SESSION 2002

MPIN08

CONCOURS COMMUNS POLYTECHNIQUES

EPREUVE SPECIFIQUE - FILIERE MP

INFORMATIQUE

Durée : 3 heures

Tournez la page S.V.P.

Les calculatrices sont interdites.

N.B. : Le candidat attachera la plus grande importance à la clarté, à la précision et à la concision de la rédaction.
Si un candidat est amené à repérer ce qui peut lui sembler être une erreur d'énoncé, il le signalera sur sa copie et devra poursuivre sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

PRÉAMBULE : Les trois parties qui composent ce sujet sont indépendantes et peuvent être traitées par les candidats dans un ordre quelconque.

Partie I : Logique et calcul des propositions

Lors de ses aventures au pays des merveilles rapportées par Lewis Carroll, Alice est souvent accompagnée par le chat de Cheshire. Ce félin énigmatique s'exprime sous la forme d'affirmations logiques qui sont toujours vraies.

Alice se trouve dans un corridor dont toutes les portes à sa taille sont fermées. La seule porte ouverte est nettement trop petite pour qu'elle puisse l'emprunter. Une étagère est fixée au-dessus de cette porte. Le chat dit alors à Alice : « L'un des flacons posés sur cette étagère contient un liquide qui te permettra de prendre une taille plus adéquate. Mais attention, les autres flacons peuvent contenir un poison fatal. »

Trois flacons sont effectivement posés sur l'étagère. Le premier est rouge, le second jaune, le troisième bleu. Une étiquette est collée sur chaque flacon. Alice lit l'inscription figurant sur chaque étiquette :

- Flacon rouge : le flacon jaune contient un poison. Le flacon bleu ne contient pas un poison;
- Flacon jaune : si le flacon rouge contient un poison, alors le flacon bleu aussi;
- Flacon bleu : je ne contiens pas un poison, mais au moins l'un des deux autres flacons contient un poison.

Nous noterons R, J et B les variables propositionnelles correspondant au fait que les flacons rouge, jaune et bleu contiennent un poison.

Nous noterons I_R, I_J et I_B les propositions correspondant aux inscriptions sur les flacons rouge, jaune et bleu.

Question I.1 *Exprimer I_R, I_J et I_B sous la forme de formules du calcul des propositions dépendant de R, J et B.*

Traduire les questions suivantes en formules du calcul des propositions puis résoudre les problèmes posés en utilisant le calcul des propositions (formules de De Morgan et/ou tables de vérité).

Question I.2 *Les inscriptions sur les trois flacons sont-elles compatibles (c'est-à-dire, peuvent-elles être toutes vraies) ?*

Question I.3 *Est-ce que les inscriptions sont dépendantes, c'est-à-dire est-ce qu'une inscription est impliquée par la conjonction des deux autres ? Si oui, préciser la (ou les) dépendance(s).*

Question I.4 *Dans le cas où aucun des trois flacons ne contient un poison, est-ce qu'une ou plusieurs inscriptions sont fausses ? Si oui, laquelle ou lesquelles ?*

Question I.5 *Si les trois inscriptions sont vraies, est-ce qu'un ou plusieurs flacons contiennent un poison ? Si oui, lequel ou lesquels ?*

Question I.6 *Si seuls les flacons ne contenant pas un poison ont une inscription vraie, est-ce qu'un ou plusieurs flacons ne contiennent pas un poison ? Si oui, lequel ou lesquels ?*

Partie II : Automates et langages

Le but de cet exercice est l'étude des propriétés de l'opération $\oplus$ de composition de deux automates finis déterministes.

Soit l'alphabet X un ensemble de symboles, soit Λ (parfois noté abusivement ϵ) le symbole représentant le mot vide ($\Lambda \notin X$), soit $X^\star$ l'ensemble des mots composés de symboles de X ($\Lambda \in X^\star$), un automate fini déterministe sur X est un quintuplet $A = (Q,X,i,T,\delta)$ composé de :

- Un ensemble d'états : Q,
- L'état initial : $i \in Q$,
- Un ensemble d'états terminaux : $T \subseteq Q$,
- Une fonction de transition : $\delta : X \times Q \to Q$, définie a priori sur une partie de $X \times Q$.

Les valeurs de δ seront représentées par un graphe dont les nœuds sont les états. Un état initial sera entouré d'un cercle (i) et un état final sera entouré d'un double cercle $((t))$.

Un automate déterministe est dit « complet » si δ est totale (définie sur $X \times Q$ tout entier).

Dans cette partie, les automates finis considérés seront complets.

Soit $\delta^\star$ l'extension de δ à $X^\star \times Q$ définie par :

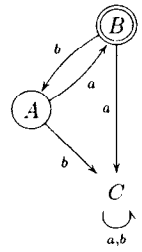
$$\begin{cases} \forall q \in Q,\ \delta^\star(\Lambda,q) = q \\ \forall x \in X, \forall q \in Q,\ \delta^\star(x,q) = \delta(x,q) \\ \forall m \in X^\star, \forall x \in X, \forall q \in Q,\ \delta^\star(m.x,q) = \delta(x,\delta^\star(m,q)) \end{cases}$$

Le langage sur $X^\star$ reconnu par cet automate fini est :

$$L(A) = \{m \in X^\star \mid t \in T, \delta^\star(m,i) = t\}$$

Étude de l'exemple $\mathcal{E}_1$

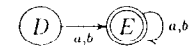
Soit l'automate fini déterministe complet $\mathcal{E}_1 = (\{A,B,C\},\{a,b\},A,\{B\},\delta_{\mathcal{E}_1})$ dont la fonction de transition est définie par :



Question II.1 *Donner sans la justifier une expression régulière ou ensembliste représentant le langage reconnu par $\mathcal{E}_1$.*

Étude de l'automate $\mathcal{E}_2$

Soit l'automate fini déterministe complet $\mathcal{E}_2 = (\{D,E\},\{a,b\},D,\{E\},\delta_{\mathcal{E}_2})$ dont la fonction de transition est définie par :



Question II.2 *Donner sans la justifier une expression régulière ou ensembliste représentant le langage reconnu par $\mathcal{E}_2$.*

Composition des automates : $\mathcal{E}_1 \oplus \mathcal{E}_2$

Soit l'opération interne $\oplus$ sur les automates finis déterministes définie par :

Déf. II.1 (Composition d'automates) *Soient $\mathcal{A}_1 = (Q_1,X,i_1,T_1,\delta_1)$ et $\mathcal{A}_2 = (Q_2,X,i_2,T_2,\delta_2)$ deux automates finis déterministes, l'automate $\mathcal{A} = \mathcal{A}_1 \oplus \mathcal{A}_2$ est défini par :*

$$\mathcal{A} = (Q_1 \times Q_2,X,(i_1,i_2),(T_1 \times (Q_2 \setminus T_2)) \cup ((Q_1 \setminus T_1) \times T_2),\delta_{1\oplus 2})$$
$$\delta_{1\oplus 2}(a,(x,y)) = (x',y') \text{ si } \delta_1(a,x) = x' \text{ et } \delta_2(a,y) = y'$$

On remarquera que les états de $\mathcal{A}$ sont des paires d'états de $\mathcal{A}_1$ et $\mathcal{A}_2$.

