Outils logiques et algorithmiques – Examen – Mai 2023

Durée 2h. Le sujet comporte 3 exercices indépendants. Notes personnelles et documents de cours autorisés.
Barème approximativement proportionnel au temps nécessaire à la résolution de chaque question.

Exercice 1. Recherche dichotomique sans branchement *(30 minutes)*

Voici une variante de l'algorithme de recherche dichotomique vu en cours, dans laquelle la boucle principale est simplifiée (il y a moins de **`if`**, et aucun **`else`**).

```
1  static int search(int x, int[] t) {
2    int lo = 0;
3    for (int s = t.length; s > 1; s = s/2) {
4      int mid = lo + s/2;
5      if (x >= t[mid]) { lo = mid; }
6    }
7    if (t[lo] == x) { return lo; }
8    else            { return -1; }
9  }
```

Questions.

1. Rappeler la précondition que doit vérifier le tableau `t` pris en argument par la recherche dichotomique.
2. Pour cette question, on se donne un tableau `t0 = {1, 3, 5, 7, 9, 11, 13, 15}`.
   Détailler les exécutions de `search(5, t0)` et `search(12, t0)`, sous la forme de tableaux donnant les valeurs des variables `lo`, `s` et `mid` à chaque étape.
3. Combien cette fonction effectue-t-elle d'accès au tableau `t` pris en argument ? Exprimer le résultat en fonction de la longueur de `t`, que l'on supposera être une puissance de 2.
4. Expliquer le fonctionnement de cet algorithme par un schéma matérialisant le tableau `t`, les différentes variables du programmes, et ce qu'on connaît des valeurs du tableau.
5. Montrer sur un exemple que cette fonction peut échouer si la longueur du tableau `t` n'est pas une puissance de 2.
6. (Bonus) Modifier le code pour qu'il fonctionne quelle que soit la longueur du tableau. *Indication. Quatre caractères suffisent à corriger le programme, mais il vous faudra aussi quelques lignes de texte pour justifier cette modification.*

Exercice 2. *Checking a large routine (30 minutes)*

La fonction suivante calcule la factorielle de l'entier n, c'est-à-dire le produit $n! = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$, en n'utilisant que des additions.

```
1  static int fact(int n) {
2    int u = 1;
3    for (int r=0; r < n; r++) {
4      int v = u;
5      for (int s=1; s <= r; s++) {
6        u += v;
7      }
8    }
9    return u;
10 }
```

Questions.

1. Détailler le calcul réalisé par l'appel `fact(4)`, sous la forme d'un tableau donnant les valeurs des variables `u`, `r`, `v` et `s` à chaque étape.
2. Complexité.
   (a) Donner une expression exacte pour le nombre $A(\mathtt{n})$ d'additions réalisées par cet algorithme, en fonction de la valeur de `n`. *Indication : l'opération* `u += v` *n'est pas la seule addition présente ici.*
   (b) Donner un ordre de grandeur et un équivalent pour $A(\mathtt{n})$.
3. Invariants.
   (a) Donner un invariant pour la boucle externe, liant les valeurs des variables `u` et `r`.
   (b) Donner un invariant pour la boucle interne, liant les valeurs des variables `u`, `s` et `r`.

Contexte : en 1949, Alan Turing a publié à propos de cet algorithme l'une des premières preuves de programme de l'histoire. Notez que le code que je vous fournis est modernisé : le langage java n'existait pas en 1949 ! (ni même la boucle **`for`** *d'ailleurs) Le titre de l'exercice fait référence au titre de l'article de Turing.*

Exercice 3. Sommes cumulatives *(60 minutes)*

On considère une séquence $S[0], S[1], ..., S[N-1]$ de N nombres, indicés de 0 à $N-1$ à la manière des éléments d'un tableau, sur laquelle on souhaite pouvoir réaliser les opérations suivantes.

— La mise à jour $\mathsf{add}(i, x, S)$ modifie S en ajoutant x à la valeur $S[i]$.

— La somme cumulative $\mathsf{csum}(i, S)$ renvoie la somme $S[0] + S[1] + ... + S[i-1]$ des i premiers éléments de S.

