pire des cas. $T(n)$ est donc un majorant du nombre de comparaisons effectuées pour trier une entrée de taille n quelconque. Quand on demande des exemples réalisant un pire cas, on ne demande pas une liste particulière, mais plutôt la forme de telles listes, en fonction du paramètre n (par exemple une liste triée par ordre croissant ou décroissant, une liste constante, etc.).

III.11) Estimer $P(n)$. Donner des exemples qui correspondent au pire des cas.

III.12) Que valent $T(0)$ et $T(1)$?

On suppose que le pire des cas pour la fonction `tri` correspond à celui où lors de chaque appel, en dehors du cas de base, l'une des deux partitions est vide.

III.13) Donner un exemple de forme de liste aboutissant à ce pire cas.

III.14) Trouver, dans ce cas précis, une relation de récurrence vérifiée par $T(n)$ en fonction de $T(n-1)$, pour $n \in \mathbb{N}^*$.

III.15) Montrer qu'alors, pour $n \in \mathbb{N}^*$, $T(n) = \frac{n(n-1)}{2}$.

On va maintenant montrer rigoureusement que ce cas est bien le pire des cas, c'est-à-dire que $T(n) = \frac{n(n-1)}{2}$

III.16) Justifier que $T(n) = \max_{0 \leq p \leq n-1} (T(p) + T(n-p-1)) + n - 1$ pour $n \geq 1$.

III.17) À l'aide d'une récurrence forte sur $n \in \mathbb{N}^*$, montrer le résultat attendu. On pourra étudier la fonction $p \mapsto p(p-1) + (n-p-1)(n-p-2)$ et montrer qu'elle atteint son maximum sur $\{0, \dots, n-1\}$ à ses extrémités.

La complexité dans le pire des cas est donc de l'ordre de $\Theta(n^2)$ ce qui n'est pas très satisfaisant. Nous allons montrer que la complexité *en moyenne*, que l'on note $T_{\text{moy}}(n)$, est cependant bien meilleure. On fait ici l'hypothèse forte consistant à supposer que lors de chaque appel récursif, la position de la valeur du pivot dans la liste une fois triée est équiprobable. On a alors pour $n \in \mathbb{N}^*$:

$$T_{\text{moy}}(n) = \frac{1}{n} \sum_{k=0}^{n-1} \left(T_{\text{moy}}(k) + T_{\text{moy}}(n-k-1) \right) + n - 1$$

III.18) Montrer que pour tout $n \geq 1$ on a

$$\frac{T_{\text{moy}}(n)}{n+1} - \frac{T_{\text{moy}}(n-1)}{n} = 2\frac{n-1}{n(n+1)}$$

III.19) En admettant que $\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$, en déduire que $T_{\text{moy}}(n) = \Theta(n \log n)$.