Question II.3 *Construire l'automate $\mathcal{E}_1 \oplus \mathcal{E}_2$ (seuls les états et les transitions utiles, c'est-à-dire accessibles depuis l'état initial, devront être construits).*

Question II.4 *Caractériser le langage reconnu par $\mathcal{E}_1 \oplus \mathcal{E}_2$ par une expression régulière ou ensembliste.*

Étude de l'automate « composé »

Question II.5 *Montrer que : si $\mathcal{A}_1$ et $\mathcal{A}_2$ sont des automates finis déterministes complets, alors $\mathcal{A}_1 \oplus \mathcal{A}_2$ est un automate fini déterministe complet.*

Question II.6 *Montrer que* $\forall q_1 \in Q_1, \forall q_2 \in Q_2,\ \delta^\star_{1\oplus 2}(m,(q_1,q_2)) = (\delta^\star_1(m,q_1),\delta^\star_2(m,q_2))$

Question II.7 *Montrer que :* $m \in L(\mathcal{A}_1 \oplus \mathcal{A}_2) \Leftrightarrow (m \in L(\mathcal{A}_1) \vee m \in L(\mathcal{A}_2)) \wedge (m \notin (L(\mathcal{A}_1) \cap L(\mathcal{A}_2))$.

Question II.8 *Quelle relation liant les langages reconnus par les automates $\mathcal{A}_1$, $\mathcal{A}_2$ et $\mathcal{A}_1 \oplus \mathcal{A}_2$ peut-on en déduire ?*

Partie III : Algorithmique et programmation en CaML

Cette partie doit être traitée par les étudiants qui ont utilisé le langage CaML dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (`for`, `while`, ...) ni de références.

L'explication du comportement d'une fonction doit au moins contenir :

- L'objectif général,
- Le rôle des paramètres de la fonction,
- Les contraintes sur les valeurs des paramètres,
- Les caractéristiques du résultat renvoyé par la fonction,
- Le rôle des variables locales à la fonction,
- Le principe de l'algorithme,
- Des arguments de terminaison du calcul quelles que soient les valeurs des paramètres.

Les équipements informatiques les plus courants sont basés sur des composants électroniques pour lesquels une information correspond à la présence ou l'absence d'un signal électrique sur une connexion. L'unité élémentaire d'information (dite binaire) est donc le « bit » prenant deux valeurs : 0 et 1.

Toutes les données sont ensuite codées en base 2 en utilisant un nombre suffisant de bits pour représenter toutes les valeurs utiles. Dans un cadre général où n'importe quelle valeur peut apparaître un nombre de fois quelconque, il est nécessaire d'utiliser un codage « uniforme » qui exploite le même nombre de bits pour représenter n'importe quelle valeur d'une même donnée. Par contre, si l'ensemble des valeurs possibles est connu ainsi que le nombre de fois où elles apparaissent (leur nombre d'occurrences), il est possible de choisir des codages qui utilisent moins de bits pour coder une même suite de valeurs. Ce type de codage, appelé stochastique, est couramment utilisé pour compresser des données, c'est-à-dire réduire l'espace occupé par une information connue a priori.

Nous allons étudier dans cette partie une technique de construction d'un codage minimal due à Huffman. Nous appliquerons cette technique à la compression et la décompression d'une suite de caractères dont le contenu est connu a priori.

1 Principe de l'algorithme

La suite de n caractères que nous allons traiter est notée $c_1 \dots c_n$. Un même caractère peut apparaître plusieurs fois. La suite est donc composée de p caractères distincts appartenant à l'ensemble $\{v_1, \dots, v_p\}$ avec $p \leq n$. Nous appellerons nombre d'occurrences d'un caractère v_i de la suite le nombre de fois que ce caractère figure dans la suite. La fonction $\mathcal{O}(v_i)$ renvoie le nombre d'occurrences de v_i dans la suite étudiée.

1.1 Principe du codage

Dans une première étape, définissons formellement la notion de codage.

1.1.1 Codage d'un caractère

Déf. III.1 *Une clé de codage binaire d'un ensemble de caractères est une application qui associe à chaque caractère un code composé d'une suite non vide de valeurs binaires (0 ou 1).*

Exemple III.1 $\{a \mapsto 00, b \mapsto 010, c \mapsto 011, d \mapsto 1\}$, $\{a \mapsto 00, b \mapsto 01, c \mapsto 10, d \mapsto 11\}$ *et* $\{a \mapsto 0, b \mapsto 01, c \mapsto 1, d \mapsto 10\}$ *sont des clés de codage possibles pour l'ensemble* {a,b,c,d}. *La seconde clé est uniforme car elle utilise le même nombre de bits pour chaque caractère.*

1.1.2 Codage d'une suite

Une clé de codage permet de coder une suite de caractères par une suite de bits en remplaçant chaque caractère de la suite par son code.

Si nous considérons la suite *abcd*, son code est 000100111 selon la première clé et 00011011 selon la clé uniforme (seconde clé). Remarquons que le premier codage utilise 9 bits alors que le codage uniforme n'utilise que 8 bits.

Par contre, si nous considérons la suite *dadbcd*, son code est 10010100111 selon la première clé et 110011011011 selon la clé uniforme. Remarquons que le codage uniforme utilise 12 bits alors que le premier codage n'utilise que 11 bits.

La première clé de codage semble donc plus adaptée aux suites contenant un plus grand nombre de d que de a, b ou c.

1.1.3 Représentation en CaML

Un caractère est représenté par le type de base `char`.

Une suite de caractères est représentée par le type `suite` équivalent à une liste de caractères.

Un code binaire est représenté par le type `code` équivalent à une liste de booléens.

Une clé de codage qui associe à un caractère son code binaire est représentée par le type `cle` équivalent à une liste de paires caractère/code binaire.

```
type suite == char list;;
type code == boolean list;;
type cle == (char * code) list;;
```

1.1.4 Application au codage d'une suite

Nous supposons que nous disposons déjà d'une clé de codage nous permettant de coder la suite de caractères.

Question III.1 *Écrire en CaML une fonction* `coder` *de type* `suite -> cle -> code` *telle que l'appel* `(coder s c)` *renvoie le code de la suite de caractères* `s` *selon la clé de codage* `c`. *Nous supposons que la clé* `c` *contient tous les caractères de la suite* `s`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.2 *Expliquer la fonction* `coder` *proposée pour la question précédente.*

Question III.3 *Calculer une estimation de la complexité dans le pire cas de la fonction* `coder` *en fonction de la taille des listes* `s` *et* `c`. *Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.*

1.1.5 Propriétés d'un codage

La définition précédente est trop générale pour permettre une opération de décodage simple. Nous étudierons donc des codages possédant des propriétés supplémentaires.

Nous noterons $\mathcal{I}$ l'image de $\{v_1, \ldots, v_p\}$ selon la clé de codage. $\mathcal{I}$ contient donc l'ensemble des valeurs de code possibles.