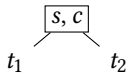
Questions.

1. Supposons que la séquence S est représentée par un tableau `a` tel que `a[i]` $= S[i]$. Décrire les opérations `add` et `csum` et donner des ordres de grandeur pour leur complexité.
2. Supposons maintenant que la séquence S est représentée par un tableau `c` tel que `c[i]` $= S[0] + ... + S[i-1]$. Décrire les opérations `add` et `csum` et donner des ordres de grandeur pour leur complexité.

Arbres de Fenwick. Pour représenter la séquence S de sorte à ce que les opérations `add` et `csum` soient toutes les deux efficaces, on utilise des arbres binaires dans lesquels les feuilles portent les éléments de la séquence, et les nœuds internes portent des sommes cumulatives précalculées.

Plus précisément :

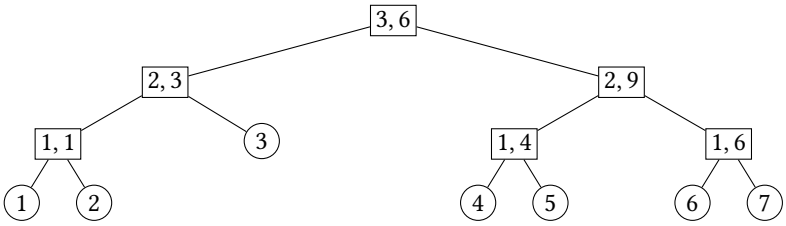
— on note $\mathsf{L}(k)$ et on dessine $\textcircled{k}$ une feuille portant l'élément k

— on note $\mathsf{N}(t_1, s, c, t_2)$ et on dessine  un nœud avec un sous-arbre gauche t_1 et un sous-arbre droit t_2, étiqueté par deux nombres s et c dont on précisera la signification plus tard.

On peut se donner le type caml associé suivant.

```
type tree =
  | L of int
  | N of tree * int * int * tree
```

On dessine ainsi l'arbre $t_0 = \mathsf{N}(\mathsf{N}(\mathsf{N}(\mathsf{L}(1), 1, 1, \mathsf{L}(2)), 2, 3, \mathsf{L}(3)), 3, 6, \mathsf{N}(\mathsf{N}(\mathsf{L}(4), 1, 4, \mathsf{L}(5)), 2, 9, \mathsf{N}(\mathsf{L}(6), 1, 6, \mathsf{L}(7))))$.



Les équations suivantes définissent une fonction size, qui calcule le nombre de feuilles d'un tel arbre, et tsum, qui calcule la somme de tous les éléments des feuilles (à droite : code caml équivalent aux équations données à gauche).

$$\begin{aligned} \mathsf{size}(\mathsf{L}(k)) &= 1 \\ \mathsf{size}(\mathsf{N}(t_1, s, c, t_2)) &= \mathsf{size}(t_1) + \mathsf{size}(t_2) \\ \mathsf{tsum}(\mathsf{L}(k)) &= k \\ \mathsf{tsum}(\mathsf{N}(t_1, s, c, t_2)) &= \mathsf{tsum}(t_1) + \mathsf{tsum}(t_2) \end{aligned}$$

```
let rec size t = match t with
  | L _              -> 1
  | N(t1, _, _, t2) -> size t1 + size t2
let rec tsum t = match t with
  | L k              -> k
  | N(t1, _, _, t2) -> tsum t1 + tsum t2
```

On définit les arbres de Fenwick corrects par les critères suivants.

— Pour tout k, l'arbre $\mathsf{L}(k)$ est un arbre correct.