Déf. III.2 *Une clé de codage est séparable si et seulement si :*

$$\forall b \in \mathcal{I}, b = b_1 \ldots b_r, \forall i \in [1,r-1], b_1 \ldots b_i \notin \mathcal{I}$$

Cette propriété indique que le code d'un caractère n'est jamais un préfixe du code d'un autre caractère. Elle est nécessaire pour permettre le décodage.

Les deux premiers exemples sont des clés séparables. Le troisième exemple n'est pas séparable.

Déf. III.3 *Une clé de codage est optimale si et seulement si :*

$$\forall b \in \mathcal{I}, b = b_1 \ldots b_r, \forall i \in [1,r], \exists b_1 \ldots b_{i-1} b'_i b'_{i+1} \ldots b'_s \in \mathcal{I}, b'_i = \neg b_i$$

Cette propriété indique que le i-ème bit permet de distinguer $b_1 \ldots b_r$ de $b_1 \ldots b'_s$.

Les deux premiers exemples sont optimaux. Le troisième exemple n'est pas optimal.

Cette propriété permet de minimiser la taille des codes indépendamment de la suite considérée. La propriété suivante prend celle-ci en compte.

Nous noterons $\mathcal{C}(v)$ le nombre de bits utilisé pour coder le caractère v.

Le coût du codage de la suite est alors : $\mathcal{C}(c_1 \ldots c_n) = \sum_{i=1}^{p} \mathcal{O}(v_i) \times \mathcal{C}(v_i)$

Déf. III.4 *Un codage est minimal pour une suite de caractères donnée s'il permet de coder tous les caractères en utilisant le plus petit nombre de bits possible lors du codage de la suite, c'est-à-dire s'il permet de minimiser $\mathcal{C}(c_1 \ldots c_n)$.*

L'algorithme étudié dans cette partie construit une clé de codage optimale séparable assurant un codage minimal pour une suite de caractères donnée. Nous appellerons cette clé « *minimale* ».

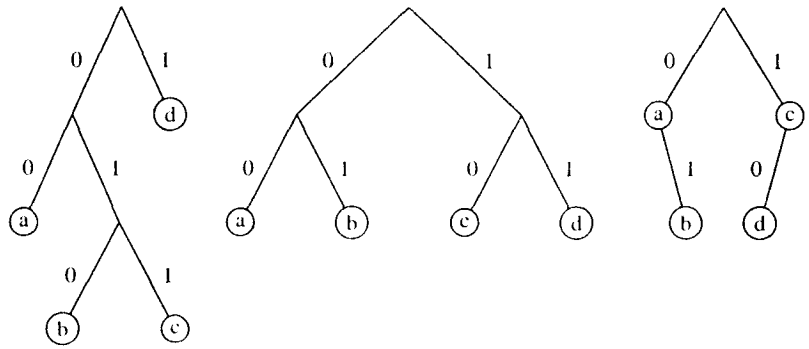
1.2 Arbre de Huffman

La clé de codage d'un ensemble de caractères peut être représentée par un arbre binaire dont certains nœuds sont étiquetés par un caractère et toutes les arêtes sont étiquetées par la valeur d'un bit de code.

Nous considérerons uniquement des arbres dont toutes les feuilles contiennent un caractère.

1.2.1 Structure arborescente d'un codage

Les arbres suivants correspondent aux trois exemples précédents :



Déf. III.5 *Un arbre binaire est complet si et seulement si chaque nœud de cet arbre possède soit deux fils, soit aucun fils (ces nœuds sont alors appelés des feuilles).*

Question III.4 *Montrer que la clé de codage associée à un arbre complet est optimale.*

Question III.5 *Montrer que la clé de codage associée à un arbre complet est séparable si et seulement si seules les feuilles de cet arbre contiennent des caractères.*

Les arbres utilisés pour construire une clé de codage optimale séparable seront donc tels que :

- les nœuds ont deux fils et ne contiennent pas de caractère;
- les feuilles n'ont pas de fils et contiennent un caractère.

Nous appellerons ces arbres des arbres de codage. Nous supposerons par la suite qu'un même caractère ne figure pas dans plusieurs feuilles distinctes.

Déf. III.6 *Soit une suite $c_1 \ldots c_n$ de caractères appartenant à l'ensemble $\{v_1, \ldots, v_p\}$ avec les nombres d'occurrences $\mathcal{O}(v_i)$, à chaque arbre de codage de cet ensemble est associé un arbre appelé arbre de Huffman obtenu en ajoutant à chaque feuille de l'arbre de codage contenant le caractère v_j l'occurrence $\mathcal{O}(v_j)$ du caractère v_j dans la suite $c_1 \ldots c_n$. Chaque feuille de l'arbre de Huffman contient donc un caractère v_j et le nombre d'occurrence $\mathcal{O}(v_j)$ de v_j dans $c_1 \ldots c_n$.*

1.2.2 Représentation en CaML

Un arbre de Huffman est représenté par le type `arbre` :

```
type arbre =
  Vide |
  Feuille of char * int |
  Noeud of arbre * arbre;;
```

Une feuille contient un caractère ainsi que le nombre d'occurrences de ce caractère dans la suite $c_1 \ldots c_n$. Le cas `Vide` permet de construire des arbres non complets. Il ne sera utilisé que lors de la reconstruction de l'arbre dans la phase de décodage.

Exemple III.2 *Les deux premiers exemples précédents seront codés pour la suite daacadbcd par :*

```
Noeud(
   Noeud(Feuille('a',3),
         Noeud(Feuille('b',1),Feuille('c',2))),
   Feuille('d',3))

Noeud(
   Noeud(Feuille('a',3),Feuille('b',1)),
   Noeud(Feuille('c',2),Feuille('d',3)))
```

Le dernier exemple ne peut pas être codé car les nœuds ne peuvent pas contenir de caractères (clé de codage séparable).

Une liste d'arbres est représentée par le type `liste`.

```
type liste == arbre list;;
```

1.2.3 Calcul du nombre d'occurrences

Les feuilles des arbres sont étiquetées par un caractère et par un entier qui indique le nombre d'occurrences du caractère dans la suite étudiée. Le nombre global des occurrences des différents caractères qui figurent dans un arbre correspond alors à la somme des nombres d'occurrences de toutes les feuilles de cet arbre. La fonction $\mathcal{O}(a)$ est ainsi étendue à l'arbre a.

Question III.6 *Écrire en CaML une fonction* `nombre` *de type* `arbre -> int` *telle que l'appel* `(nombre a)` *renvoie le nombre d'occurrences* $\mathcal{O}$(a) *des caractères figurant dans l'arbre* a. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

1.3 Codage de Huffman

1.3.1 Principe général

L'algorithme de Huffman exploite la connaissance du contenu de la suite à coder pour générer une clé de codage minimale. Son principe repose sur la construction d'un arbre de Huffman minimal par transformations successives d'un arbre de Huffman quelconque.

L'algorithme découle de l'idée naturelle suivante : « *Pour minimiser le nombre de bits utilisés par le codage d'une suite de caractères, il faut associer le plus petit nombre de bits aux caractères les plus fréquents et le plus grand nombre de bits aux caractères les moins fréquents* ».

La longueur d'un code correspond à la profondeur du caractère associé dans l'arbre. Pour optimiser les codes, il faut donc que les caractères les plus rares étiquettent les feuilles les plus profondes de l'arbre et que les caractères les plus nombreux étiquettent les feuilles les moins profondes.

Réduire le coût d'un codage consiste donc à échanger une des feuilles les plus profondes de l'arbre avec un sous-arbre dont la profondeur et le nombre d'occurrences sont strictement plus petits.

Pour éviter les nombreux parcours d'arbres intermédiaires non minimaux pour rechercher les candidats à un échange, l'algorithme de Huffman construit directement l'arbre de Huffman minimal. Pour cela, il manipule un ensemble de sous-arbres de Huffman minimaux partiels. Cet ensemble contient initialement l'ensemble des feuilles étiquetées par les caractères de $\{v_1, \ldots, v_p\}$. Chaque étape de l'algorithme remplace les deux sous-arbres g et d contenant les plus petits nombres d'occurrences par un

arbre de Huffman dont les fils sont g et d. L'algorithme s'arrête lorsque l'ensemble ne contient plus qu'un seul arbre qui est alors l'arbre de Huffman minimal de l'ensemble des caractères de la suite.

Question III.7 *Appliquer l'algorithme précédent pour construire l'arbre de Huffman correspondant à l'exemple dadbcd en détaillant chaque ensemble intermédiaire.*

1.3.2 Preuve de minimalité

Considérons H un arbre de Huffman de $\{v_1, \cdots, v_p\}$. L'arbre H' est obtenu en permutant les feuilles v_i et v_j dans H.

Question III.8 *Calculer la différence, notée Δ, entre le coût de H et le coût de H' en fonction des occurrences et profondeurs de v_i et v_j.*

Question III.9 *Montrer que dans un arbre de Huffman minimal, les feuilles de profondeur maximale contiennent des caractères d'occurrence minimale.*

Considérons H un arbre de Huffman minimal de $\{v_1, \ldots, v_p\}$. v_i et v_j sont deux caractères d'occurrence minimale possédant un même nœud père n dans H. L'arbre H' est obtenu en enlevant les feuilles v_i et v_j dans H et en considérant que n est une feuille d'occurrence $\mathcal{O}(n) = \mathcal{O}(v_i) + \mathcal{O}(v_j)$.

Question III.10 *Montrer que H' est un arbre de Huffman minimal (Indication : supposons que H est minimal et que H' n'est pas minimal).*

Question III.11 *Montrer que l'arbre de Huffman construit par l'algorithme de Huffman est minimal (Indication : H correspond à un problème de taille n, H' correspond à un problème de taille $n-1$).*

2 Préliminaires : Tri par insertion

L'algorithme de Huffman extrait de l'ensemble les sous-arbres contenant les plus petits nombres d'occurrences. Pour cela, nous exploiterons une liste de sous-arbres triée selon le nombre d'occurrences. Cette liste sera triée par un algorithme de tri par insertion.

Question III.12 *Écrire en CaML une fonction* `comparer` *de type* `arbre -> arbre -> boolean` *telle que l'appel* `(comparer a1 a2)` *renvoie la valeur faux si* $\mathcal{O}$(a2) *est strictement plus petit que* $\mathcal{O}$(a1) *et la valeur vrai sinon.*

Question III.13 *Écrire en CaML une fonction* `inserer` *de type* `arbre -> liste -> liste` *telle que l'appel* `(inserer a l)` *renvoie la liste d'arbres obtenue en insérant l'arbre* a *dans la liste* `l` *en respectant l'ordre croissant du nombre d'occurrences. Nous supposons que la liste* `l` *respecte l'ordre croissant du nombre d'occurrences. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.14 *Écrire en CaML une fonction* `trier` *de type* `liste -> liste` *telle que l'appel* `(trier l)` *renvoie une liste d'arbres triée selon l'ordre croissant des nombres d'occurrences et composée des mêmes arbres que* `l`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

3 Construction des codes de Huffman

Pour calculer le code minimal pour chaque caractère de la suite, il faut déterminer l'ensemble des caractères distincts et le nombre d'occurrences de chaque caractère, puis construire l'arbre par fusion successive des éléments de cet ensemble.

3.1 Construction de la liste d'occurrences

Question III.15 *Écrire en CaML une fonction* `ajouter` *de type* `char -> liste -> liste` *telle que l'appel* `(ajouter v l)` *renvoie une liste de feuilles étiquetées par les mêmes caractères que* `l` *et telle que si* `l` *contenait* `v` *alors son occurrence est augmentée de 1 et si* `l` *ne contenait pas* `v`*, le résultat contient* `v` *avec l'occurrence 1. Nous supposons que* `v` *figure au plus une fois dans* `l`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.16 *Écrire en CaML une fonction* `compter` *de type* `suite -> liste` *telle que l'appel* `(compter s)` *renvoie une liste de feuilles étiquetées par les mêmes caractères que* `s` *et par les nombres d'occurrences de ces caractères dans* `s`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

3.2 Construction de l'arbre

L'algorithme de Huffman de construction de l'arbre minimal présenté dans la sous-section 1.3.1 consiste à remplacer les deux sous-arbres ayant les plus petits nombres d'occurrences par le nœud ayant pour fils ces deux sous-arbres tout en préservant l'ordre de la liste triée.

Question III.17 *Écrire en CaML une fonction* `fusionner` *de type* `liste -> arbre` *telle que l'appel* `(fusionner l)` *sur une liste non vide de feuilles* `l` *triée selon le nombre d'occurrences des caractères renvoie un arbre de Huffman contenant les mêmes caractères que la liste* `l` *et correspondant au codage minimal de la suite. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.18 *Expliquer la fonction* `fusionner` *proposée pour la question précédente.*

Question III.19 *Écrire en CaML une fonction* `huffman` *de type* `suite -> arbre` *telle que l'appel* `(huffman s)` *renvoie un arbre de Huffman contenant les mêmes caractères que la suite* `s` *et correspondant au codage minimal de la suite.*

4 Codage d'une suite

L'étape précédente a permis de construire un arbre dont les chemins qui conduisent à chaque caractère sont étiquetés implicitement (0 pour le fils gauche, 1 pour le fils droit) par le code associé à ce caractère. Pour éviter de multiples parcours de l'arbre lors du codage d'une suite de caractères, il faut construire une clé de codage à partir de l'arbre qui associe à chaque caractère son code puis parcourir la suite pour remplacer chaque caractère par son code en utilisant la fonction `coder` proposée dans la question III.1.

4.1 Construction de la clé

Question III.20 *Écrire en CaML une fonction* `construire` *de type* `arbre -> cle` *telle que l'appel* `(construire a)` *renvoie une clé de codage associant à chaque caractère contenu dans l'arbre de Huffman* `a` *son code, c'est-à-dire la suite de valeurs binaires (0 ou 1) correspondant au chemin suivi dans l'arbre pour atteindre le caractère. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

5 Décodage d'une suite

Le principal défaut du codage de Huffman est la nécessité de transmettre la clé de codage avec chaque suite codée. En effet, il faut reconstruire l'arbre de Huffman propre à chaque suite codée pour décoder celle-ci par un parcours de l'arbre en suivant chaque code de la suite codée.