— L'arbre $\mathsf{N}(t_1, s, c, t_2)$ est correct lorsque les conditions suivantes sont vérifiées :

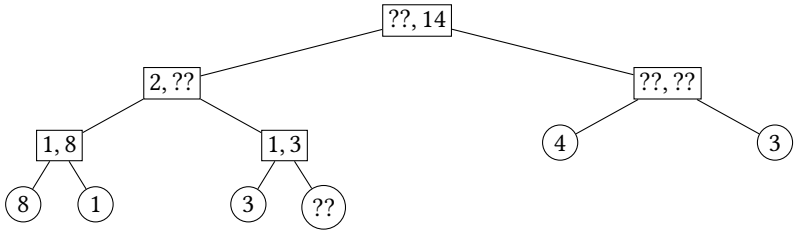
  — Les sous-arbres t_1 et t_2 sont corrects.

  — L'entier s est égal à $\mathsf{size}(t_1)$ et le nombre c est égal à $\mathsf{tsum}(t_1)$.

On peut constater que l'arbre t_0 donné en exemple ci-dessous est bien un arbre de Fenwick correct.

Questions.

3. Dessiner l'arbre de Fenwick représenté par la notation N(N(L(3), 1, 3, L(2)), 2, 5, N(L(7), 1, 7, L(5))).
4. Compléter les éléments manquant à l'arbre suivant, pour former un arbre de Fenwick correct.



Calculs de taille. Voici des équations pour une fonction size', proposant un calcul alternatif du nombre de feuilles d'un arbre.

$$\begin{aligned} \text{size'}(\mathsf{L}(k)) &= 1 \\ \text{size'}(\mathsf{N}(t_1, s, c, t_2)) &= s + \text{size'}(t_2) \end{aligned}$$

```
1 let rec size' t = match t with
2   | L _              -> 1
3   | N(_, s, _, t2) -> s + size' t2
```

Questions.

5. Démontrer que pour tout arbre t correct, on a size'(t) = size(t).
   Indication : par récurrence sur la structure de t.
6. Combien d'additions faut-il réaliser pour calculer la taille d'un arbre t avec la fonction size ?
   Indication : en fonction de la taille et/ou de la hauteur de t.
7. Combien d'additions faut-il réaliser pour calculer la taille d'un arbre t avec la fonction size' ?
   Indication : en fonction de la taille et/ou de la hauteur de t.
8. Définir une fonction tsum', calculant le même résultat que tsum mais plus efficacement.
   Réponse au choix par des équations ou du code caml.

Somme cumulative et mise à jour. Si t est un arbre de Fenwick correct, la fonction get(i, t) suivante donne l'élément porté par la feuille d'indice i (l'indice étant supposé valide).

$$\begin{aligned} \text{get}(0, \mathsf{L}(k)) &= k \\ \text{get}(i, \mathsf{N}(t_1, s, c, t_2)) &= \begin{cases} \text{get}(i, t_1) & \text{si } i < s \\ \text{get}(i - s, t_2) & \text{si } i \geq s \end{cases} \end{aligned}$$

```
1 let rec get i t = match t with
2   | L k ->
3       assert (i=0); k
4   | N(t1, s, _, t2) ->
5       if i < s then get i t1
6                else get (i-s) t2
```

Questions.

9. Détailler l'exécution de get(4, t_0), en montrant notamment quels appels récursifs sont effectués.
10. Définir une fonction csum(i, t) qui renvoie la somme des éléments portés par les i premières feuilles de t, en supposant que t est un arbre de Fenwick correct. La fonction doit réaliser aussi peu d'opérations que possible.
    Réponse au choix par des équations ou du code caml.
11. Supposons que, dans l'arbre exemple t_0, on veuille ajouter 5 à la feuille d'indice 3. La feuille qui portait l'élément 4 porte donc maintenant 9 à la place. Quels autres modifications faut-il apporter à l'arbre pour que le résultat soit toujours un arbre de Fenwick bien formé ?
12. Définir une fonction add(i, x, t) qui prend en paramètre un indice i, une valeur x et un arbre de Fenwick t, et qui calcule un nouvel arbre de Fenwick obtenu en ajoutant x à la feuille d'indice i de t.
    Réponse au choix par des équations ou du code caml.
13. Démontrer que, si t est un arbre de Fenwick correct, alors le résultat de votre fonction add(i, x, t) est toujours un arbre de Fenwick correct.
    Indication : par récurrence sur la structure de t.