5.1 Reconstruction de la clé

Question III.21 *Écrire en CaML une fonction* `reconstruire` *de type* `cle -> arbre` *telle que l'appel* `(reconstruire c)` *renvoie un arbre de Huffman obtenu à partir de la clé de codage* `c`*, c'est-à-dire contenant les mêmes caractères que* `c` *placées dans l'arbre selon le code associé au caractère dans la clé* `c`*. L'occurrence du caractère* `c` *prendra la valeur 0. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

5.2 Décodage de la suite

Question III.22 *Écrire en CaML une fonction* `decoder` *de type* `suite -> arbre -> suite` *telle que l'appel* `(decoder s a)` *renvoie une suite de caractères correspondant au décodage de la suite de valeurs binaires (0 ou 1) contenues dans* `s`*. Ce décodage est effectué par un parcours de l'arbre de Huffman* `a` *en suivant les chemins correspondant aux valeurs binaires contenues dans* `s`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

6 Optimisation de la construction de l'arbre

Question III.23 *Quelle est la fonction la plus souvent appelée lors de la construction de l'arbre ?*

Question III.24 *Proposer une ou plusieurs modifications des structures de données et/ou des fonctions pour réduire le nombre d'appels à cette fonction.*

La construction de l'arbre exploite une liste triée d'arbres. L'algorithme étudié dans la section 3.1 construit la liste dans un ordre quelconque puis effectue un tri de la liste obtenue. Il est possible d'éviter ce tri par une construction de la liste qui assure que les feuilles soient triées selon l'ordre croissant des nombres d'occurrences.

Question III.25 *Écrire en CaML une fonction* `ajouter` *de type* `char -> liste -> liste` *telle que l'appel* `(ajouter v l)` *sur une liste* `l` *de feuilles triée selon l'ordre croissant des nombres d'occurrences renvoie une liste de feuilles triée selon l'ordre*

croissant des nombres d'occurrences, étiquetées par les mêmes caractères que l et telle que si l contenait v alors son occurrence est augmentée de 1 et si l ne contenait pas v, le résultat contient v avec l'occurrence 1. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.

Partie III : Algorithmique et programmation en PASCAL

Cette partie doit être traitée par les étudiants qui ont utilisé le langage PASCAL dans le cadre des enseignements d'informatique. Les fonctions écrites devront être récursives ou faire appel à des fonctions auxiliaires récursives. Elles ne devront pas utiliser d'instructions itératives (`for`, `while`, `repeat`, ...).

L'explication du comportement d'une fonction doit au moins contenir :

- L'objectif général,
- Le rôle des paramètres de la fonction,
- Les contraintes sur les valeurs des paramètres,
- Les caractéristiques du résultat renvoyé par la fonction,
- Le rôle des variables locales à la fonction,
- Le principe de l'algorithme,
- Des arguments de terminaison du calcul quelles que soient les valeurs des paramètres.

Les équipements informatiques les plus courants sont basés sur des composants électroniques pour lesquels une information correspond à la présence ou l'absence d'un signal électrique sur une connexion. L'unité élémentaire d'information (dite binaire) est donc le « bit » prenant deux valeurs : 0 et 1.

Toutes les données sont ensuite codées en base 2 en utilisant un nombre suffisant de bits pour représenter toutes les valeurs utiles. Dans un cadre général où n'importe quelle valeur peut apparaître un nombre de fois quelconque, il est nécessaire d'utiliser un codage « uniforme » qui exploite le même nombre de bits pour représenter n'importe quelle valeur d'une même donnée. Par contre, si l'ensemble des valeurs possibles est connu ainsi que le nombre de fois où elles apparaissent (leur nombre d'occurrences), il est possible de choisir des codages qui utilisent moins de bits pour coder une même suite de valeurs. Ce type de codage, appelé stochastique, est couramment utilisé pour compresser des données, c'est-à-dire réduire l'espace occupé par une information connue a priori.

Nous allons étudier dans cette partie une technique de construction d'un codage minimal due à Huffman. Nous appliquerons cette technique à la compression et la décompression d'une suite de caractères dont le contenu est connu a priori.

1 Principe de l'algorithme

La suite de n caractères que nous allons traiter est notée $c_1 \dots c_n$. Un même caractère peut apparaître plusieurs fois. La suite est donc composée de p caractères distincts appartenant à l'ensemble $\{v_1, \dots, v_p\}$ avec $p \leq n$. Nous appellerons nombre d'occurrences d'un caractère v_i de la suite le nombre de fois que ce caractère figure dans la suite. La fonction $\mathcal{O}(v_i)$ renvoie le nombre d'occurrences de v_i dans la suite étudiée.

1.1 Principe du codage

Dans une première étape, définissons formellement la notion de codage.

1.1.1 Codage d'un caractère

Déf. III.1 *Une clé de codage binaire d'un ensemble de caractères est une application qui associe à chaque caractère un code composé d'une suite non vide de valeurs binaires (0 ou 1).*

Exemple III.1 $\{a \mapsto 00, b \mapsto 010, c \mapsto 011, d \mapsto 1\}$, $\{a \mapsto 00, b \mapsto 01, c \mapsto 10, d \mapsto 11\}$ *et* $\{a \mapsto 0, b \mapsto 01, c \mapsto 1, d \mapsto 10\}$ *sont des clés de codage possibles pour l'ensemble* {a,b,c,d}. *La seconde clé est uniforme car elle utilise le même nombre de bits pour chaque caractère.*

1.1.2 Codage d'une suite

Une clé de codage permet de coder une suite de caractères par une suite de bits en remplaçant chaque caractère de la suite par son code.

Si nous considérons la suite *abcd*, son code est 000100111 selon la première clé et 00011011 selon la clé uniforme (seconde clé). Remarquons que le premier codage utilise 9 bits alors que le codage uniforme n'utilise que 8 bits.

Par contre, si nous considérons la suite *dadbcd*, son code est 10010100111 selon la première clé et 110011011011 selon la clé uniforme. Remarquons que le codage uniforme utilise 12 bits alors que le premier codage n'utilise que 11 bits.

La première clé de codage semble donc plus adaptée aux suites contenant un plus grand nombre de d que de a, b ou c.

1.1.3 Représentation en PASCAL

Un caractère est représenté par le type de base `CHARACTER`.

Une suite de caractères est représentée par le type de base `SUITE` représentant une liste de caractères.

Un code binaire est représenté par le type de base `CODE` représentant une liste de booléens.

Une clé de codage qui associe à un caractère son code binaire est représentée par le type de base `CLE` représentant une liste de paires caractère/code binaire.

Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :

- `NIL` représente la suite, le code et la clé de codage vide,
- `FUNCTION S_Ajouter(c:CHARACTER;s:SUITE):SUITE;` renvoie une suite de caractères composée d'une première valeur `c` et du reste de la suite contenu dans `s`,
- `FUNCTION S_Premier(s:SUITE):CHARACTER;` renvoie le premier caractère de la suite `s`,
- `FUNCTION S_Reste(s:SUITE):SUITE;` renvoie le reste de la suite `s` privée de son premier caractère,
- `FUNCTION S_Juxtaposer(s1,s2:SUITE):SUITE;` renvoie la suite composée de la suite `s1` suivie de la suite `s2`;
- `FUNCTION C_Ajouter(b:BOOLEAN;c:CODE):CODE;` renvoie un code composé d'un premier bit `b` et du reste du code contenu dans `c`,
- `FUNCTION C_Premier(c:CODE):BOOLEAN;` renvoie le premier bit du code `c`,

- `FUNCTION C_Reste(c:CODE):CODE;` renvoie le reste du code `c` privée de son premier bit,
- `FUNCTION C_Juxtaposer(c1,c2:CODE):CODE;` renvoie le code composé du code `c1` suivi du code `c2`;
- `FUNCTION D_Ajouter(v:CHARACTER;c:CODE;d:CLE):CLE;` renvoie une clé composée d'un premier caractère `v`, de son code `c` et du reste de la clé contenu dans `d`,
- `FUNCTION D_Caractere(d:CLE):CHARACTER;` renvoie le premier caractère de la clé `d`,
- `FUNCTION D_Code(d:CLE):CODE;` renvoie le code du premier caractère de la clé `d`,
- `FUNCTION D_Reste(c:CLE):CLE;` renvoie le reste de la clé `d` privée de son premier caractère et du code de celui-ci,
- `FUNCTION D_Juxtaposer(d1,d2:CLE):CLE;` renvoie la clé composée de la clé `d1` suivi de la clé `d2`.

1.1.4 Application au codage d'une suite

Nous supposons que nous disposons déjà d'une clé de codage nous permettant de coder la suite de caractères.

Question III.1 *Écrire en PASCAL une fonction `coder(s:SUITE;c:CLE):CODE;` telle que l'appel `coder(s,c)` renvoie le code de la suite de caractères `s` selon la clé de codage `c`. Nous supposons que la clé `c` contient tous les caractères de la suite `s`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.2 *Expliquer la fonction `coder` proposée pour la question précédente.*

Question III.3 *Calculer une estimation de la complexité dans le pire cas de la fonction `coder` en fonction de la taille des listes `s` et `c`. Cette estimation ne prendra en compte que le nombre d'appels récursifs effectués.*

1.1.5 Propriétés d'un codage

La définition précédente est trop générale pour permettre une opération de décodage simple. Nous étudierons donc des codages possédant des propriétés supplémentaires.

Nous noterons $\mathcal{I}$ l'image de $\{v_1, \ldots, v_p\}$ selon la clé de codage. $\mathcal{I}$ contient donc l'ensemble des valeurs de code possibles.

Déf. III.2 *Une clé de codage est séparable si et seulement si :*

$$\forall b \in \mathcal{I}, b = b_1 \ldots b_r, \forall i \in [1,r-1], b_1 \ldots b_i \notin \mathcal{I}$$

Cette propriété indique que le code d'un caractère n'est jamais un préfixe du code d'un autre caractère. Elle est nécessaire pour permettre le décodage.

Les deux premiers exemples sont des clés séparables. Le troisième exemple n'est pas séparable.

Déf. III.3 *Une clé de codage est optimale si et seulement si :*

$$\forall b \in \mathcal{I}, b = b_1 \ldots b_r, \forall i \in [1,r], \exists b_1 \ldots b_{i-1} b'_i b'_{i+1} \ldots b'_s \in \mathcal{I}, b'_i = \neg b_i$$

Cette propriété indique que le i-ème bit permet de distinguer $b_1 \ldots b_r$ de $b_1 \ldots b'_s$.

Les deux premiers exemples sont optimaux. Le troisième exemple n'est pas optimal.

Cette propriété permet de minimiser la taille des codes indépendamment de la suite considérée. La propriété suivante prend celle-ci en compte.

Nous noterons $\mathcal{C}(v)$ le nombre de bits utilisé pour coder le caractère v.

Le coût du codage de la suite est alors : $\mathcal{C}(c_1 \ldots c_n) = \sum_{i=1}^{p} \mathcal{O}(v_i) \times \mathcal{C}(v_i)$

Déf. III.4 *Un codage est minimal pour une suite de caractères donnée s'il permet de coder tous les caractères en utilisant le plus petit nombre de bits possible lors du codage de la suite, c'est-à-dire s'il permet de minimiser $\mathcal{C}(c_1 \ldots c_n)$.*

L'algorithme étudié dans cette partie construit une clé de codage optimale séparable assurant un codage minimal pour une suite de caractères donnée. Nous appellerons cette clé « ***minimale*** ».

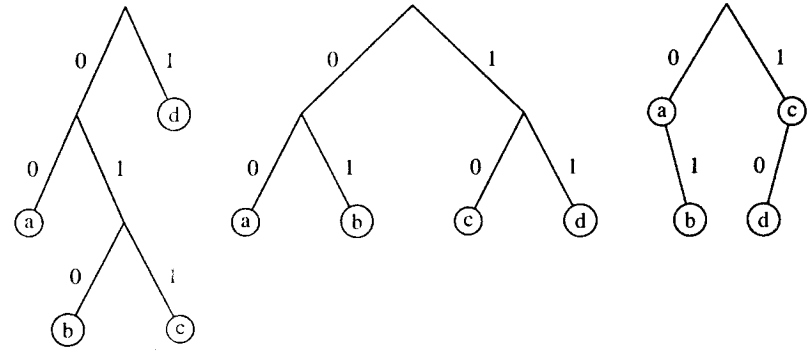
1.2 Arbre de Huffman

La clé de codage d'un ensemble de caractères peut être représentée par un arbre binaire dont certains nœuds sont étiquetés par un caractère et toutes les arêtes sont étiquetées par la valeur d'un bit de code.

Nous considérerons uniquement des arbres dont toutes les feuilles contiennent un caractère.

1.2.1 Structure arborescente d'un codage

Les arbres suivants correspondent aux trois exemples précédents :



Déf. III.5 *Un arbre binaire est complet si et seulement si chaque nœud de cet arbre possède soit deux fils, soit aucun fils (ces nœuds sont alors appelés des feuilles).*

Question III.4 *Montrer que la clé de codage associée à un arbre complet est optimale.*

Question III.5 *Montrer que la clé de codage associée à un arbre complet est séparable si et seulement si seules les feuilles de cet arbre contiennent des caractères.*

Les arbres utilisés pour construire une clé de codage optimale séparable seront donc tels que :

- les nœuds ont deux fils et ne contiennent pas de caractère;
- les feuilles n'ont pas de fils et contiennent un caractère.

Nous appellerons ces arbres des arbres de codage. Nous supposerons par la suite qu'un même caractère ne figure pas dans plusieurs feuilles distinctes.

Déf. III.6 *Soit une suite $c_1 \dots c_n$ de caractères appartenant à l'ensemble $\{v_1, \dots, v_p\}$ avec les nombres d'occurrences $\mathcal{O}(v_i)$, à chaque arbre de codage de cet ensemble est associé un arbre appelé arbre de Huffman obtenu en ajoutant à chaque feuille de l'arbre de codage contenant le caractère v_j l'occurrence $\mathcal{O}(v_j)$ du caractère v_j dans la suite $c_1 \dots c_n$. Chaque feuille de l'arbre de Huffman contient donc un caractère v_j et le nombre d'occurrence $\mathcal{O}(v_j)$ de v_j dans $c_1 \dots c_n$.*

1.2.2 Représentation en PASCAL

Un arbre de Huffman est représenté par le type de base `ARBRE`.

Une feuille contient un caractère ainsi que le nombre d'occurrences de ce caractère dans la suite $c_1 \dots c_n$. L'arbre vide utilisé comme une feuille sans caractère permet de construire des arbres non complets. Il ne sera utilisé que lors de la reconstruction de l'arbre dans la phase de décodage.

Une liste d'arbres est représentée par le type de base `LISTE`.

Nous supposons prédéfinies les constantes et les fonctions suivantes dont le calcul se termine quelles que soient les valeurs de leurs paramètres. Elles pourront éventuellement être utilisées dans les réponses aux questions :

- `NIL` représente l'arbre vide et la liste vide d'arbres,
- `FUNCTION Noeud(g:ARBRE;d:ARBRE):ARBRE;` est une fonction qui renvoie un nœud dont le fils gauche et le fils droit de la racine sont respectivement `g` et `d`,
- `FUNCTION Feuille(v:CHARACTER;n:INTEGER):ARBRE;` est une fonction qui renvoie une feuille étiquetée par le caractère `v` et le nombre d'occurrences de ce caractère `n`,
- `FUNCTION EstFeuille(a:ARBRE):BOOLEAN;` est une fonction qui renvoie la valeur vrai si `a` est une feuille et la valeur faux sinon,
- `FUNCTION EstNoeud(a:ARBRE):BOOLEAN;` est une fonction qui renvoie la valeur vrai si `a` est un nœud et la valeur faux sinon,
- `FUNCTION Valeur(a:ARBRE):CHARACTER;` est une fonction qui renvoie le caractère qui étiquette la feuille `a`,
- `FUNCTION Nombre(a:ARBRE):INTEGER;` est une fonction qui renvoie le nombre d'occurrences du caractère qui étiquette la feuille `a`,
- `FUNCTION Gauche(a:ARBRE):ARBRE;` est une fonction qui renvoie le fils gauche du nœud `a`,
- `FUNCTION Droit(a:ARBRE):ARBRE;` est une fonction qui renvoie le fils droit du nœud `a`;
- `FUNCTION L_Ajouter(a:ARBRE;l:LISTE):LISTE;` renvoie une liste d'arbres composée d'un premier arbre `a` et du reste de la liste contenu dans `l`,
- `FUNCTION L_Premier(l:LISTE):ARBRE;` renvoie le premier arbre de la liste `l`,
- `FUNCTION L_Reste(l:LISTE):LISTE;` renvoie le reste de la liste `l` privée de son premier arbre,

- `FUNCTION L_Juxtaposer(l1,l2:LISTE):LISTE;` renvoie la liste composée de la liste `l1` suivie de la liste `l2`.

Exemple III.2 *Les deux premiers exemples précédents seront codés pour la suite daacadbcd par :*

```
Noeud(
   Noeud(Feuille('a',3),
         Noeud(Feuille('b',1),Feuille('c',2))),
   Feuille('d',3))

Noeud(
   Noeud(Feuille('a',3),Feuille('b',1)),
   Noeud(Feuille('c',2),Feuille('d',3)))
```

Le dernier exemple ne peut pas être codé car les nœuds ne peuvent pas contenir de caractères (clé de codage séparable).

1.2.3 Calcul du nombre d'occurrences

Les feuilles des arbres sont étiquetées par un caractère et par un entier qui indique le nombre d'occurrences du caractère dans la suite étudiée. Le nombre global des occurrences des différents caractères qui figurent dans un arbre correspond alors à la somme des nombres d'occurrences de toutes les feuilles de cet arbre. La fonction $\mathcal{O}(a)$ est ainsi étendue à l'arbre a.

Question III.6 *Écrire en PASCAL une fonction `nombre(a:ARBRE):INTEGER` telle que l'appel `nombre(a)` renvoie le nombre d'occurrences $\mathcal{O}$(`a`) des caractères figurant dans l'arbre `a`. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

1.3 Codage de Huffman

1.3.1 Principe général

L'algorithme de Huffman exploite la connaissance du contenu de la suite à coder pour générer une clé de codage minimale. Son principe repose sur la construction d'un arbre de Huffman minimal par transformations successives d'un arbre de Huffman quelconque.

L'algorithme découle de l'idée naturelle suivante : « *Pour minimiser le nombre de bits utilisés par le codage d'une suite de caractères, il faut associer le plus petit nombre de bits aux caractères les plus fréquents et le plus grand nombre de bits aux caractères les moins fréquents* ».

La longueur d'un code correspond à la profondeur du caractère associé dans l'arbre. Pour optimiser les codes, il faut donc que les caractères les plus rares étiquettent les feuilles les plus profondes de l'arbre et que les caractères les plus nombreux étiquettent les feuilles les moins profondes.

Réduire le coût d'un codage consiste donc à échanger une des feuilles les plus profondes de l'arbre avec un sous-arbre dont la profondeur et le nombre d'occurrences sont strictement plus petits.

Pour éviter les nombreux parcours d'arbres intermédiaires non minimaux pour rechercher les candidats à un échange, l'algorithme de Huffman construit directement l'arbre de Huffman minimal. Pour cela, il manipule un ensemble de sous-arbres de Huffman minimaux partiels. Cet ensemble contient initialement l'ensemble des feuilles étiquetées par les caractères de $\{v_1, \dots, v_p\}$. Chaque étape de l'algorithme remplace les deux sous-arbres g et d contenant les plus petits nombres d'occurrences par un

arbre de Huffman dont les fils sont g et d. L'algorithme s'arrête lorsque l'ensemble ne contient plus qu'un seul arbre qui est alors l'arbre de Huffman minimal de l'ensemble des caractères de la suite.

Question III.7 *Appliquer l'algorithme précédent pour construire l'arbre de Huffman correspondant à l'exemple dadbcd en détaillant chaque ensemble intermédiaire.*

1.3.2 Preuve de minimalité

Considérons H un arbre de Huffman de $\{v_1, \cdots, v_p\}$. L'arbre H' est obtenu en permutant les feuilles v_i et v_j dans H.

Question III.8 *Calculer la différence, notée Δ, entre le coût de H et le coût de H' en fonction des occurrences et profondeurs de v_i et v_j.*

Question III.9 *Montrer que dans un arbre de Huffman minimal, les feuilles de profondeur maximale contiennent des caractères d'occurrence minimale.*

Considérons H un arbre de Huffman minimal de $\{v_1, \ldots, v_p\}$. v_i et v_j sont deux caractères d'occurrence minimale possédant un même nœud père n dans H. L'arbre H' est obtenu en enlevant les feuilles v_i et v_j dans H et en considérant que n est une feuille d'occurrence $\mathcal{O}(n) = \mathcal{O}(v_i) + \mathcal{O}(v_j)$.

Question III.10 *Montrer que H' est un arbre de Huffman minimal (Indication : supposons que H est minimal et que H' n'est pas minimal).*

Question III.11 *Montrer que l'arbre de Huffman construit par l'algorithme de Huffman est minimal (Indication : H correspond à un problème de taille n, H' correspond à un problème de taille $n-1$).*

2 Préliminaires : Tri par insertion

L'algorithme de Huffman extrait de l'ensemble les sous-arbres contenant les plus petits nombres d'occurrences. Pour cela, nous exploiterons une liste de sous-arbres triée selon le nombre d'occurrences. Cette liste sera triée par un algorithme de tri par insertion.

Question III.12 *Écrire en PASCAL une fonction* `comparer(a1:ARBRE;a2:ARBRE):BOOLEAN;` *telle que l'appel* `comparer(a1,a2)` *renvoie la valeur faux si* `a2` *contient moins d'occurrences que* `a1` *et la valeur vrai sinon.*

Question III.13 *Écrire en PASCAL une fonction* `inserer(a:ARBRE;l:LISTE):LISTE;` *telle que l'appel* `inserer(a,l)` *renvoie la liste d'arbres obtenue en insérant l'arbre* `a` *dans la liste* `l` *en respectant l'ordre croissant du nombre d'occurrences. Nous supposons que la liste* `l` *respecte l'ordre croissant du nombre d'occurrences. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.14 *Écrire en PASCAL une fonction* `trier(LISTE):LISTE;` *telle que l'appel* `trier(l)` *renvoie une liste d'arbres triée selon l'ordre croissant des nombres d'occurrences et composée des mêmes arbres que* `l`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

3 Construction des codes de Huffman

Pour calculer le code minimal pour chaque caractère de la suite, il faut déterminer l'ensemble des caractères distincts et le nombre d'occurrences de chaque caractère, puis construire l'arbre par fusion successive des éléments de cet ensemble.

3.1 Construction de la liste d'occurrences

Question III.15 *Écrire en PASCAL une fonction* `ajouter(v:CHARACTER;l:LISTE):LISTE;` *telle que l'appel* `ajouter(v,l)` *renvoie une liste de feuilles étiquetées par les mêmes caractères que* `l` *et telle que si* `l` *contenait* `v` *alors son occurrence est augmentée de 1 et si* `l` *ne contenait pas* `v`, *le résultat contient* `v` *avec l'occurrence 1. Nous supposons que* `v` *figure au plus une fois dans* `l`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.16 *Écrire en PASCAL une fonction* `compter(s:SUITE):LISTE;` *telle que l'appel* `compter(s)` *renvoie une liste de feuilles étiquetées par les mêmes caractères que* `s` *et par les nombres d'occurrences de ces caractères dans* `s`. *Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

3.2 Construction de l'arbre

L'algorithme de Huffman de construction de l'arbre minimal présenté dans la sous-section 1.3.1 consiste à remplacer les deux sous-arbres ayant les plus petits nombres d'occurrences par le nœud ayant pour fils ces deux sous-arbres tout en préservant l'ordre de la liste triée.

Question III.17 *Écrire en PASCAL une fonction* `fusionner(l:LISTE):ARBRE;` *telle que l'appel* `fusionner(l)` *sur une liste non vide de feuilles* `l` *triée selon les nombres d'occurrences des caractères renvoie un arbre de Huffman contenant les mêmes caractères que la liste* `l` *et correspondant au codage minimal de la suite. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Question III.18 *Expliquer la fonction* `fusionner` *proposée pour la question précédente.*

Question III.19 *Écrire en PASCAL une fonction* `huffman(s:SUITE):ARBRE;` *telle que l'appel* `huffman(s)` *renvoie un arbre de Huffman contenant les mêmes caractères que la suite* `s` *et correspondant au codage minimal de la suite.*

4 Codage d'une suite

L'étape précédente a permis de construire un arbre dont les chemins qui conduisent à chaque caractère sont étiquetés implicitement (0 pour le fils gauche, 1 pour le fils droit) par le code associé à ce caractère. Pour éviter de multiples parcours de l'arbre lors du codage d'une suite de caractères, il faut construire une clé de codage à partir de l'arbre qui associe à chaque caractère son code puis parcourir la suite pour remplacer chaque caractère par son code en utilisant la fonction `coder` proposée dans la question III.1.

4.1 Construction de la clé

Question III.20 *Écrire en PASCAL une fonction* `construire(a:ARBRE):CLE;` *telle que l'appel* `construire(a)` *renvoie une clé de codage associant à chaque caractère contenu dans l'arbre de Huffman* `a` *son code, c'est-à-dire la suite de valeurs binaires (0 ou 1) correspondant au chemin suivi dans l'arbre pour atteindre le caractère. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

5 Décodage d'une suite

Le principal défaut du codage de Huffman est la nécessité de transmettre la clé de codage avec chaque suite codée. En effet, il faut reconstruire l'arbre de Huffman propre à chaque suite codée pour décoder celle-ci par un parcours de l'arbre en suivant chaque code de la suite codée.

5.1 Reconstruction de la clé

Question III.21 *Écrire en PASCAL une fonction* `reconstruire(c:CLE):ARBRE;` *telle que l'appel* `reconstruire(c)` *renvoie un arbre de Huffman obtenu à partir de la clé de codage* `c`*, c'est-à-dire contenant les mêmes caractères que* `c` *placées dans l'arbre selon le code associé au caractère dans la clé* `c`*. L'occurrence du caractère* `c` *prendra la valeur* `0`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

5.2 Décodage de la suite

Question III.22 *Écrire en PASCAL une fonction* `decoder(s:SUITE;a:ARBRE):SUITE;` *telle que l'appel* `decoder(s,a)` *renvoie une suite de caractères correspondant au décodage de la suite de valeurs binaires (0 ou 1) contenues dans* `s`*. Ce décodage est effectué par un parcours de l'arbre de Huffman* `a` *en suivant les chemins correspondant aux valeurs binaires contenues dans* `s`*. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

6 Optimisation de la construction de l'arbre

Question III.23 *Quelle est la fonction la plus souvent appelée lors de la construction de l'arbre ?*

Question III.24 *Proposer une ou plusieurs modifications des structures de données et/ou des fonctions pour réduire le nombre d'appels à cette fonction.*

La construction de l'arbre exploite une liste triée d'arbres. L'algorithme étudié dans la section 3.1 construit la liste dans un ordre quelconque puis effectue un tri de la liste obtenue. Il est possible d'éviter ce tri par une construction de la liste qui assure que les feuilles soient triées selon l'ordre croissant des nombres d'occurrences.

Question III.25 *Écrire en PASCAL une fonction* `ajouter(v:CHARACTER;l:LISTE):LISTE;` *telle que l'appel* `ajouter(v,l)` *sur une liste* `l` *de feuilles triée selon l'ordre croissant des nombres d'occurrences renvoie une liste de feuilles triée selon l'ordre croissant des nombres d'occurrences*

étiquetées par les mêmes caractères que `l` *et telle que si* `l` *contenait* `v` *alors son occurrence est augmentée de 1 et si* `l` *ne contenait pas* `v`*, le résultat contient* `v` *avec l'occurrence 1. Cette fonction devra être récursive ou faire appel à des fonctions auxiliaires récursives.*

Fin de l'énoncé